Evolvable Programming based on Collaboration-Field and Role Model

Tetsuo TAMAI
Interfaculty Initiative in Information Studies
The University of Tokyo
3-8-1 Komaba, Meguro-ku
Tokyo 153-8902, Japan
tamai@graco.c.u-tokyo.ac.jp

ABSTRACT

This is a brief introduction to our research on a collaboration field and role model aiming to support evolvable software design and programming.

1. INTRODUCTION

Software evolution is so challenging a them that various approaches are conceivable to attack the problem. Promising approaches include: 1) observe evolution processes and find patterns or laws governing software evolution; 2) design computational models or languages that support development of evolvable software. We have been conducting research on both of these approaches. Some results of the former approach have been published [20, 12], including a short paper submitted to this workshop [19].

This talk is concerned with the latter approach, highlighting the model comprising of fields of collaboration and roles played in the fields [21, 22, 23].

2. MOTIVATION

Evolution is caused by a number of factors. One of the most effective ways of comprehending the evolution mechanism is to see it in relation with environments. Environment changes trigger evolution and software evolves to adapt to the change of its environment. In this sense, evolution is strongly related with adaptation.

It is an ultimate target of AI, specifically that of intelligent agent systems, to realize autonomously adaptable software. It is a long way to go to achieve that goal and even when it is realized in the distant future, there will remain the problem of how to control such software systems.

Our goal is a little more modest, i.e. to build a computational model that is flexible enough to cope with future changes but also expressive enough to explicitly show design intentions and allow tracing of control.

To explain motivation for our work, we give some typical examples introduced by other researchers to illustrate their works. We share similar objectives and the difference of approaches will become clear by handling the same problems.

Honda et al. [6] gives an example of adaptation. A woman Hanako, modeled as an object, marries with Taro and adapts to the environment *family*. She then gets employed as a researcher by a research laboratory and adapts to the environment *laboratory*. The adaptation should be made dynamically, thus it can be regarded as a kind of evolution. At the same time, the object Hanako should preserve its identity when she enters a new environment like the lab or even after she quits the lab for some reason.

In Honda et al.'s model *Morphe*, suppose an object (e.g. Hanako) enters an environment (e.g. a laboratory) and assumes a role (e.g. a researcher), then the object acquires a new set of attributes and behaviors or alters some of the attributes and behaviors already possessed by the object through following transformation rules associated with the role. This strategy of employing transformation rules must have been adopted mainly because the underlying language of their work was a constraint based object-oriented language.

M. Fowler [1] gives an example of personnel roles in a company to be assumed by employees. He lists up engineers, salesmen, directors and accountants as roles and put a question how to deal with situations such that a person plays more than one role or a person changes his or her role in the lifetime. The latter is a case of object evolution. He shows several patterns that solve this problem and gives a generic name *role pattern*.

E. Kendall [7] gives an example of the bureaucracy pattern. There are five roles in the pattern: Director, Manager, Subordinate, Clerk and Client. A client deals with a clerk. Manager and Subordinate are subclasses of Clerk. A manager supervises subordinates and reports to a director. Although evolution is not explicitly treated in Kendall's paper, this situation possibly raises the case that a person evolves from the role of Subordinate to Manager or ultimately to Director.

3. COLLABORATION FIELD AND ROLE MODEL

3.1 Design principles

Our model called Epsilon is based on the constructs of collaboration fields and roles interacting each other within those fields. Our basic design principles are as follows.

Support adaptive evolution In our model, objects evolve by participating in a collaboration field and assumes a role in the field. Participation can be made dynamically and quitting the field is also dynamically allowed. An object is free to belong to multiple fields at a time.

Describe separation of concerns Each collaboration field represents a concern so that separation of concerns is explicitly supported by the model. Interrelation of concerns are realized through objects assuming roles of different collaboration fields.

Advance reuse Besides objects, collaboration fields including roles can be units of reuse. Moreover, since collaboration fields and roles are given the status of first class constructs in a proposed programming language, design patterns can be reused directly at the programming level components.

Figure 1 illustrates an example of Contract Net Protocol [17]. It is a protocol to solve a problem collaboratively through negotiation of multiple processing nodes. A contract may be given by a manager to a contractor who has bidden the lowest price. A node can be a manager of one contract and a contractor of another. This problem can be conveniently modelled by Epsilon; a contract is represented by a context and a manager and contractor(s) by roles.

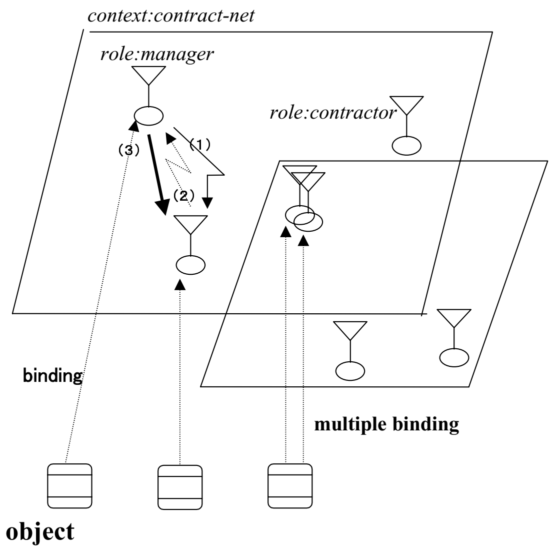

Figure 1: Contract Net Protocol

3.2 Language

Our language also named Epsilon has the following constructs to support the above mentioned model features.

Declaration of collaboration fields and roles In our language, collaboration fields are called "contexts". `Context` and `role` are declared with attributes and methods just like object classes. Declaration of `role` is placed inside of `context` declaration, similar to inner classes of Java. Instances of contexts and roles are created dynamically.

Encapsulation of roles in fields As declaration of roles is confined in a context, their interaction is encapsulated within the context. Roles in a context can communicate with each other but cannot access to other contexts and roles in other contexts directly. Collaboration is naturally described on instance bases. When a context instance is created, a unique instance of each role of the context is created at the same time. Each role instance can be referred by the role name qualified by the context instance *id* but it is also possible to generate multiple role instances of the same role types (role in this case is regarded as a class or a template).

Below is an example to show how `context` and `role` are declared and collaboration between roles are described.

```
context Company {
  role Employer {
    int salary = 100;
    void pay() {
      Employee.getPaid(salary);
    }
  }
  role Employee {
    int deposit;
    void getPaid(int salary) {
      deposit += salary;
    }
  }
}
```

Binding of objects with roles An object can be dynamically bound to a role of a context and can be unbound later. An object may be bound to multiple roles of different contexts. When an object is bound to a role, it can make access to other roles of the same context and conversely can be accessed by other roles through the interface of the binding role. The attributes of the combined object and role are merged and so are the methods but there is a replacing (or renaming) convention that enables an attribute or a method of the role to be regarded the same as a designated attribute or a designated method of the combined object, respectively.

Below is an example showing how an object is bound to a role. A method `bind(Object o)` is defined to all roles, meaning to bind the Object *o* to the role itself. It can be interpreted that the declared role class inherits the interface of class `Role`, including `bind`, `unbind`, and other methods.

```
class Person {
  int money;
}
Person Tanaka = new Person();
Company todai = new Company();
tadai.Employee.bind(Tanaka)
  replacing Employee.deposit
  with Tanaka.money;
```

As collaborations are explicitly described and encapsulated as contexts in Epsilon, they can be reused as program components. Thus, design patterns of Gamma et al. [2], for example, are good targets for building reusable program components and they will be used not just as a catalogue of design know-how's but reusable components.

An example of defining the pattern "Mediator" and using it as a component is shown in the appendix.

A preliminary version of an Epsilon compiler was implemented on ABCL/R3, a reflective concurrent object-oriented language [11]. We have also designed EpsilonJ, a language with Java like syntax as shown above, and are on the way of implementing it as a translator to Java.

4. RELATED WORKS

There are a number of works on "Role Models" [18]. A typical example is the OOram methodology [14], which not only defines role models but also integrates them by the step of role model synthesis. Design patterns can also be regarded as describing patterns of collaboration. The works of D. Riehle further extend this view and employ the notion of role modeling to model object migration [25] and to design composite patterns [15] and frameworks [16]. Also related is the work by B. Kristensen et al. [10].

Gottlob et al. [3] deals with dynamic change of objects (but since their main concern is data base, objects are more like data base schemas) using the concept of roles. They claim that inheritance is class based and thus inconvenient for handling dynamic changes. Instead, they propose a role hierarchy and realize specialization and inheritance at the instance level.

Also related is the notion of *contracts*. Contracts, proposed by R. Helm et al. [5] is a construct for the explicit specification of behavioral compositions. A contract defines a set of communicating participants and their contractual obligations. This notion of *participants* correspond to roles but participants are actually objects and thus the separation of objects and roles are somewhat blurred.

In these methodologies, roles play an important part at the phases of analysis and design but usually become invisible in the implementation. However, there are some works that aim at preserving roles explicitly in programs. For example, VanHilst and Notkin [24] used class templates of C++ to implement roles. One of the objectives of this proposed method is to reuse roles besides or even in stead of objects.

There are many other works that by and large share motivations as described above but take different approaches for solutions. Notable ones are subject oriented programming and aspect oriented programming. They share the notion that models or systems can be grasped differently by views. The former calls the view *subject* and the latter *aspect*.

Subject Oriented Programming

Harrison & Ossher [4] states that their goal is "to facilitate the development and evolution of suites of cooperating applications." They specifically emphasize that the same object would be seen differently by "subjects" and yet there should be coherent intrinsic properties inside the object. They propose some probable methods for reconciling various views. The way they see cooperation in this framework is by sharing an object and explicit collaborations as discussed in Section 3 are not necessarily intended.

Their work has been extended to "multi-dimensional separation of concerns" [13].

Aspect Oriented Programming

Kiczales et al. [9] claims that a system modularization structure designed from one aspect is often in conflict with modularization from another aspect. Thus, they propose a method of describing aspects separately and then weaving them together to obtain a consolidated system. They implemented a language AspectJ to provide a general mechanism for writing aspects and weaving them together [8].

Compared to these related works, our approach of binding objects and roles have the following characteristics:

1. Composition takes place when an object instance and a role instance are bound together;
2. An object instance can be bound to multiple role instances residing in different contexts;
3. As a role is also an instance it has its own state as well as its own set of methods and preserve the state even after the separation from its pair object;
4. The state of an object and that of a role construct a Cartesian product state after composition;
5. A method of an object and a method of a role can be overridden or renamed by another method of the counterpart role or object and thus interaction between the object and role states is made possible;
6. The above mechanism indicates that the binding of an object and a role can be bi-directional as opposed to the unidirectional relation of delegation.

5. REFERENCES

[1] M. Fowler. Dealing with roles. http://www2.awl.com/cseng/titles/0-201-89542-0/apsupp/. supplemental information to *Analysis Pattern*, Addison-Wesley, 1997.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[3] G. Gottlob, M. Schrefl, and R"ock. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3):268–296, July 1996.

[4] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *OOPSLA '93*, pages 411–428, 1993.

[5] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *ECOOP/OOPSLA '90 Proceedings*, pages 169–180, October 1990.

[6] Y. Honda, S. Watari, and M. Tokoro. Compositional adaptation: A new method for constructing software for open-ended systems. *Computer Software*, 9(2):122–136, 1992. in Japanese.

[7] E. A. Kendall. Role model designs and implementations with aspect-oriented programming. In *OOPSLA' 99*, pages 353–369, Nov. 1999.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with aspectj. *CACM*, 44(10):59–65, Oct. 2001.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming(ECOOP), Finland*. Springer-Verlag, June 1997.

[10] B. B. Kristensen and K. Osterbye. Roles: Conceptual abstraction theory and practical language issues. *Theory and Practice of Object Systems*, 2(3):143–160, 1996.

[11] H. Masuhara, S. Matsuoka, and A. Yonezawa. Implementing parallel language constructs using a reflective object-oriented language. In *Reflection Symposium '96*, pages 79–91, Apr. 1996.

[12] T. Nakatani and T. Tamai. Empirical observations on object evolution. In *Asia-Pacific Software Engineering Conference (APSEC'99)*, pages 2–9, Takamatsu Japan, Dec. 1999.

[13] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *CACM*, 44(10):43–50, Oct. 2001.

[14] T. Reenskaug, P. Wold, and O. Lehne. *Working with Objects: the OOram Software Engineering Method*. Manning Publications, Greenwich, 1996.

[15] D. Riehle. Composite design patterns. In *OOPSLA '97*, pages 218–228, Oct. 1997.

[16] D. Riehle and T. Gross. Role model based framework design and integration. In *OOPSLA '98*, pages 117–133, Vancouver, Oct. 1998.

[17] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Trans. on Computers*, 29(12):1104–1113, 1980.

[18] T. Tamai. Objects and roles: modeling based on the dualistic view. *Information and Software Technology*, 41(14):1005–1010, 1999.

[19] T. Tamai. Analysis of software evolution processes using statistical distribution models. In *International Workshop on Principles of Software Evolution (IWPSE'02)*, Orlando, Florida, May 2002. ACM.

[20] T. Tamai and T. Nakatani. An empirical study of object evolution processes. In *International Workshop on Principles of Software Evolution (IWPSE'98)*, pages 33–37, Kyoto, Oct. 1998.

[21] N. Ubayashi and T. Tamai. An evolutional cooperative computation based on adaptation to environment. In *Proc. Asia Pacific Software Engineering Conference '99*, pages 334–341, Takamatsu, Japan, Dec. 1999. IEEE Computer Society.

[22] N. Ubayashi and T. Tamai. RoleEP: Role based evolutionary programming for cooperative mobile agent applications. In *International Symposium on Principles of Software Evolution*, pages 232–240, Kanazawa, Japan, Nov. 2000. IEEE Computer Society.

[23] N. Ubayashi and T. Tamai. Separation of concerns in mobile agent applications. In *Proceedings of the 3rd International Conference REFLECTION 2001, LNCS 2192*, pages 89–109, Kyoto, Sept. 2001. Springer.

[24] M. VanHilst and D. Notkin. Using Role Components to Implement Collaboration-Based Designs. In *OOPSLA '96*, pages 359–369, 1996.

[25] R. Wieringa, W. de Jonge, and P. Spruit. Using dynamic classes and role classes to model object migration. *Theory and Practice of Object Systems*, 1(1):61–83, 1995.

APPENDIX

We write an example of Mediator Pattern as described in the GOF design pattern book [2]. Here, a (static) role method `newBind` is used. It is similar to `bind` except that it creates a new instance of the role and bind the object passed as an argument to the role.

```
context MediatorPattern {
  role Mediator {
    void notify(Colleague a) {
    }
  }
  role Colleague {
    void raiseNotification() {
      Mediator.notify(this);
    }
  }
}
class FontDialogueDirector {
  MediatorPattern pattern;
  ListBox fontList;
  EntryField fontName;
  Button ok;
  Button cancel;
  FontDailogueDirector() {
    pattern = new MediatorPattern();
    fontList = new ListBox();
    fontName = new EntryField();
    ok = new Button();
    cancel = new Button();
    pattern.Colleague.newBind(fontList)
        renaming fontList.Changed
        by Colleague.raiseNotification;
    pattern.Colleague.newBind(fontName)
        renaming fontName.Changed
        by Colleague.raiseNotification;
    pattern.Colleague.newBind(ok)
        renaming ok.Changed
        by Colleague.raiseNotification;
    pattern.Colleague.newBind(cancel)
        renaming cancel.Changed
        by Colleague.raiseNotification;
    pattern.Master.bind(this)
        renaming Master.notify
        by this.WidgetChanged;
  }
  void WidgetChanged(Widget theChangedWidget) {
    if (theChangedWidget==fontList) {
        fontName.SetText(fontList.GetSelection());
    } else if (theChangedWidget==ok) {
        // apply font change and dismiss dialog
```

```
         // ...
      } else if (theChangedWidget==cancel) {
         // dismiss dialog
      }
   }
}
```