Harmony-Oriented Programming and Software Evolution

Sebastian Fleissner Elisa Baniassad

Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
{seb, elisa}@cse.cuhk.edu.hk

Abstract

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 even well-designed object-oriented programs can become brittle as they evolve, because their design has to be fixed at some point, and the more their implementation has progressed, the more difficult it becomes to adjust object interfaces and relationships.

We assert that the complexity of software evolution can be reduced by relaxing strong encapsulation and information hiding, and introducing concepts such as continuous information flow. These principles are captured in harmony-oriented programming, a paradigm inspired by concepts of Asian philosophy, such as harmony, resonance, and fields of interactions. This paper illustrates the constructs of harmony-oriented programming and several studies aimed at showing that, in comparison with traditional object-oriented programming, harmony-oriented programming is a more suitable approach for dealing with software evolution effectively.

Categories and Subject Descriptors D.3.3 [*Software / Programming Languages*]: Language Constructs and Features

General Terms Languages

Keywords Harmony, Resonance

1. Introduction

Object-oriented programming is strongly influenced by ideas of ancient Greek philosophy and thought. As described in [6], 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. Object-oriented software design and development strongly resembles the world view put forward by these Greek philosophers. It is common practice in object-oriented programming to isolate objects from their context and then describe them by their attributes (i.e. methods and instance variables).

It is generally accepted that even well-designed object-oriented programs can become more brittle as they evolve. 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, such design is non-trivial, as interfaces and object relationships have to be fixed at some point, and any subsequent change to the interface of one object can lead to many potential changes to dependent objects. As pointed out in [2], software evolution can eventually causes brittleness even in well-designed object-oriented programs.

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 [7], ancient Chinese philosophers considered the world as a mass of continuously interacting substances rather than a collection of discrete objects.

Our previous work [1, 4] introduces the conceptual idea and initial principles of harmony-oriented programming (HOP), a programming paradigm inspired by concepts of

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

Eastern thinking and reasoning, such as context and resonance. 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 1 illustrates important conceptual differences between harmony-oriented software programming and object-oriented programming (OOP).

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

Table 1. Object-Orientation and Harmony-Orientation

Harmony-oriented programming relaxes established and widely accepted object-oriented principles, such as strong encapsulation and information hiding, and favors more flexible and ah-hoc approaches for structuring and implementing programs.

1.1 Hypothesis

Software evolution is significantly affected by the following factors: Ease of changing the program's design, extensibility and maintainability of the program, quality feedback, and error recovery.

The hypothesis of this research is that in comparison to object-oriented programming, harmony-oriented programming improves the ease of dealing with some of these factors, and thus the ease of dealing with software evolution effectively. In particular, the hypothesis posits that harmony-oriented programming has the following advantages over traditional object-oriented programming:

1. Improved changeability: Fewer changes are required in order to reflect adjustments of a program's design in the code.
2. Improved extensibility and maintainability: Extending a program requires less effort (steps/changes).

2. Principles of Harmony Orientation

This section presents a revised and extended discussion of the principles of harmony-orientation originally introduced in [4]. 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 (figure 1), which are denoted as *balance*, *exposure*, *spaciality*, *information sharing*, and *information diffusion*.

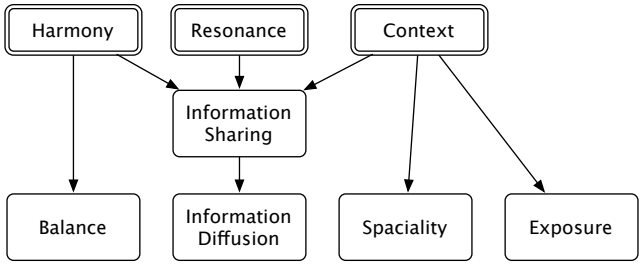

Figure 1. Principles of Harmony-Oriented Programming

2.1 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.

2.2 Exposure

The design of object-oriented programming and other programming languages is based on the principle of encapsulation. Unlike encapsulation, the exposure principle 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.

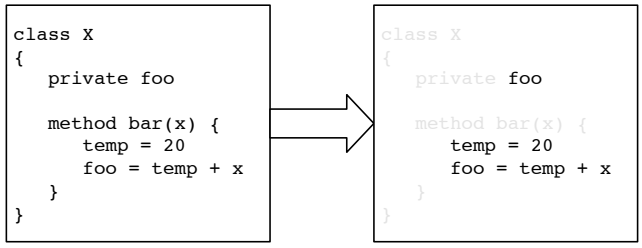

Figure 2. Exposure Principle

2.3 Spaciality

Every part of a program is assigned to one or more locations in a virtual space. Related parts of a program are positioned close to each other to form a specific context. For example, snippets that implement a user interface are placed in each others 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.

2.4 Information Sharing and Diffusion

The information sharing principle suggests that all data is shared between the pieces of a program. This principle facilitates the resonance concept, as one part of the program can

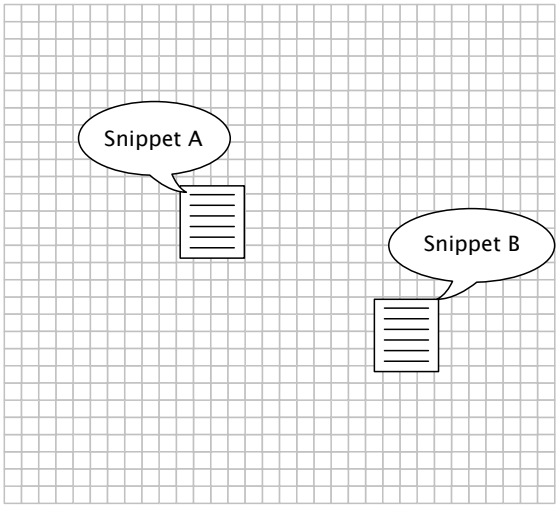

Figure 3. Spaciality Principle

react to changes made by any other part of the program. The information sharing principle is the basis for the information diffusion principle.

Diffusion is a gradual process in which a substance is spread over a space over time. The information diffusion principle 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 code positioning principles ensures that data generated by one code fragment (or the description of that data) reaches other code fragments that are located close within the virtual space first.

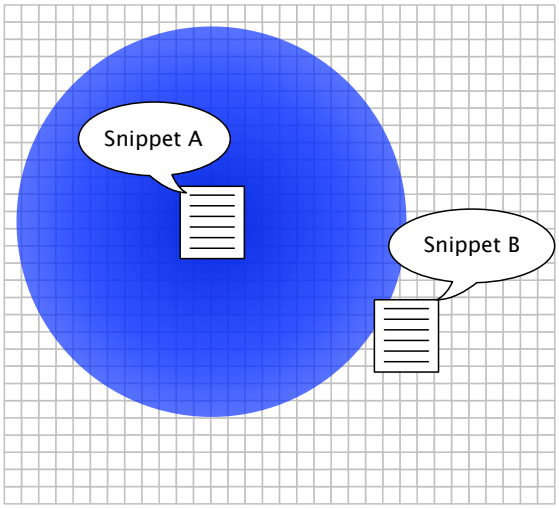

Figure 4. Information Diffusion Principle

3. 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 can not see or interact with other spatial constructs. Whenever a space receives data from a spatial construct, it automatically diffuses it. 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.

3.1 Spatial Constructs

Spatial constructs are program constructs that are assigned to a location in a spaces. 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, and observe data inside its location in the space.

The most important spatial construct is the *snippet*. 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.

3.2 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. Spaces can contain other spaces, and space hierarchies can be constructed to organize huge programs.

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.

3.2.1 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 5 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.

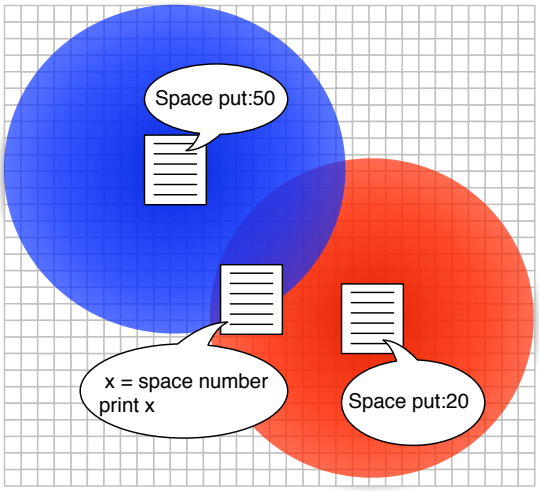

Figure 5. Substances and Diffusion

As shown in figure 5, 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 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 its 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.

4. Harmony-Oriented Smalltalk

Harmony-Oriented Smalltalk (HOS)[1] is a harmony-oriented runtime and visual development environment that allows programmers to implement harmony-oriented programs whose snippets are written in Squeak Smalltalk. The visual development environment is based on Morphic and provides programmers with tools for inspecting spaces, editing snippets, changing diffusion settings and debugging.

Since HOS is based on Smalltalk, 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 terms of spaces and spatial constructs, and not classes and objects, even though those are available.

5. Evaluation

This section presents studies comparing harmony-oriented and object-oriented programming in regard to factors affecting software evolution. The harmony-oriented parts of the studies are implemented using Harmony-Oriented Smalltalk and the object-oriented parts are implemented in plain Squeak Smalltalk.

5.1 Changeability Study: Relationships

Several object-oriented design patterns that facilitate dynamic relationships between objects, such as the Observer pattern, are proposed in [5]. 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.

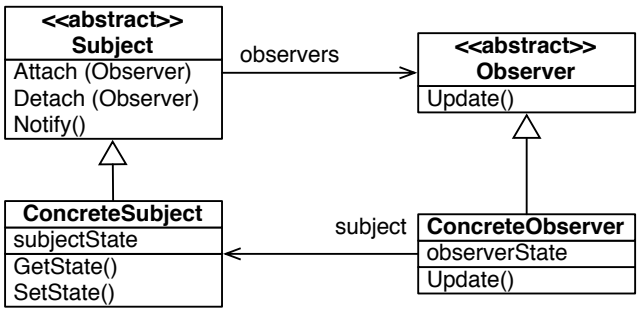

Figure 6. The Observer Design Pattern ([5])

Figure 6 illustrates the conceptual design of the Observer pattern. 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.

[1] http://www.squeaksource.com/hos.html

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.

Although the implementation of the Observer pattern is not very complex, its existence alone underlines the lack of mechanisms for defining relationships, other than static inheritance relationships, between objects in OOP. 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 each other in the space, and broken off by moving them apart from each other.

This study considers an implementation of a subject-observer relationship 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.

5.1.1 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 1 and 2 show the methods of the *AccountSubject* and *AccountObserver* classes. As shown in listing 1, the object-oriented implementation maintains its observers in an ordered collection. Listing 3 contains a Smalltalk script setting up instances of *AccountSubject* and *AccountObserver*, and changing the state of the subject multiple times.

5.1.2 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 3.1, 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 change the type of the snippet state to*Account* and then access it 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 7 illustrates the harmony-oriented implementation of the account subject-observer relationship and listings 4 and 5 show the code of the subject and observer snippets.

```
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|
10      observer update: account.].
11 balance: aNumber.
12     account balance: aNumber.
13     self notify.
14 withdraw: aNumber
15     account withdraw: aNumber. self notify.
16 deposit: aNumber
17     account deposit: aNumber. self notify.
```

Listing 1. Methods of *AccountSubject* class (OOP).

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

Listing 2. Methods of *AccountObserver* class (OOP).

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

Listing 3. Account subject and observer example (OOP).

In the harmony-oriented program shown in figure 7, the account subject-observer relationship is established already, since the substance 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.

5.1.3 Comparison

As the implementations in listings 4 and 5 show, it is not necessary to explicitly implement support for subject-observer

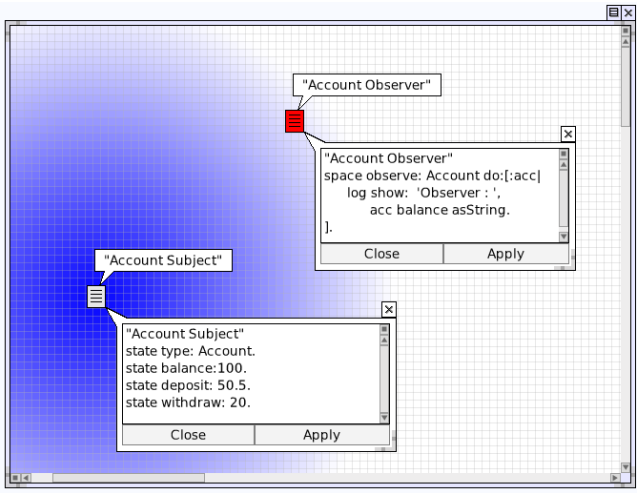

Figure 7. Account Subject and Account Observer (HOP)

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

Listing 4. Account subject snippet (HOP).

```
1 "Account Observer"
2 space observe: Account do:[:acc|
3     log show: 'Observer : ',
4         acc balance asString.
5 ].
```

Listing 5. Account observer snippet (HOP).

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: The subject class has to create a list for maintaining observers and provide methods for attaching, detaching, and notifying observers. Additionally, it has to manually invoke the observer's notification method each time its state changes. The observer class has to implement a method processing the updated state.

5.2 Extensibility and Maintainability Study

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 8 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.
- *"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*. New applications can be added by creating a new snippet implementing the desired functionality and placing it in the

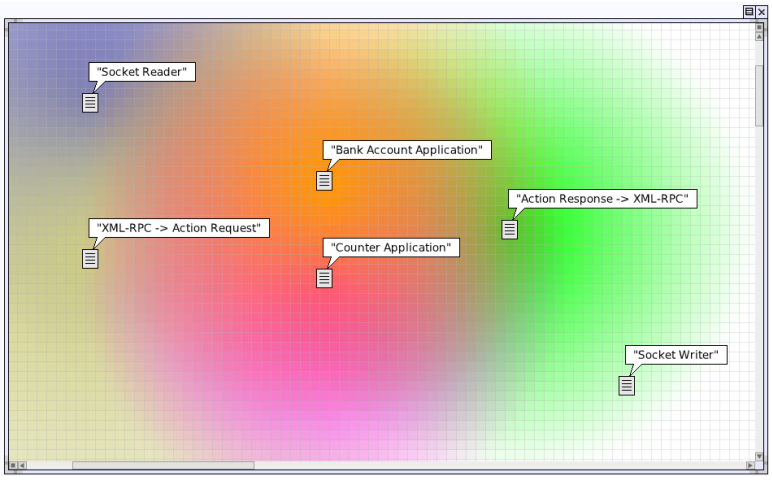

Figure 8. Harmony-Oriented Extensible Application Server

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 registering new protocols and applications.

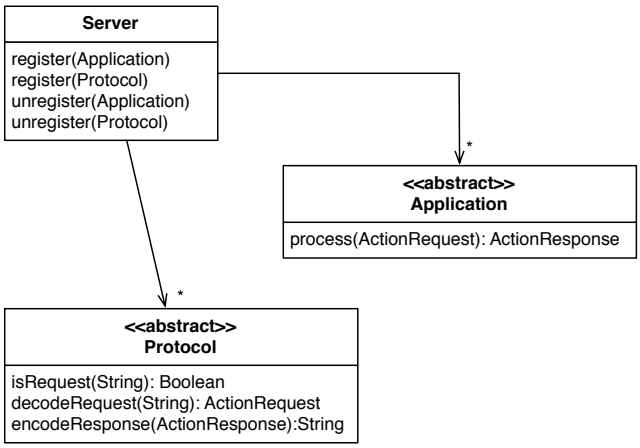

Figure 9. Minimal Object-Oriented EAS Design

One possible minimal object-oriented design is shown in figure 9. 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 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.

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. These changes include, but are not limited to:

- Defining a new abstract class for stream based applications, or changing the interface of the existing *Application* class. If the interface of the *Application* class is changed, all subclasses have to be adjusted as well.
- Changing the interface of the *Server* class to support registration of the new type of applications.
- Changing the logic of the *Server* to treat and process incoming data as a stream, if none of the *Protocol* classes can process it.

5.3 Summary

The previous sections 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 2 summarizes how factors affecting software evolution are fulfilled by harmony-oriented programming.

Factor	**Harmony-Oriented Programming**
Ease of change	Improved in comparison with OOP, as the structure of programs can be changed easily by moving snippets.
Extensibility	Improved in comparison with OOP, because new snippets can be added at runtime, and existing snippets do not have to be changed.
Maintainability	Improved in comparison with OOP, because snippets do not have any direct dependencies on each other.

Table 2. Study Results

6. Related Work

Harmony-oriented programming utilizes diffusion, a process in which substance or matter is spread over space over time, has been adapted in computer graphics [8] and multi-agent systems, such as [12], [10], [11], and [9]. For example, in [11] Repenning proposes collaborative diffusion as an agent-based artificial intelligence system for computer games.

On the surface, the harmony-oriented programming runtime environment appears to be similar to multi-agent systems, such as [11] . However, agent systems can be considered as a specific application domain while harmony-oriented programming is a new approach for implementing arbitrary programs. Snippets and agents are different from each other: Agents have features like autonomy, social ability, reactivity, goal-orientation, and adaptability. It is possible to write snippets that implement agent features, but as a conceptual construct, snippets are not comparable to agents.

Spreadsheet programming languages and languages inspired by spreadsheets, such as subtext [3], 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.

7. Conclusion

This paper presents principles and constructs of harmony-oriented programming and studies that illustrate the strengths of the harmony-oriented programming approach in the context of software evolution. The studies presented in this paper are only the first step towards evaluating and validating harmony-oriented programming. We expect further research to focus on the following areas: One area is to validate harmony-oriented programming in the context of large real world applications, such as application servers. The second area is work towards designing a pure harmony-oriented programming language and corresponding virtual machine.

References

[1] E. Baniassad and S. Fleissner. The geography of programming. In *OOPSLA 2006 Companion*, pages 560–573. ACM Press, 2006.

[2] O. Ciupke. Automatic detection of design problems in object-oriented reengineering. In *TOOLS '99 Proceedings*, page 18, 1999. IEEE Computer Society.

[3] J. Edwards. Subtext: uncovering the simplicity of programming. In *OOPSLA '05 Proceedings*, pages 505–518, 2005. ACM Press.

[4] S. Fleissner and E. Baniassad. Towards harmony-oriented programming. In *OOPSLA '08 Companion*, pages 819–822. ACM Press, 2008.

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

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

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

[8] M. J. Harris, G. Coombe, T. Scheuermann, and A. Lastra. Physically-based visual simulation on graphics hardware. In *HWWS '02 Proceedings*, pages 109–118, 2002. Eurographics Association.

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

[10] Y. Jiang, J. Jiang, and T. Ishida. Agent coordination by tradeoff between locally diffusion effects and socially structural influences. In *AAMAS '07 Proceedings*, pages 1–3, 2007. ACM.

[11] A. Repenning. Collaborative diffusion: programming antiobjects. In *OOPSLA '06 Companion*, pages 574–585, 2006. ACM.

[12] K. C. Tsui and J. Liu. Multiagent diffusion and distributed optimization. In *AAMAS '03 Proceedings*, pages 169–176, 2003. ACM.