Towards Harmony-Oriented Programming

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

Object-oriented programming and other programming paradigms are heavily influenced by Western thought and reasoning, which focuses on understanding the world in terms of categories, objects and their attributes: A typical program is decomposed into clearly defined units, such as modules, functions, objects, components and aspects, and each of these units is described by its properties, functionality, and direct relationships to other units. Eastern philosophy, however, focuses on fields of interactions rather than individual units. Harmony-oriented programming is a new programming paradigm based on concepts found in Eastern philosophy. This paper presents principles and constructs of harmony-oriented programming and introduces ongoing work towards creating a harmony-oriented software development environment for further experimental studies.

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

General Terms Languages

Keywords Asian Philosophy, Programming Paradigms

1. Introduction

Western reasoning and thought is influenced by several well-known ancient Greek philosophers, such as Aristotle, who posit the view that the world is a static and unchanging collection of objects that can be described and analyzed through categorization and formal logic. Attributes are used as the basis of categorization of objects, and the resulting categories are employed to construct rules governing the behavior of objects.

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.
ACM 978-1-60558-220-7/08/10.

Popular programming paradigms, especially object-oriented programming, resemble the worldview of such ancient Greek philosophers. It is common practice in object-oriented programming to isolate objects as much as possible from their context, and then describe them by their attributes (i.e. methods and instance variables); any program is decomposed into well-defined units with clear boundaries, such as modules, functions, objects, components and aspects. Each of these units is categorized and defined in terms of its attributes, functionality, and relationships to other units.

Because the focus is on individual program units, modeling dynamic and ad-hoc relationships is not always straightforward and often requires significant negotiation that has to be implemented explicitly. In addition, it is commonly accepted that large programs become more brittle as they evolve [2]. Interfaces that facilitate interaction between program units cannot easily be changed once the implementation of a program has commenced: A change in one unit's interface can result in many necessary adjustments to other units.

In contrast to the ancient Greek philosophers, Eastern philosophy does not focus on objects and their attributes, but rather considers the broad context and defines the world in terms of harmony, context, and resonance. As described in [4] and [9], ancient Chinese philosophers describe the world as a mass of continuously interacting substances rather than a collection of discrete objects. Each substance and every event in the world is considered to be related to every other event.

We first introduced the conceptual idea of a harmony-oriented programming (HOP), a programming paradigm inspired by concepts of Eastern thinking and reasoning, as part of a study [1] that explored how Easterners interpret a typical object-oriented scene. Our hypothesis is that harmony-oriented programming facilitates development of dynamic and constantly evolving systems and has the potential to overcome brittleness caused by software evolution. This paper presents our work towards developing this conceptual idea into a concrete programming paradigm. In particular, we introduce principles and constructs of harmony-oriented programming and provide a description of the anatomy harmony-oriented programs. In addition, we present ongoing work on a virtual machine and development environment for harmony-oriented programming, which will be used for testing our hypothesis.

2. Principles of Harmony-Orientation

Harmony, resonance, and context are three key concepts found in East Asian (in particular Chinese) philosophy. These three concepts are the basis of the principles of harmony-oriented programming, which are denoted as *balance*, *code exposure*, *code positioning*, *information sharing*, and *information diffusion*. Figure 1 illustrates the relationship of the three key concepts with the principles of harmony-oriented programming.

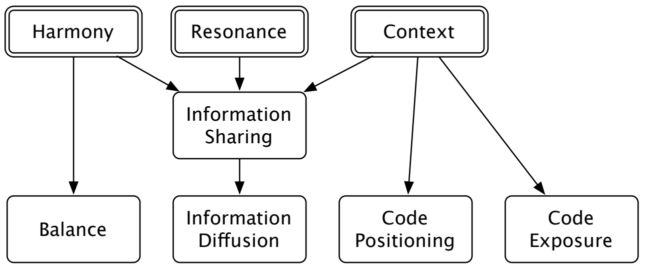

Figure 1. Principles of Harmony-Oriented Programming

Balance: This 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 during runtime, which is achieved when any data produced by one part of the program is consumed by one or more other parts of the program.

Code Exposure: The design of object-oriented programming and other programming languages is based on the principle of encapsulation. Unlike encapsulation, the code exposure principle suggests decomposing a program into pieces, denoted as *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.

Code Positioning: 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. Certain snippets can be assigned to both example contexts to serve as a bridge between user interface and business logic.

Information Sharing: All data is readable (or can be queried) by any part of the program. Information Sharing facilitates the resonance concept, as one part of the program can react to changes made by any other part of the program. The information sharing principle is the basis for the information diffusion principle.

Information Diffusion: Data or a description of the data generated by any part of the program is diffused throughout the virtual program space during runtime. 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 snippet (or the description of that data) reaches other snippets that are located close within the virtual space, and hence related, first.

3. Harmony-Oriented Programs

A harmony-oriented program is a collection of *snippets* assigned to locations in one or more *fields*. A field is a two-dimensional virtual space that serves as the runtime environment of the harmony-oriented program. During runtime, the field owns and contains all data of the program and facilitates data exchange between snippets via diffusion. The following sections explain the constructs of harmony-oriented programs in more detail using the example of a simple calculator application. This calculator application consists of a window allowing input of calculations to be performed, and an implementation that performs basic arithmetic operations. Figure 2 shows a possible object-oriented design of the simple calculator application for reference.

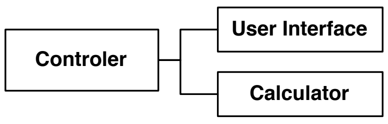

Figure 2. Object-Oriented Calculator Application

3.1 Snippets

As the principle of code 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. For example, the calculator application could be split into a snippet implementing the input window and one or more snippets for performing arithmetic calculations. Snippets can be implemented using imperative or functional programming languages. However, snippets in a harmony-oriented program do not "own" any variables (data). Any variable declared inside a snippet is created and instantiated inside the field (or fields) at a location next to the snippet during runtime. In addition, every access to a variable inside a snippet results in an access to the relevant variable inside the field. Apart from indirectly producing data in the field by declaring and modifying variables, a snippet can scan its immediate vicinity in the field for additional variables and announce its desire to consume specific data. Announcing desire to consume specific data can be done implicitly, for example by writing a snippet that operates on data it does not initialize, or explicitly.

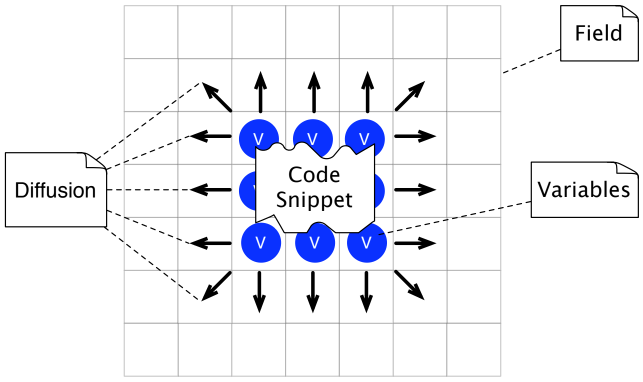

Figure 3. Snippet with Data/Variables

3.2 Variables and Type System

Harmony-oriented programming uses a structural type system and supported types include common primitive and composite data types found in various programming languages. Like snippets, variables have a specific location in the field. Whenever a snippet declares (or instantiates) a variable of a certain type, this variable is created inside the field at a position in the direct vicinity of the snippet. Figure 3 illustrates a snippet and its data in the surrounding field.

3.3 Substances and Diffusion

A *substance* is the combination of a data type description, an intensity, and optionally a value of the described data type. Substances are used by the field to diffuse information on available variables and data desired by snippets. The field generates a new substance with full intensity every time a variable is created or a snippet announces its desire for a certain data, and places it into the same virtual position as the variable or snippet. A substance describing a variable is denoted as a *positive substance* and contains a description of the variable's data type and the variable's value. A substance describing a data request is denoted as a *negative substance* and only contains a description of the desired data.

After the field has created a new substance, it diffuses it from its origin further into the two-dimensional virtual space. This diffusion process is indicated by the arrows shown in figure 3 and a concrete example for the calculator application is shown in figure 4. The further a substance is diffused, the lower its intensity. Because of the diffusion process, two or more substances can meet each other. If substances meet and the field detects that one of the substances represents availability of a certain variable and the other substance represents desire for the same or similar data, the field puts a copy of the variable described by the former substance next to the snippet where the latter substance originates. The arithmetic snippet in figure 4 might implement an arithmetic operation using a composite data type as input that is equivalent to the data type of the variable declared by the user interface snippet. Since no data is available in the arithmetic snippet, the field generates a negative substance. As figure 4 illustrates, this negative substance is diffused and meets with the positive substance generated by the variable holding the input of the user interface snippet.

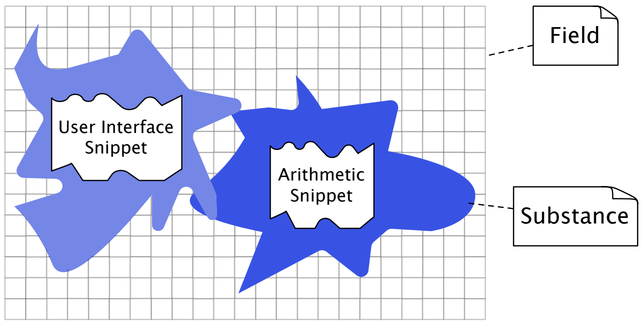

Figure 4. Harmony-Oriented Calculator Application

4. Runtime and Development Environment

As stated in the introduction, our hypothesis is that harmony-oriented programming facilitates dynamic and constantly evolving systems and has the potential to overcome brittleness caused by software evolution. In order to test our hypothesis, we are working on a virtual machine and interactive visual development environment called *Yiktai*. The description in this section is a brief summary of features under development.

4.1 Virtual Machine

The virtual machine provides the runtime environment for harmony-oriented software. An instance of the virtual machine runs as a service that can be accessed and configured through the XML-RPC protocol. The virtual machine is designed to expose the following core functionality:

- Creation, modification, import and export of fields and snippets.
- Monitoring of fields and performance.
- Import and export of field configurations and snippets.

Since no harmony-orientation specific language has been designed yet, the virtual machine can be configured to support various existing programming languages for implementing snippets by means of plug-ins. The initial version of the VM supports Lua for implementation of snippets.

4.2 Development Environment

Yiktai is a visual integrated development environment. Because of the interactive nature of harmony-oriented programming, the development environment requires a virtual machine in order to run. The development environment is designed to provide the following main features:

- Configuration and real-time visualization of the runtime environment.
- Creation and modification of fields and snippets.

5. Related Work

Phenotropics: Lanier proposes phenotropics as an alternative to argument-based interfaces in [8]. 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.

Diffusion: Harmony-oriented programming utilizes diffusion, a process in which substance or matter is spread over space over time, has been adapted in computer graphics [5] and multi-agent systems, such as [11], [7], [10], and [6]. For example, in [10] Repenning proposes collaborative diffusion as an agent-based artificial intelligence system for computer games.

Agent Systems: On the surface, the harmony-oriented programming runtime environment appears to be similar to multi-agent systems, such as [10] . 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.

Spreadsheets: 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.

6. Conclusion

In this paper we introduced our work towards realizing harmony-oriented programming, a new programming paradigm inspired by Asian philosophy.

Harmony-oriented programming can overcome problems caused by inflexible interfaces and explicit negotiations between program parts, because it is based on the principles of code exposure and code positioning. The benefits of code positioning can be demonstrated by considering the observer pattern whose implementation establishes a listening relationship between a subject and an observer. An object-oriented implementation of the observer pattern requires explicit negotiations, such as registering and unregistering observer objects with a subject object. In a harmony-oriented implementation it is sufficient to place an observer snippet next to a subject snippet. No explicit negotiation for establishing a subject-observer relationship is necessary.

Since this paper describes preliminary work, we focus on principles and constructs of harmony-oriented programming and give a short overview on the features of a virtual machine and visual programming environment currently under development. After completing the virtual machine and visual development environment, our research will focus on empirical studies aimed at validating the hypothesis that harmony-oriented programming facilitates development of dynamic and constantly evolving systems and has the potential to overcome brittleness caused by software evolution.

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. IEEE Computer Society, 1999.

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

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

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

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

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

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

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

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

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