Resonance-Oriented Software Design and Development

FLEISSNER, Sebastian

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Doctor of Philosophy
in
Computer Science And Engineering

The Chinese University of Hong Kong
July 2009

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Abstract of thesis entitled:
Resonance-Oriented Software Design and Development
Submitted by FLEISSNER, Sebastian
for the degree of Doctor of Philosophy
at The Chinese University of Hong Kong in July 2009.

Software evolution draws its complexity from a variety of factors, including extensibility, maintainability, and the difficulty of changing a program's design. It is widely accepted that object-oriented programs become brittle as they evolve, because their design has to be fixed in the early stages of development, and the more their implementation has progressed, the more difficult it becomes to adjust interfaces of and relationships between objects.

This thesis introduces the notion of resonance-oriented software design and development, a family of software development approaches directly or indirectly inspired by concepts found in Asian philosophy, such as harmony, resonance, and fields of interactions, which significantly differ from the principles object-oriented programming is based on.

In particular, this thesis proposes two concrete resonance-oriented approaches called harmony-oriented programming, a paradigm that relaxes strong encapsulation and information hiding, and epi-aspects, a self-sustaining software architecture based on the notion of conscientious software. Apart from introducing the principles, constructs and conceptual architecture of the two proposed resonance-oriented approaches, this research describes concrete implementations, in particular runtime and development environments. These implementations are used to conduct studies aimed at supporting the hypothesis that, in comparison to traditional object-oriented programming, resonance-oriented software development is a more suitable approach for dealing with software evolution effectively.

摘要

很多因素構成了軟件演化的複雜性，包括可擴展性、可維持性，以及改變程序設計的困難度。現在人們普遍都認為，“面向對象程序”（或稱為“物件導向程式”）越發演變就越不穩定，因為它們的設計必須在程序開發早期就固定下來，因此隨著程序的繼續發展，就越來越難調整對象間的接口和關系。因此有必要發展一套較穩定的程序設計。

此論文提出了“反響導向程序設計”的概念，並介紹了一套相關的軟件開發方法。反響導向程序設計的原理大大不同於面向對象程序設計的原理，它的靈感和基礎主要是來自於東方哲學的思想，例如和諧、共鳴、互動，以及關系等文化形態。

此論文特別提出了兩種具體的軟件開發方法，一個是“和諧導向程序設計”，是一種可以調低封裝性和資訊隱藏的範式；另一個是“自生系統層面”，是一種建立在“自覺軟件”觀念基礎上的、可以自我維持的體系結構。

除了介紹反響導向程序設計的原理、兩個相關的軟件開發方法的實際建構和概念性體系結構外，本文也描述了具體的程序設計，特別是運行時間和集成開發環境。作者同時利用這些程序進行了一些試驗來支持本研究的主要假設：要有效地處理軟件演化中出現的問題，與傳統的面向對象程序設計相比較，反響導向程序是一個更適當的程序。

Acknowledgement

I would like to express my deep gratitude to my adviser Elisa Baniassad for sponsoring and supporting my research during the last three years. I also would like to thank all the researchers and reviewers who provided valuable comments and ideas for my work. Especially Ron Goldman, Doug Lea, David Ungar, Richard P. Gabriel, Michael R. Lyu, Jimmy H. M. Lee, and Yvonne Coady.

In addition, I would like to thank the people I met at OOPSLA'08 for encouraging me to discard an Objective-C based implementation of a harmony-oriented runtime and development environment and to switch to Smalltalk instead. Switching to Smalltalk made programming fun again and saved me a lot of time.

My appreciation also goes to Sui-Chu Wu for helping me to translate the abstract of this thesis into Chinese and my former and current lab-mates: Jacky Chan, K.K. Lo, Brian Chiu, and Clayton Myers.

Finally, I would like to thank Blizzard Entertainment for not releasing Starcraft 2 during my Ph.D. studies, and thus allowing me to graduate on schedule.

This work is dedicated to my parents
Dr. Jörg Fleissner and Dr. Gabriele Fleissner-Busse.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Gravity Attribute

It is the year 350 BCE[1]. Aristotelian physics [2], a theory developed by the philosopher Aristotle, suggests that gravity can be considered to be an attribute that resides within objects. Heavy objects, such as stones and rocks, possess a stronger gravity attribute than lighter objects. In fact, very light objects like gas and air have a "levity" attribute instead of a gravity attribute. Since gravity is an attribute of objects and not an outside force, Aristotelian physics posit the view that the world can be understood and described as a collection of more or less independent objects categorized by their attributes.

Forward to the year 2009. The slides of a tutorial on object-oriented design from an unknown author [92] illustrate an example of modeling fluids and pipes as classes. The example introduces a *Pipe* class that has a *gravity* attribute: a static double precision floating point number that is initialized with a value of 9.8. The reasons of the author for assigning a gravity attribute to pipe objects are not clear. But even though this decision might seem peculiar in the context of modern

[1] BCE: Before (the) Common Era.

science, it is acceptable in the context of object-oriented software design and development, which encourages isolation of objects from their environment.

1.2 Greek Philosophy and Object-Orientation

Object-oriented programming (OOP) is strongly influenced by ideas of ancient Greek philosophy and thought. As described in [73, 17, 43], ancient Greeks had a strong sense of personal agency and considered themselves to be individuals with unique, distinctive attributes and goals. Greek philosophers, such as Plato and Aristotle, posited the view that the world is a static and unchanging collection of objects that can be described and analyzed through categorization and formal logic. It was the habit of Greek philosophers to regard objects, such as persons, places, and things, in isolation from their context and to analyze their attributes. The attributes were used as the basis of categorization of an object, and the resulting categories are employed to construct rules governing the behavior of the object. The relevance of possible outside forces that can affect an object was completely ignored.

In object-oriented design and programming, it is common practice to isolate objects from their context and then describe them by their attributes (i.e. methods and instance variables) and static relationships to other objects. Isolation is achieved by applying the principles of information hiding and encapsulation. Program units, such as objects, components, and modules, interact through well-defined interfaces that expose functionality. As interfaces hide the implementation details of each program unit, internal changes do not affect other parts of the program. Because the focus is on objects, modeling relationships is not straightforward in object-oriented programming and often requires

significant negotiation for establishing and breaking off relationships between two or more objects.

1.3 Brittleness Through Software Evolution

The term software evolution refers to changing both design and code of software repeatedly over time in order to comply with changing requirements or initially unexpected usage scenarios. As software evolves, it becomes more brittle: the difficulty of maintaining and fixing the software increases, and partial or complete failure occurs, when small changes are made or unexpected data is encountered.

To cope with software evolution, the traditional object-oriented approach of making a complete software design before coding has been replaced with other strategies, such as design for extensibility and maintainability. However, designing for extensibility and maintainability is non-trivial in object-oriented programming, as interfaces and object relationships have to be fixed at some point. Any subsequent change to the interface of one object can lead to many potential changes to dependent objects. As pointed out in [16] and [72], software evolution eventually causes brittleness even in well-designed object-oriented programs.

1.4 Resonance-Oriented Software Development

The purpose of this thesis is the proposal and evaluation of *resonance-oriented software development*, a family of software development approaches based on a school of thought that promotes holism and the idea of mutual influence of everything on almost everything else. This school of thought, which originated in Asia, posits the view that the

world cannot be described by focusing on objects and their attributes alone, but rather by considering the broad context and see the world in terms of resonance, harmony, and context. For example, as described in [44] and [35], philosophers in ancient China described the world as a mass of continuously interacting substances rather than a collection of discrete objects, and each substance and every event in the world were considered to be related to each other.

Resonance-oriented software development approaches have the following characteristics:

- The runtime environment of resonance-oriented programs is well-defined.
- Code entities always run inside the well-defined environment and can directly or indirectly interact with it.
- The execution of code affects or changes the environment.
- The environment (or changes inside the environment) can affect the behavior of code.

This thesis proposes and evaluates two concrete resonance-oriented software development approaches. The first approach is a new programming paradigm called *harmony-oriented programming* [31, 32, 6]. The second approach an architecture based on the theoretical notion of conscientious software [36] called *epi-aspects* [30].

1.5 Hypothesis

Software evolution is significantly affected by the following factors:

- Ease of changing the program's design (changeability).
- Extensibility of the program.
- Maintainability of the program.
- Quality feedback.
- Error recovery.

Ease of changing the design refers to the complexity of changing a program's structure. This includes relationships between parts of the program, such as associations and inheritance relationships in object-oriented programming.

Extensibility refers to the ease of extending a program with or without changing the program's overall structure. Extensibility can be affected by the ease of changing the program's design.

Maintainability refers to the ease of changing a program in general. It is affected by both extensibility and ease of changing the program's design.

Quality feedback and error recovery refer to mechanisms that facilitate reporting, observation and correction of problems and errors.

The hypothesis of this research is that in comparison to object-oriented programming, resonance-oriented programming improves the ease of dealing with the above mentioned issues, and thus the ease of dealing with software evolution effectively. In particular, this hypothesis posits that the combination of the proposed resonance-oriented programming approaches provides the following advantages over traditional object-oriented programming:

1. Fewer changes are required in order to reflect adjustments of a program's design in the code. Changes include source code modifications and other adjustments to a program.
2. Extending a program requires less effort (steps/changes) .
3. Implementation of reliable feedback and error recovery mechanisms requires fewer steps.

1.6 Thesis Organization

The rest of this thesis is organized as follows:

Chapter 2: Research Background
This chapter provides an overview of research that inspired the work presented in this thesis. In particular, it introduces the work of Richard Nisbett on how "Easterners" and "Westerners" think differently, and the conscientious software paradigm proposed by Gabriel and Goldman.

Chapter 3: Preliminary Study
This chapter describes a preliminary study that explores how Nisbett's findings regarding different reasoning styles of individuals from different cultural backgrounds apply to the realm of software development.

Chapters 4 and 5: Proposed Approaches
These chapters introduce the proposed resonance-oriented software design and development approaches: Harmony-oriented Programming and Epi-Aspects. Apart from providing conceptual descriptions of the approaches, these chapters introduce concrete implementations and development environments.

Chapter 6: Case Studies and Validation

This chapter describes several case studies aimed at supporting the hypothesis formulated in section 1.5. In particular, several scenarios are used to compare the resonance-oriented approaches to object-oriented programming. The results of the various studies are summarized and analyzed to validate the hypothesis.

Chapter 7: Discussion

This chapter discusses various aspects of the proposed resonance-oriented software development approaches, such as practical issues and limitations.

Chapter 8: Related Research

This chapter covers various approaches related to resonance-oriented software development, such as diffusion-based agent systems, software evolution research, related programming approaches, self-sustaining systems, and error recovery. The more closely related work is compared with the proposed resonance-oriented software development approaches.

Chapter 9: Conclusions

This chapter summarizes the research presented in this thesis, describes its contributions, and discusses future work.

□ **End of chapter.**

Chapter 2

Research Background

This chapter provides an overview of research that inspired the work presented in this thesis. In particular, it introduces the work of Richard Nisbett on how "Easterners"[1] and "Westerners"[2] think differently, and the conscientious software paradigm proposed by Gabriel and Goldman.

2.1 The Geography of Thought

In his book *The Geography of Thought* [73] Nisbett makes the case that individuals from different cultural backgrounds reason about objects and spaces differently from one another. In particular, he asserts, and shows through peer reviewed studies, that subjects from Chinese and Japanese cultures do not relate in the same way to objects as those from the west. According to Nisbett, "Western" thinkers identify the world by objects and view the world as a set of individuals, working essentially independently, maintaining their own world view, and acting upon it. "Eastern" thinkers, on the other hand, view the context of objects as

[1]The term "Easterners" is used to refer to people from Chinese and Japanese cultures.
[2]The term "Westerners" is used to refer to individuals from Europe and America.

centrally as the objects themselves. They notice changes in a scenery before they notice changes in individuals within that scenery. When considering objects, they consider the fields of interactions between those objects, rather than seeing the objects as autonomous.

2.1.1 Western Obsession with Modularization

According to Nisbett, "Westerners" are keenly interested in *atomizing* the world. He notes that this contributed largely to economic advances and industrialization: allowing manufacture to happen in generic ways, which enhanced efficiency and interoperability of approaches. He goes on to say that this modular view carries through to social infrastructure. He asked "Easterners" and "Westerners" to consider phrases that described companies as either social networks where people work together, or as an institution with a goal, where people are hired to perform functions. Most "Westerners" related best to the atomized view described in the second statement. "Easterners" predominantly chose the first statement as more accurate [73], p 84.

2.1.2 Eastern and Western Reasoning

Nisbett and other researchers, such as Peng [77], Morris [70], and Gries [41], contrast "Eastern" versus "Western" reasoning by describing dialectic reasoning from the east, and identity and non-contradiction in the west.

Dialectic reasoning held in eastern traditions involves:

- The Principle of Change: this captures the constantly changing nature of reality.

- The Principle of Contradiction: due to constant change, paradoxes are constantly being introduced. Both A and $!A$ might be true at the same time.
- The Principle of Relationship; Holism: nothing exists in isolation. Everything must be described by its relationship with other things.

"Westerners", on the other hand, hold two logical principles dear:

- The Law of Identity: A is always A, regardless of context.
- The Law of Noncontradiction: A and $!A$ cannot both hold true.

2.1.3 Control Flow

Nisbett's research indicates that "Westerners" place great importance in feeling a sense of control, whereas "Easterners" are more likely to acknowledge that they are out of control, and make adjustments to fit into an uncontrollable situation ([73], p 97). Adjustments for "Westerners", on the other hand, were assessed to feel unnatural or "awkward". "Westerners" are also more interested in knowing who is in control, as is evidenced by "Westerner's" dislike of working in groups, where control may be ambiguous. "Easterners", on the other hand, would rather work in a group, regardless of the quality of that group, and simply adjust to the group dynamics without establishing explicit control.

2.1.4 Substances Versus Objects

Nisbett's book refers to an experiment by Imae and Gentner [48] called *The Dax Experiment*. In it, Imae and Gentner showed "Easterners"

and "Westerners" a shape made out of some substance, and told the subject to "look at this *dax*". Then, they showed the subjects two trays of objects, one carrying objects made from the same substance as the dax, and other carrying objects that were the same shape as the dax. They were then asked to identify the tray with the dax on it. "Westerners" predominantly chose the tray with objects of the same shape, whereas "Easterners" chose the tray with objects of the same substance. Nisbett points out that this study indicates that while "Westerners" see the world as a set of disconnected objects, modern "Easterners" view the world as continuous masses of matter.

2.2 Conscientious Software

Conscientious software is a theoretical paradigm and philosophy for developing reliable, self-sustaining software systems proposed by Gabriel and Goldman in [36]. Unlike other approaches for self-sustaining software, such as IBM's autonomic computing [54, 71], conscientious software consists of two distinct parts written in fundamentally different programming languages: an *allopoietic*[3] part that encapsulates application functionality, and an *autopoietic*[4] part that continuously re-creates itself and is entirely devoted to keeping the system running smoothly.

Conscientious software is based on the realization that, even though error recovery and monitoring are well-understood concepts, its techniques are not frequently applied in practice. The separation of software into autopoietic and allopoietic parts is meant to encourage devel-

[3] "Allopoiesis is the process whereby a system produces something other than the system itself. One example of this is an assembly line, where the final product (such as a car) is distinct from the machines doing the producing. This is in contrast with autopoiesis." (From [102]).

[4] "Autopoiesis literally means auto (self)-creation (from the Greek: auto - for self- and poiesis - for creation or production)." (From [101]).

opers to devote equal efforts towards implementing application functionality and error recovery.

2.2.1 Allopoietic Part

The allopoietic part encapsulates traditional application functionality. It is written in a general purpose programming language, such as C++ or Java, and produces some computational results or provides services to users.

2.2.2 Autopoietic Part

The autopoietic part monitors and adapts to environmental changes, and observes and evaluates the health of the allopoietic part. In case the allopoietic part fails, the autopoietic part assists with error recovery. Since general purpose programming languages are potentially fragile and hence pose a threat to the stability of the system, the autopoietic part is written in a dedicated autopoietic programming language designed to make it difficult for programmers to implement programs with critical bugs.

2.2.3 Epimodules

To maintain the health of the application, the autopoietic part must be able to observe and affect the operation of the allopoietic part. In [36], Gabriel and Goldman propose the concept of *epimodules*, which serve as a bridge between the autopoietic and allopoietic parts. Epimodules are attached to allopoietic components and monitor their behavior. When necessary, epimodules can affect and alter allopoietic components. For

example epimodules can instruct allopoietic components to run tests, restart, upgrade, clone, or kill themselves.

□ **End of chapter.**

Chapter 3

Preliminary Study

This study [6] is an exploration on how Nisbett's findings regarding different reasoning styles of individuals from different cultural backgrounds apply to the realm of software development. As explained in section 2.1, Nisbett asserts that individuals from "Western" societies tend to focus on objects isolated from their context and their attributes, while individuals from "Eastern" societies rather consider fields of interactions between objects and, apparently, look for harmony.

Most major programming languages, especially object-oriented languages, were developed in the west, by, what Nisbett would classify, as individual thinkers. The roots of object-orientation were to help programmers model the world as they saw it and to better align their programmatic representations with their mental models of a problem space.

Problems arise, however, when systems do not align well with a pure object-oriented modularization. This misalignment is often evident in systems that involve a great deal of object interaction and negotiation. Since the rise of object-oriented programming, attempts have been made to break apart the rigid adherence to the individual nature

of object-oriented languages. Aspect-orientation is an example of such an attempt: through aspects, developers can describe concerns that crosscut objects [57], or can capture, in one location, the relationships between them [76]. This multidimensional movement may fit into the concepts of postmodern programming, as described by Noble and Biddle [74]. It is also possible that these attempts are edging towards capturing a more "Eastern" philosophy of programming, where fields of interaction are as important as the objects themselves, and where writing a working program means attaining programmatic harmony.

This study is motivated by how closely the object-oriented paradigm resembles typical "Western" thought. It investigates the use of an "Eastern" reasoning approach for capturing object dynamics and interactions between objects. As object-orientation grew from the minds and reasoning style of "Westerners", the purpose of this study is to look deeper into the minds of "Easterners" in an attempt to capture their world view, and then distill their descriptions into the guidelines for a new, more harmony-oriented, programming paradigm.

The main part of this study is an experiment to capture the way in which "Easterners" would describe what would be considered a typical object-oriented scene: Nygaard's, now famous, Restaurant Picture [28] (shown in Figure 3.1). Nygaard introduced this picture as a mechanism for teaching students about object-orientation. He motivated the use of this image by saying *"to teach object-orientation, you need a sufficiently complex example"*. The idea, as he presented it, was that students would be able to look at this picture, and identify different kinds of people, their traits, and the activities in which they were engaged. This would help them think about objects (individuals).

During the experiment, "Eastern" subjects were interviewed about how

Figure 3.1: Nygaard's Restaurant Picture

they would describe the picture. The following sections summarize the analysis of the subjects' responses, and, based on that analysis, suggest possible avenues for pursuing *harmony-oriented programming*: a new paradigm that allows for straightforward modeling of how program entities and their behaviors affect one another.

3.1 Picture Description Interviews

The picture description interviews were conducted with two groups of subjects. The first group consisted of three participants from Europe and served as control group. The subjects of this "Western" group were university students of non-engineering majors: two females and one male between 22 and 26 years old. The second group consisted

of university students from Hong Kong and mainland China: 3 female students of non-engineering majors and 8 male computer science students.

During the interview sessions, the subjects were asked to describe the scene shown in Nygaard's picture. They were instructed to describe anything they saw, with no restriction, and they were allowed speak for as long as they felt comfortable (durations ranged from 2-8 minutes). Additionally, the subjects were allowed to annotate a copy of the Nygaard's picture if they wished (figure 3.2).

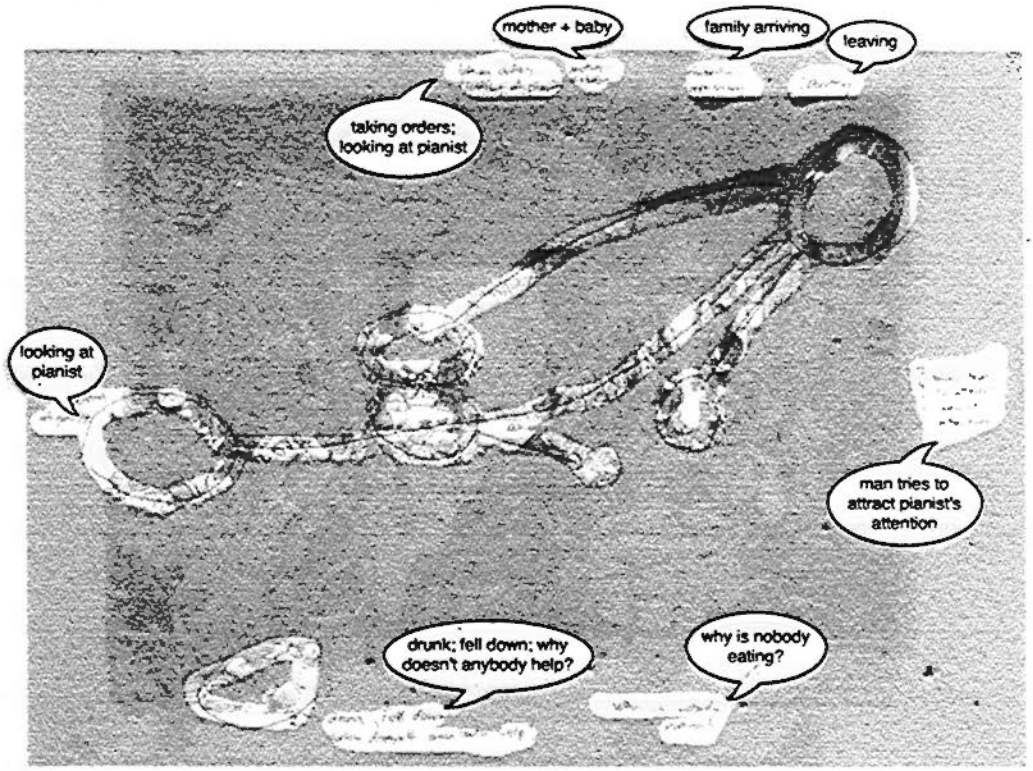

Figure 3.2: Annotated Restaurant Picture

The subjects were not aware that they were being interviewed because of their cultural background. Instead, its was indicated that they were taking part in a study on how pictures are described by different individuals.

After the interviews, the transcripts were analyzed in two passes. The first pass was a cursory read to derive general commonalities between the descriptions, and arrived at a set of statement-categories. The second pass included performing a detailed analysis to categorize each statement made by each subject.

As Nisbett's findings predicted, the European subjects focused on describing individual objects in isolation, while the Chinese subjects did not comment to any great extent on individuals or their traits. Instead, any descriptions of individuals were couched in descriptions of how an individual related to a group.

The following sections provide an overview of the descriptions by the Chinese subjects, which can be grouped in three main categories: context, relationships and puzzlement. The first two of these can be further decomposed into the following categories:

- Context: Environment
- Context: Interpretation
- Context: Observation
- Context: Role
- Relationship: Short Range
- Relationship: Long Range

3.1.1 Descriptions of Context

Context refers to a description of the situation in which objects are placed. As mentioned above, this can be sub-categorized into:

- Context: Environment
- Context: Interpretation
- Context: Observation
- Context: Role

Context: Environment

This category covers remarks that pertain to the environment of the restaurant. For instance, one subject talked about the poor lighting inside the restaurant. Other remarks placed objects (people in the restaurant) in context. Seven quotations that fit into this category were identified.

Context: Interpretation

This category covers remarks where the respondent is interpreting what is going on in the restaurant, and perhaps trying to determine the reality behind something they perceived to be unclear. For example, one subject provided an interpretation why a small child with his mother near the entrance of the restaurant is crying. 20 remarks were categorized as *Context: Interpretation.*

Context: Observation

This category includes remarks that are simply observations about the restaurant in general (not the environment or ambiance). A total of 21 remarks that fit into this category were identified.

Context: Role

This category includes the different roles that respondents identified. Respondents identified:

- Waiter (Respondents 1, 4, 6, 7, 8, 11)
- Pianist or Musician (Respondent 3, 8)
- Cook or Chef (Respondent 3, 6, 7)
- Boss (Respondent 4)
- Doctor (Respondent 4)
- Workers (Respondent 4)
- Guest (Respondent 8)

A total of 17 remarks related to roles was identified.

3.1.2 Relationships

Two categories related to relationships between people in the restaurant were identified: long and short distance. Short distance relationships are between people at the same table (or in some way involved with the table), whereas long distance relationships are between tables.

Long Distance

Long distance relationships refer to remarks that describe how two sets of people relate, when those people are not seated at, or in some way involved with, the same table.

Ten such quotations were identified, some of which were duplicates of others. They are depicted in figure 3.3 as white lines linking the groups of people included in the remark. The longest distance relationship spanned from the highlighted area A to the highlighted area B. A shows two people from table 20, and B highlights what is identified as a cake. The two respondents who noted this relationship stated that the people

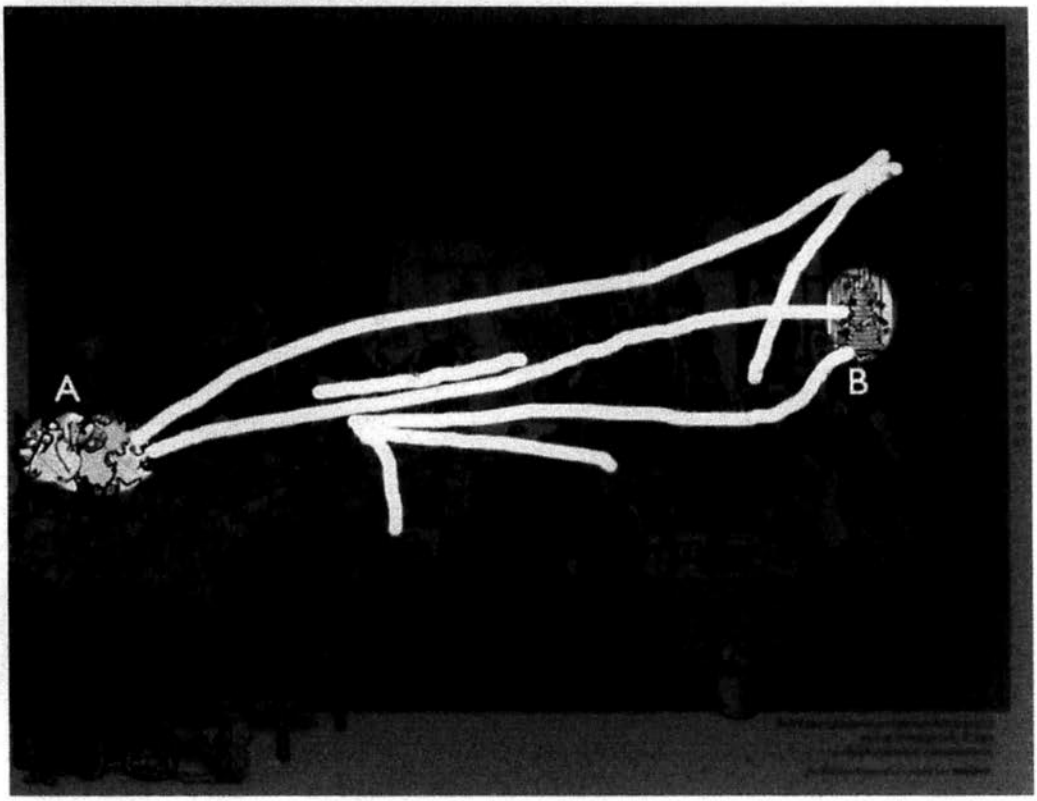

Figure 3.3: Identified Long Distance Relationships

in area A were watching and waiting for their cake, which was being brought to them from area B.

Short Distance

As described above, short distance relationships refer to remarks describing how people at a single table relate. The following short distance relationships were found:

- Mother and Son (table 1—Respondent 2, 4, 6, 7, 11) (table 7 Respondent 3)
- Friends (Respondent 3, 4, 7)
- Dating (Respondent 3, 4, 11)
- Family (Respondent 6)

3.1.3 Puzzlement

A total of 15 remarks that relayed confusion about the restaurant scene were identified. For instance, Respondent 9 remarked that the piano area of the restaurant is cut off from the rest of the restaurant, and another respondent commented that the man at the coat-check did not fit in well with the rest of the guests.

3.1.4 Common Observations

Several observations were made by multiple respondents. Table B.1 (in appendix B) provides the results of the analysis of the observations, and shows which category the responses fit under. Some observations fit into multiple categories. The total number of responses counts the individual responses from all categories; the same respondent may have uttered more than one response for a particular observation.

The four most popular observations were:

1. *Table 6: Man unable to pay*
   Three respondents commented on the man at table 6 (highlighted in area "A" in Figure 3.4) who is apparently unable to pay his bill, and the women at table 4 who are perceived to observe him (highlighted in area "B" in Figure 3.4).

2. *Empty seats, but people are waiting*
   Several of the participants in the study noted, with consternation, that there were people waiting outside the restaurant while there were empty seats inside (at table 3). They wondered why the waiter would not seat people at that table, and commented that the restaurant was "strange" because of it.

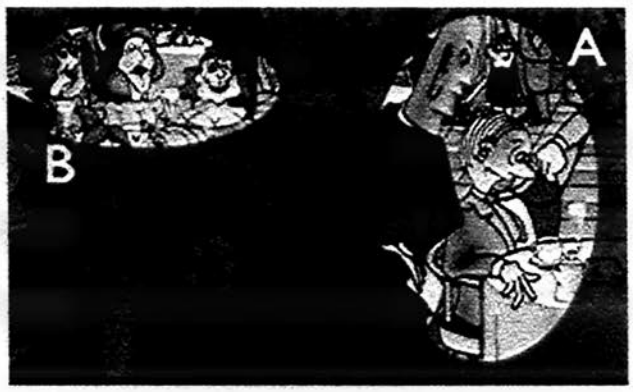

Figure 3.4: Man unable to pay

3. *The "bad" mother*

   Another great cause for concern among the participants was the mother sitting at table 1 with her child (shown in figure 3.5). Five of the participants noted something very similar to one another: that the mother is covering her ears attempting not to hear her child crying.

Figure 3.5: A mother perceived to ignore her crying child

4. *The misplaced chef*

   The fourth most frequently observed element of the restaurant was the chef, who is, apparently, out of place. Many of the respondents noted that the chef was not in the kitchen, that he was lazy, or that he was simply taking a break.

Figure 3.6: A chef perceived as out of place

3.2 Analysis

This section presents a qualitative analysis of the subjects' responses. All of the most common responses (listed in Table B.1) can be distilled into two major kinds of observations: identification of disharmony in the restaurant, and collective action.

3.2.1 Identification of Disharmony

The respondents were overwhelmingly troubled by the "strange" or crazy nature of the restaurant, where things were not as they should be, and where imbalance reigned supreme. This generally fits into several categories:

- People or things being physically out of place.
- People who did not fit in, visually, with others.
- People shirking their responsibilities.
- Things simply going wrong.

People or Things Physically Out of Place

There were many instances when respondents noted that people or things were not where they should be, but the two most common were the out of place cook and the non-sequential table numbers.

The respondents who commented on the cook all noted that he was behaving strangely, and that his place was supposed to be in the kitchen. They wondered why he was not inside the kitchen. This can be interpreted as an identification of disharmony, because the cook's resting state was to be working in the kitchen, and he was upsetting the balance of the restaurant by leaving that state and emerging into the dining area.

The respondents who commented on the table numbers seemed jarred by what they perceived to be a gap in the continuous space of the restaurant. This was also found in the statement about how a guest might climb up to the piano area (a frustrated respondent noted that there was no clear way to get up to there). The respondents suggested solutions to the disharmony identified: that there must be more tables that are not shown in the restaurant, and there must be hidden stairs somewhere.

Another example of a discontinuity was the perceived lack of teenagers in the restaurant. No solution was given for this discontinuity.

People Not Fitting In

Many respondents referred to people who did not look quite right, and described this visual disharmony as the fault of the persons who are improperly dressed. Some respondents tried to find reasons that the people were crazily dressed, not willing to simply accept that it might just be a scene that was beyond reasonable explanation.

For instance, the man by the coat check, who is comedically dressed in north pole regalia, and who is holding a fish on a skewer, was the source of much concern for several respondents. Explanations for why he looked the way he did was that he had found his way into the wrong location, thinking he was going to a barbecue, and that he was having take out. The respondents noted that he might be about to take off his coat (hence adjusting himself to blend in with the other diners).

The other popular observation about visual disharmony was the strange table of party goers. This group wore costumes, and it was remarked upon by three respondents that it is not clear "what kind of people" sat at this table, because not only did their clothes not fit in with the restaurant, they were not even a cohesive group.

People Not Fulfilling Their Roles

Respondents were quick to point out that some people in the restaurant were stubbornly disturbing the peace. The mother with her crying child refused to perform her duty and calm her child. The waiter was being careless, and was about to drop all the plates, which would further disrupt the harmony of the restaurant. It was perceived that none of the guests in the restaurant were doing their jobs all that well, because no one was eating or having a good time. Some respondents noted that guests seemed upset or distracted in some way. These observations echo Nisbett's findings that "Easterners" see a strong relationship between roles and harmony.

Other Imbalances

Respondents noticed other situations in which harmony was not maintained. The most common were the imbalance between the empty tables and the people waiting for tables, and the lack of light in the restaurant.

3.2.2 Collective Action

Other responses could be categorized as "actions". However, unless they were counted as disharmonious (the mother with her child, the strange man with the fish, the bad waiter), they were described as *collective action*: a group of people doing something together, and feeding off each other in doing so. Once again, the people at the party-going table were perceived to be actively waiting for their cake to arrive (both anticipating, and looking at the arriving cake). Another popular example of collective action was the three women sitting together and laughing at the man who could not pay his bill. Finally, respondents identified a group at a table who were all trying to gain the attention of the pianist by waving to him.

There was no sense that the actors in the groups were individuals working together - instead, the actors were considered by the respondents as one larger actor: the group. The individuals were at times described as behaving specifically within the group, but the action belonged completely to the group, not to one of the people in it.

3.3 Harmony-Orientation In Software

The analysis shows that commonalities between subjects' responses are further categorizable as descriptions of *harmonious action*, and of *harmonious situations and flow*.

Both of these concepts play important roles in current software systems, even if they are not easily attainable. For example, in a load balancing system, a harmonious situation, or equilibrium can be achieved when various instances of a component share the same workload. This

equilibrium is lost if some instances are busy while other instances are idle.

Similarly, harmonious situations are the goal for system monitoring and management software, such as intrusion detection, virus detection, and network monitoring systems. The purpose of such systems is to preserve the harmony of a system of components. Such components are in harmony, if they operate at full capacity and are able to interact with each other without any disturbance. The balance of this harmony can be disrupted by system anomalies, which might be caused by viruses, intruders, or component failures.

Harmonious action and situation also play a significant role for the components of a single software application or server. Within an application, a harmonious situation is achieved if all components are working well. However, each time a component of the system is updated or otherwise changed, the balance of the system is disturbed and has to be restored. This is especially the case when component interfaces are adjusted or components with critical bugs are added to the application.

On a finer grain, harmonious action between objects in a system might be found when they are communicating well, and using interfaces provided by one another correctly. In traditional systems, attaining this harmony involves negotiation between objects. Ensuring up front that such harmony will be achieved involves checks by the compiler.

Harmony-oriented programming (chapter 4) is based on the findings of this preliminary study.

□ **End of chapter.**

Chapter 4

Approach I: Harmony-Orientation

Ancient Chinese philosophers did not focus on objects and their attributes, but rather considered the broad context and saw the world in terms of harmony, context, roles, obligations, and resonance. For example, a person was considered not as an individual with a constant unique identity, but rather as a member of several collectives. As described in [44], ancient Chinese philosophers and people saw the world as a mass of continuously interacting substances rather than a collection of discrete objects. Each substance and every event in the world was considered to be related to every other event.

This chapter proposes a resonance-oriented software development approach called harmony-oriented programming [6, 31, 32], a new programming paradigm inspired by concepts of "Eastern" thinking and reasoning, such as harmony, context, and resonance, and the preliminary work presented in chapter 3.

The main idea behind harmony-oriented programming is that pieces of a program always interact with their environment as a whole and usually not with other program parts directly. Table 4.1 illustrates

important conceptual differences between harmony-oriented programming and object-oriented programming (OOP).

Object-Orientation	Harmony-Orientation
Individualism	Holism
Explicit Boundaries	Fuzzy Boundaries
Explicit Relationships	Implicit Relationships
Protocols / Negotation	Observation

Table 4.1: Object-Orientation and Harmony-Orientation

Harmony-oriented programming challenges established and widely accepted object-oriented design principles [9], such as strong encapsulation, information hiding, and inheritance, and favors more flexible and ah-hoc approaches for structuring and implementing programs. Apart from presenting a discussion of harmony-oriented principles, this chapter introduces constructs of harmony-oriented programs and a Smalltalk-based runtime and development environment.

4.1 Principles of Harmony Orientation

Harmony, resonance, and context are three key concepts found in Asian (in particular Chinese) philosophy. These three concepts are the basis of the principles of harmony-oriented programming, which are denoted as *balance*, *exposure*, *spaciality*, *information sharing*, and *information diffusion*. Figure 4.1 illustrates the relationship of the three key concepts with the principles of harmony-oriented programming.

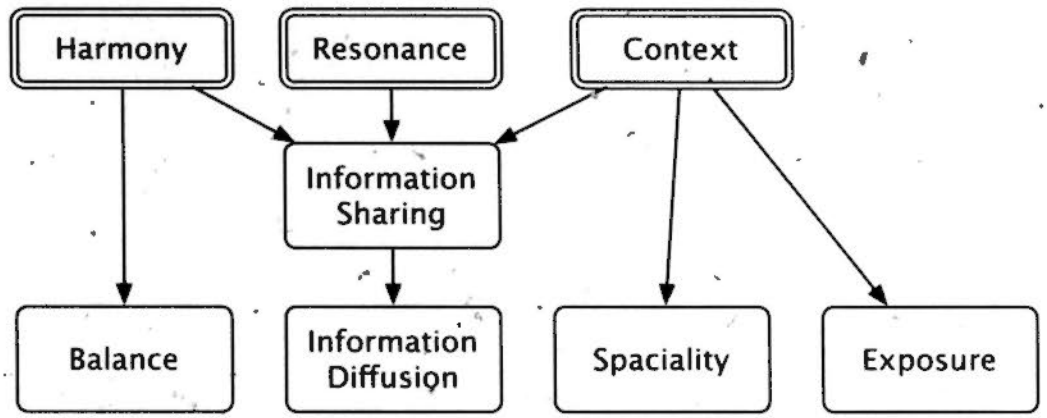

Figure 4.1: Principles of Harmony-Oriented Programming

4.1.1 Balance

The balance principle is inspired by the concept of harmony and refers to balance of data production and consumption. The overall goal of a harmony-oriented program is a balanced state, which is achieved when any data produced by one part of the program is consumed by one or more other parts of the program. For example, if a part of the program produces data that is not consumed, or a part wishes to consume data that is not available, the program is in an imbalanced state.

4.1.2 Exposure

The design of object-oriented programming and other programming languages is based on the principle of encapsulation. Unlike encapsulation, the exposure principle (figure 4.2) suggests decomposing a program into pieces called *snippets*, without the need to encapsulate these pieces using constructs with well defined boundaries, such as modules, functions, and objects. Hence, snippets do not conform to or expose any specific interface. However, the code inside snippets can contain constructs based on the encapsulation principle. In the simplest case, a snippet is a single statement.

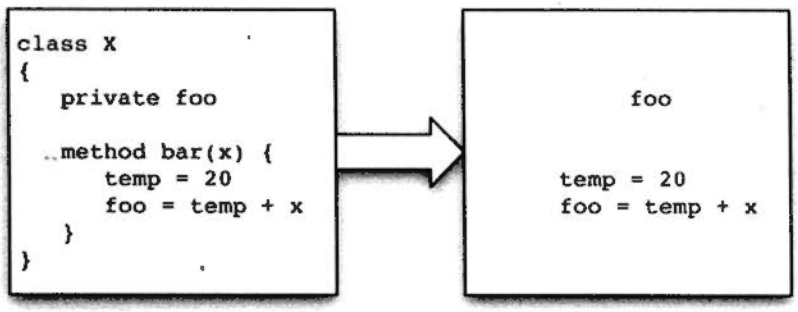

Figure 4.2: Exposure Principle

4.1.3 Spaciality

The spaciality principle (figure 4.3) suggests that every part of a program is assigned to one or more locations in a virtual space.

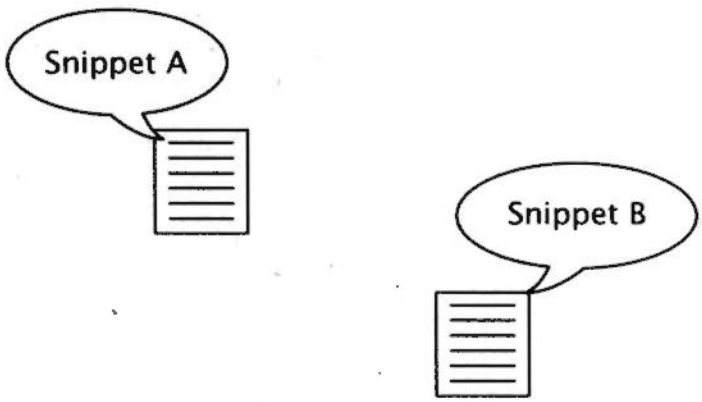

Figure 4.3: Spaciality Principle

Related parts of a program are positioned close to one another to form a specific context. For example, snippets that implement a user interface are placed in one another's vicinity to form a user interface

context, and snippets that implement a certain part of business logic are placed somewhere else to form another context. Certain snippets can be assigned to both example contexts to serve as a bridge between user interface and business logic.

Spaciality can be considered as an alternative to hierarchies of program entities like the object hierarchies found in object-oriented programs.

4.1.4 Information Sharing and Diffusion

The information sharing principle suggests that all data is shared between the pieces of a program. This principle facilitates resonance between program parts, as one part of the program can react to changes made by any other part of the program.

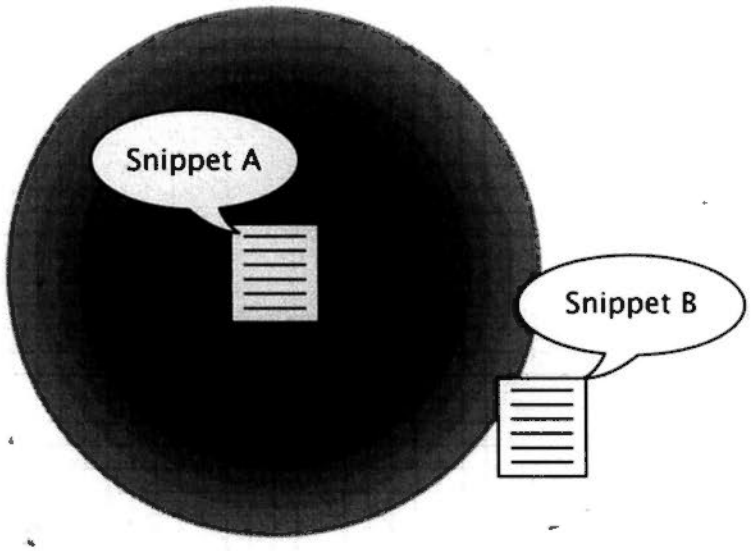

Figure 4.4: Information Diffusion Principle

Diffusion is a gradual process in which a substance is spread over a space over time. The information diffusion principle (figure 4.4) states

that data or a description of the data generated by any part of the program is diffused throughout the virtual program space. Data has an associated intensity that decreases the further it is diffused. The combination of the diffusion and spaciality principles ensures that data generated by one code snippet (or the description of that data) reaches other code snippets that are located close within the virtual space first.

4.2 Harmony-Oriented Programs

Figure 4.5 illustrates the anatomy of harmony-oriented programs.

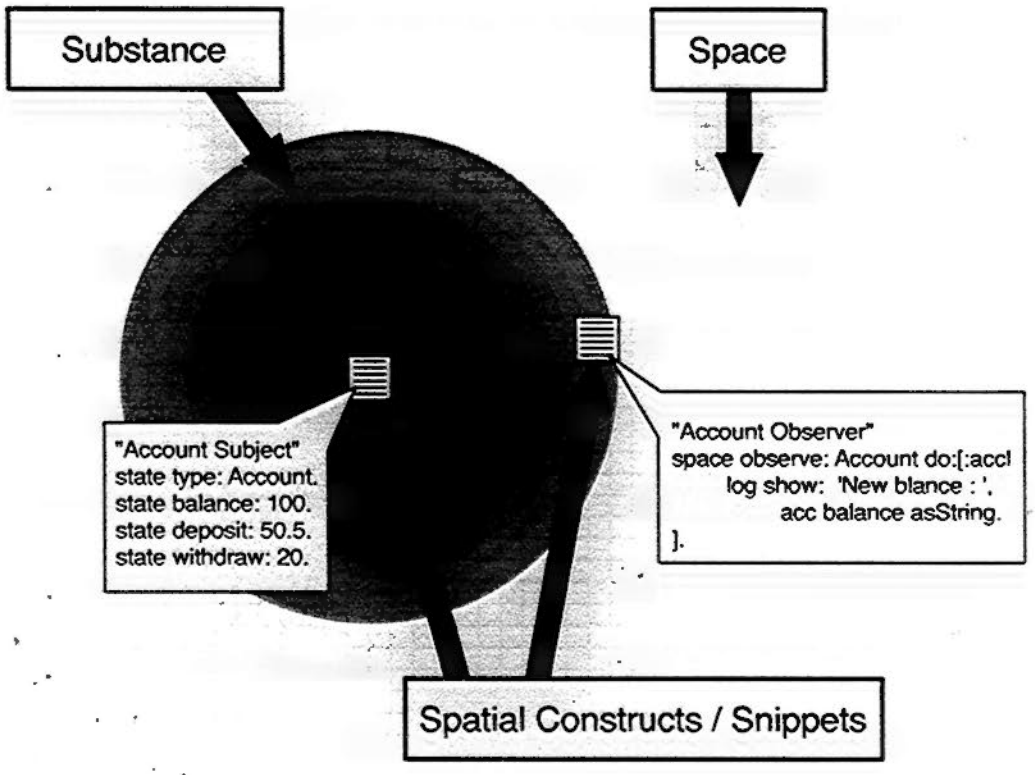

Figure 4.5: Anatomy of Harmony-Oriented Programs

A harmony-oriented program consists of virtual *spaces* with two or more dimensions that contain *spatial constructs*. Spaces serve as the runtime environment of the harmony-oriented program. Each spatial construct is assigned to a specific location in a space and can interact

with the space by putting data into and consuming data from its location. Spatial constructs are only aware of the space containing them and cannot see or interact with other spatial constructs. Whenever a space receives data from a spatial construct, it automatically diffuses it by generating a virtual *substance*. Because of the diffusion, the data eventually reaches the locations of other spatial constructs, which then can consume the data. Hence, the diffusion process facilitates indirect data exchange between the spatial constructs inside a space.

In addition to spatial constructs, concrete harmony-oriented programming languages and runtime environments can choose to support object-oriented constructs like classes and objects for the purpose of realizing abstract data types and accessing existing application programming interfaces. As a result, harmony-oriented programming can be realized as an extension to object-oriented programming. However, when writing harmony-oriented programs, the primary decomposition is always in terms of spaces and spatial constructs, and not objects.

The following sections provide a detailed description of spatial constructs, spaces, and diffusion.

4.2.1 Spatial Constructs

Spatial constructs are program constructs that are assigned to a location in a space. As mentioned above, spatial constructs can only interact with the space containing them and not with other spatial constructs directly. In particular, spatial constructs can:

- Put data into its location in the space.
- Consume data from its location in the space.

- Observe data inside its location in the space.

Spatial constructs cannot move themselves: They are placed into the space by software developers who can move them around while the harmony-oriented program is running.

Snippets

The most important spatial construct is the *snippet*. As the principle of exposure suggests, a snippet is a piece of source code that is not encapsulated using a construct with well-defined boundaries.

In the simplest case, a snippet is a single statement or a list of statements. Like objects, snippets can maintain a state. However, objects use encapsulation and information hiding to isolate their state from other parts of the program. The state of a snippet, on the other hand, is owned by the space containing the snippet, and, like any other data placed into the space, is diffused and thus available to other spatial constructs.

Diffusion Barriers

Diffusion barriers are spatial constructs that block or weaken diffusion. Programmers can use such spatial constructs to exercise fine grained control on diffusion within a space.

Hole Constructs

Holes are spatial constructs that consume data from their location and place it somewhere outside the space containing them. For example, a hole construct can be used to allow a space to leak data into another space enclosing it. Hole constructs can be unidirectional or bidirectional. An bidirectional hole can be used to facilitate data exchange between spaces in both directions.

Hyper Constructs

Hyper constructs are spatial constructs that can be assigned to more than one location in one or more spaces. Possible hyper constructs are aspect-like snippets that can observe and affect multiple locations within one or more spaces.

4.2.2 Spaces

Spaces can have two or more dimensions and serve as a runtime environment for spatial constructs. In particular, spaces are responsible for maintaining and diffusing data generated by spatial constructs.

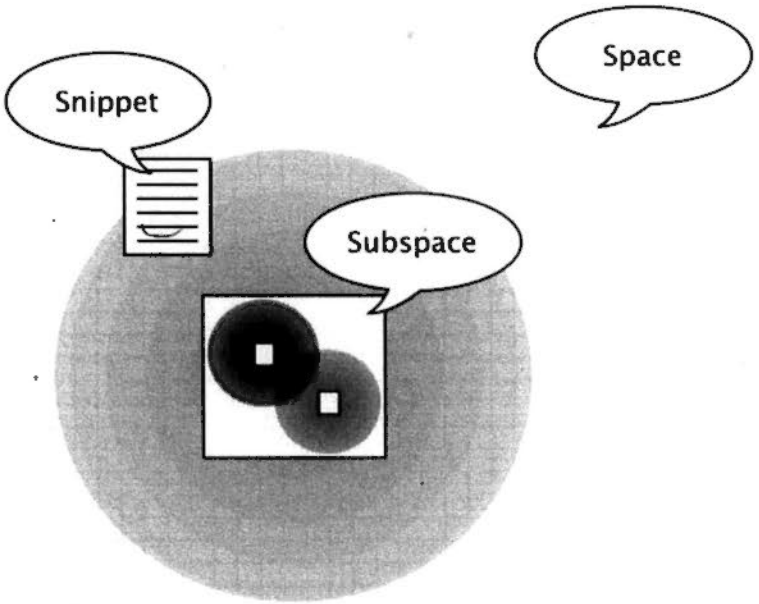

Figure 4.6: Space with Sub-Space

Spaces can be treated as spatial constructs themselves. As a result it is possible to construct flexible hierarchies of spaces (or hyper-spaces). Figure 4.6 shows a harmony-oriented space containing one snippet and one sub-space. Like other spatial constructs, the sub-space can moved around by software developers, and can generate and consume data.

Data generated or "leaked" by the sub-space is diffused in the parent space. Developers can place hole constructs into the sub-space to specify which data is leaked into the parent space.

Space Data

Harmony-oriented programs use dynamic typing and support data tagging. Tags are used by spatial constructs to describe and filter data. When a spatial construct puts data into the space, it is stored in the same location the spatial construct is in. For example, if the location of a spatial construct is (30, 50) in a two-dimensional space, then this location contains the state of the spatial construct, and initially all data the snippet explicitly puts into the space (before diffusion begins). If a spatial construct puts several values of the same data type into a space, the location stores the various values. As a result, no data generated by a spatial construct is ever discarded and spaces can be considered as the memory of a harmony-oriented program.

Substances and Diffusion

Spaces use so-called virtual *substances* to diffuse the state of and data produced by spatial constructs. Each time a new spatial construct is created, the space generates a corresponding substance in the same location.

Figure 4.7 shows two substances and their corresponding spatial constructs (snippets). A substance absorbs all data the spatial construct implicitly or explicitly produces, and the space starts diffusing it after it absorbs data for the first time. The diffusion process gradually increases the area covered by the substance. At its origin, substances have a very high intensity, which decreases when going towards the edges. For example, the substance in the upper left area of figure 4.7

might have an intensity of 1.0 at its center and intensities between 0.5 and 0.0 in area where it overlaps with the other substance corresponding to the spatial construct on the lower right.

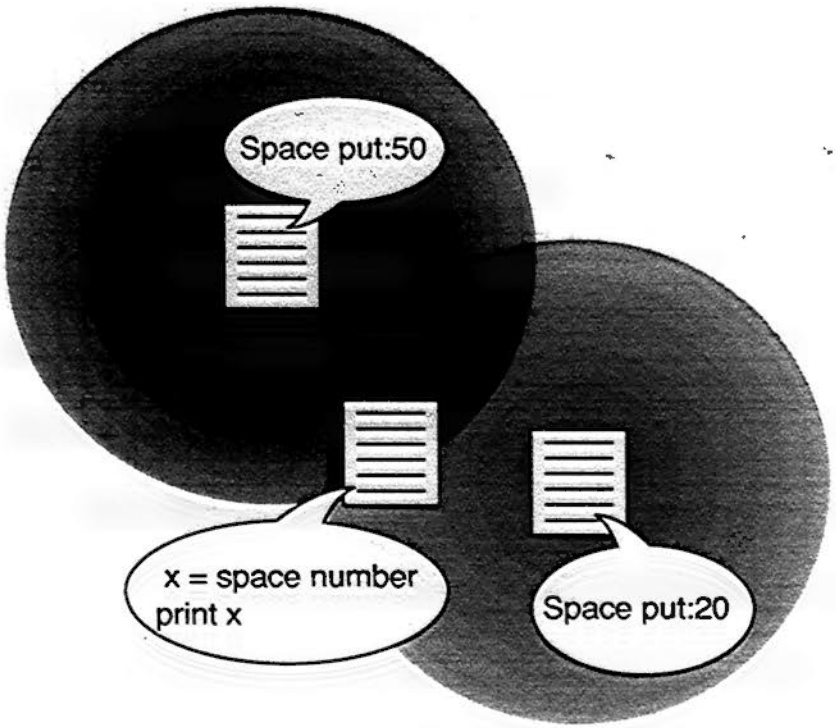

Figure 4.7: Substances and Diffusion

As shown in figure 4.7, the diffusion process eventually increases the extent of the substance so far, that it covers the locations of other spatial constructs. Once this happens, the space makes the data carried by the substance available to those other spatial constructs. In particular, when a spatial construct requests data from the space for consumption or observation, the space goes through the substances covering the spatial construct's location and selects and passes a matching data. If more than one substance contains data matching the requirements of the spatial construct, the space selects the data from the substance with the highest intensity value at the spatial construct's location. This selection process can be considered as a competition between substances.

The knowledge that the substance with the highest intensity is always favored, allows programmers to change the semantics of a program by adjusting diffusion parameters, such as setting the diffusion strength for a substance generated for the data of a specific snippet.

The example shown in figure 4.7 illustrates a two-dimensional space with three snippets. The upper left and lower right snippets both put a number into the space (50 and 20 respectively). The space generates corresponding substances and diffuses them. As shown in the figure, the diffused substances both reach the center snippet, which consumes a number from the space and outputs it to the console. Since the lower right snippet is closer to the center snippet, the intensity of its substance is higher than the intensity of the substance corresponding to the upper right snippet. As a result, the space passes the number 20 to the center snippet that outputs the number to the console.

4.3 Harmony-Oriented Smalltalk

Harmony-Oriented Smalltalk (HOS) [33] (figure 4.8) is a harmony-oriented runtime and visual development environment that allows programmers to implement harmony-oriented programs written in Squeak Smalltalk [7, 42, 18], a dialect of the Smalltalk programming language [59]. The visual development environment is based on Morphic [88, 64] and provides programmers with tools for inspecting spaces, editing snippets, changing diffusion settings and debugging.

Since HOS is based on the Smalltalk programming language, programmers have access to a vast object-oriented library providing networking, file access, and multimedia features. However, when constructing harmony-oriented programs, the primary decomposition is always in

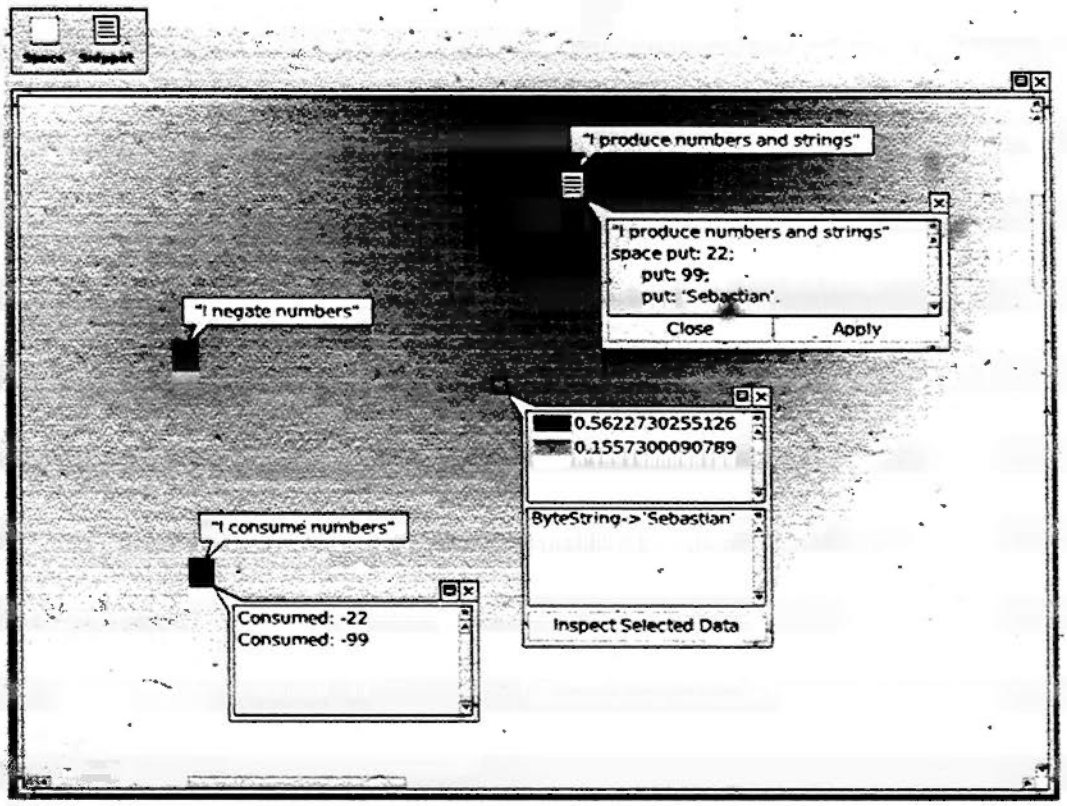

Figure 4.8: Harmony-Oriented Smalltalk

terms of spaces and spatial constructs, and not classes and objects, even though those are available.

The following sections provide an informal description of the HOS runtime and visual development environment. An initial semantics for Harmony-Oriented Smalltalk is introduced in appendix A.

4.3.1 Runtime Environment Overview

The HOS runtime environment provides an object-oriented interface for creating and running harmony-oriented programs consisting of two-dimensional spaces and snippets. In theory, harmony-oriented programming is not limited to two-dimensional spaces and snippets, but

the current version of HOS does not yet support other spatial constructs or spaces with more than two dimensions.

HOS spaces provide and implement the following features:

- Snippet scheduling. The space is responsible for scheduling snippet execution and controlling concurrency.
- Data management. The space is responsible for storing, diffusing and delivering data.
- Debug Interface. The space allows programmers to start and stop snippets and to observe and change its data.

In HOS, snippets are pieces of plain Smalltalk code that are assigned to a location within a space. Each time a snippet is executed, it receives a collection of objects implementing the so-called *snippet runtime interface*. These objects allow the snippet to exchange data with its space, maintain a state within the space, and to log messages. The snippet runtime interface is covered in detail in section 4.3.2. Here, one of these objects, which is named*space*, is considered to illustrate how to implement simple snippets that exchange data with their space. The *space* object provides methods for:

- Putting objects (data) into the space.
- Consuming objects (data) from the space.
- Peeking at / observing objects (data) in the space.

When snippets invoke methods of the *space* object to place data into the space, they can attach one or more string tags. Listing 4.1 shows a simple code snippet that puts three objects into the space.

```
1 "Example Snippet 1"
2 space put: 20 tag:'x'.
3 space put: 'Hello World!'.
4 space put: (TaggedData new
5     value:50;
6     addTag:'tagOne';
7     addTag:'tagTwo').
```

Listing 4.1: Snippet putting objects into the space.

The first object is a number with a tag, the second object is a string without any tags, and the third object is a *TaggedData* object that contains another number with two tags. The methods provided by the *TaggedData* class are summarized in section 4.3.3.

The methods provided by the *space* object for consuming data from the space let the programmer specify the required data type (e.g. Object, Number), required tags, and also allow passing a Smalltalk code block that can be used for implementing more advanced matching. These methods either consume a single matching object or set up a loop for consuming all available and future matching objects. Furthermore it is possible to instruct the space to wait for a specific combination of multiple objects (a set) and then consume it.

```
1 "Example Snippet 2".
2 space consume: Number do:[:num|
3     log show: num asString.
4 ].
```

Listing 4.2: Snippet consuming numbers.

Listing 4.2 illustrates a snippet that consumes all number objects. Listing 4.3 demonstrates a snippet that uses a *DataDescription* object to

consume all strings containing more than ten characters. Like *TaggedData*, the *DataDescription* class is covered in section 4.3.3.

```
1 "Example Snippet 3"
2 |description|
3 description := DataDescription new
4     type: String;
5     addTag: 'Name';
6     constraint:[:val| val size > 10].
7 space consume: description do:[:str| log show: str.].
```

Listing 4.3: Consuming strings with more than ten characters.

4.3.2 Snippet Runtime Interface

The snippet runtime interface is realized by three objects denoted as *space*, *state*, and *log*, which are available as soon as a snippets begin execution. These three objects provide a means for snippets to access the space, their state, and their log.

Log Object

The *log* object provides a method called *show:* method for outputting a string on the snippet's log. Each snippet has a private log that can be viewed by opening the snippet's console window (see section 4.3.6).

The code in listings 4.2 and 4.3 illustrate usage of the *log* object.

State Object

As explained in section 4.2.1, the state of a snippet is owned and diffused by the space. The *state* object provides a means for a snippet to access and change its state. In particular, snippets can change the data type of their state dynamically. For example, it is possible to change

the type of the state to *OrderedCollection* , and then treat it like a normal Smalltalk *OrderedCollection* object. The difference to using an *OrderedCollection* object directly is that all changes are delegated to the space automatically, which then diffuses the updated state. Apart from changing the type of their state, snippets can add tags to the state describing its contents.

Listing 4.4 shows how to dynamically change the type of the snippet state to *OrderedCollection* and then add three entries to the collection.

```
1 "State Example"
2 state type: OrderedCollection.
3 state add: 'Hello';
4       add: 'Harmony-Oriented';
5       add: 'World'.
```

Listing 4.4: Changing type and contents of snippet state.

Space Object

The *space* object facilitates interaction with the space containing the snippet. In particular, the *space* object provides methods for putting data into and consuming data from the space. The most important methods of the *space* object are:

- *consume: aDescription*

  Consumes one object matching the specified description from the space. The description parameter can either be a class name, such as *Object* or *Number*, a tag, or a data description object. If the enclosing space does not contain a matching object in the snippet's location, this method blocks and only returns when a matching object arrives via diffusion.

- *consume: aDescription do:aBlock*

  Consumes all objects matching the specified description. This method observes the snippet's location in the space and whenever a matching object arrives, the specified code block is evaluated with the matching object as parameter. After evaluation, the object is marked as consumed and discarded. See listing 4.2 for an example.

- *consumeAll: aBlock*

  Consumes all objects arriving at the snippet's location in the space.

- *consumeSet: tttdList do:aBlock*

  In certain cases, snippets might want to consume multiple objects at the same time. For example, consider a snippet that performs a repeated calculation that requires a two numbers as input for each iteration. Lets assume a space containing one or more snippets that produce various kinds of numbers where each number is tagged with either 'x' or 'y'. In order for the snippet performing the calculation to retrieve one number tagged 'x' and one number tagged 'y' from the space, the *consumeSet:do:* method can be used as illustrated in listing 4.5.

- *observe: aDescription do:aBlock*

  This method is similar to the *consume:do:* method. It evaluates the specified code block for all objects matching the specified description, but does not mark the object as consumed after evaluation and discards them.

- *observe: aBlock*

  This method passes all objects arriving at the snippet's location in the space to the specified code block, and retains them after

the code block is evaluated.

- *retrieveLatest: aDescription ifAvailable: block1 else: block2*
  This method is different from the previous methods for observing and consuming objects, because it does not block until matching objects arrive. This method inspects the snippet's location and searches for the most recently arrived object matching the specified description. If a matching object is found, the first code block is evaluated with the object as a parameter. If no matching object is found, the second code block is evaluated.

- *put: anObject*
  Puts an object into the space. The object can be any Smalltalk object. When the method processed the passed object, it creates a *TaggedData* object and places it in the space. If the passed object is a *TaggedData* object, this method passes it through to the space without making any modifications. The third line of listing 4.1 illustrates usage of this method:

- *put: anObject tag: aString*
  This method is a convenience method that creates a *TaggedData* object from a Smalltalk object and a specified tag, and then puts it into the space.

```
1 "Example Snippet 4"
2 space consumeSet:{'x'.'y'} do:[:x :y|
3     log show: (x+y) asString.
4 ].
```

Listing 4.5: Snippet consuming sets of objects.

4.3.3 Data Descriptions and Tagged Data

As explained in section 4.3.1, the HOS runtime environment provides two classes for defining data descriptions and tagged data: *DataDescription* and its subclass *TaggedData*.

Instances of the *DataDescription* class are used by snippets to describe what kind of data they would like to receive from the space. The methods of this class allow specification of type, tags, and Smalltalk code blocks implementing customized comparisons. Instances of the *TaggedData* class are used explicitly or implicitly to wrap data snippets put into the space. Listing 4.1 contains an example for using the *TaggedData* class and listing 4.3 illustrates the usage of the *DataDescription* class.

The most important methods of the two classes are:

- *DataDescription* ≫ *addTag: aString*
  Add a new tag to the data description.
- *DataDescription* ≫ *tags*
  Return a collection containing all tags.
- *DataDescription* ≫ *type: aType*
  Set the type of the data description. The *aType* parameter can be any Smalltalk class.
- *DataDescription* ≫ *type*
  Return the type of the data description.
- *DataDescription* ≫ *constraint: aBlock*
  Specify a Smalltalk code block that can be used for customized comparisons of a data description with other objects. See line 6 of listing 4.3 for an example.

- *DataDescription* ≫ *constraint*
  Return the constraint code block.

- *TaggedData* ≫ *value: aValue*
  Set the value (data) of a tagged data object.

- *TaggedData* ≫ *value*
  Return the value (data) of a tagged data object.

4.3.4 Snippet Scheduling

The spaces of the HOS runtime use lightweight (green) threads for controlling concurrency of snippet execution. Generally, whenever a snippet is created or changed, it is automatically (re-)compiled and then scheduled for execution. Spaces provide methods that programmers can use to start, stop, restart and discard snippets.

4.3.5 Data Management and Diffusion

Data exchange in harmony-oriented programs is facilitated via diffusion of substances that represent all data a snippet has put into its enclosing space. In HOS a substance is essentially a data queue with an associated intensity. Whenever a snippet puts an object (data) into the space, it is stored into the data queue of the corresponding substance and information about all data inside the queue is diffused throughout the space.

For performance reasons, HOS spaces do not continuously perform diffusion at all times. Each snippet has a specific "diffusion limit" that can be set by the programmers. One the limit is reached, diffusion of the corresponding substance stops and remains at its current level. As a result, diffusion only occurs after the following events:

- When a snippet puts data into the space for the first time.
- After a programmer manually resets the diffusion.
- When the position of a snippet changes.

The HOS runtime environment is designed to support various diffusion algorithms and strategies, and provides facilities for running diffusion at different speeds for visualization purposes.

The default diffusion algorithm used by the HOS runtime is a combination of the general diffusion equation described in [45] and a distance function over locations and snippets in order to enhance performance. The equation in [45] is:

$$S_{x,y} = \frac{1}{4}\sum_{k=1}^{4}[(1-c_d)S_{x,y} + c_d S_{n_k(x,y)}] \qquad (4.1)$$

In the above equation, $S_{x,y}$ stands for the intensity of a given substance S and at the position (x,x) in a given field, and $n_k(x,y)$ represents the kth nearest neighbor of (x, y).

The equation above can also be written as follows:

$S_{x,y} = (1-c_d)S_{x,y} +$
$\frac{c_d}{4}(S_{x-1,y} + S_{x+1,y} + S_{x,y-1} + S_{x,y+1})$.

It is important to note that any diffusion algorithm (or combination of algorithms) can be used instead without affecting the functionality of the HOS runtime environment. However, since Smalltalk is slow in comparison with languages like C, it is desirable to use efficient algorithms.

4.3.6 Visual Development Environment

Harmony-Oriented Smalltalk provides a complete visual integrated development environment called HOS IDE. Programmers can create spaces and snippets via drag and drop, and examine and change the state of harmony-oriented programs. In particular, the HOS IDE provides the following features:

- Create spaces and snippets via drag and drop.
- Edit, start, stop, and delete snippets
- Scroll and zoom the contents of spaces. Center the scroll view on specific snippets.
- Set diffusion parameters, such as diffusion type and limit.
- Select locations (cells) of the space and inspect and change their data.

The following sections provide an overview of the main features provided by the HOS IDE.

Creating Snippets And Spaces

When opening the HOS image, a toolbox (parts bin) for creating new spaces and snippets is shown in the upper left corner of the screen (figure 4.9). New spaces and snippets can be created by dragging the corresponding icon to the screen.

Snippets can also be created by left clicking a location in a space. A new HOP parts bin can be programmatically opened by executing the following code in a workspace:

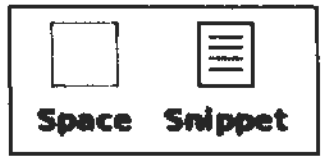

Figure 4.9: HOP Parts Bin

```
HopPartsBin new openInWorld.
```

Space Context Menu

The space context menu is shown when clicking a location in a space. It provides the following options:

- Create Snippet.
  Create a new snippet in the selected location.
- Inspect Location.
  Open an inspector window that allows the programmer to view and manipulate data at the location.

Figure 4.10 shows an open location inspector for a cell that contains two substances. The upper part of the inspector shows all substances available at the location. For each substance, the location inspector displays its color and intensity, which has a value between 0.0 and 1.0. The lower part of the location inspector shows the contents of the substance, which is a summary of all unconsumed data the corresponding snippet has put into the space. It is possible to select those values and inspect them using Squeak's default object inspector.

The location inspector allows programmers to inspect and change the data, and to find out what kind of data a newly created snippet in the selected location could consume. The substances are ordered according

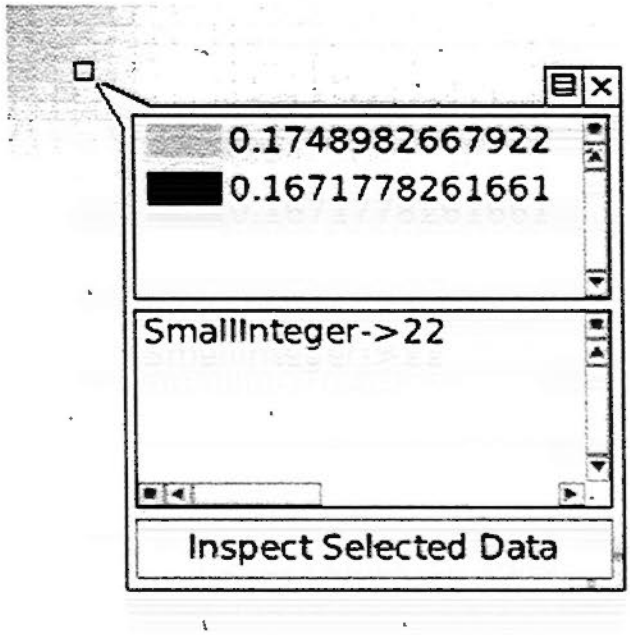

Figure 4.10: Location Inspector

to the values of their intensities. As explained in section 4.2.2, the higher the value of the intensity the higher is priority the space uses for passing data to a snippet, if multiple substances provide data that matches the kind of data a snippet wishes to consume.

Space Main Menu

The space main menu (figure 4.11) is opened by clicking the menu icon in the upper right corner of a space. It provides the following options:

- Hide / Show Grid
  Hide or shows a grid indicating the locations (cells) of the space.
- Hide / Show Diffusion
  Hide or shows diffusion of substances.
- Snippets (Submenu)
  A submenu for selecting a specific snippet. After a snippet is

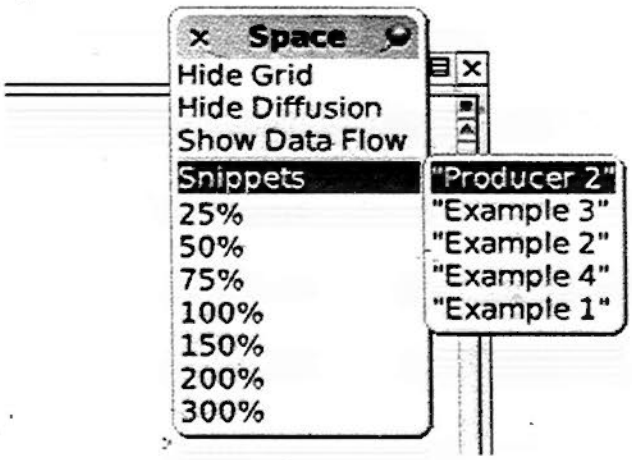

Figure 4.11: Space Main Menu

selected the space scrolls automatically to ensure the snippet is visible.

- Zoom

  The zoom options allow the programmer to zoom the space. The maximum zoom is 300 percent and the minimum zoom is 25 percent.

Snippet Context Menu

The snippet context menu is shown when clicking a snippet inside a space and provides the following options:

- Pick Up

  Pick up the snippet (start dragging the snippet).

- Hide / Show Label

  Hide or show a label displaying the first line of code of the snippet. The sample programs shown in this paper follow the convention that the first line of each snippet is a comment that names or describes the purpose of the snippet.

- Hide / Show Console
  Hide or show the console of a snippet. Any log messages a snippet generates are displayed in its console.
- Inspect Location
  Open an inspector window that allows the programmer to view and manipulate data at the location of the snippet.
- Inspect Diffusion
  Open an inspector window that allows the programmer to view and set diffusion parameters for the snippet.
- Edit
  Open an editor for changing the snippet code.
- Delete
  Delete the snippet.
- Run / Stop
  Starts or stops the execution of the snippet.

Figure 4.12 shows the snippet console. It displays all log entries snippets produce by sending the *show:* message to the *log* object. The snippet console provides a menu that allows clearing all log entries.

The diffusion inspector window (figure 4.13) allows programmers to adjust diffusion parameters for the substance associated with the selected snippet. When opened, the diffusion inspector shows the selected diffusion type and the current level (progress) of diffusion.

The programmer can set the diffusion type to "quick" or "slow". The former instructs the space to perform diffusion as quickly as possible, and the later instructs the space to perform diffusion slow enough that

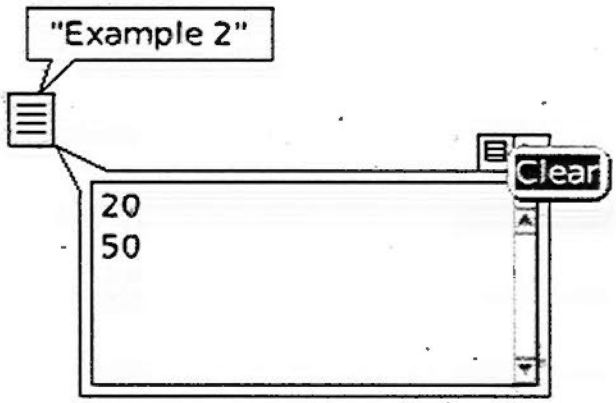

Figure 4.12: Snippet Console

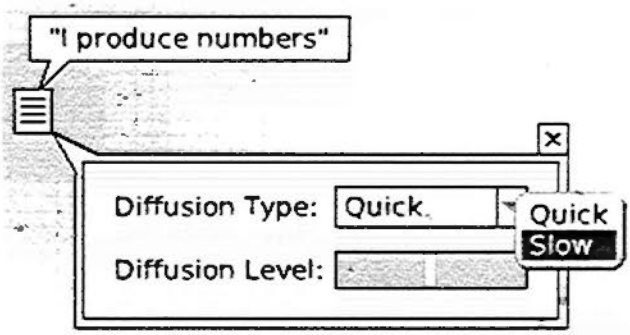

Figure 4.13: Diffusion Inspector

programmers can watch its progress. The "slow" option is provided for visualization/demonstration purposes.

The diffusion level indicates the current progress of diffusion. Programmers can use the slider to adjust the diffusion level manually. As a result, the diffusion level both serves as progress indicator and limit for the diffusion of the substance corresponding to the selected snippet.

The snippet code editor shown in figure 4.14 is similar to standard Smalltalk workspaces and provides basic editing features. When the programmer presses the *Apply* button, the snippet code is compiled and executed immediately.

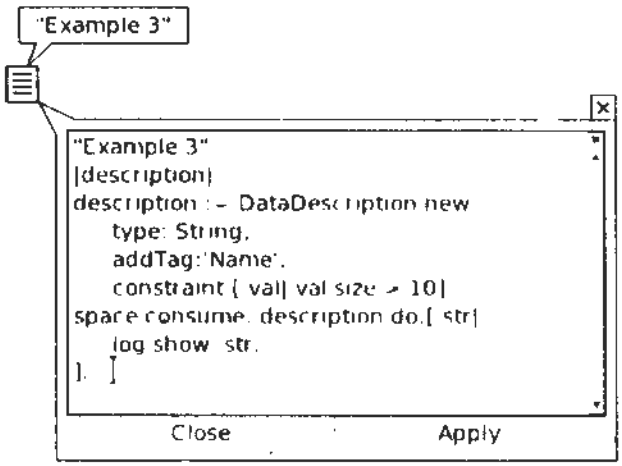

Figure 4.14: Snippet Editor

4.3.7 Debugging

HOS snippets can be debugged like any other Smalltalk code. If an error occurs or the snippet sends the *halt* message to itself, the standard Squeak debugger window is shown.

4.4 Summary

This chapter introduces the first resonance-oriented software development approach: harmony-oriented programming. The first part of the chapter describes the principles of harmony-orientation and the conceptual structure and constructs of harmony-oriented programs.

The second half of the chapter introduces HOS, a Smalltalk-based harmony-oriented runtime and visual development environment. The description of HOS covers details of programming interfaces and the main elements and features of the visual development environment.

□ **End of chapter.**

Chapter 5

Approach II: Epi-Aspects

This chapter proposes a second resonance-oriented approach called *epi-aspects* [30]. The epi-aspects architecture is based on aspect-oriented programming [57] and conscientious software. Its goal, and that of conscientious software, is to allow the separation of the core application functionality (the allopoietic part) from the monitoring, regulation, and error recovery concerns, as provided by the autopoietic part. The epi-aspects architecture is the first concrete realization of the theoretical notion of conscientious software envisioned by Gabriel and Goldman. Since conscientious software consists of distinct allopoietic and autopoietic parts, aspects are a natural choice for "gluing" the two parts together to form a conscientious system.

The proposed architecture (figure 5.1) consists of three portions: the allopoietic part (referred to as the application), the autopoietic system, and epi-aspects, which are the glue that binds the other two portions. Epi-aspects are able to advise on join points in both the application and autopoietic system, and so facilitate resonance and feedback, and assist the autopoietic system in keeping the software in a harmonious state. In particular, they extend the application with functionality required for evaluating its health, and for performing adjustments requested by

the autopoietic system. Such extensions include functionality for testing, upgrading, cloning, restarting, and killing allopoietic components. As a result, epi-aspects can be used to upgrade existing applications into conscientious software in a non-invasive manner.

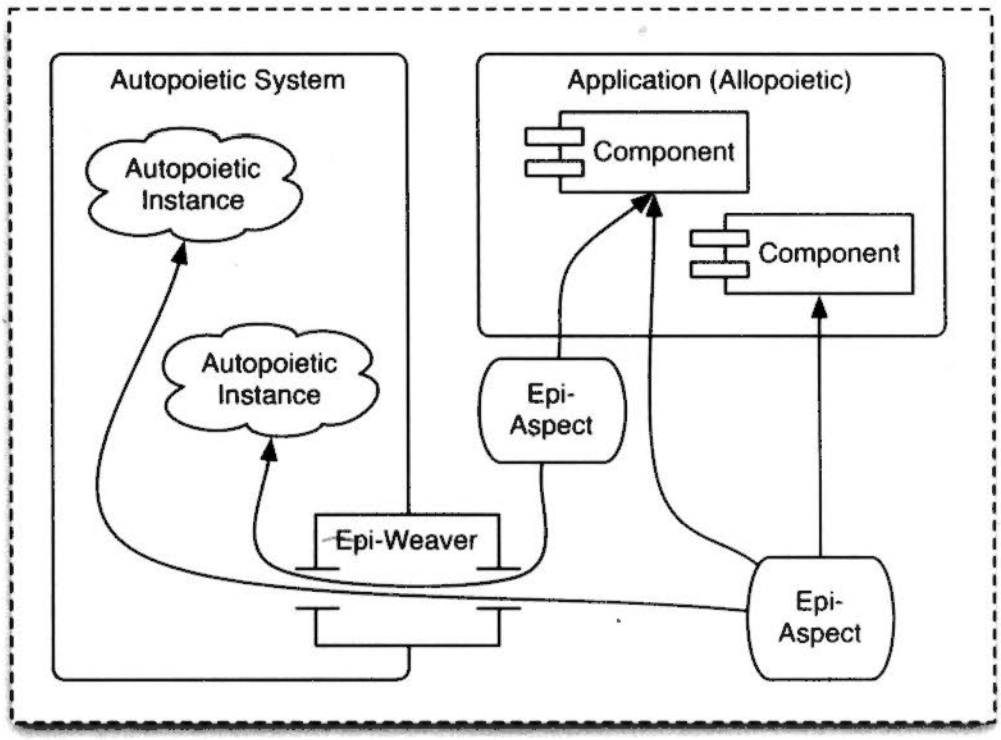

Figure 5.1: Epi-Aspects Architecture

This chapter describes the conceptual design of epi-aspects and proposes a concrete framework for developing epi-aspects in Java.

5.1 Proposed Architecture

The epi-aspects architecture (figure 5.1) consists of three portions:

- An application (the allopoietic system) that implements some desired functionality.

- An *autopoietic system*, responsible for keeping the system running smoothly.
- *Epi-aspects*, which act as a bridge between the other two portions by extending the application with functionality related to self-sustainment, such as test routines and maintenance function.

The following sections provide a detailed description of the proposed epi-aspects architecture. Section 5.1.1 covers the allopoietic application, section 5.1.2 illustrates the features and behavior of the autopoietic system, and section 5.1.3 describes the anatomy of epi-aspects.

5.1.1 Allopoietic Application

This portion of the proposed architecture is a traditional application, such as a content management system. No alteration of this application is necessary since all conscientious functionality is encapsulated in the other two portions of the system. This chapter assumes object-oriented or component-based applications and the term *application entity* is used to refer to an instance of an object or component.

5.1.2 Autopoietic system

The autopoietic system is a network of instances that constantly observes itself, its environment, and the state of the application, which is exposed by epi-aspects. In case of any problems, such as errors in the application, the autopoietic system makes *queries*, and *recommendations* to correct the problem. The autopoietic system does not employ artificial intelligence; software developers explicitly implement the conditions for triggering autopoietic recommendations and queries.

Queries are used by the autopoietic system to monitor the health of the application. Recommendations are made either routinely, or based on the result of a query. Both recommendations and queries can be defined and customized by the developer of the autopoietic system. The proposed architecture provides a core set of each, as listed in table 5.1 (recommendations) and table 5.2 (queries).

Start, Stop, Create, Destroy, Clone	These suggestions are directed at application entities.
Update	Recommends to update the application software. This can be directed at the whole application or application entities.
Revert	After a software update, the autopoietic system an advise the application to revert to the previous version. This recommendation is used if a problematic software update is applied.
Test	Recommends to test an application entity.
Custom	This recommendation allows the autopoietic system to advise the application to perform a custom task.

Table 5.1: Autopoietic Recommendations

In order to facilitate epi-aspects, which implement advice for recommendations and queries, the autopoietic system includes an internal run-time weaver for epi-aspects, called the *epi-weaver*. The epi-weaver depends on the programming language used to implement the epi-aspects, because not all programming languages provide compatible approaches for invoking methods or routines. In order to support a specific programming language, a customized weaver has to be devel-

Reveal	This query indicates that the autopoietic system requires information about specific application entities or the application as a whole. For example, in order to evaluate the health of a component, autopoietic instances have to be able to examine its internal state.
Speed	This query indicates that an autopoietic system wishes to obtain information about the current speed (performance) of a certain application entity.
Custom	Allows specification of custom queries. The purpose of custom queries is to provide a flexible mechanism for extending the autopoietic system.

Table 5.2: Autopoietic Queries

oped and integrated with the autopoietic system.

5.1.3 Epi-Aspects

The purpose of epi-aspects is to facilitate resonance and feedback by making the application visible to and controllable by the autopoietic system. In particular, epi-aspects are responsible for implementing self-sustaining concerns and any functionality required for smooth interaction between the autopoietic system and application.

Since epi-aspects crosscut the autopoietic system and application, they support two types of advice. The first type advises on recommendations and queries of the autopoietic system, and performs required tasks, such as testing and other maintenance. The second type advises on join points in the application and is responsible for keeping the autopoietic system updated on the application's status and performance.

Each epi-aspect contains:

1. An *epi-queue*, which is used to dispatch information from epi-aspects to the autopoietic system.

2. *Application advice*, which are responsible for providing feedback on the application's health to the autopoietic system. This feedback is passed to the autopoietic system through the epi-queue.

3. A mechanism for defining advice on autopoietic recommendations and queries. Such *query advice* and *recommendation advice* perform maintenance or update operations according to suggestions by the autopoietic system, or to provide information to satisfy an autopoietic query.

As epi-aspects implement advice for both application and autopoietic system, they have to be woven twice by different weavers[1]. One weaver is responsible for weaving application advice during compilation or at run time. The other weaver is the epi-weaver of the autopoietic system. For example, let's consider an epi-aspect denoted as *DatabaseEpi-Aspect*, which is part of an upgrade to turn an application using a database into conscientious software. The application advice of this epi-aspect are woven into the code of the application's database engine during compile-time, and the recommendation and query advice are woven into the autopoietic system during run-time.

Epi-Messages and Epi-Queue

The autopoietic system and epi-aspects communicate by exchanging *epi-messages*, which consist of the attributes shown in table 5.3. The autopoietic system can pass epi-messages as parameters to recommendation and query advice implemented by epi-aspects.

[1] Aspect weavers combine aspects and the code they advise on.

Sender, Receiver	Both autopoietic instances and application entity can acts as sender and receiver.
Type	Indicates the type of the epi-message. Can be one of the following: • Parameter: Epi-message is passed to an autopoietic recommendation or query advice as parameter. • Answer: Epi-message is an answer to an autopoietic query. • Feedback: Epi-message contains feedback regarding the allopoietic application. • Error: Epi-message reports an error that occurred in the allopoietic application. • Epi-Error: Epi-message reports an error that occurred in an epi-aspect.
Contents	Contains the contents of the message. The format of this message could be anything from strongly typed data over XML to natural language. The only requirement is that both autopoietic instances and epi-aspects are able to interpret this format.

Table 5.3: Epi-Message Attributes

Each epi-aspect has access to an *epi-queue*, which can be used to dispatch information to the autopoietic system. Before an epi-aspect is woven into the autopoietic system, it merely contains a stub for the epi-queue that does not contain any functionality. When the epi-weaver processes an epi-aspect, it does not only weave recommendation and query advice, but also injects a concrete implementation of the epi-queue. This approach ensures that epi-aspects do not depend on the concrete realization of the autopoietic system.

Application advice

Application advice (table 5.4) are defined on joint points in the allopoietic application. Their purpose is to observe one or more specific application entities and to expose their state to the autopoietic environment. Depending on the join point it advises on, the implementation of an application advice gathers and optionally forwards information to the autopoietic system. Information is dispatched to the autopoietic system via the epi-queue.

Existing aspect-oriented programming languages, such as AspectJ [56], provide sufficient pointcut primitives to describe most of the application join points required by epi-aspects. As a result, epi-aspects can be realized as an extension of existing aspect-oriented programming languages.

The following application-level joinpoints are required by the epi-aspect architecture:

- *Create*: Creation of a new application entity, such as the construction of an object.
- *Destroy*: Destruction of an application entity, such as a component or object.
- *Error*: Unexpected errors or exceptions.
- *Invocation/Execution*: Invocation and execution of methods.
- *Event*: Events, in an event handling system.
- *Set*: Setting of a variable or property value.

Table 5.4 provides the details of how a selection of the application advice might be used.

Error	Advice for errors or exceptions gather as much information on the error as possible, including the source of and reason for the error, and then dispatch this information to the autopoietic system via the information queue. For example, in the OIM service, error application advice can be defined for exceptions thrown by the database engine and XML-RPC service.
Creation	Advice for creation join points are responsible for informing the autopoietic system which application entities, such as objects, components, and modules, exist. The autopoietic system uses this information to decide which entities should be monitored.
Performance	Performance advice are invoked before and after certain methods or procedures in the application. Their purpose is to measure the execution time of methods and report it to the autopoietic system. This allows the autopoietic system to keep track of the application's performance and detect possible timeouts.

Table 5.4: Application Advice

Query and Recommendation Advice

Query and recommendation advice implement maintenance and information retrieval operations proposed by the autopoietic system. They are woven into the autopoietic system by the epi-weaver at runtime.

Developers of a software system might implement recommendation advice to implement a unit test to verify the proper operation of a network communication component, or to apply updates to the system. They may implement a query advice to implement an evaluator that evaluates the performance (speed) of the network communication component. That implementation would advise on the *Speed* suggestion, and would help the autopoietic system to keep track of the size of the workload on the network communication component.

Query and recommendation advice consist of two portions:

- *Header*: Contains the attributes *name* and *receiver-pattern*. The *name* attribute is the name of a predefined (see tables 5.1 and 5.2) or custom recommendation or query. The *receiver-pattern* attribute contains a regular expression for matching the target application entity.
- *Implementation*: Contains the allopoietic code for implementing the requested action. It uses epi-messages as input and output parameters as a means for communication between the autopoietic system and epi-aspect.

5.2 Epi-Aspects Java Framework

This section specifies a framework for developing aspect-oriented conscientious software in Java. This framework, which is called *Epi-AJ*, provides an autopoietic simulator and constructs for implementing epi-aspects in the Java programming language. The autopoietic simulator includes a weaver for epi-aspects, and contains a logic engine based on declarative rules, which implements the behavior of the autopoietic system described in section 5.1.2.

The Epi-AJ framework is designed as a supplement to Aspect-J [56]. Since version 5, AspectJ has supported the usage of Java annotations [69] for defining aspects and advice. Epi-AJ provides a set of Java annotations, which allow the definition of autopoietic recommendation and query advice. As a result, an epi-aspect can be implemented using a combination of AspectJ annotations and Epi-AJ annotations. The usage of annotations is convenient in a sense that it is not necessary

to use tools like the AspectBench compiler [5] to extend the grammar of the AspectJ pointcut language with new pointcut primitives for epi-aspects. Listing 5.1 illustrates the definition of an epi-aspect using the combination of Aspect-J and Epi-AJ.

```
1  @Aspect public class XMLRPCEpiAspect extends EpiAspect
2  {
3      @After("this(s) && execution(XMLRPCService.new(..))")
4      public void newInstance(XMLRPCService s) {/*...*/}
5
6      @AfterThrowing("target(s) &&
7      execution(* XMLRPCService.run(..))")
8      public void reportException(XMLRPCService s)
9      {/*...*/}
10
11     @RecommendationAdvice(recommendation="start",
12     recipientPattern=".*")
13     public EpiMessage
14     startXMLRPCServer(EpiMessage message) {/*...*/}
15
16    /*...*/
17 }
```

Listing 5.1: Epi-Aspect Example Code (Epi-AJ Framework)

The Epi-AJ framework is divided into three Java packages:

1. The package *conscientious.epiaj* (shown in figure 5.2) contains the base classes and interfaces of the framework.
2. The package *conscientious.epiaj.annotations* contains annotations for declaring autopoietic recommendation and query advice in epi-aspects.

3. The package *conscientious.simulator* contains the Java part of the autopoietic simulator, such as the implementations of the epi-weaver and epi-queue.

5.2.1 Base Classes and Interfaces

As illustrated in figure 5.2, The Epi-AJ framework provides the following set of base classes and interfaces for realizing epi-aspects, epi-messages, and epi-queues: *EpiAspect*, *EpiQueue*, and *EpiMessage*.

As described in section 5.1.3, epi-aspects and the autopoietic system communicate by exchanging epi-messages. Each epi-aspect has access to an epi-queue that allows them to dispatch epi-messages to the autopoietic system.

The abstract class *EpiAspect* is the base class for epi-aspect implementations, *EpiQueue* defines the interface of epi-queue implementations, and the class *Epi-Message* is the implementation of the epi-message illustrated in table 5.3.

The *EpiAspect* class contains an instance variable whose type is the *EpiQueue* interface. When an the implementation of an epi-aspect is woven, the autopoietic system (or autopoietic simulator) assigns a concrete epi-queue implementation to this instance variable. After that, the epi-aspect implementation can start dispatching epi-messages to the autopoietic system.

The *Epi-Message* class contains four instance variables, which are equivalent to the epi-message attributes *Sender*, *Receiver*, *Type*, and *Contents* described in table 5.3.

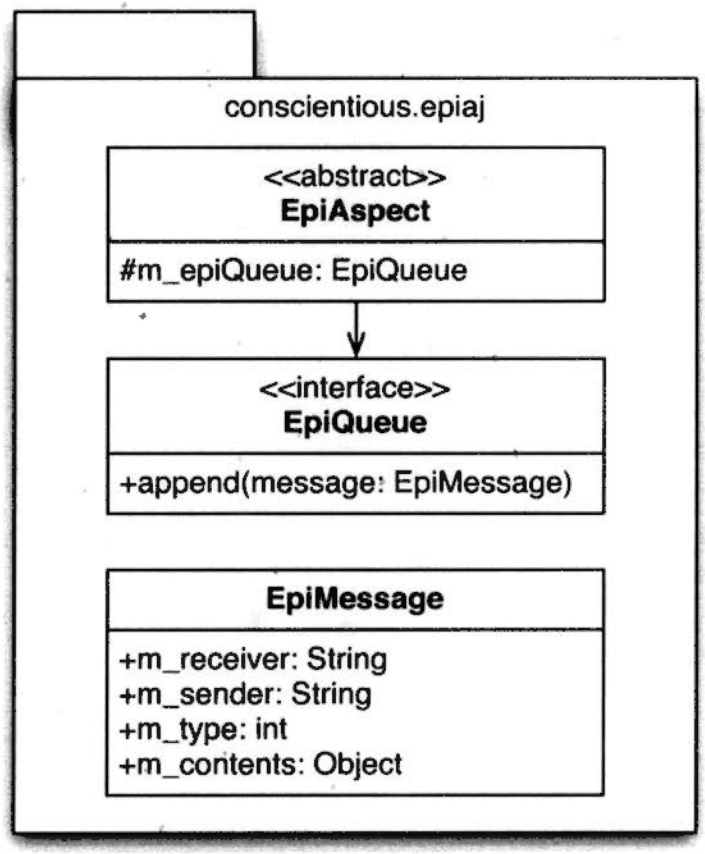

Figure 5.2: Epi-AJ Base Classes

5.2.2 Advice and Annotations

The Epi-AJ framework provides the following set of Java annotations for declaring recommendation and query advice:

- @RecommendationAdvice(name, receiverPattern)
- @QueryAdvice(name, receiverPattern)
- @RevealQA(receiverPattern)
- @SpeedQA(receiverPattern)
- @TestRA(receiverPattern)

- @UpdateRA
- @RevertRA(receiverPattern)
- @CloneRA(receiverPattern)
- @CreateRA / @DestroyRA(receiverPattern)
- @StartRA / @StopRA(receiverPattern)

All annotations have a *receiverPattern* attribute that can be used to specify a regular expression for matching the class/epi-aspect at which the autopoietic recommendation or query is directed. The @RecommendationAdvice and @QueryAdvice annotations are generic annotations that can be used to declare advice on any autopoietic recommendation and query, including custom recommendations and queries. The remaining annotations are provided for convenience and can be used to specify advice on the pre-defined autopoietic recommendations and queries described in tables 5.1 and 5.2.

As shown in listing 5.1, the implementation part of the recommendation and query advice is a Java method that receives an *EpiMessage* object as parameter and returns another *EpiMessage* object to the autopoietic system. The *EpiMessage* parameter is set up by the autopoietic system to specify details regarding the recommendation or query, and the *EpiMessage* return value contains feedback or other information for the autopoietic system.

5.2.3 Autopoietic Simulator

The Epi-AJ framework provides an autopoietic simulator that can be used for developing and testing epi-aspects. This simulator consists

of a runtime, an epi-weaver written in Java, and uses the Prolog programming language [10, 89] to implement the rules of the autopoietic system. Prolog is not an autopoietic programming language that is specifically designed to prevent bugs that can lead to program crashes. However, as the design of an autopoietic programming language is not within the scope of this paper, Prolog is a suitable substitute for simulation purposes, because it is declarative and it is not easy to write a Prolog program that crashes.

The autopoietic simulator can be invoked from a Java program by creating and configuring an instance of the *Simulator* class shown in figure 5.3. Internally, the *Simulator* class uses the SWI-Prolog engine [100] to simulate the behavior of the autopoietic system. Interaction between the *Simulator* instance and the SWI Prolog engine is accomplished through the JPL (Java Interface to Prolog) API, which is part of the SWI-Prolog distribution.

When a new instance of the *Simulator* class is created, the calling application provides a list of epi-aspects. During its initialization, the *Simulator* instance performs the following tasks:

1. The SWI Prolog engine is initialized and the Prolog program(s) mimicking the autopoietic systems are loaded.
2. An instance of the *AdviceRepository* shown in figure 5.3 is created.
3. An instance of the *Weaver* class is created and the list of epi-aspects is passed to it.
4. The *Weaver* instance weaves the epi-aspects into the *AdviceRepository* instance. Moreover, it injects an instance of the *EpiQueueImp* class, which implements the simulators epi-queue into each woven epi-aspect.

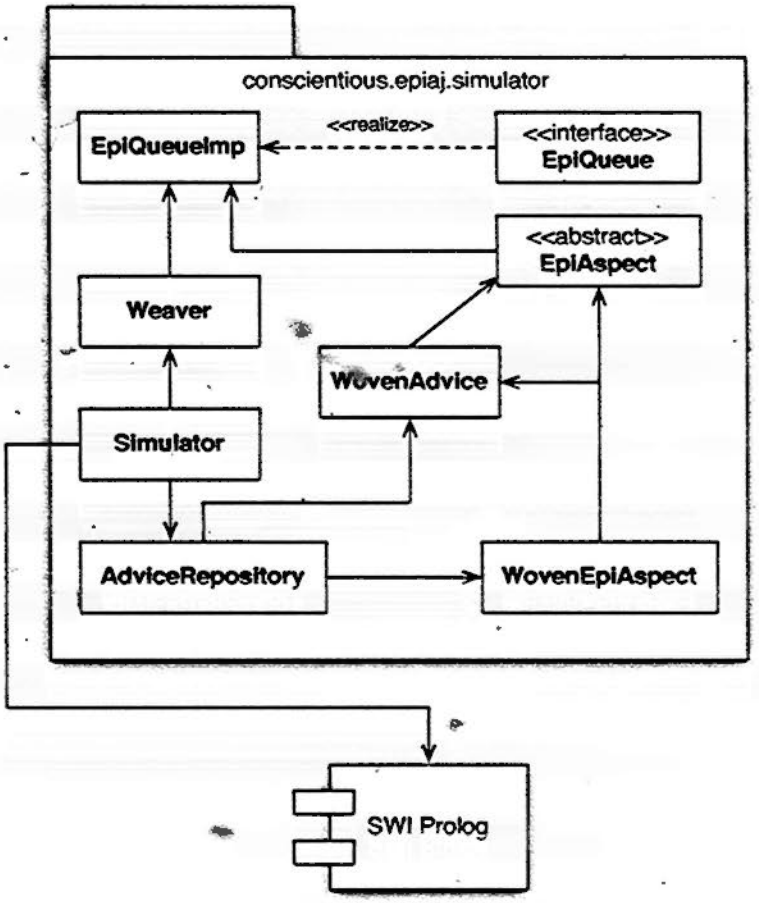

Figure 5.3: Epi-AJ Autopoietic Simulator

5. The *Simulator* instance issues the autopoietic recommendation *Start*, which is dispatched to all woven epi-aspects.

Once the autopoietic simulator is running, the woven epi-aspects can dispatch epi-messages to it via the *EpiQueueImp* instance. Whenever an epi-message is received, the Prolog program(s) are invoked and the result can be an autopoietic recommendations or query. Autopoietic recommendations and queries are dispatched to relevant advice of the woven epi-aspects.

5.3 Summary

This chapter introduces a second resonance-oriented approach called epi-aspects. Apart from describing the conceptual architecture, Epi-AJ, a Java framework for realizing epi-aspects is introduced.

□ **End of chapter.**

Chapter 6

Studies and Validation

This chapter describes studies aimed at supporting the hypothesis formulated in section 1.5. The hypothesis states that, in comparison with traditional object-oriented programming, resonance-oriented software design and development improves the ease of dealing with the main factors affecting software evolution: ease of changing a program's design and structure (changeability), extensibility, maintainability, quality feedback, and error recovery. The studies evaluate how the proposed resonance-oriented approaches improve these factors in comparison with object-oriented programming.

Section 6.1 introduces the general design for the studies presented in this chapter and discusses how possible threats to validity are addressed. Section 6.2 compares the first resonance-oriented software development approach, harmony-oriented programming, with object-oriented programming in terms of changeability, extensibility, and maintainability. In section 6.3, the quality feedback and error recovery capabilities of epi-aspects are evaluated and compared to object-oriented programming. Section 6.4 describes a software evolution study based on a real world example using an architecture called *harmony-oriented*

cpi-aspects that combines the strengths of the two proposed resonance-oriented software development approaches.

6.1 General Study Design

The general setup for the studies presented in this chapter is as follows: each study consists of one or more experiments, and each experiment has at least one part that conducts a series of design or development tasks using a resonance-oriented approach, and at least one other part that performs the same series of tasks using traditional object-oriented programming. At the end of each experiment the design and development processes are compared.

The studies are designed to achieve construct validity, internal validity, and external validity by addressing common threats.

6.1.1 Construct Validity

One possible threat to construct validity is that experiments do not test the factors meant to be evaluated. The aim of the studies presented in this chapter is to evaluate and compare the following factors in the context of resonance-oriented software development and object-oriented programming: changeability, extensibility, maintainability, quality feedback, and error recovery.

- Changeability refers to changing a programs design and structure. In the context of object-oriented programming, changeability refers to adjusting interfaces of and relationships between objects. The changeability studies in sections 6.2.1 and 6.2.2 thus focus on relationships and interaction between program parts.

- Extensibility and maintainability refer to the case of extending or changing a program with or without having to change the program's overall structure. This includes unpredicted extensions and changes that were not considered in the initial design of the program. The studies in sections 6.2.3 and 6.4 evaluate extensibility and maintainability by comparing the implementation of extension mechanisms and unexpected changes.

- Feedback and error recovery refer to the case of implementing reliable mechanisms for generating information about the status (health) of program parts and assisting them with recovering from errors. The studies in sections 6.3 and 6.4 evaluate these two factors by studying the implementation of such mechanisms, and generating and simulating failures.

Biases, such as the mono-operation bias and the "researcher bias", are another common thread to construct validity.

The mono-operation bias refers to evaluating a certain method or program once in a single place at a single point in time. The studies presented in this chapter avoid this bias by using each of the two proposed resonance-oriented approaches for various studies and combining the two approaches into a new one in section 6.4.

The "researcher bias" refers to the fact that a person conducting an experiment can affect its result consciously and unconsciously. While it is impossible to completely eliminate this bias, the studies attempt to make fair comparisons between the resonance-oriented approaches and object-oriented programming by avoiding scenarios whose conditions significantly favor one over the other. For example, the software evolution study in section 6.4 uses the same initial (flawed) application

design for both resonance-oriented and object-oriented approaches to avoid giving an unfair advantage to the former.

6.1.2 Internal Validity

Within each experiment two programming approaches are used to perform the same series of tasks: a resonance-oriented approach and traditional object-oriented programming. The programming approach is the independent variable of each experiment. The scope of the experiments conducted within the studies in this chapter is relatively small and does not generate large amounts of data. As a result, extraneous variables common in large scale studies, such history, maturation, testing, instrumentation, statistical regression, selection, and mortality, do not affect the outcome of the experiments and thus do not threaten their internal validity. The only variable affecting the outcome is the independent variable and thus internal validity is achieved.

6.1.3 External Validity

The studies presented in this example use concrete scenarios, such as subject-observer relationships, an application server, a content management system, and an order and inventory management system. Results that are not generalizable are a thread to external validity. However, the studies do not focus on any aspects specific to those particular scenarios. For example, the application server example deals with aspects like network protocols and adding new features in general, and the subject-observer relationship example examines how to establish dynamic relationships in a programming environment that does not provide any specific constructs for doing so. Hence, the results of these studies also apply to other kinds of servers and dynamic relationships.

6.2 Changeability and Extensibility Studies

The studies described in the following sections compare the first proposed resonance-oriented software development approach, harmony-orientation, with object-oriented programming in terms of changeability, extensibility, and maintainability.

The study in section 6.2.1 compares changeability in harmony-oriented and object-oriented programming through the example of subject-observer relationships, and the study in section 6.2.2 compares changeability through the example of implementing processing chains of producers, consumers, and filters. The study presented in section 6.2.3 compares extensibility and maintainability.

6.2.1 Changeability: Relationships

Several object-oriented design patterns that facilitate dynamic relationships between objects, such as the Observer pattern, are proposed in [37]. The purpose of the Observer pattern is to realize a one-to-many dependency between objects, such that when one object changes its state, all other objects are notified.

Figure 6.1 illustrates the conceptual design of the Observer pattern. The participants of this pattern are classes called *Subject*, *Observer*, *ConcreteSubject*, and *ConcreteObserver*. The *Subject* class defines the interface (and thus a concrete and fixed protocol) for attaching and detaching instances of the *Observer* class. The *Observer* class defines the interface for updating observers that are notified, when the state of the subject changes. The *ConcreteSubject* and *ConcreteObserver* classes are concrete implementations of observers and subjects.

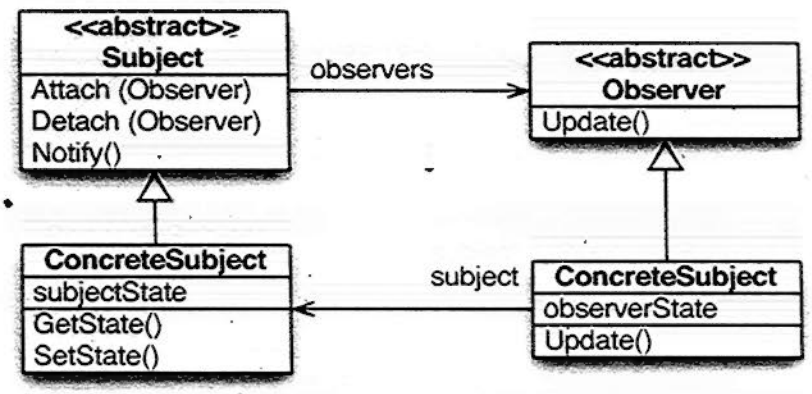

Figure 6.1: The Observer Design Pattern ([37])

To attach and detach observers to a subject, the *Attach* and *Detach* methods have to be invoked for each observer. These methods change an internal observer list that the subject instance maintains. Whenever the state of a subject changes, the following sequence of statements (protocol) is executed:

1. The subject invokes its *Notify* method.
2. The *Notify* method iterates through the subject's observer list and invokes the *Update* method of each observer.
3. The *Update* method of each observer invokes the *GetState* method of the subject and processes the new state.

In dynamically typed object-oriented programming languages like Ruby and Smalltalk, the abstract superclasses are not necessary. Also, the subject's state can be passed as a parameter of the observer's *Update* method, so that the observer does not have to invoke the subject's *GetState* method during the third step of the protocol described above.

Although the implementation of the Observer pattern is not very complex, its existence alone underlines the lack of mechanisms for defin-

ing relationships, other than static inheritance relationships, between objects in OOP. Furthermore, even though the Observer pattern facilitates a small degree of flexibility, the interfaces/protocols for registering, unregistering, and notifying observers have to be fixed during the design phase and later even small changes, such as adjusting the parameters of the *Update()* method, can result in a snowball effect that forces modification of many other objects. Another issue is that the Observer pattern requires the subject to be aware of its observers, to maintain a list, and to explicitly dispatch information to each of them whenever its state changes. However, the spirit of observation is that an observer should be able to observe a subject that is oblivious of being observed.

In harmony-oriented programming, code snippets interact with their space exclusively and are not aware of other snippets. However, programmers can set up subject-observer and other relationships through the spaciality principle: a relationship between two snippets can be established by moving them close to one another in the space, and broken off by moving them apart from one another. For example, consider a snippet S that produces numbers that are automatically diffused by the space. Since the diffusion process is not infinite, the numbers are only diffused inside a limited virtual area surrounding snippet S. Furthermore consider a second snippet O that observes numbers. If this snippet is far away from the number producing snippet S (i.e. outside the limited diffusion area), it does not process the generated numbers. However, as soon as snippet O is moved into the area containing the diffused numbers, it starts processing them and, as a result, a subject-observer relationship is established. If snippet O is moved outside the area again, the subject-observer relationship is broken off.

This study considers an implementation of a subject-observer relation-

ship between a subject that maintains a bank account state and an observer that is interested in being notified whenever the balance of the account changes. The following sections present an object-oriented implementation, a harmony-oriented implementation, and a comparison of the two implementations.

Object-Oriented Implementation

The subject is an instance of a class called *AccountSubject*, the observer is an instance of a class called *AccountObserver*, and the state itself is maintained by an instance of a class called *Account* that provides methods related to managing the account's balance, such as deposit and withdraw methods. Whenever the subject changes the state of the account, the observer is notified and provided with the *Account* instance.

Listings 6.1 and 6.2 show the methods of the *AccountSubject* and *AccountObserver* classes. As shown in listing 6.1, the object-oriented implementation maintains its observers in an ordered collection. Listing 6.3 contains a Smalltalk script setting up instances of *AccountSubject* and *AccountObserver*, and changing the state of the subject multiple times.

Harmony-Oriented Implementation

The harmony-oriented implementation consists of a single space and two snippets called "account subject" and "account observer" snippets. The "account subject" snippet reuses the *Account* class from the object-oriented implementation to realize its state. As explained in section 4.2.1, the state of a snippet is owned and diffused by the space, just like all other data in harmony-oriented programs. The implementation of the "account subject" snippet is a list of statements that perform the following actions:

```
1  initialize
2     observers := OrderedCollection new.
3     account := Account new.
4  attach: anObserver
5     observers add: anObserver.
6  detach: anObserver
7     observers remove: anObserver.
8  notify
9     observers do:[:observer| observer update: account.].
10 balance: aNumber.
11    account balance: aNumber.
12    self notify.
13 withdraw: aNumber
14    account withdraw: aNumber.
15    self notify.
16 deposit: aNumber
17    account deposit: aNumber.
18    self notify.
```

Listing 6.1: Methods of *AccountSubject* class (OOP).

- Change the type of the snippet state to *Account*.
- Use the state to change the balance (deposit, withdraw, etc).

The implementation of the "account observer" snippet observes the space and processes any *Account* objects that are diffused to its location. Figure 6.2 illustrates the harmony-oriented implementation of the account subject-observer relationship and listings 6.4 and 6.5 show the code of the subject and observer snippets.

In the harmony-oriented program shown in figure 6.2, the account subject-observer relationship is established already, since the substance

```
1 update: anAccount
2     Transcript cr; show: 'Observer : ',
3         anAccount balance asString.
```

Listing 6.2: Methods of *AccountObserver* class (OOP).

```
1 subject := AccountSubject new.
2 subject attach: AccountObserver new;
3     balance:100;
4     deposit: 50.5;
5     withdraw: 20;
6     detachAll.
```

Listing 6.3: Account subject and observer example (OOP).

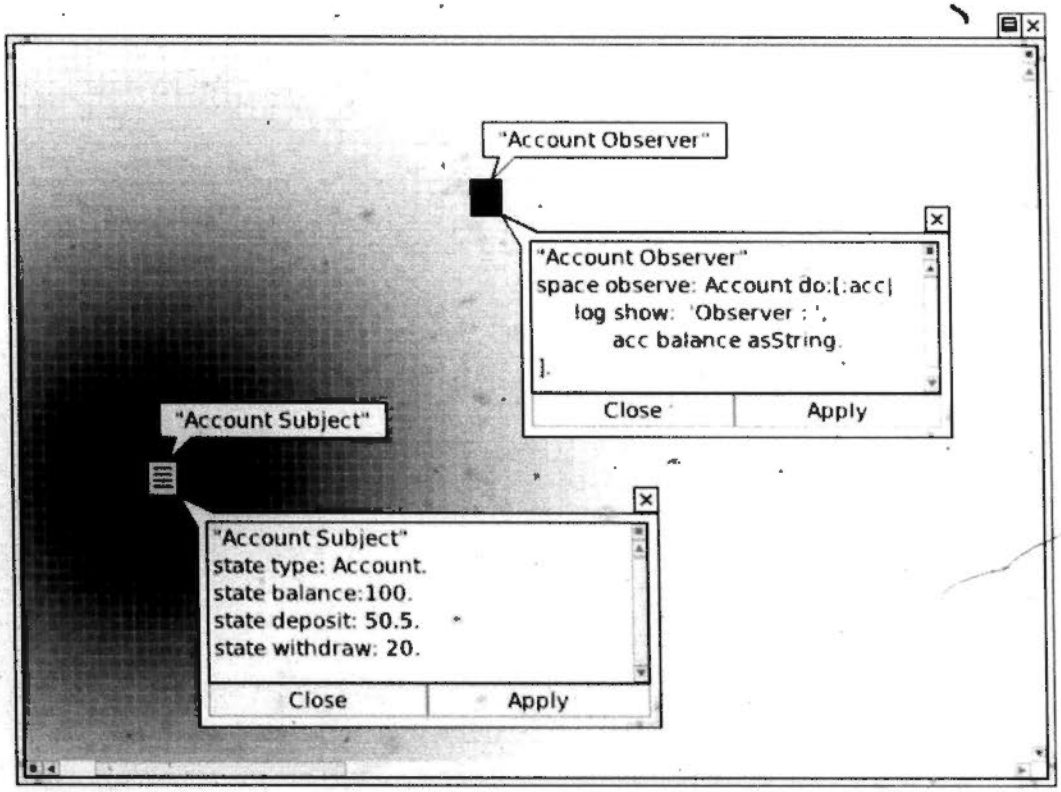

Figure 6.2: Account Subject and Account Observer (HOP)

```
1 "Account Subject"
2 state type: Account;
3       balance:100;
4       deposit: 50.5;
5       withdraw: 20.
```

Listing 6.4: Account subject snippet (HOP).

```
1 "Account Observer"
2 space observe: Account do:[:acc|
3     log show: 'Observer : ', acc balance asString.
4 ].
```

Listing 6.5: Account observer snippet (HOP).

diffusing the state of the account subject snippet reaches the account observer snippet. Moving the two snippets further apart from each other results in breaking off the subject-observer relationship.

Comparison

As the implementations in listings 6.4 and 6.5 show, it is not necessary to explicitly implement support for subject-observer relationships in harmony-oriented programs. It is enough to define two snippets: a subject snippet whose state is of type *Account* and an observer snippet that consumes data of type *Account*.

In the object-oriented implementation, however, support for the subject-observer relationship has to be implemented explicitly. Apart from defining the *AccountSubject* and *AccountObserver* classes, the programmer has to:

- Define nine methods (eight methods in *AccountSubject* and one method in *AccountObserver*).

- Implement the methods (containing a total of 18 message sends).

The minimal implementation overhead for supporting subject-observer relationships in object-oriented programs is as follows:

- Subject class:
  - Create a list for maintaining observers.
  - Provide method for attaching observers.
  - Provide method for detaching observers.
  - Provide method for notifying observers.
  - Add code to invoke notification method after any code that changes the state.
- Observer class:
  - Create method processing updated state.

The process of establishing and breaking off subject-observer relationships, once support for them has been implemented, is simple in both implementations. The difference is that relationships are established and broken off by moving snippets in the harmony-oriented version, and programmatically in the object-oriented version. The programmer can make changes to the relationship during runtime in the harmony-oriented version, but has to stop, edit, and then restart the object-oriented version.

6.2.2 Changeability: Processing Chains

This study provides a comparative example for creating and changing processing chains (like producer, consumer, and filter chains) in

harmony-oriented and object-oriented programs. It considers the following example processing chain:

1. A producer producing numbers between 0.2 and 10.0.
2. A filter that negates numbers.
3. A filter that rounds numbers down.
4. A consumer consuming numbers.

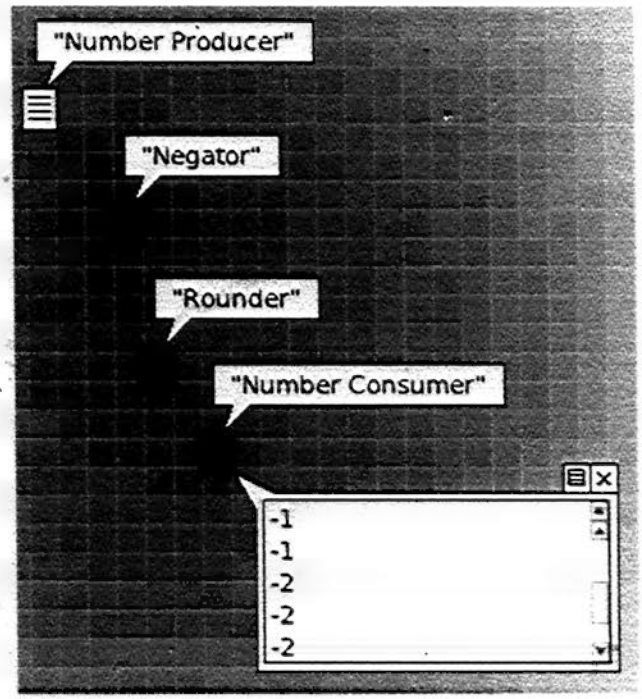

Figure 6.3: HOP Filter Chain

Figure 6.3 shows a harmony-oriented program implementing the example. The processing chain elements are implemented by four snippets whose code is shown in listings 6.6, 6.7, 6.8 and 6.9.

The two filters are implemented as snippets that consume numbers, perform an operation on them, and then put them back into the space. None of them contains any code for establishing a processing chain.

```
1 "Number Producer"
2 1 to: 50 do: [:idx|
3   / space put: (idx / 5) asFloat.
4 ].
```

Listing 6.6: Snippet producing numbers.

```
1 "Number Consumer"
2 space consume: Number do:[:num|
3     log show: num asString.
4 ].
```

Listing 6.7: Snippet consuming numbers.

```
1 "Negator"
2 space consume: Number do:[:num|
3     space put: num negated.
4 ].
```

Listing 6.8: Snippet negating numbers.

```
1 "Rounder"
2 space consume: Number do:[:num|
3     space put: num rounded.
4 ].
```

Listing 6.9: Snippet rounding numbers.

The processing chain can be constructed and changed during runtime by moving the snippets around in the space.

To create the same processing chain in an object-oriented program, an interface for chaining objects and passing data from one to another has to be designed first. Object-oriented processing chains can be realized

by applying design patterns like chain of responsibility [96], intercepting filters [34], and composite filters [104]. Figure 6.4 shows a possible design that defines a superclass called *ChainLink* that can be used for creating chains of objects. In addition, the diagram includes four classes derived from *ChainLink* corresponding to the snippets of the harmony-oriented version.

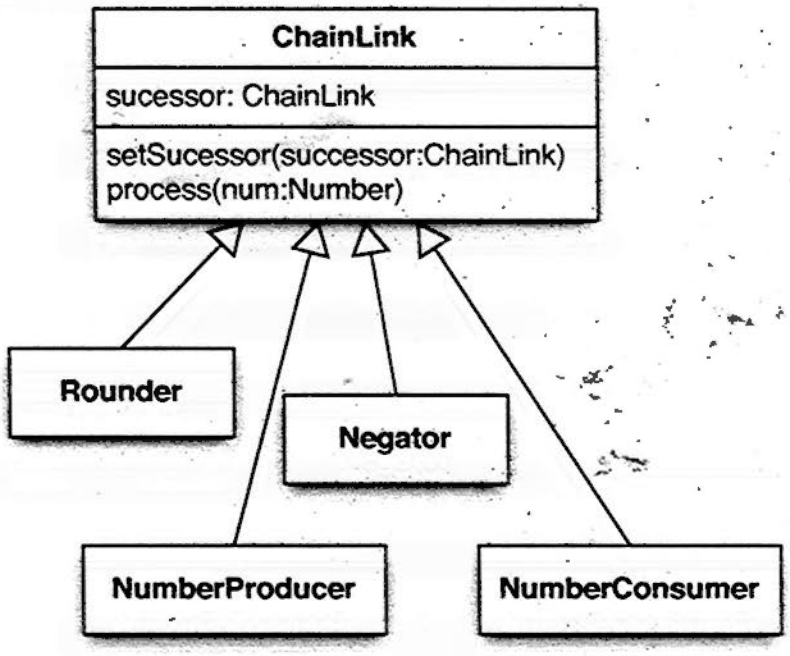

Figure 6.4: Object-Oriented Filter Chain

The *ChainLink* class provides the method *setSuccessor* for setting the next object (successor) in the chain. To build the example processing chain, instances of all four subclasses have to be created and then the following sequence of commands has to be executed:

1. Set *Rounder* instance as successor of *NumberProducer* instance.
2. Set *Negator* instance as successor of *Rounder* instance.
3. Set *NumberConsumer* instance as successor of *Negator* instance.

The *process* method shown in figure 6.4 is overridden by each of the four classes. This method has two responsibilities: process the received data and then, if the object has a successor, pass the processed data on.

6.2.3 Extensibility and Maintainability

The extensibility and maintainability study considers the example of an extensible application server (EAS) that receives requests from clients via a TCP socket and then passes these requests to registered applications, which process them and produce a replies. EAS is extensible in two ways: firstly, it is possible to add new protocols for interacting with clients, such as XML-RPC, SOAP and others. Secondly, it is possible to register and unregister applications during runtime.

Harmony-Oriented EAS

Figure 6.5 shows a possible harmony-oriented implementation of the EAS. This particular implementation contains two registered applications called "Bank Account Application" and "Counter Application", and supports the XML-RPC protocol for interacting with clients.

The snippets implementing the server are:

- *"Socket Reader"*
  A snippet that listens on a specified TCP port, creates sockets for incoming connections, and puts any data chunks received from these sockets into the space.
- *"XML-RPC → Action Request"*
  A snippet that consumes data chunks containing XML-RPC. The XML-RPC is converted into a protocol independent *ActionRequest* object, which is put into the space.

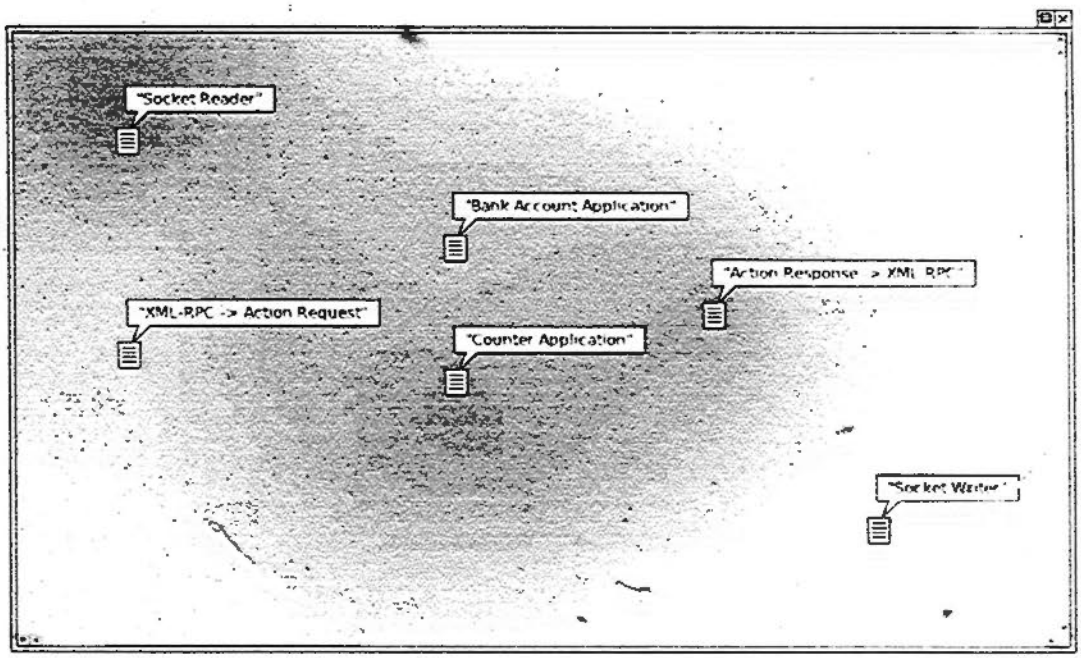

Figure 6.5: Harmony-Oriented Extensible Application Server

- *"Bank Account Application"* and *"Counter Application"*

  These two snippets represent registered applications. They observe the space and consume any *ActionRequest* objects matching the functionality they provide. After an *ActionRequest* has been consumed, the snippet performs the corresponding action, generates an *ActionResponse* object and puts it into the space.

- *"Action Response → XML-RPC"*

  A snippet that consumes *ActionResponse* objects converts them into a XML-RPC response string. The generated XML-RPC string is put into the space as a data chunk.

- *"Socket Writer"*

  A snippet that consumes data chunks and passes them to the client.

To add support for additional protocols, it is sufficient to implement

two additional snippets converting requests and responses to and from *ActionRequest* and *ActionResponse*. For example, to add support for the SOAP protocol, two snippets called "*SOAP → Action Request*" and "*Action Reply → SOAP*" have to be implemented and placed next to the "*XML-RPC → Action Request*" and "*Action Response → XML-RPC*" snippets in the space.

New applications can be added by creating a new snippet implementing the desired functionality and placing it in the center of the space, close to the other two "application" snippets. These applications can be unregistered without being shut down by moving them far away from the other snippets of the extensible application server.

Object-Oriented EAS

An object-oriented version of the of the extensible application server requires significant design before coding. In particular, the programmer has to design interfaces for:

- Implementing and deploying new protocol implementations.
- Implementing, registering, and unregistering new application.

One possible minimal object-oriented design is shown in figure 6.6. It defines a *Server* class and two abstract base classes for applications and protocols called *Application* and *Protocol*.

The *Server* class provides methods for registering and unregistering applications and protocols.

The *Protocol* class provides a method for checking wether a certain request string received by a socket is a valid request in the protocol it implements. In addition, the protocol class has two methods for

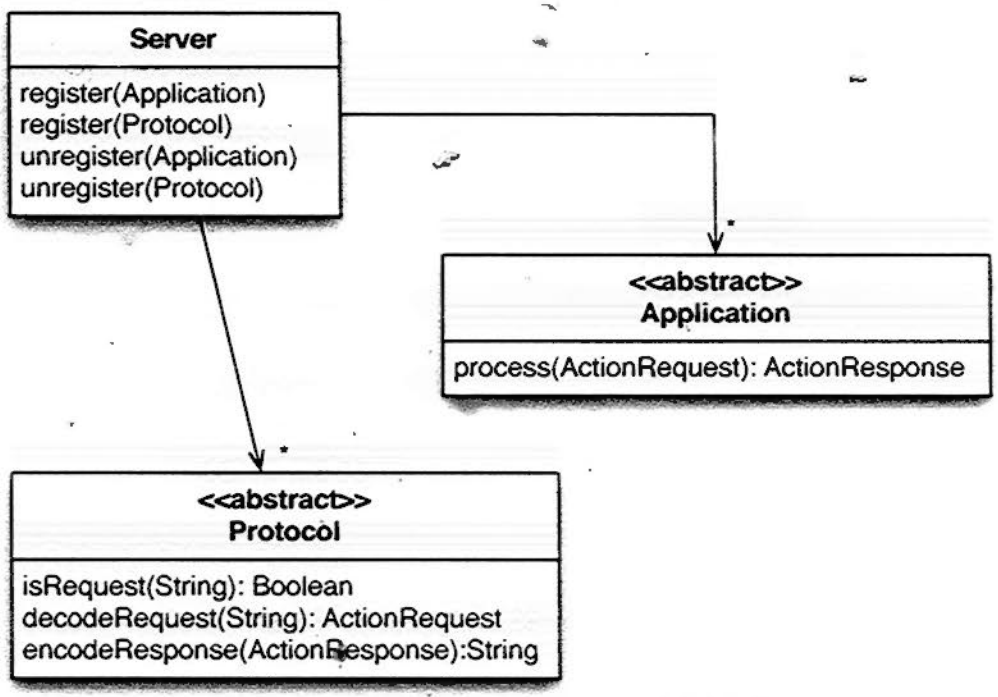

Figure 6.6: Minimal Object-Oriented EAS Design

encoding and decoding requests and responses into and from instances of protocol-independent *ActionRequest* and *ActionResponse* classes.

The *Application* class exposes a single method that takes an *ActionRequest* object as a parameter and returns an *ActionResponse* object.

Using this design, protocols and applications can be registered programmatically during startup. However, to support loading and registering applications and protocols during runtime, as it is possible in the harmony-oriented version, the object-oriented extensible application server has to provide a plugin mechanism that can be accessed via a client (like a Web browser) to load and register plugins.

Comparison: Dealing With Unpredicted Changes

The following paragraphs briefly examine the complexity of applying initially unexpected extensions to both versions of EAS. Lets consider

a scenario where EAS is updated to support applications that process continuous data streams, such as video or audio, and do not use a request-response model for interacting with their clients.

To support stream based EAS applications in the harmony-oriented version of the server, it is sufficient to update the *"Socket Reader"* snippet to add tags to data chunks that indicate which client they come from. A stream-based EAS application can then be implemented as a snippet consuming data chunks that are not consumed by the snippets processing protocol messages. After a chunk has been processed, it is put back into the space. The socket writer then receives the processed chunk and passes it back to the client. No major changes to snippets or data are required.

In the object-oriented version of the EAS, both logical and structural changes are required for supporting stream-based applications. Figure 6.7 illustrates a simplified version of the object-oriented EAS design including the *Server* class and the *Protocol* and *Application* interfaces.

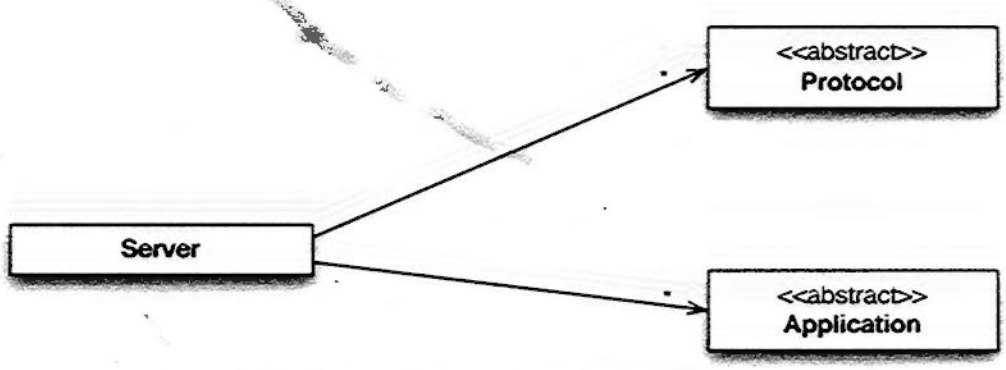

Figure 6.7: Object-Oriented EAS Design (Simplified)

The first structural change required for supporting stream-based applications is to rename the two interfaces to indicate that they are meant for applications that operate according to a message-based model, where applications generate responses for requests they receive. Figure 6.8

illustrates this adjustment: the interfaces *Protocol* and *Application* are renamed to *MessageProtocol* and *MessageApplication* respectively. Since the interfaces are renamed, all existing classes containing applications and protocols have to be updated to refer to the new interface names.

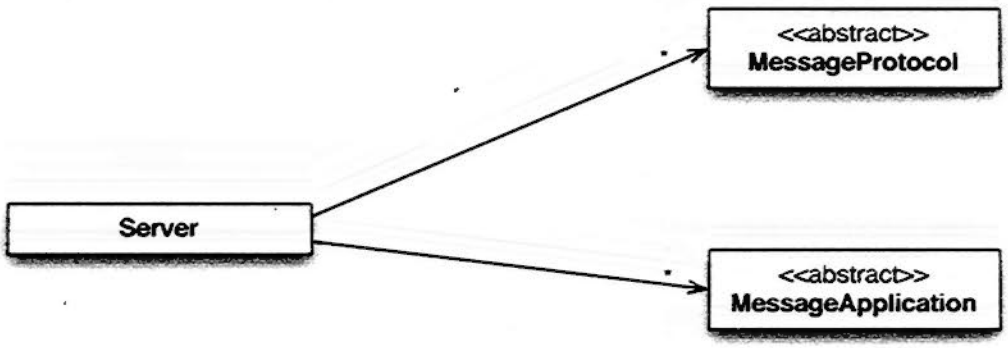

Figure 6.8: Change 1: Rename interfaces.

The second structural change is adding new, more general interfaces for protocols and applications, which encapsulate methods shared by both message-based and stream-based applications and protocols. These new interfaces and their relationship to the message-based interfaces are depicted in figure 6.9.

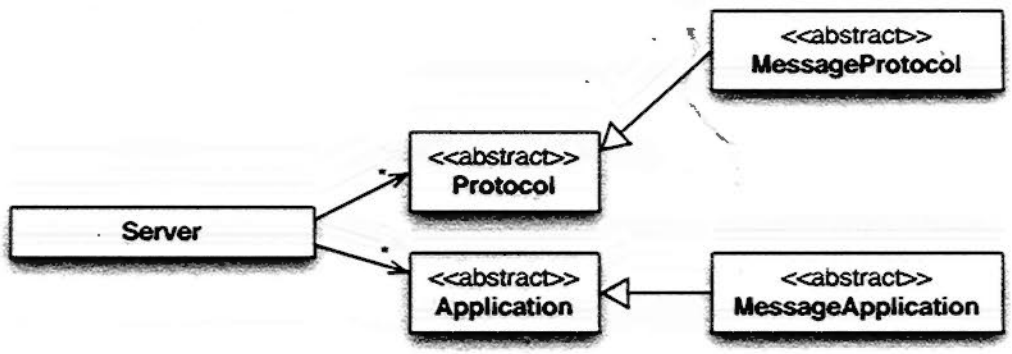

Figure 6.9: Change 2: Add base interfaces.

The final structural change, which is shown in figure 6.10, is adding two new interfaces for stream-based protocols and applications that

are derived from the general interfaces.

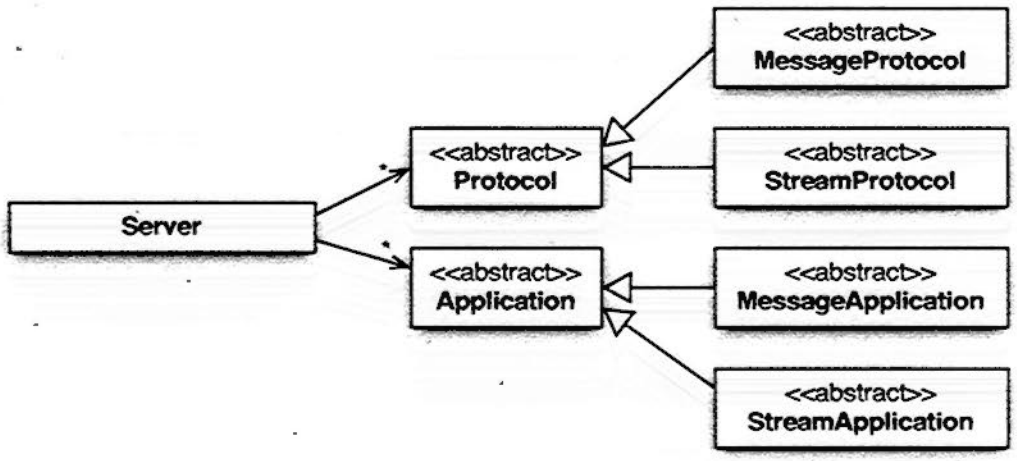

Figure 6.10: Change 3: Add stream-based interfaces.

In addition to the three changes described above, the interface of the *Server* class has to be adjusted to support registration of all types of applications and protocols. Moreover, the logic of the *Server* has to be changed to treat and process incoming data as a stream, if none of the registered *Protocol* classes can process it.

6.2.4 Analysis and Discussion of Validity

Sections 6.2.1, 6.2.2, and 6.2.3 provide evidence that, in comparison to traditional object oriented programming, the strengths of harmony-oriented programming are ease of changing the program's design, extensibility, and maintainability.

Table 6.1 summarizes how factors affecting software evolution are fulfilled by harmony-oriented programming.

Factor	Harmony-Oriented Programming
Ease of change	Easier than in OOP, because the structure of programs can be changed easily by moving snippets around.
Extensibility	Better than in OOP, because new snippets can be added at runtime, and existing snippets do not have to be changed.
Maintainability	Better than in OOP, because snippets do not have any direct dependencies on each other.
Quality feedback	Not available.
Error recovery	Not available.

Table 6.1: Harmony-Orientation and Software Evolution Factors

6.3 Error Feedback and Recovery Study

The following sections present a study comparing the second resonance-oriented software development approach, epi-aspects, to a traditional object-oriented application in regard to quality feedback and error recovery. The study consists of three parts that are based on a concrete application scenario: a Java-based content management system (CMS) for a logistics company whose staff frequently shares and distributes documents.

As shown in figure 6.11, this system consists of a HSQLDB database engine [86], an application server, and client applications that access this server through the XML-RPC protocol [103]. The application

server contains two major components called *Respository* and *Policies* that implement a document repository and access rules.

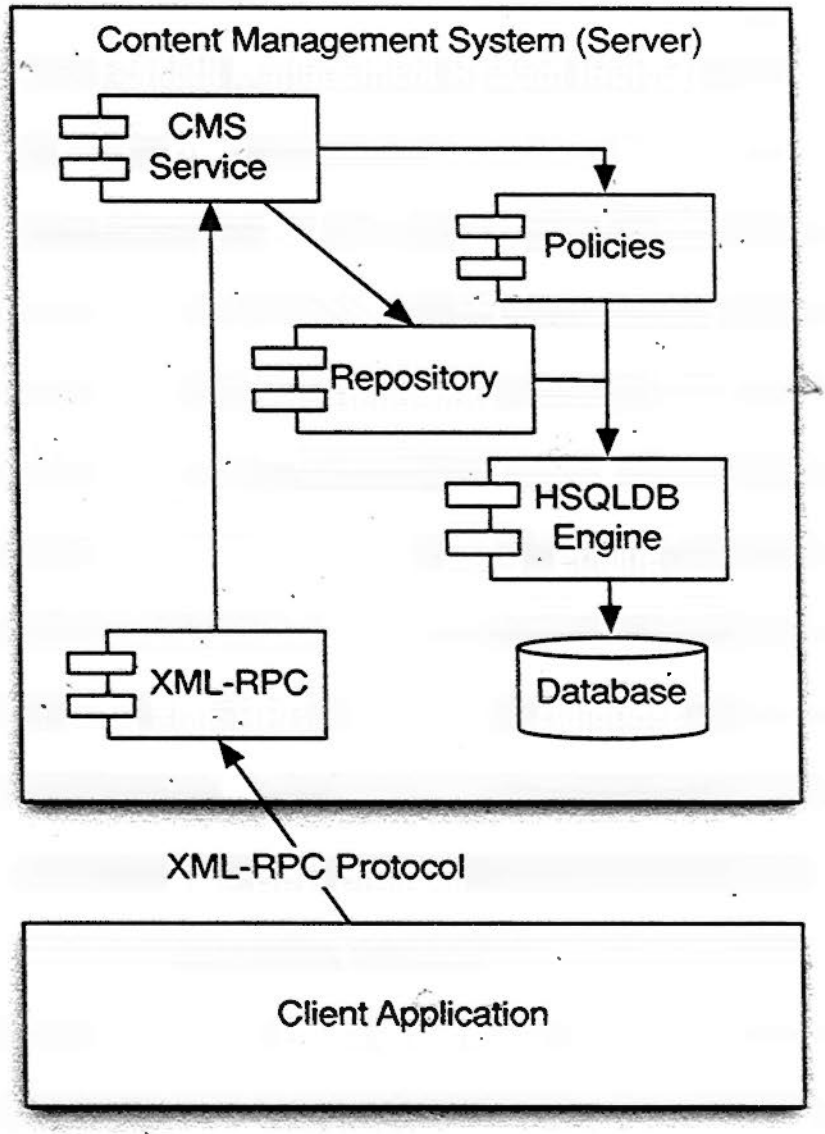

Figure 6.11: CMS Application Scenario

The staff of the company frequently request new features, and the system is continuously updated by a small team of developers. Also, as the properties of shared content and requirements regarding searching for and presenting content are changing over time, occasional modifications of the database are necessary. As a result, adjusting the application

server, database, and user interface of the client applications is common, and the system as a whole is constantly evolving. However, since the system is essential for the operation of the logistics company, long down times due to programming errors or maintenance operations are unacceptable. The CMS is developed and improved with the focus on features that are explicitly requested by staff of the logistics company, and mechanisms for self-maintenance and error recovery either receive a low priority or are omitted completely. As a result, the system is bound to become more fragile over time, and eventually a complete failure is possible.

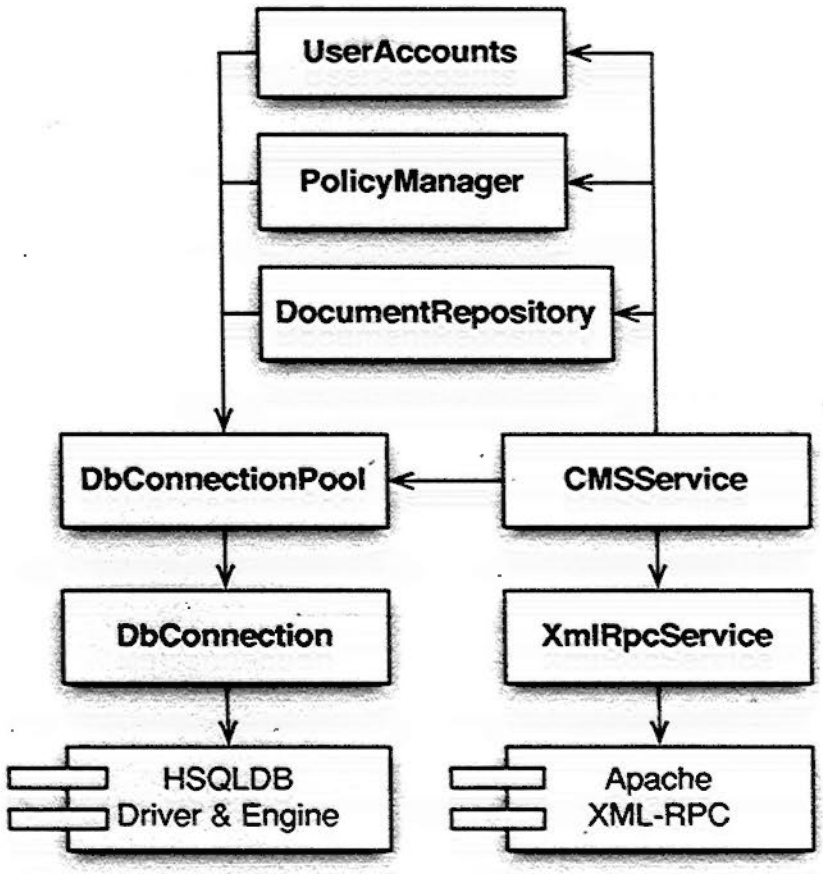

Figure 6.12: CMS Classes

Figure 6.12 shows a more detailed view of the content management sys-

tem's design, and highlights the main Java classes of the system: *XmlRpcService*, *DocumentRepository*, *CMSService*, *PolicyManager*, *DbConnectionPool*, *UserAccounts* and *DbConnection*.

The *XmlRpcService* class uses Apache's XML-RPC distribution [93] to initialize a HTTP server that accepts XML-RPC requests. This server uses reflection to map incoming XML-RPC requests to an instance of the *CMSService* class. Additionally, it converts return values provided by methods of the *CMSService* instance into XML-RPC responses. Even though Apache's XML-RPC distribution is mature and stable, these classes can generate critical exceptions in case of invalid requests and network problems.

The core of the CMS is implemented by the *DocumentRepository*, *PolicyManager*, and *UserAccounts* classes. The class *UserAccounts* implements user management and authentication. The CMS uses the HSQLDB database engine, and the database is accessed via the classes *DbConnectionPool* and *DbConnection*. The class *DbConnectionPool* maintains a pool of re-usable *DbConnection* instances, which provide access to the database via the JDBC driver supplied with the HSQLDB distribution.

6.3.1 Part 1: Conscientious CMS

The purpose of the first part of this case study is to use epi-aspects and the Epi-AJ framework to upgrade the CMS into conscientious software. The aim of this upgrade is to make the CMS observable and controllable by an autopoietic system. The following sections describe the implementation of four epi-aspects, which add necessary conscientious extensions to the CMS: software maintenance, XML-RPC monitoring, database monitoring, and CMS monitoring. This upgrade is

non-invasive, since it is unnecessary to modify the existing source of the CMS, the HSQLDB engine, and Apache's XML-RPC distribution.

Database Epi-Aspect

The database epi-aspect (figure 6.13) encapsulates functionality that allows the autopoietic system to observe and interfere with the operation of the HSQLDB database engine.

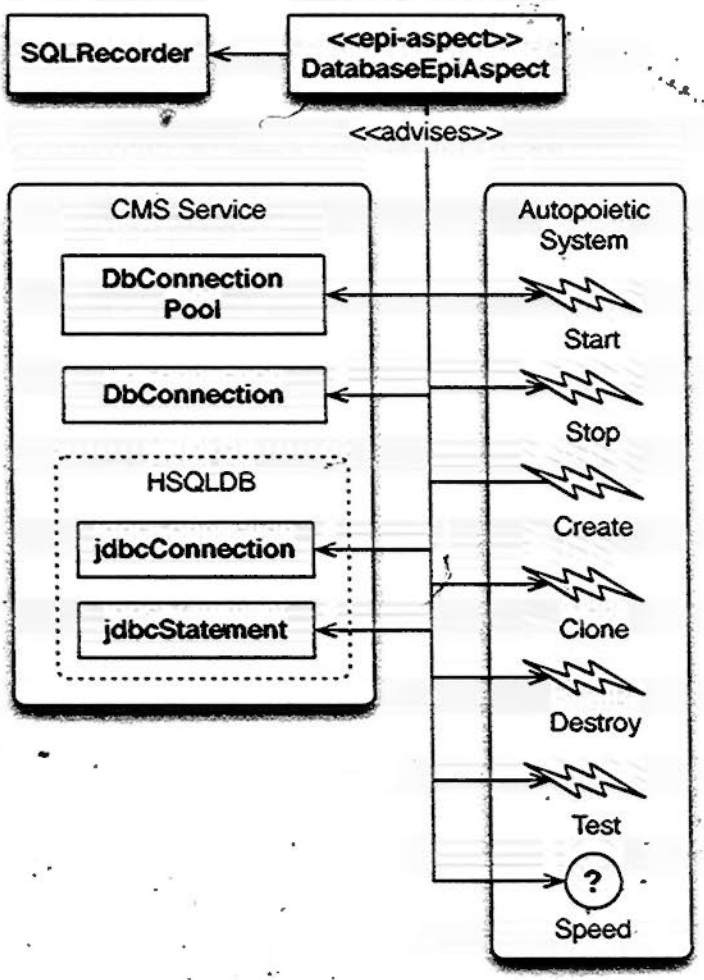

Figure 6.13: Database Epi-Aspect

In addition, the database epi-aspect implements database backup and recovery features. The backup feature periodically backs up the database files and allows restoring the database to a previous version.

It is useful for preventing problems related to data corruption. The database epi-aspect records all SQL commands that are issued to the HSQLDB engine from within the CMS. This recorded history can be used for undoing changes to the database and its schema.

XML-RPC Epi-Aspect

The CMS is accessed by clients via the XML-RPC protocol. The service providing this access is implemented by the *XmlRpcService* class, which utilizes Apache's XML-RPC distribution. It is imperative for the CMS that the XML-RPC service does not fail.

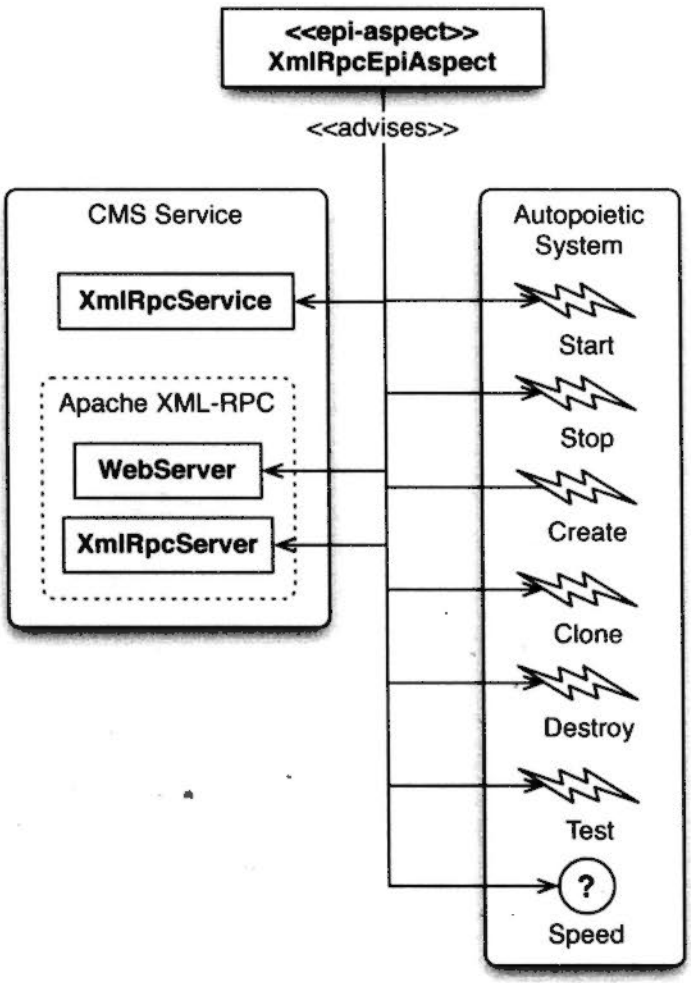

Figure 6.14: XML-RPC Epi-Aspect

The XML-RPC epi-aspect (figure 6.14) is responsible for implementing error recovery, testing and application monitoring concerns. It also implements an observer feature to evaluate and store the current speed of the XML-RPC service. The speed is defined as the time required to execute a dummy XML-RPC request.

Software Maintenance Epi-Aspect

The software maintenance epi-aspect (figure 6.15) implements functionality for updating and reverting the components of the CMS system. It provides advice for the autopoietic *Update* and *Revert* recommendations.

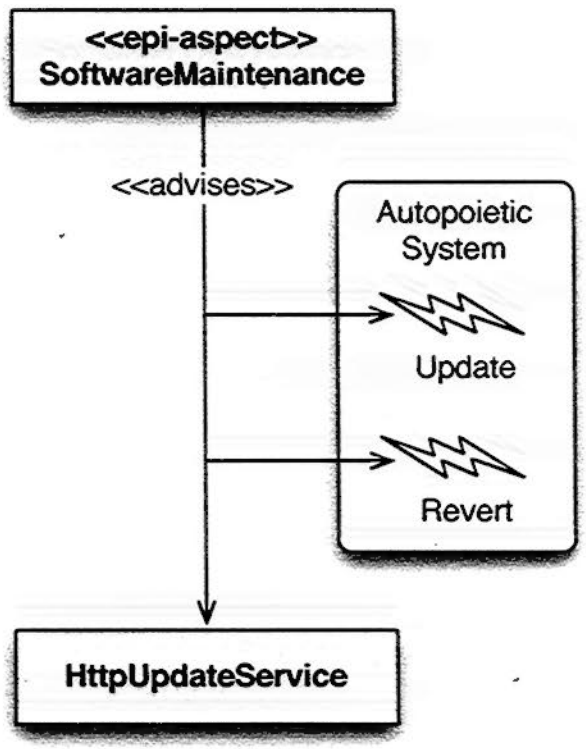

Figure 6.15: Software Maintenance Epi-Aspect

When the software maintenance epi-aspect is initialized, it creates a minimal HTTP service, which developers can use to submit software updates via a web-browser. Whenever the software maintenance epi-

aspect receives an update through the HTTP service, it does not immediately install the update, but stores it for later use, and dispatches an epi-message to notify the autopoietic system that an update is available. If the autopoietic system approves of the update, it first issues recommendations to affected components to prepare for an imminent update, and then issue the *Update* recommendation, which causes the software maintenance epi-aspect to install the update.

If the autopoietic system notices that certain components experience problems after an update, such as uneven performance, it can issue a *Revert* recommendation that indicates that the problematic component should be reverted to a previous version. The software maintenance epi-aspect implements an advice on the *Revert* recommendation that checks if a previous version of the affected component exists. If a previous version is available, the advice disables the current version, and re-installs the previous version.

CMS Epi-Aspect

The CMS epi-aspect is shown in figure 6.16. It is responsible for exposing the health of the main classes of the CMS, namely *UserAccounts*, *CMSService*, *DocumentRepository*, and *PolicyManager* to the autopoietic system. Additionally, the CMS epi-aspect extends these main classes with functionality to comply recommendations issued by the autopoietic system.

6.3.2 Part 2: Software Update Experiment

In this phase, working and buggy updates are applied to the original and conscientious versions of the CMS, and the behavior of the systems during and after the update is observed, compared, and evaluated.

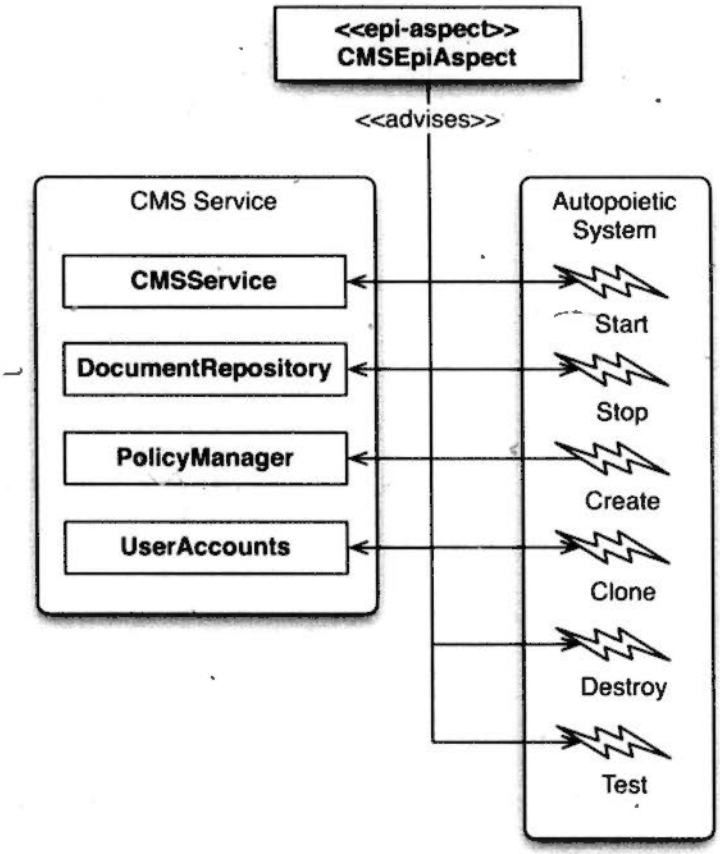

Figure 6.16: CMS Epi-Aspect

The experiment consists of two phases and each phase consists of two parts:

In the first part, the experiment is conducted with the original CMS, which does not make use of any epi-aspects. In the second part, the experiment is repeated using the conscientious version of the CMS and the autopoietic simulator. Then the results of both parts are compared and evaluated.

The experiment uses the original CMS, the conscientious CMS, the autopoietic simulator, and an additional simulator that mimics the behavior of a client application that accesses the CMS. This client application simulator can be configured to generate a specific number

of requests per minute, allowing adjustment of the workload that the CMS has to handle. The client application simulator uses a log file to record requests results and exceptions that occur when accessing the CMS.

Phase 1: Install Working Update

A working update of the *XmlRpcService* class is installed, and the necessary steps and time required for updating the original and conscientious CMS are compared. This phase is initialized by performing the following steps:

1. Start the original CMS.
2. Start the conscientious CMS and autopoietic simulator.
3. Start two instances of the client simulator and configure them to issue one request per second to the original and conscientious versions of the CMS. These instances are denoted as *client simulator 1*, which issues requests to the original CMS server, and *client simulator 2*, which issues requests to the conscientious CMS server.

After phase one has been initialized, the following steps are executed in sequence:

1. Manually overwrite the Java class file of the *XmlRpcService* class in the original CMS with the new version.
2. Shutdown and restart the original CMS.
3. Use the HTTP update service of the software maintenance epi-aspect to submit the source code of the updated *XmlRpcService* class to the conscientious version of the CMS.

The results of the first phase are shown in table 6.2. While the update is successful for both the original and conscientious versions of the CMS, the original CMS has to be restarted which makes the system unavailable for approximately 17 seconds. This short period causes a number of XML-RPC requests issued by the client simulator 1 to fail. The conscientious CMS, however, experiences no downtime, because the software update epi-aspect applies and initializes the updated version of the *XmlRpcService* class in the background and then immediate replaces the old instance with the new one. This result indicates that the conscientious CMS is more suitable for being updated during production use.

Original CMS	Update successful. Approx 17 seconds downtime, because the entire CMS system is restarted.
Client Simulator 1	Log file indicates 19 failed requests.
Conscientious CMS	Update successful. No downtime.
Client Simulator 2	Log file indicates no failed requests.

Table 6.2: Software Update Experiment Phase 1 Results

Phase 2: Install Buggy Update

In the second phase a buggy update of the *XmlRpcService* class is installed. The bug in this update is a latent bug that causes critical failure after the *XmlRpcService* instance has been running for approximately one hour. This phase uses the same steps as the first phase, except that the updated version of the *XmlRpcService* class contains a latent critical bug that starts causing failures after approximately one hour. After the updates are applied to both versions of the CMS, their

behavior is observed for three hours.

Experiment Time (Min)	Event
61:23	First exception in the *XmlRpcService* instance the original CMS.
61:27	First exception in the *XmlRpcService* instance the conscientious CMS.
61:27	The autopoietic simulator recommends to restart the *XmlRpcService* instance of the conscientious CMS, which is done by the XML-RPC service epi-aspect.
61:29	The *XmlRpcService* instance of the original CMS terminates. Original CMS not accessible.
125:14	Exception in the *XmlRpcService* instance the conscientious CMS. Autopoietic simulator recommends reverting the *XmlRpcService* to a previous version, which is done by the software maintenance epi-aspect.
180:00	Conscientious CMS is still running properly.

Table 6.3: Software Update Experiment Phase 2 Log

Table 6.3 shows the observation log of the second phase. As the observation log indicates, the XML-RPC service of the original CMS fails after approximately 61 minutes. The conscientious CMS restarts the XML-RPC service after it causes an initial exception in the 61st minute of the experiment. The restart prevents further exceptions for approximately one hour. When exceptions start occurring again in the 125th minute of the experiment, the conscientious CMS reverts the XML-

RPC service to its previous version and continues running smoothly until the end of the experiment.

6.3.3 Part 3: Fine-Grained Error Monitoring

The third part of this study illustrates extending the conscientious CMS with a finer-grained error monitoring and recovery mechanism. This extension involves extending the CMS epi-aspect as well as writing new rules for the autopoietic simulator. Here, we describe the design and implementation, and the approach for testing the extended conscientious CMS.

The finer-grained error monitoring and recovery mechanism maintains a history of exceptions for the each of the classes *UserAccounts*, *CMSService*, *DocumentRepository*, and *PolicyManager*. The functionality for the exception history is implemented by a component that is added to the CMS epi-aspect. Whenever an exception occurs in one of the CMS's main classes, this component makes an entry into a log file associated with the class that threw the exception. Apart from creating the entries, the component counts the number of exceptions in pre-defined intervals. The number of exceptions per interval is recorded and submitted the numbers of exceptions for the five most recent intervals are submitted to the autopoietic system via the epi-queue.

The autopoietic simulator requires additional rules for processing the epi-messages containing the exception counts of the five previous intervals:

1. If the number of exceptions for the current interval is greater than zero, issue a *Test* suggestion. The corresponding advice in the CMS epi-aspect verifies the responsiveness of the CMS

system classes. If the test fails, the default rules of the autopoietic simulator trigger a *Restart* suggestion.

2. If the number of exceptions between the four previous and current interval have increased more than a pre-defined threshold, issue a *Restart* suggestion.

To test the proper operation of the implementation of the fine-grained monitoring feature, updated versions of the *CMSService* and *DocumentRepository* classes, which randomly throw non-critical and critical exceptions, are added to the conscientious CMS. Non-critical exceptions do not affect the proper operation of instances of these two classes, and critical exceptions lead to a crash of the CMS. The test is run according to the following protocol:

1. The updated CMS epi-aspect, autopoietic rules, new versions of *CMSService* and *DocumentRepository* are deployed in the conscientious CMS.
2. The component in the CMS epi-aspect is configured to use intervals of three minutes.
3. The autopoietic simulator is configured to use 10 exceptions as the threshold value for triggering *Restart* suggestion.
4. One instance of the client simulator is started with the same configuration used in the software update experiment described in section 6.3.2.
5. Let the experiment run for 120 minutes.

This test can only fail if there is an implementation error in the CMS epi-aspect. After the test is run successfully, the development of the fine-grained monitoring feature is completed.

6.3.4 Analysis and Summary

The previous sections demonstrate how to use the epi-aspects architecture for software maintenance, quality feedback, and error recovery in a content management system for a logistics company.

Factor	Epi-Aspects Architecture
Ease of change	No improvement. Same as in plain OO application.
Extensibility	No improvement. Same as in plain OO application.
Maintainability	Better than in plain OO application, as epi-aspects can be used to facilitate upgrades and downgrades.
Quality feedback	Better than in plain OO application, as quality feedback is provided through epi-aspects.
Error recovery	Better than in plain OO application, as error recovery is provided through the autopoietic system and epi-aspects.

Table 6.4: Epi-Aspects and Software Evolution Factors

The experiment conducted during the second part of the study provides strong evidence, that epi-aspects are sufficient to keep real software systems running smoothly and to ease the burden of software evolution in relation to maintenance, quality feedback, and error recovery. In particular, the experiment shows that in comparison with a plain Java application, an aspect-oriented conscientious application adapts better to changes and problems. Table 6.4 summarizes how factors affecting software evolution are fulfilled by the epi-aspects architecture.

6.4 Software Evolution Study

Sections 6.2 and 6.3 show that harmony-orientation and epi-aspects have different strengths in relation to the hypothesis. These are ease of changing the program's design, extensibility, and maintainability for harmony-orientation, and maintainability, quality feedback, and error recovery for epi-aspects.

This study uses a combination of harmony-orientation and epi-aspects called *harmony-oriented epi-aspects* to evaluate resonance-oriented software development in the context of the evolutionary stages of an actual software development project that was conducted by a German company.

Section 6.4.1 briefly introduces the combined harmony-oriented epi-aspects architecture and its implementation in HOS, and section 6.4.2 presents the details of the study.

6.4.1 Harmony-Oriented Epi-Aspects

The combined harmony-oriented epi-aspects architecture is illustrated in figure 6.17. It consists of harmony-oriented spaces (as described in section 4.2), an autopoietic evaluator based on declarative rules (similar to the autopoietic system introduced in section 5.1.2), and a modified version of epi-aspects denoted as *spatial epi-aspects*. Harmony-oriented spaces and spatial constructs are used for implementing allopoietic programs.

To support spatial epi-aspects, the harmony-oriented space is supplemented with a mechanism that allows aspects to advise on events related to snippets, data, and diffusion. These events include:

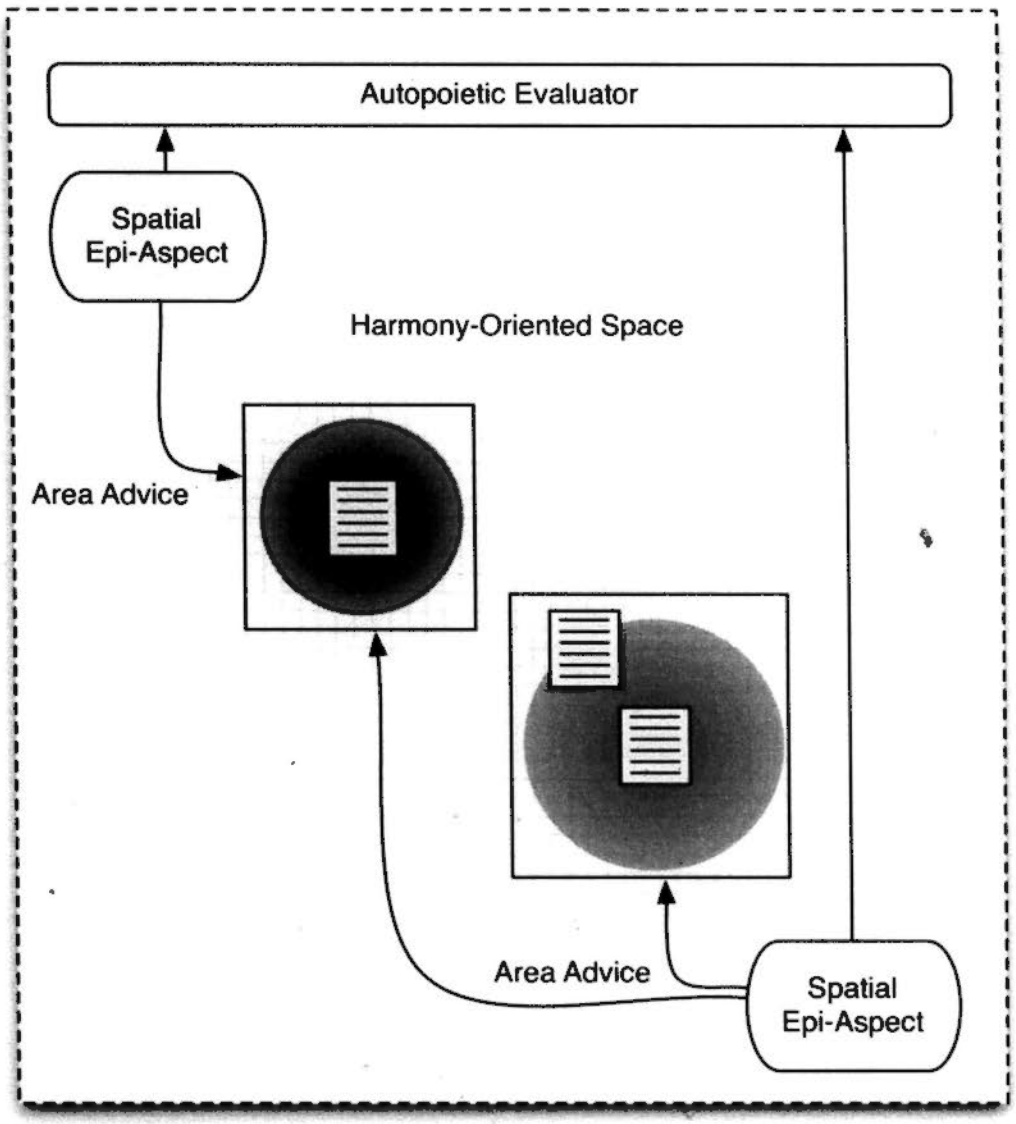

Figure 6.17: Harmony-Oriented Epi-Aspects

- Creation of snippets and other spatial constructs.
- Destruction / termination of spatial constructs.
- Change of a spatial construct's location.
- Occurrence of errors within spatial constructs.
- A spatial construct putting data into the space.

- A spatial construct consuming data into the space.
- Arrival of data in a location via diffusion.

The following paragraphs provide detailed descriptions of spatial epi-aspects, the autopoietic evaluator, and their implementation in HOS.

Autopoietic Evaluator

The autopoietic evaluator implements rules for keeping the system running smoothly. Spatial epi-aspects forward events that occur in the space to the autopoietic evaluator for processing. While processing, the autopoietic evaluator issues autopoietic recommendations and queries to either instruct the spatial epi-aspects what to do or to request more information.

While the autopoietic evaluator supports the same recommendations as the autopoietic system of the epi-aspects architecture (see table 5.1), it defines new, harmony-oriented versions of autopoietic queries. These are:

- *Reveal Location Data*
  This query indicates that the autopoietic system wishes to obtain the data currently available in a specific location in a specific space.
- *Examine Construct Activity*
  This query indicates that the autopoietic system wishes to receive information on construct activity. For example, information on when the spatial construct consumed data from the space for the last time, and how much and what kind of data the construct produces.

Spatial Epi-Aspects

Like the aspects proposed in the epi-aspects architecture, spatial epi-aspects implement advice on autopoietic recommendations and queries.

However, spatial epi-aspects do not advise on join points in a application's code (application advice), but on events within one or more specified virtual areas in one or more harmony-oriented spaces. For example, the spatial epi-aspect in the bottom right corner of figure 6.17 advises on two areas (marked by rectangles) in a harmony-oriented space. The spatial epi-aspect in the upper left corner only advises on one area. This kind of advice, which is defined on areas of a harmony-oriented space, is called *area advice*.

Additionally, spatial epi-aspects do not use epi-messages to communicate with the autopoietic evaluator, but simply return an object or a hierarchy of objects from advice on autopoietic queries.

HOS Implementation

The HOS implementation of harmony-oriented epi-aspects extends spaces to generate *space events*, when spatial constructs and data changes, and to provide a mechanism for spatial epi-aspects to advise on such events. Space events are realized as instances of the class *SpaceEvent* and its various subclasses (figure 6.18). The *SpaceEvent* class provides instance variables to store information on:

- The space in which the event occurred.
- The exact location in the space where the event occurred.
- If applicable, the construct involved in the event.

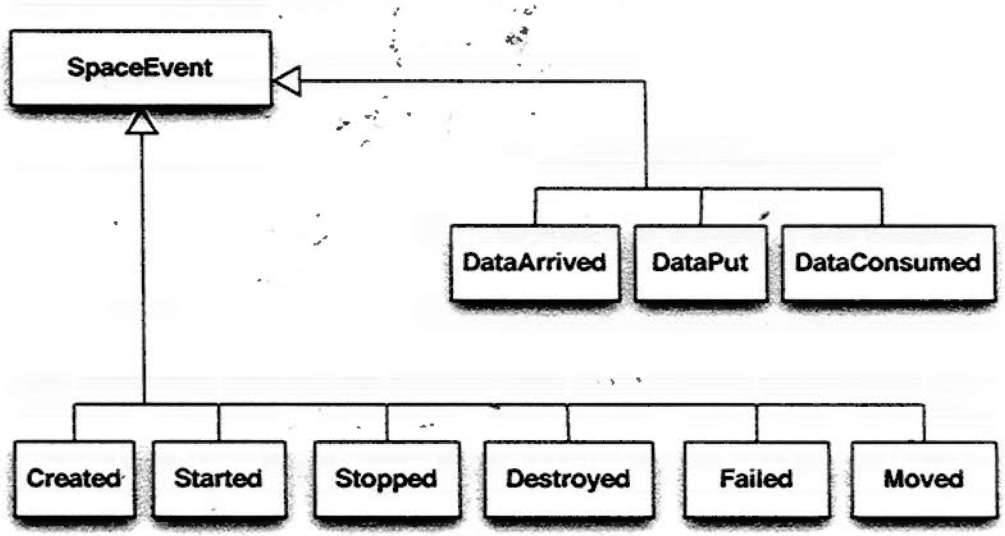

Figure 6.18: Space Events

Additional information can be stored in the various subclasses of the *SpaceEvent*.

The autopoietic evaluator is implemented by a singleton called *AutopoieticEvaluator*, which allows defining autopoietic rules via Smalltalk code blocks and associations. The autopoietic rules have the following format:

```
[Condition Block] -> [Action Block]
```

or

```
[Condition Block] -> {Recommendations}
```

Both blocks take an instance of a *SpaceEvent* class (or subclass) as parameter. For example, a rule for recommending to restart a snippet that failed because of an error looks like:

```
[:event |event isKindOf:Failed] -> {'Stop'.'Start'}
```

After the autopoietic evaluator has come to a decision regarding a space event, it issues an autopoietic recommendation or query accompanied

by a *RQTarget* object containing attributes like *space*, *location*, and *construct*. The information in this object helps advice in spatial epi-aspects to determine at which location or spatial construct the query or recommendation is directed.

Spatial epi-aspects are defined by creating instances of the class *SpatialEpiAspect*. As shown in table 6.5, this class provides methods for defining area advice on spaces and advice on autopoietic recommendations and queries.

name:*aName*	Specifies the name of the spatial epi-aspect.
defineAdvice:*block* on:*evts* in:*space* at:*areas*	Specifies an area advice on one or more areas in a specific space.
defineAdvice:*block* onRecommendation:*rec*	Specifies an advice on an autopoietic recommendation.
defineAdvice:*block* onQuery:*query*	Specifies an advice on an autopoietic query.

Table 6.5: Methods of SpatialEpiAspect

The *block* argument of the advice definition methods shown in table 6.5 contains the code implementing the advice. The code blocks passed to these methods takes one parameter. When the advice is invoked, this parameter is an instance of one of the *SpaceEvent* classes, if the advice is an area advice, or a *RQTarget* object, if the advice is on an autopoietic recommendation or query.

Listing 6.10 provides a sample definition of a spatial epi-aspect.

```
1  SpatialEpiAspect new name: 'Exampe';
2      defineAdvice: [:evt |
3          AutopoieticEvaluator evaluate: evt.
4      ] on: Failed in: space at: (20@20 extent:50@50);
5      defineAdvice:[:target |
6          target.construct.stop.
7      ] onRecommendation: 'stop';
8      defineAdvice:[:target |
9          target.construct start.
10     ] onRecommendation: 'start';
11     defineAdvice:[:target |
12         target.space.dataAt: (target location).
13     ] onQuery: 'reveal-location'.
```

Listing 6.10: A Spatial Epi-Aspect.

6.4.2 Study Description

The software evolution study is based on a real-world software development project within a German company called "Großhandel für Modernes Antiquariat - Dunker & Nellissen GmbH" (GMA) specializing in wholesale of antique books and books with minor defects[1]. The project, which was conducted in the years 1998 to 2001, was the development of an inventory and order management system optimized for the special nature of the company's business. It serves as a good example for a constantly changing and evolving software that eventually became brittle and was discarded.

The software evolution study repeats the development process of the GMA inventory and order management system using the harmony-oriented epi-aspects architecture. This repeated development process

[1]Permission to disclose details of the project in this thesis has been obtained.

is not a complete harmony-oriented re-implementation of the entire system, but rather a simulation of the changes and failures that occurred during the evolution of the system.

The following paragraphs first provide more background on the nature of the company's business and the history of the inventory and order management system's evolution, and then present a detailed description of the study simulating a repetition of the evolution using the harmony-oriented epi-aspects architecture.

Company Background

GMA is specialized in selling antique books and books with slight defects to book shops throughout Germany. Antique books are acquired in small volumes from various sources and defective books are bought from publishing houses in large volumes. GMA sells books to other businesses, such as bookshops and resellers, and does have any retail outlets. The company has a small IT department that is responsible for developing customized software in-house and maintaining the company's web site. Staff of the company is encouraged by the management to request new software features, if they feel the software is lacking in some aspect.

GMA Software Evolution

The GMA inventory and order management system started in 1998 as a database-centric application with an application server developed in Java and client programs in C++ that accessed the server through a custom remote procedure call (RPC) protocol. The original database design of the GMA software is shown in appendix C. The implementation of the application server provides model classes corresponding to the entities defined in the database design. Figure 6.19 provides a

simplified overview of the application server's most important model classes and their relationships.

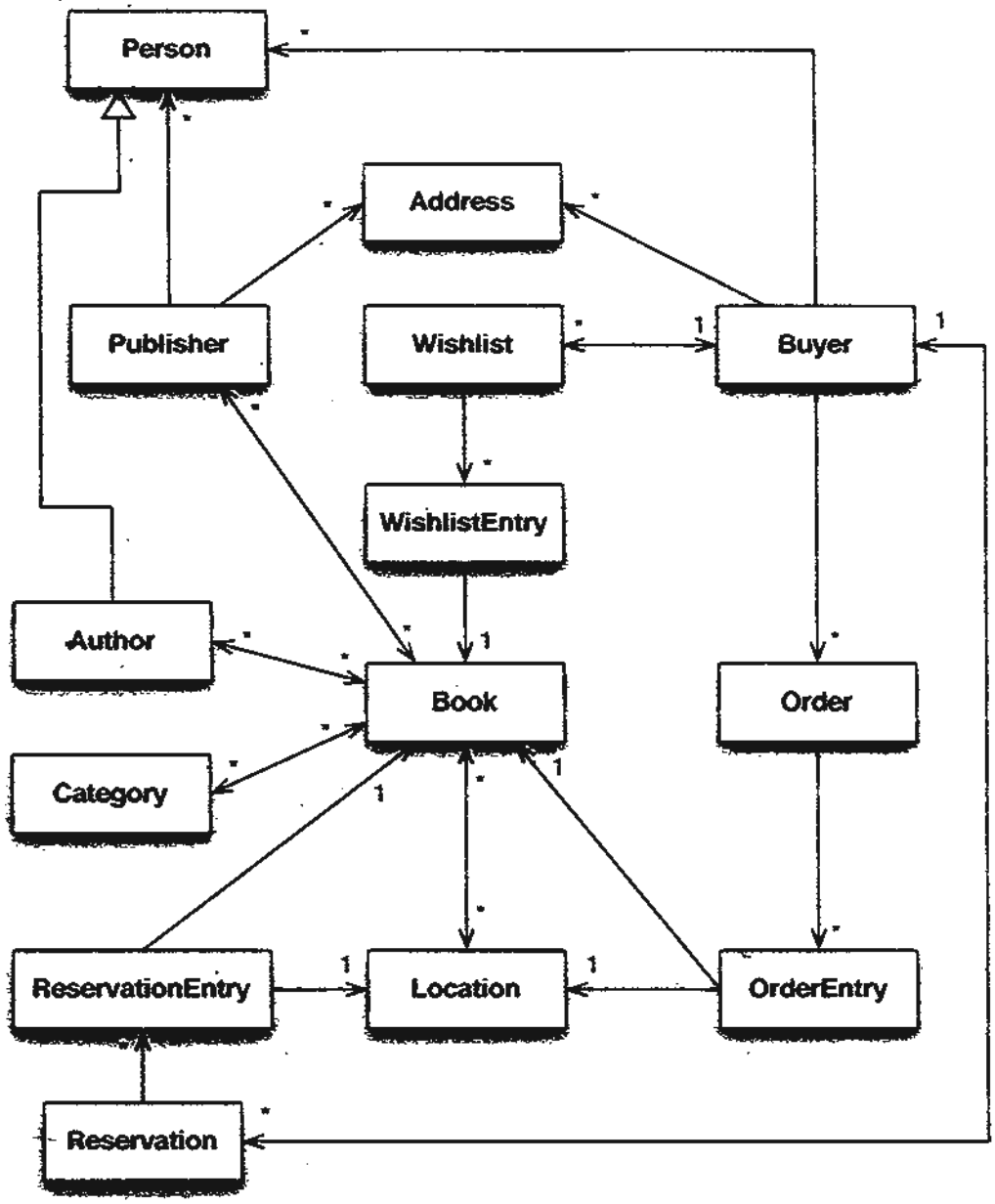

Figure 6.19: GMA Application Model (Simplified)

The following list highlights the major stages, including changes, challenges, and failures, the GMA inventory and order management system went through during its evolution:

1. *Unique identifiers for books*

   The original design of the GMA application proposed using ISBNs as a book's unique identity. After the first version of the software was implemented, it was used without problems for a while. The majority of books passing through the warehouses of the company are modern books with minor defects that all have ISBN numbers. When antique books, which usually don't have ISBN numbers, were entered into the system, the staff assigned unique numbers by themselves.

   However, at some point it turned out that ISBN numbers are not truly unique, as some publishing houses re-use ISBN numbers from books that went out of print. As a result, the management decided that all books should be tagged with an additional unique identifier, and the software was changed so that *Book* classes support an additional property. Apart from changing the application model and database design and queries, the user interface had to be adjusted. The changes were applied by the developers without any major problems, but the system had to be shutdown for extended periods of time, and thus there was a negative impact on the business of the company.

2. *Initially unpredicted data queries*

   During the initial design of the inventory and order management application, the staff of GMA was asked to help designing frequently used queries, such as searching for books by author, title, ISBN number. The application server and C++ client applications were implemented to support a set of pre-defined queries based on the suggestions of the staff. However, after the initial version of the GMA software was deployed, it became apparent that the pre-defined set of queries was not sufficient to cover all

situations staff encountered during work. As a result, staff frequently requested support for new queries. In the beginning, the requests were entertained by the developers changing the application server and user interface of the C++ client applications. However, as the requests for adding new or adjusting existing queries became more frequent, the management decided to change the GMA software to allow staff to define and run custom queries.

Adding support for creating custom queries involved changes to the user interface of the C++ client applications, and implementation of set of new classes in the application server for managing, interpreting and running custom queries.

3. *Plug-in support*

Apart from queries on the application model, the staff of GMA frequently requested new import and export features. For example, VLB, a association of German book publishers maintains a list of all published and currently available German books. This list updated and distributed via CD-ROM quarterly. The staff of GMA wished to be able to import this VLB list and also book lists from other, foreign, associations. Since the various lists were distributed in different file formats, and it was not clear at that time what other export and import functions might be required in the future, it was decided to upgrade the GMA system with a plug-in mechanism.

The requirements for the plug-in mechanism were that it allows loading and unloading plug-ins while the application server is running, and that it exposes the entire application model to plug-ins. Since the application sever was written in Java, the plug-in mechanism was designed to load and initialize JAVA archives during runtime. Each plug-in was designed to contain import or export

logic and a HTML-based user interface description that could be rendered by the C++ client applications. The development and deployment of the plug-in mechanism required significant changes and took two developers a total of 7 months to complete.

The first two plug-ins developed for the GMA application were an import plug-in for the VLB list and an export plug-in that generates barcodes for the ISBN numbers of books.

4. *Authentication support*

   The initial design of the GMA inventory and order management application did not consider user authentication and access policies, since the company's computers were initially located in a restricted area only accessible by authorized staff. However, the management decided to set up computers in other locations, such as the warehouse to facilitate more convenient inventory management. The computers in the warehouse were equipped with bar code readers that could be used by staff to update in inventory faster.

   Since fine-grained authentication and access policy support would have required significant changes to the design of the application's model, it was decided that once a user was authenticated, he or she could use any function provided by the GMA application. Authentication was done against a list of staff user names and passwords stored in the application server.

5. *Support for articles other than books*

   At some point the management of GMA decided to trade articles other than books, such as calendars, paintings, and small merchandise based on characters in popular books and comic books.

   To support such articles, the model of the inventory and order

management system had to be changed again. A model class called *Article* was introduced and classes implementing the properties of the various items, such as books and paintings, were derived from this class. Apart from the application model, the database model was updated to reflect the changes. The updated database and application server were deployed during night time, but on the next working day, data entry did not work properly, and some data was lost. The original database and application server were restored and the new versions were redeployed again after a few days of finding and fixing bugs.

6. *Upgrade of plug-in mechanism*

   The changes to the model classes of the GMA inventory and order management system resulted in required changes to the plug-in mechanism, which was developed to expose the original application model, where books were the only type of article. Apart from upgrading the plug-in mechanism, parts of the two plug-ins in use, the VLB list importer and barcode generator, had to be rewritten.

7. *Upgrade of custom query mechanism and existing queries*

   The changes of the model classes also affected the custom query mechanism. It was upgraded to support searching for articles other than books. The format for defining custom queries was changed and so were the classes in the application server for managing, interpreting, and running the custom queries. Slight adjustments were also made to the user interface of the C++ client applications.

   Since the format for defining custom queries was altered, all existing custom queries defined by staff of the company had to be

rewritten using the new format.

8. *Data corruption discovery and system downgrade*
   Several weeks after stages 5 to 7 were completed, staff of the company noticed that some custom queries did not work as expected. For example, when searching for books, other articles, such as calenders were included in the search results. An investigation discovered that slight mistakes had been made when changing the database to the new application model and as a result, various entries had been corrupted. For example, some books were classified as other articles and vice versa. To control damage, the management of the company decided to restore the database and application server to versions that were backed up before stages 5 to 7 were implemented and deployed. The outdated data was updated by hiring people to help conducting a company-wide inventory.

9. *Decision to develop a new application from scratch*
   Shortly after the downgrade, the management of the company decided not to add any more features to the existing version of the GMA inventory and order management system, and to begin development on a new application from scratch. It was decided that this new application should be a web-based application that can be accessed by staff via browsers instead of C++ clients.

Evolution With Harmony-Oriented Epi-Aspects

The following paragraphs describe a repetition of the evolution process of the GMA inventory and order management system with the harmony-oriented epi-aspects architecture. This study is not meant to be considered as a direct comparison to the implementation of the

GMA application, which used Java and C++, two languages whose programs are more inflexible and less maintainable than programs written in Smalltalk, the language HOS is based on. Rather, this study explores what the development and evolution process would be like, if a harmony-oriented epi-aspects architecture was used to implement the GMA application. The focus is on harmony-orientation and epi-aspects features, but not on the programming languages themselves. The study describes a harmony-oriented design of the GMA application and explores necessary changes and results for the stages the GMA system went through during its evolution. It focuses on the application server only and does not consider client programs and their user interfaces.

To avoid creating an unfair advantage for the simulated harmony-oriented epi-aspects version of the GMA application, the model classes from the initial design previously introduced in figure 6.19 are re-used. When designing a new harmony-oriented application from scratch, a different application model with more loosely coupled abstract data types would likely be chosen.

Figure 6.20 shows the initial design of the GMA application server using harmony-oriented epi-aspects. Apart from a group of snippets implementing communication with clients ("RPC" snippets), the harmony-oriented space contains snippets managing the data (model) of the GMA application ("data management" snippets), and snippets implementing pre-defined queries ("query" snippets). The "RPC" snippets group consists of snippets almost identical to the snippets used by the extensible application server used in the study described in section 6.2.3:

- *"Socket Reader"*

  A snippet that listens on a specified TCP port, creates sockets for

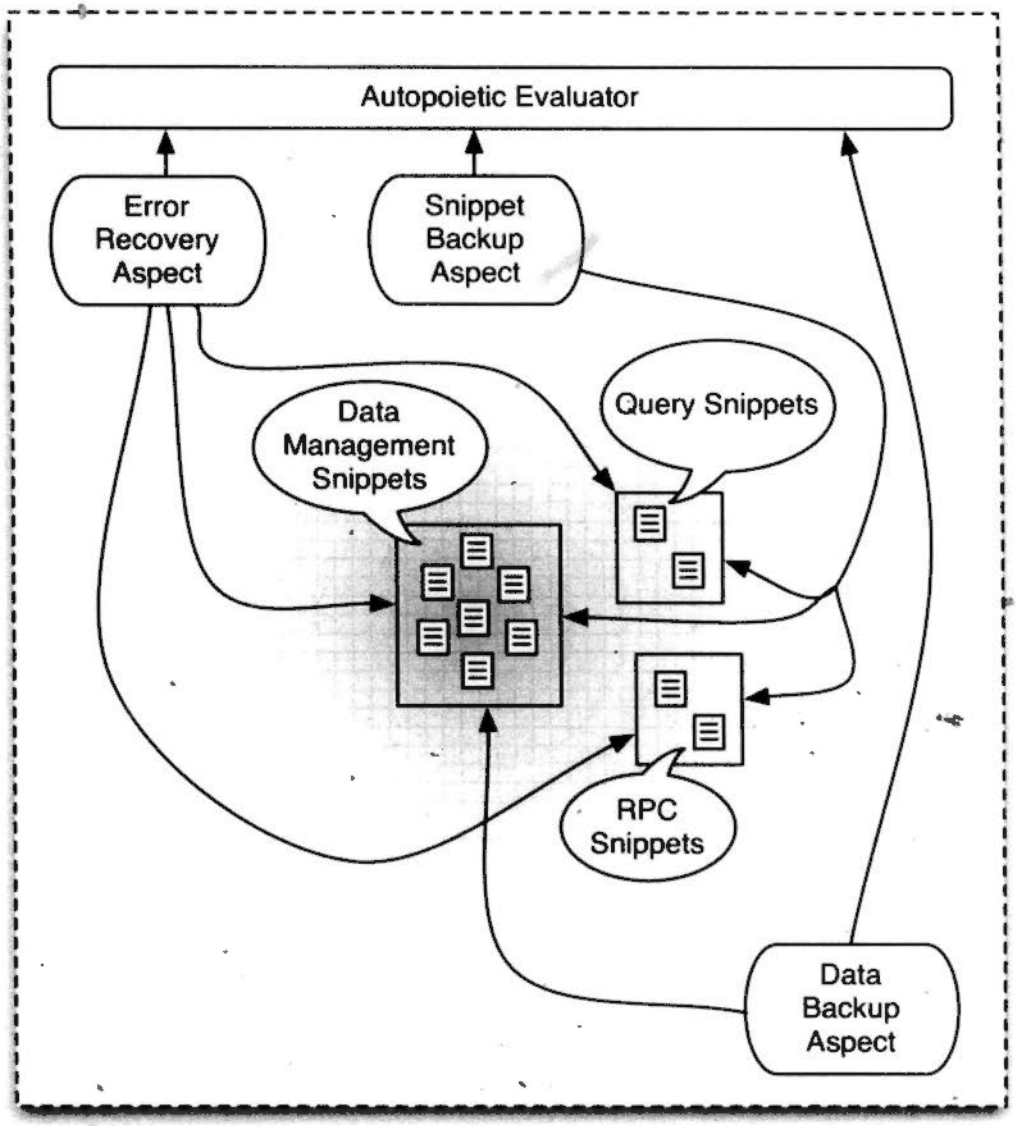

Figure 6.20: GMA Server Using Harmony-Oriented Epi-Aspects

incoming connections, and puts any data chunks received from these sockets into the space.

- *"Custom RPC → Command"*
  A snippet that consumes data chunks containing custom RPC instructions. These are converted into a protocol independent *Command* object, which is put into the space.

- *"Result → Custom RPC"*
  A snippet that consumes *Result* objects, converts them into a binary custom RPC response strings, and puts into the space as data chunks.

- *"Socket Writer"*
  A snippet that consumes data chunks and passes them to the client.

The "data management" snippets group consists of a set of snippets that manage and change the data (model) of the GMA application based on commands received from clients. Each of the snippets in this group is responsible for one type of model objects. For example, one snippet is responsible for managing *Book* objects and another snippet is responsible for managing *Publisher* objects. Each of these snippets maintains a state that contains a list of all model objects of the type it is responsible for. As these states are owned and diffused throughout the space, it is accessible by other snippets. The snippets belonging to the "data management" are named after the model objects they are responsible for:

- "Books" snippet.
- "Categories" snippet.
- "Authors" snippet.
- "Buyers" snippet.
- "Wishlists" snippet.
- "Reservations" snippet.
- "Publishers" snippet.

Apart from maintaining lists of the model objects, these snippets implement persistence. Figure 6.21 shows the area of the space containing the "data management" snippets group and an open location inspector that lists the contents (data) available to other snippets.

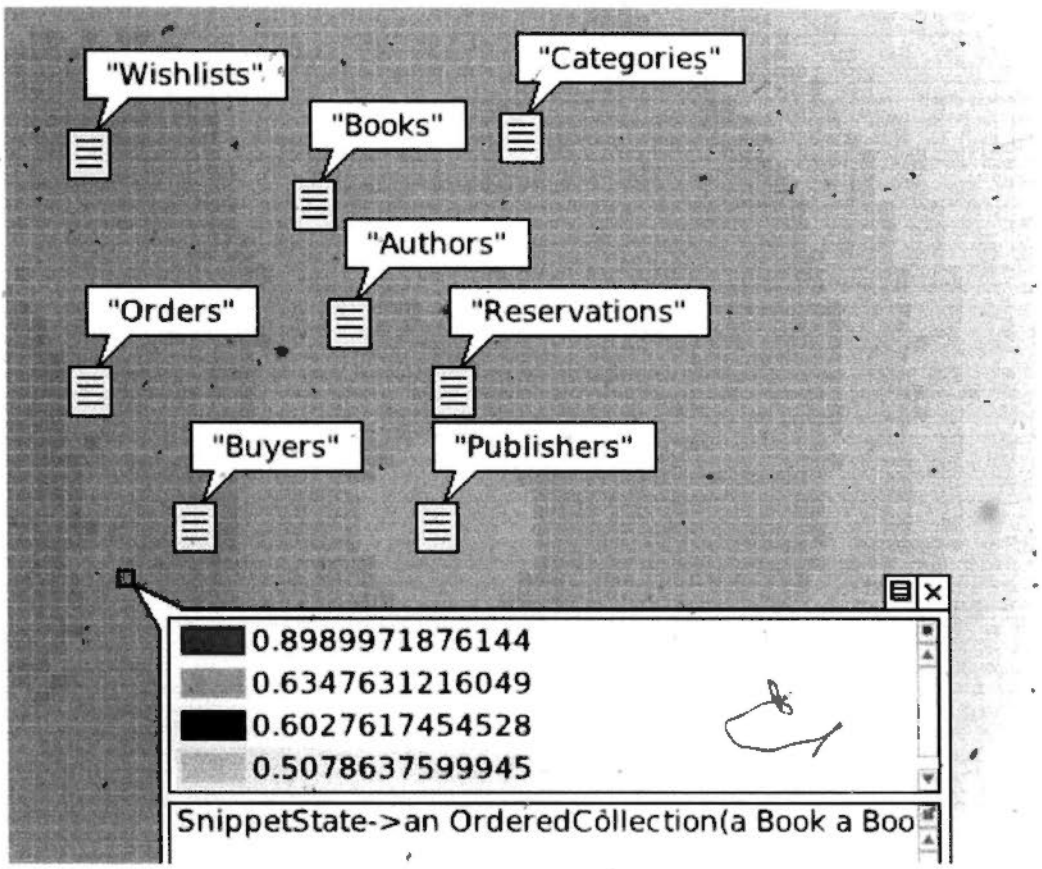

Figure 6.21: "Data Management" Snippets

Apart from the snippets groups, the initial design shown in figure 6.20 contains three spatial epi-aspects that advise on the areas occupied by the snippet groups: the "error recovery" aspect, the "snippet backup" aspect, and the "data backup" aspect. The "error recovery" aspect is responsible for informing the autopoietic evaluator about any errors that occur within snippets. In addition, it contains code to accommodate autopoietic recommendations, such as cloning and restarting snippets. The two backup spatial epi-aspects are responsible for back-

ing up data and code and reverting to older versions, if recommended by the autopoietic evaluator. For example, whenever a snippet is modified by a programmer, the "snippet backup" aspect makes a backup, and keeps all previous versions.

Evolution stage 1: Unique identifiers for books

Adding an additional attribute to the *Book* class and assigning unique identifiers to all existing *Book* objects can be achieved by following these steps while the server is running:

- Stop the "books" snippet. No data is lost, because it is located in the space and not in the snippet.
- Add an additional attribute called *UID* and corresponding getter and setter methods to the *Book* class.[2]
- Change the code in the "books" snippet to iterate through the list of books stored in its location and update each object by assigning a value to the new *UID* attribute.
- Restore the previous code in the "books" snippet and change it to assign unique identifiers to each newly created instance of the *Book* class.

Evolution stage 2: Initially unpredicted data queries

New data queries can be added by programmers by creating new "query" snippets and placing them next to the existing ones. Since this process is trivial, that development of a custom query feature for staff of the company appears unnecessary.

[2]Changing classes during runtime is only possible in dynamic languages like Smalltalk [59], Ruby [90], and Python [65].

Evolution stage 3: Plugin support

Since all data is diffused in the harmony-oriented space, it is not necessary to explicitly implement support for plug-ins. It is sufficient for programmers to add new snippets.

For example, to add the VLB import feature to the server, it is sufficient to create a new snippet in the vicinity of the "data management" snippet group. This snippet can read the information from the VLB CD-ROM, generate corresponding commands for creating the relevant data, and place it into the space. Once the commands reach the various snippets of the "data management" group via diffusion, the snippets process the commands and add the imported data to their lists.

Evolution stage 4: Authentication Support

Authentication support can be added to the server by adding an attribute to *Command* objects that indicates whether the command was sent by an authenticated user. The authentication can either be performed by a modified version of the "Custom RPC → Command" snippet or an additional snippet placed into its vicinity. Additionally, one or more "authentication filter" snippets have to be placed between the "RPC" snippets and other groups to automatically consume (and discard) any *Command* objects whose attributes indicates that they have not been authenticated.

Evolution stage 5: Support for articles other than books

During this stage, the object-oriented version of the GMA introduced an *Article* superclass and derived concrete articles, such as books and paintings, from it. In the harmony-oriented epi-aspects version of the server, new classes for articles are created without defining an inheri-

tance relationship between them. For each newly created class a corresponding "data management" snippet is created. These are:

- "Calendars" snippet.
- "Merchandise" snippet.
- "Artworks" snippet.

Evolution stage 6: Upgrade of plug-in mechanism

Since the harmony-oriented epi-aspects version of the server does not have an explicit plug-in mechanism only the plug-ins themselves have to be slightly modified to process instances the new article classes in addition to books.

Evolution stage 7:
Upgrade of custom query mechanism and existing queries

The harmony-oriented epi-aspects version of the server does not have a custom query mechanism and existing queries do not have to be changed, because the existing model classes have not been changed. However, new queries have to be defined to support searching for articles other than books.

Evolution stage 8: Data corruption discovery and system downgrade

The harmony-oriented epi-aspects architecture does not prevent programmers from making mistakes and introducing bugs. As a result, even though some of the evolution stages require less effort, data corruption or other kinds of failures of the GMA server at some point are likely. The three spatial epi-aspects introduced as part of the initial design implement quality feedback and error recovery features that can

help to keep a buggy system running and might give developers enough room to fix critical bugs without having to shut down the system for extended periods of time.

6.4.3 Analysis and Summary

The software evolution study shows how the evolution of the GMA inventory and order management system might have progressed, if a harmony-oriented epi-aspects architecture had been used. This study illustrates that, during certain stages of the evolution, the harmony-oriented epi-aspects version of the GMA server is easier to change, extend, and maintain, and that it is possible to use the architecture to prepare for and adapt to failures by implementing quality feedback and error recovery mechanisms. However, the architecture cannot prevent programmers from introducing critical bugs into the system, so failure is still possible. Table 6.6 summarizes how factors affecting software evolution are fulfilled by the harmony-oriented epi-aspects architecture.

6.5 Hypothesis Validation

The results of the studies conducted in this chapter and their relation to factors affecting software evolution summarized in tables 6.1, 6.4, and 6.6 support the hypothesis formulated in section 1.5 : That, in comparison to object-oriented programming, resonance-oriented software development improves the ease of dealing with the above mentioned issues, and thus the ease of dealing with software evolution effectively. The hypothesis posits that the proposed resonance-oriented software design and development approaches provide the following advantages over traditional object-oriented programming:

Factor	Combined Approach: Harmony-Oriented Epi-Aspects
Ease of change	Easier than in OOP, because the structure of programs can be changed easily by moving snippets around.
Extensibility	Better than in OOP, because new snippets can be added at runtime, and existing snippets do not have to be changed.
Maintainability	Better than in OOP, because snippets do not have any direct dependencies on each other. Also, spatial epi-aspects can be used to facilitate upgrades and downgrades.
Quality feedback	Better than in OOP, as quality feedback is provided through spatial epi-aspects.
Error recovery	Better than in OOP, as error recovery is provided through the autopoietic evaluator and spatial epi-aspects.

Table 6.6: Combined Approach and Software Evolution Factors

1. Fewer changes are required in order to reflect adjustments of a program's design in the code. Changes include source code modifications and other adjustments to a program.
2. Extending a program requires less effort (steps/changes).
3. Implementation of reliable feedback and error recovery mechanisms requires fewer steps.

The following sections summarize how the studies in this chapter support these claims.

6.5.1 Evidence Supporting Claim 1

"Compared to traditional object-oriented programming, fewer changes are required in resonance-oriented programs to reflect adjustments of a program's design in the code."

The first changeability study regarding subject-observer relationships in section 6.2.1 shows that, in traditional object-oriented programming, establishing and breaking off subject-observer relationships between objects requires the explicit implementation of a registration and notification mechanism. In the concrete Smalltalk-based example given in section 6.2.1 implementation of this mechanism requires definition of eight methods containing a total of 18 message sends. In harmony-oriented programming however, it is not necessary to implement such a mechanism, since the information diffusion principle allows establishing and breaking off subject-observer relationships by moving snippets around.

The second changeability study (section 6.2.2) examining processing chains of producers, consumers, and filters is another example supporting the claim that fewer changes are required in resonance-oriented programming when changing a program's design. In the example used in the study, members of a processing chain are set up by deriving classes from a superclass defining methods and attributes to specify the next objects in a processing chain. Changes in the super class results in changes to all classes used for establishing processing chains. Also, each class participating in a processing chain has to manually forward data to the next object. As a result, there is a tight coupling between

classes, which makes changes more complex. In the harmony-oriented version, it is not necessary to explicitly implement a mechanism for creating and operating a processing chain.

Further evidence supporting claim 1 is provided by the software evolution study using the harmony-oriented epi-aspects architecture described in section 6.4.2. Especially the software evolution stages where plug-in and authentication support are added underline how small, in comparison to traditional object-oriented programming, the impact of fundamental changes to a program's design is in a resonance-oriented software development approach. Both stages require significant changes in the object-oriented version . For example, in the object-oriented version only a limited form of authentication was implemented to avoid changes throughout the application. In the harmony-oriented version, however, authentication can be implemented by adding authentication filter snippets at various locations to enforce access policies, and the majority of existing snippets does not have to be changed.

6.5.2 Evidence Supporting Claim 2

"Compared to traditional object-oriented programming, extending a program requires less effort (steps/changes) when using resonance-oriented software development approaches."

Section 6.2.3 presents a study comparing extensibility and maintainability in harmony-orientation and object-orientation. This study uses the example of EAS, an extensible application server, that can be extended with new applications and network protocols. To support such extensibility, the object-oriented version has to implement a mechanism for registering and unregistering new protocols. This mechanism defines strict interfaces applications and protocols have to adhere to. The

study shows that extensions that do not fit these fixed interfaces require significant changes in the extension mechanism of the object-oriented EAS version. It further shows that the harmony-oriented version of the EAS does not require explicit implementation of an extension mechanism, and that it can be easily adjusted to support initially unexpected types of applications and protocols. As a result, significantly less effort is required for extending the harmony-oriented version of the EAS.

Claim 2 is also supported by the software evolution study using the harmony-oriented epi-aspects architecture (6.4.2). The following evolution stages in particular show that extensions to (and maintenance of) the harmony-oriented epi-aspects version require fewer changes:

- *Stage 2: Initially unpredicted data queries*
  This stage shows that the harmony-oriented epi-aspects version can be extended with new queries by adding new snippets during runtime. However, adding new queries during runtime is not possible in the object-oriented version. As a result, a complex mechanism for defining and managing custom queries is added to the object-oriented version.

- *Stage 3: Plug-in support*
  To support plug-ins a complex plug-in mechanism that allows adding and changing plug-ins during runtime has to be implemented in the object-oriented version. In the harmony-oriented version, no plug-in mechanism is necessary. Plug-ins can be realized as conventional snippets.

- *Stage 6 and 7: Upgrade of custom query and plug-in mechanisms*
  Changes to the application's model result in required upgrades of the custom query and plug-in mechanisms in the object-oriented

version. However, since no such mechanisms had to be implemented in the harmony-oriented epi-aspects version, no upgrades (apart from slight changes to existing plug-ins) are necessary.

6.5.3 Evidence Supporting Claim 3

"Compared to traditional object-oriented programming, implementation of reliable feedback and error recovery mechanisms requires fewer steps when using resonance-oriented software development approaches."

The epi-aspects studies in section 6.3 show that resonance-oriented software development approaches can be used to add quality feedback and error recovery mechanisms to applications in a non-invasive manner: the code within applications does not have to be changed, as all quality feedback and error recovery concerns are handled by the autopoietic system and epi-aspects. Significant effort is required to add quality feedback and error recovery to a traditional object-oriented application without using epi-aspects or other frameworks for creating self-sustaining software. Additionally, the software evolution study using the harmony-oriented epi-aspects architecture (6.4.2) suggests that development of only three spatial epi-aspects implementing quality feedback and error recovery features can be sufficient to help keep a buggy implementation running, and to provide developers with enough room to fix critical bugs.

□ **End of chapter.**

Chapter 7

Discussion

This chapter discusses various aspects of resonance-oriented software development, such as practical issues and limitations, and compares them to approaches proposed by other researchers.

7.1 Resonance-Oriented Development Style

As explained in section 1.4, resonance-oriented software development is characterized by a well-defined environment that interacts with code entities. Executing code results in changes to the environment, and changes in the environment can affect the behavior of code.

The environment of resonance-oriented programs is always active. As a result, developers interact with the running environment when adding new code entities and applying changes. This development style is comparable to the *living objects* concept found in image-based programming languages, such as Smalltalk and Self, and the hot code swapping feature of Erlang. However, the mutual and continuous effect of environment and code entities in resonance-oriented programs on each other results in an even more dynamic development style. In

particular, the resonance-oriented development style can be characterized as follows:

- Software is not developed in terms of clearly defined, distinct phases (i.e. design, implementation, test).
- Thorough design before coding is discouraged, because of the complexity of interdependencies between environment and code entities.
- Developers can simulate and analyze changes to environment and code entities, before they actually apply them.

7.2 Harmony-Orientation

The following sections provide a discussion of conceptual and practical issues of harmony-orientation and harmony-oriented programming.

7.2.1 Encapsulation and Information Hiding

Harmony-oriented programming is based on the principles of information sharing and information diffusion. As a result, spatial constructs in harmony-oriented programs do not encapsulate and hide their data like objects. Snippets can have a state, but the state is owned and diffused by the space.

However, spatial constructs only receive copies of diffused data. Thus the state of snippets, should they decide to maintain one, is protected and cannot accidentally be changed by other spatial constructs.

Also, the data diffusion process itself can be considered as a kind of flexible or temporary kind of encapsulation and information hiding.

For example, after a spatial construct puts data into the space and before the diffusion starts, this data cannot be accessed by any other spatial construct. At this particular moment, perfect encapsulation and information hiding is achieved. Then, once the diffusion begins and the further it proceeds, the more spatial constructs can access the data, decreasing the degree of encapsulation and information hiding. Eventually, the diffusion reaches all spatial constructs.

In Harmony-Oriented Smalltalk, programmers can use the diffusion inspector to adjust or even disable diffusion. Hence, programmers can dynamically change the degree of encapsulation and information hiding within spaces.

7.2.2 Software Reusability

Object-oriented design and programming supports software reusability, because objects are defined and implemented as more or less independent entities. The more independent and generic an object is, the higher is the probability that it can be reused in other programs. For example, object-oriented libraries provide reusable objects like containers, such as linked lists, maps, and trees.

As explained in [61], code reusability can be classified as "reusability in the large" and "reusability in the small". Code reusability in the small refers to taking a piece of code (or object) from one program and reusing it in a closely related program. Code reusability in the large refers to implementing a general algorithm, such as a sorting algorithm, and making it reusable widely through libraries or application frameworks.

Even though harmony-oriented programming relaxes the principles of encapsulation and information hiding and is based on the spaciality

principle, it does support reusability. For example, single snippets that perform generic or specific algorithms, such as a data filter, can be reused in the large or in the small.

However, because of the spaciality principle, reusing groups of snippets that cooperate to provide functionality is challenging. Especially if the number of snippets is large, it becomes difficult to insert them into another virtual space without having to rearrange the snippets already existing in that space. The challenge of reusing a group of entities is not limited to harmony-oriented programming and also applies to object-oriented programming. However, the spaciality principle increases the complexity of reusability in harmony-oriented programs.

7.2.3 Applications and Limitations

Harmony-oriented programming is most suitable for developing applications whose parts process data flows or messages. As explained in the preliminary study on harmony-orientation in section 3.3, possible concrete applications are load balancing systems and system monitoring and management software. Additionally, the studies in section 6.2 indicate that harmony-oriented programming is suitable for implementing servers and various kinds of data processing filters.

The most important limitations of harmony-oriented programming are performance and memory requirements. Compared to traditional object-oriented programs, harmony-oriented programs require additional resources for performing diffusion and exchanging data between spatial constructs and spaces. Assuming that diffusion intensities are stored in 32-bit floating point numbers and a two-dimensional space with a dimension of 500 by 500 is used, then the required memory for the diffusion data of spatial constructs can be as high as almost one megabyte

(976 KB). In practice, the memory issue can be tackled by limiting diffusion of each spatial construct to a maximum extent, such as 50 by 50 or 100 by 100.

Another limitation of harmony-oriented programming is related to security and safety. In harmony-oriented programs, all data is openly available inside the space (or spaces) and can be easily accessed and modified during run-time. As a result, in comparison with other programming approaches, it is easier for attackers to obtain potentially confidential data, change programs, and plant viruses. However, in actual implementations of harmony-oriented runtime environments, this problem can be partially mitigated through techniques like code signing and developer authentication.

A further limitation is that, because of the flexible nature of harmony-oriented programs, the complexity of testing increases and that, compared to other programming approaches, it is easier for programmers to write code that behaves in an unpredicted way. The complexity of testing is increased, because the output of a snippet or other spatial construct is not only defined by its implementation, but also by its location in the space. However, because all data is held by the space, it is possible to implement development environments that can simulate changes to code and location before they are actually applied by a programmer. As a result, the complexity of testing and probability of writing code that behaves in unpredicted ways can be reduced by concrete implementations of visual development environments.

7.2.4 Harmony-Orientation on Manycore CPUs

As suggested by David Ungar, harmony-orientation might be a possible model for programming future "manycore" processors[1] [4] that have hundreds or even thousands of cores. One issue with future "manycore" processor architectures is how synchronization and communication between processors is realized, and how to write programs that effectively utilize the provided computing power. Global synchronization of thousands of processor cores would have a serious performance impact and is thus not a viable option. Local synchronization is more promising, because of the reduced overhead.

If the cores of a future "manycore" processor were logically arranged into one or more a two-dimensional grids, these grids could be considered as harmony-oriented spaces with each core being one "location" of the space. Using the harmony-oriented approach, each core would only interact with its direct neighbors to exchange data. Each piece of data would have an associated intensity, which is decreased whenever it is passed from one core to another. Once the intensity reaches zero, the data is not passed on. Such an approach would be equivalent to the diffusion of harmony-oriented programming.

7.2.5 GPU-Acceleration

Graphics processing units (GPUs) operate in a parallel, pipelined fashion, and are optimized for operations with low arithmetic density. Fourth and later generation GPUs provide programmability of vertex and pixel transformations. General purpose GPU computing (GPGPU) [26] refers to the concept of exploiting the processing power of GPUs for

[1] Also called massively multi-core processors.

performing general purpose calculations. To set up a GPU for a general purpose computation, a so-called *fragment program* implementing the desired computation in a high level shading language, such as Cg [27] and the OpenGL Shading Language (GLSLang) [55], is loaded onto the GPU, and two or more two-dimensional textures are created inside the graphics memory. The textures are used for exchanging data with the computers main processor or processors with one texture holding the input for and one texture receiving the output of the general purpose computation.

Harmony-oriented spaces perform diffusion to facilitate data exchange. The larger a space, the more serious is the impact the diffusion process has on the overall performance of the program. Various diffusion equations, such as [45] , have been optimized for implementation on GPUs. As a result, the performance of harmony-oriented runtime environments can be improved significantly by offloading all diffusion computations to the GPU.

The implementation of the HOS runtime environment uses two-dimensional arrays for holding the various intensity values for each substance. During each diffusion step, these arrays are processed sequentially and the intensity values are updated. The HOS runtime could be accelerated significantly by storing the intensity values for each substance in textures and letting the GPU perform the calculations.

7.3 Conscientious Resonance-Orientation

The discussion in the following sections applies to both epi-aspects and the combined approach described in section 6.4 (harmony-oriented epi-aspects).

7.3.1 Limitations of Epi-Aspects

The epi-aspects architecture encourages a clear separation between application functionality and an autopoietic system for monitoring, regulation, and error recovery. This architectural separation is a shift in software engineering practice, which focuses on application functionality and often neglects well-known error recovery and adaptation techniques. Since the autopoietic system is not an artificial intelligence, but implemented by developers who have designed rules for keeping an application running as smoothly as possible, certain unpredictable conditions can still cause the application to perform unwanted actions. Critical failures that crash the system can be handled by the autopoietic system. However, it is not possible to prevent an application from doing something it is not supposed to do. As such, the epi-aspects architecture is prone to human failure.

A practical issue of epi-aspects not addressed in the previous chapters is the problem of potential buggy epi-aspects. Since epi-aspects can contain a significant amount of code, the introduction of latent bugs is possible. As a result, epi-aspects have to provide a mechanism that reliably performs self-updates. One possible approach is the usage of a "meta" epi-aspect that monitors the epi-aspects for internal problems.

Another issue is the update of epi-aspects. The study presented in section 6.3 only implements a dedicated epi-aspect that provides a mechanism for updating the classes of the CMS system, but not the epi-aspects. The most straightforward approach for dealing with the issue is to implement a dedicated epi-aspect that provides functionality for reliably updating other epi-aspects and itself.

Human failure, in general, is an important factor when developing self-

sustaining systems. It is impossible to prevent developers from creating problematic epi-aspects that directly or indirectly harm the system. Even if near-perfect autopoietic programming languages and a stable autopoietic system are available, there is still room for failure. For example, a developer is still able to develop epi-aspects that are not optimal and cause minor irregularities. If such irregularities accumulate, then the system may fail to deliver expected results despite the autopoietic part keeping it alive.

7.3.2 Realizing an Autopoietic system

The autopoietic simulator of the Epi-AJ framework is meant for development and test purposes. To use epi-aspects in real world applications, the development of a full autopoietic system is necessary. Apart from the lack of autopoietic programming languages as envisioned by Gabriel and Goldman in [36], the following issues have to be addressed.

The first question is how to implement and deploy an autopoietic system. One option is to implement is as a program that runs directly on the computer's hardware and provides a virtual machine for running an operating system, similar to VMWare [98], Virtual Box [94], and Colinux [1]. The advantage of this approach is that the autopoietic system does not depend on other software, which might be buggy. Furthermore, components, drivers, and applications of the operating system can be realized as aspect oriented conscientious software that is woven into the autopoietic system on startup.

Another similar option is running the autopoietic system on top of an existing, stable operating system kernel, which provides hardware abstraction, basic services, and includes drivers.

A third option is to implement the autopoietic system as an application running on an operating system or inside a virtual machine. Advantages are that this approach has lower implementation complexity. The major disadvantage is that the autopoietic system depends on an operating system or virtual machine and therefore is only as stable as the underlying software.

Another technical issue that has to be resolved are the exact mechanisms for invoking recommendation and query advice woven into the autopoietic system, and for transporting messages from epi-aspects to the autopoietic system via an epi-queue. If autopoietic system and application run in the same process, which is the approach used by the autopoietic simulator, this issue is trivial. However, running the autopoietic system and application in the same process defeats the purpose of conscientious software, because a critical failure in the application might terminate the process and thus the autopoietic system.

□ **End of chapter.**

Chapter 8

Related Research and Comparison

This chapter covers various approaches related to resonance-oriented software development, such as diffusion-based agent systems, software evolution research, related programming approaches, self-sustaining systems, and error recovery. The more closely related work is compared with the proposed resonance-oriented software development approaches.

8.1 Agent-Oriented Software Development

8.1.1 Agent-Oriented Programming

Agent-oriented programming, first proposed by Shoham in [84] and [85], changes the notion of programs from hierarchies of entities with static relations into societies of actively interacting autonomous agents with goals and intentions. Agent-oriented programming languages allow developers to tackle a problem by abstracting in terms of individuals with roles and intentions. Unlike objects in object-oriented pro-

grammming, agents are constructs that possess an independent thread of control, are adaptable, and actively pursue goals.

8.1.2 Diffusion-Based Agent Systems

Diffusion has been adapted as a means for interaction in various concrete multi-agent systems, such as [49] and [46]. For example, Tsui et al. propose a diffusion-based multiagent framework for solving optimization tasks in [91]. In particular, diffusion is used for allowing agents to cooperate towards finding a global optimal solution in their framework.

Repenning proposes collaborative diffusion as an agent-based artificial intelligence system for computer games in [80] and [81]. In this system, agents can emit a "scent" that is diffused by the tiles of a game and can be used by other agents for tracking.

8.1.3 Comparison With Harmony-Orientation

Like agent-oriented programming, harmony-oriented programming does not use static hierarchies of program entities: spatial constructs can be re-arranged by developers during run-time to change the program. On the surface, harmony-oriented programming appears to be similar to diffusion-based multi-agent systems, such as [80], in particular. However, spatial constructs in harmony-oriented programming, such as snippets, are fundamentally different from agents. Consider the following typical agent features:

- **Autonomy:** Agents operate without the direct intervention of humans or others, and have some kind of control over their ac-

tions and internal state (an agent usually has its own thread of execution).

- **Social ability:** Agents interact with other agents (and possibly humans) via some kind of agent-communication language.
- **Reactivity:** Agents perceive their environment and respond to changes that occur in it.
- **Goal-orientation:** Agents do not simply act in response to their environment, they are able to exhibit goal-directed behavior by taking the initiative and conducting negotiations.
- **Adaptiveness:** Agents learn and change their behavior based on its previous experience.

A spatial construct, on the other hand, is just a piece of logic that runs in an environment that facilitates reactivity. It is possible to write snippets and other spatial constructs that implement some or all of the agent feature above, but as a conceptual construct, spatial constructs are not comparable to agents.

8.2 Software Evolution

Researchers are investigating various approaches and strategies for dealing with software evolution more effectively. One area is the evaluation of benefits of aspect-oriented software development, reflection, and meta-data to software evolution. For example, in [60] Liu et al. describe an approach combining aspects, XML, and management tools for enabling system-wide software evolution, and Rank makes the case for reflective software architectures to facilitate software evolution in [78].

Like the various architectures proposed within this research area, the resonance-oriented approaches based on epi-aspects make use of aspect-orientation. However aspect-orientation is only one of many possible choices for realizing a concrete resonance-oriented software architectures, and cannot be considered as a design principle of resonance-oriented programming.

Another research field is software evolvability in general. Publications in this field cover topics like models for and simulation of software evolution [99], software change prediction [53], and tools for software evolution management [51].

8.3 Programming Approaches

8.3.1 Spreadsheets, Subtext and Coherence

Spreadsheets can be considered as visible, functional programs that are being executed continuously. Subtext [19, 20, 21] is a non-textual programming language inspired by spreadsheets whose programs consists of trees of nodes. Programs are constructed by copying nodes and run by evaluating the tree. Subtext has evolved into the Coherence [22] language, which is based on a model of change-driven computation that relives programmers from the burden of managing side effects.

Spreadsheet programming languages and languages inspired by spreadsheets, such as Subtext, share the following similarities with harmony-oriented programming: firstly, spreadsheets and harmony-oriented programs are continuously executing, even when code and data are being edited. Hence, any change is immediately applied and visualized. Secondly, like harmony-oriented programs spreadsheets arrange code and data in a two dimensional space and are data driven. The difference

is that spreadsheet languages are functional programming languages while harmony-oriented programming is based on principles like information diffusion, balance, and code exposure.

8.3.2 Erlang

Erlang [3, 12] is a general purpose, functional programming language and runtime environment designed for concurrency and robustness. The Erlang runtime environment is designed to reduce the complexity of software maintenance and error recovery through hot code swapping and incremental code loading. The hot code swapping feature allows developers to change running programs, and provides facilities for phasing out or recovering old code. Incremental code loading allows developers to specify how much code is loaded (or unloaded) at what time, and hence facilitates isolation and correction of buggy code in running programs.

In harmony-oriented programs, it is also possible to change or replace code and make other changes during runtime. However, harmony-orientation itself does not provide mechanisms for improved maintenance and error recovery. Such functionality is realized by epi-aspects, the second resonance-oriented approach, and the combined harmony-oriented epi-aspects approach introduced in section 6.4.1.

8.3.3 Dataflow Programming

Dataflow programming languages like Luicd [97] and LabView's "G" language [50] allow programmers to model programs as graphs that define how data is passed from one operation to another. Many dataflow programming languages are visual languages that provide a user interface for programmers to define operations that can be connected to

create a directed graph. They are typically used for applications implementing data transformation or acquisition, such as image and video filters.

The first resonance-oriented approach, harmony-oriented programming, shares some characteristics with dataflow programming, because spatial constructs and the spaces exclusively interact via exchanging data. Also, like many dataflow languages, the harmony-oriented runtime and development environment, Harmony-Oriented Smalltalk, is visual. The main difference between harmony-oriented programming and dataflow languages is that the latter encourage the programmer to specify a complete dataflow description in the form of a directed graph before the program is run. However, harmony-oriented programming does not require the programmer to think about a complete description of the program's dataflow. Rather, the programmer focuses on small parts (areas) of the program and incrementally develops it while it is running.

8.3.4 Blackboard Architectures

Blackboard architectures [47, 25] facilitate information exchange between program entities through a global database. This database, which is called a blackboard, can be freely accessed by all entities to publish and obtain data. An important characteristic is that all data in the blackboard is available to all entities at all times. Concrete blackboard implementations allow entities to subscribe to desired kinds of data, and to receive notifications whenever matching data is published or modified. As explained in [47], a common application of blackboard architectures is problem solving: a set of problem solving entities uses the blackboard to publish, process, and respond to hierarchically struc-

tured hypotheses.

In harmony-oriented programming, the data put into spaces by snippets can be freely consumed by other snippets. However, because of the diffusion process, the data is not available globally at all times like in blackboard architectures. Rather, the virtual location of a spatial construct inside a space and the extend of diffusion determine which data a spatial construct can access. Also, a spatial construct can not directly query or search the space for particular data, and the space controls which data snippets receive.

8.3.5 Phenotropic Computing

Lanier proposes phenotropics as an alternative to argument-based interfaces in [58]. The main idea of phenotropics is that components have surfaces that display information about their functionality rather than rigid interface definitions. Interacting components observe and interpret and the meaning of each other's surfaces, and react accordingly. Unlike interfaces, phenotropics uses approximation rather than clearly defined protocols.

8.4 Self-Sustainment and Reliability

8.4.1 Autopoietic Software Systems

Autopoietic software systems were first proposed in the 1970s [95]. They have been widely considered a computational model and applied in the field of artificial intelligence [105, 66]. During the 1980s the concept received less attention and was rediscovered in the late 1990s [67]. Since autopoiesis is widely considered a computational concept, most

research focuses on algorithms and simulations of simple autopoietic systems. Various versions of such simulations implemented in Pascal and Fortran programming languages are available [52, 68].

8.4.2 Autonomic Computing

IBM devised the notion of autonomic computing [54, 71]. It refers to concepts and technologies that enable software to become more self-managing. To achieve this goal, autonomic computing proposes four principles: self-configuration, self-healing, self-optimization, and self-protection. According to [54], self-configuration refers to software components and systems that automatically follow a set of high-level configuration policies. In case of policy changes, the entire system adjusts itself automatically. Self-optimization is a process in which components continually seek opportunities to improve their own performance. The self-healing process allows the system to automatically detect and repair software and hardware problems and the self-protection mechanism defends the system against malicious attacks and failures. The self-protection mechanism uses an early warning system that allows anticipation and prevention of system failures. Researchers are exploring aspect-oriented approaches for realizing autonomic computing. For example, Engel et al. propose the usage of dynamic operating system aspects for realizing autonomic software in [24], and Greenwood et al. describe how to use dynamic aspects for implementing an autonomic system in [40].

8.4.3 Commensalistic Software

Commensalistic software [29] is a hypothetical conscientious software architecture based on commensalistic symbiosis proposed by Fleissner

and Baniassad. Commensalistic software and epi-aspects are the first architectures inspired by the theoretical notion of conscientious software.

8.4.4 Reflective and Adaptive Middleware

Research in the field of reflective and adaptive middleware [8, 83, 63, 23] shares some goals with autonomic computing and conscientious software. As described in [39], openness and dynamic self-adaptation are fundamental properties of reflective middleware, and therefore, reflective middleware is suited to support autonomic computing and self-sustaining systems.

For instance, the approach by Rasche et al. [79] proposes the usage of dynamic aspect weaving for reconfiguration. The Rainbow framework proposed by Garlan et al. in [38] is a concrete adaptive middleware architecture that uses monitoring and constraint evaluation for adaptation.

8.4.5 Monitoring-Oriented Programming

Monitoring-oriented programming, as described by Chen et al. in [14, 15, 13], is a practical programming paradigm that uses monitoring as the fundamental principle for implementing reliable software. The formal specification of an application is used as the basis for generating a set of monitors that are integrated into the software. During runtime, these monitors observe the runtime behavior of the application and trigger user-defined routines, when a specification is validated or violated.

8.4.6 Recovery-Oriented Computing

Recovery oriented computing (ROC), explored by Patterson, Brown et al. in [75] and [11], suggests planning to incorporate or recover from a certain class of errors, rather than trying to prevent them from arising The major aim of recovery oriented computing is to minimize the mean time to repair in case a system failure occurs. In order to enable fast recovery after a failure, ROC employs the following six techniques: recovery experiments, diagnosis, partitioning, reversible systems, defense in depth, and redundancy.

8.4.7 Acceptability Envelope

Rinard et al. explore imperfect but acceptable software systems in [82]. Their research proposes an *acceptability envelope*, a concept referring to software that is flawed, but delivers acceptable service to users. According to Rinard, many deployed systems do an acceptable job despite errors and attempting to develop a flawless system can be considered as counter-productive, because of the burden placed on the developer.

8.4.8 Software Reliability Engineering

Compared to the various software architectures and philosophies regarding self-sustaining systems and error recovery introduced in the previous sections, software reliability engineering refers to clearly defined software engineering processes and practices. As indicated in chapter 1 of [62], software reliability started "evolving from an art into a practical engineering discipline" ([62], p 9) in the mid 1990s.

8.4.9 Comparison With Epi-Aspects

As illustrated in the previous sections, error recovery, self-sustaining software, and software reliability are well-understood concepts.

Conscientious software, the paradigm that inspired epi-aspects, is inspired by the realization that, even though various error recovery approaches exist, they are not frequently applied in practice by the software engineering and programming communities. Conscientious software requires two distinct system parts written in different programming languages whereby one part is solely responsible for error recovery and keeping the system alive, and the other part implements application functionality. This approach is meant to encourage software developers to allocate sufficient resources for implementing each part.

The epi-aspects architecture goes one step further. It allows developers to write an application without considering stability and error recovery at all, and then later upgrade the application into self-sustaining software in a non-invasive manner. Unlike many existing approaches, the epi-aspects architecture does not require design before coding in regard to error recovery and self-sustainment. Component tests, mechanisms for error monitoring and recovery, and software maintenance mechanisms can be added at a later stage via epi-aspects without modifying the existing application.

Additionally, even though the epi-aspects architecture is designed for realizing features related to self-sustainment, epi-aspects can be used to implement various other features, such as plug-in mechanisms and application extensions.

□ **End of chapter.**

Chapter 9

Conclusions

This chapter summarizes the research presented in this thesis, describes its contributions, and discusses future work.

9.1 Summary

The aim of this research was to introduce the notion of resonance-oriented software design and development and to show that, in comparison with traditional object-oriented programming, concrete resonance-oriented approaches allow programmers to deal with software evolution more effectively.

After describing a preliminary study aimed at exploring how different reasoning styles of individuals from different cultural backgrounds apply to the realm of software development, two concrete resonance-oriented approaches called harmony-oriented programming and epi-aspects were introduced.

The main idea behind harmony-oriented programming is that pieces of a program always interact with their environment as a whole and

usually not with other program parts directly. Harmony-oriented programming challenges established and widely accepted object-oriented principles, such as strong encapsulation, information hiding, and inheritance, and favors more flexible and ah-hoc approaches for structuring and implementing programs

Epi-aspects are a concrete resonance-oriented architecture based on aspect-oriented programming and conscientious software that introduces epi-aspects as a construct for combining an autopoietic system and applications into self-sustaining software by facilitating feedback and resonance.

Development environments for both proposed approaches were implemented and used to evaluate resonance-oriented programming through various studies. As part of the studies, a third resonance-oriented approach, called harmony-oriented epi-aspects, combining harmony-oriented programming and epi-aspects was introduced and evaluated.

The studies showed that resonance-oriented software development enhances factors affecting software evolution, such as ease of change, extensibility, maintainability, and error recovery, in comparison to traditional object-oriented programming.

9.2 Contributions

In addition to proposing and validating resonance-oriented software design and development, this research makes various contributions.

9.2.1 Research Contributions

This research makes the following contributions to software engineering and programming languages research:

1. It illustrates a new way of programming that relaxes encapsulation and information hiding without increasing the dependencies of program parts on each other. In fact, direct dependencies between program parts are decreased.
   (Harmony-oriented programming).

2. It proposes the first concrete architecture based on the theoretical notion of conscientious software as envisioned by Gabriel and Goldman. (Epi-aspects).

3. It illustrates a concrete aspect-oriented architecture where aspects advise on program parts written in two fundamentally different programming languages. (Epi-aspects).

4. It introduces a new kind of aspect that is able to advise on areas in virtual spaces. (Spatial epi-aspects).

5. It shows that cultural differences in reasoning can be applied to the areas of programming languages and software engineering.
   (Preliminary study and harmony-oriented programming).

9.2.2 Software Contributions

The major software contribution of this research is HOS, the harmony-oriented runtime and development environment described in chapter 4.3.

- HOS is an open source project hosted on SqueakSource:
  `http://www.squeaksource.com/hos.html`.

- Its current version provides visual support for harmony-oriented spaces and snippets, and non-visual support for spatial epi-aspects.

9.3 Future Work

The work presented in this thesis can only be considered as the first steps towards defining and validating resonance-orientation as a software development paradigm. Even though the studies presented in this thesis provide strong evidence in favor of resonance-oriented software development, they cannot be considered as a definitive proof that resonance-oriented software development is indeed more suitable for actual large-scale software development projects than traditional object-oriented programming. Hence, further experimental studies are required to evaluate resonance-oriented software development in the context of long-running industrial software development projects.

Another future topic is the design of resonance-oriented languages and corresponding virtual machines, such as a pure harmony-oriented programming language that could be used instead of Smalltalk in the HOS runtime and development environment. Additionally, epi-aspects and other software architectures using an autopoietic part would benefit from concrete autopoietic programming languages, languages that are designed in a way to make it difficult for programmers to introduce bugs.

Additionally, the implementations of the development environments for the proposed resonance-oriented approaches are still in an early stage. Especially the HOS development environment and its support for harmony-oriented epi-aspects requires more work in both the virtual machine and graphical user interface to become suitable for large-scale projects and studies.

□ **End of chapter.**

Appendix A

A Semantics for HOS

This appendix is an initial semantics for the experimental version of Harmony-Oriented Smalltalk (HOS) introduced in section 4.3 focusing on spaces, snippets, and diffusion of substances. In particular the following paragraphs describe the semantics for the methods provided by the *space* and *state* objects described section 4.3.2, which facilitate maintenance of snippet state and interaction between snippets and spaces.

The notation used in this appendix is inspired by axiomatic semantics, as described in [87], and uses the following format for defining semantics:

- A precondition block.
- Harmony-Oriented Smalltalk statement(s).
- A result block.

The precondition block describes the state of space, snippets and substances before the HOS statements are executed, and the result block describes the effect the HOS statements have. The following notations are used to define the state of spaces, snippets and substances:

```
space (width, height)
snippet (posX, posY, limit, log)
state (posX, posY, type, value)
substance (originX, originY, area, data)
substance-particle (posX, posY, intensity, data)
```

Spaces are defined by their width and height. Snippets are defined by their position inside the space (posX and posY), a limit value that controls the maximum extend to which a substance carrying data produced by the snippet is diffused, and the contents of their associated log. States are defined by their position in the space (which the same as the snippet they are associated with), their type, and their value. There are two notions for defining substances. The first notation defines a substance by its origin, the surface area it covers, and the data it carries. The second notation, denoted as "substance-particle", defines the intensity a substance has at a given position in the space and its data. The data of substances is defined as follows:

```
data := [ taggedValues ]
taggedValues := taggedValue | taggedValues, taggedValue
taggedValue := typeOrValue | typeOrValue "_{" tags "}"
typeOrValue := [a - z, A - Z, 0 - 9, "]
tags := tags | tag
tag := [a - z, A - Z, 0 - 9]
```

For example, a possible definition of a substance's data might look like this:

```
data = ["Hello", 22, 50_{tag1, tag2}]
```

In addition to the notations for spaces, snippets and substances, the semantic descriptions in the paragraphs below use the following special tags:

- TIME. This tag denotes the current time in milliseconds.
- DF. This tag denotes a factor that controls how much a substance extends during each diffusion step.

A.1 Semantics for Producing Data

Snippet putting an object into the space for the first time:

```
Precondition:[
   snippet (posX = x, posY = y, limit = 1, log = [])
   substance (
      originX = x,
      originY = y,
      area = 0,
      data = []
   )
   t = TIME
]

space put: object.

Result:[
   substance(
      originX = x, originY = y,
      area = max(1, (TIME - t) DF)) π^2,
      data = [object]
   )
]
```

Snippet putting a tagged object into the space when substance has already reached maximum diffusion:

```
Precondition:[
   snippet (posX = x, posY = y, limit = 1, log = [])
   substance (
      originX = x, originY = y,
      area = 1 * π * 2,
      data = [10, 20, 30]
   )
]

space put: 'Sebastian' tag:'Name'.

Result:[
   substance(
      originX = x, originY = y,
      area = 1 * π * 2,
      data = [10, 20, 30, "Sebastian"_{Name}]
   )
]
```

A.2 Semantics for Consuming Data

Snippet consuming one object of type *String* with the tag *Name*:

```
Precondition:[
   snippet (posX = 5, posY = 5, limit = 1, log = [])
   substance-particle(
      posX = 5,
      posY = 5,
      intensity = 0.5,
      data = ['Sebastian', 'Tilman'_{Name}, 'Stefan']
   )
]
```

```
description := DataDescription new
    type: String;
    addTag:'Name';

name := space consume: description.

Result:[
   substance-particle(
      posX = 5, posY = 5, intensity = 0.5,
      data = [‘Sebastian’, ‘Stefan’]
   )
]
```

Snippet consuming one object of type *String* when substance carrying matching data does not reach snippet:

```
Precondition:[
   snippet (
      posX = 5, posY = 5,
      limit = 1,
      log = []
   )
   substance-particle(
      posX = 5, posY = 5,
      intensity = 0.5,
      data = [‘Sebastian’, ‘Tilman’_{Name}, ‘Stefan’]
  )
]

name := space consume: String.

Result:[
   substance-particle(
      posX = 5, posY = 5, intensity = 0.5,
      data = [‘Sebastian’, ‘Tilman’_{Name}, ‘Stefan’]
   )
]
```

Snippet consuming one object of type *String* when two substances carrying matching data are available:

```
Precondition:[
   snippet (
      posX = 5,
      posY = 5,
      limit = 1,
      log = []
   )
   substance-particle(
      posX = 5,
      posY = 5,
      intensity = 0.5,
      data = ['Sebastian', 'Tilman'_{Name}, 'Stefan']
   )
   substance-particle(
      posX = 5,
      posY = 5,
      intensity = 0.83,
      data = ['Gabi', 'Wolfgang']
   )
]

name := space consume: String.

Result:[
   substance-particle(
      posX = 5,
      posY = 5,
      intensity = 0.5,
      data = ['Sebastian', 'Tilman'_{Name}, 'Stefan']
   )
   substance-particle(
      posX = 5,
      posY = 5,
      intensity = 0.83,
      data = ['Wolfgang']
   )
]
```

Snippet consuming all objects of type *Number* when two substances carrying matching data are available:

```
Precondition:[
   snippet (
      posX = 5, posY = 5,
      limit = 1, log = []
   )
   substance-particle(
      posX = 5, posY = 5,
      intensity = 0.5,
      data = [10, 30, 50]
   )
   substance-particle(
      posX = 5, posY = 5,
      intensity = 0.83,
      data = [20, 40, 60]
   )
]

space consume: Number do:[:num|
    log show: num asString.
].

Result:[
   snippet (
      log = [20, 40, 60]
   )
   substance-particle(
      posX = 5, posY = 5,
      intensity = 0.5,
      data = [10, 30, 50]
   )
   substance-particle(
      posX = 5, posY = 5,
      intensity = 0.83,
      data = []
   )
]
```

Snippet consuming all objects of type *Number* whose value is less than six:

```
Precondition:[
   snippet (
      posX = 5,
      posY = 5,
      limit = 1,
      log = []
   )
   substance-particle(
      posX = 5,
      posY = 5,
      intensity = 0.5,
      data = [1, 2, 3, 4, 5, 6, 7, 8]
   )
]

description := DataDescription new
    type: Number;
    constraint:[:val| val < 6].

space consume: description do:[:num|
    log show: num asString.
].

Result:[
   snippet (
      log = [1, 2, 3, 4, 5]
   )
   substance-particle(
      posX = 5,
      posY = 5,
      intensity = 0.5,
      data = [6, 7, 8]
   )
]
```

A.3 Semantics for Observing Data

Snippet observing all objects of type *Number* whose value is less than six:

```
Precondition:[
   snippet (
      posX = 5,
      posY = 5,
      limit = 1
   )
   substance-particle(
      posX = 5,
      posY = 5,
      intensity = 0.5,
      data = [1, 2, 3, 4, 5, 6, 7, 8]
   )
]

description := DataDescription new
    type: Number;
    constraint:[:val| val < 6].

space observe: description do:[:num|
    log show: num asString.
].

Result:[
   snippet (
      log = [1, 2, 3, 4, 5]
   )
   substance-particle(
      posX = 5,
      posY = 5,
      intensity = 0.5,
      data = [1, 2, 3, 4, 5, 6, 7, 8]
   )
]
```

Snippet observing all objects:

```
Precondition:[
   snippet (
      posX = 5, posY = 5, limit = 1,
      log = []
   )
   substance-particle(
      posX = 5,
      posY = 5,
      intensity = 0.5,
      data = ['Sebastian', 'Tilman'_{Name}, 'Stefan']
   )
   substance-particle(
      posX = 5,
      posY = 5,
      intensity = 0.83,
      data = ['Gabi', 'Wolfgang']
   )
]

space observe:[:str|
    log show: str.
].

Result:[
   snippet (
      log = [Gabi, Wolfgang, Sebastian, Tilman, Stefan]
   )
   substance-particle(
      posX = 5, posY = 5,
      intensity = 0.5,
      data = ['Sebastian', 'Tilman'_{Name}, 'Stefan']
   )
   substance-particle(
      posX = 5, posY = 5,
      intensity = 0.83,
      data = ['Gabi', 'Wolfgang']
   )
]
```

A.4 Semantics for Snippet State

Snippet changing the type of its state to *Number* and setting its value to seven:

```
Precondition:[
   snippet (
      posX = 10,
      posY = 10,
      limit = 1, log = []
   )
   substance (
      originX = 10, originY = 10,
      area = 0, data = []
   )
   state(
      posX = 10, posY = 10,
      type = ?, value = ?
   )
   t = TIME
]

state type: Number
state value: 7.

Result:[
 state(
      posX = 10,
      posY = 10,
      type = Number,
      value = 7
   )
   substance(
      originX = 10, originY = 10,
      area = max(1, (TIME - t) DF)) π`2,
      data = [7]
   )
]
```

Snippet changing the type of its state to *OrderedCollection* and adding values:

```
Precondition:[
   snippet (
      posX = 10, posY = 10,
      limit = 1, log = []
   )
   substance (
      originX = 10, originY = 10,
      area = 0, data = []
   )
   state(
      posX = 10, posY = 10,
      type = ?, value = ?
   )
   t = TIME
]

state type: OrderedCollection.
state add: 'Hello';
      add: 'Harmony-Oriented';
      add: 'World'.

Result:[
 state(
      posX = 10, posY = 10,
      type = OrderedCollection,
      value = ['Hello', 'Harmony-Oriented', 'World']
   )
   substance(
      originX = 10, originY = 10,
      area = max(1, (TIME - t) DF)) π 2,
      data = ['Hello', 'Harmony-Oriented', 'World']
   )
]
```

□ **End of chapter.**

Appendix B

Common Observations

Table B.1 provides the results of the analysis of the observations recorded during the preliminary study on how Nisbett's findings regarding different reasoning styles of individuals from different cultural backgrounds apply to the realm of software development (section 3). It shows which category the responses fit under, and some observations fit into multiple categories. The total number of responses counts the individual responses from all categories; the same respondent may have uttered more than one response for a particular observation. The number of individual respondents who reported an observation is shown in parentheses under *Total Number*.

Remark	Total #	Context: Env	Context: Int	Context: Obs	Context: Role	Rel: Short	Rel: Long	Puzzlem't
Empty seats, but people are waiting	6 (5)	2:1	1:5, 2:1, 2:6, 6:5, 8:8, 11:11					2:6
Bad mother	6 (5)	2:8, 3:3				2:8, 3:8, 6:3, 7:12, 11:1		
Man unable to pay	6 (4)		3:10, 4:3, 7:3	1:1			3:11, 4:10	
Misplaced cook	6 (4)		6:11		4:5, 6:10, 7:15			3:15, 7:10
Looking at the piano player	4 (3)						3:5, 4:12, 7:2, 7:7	
Tables unaccounted for	3 (3)			2:2, 6:6, 8:12				2:2
Strange table	3 (3)							1:2, 3:13, 9:1
Man "From the north pole"	3 (3)		6:12, 7:11					1:1
People are drinking, not eating	3 (3)			2:1, 8:1, 9:3				
Busy workers	3 (2)	4:6		8:6	4:18			
Plates will fall	2 (2)	6:4, 11:1		11:1	11:1			
Crying chef	2 (2)							2:7, 4:13
Strange picture	2 (2)	1:3		8:10				
Dark restaurant	2 (2)	2:3		6:7				
Sad pianist	2 (2)	6:8	6:8, 11:2					
People outside	2 (2)	6:13		8:2				
Man under table	2 (2)		4:14, 7:9					7:8
Cake coming	2 (2)						3:14, 4:15	

Table B.1: Common Observations

□ **End of chapter.**

Appendix C

Original GMA Database Design

Figures C.1 and C.1 show the original database design of the GMA order and inventory management system. The figures show an entity-relationship model with German labels. Even though object-oriented programming was used for implementing the applications, the initial designs were database-centric. In the initial versions application, there was no direct mapping between objects and the tables of the database. However, in later versions objects representing the entire application model were introduced and mapped to the tables in the database.

Figure C.1 shows the left part of the entity relationship model. The main entities defined in this part are *Buch* (book), *Kunde* (customer), *Auftrag* (order), *Warengruppe* (category), *Reservierungen* (reservations), and *Vormerkungen* (holds).

Figure C.2 shows the right part of the entity relationship model. The main entities defined in this part are *Lagerort* (storage location), *Autoren* (authors), *Verlag* (publisher), *Ansprechpartner* (contact person), *AdrTexFax* (contact information), and *V_History* (delivery history of publishers).

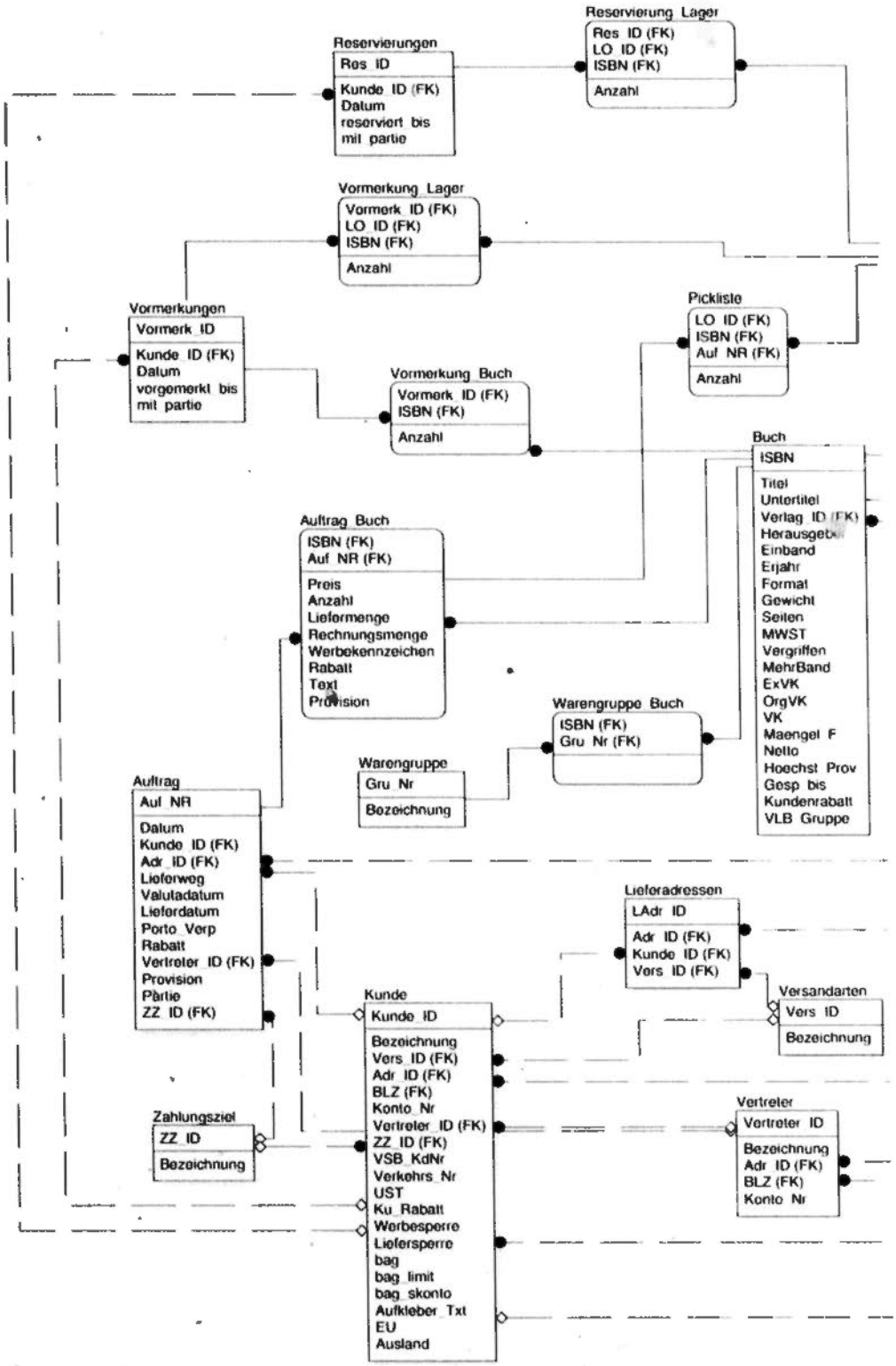

Figure C.1: Original GMA Database Design (Left Part)

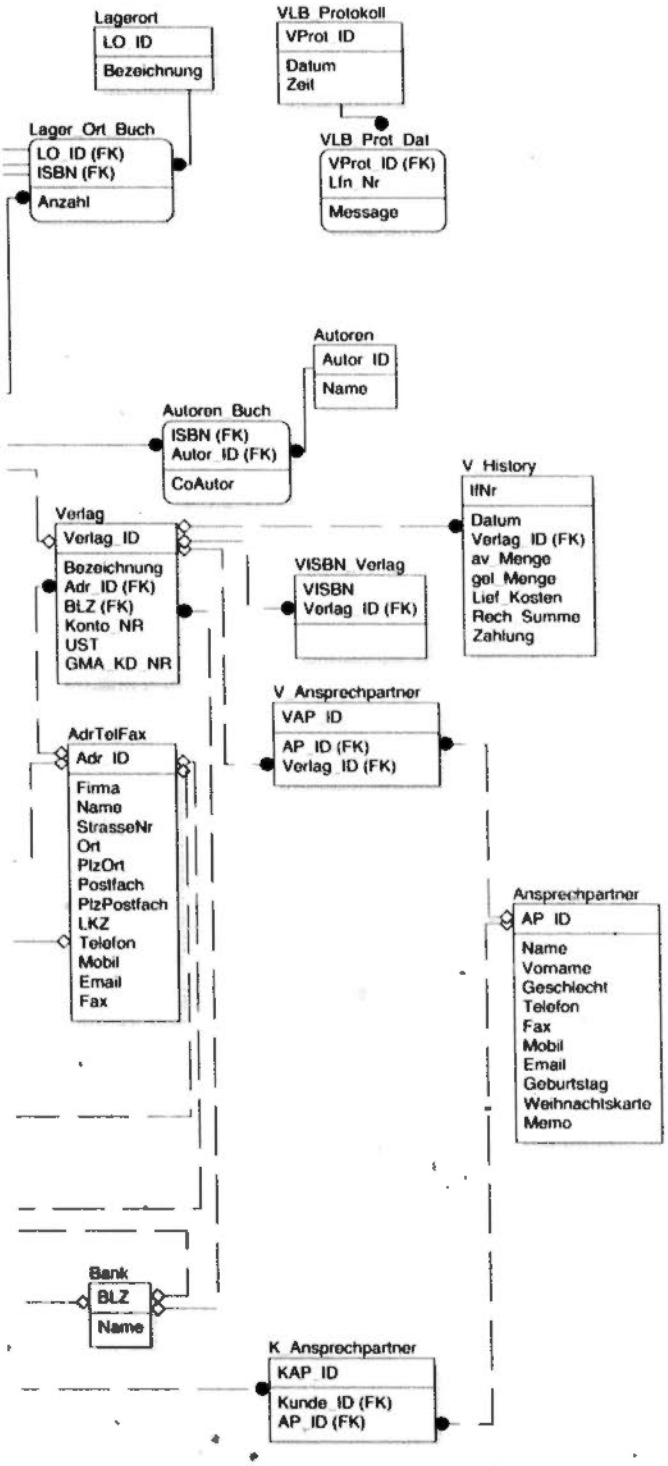

Figure C.2: Original GMA Database Design (Left Right)

□ **End of chapter.**

Bibliography

[1] D. Aloni. Cooperative linux. In *Proceedings of the Linux Symposium*, volume 1, pages 23–31, 2004.

[2] Aristotle. *Physics*. 350 BCE.

[3] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, University of California at Berkeley, 2006.

[5] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc : An extensible aspectj compiler. In *Proceedings of the Eleventh International Conference on Tools and Algorithms for the construction and analysis of systems (TACAS'05)*, volume 3440 of *LNCS*, pages 293–334. Springer-Verlag, 2006.

[6] E. Baniassad and S. Fleissner. The geography of programming. In *OOPSLA 2006: Companion to the 21st annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 560–573. ACM Press, 2006.

[7] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Squeak by Example*. Square Bracket Publishing, 2008.

[8] G. S. Blair, G. Coulson, and P. Grace. Research directions in reflective middleware: the lancaster experience. In *ARM '04: Proceedings of the 3rd workshop on Adaptive and reflective middleware*, pages 262–267, New York, NY, USA, 2004. ACM Press.

[9] G. Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Professional, 2 edition, 1993.

[10] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 1990.

[11] A. Brown. *A Recovery-Oriented Approach to Dependable Services: Repairing Past Errors with System-Wide Undo*. PhD thesis, University of California, Berkeley, 2003.

[12] F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly Media, Inc., 2009.

[13] F. Chen and G. Roşu. Mop: Reliable software development using abstract aspects. Technical Report UIUCDCS-R-2006-2776, Department of Computer Science, University of Illinois at Urbana-Champaign, 2006.

[14] F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Workshop on Runtime Verification (RV'03)*, volume 89(2) of *ENTCS*, pages 108 - 127, 2003.

[15] F. Chen and G. Roşu. Java-mop: A monitoring oriented programming environment for java. In *Proceedings of the Eleventh International Conference on Tools and Algorithms for the construction and analysis of systems (TACAS'05)*, volume 3440 of *LNCS*, pages 546–550. Springer-Verlag, 2005.

[16] O. Ciupke. Automatic detection of design problems in object-oriented reengineering. In *TOOLS '99: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 18. Washington, DC, USA, 1999. IEEE Computer Society.

[17] A. Cromer. *Uncommon Sense: The Heretical Nature of Science*. Oxford University Press, New York, 1993.

[18] S. Ducasse. *Squeak: Learn Programming with Robots*. Apress, 2005.

[19] J. Edwards. Example centric programming. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 124–124, New York, NY, USA, 2004. ACM.

[20] J. Edwards. Subtext: uncovering the simplicity of programming. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 505–518, New York, NY, USA, 2005. ACM Press.

[21] J. Edwards. No ifs, ands, or buts: uncovering the simplicity of conditionals. *SIGPLAN Not.*, 42(10):639–658, 2007.

[22] J. Edwards. Coherent reaction. 2009.

[23] F. Eliassen, E. Gjørven, V. S. W. Eide, and J. A. Michaelsen. Evolving self-adaptive services using planning-based reflective middleware. In *ARM '06: Proceedings of the 5th workshop on Adaptive and reflective middleware (ARM '06)*, page 1, New York, NY, USA, 2006. ACM Press.

[24] M. Engel and B. Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *AOSD '05; Proceedings of the 4th international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, 2005. ACM Press.

[25] R. Engelmore. *Blackboard Systems*. Addison-Wesley, 1988.

[26] R. Fernando. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley, 2004.

[27] R. Fernando and M. J. Kilgard. *The Cg Tutorial*. Addison-Wesley, New York, 2003.

[28] A. Fjuk. *Comprehensive Object-Oriented Learning: The Learner's Perspective*. Informing Science, 2006.

[29] S. Fleissner and E. Baniassad. A commensalistic software system. In *OOPSLA 2006: Companion to the 21st annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 510–520. ACM Press, 2006.

[30] S. Fleissner and E. Baniassad. Epi-aspects: aspect-oriented conscientious software. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 659–674. ACM Press, 2007.

[31] S. Fleissner and E. Baniassad. Towards harmony-oriented programming. In *OOPSLA '08: Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 819–822. ACM Press, 2008.

[32] S. Fleissner and E. Baniassad. Harmony-oriented programming and software evolution. In *OOPSLA '09: Companion to the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 2009.

[33] S. Fleissner and E. Baniassad. Harmony-oriented smalltalk. In *OOPSLA '09: Companion to the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 2009.

[34] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[35] Y. L. Fung. *A Short History Of Chinese Philosophy*. Simon and Schuster Inc., 1997.

[36] R. P. Gabriel and R. Goldman. Conscientious software. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 433–450, New York, NY, USA, 2006. ACM Press.

[37] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing Series. Addison Wesley, 1995. http://www.aw.com.

[38] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[39] P. Grace, G. Coulson, G. S. Blair, and B. Porter. A distributed architecture meta-model for self-managed middleware. In *ARM '06: Proceedings of the 5th workshop on Adaptive and reflective middleware (ARM '06)*, page 3, New York, NY, USA, 2006. ACM Press.

[40] P. Greenwood and L. Blair. Using dynamic aop to implement an autonomic system. In *Proceedings of the 2004 Dynamic Aspects Workshop (DAW04), Lancaster*, pages 76–88. RICAS, March 2006.

[41] P. H. Gries and K. Peng. Culture clash? apologies east and west. *Journal of Contemporary China*, 11(30):173–178, 2002.

[42] M. J. Guzdial. *Squeak: Object-Oriented Design with Multimedia Applications*. Prentice Hall, 2000.

[43] E. Hamilton. *The Greek Way*. Avon, 1973.

[44] C. Hansen. *Language and Logic in Ancient China*. University of Michigan Press, 1983.

[45] M. J. Harris, G. Coombe, T. Scheuermann, and A. Lastra. Physically-based visual simulation on graphics hardware. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 109–118, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[46] T. Hogg. Coordinating microscopic robots in viscous fluids. *Autonomous Agents and Multi-Agent Systems*, 14(3):271–305, 2007.

[47] J. Hunt. *Blackboard Architectures*. JayDee Technology Ltd., 2002.

[48] M. Imai and D. Gentner. A cross-linguistic study of early word meaning: Universal ontology and linguistic influence. *Cognition*, 62(2):169–200, 1997.

[49] Y. Jiang, J. Jiang, and T. Ishida. Agent coordination by trade-off between locally diffusion effects and socially structural influences. In *AAMAS '07: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–3, New York, NY, USA, 2007. ACM.

[50] G. Johnson and R. Jennings. *LabVIEW Graphical Programming*. McGraw-Hill Professional, 4 edition, 2006.

[51] P. Jonsson. The anatomy - an instrument for managing software evolution and evolvability. In *Second International IEEE Workshop on Software Evolvability*, pages 31–37. IEEE, 2006.

[52] F. Jullien and B. McMullin. FRJ's Simple Autopoiesis Program, 1995. Program source in Pascal, for MS-DOS platform.

[53] H. Kagdi and J. Maletic. Software-change prediction: Estimated+actual. In *Second International IEEE Workshop on Software Evolvability*, pages 38–43. IEEE, 2006.

[54] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[55] J. Kessenich. *The OpenGL Shading Language*. The Khronos Group, 2008.

[56] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with aspectj. *Communcations of the ACM*, 44(10):59–65, 2001.

[57] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In

M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220 242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[58] J. Lanier. Why gordian software has convinced me to believe in the reality of cats and apples. *Edge*, 128, November 2003.

[59] C. Liu. *Smalltalk, Objects, and Design.* IUniverse, 2000.

[60] C. R. Liu, C. Gibbs, and Y. Coady. Safe and sound evolution with sonar. *Transactions on Aspect-Oriented Software Development*, 4:163–190, 2007.

[61] M. D. Lubars. Code reusability in the large versus code reusability in the small. *SIGSOFT Softw. Eng. Notes*, 11(1):21–28, 1986.

[62] M. R. Lyu, editor. *Handbook of Software Reliability Engineering.* IEEE Computer Society Press, 1996.

[63] R. Maia, R. Cerqueira, and F. Kon. A middleware for experimentation on dynamic adaptation. In *ARM '05: Proceedings of the 4th workshop on Reflective and adaptive middleware systems*, New York, NY, USA, 2005. ACM Press.

[64] J. Maloney. *An Introduction to Morphic: The Squeak User Interface Framework.* Walt Disney Imagineering, 2000.

[65] A. Martelli, A. Ravenscroft, and D. Ascher. *Python Cookbook.* O'Reilly Media, 2 edition, 2005.

[66] B. McMullin. Computational autopoiesis: The original algorithm. Working Paper 97-01-001, Santa Fe Institute, Santa Fe, NM 87501, USA, Jan. 1997.

[67] B. McMullin and F. J. Varela. Rediscovering computational autopoiesis. In *Fourth European Conference on Artificial Life (ECAL'97)*, pages 38–47, 1997.

[68] J. Mingers and B. McMullin. JM's Simple Autopoiesis Program, 1997. Program source in Pascal, for MS-DOS platform.

[69] R. Mordani, editor. *Common Annotations for the Java Platform.* Sun Microsystems, Inc, 2006.

[70] M. W. Morris and K. Peng. Culture and cause: American and chinese attributions for social and physical events. *Journal of Personality and Social Psychology*, 67(6):949–971, 1994.

[71] R. Murch. *Autonomic Computing.* IBM Press, March 2004.

[72] C. L. Nehaniv, J. Hewitt, B. Christianson, and P. Wernick. What software evolution and biological evolution don't have in common. In *Second International IEEE Workshop on Software Evolvability*. IEEE Computer Society, 2006.

[73] R. E. Nisbett. *The Geography of Thought*. Free Press, 2003.

[74] J. Noble and R. Biddle. Notes on notes on postmodern programming: radio edit. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 112–115, New York, NY, USA, 2004. ACM Press.

[75] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery oriented computing (roc): Motivation, definition, techniques,. Technical report, Berkeley, CA, USA, 2002.

[76] D. J. Pearce and J. Noble. Relationship aspects. In *the ACM conference on Aspect-Oriented Software Development (AOSD'06) (to appear)*, 2006.

[77] K. Peng, D. R. Ames, and E. D. Knowles. *Culture and Human Inference: Perspectives from Three Traditions*. Oxford University Press, 2000.

[78] S. Rank. Architectural reflection for software evolution. In *2nd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution*, 2005.

[79] A. Rasche, W. Schult, and A. Polze. Self-adaptive multithreaded applications: a case for dynamic aspect weaving. In *ARM '05: Proceedings of the 4th workshop on Reflective and adaptive middleware systems*, New York, NY, USA, 2005. ACM Press.

[80] A. Repenning. Collaborative diffusion: programming antiobjects. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 574–585, New York, NY, USA, 2006. ACM.

[81] A. Repenning. Excuse me, i need better ai! employing collaborative diffusion to make game ai child's play. In *Proceedings of the ACM SIGGRAPH Video Game Symposium*. ACM Press, 2006.

[82] M. Rinard, C. Cadar, and H. H. Nguyen. Exploring the acceptability envelope. In *OOPSLA '05: Companion to the 20th*

annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 21–30, New York, NY, USA, 2005. ACM Press.

[83] M. A. S. Sallem and F. J. da Silva e Silva. Adapta: a framework for dynamic reconfiguration of distributed applications. In *ARM '06: Proceedings of the 5th workshop on Adaptive and reflective middleware (ARM '06)*, page 10, New York, NY, USA, 2006. ACM Press.

[84] Y. Shoham. Agent-oriented programming. Technical report, Computer Science Department, Stanford University, Stanford, 1990.

[85] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[86] B. Simpson, editor. *Hsqldb User Guide*. The HSQLDB Development Group, 2007.

[87] K. Slonneger. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley, 1995.

[88] R. B. Smith, J. Maloney, and D. Ungar. The self-4.0 user interface: manifesting a system-wide vision of concreteness, uniformity, and flexibility. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 47–60, New York, NY, USA, 1995. ACM.

[89] L. Sterling and E. Shapiro. *The Art of Prolog, Second Edition: Advanced Programming Techniques*. MIT Press, 1994.

[90] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2 edition, 2004.

[91] K. C. Tsui and J. Liu. Multiagent diffusion and distributed optimization. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 169–176, New York, NY, USA, 2003. ACM.

[92] Unknown. Tutorial 3: A 5-step guideline for object-oriented design. http://www.universia.com.br/mit/1/100/PDF/slides-3.pdf.

[93] Unknown. *Apache XML-RPC API.* The Apache Software Foundation, 2001.

[94] Unknown. *Sun VirtualBox User Manual.* Sun Microsystems, 2009.

[95] F. J. Varela, H. R. Maturana, and R. Uribe. Autopoiesis: The organization of living systems, its characterization and a model. *BioSystems*, 5:187–196, 1974.

[96] S. Vinoski. Chain of responsibility. *IEEE Internet Computing*, 6(6):80– 83, 2002.

[97] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Languag.* Academic Press, 1986.

[98] B. Ward. *The Book of VMware: The Complete Guide to VMware Workstation.* No Starch Press, 2002.

[99] P. Wernick, T. Hall, and C. Nehaniv. Software evolutionary dynamics modelled as the activity of an actor-network. In *Second International IEEE Workshop on Software Evolvability*, pages 74–81. IEEE, 2006.

[100] J. Wielemaker. *SWI-Prolog 5.6 Reference Manual.* University of Amsterdam, 2008.

[101] Wikipedia. Autopoiesis. *Wikipedia - The Free Encyclopedia*, 2003.

[102] Wikipedia. Allopoiesis. *Wikipedia - The Free Encyclopedia*, 2005.

[103] D. Winer. *XML-RPC Specification.* UserLand Software, 1999.

[104] S. M. Yacoub. Composite filter pattern. Technical report, HP Laboratories Palo Alto, 2001.

[105] M. Zeleny. Self-organization of living systems: A formal model of autopoiesis. *International Journal of General Systems*, 4:13–28, 1977.