

University of Windsor

## Scholarship at UWindor

---

Electronic Theses and Dissertations

Theses, Dissertations, and Major Papers

---

2004

### Object-oriented programming in C# with dynamic classification.

Wenjiang Wang  
*University of Windsor*

Follow this and additional works at: <https://scholar.uwindsor.ca/etd>

---

#### Recommended Citation

Wang, Wenjiang, "Object-oriented programming in C# with dynamic classification." (2004). *Electronic Theses and Dissertations*. 2900.  
<https://scholar.uwindsor.ca/etd/2900>

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email ([scholarship@uwindsor.ca](mailto:scholarship@uwindsor.ca)) or by telephone at 519-253-3000ext. 3208.

**OBJECT-ORIENTED PROGRAMMING IN C#**  
**WITH**  
**DYNAMIC CLASSIFICATION**

by  
Wenjiang Wang

A Thesis  
Submitted to the Faculty of Graduate Studies and Research  
through Computer Science  
in Partial Fulfillment of the Requirements for  
the Degree of Master of Science at the  
University of Windsor  
Windsor, Ontario, Canada  
2004

© 2004 Wenjiang Wang



Library and  
Archives Canada

Bibliothèque et  
Archives Canada

Published Heritage  
Branch

Direction du  
Patrimoine de l'édition

395 Wellington Street  
Ottawa ON K1A 0N4  
Canada

395, rue Wellington  
Ottawa ON K1A 0N4  
Canada

*Your file* *Votre référence*

*ISBN: 0-612-96127-3*

*Our file* *Notre référence*

*ISBN: 0-612-96127-3*

The author has granted a non-exclusive license allowing the Library and Archives Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

---

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

# Canada

## **Abstract**

Object-oriented programming language has gained popularity in recent years. However, some problems exist in object-oriented programming languages. It works well with static classification, but does not support object dynamic classification. Static classification means an object always and only belongs to one class during its life spans. In real-world applications, objects may belong to different classes rendering different roles certain times during the lifetime. Dynamic classification enables the changing of object classification over time. Objects can be classified and declassified into/from acquire and release class membership during runtime.

In this thesis, many approaches to dynamic classification will be discussed in different implementing languages. Based on the thorough reviews of these approaches, we give a new approach. This approach combines the concept of object and roles and extends a class hierarchy with dynamic classification. The syntax of dynamic classification shows how to implement the function of dynamic classification in the object-oriented programming language. Finally, we present a preprocessor, by which a C# code including the extendable dynamic classification functions can be translated to standard C# code.

### **Keywords:**

object-oriented, dynamic classification, role, object hierarchy, class hierarchy, object migration

*To*  
*My Family and Friends*

## **Acknowledgements**

I would like to take this opportunity to thank many people who encouraged and supported me to complete this thesis.

I would like to thank Dr. Liwu Li, my advisor, for his valuable direction and comments. I appreciate that he spent so much time with me on my research.

I would also like to thank my committee members, Dr. Sang-Chul Suh and Dr. Jianguo Lu, for spending time to give me directions and opinions which help me to improve my thesis. I want to specially thank Dr. Bubaker Boufama for being chair in my thesis committee.

Finally, I would love to thank my family and friends for their love and support.

## Table of Contents

<b>ABSTRACT</b> .....	<b>III</b>
<b>DEDICATION</b> .....	<b>IV</b>
<b>ACKNOWLEDGEMENTS</b> .....	<b>V</b>
<b>LIST OF FIGURES</b> .....	<b>VIII</b>
<b>LIST OF TABLES</b> .....	<b>IX</b>
<b>CHAPTER 1 INTRODUCTION</b> .....	<b>1</b>
<b>CHAPTER 2 OBJECT-ORIENTED PROGRAMMING AND DYNAMIC CLASSIFICATION</b> .....	<b>4</b>
2.1 OBJECT-ORIENTED PROGRAMMING .....	4
2.1.1 Object-Oriented Technology .....	4
2.1.2 Object-Oriented Programming Concept .....	5
2.1.3 Object-Oriented Programming Language .....	7
2.1.4 Object-Oriented Programming Design and Analysis .....	8
2.2 DYNAMIC CLASSIFICATION .....	9
2.2.1 Conventional Object-Oriented Model.....	10
2.2.2 Concept of Role .....	11
2.2.3 Patterns of Role.....	12
2.2.4 Class Hierarchy .....	15
<b>CHAPTER 3 RELATED RESEARCHES</b> .....	<b>18</b>
3.1 EXTENDED SMALLTALK.....	18
3.2 DoR.....	20
3.3 FIBONACCI .....	21
3.4 ASPECT PROGRAMMING.....	21
3.5 PROTOTYPE-BASED LANGUAGE.....	23
<b>CHAPTER 4 PROGRAMMING WITH DYNAMIC CLASSIFICATION</b> .....	<b>25</b>
4.1 EXTENDED CLASS HIERARCHY .....	25
4.2 SYNTAX OF DYNAMIC CLASSIFICATION .....	31
4.2.1 Class Definition .....	31
4.2.2 Statement Definition .....	34
<b>CHAPTER 5 IMPLEMENTING DYNAMIC CLASSIFICATION IN C#</b> .....	<b>37</b>
5.1 FEATURES OF THE C# LANGUAGE.....	37
5.2 IMPLEMENTATION OF PREPROCESSOR .....	42
5.2.1 The First Process of the Preprocessor .....	42
5.2.2 The Second Process of the Preprocessor .....	45
5.3 RESTRICTIONS AND ASSUMPTIONS OF SYNTAX.....	59

5.4 HOW TO USE THE PREPROCESSOR.....	61
<b>CHAPTER 6 CONCLUSIONS.....</b>	<b>64</b>
6.1 CONCLUSIONS.....	64
6.2 FUTURE WORK .....	65
<b>REFERENCE.....</b>	<b>66</b>
<b>VITA AUCTORIS .....</b>	<b>69</b>



## List of Figures

FIGURE 2.2.1 A PERSON'S LIFE TRACK.....	9
FIGURE 2.2.3.1 STRUCTURE OF PATTERNS .....	12
FIGURE 2.2.3.2 USING ROLE OBJECTS FOR PERSON .....	13
FIGURE 2.2.3.3 TREAT THE ROLE AS A RELATIONSHIP.....	14
FIGURE 2.2.4.1 A CLASS HIERARCHY .....	15
FIGURE 2.2.4.2 CLASS INSTANTIATION IN A CLASS HIERARCHY .....	16
FIGURE 2.2.4.3 EVOLVING AN OBJECT IN CLASS HIERARCHY .....	16
FIGURE 3.1.1 A CLASS AND ROLE HIERARCHY .....	19
FIGURE 3.4.1 AN OBJECT AND ITS ROLES: INTRINSIC AND EXTRINSIC MEMBERS.....	22
FIGURE 4.1.1A AN OBJECT HIERARCHY .....	26
FIGURE 4.1.1B A CLASS HIERARCHY .....	26
FIGURE 4.1.2 AN EXTENDED CLASS HIERARCHY .....	28
FIGURE 4.1.3 IMPLIED DYNAMIC PARENT-CHILD RELATION .....	28
FIGURE 4.1.4 AN EXTENDED CLASS HIERARCHY WITH DYNAMIC CLASSIFICATION.....	29
FIGURE 4.2.1.1 RELATIONS OF THREE CLASSES .....	33

## List of Tables

TABLE 2.1.1.1 HISTORY OF OOT .....	4
TABLE 3.1.1 CLASS METHODS OF ROLETYPE .....	20
TABLE 3.1.2 CLASS METHODS OF QUALIFIEDROLETYPE.....	20
TABLE 5.2.1.1 DYNAMIC_CLASS_INFO. DAT STRUCTURE.....	43
TABLE 5.3.1.2 CLASS_INFO. DAT STRUCTURRE.....	45
TABLE 5.2.2.1 CLASS METHODS OF DYNAClassIFICATION.....	47
TABLE 5.3.1 VARIABLE TYPES.....	60

## Chapter 1 Introduction

Object migration is the phenomenon when a real-world entity is classified or declassified during its lifetime. For example, one person becomes a student first and changes into an employee later. The person belongs to two class types, and plays the role of student or employee, or student and employee. The person is not changed, but his role is changed. When the person is a student, he has the character of a student. When he is employed, he owns the character of the employee. In the real world, different kinds of entities are classified into different classes over time. For example, a frog belongs to water-living type animal at first, and then belongs to amphibian type. A company document belongs to secret archive at one time, and belongs to general archive later. From the above discussion, we get the definition of dynamic classification, which means an object may change its class membership at the run-time.

Object-oriented programming is an important concept, which is widely used in different fields of computer science research and software industry. It is a kind of method, which uses object-oriented concepts effectively and systematically to develop programs. It can thoroughly describe the real-world problems and resolve them. The traditional object-oriented programming has shortcomings when it tries to implement object with dynamic classification in the real world. Most of the existing object-oriented programming languages such as Smalltalk, Java and C# are static type, supporting single and static classification. In these languages, an object always and only belongs to its class and possesses the attributes and methods of this class. It cannot be changed into other classes in its lifetime.

The study on the dynamic classification may count back to the early 1977, when Bachman and Daya [BD77] presented the concept of the role model to extend the

traditional network model. This approach establishes the foundation for the later research. With continually comprehensive and in-depth research, the researchers proposed many approaches to challenge the theory and implement the dynamic classification from several different perspectives. Although researchers name the dynamic classification with different titles, but the meaning are almost same and consistent. For example, Wieringa et al. [WJS94] named it as “object migration”, and Drossopoulou et al. [DDD01] named it as “re-classification”. The consecutive efforts on the research lead to the great achievement: Gottlob et al. [GSR96] focused on the role and extended the Smalltalk to enable the function of the dynamic classification. Drossopoulou et al. [DDD01] introduced a new approach to resolve the object mutation from the programming perspective, and implemented it in programming language DoR and Fickle. Kendall [Ken99] researched mostly on the perspective of aspect-oriented programming based on the role object pattern. Wieringa et al. [WJS94] presented a new concept of dynamic subclass that aims to the problem of class migration, etc.

While a lot of accomplishments were made on different areas of dynamic classification, some problems remained in implementation. For example, Gottlob et al. [GSR96] provides approach which distinguished object classes from role classes and prevented any other classes to represent role, so each real-world concept needs to be redefined; In [ABG+93], the approach does not satisfy the way whether allowing a class to create dynamic subobjects as well as primary objects and whether allowing an extended class hierarchy to enclose all the types used in an object-oriented language, etc.

Li [Li02] presented a new approach, which combines the notions of role and object. It uses a primary object to represent the static (permanent) properties of entity and uses a dynamic object to represent the dynamic (temporary) properties. He theoretically redefines the definition of object hierarchy, extended class hierarchy and class. He also proposed an extending class hierarchy with dynamic classification relation to improve the views of role and object. He introduced the concept of dynamic superclass-subclass relation in which an object of a class can be classified and declassified into/from other

classes dynamically. Using this approach, an object of class is implemented only once in its program and the ascription of this object can be changed.

Based on the approach developed by Li, we design a preprocessor to implement the dynamic classification in C#. It simplifies the program development by supporting dynamic classification. Using this preprocessor, C# codes including the extendable dynamic classification functions can be translated into standard C# codes. The output code can be compiled by the C# compiler.

This paper is organized as follows. In chapter 2, we introduce the basic concepts and review the background of object-oriented programming and dynamic classification. In chapter 3, we review and analyze the existing approaches to dynamic classification in different programming languages. A new approach to dynamic classification is discussed in chapter 4. We introduce an extended class hierarchy and the operational syntax, which use the new approach. In chapter 5, we discuss how to implement the proposed approach to dynamic classification in C#. Finally, conclusion and future works will be described in chapter 6.

## Chapter 2 Object-Oriented Programming and Dynamic Classification

### 2.1 Object-Oriented Programming

Object-Oriented Programming (OOP) is widely used in all fields of computer science and software industry. It is a simply way to use object-oriented concepts and apply them effectively and systematically in developing programs [Wu99].

#### 2.1.1 Object-Oriented Technology

Object-Oriented Technology (OOT) is a development principle and software modeling which makes a complex system into a set of separate components. OOT is a software development paradigm, which is based on the object concept. To reach the task of building a system that is reusable, scalable, flexible, and easy to maintain, OOT will be a possible solution. The history of OOT development is shown in Table 2.1.1.1.

OO Programming	Mid 1960s
OO Design	Mid 1980s
OO Analysis	Late 1980s
OO Methodologies	Early 1990s

Table 2.1.1.1 History of OOT

The OOT provides better methodologies to construct complex software systems from modularized software units. The OOT will bring the following advantages to the systems [KA95]:

- Using similar metaphors to interact easily with a computational environment.
- Constructing modularized reusable software units and software modules with easily extensible libraries.
- Modeling the real world as close as possible to user's attitude.
- Easily changing and extending implementations of units without the need to recode everything.

One of the OOT's major advantages is the concept of reusability. Instead of developing all the codes, OOT provides ability to construct standardized components. Besides, it improves maintenance of system.

### **2.1.2 Object-Oriented Programming Concept**

Object-oriented programming is different from traditional computer programming. In object-oriented programming, structure and module are different from that in conventional programming. Some concepts such as class, object, inheritance, classification, object identity and polymorphism come out. In the following parts, we will discuss them in detail.

- **Class**

In object-oriented programming, a class is a set or collection of abstracted objects that share common characteristics [KA99]. It is a kind of mold or template that the computer is used to create objects [Wu99]. Object is an instance of class, and one class may have many object instances. Normally, a class has variables and methods.

- **Object**

In object-oriented programming, an object is defined as an abstraction of a person, place, or things within the problem domain of which the information system must be aware [KA99]. Each object is an instance of a particular class.

- **Inheritance**

It is a feature of object-oriented programming language in which a subclass inherits methods and variables from its superclass. All lower level or children nodes in an inheritance hierarchy inherit the characteristics of the parent node. In some languages, inheritance can be applied for both class and interface.

- **Object Identity**

It means each object is distinguished from all other objects. With this property, objects can contain or refer to other objects. Object identity is something that is at the core of all persistence containers and most distributed object systems. It combines two distinct notions: one is the facility of object reference, which permits object correlation and access to object internal states. The other is the facility of object comparison, which permits the decision if two variables actually point to the same object.

- **Classification**

Classification is the process of organizing objects into groups that have same properties and operations. It is the ordering or separation of objects into classes. In organization of information, classification is the process of determining whether an information accords with a given hierarchy and then assigning the notation associated with the appropriate level of the hierarchy to the information and its surrogate.



- **Polymorphism**

Polymorphism means an object that takes on different forms. For example, H<sub>2</sub>O molecule can take on three forms: liquid, steam, or ice. It gives the same name to service in different classes. These services may do the work differently, yet they produce the same kind of results.

### **2.1.3 Object-Oriented Programming Language**

The first object-oriented language is Simula-67, an acronym from Simulation and Language, which was debut in the 1960s. In early 1980s, object-oriented programming has become a widely accepted style of programming.

The Simula-67 took the block concept from the Algol one step further and introduced the concept of object. In 1970s, with the combination of the concept of object-oriented language from Simula and other earlier prototypes, there came a language which may be one of the most influential object-oriented languages: Smalltalk. It is the first pure object-oriented programming language developed by Xerox PARC.

During 1980s and 1990s, with the introduction of C++ and Java, object-oriented programming languages got a great development. In 1985, C++ was introduced as an extension of the C programming language. In May 1995, Sun formally announced Java at a major conference. In 2000, C# was submitted by Microsoft to the ECMA standards group and released it with the .NET Framework. It is an evolution of the C and C++ languages. Now, these languages have been widely used as the basic and mainstream object-oriented software [KA95] [DH98].

### **2.1.4 Object-Oriented Programming Design and Analysis**

In 90's, object-oriented paradigm became more and more popular, as a practical and effective approach for software development. The object-oriented programming method has already been widely used by software developers.

With the perspective of software engineering, the basic composing portion of object-oriented programming method includes object-oriented analysis, object-oriented design and object-oriented program. The famous Coad's object-oriented methodology includes four major activities [Nor96]:

- Identify information system purposes and features
- Identify model component objects and patterns
- Establish object responsibilities
- Define service scenarios

In the real world, analysis means each product construction must have detailed consideration of what is the problem to be solved, and design means how it can be resolved. Object-oriented analysis (OOA) provides detailed description of the problem. OOA is a step when transferring a real-world problem into an object-oriented implementation. This step involves understanding the problem which is to identify entities, relationships, and operations, separates the static and dynamic parts, and identifies the necessary operations. Object-oriented design (OOD) tries to provide the "blueprint" for implementation. OOD is the next step to perform after the object-oriented analysis. The goal of design is to produce detailed implementation of product structure. The OOD is composed of two parts: a coarse-grained design and detailed object-oriented design.

## 2.2 Dynamic Classification

Dynamic classification means object classification that is performed at runtime. In other words, with dynamic classification, the classification of object can change over time [MO98].

There are many similar situations in the real world. For example as Figure 2.2.1, a person can play different roles in his lifetime: student and employee. In phase 1, 2 and 5, he is a student. In phase 3 and 6, he is an employee. In phase 4, he acts both student and employee.

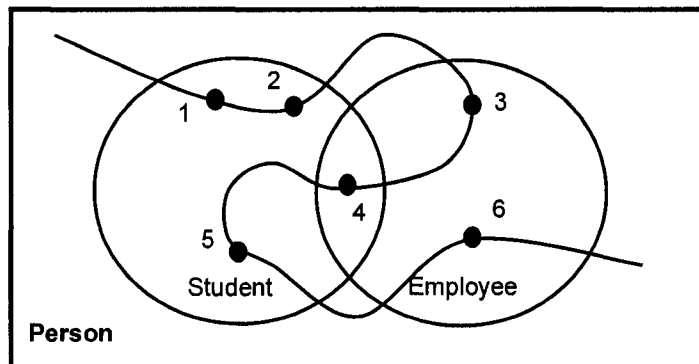


Figure 2.2.1 A person's life track

1. High-school Student
2. College Student
3. Company Employee
4. Company Employee & Part-time Master Student
5. Master Student
6. Company Employee

From the above example, we can see the phenomenon of dynamic classification, which tries to model the real world entities. Object-oriented programming is helpful to resolve the problems in the real life. So we concern is how to realize dynamic classification in

object-oriented programming. In the following parts we focus on the problems during the implementation of the dynamic classification in conventional object-oriented programming and introduce some related concepts in dynamic classification.

### **2.2.1 Conventional Object-Oriented Model**

In conventional object-oriented programming, an object is created from a class and contains all the attributes and methods. In its lifetime, this object always and only belongs to this class. But in reality there are many entities that need to play different roles in different situation. As shown in Figure 2.2.1, we shall create three classes: **Person**, **Student** and **Employee** to illustrate the different roles in one's lifetime. We hope an object which is the instance of class **Person** plays the student role that possesses the attributes and methods of the class **Student**, and plays the employee role when he is employed later. In the existed object-oriented languages this problem cannot be resolved directly, since they do not provide mechanisms for object to change their class membership. When programmers need dynamic classification, two possible solutions are provided [DDD+01]:

- Create an instance of the new class, copy the properties of old instance to the new instance, and then delete the old one.
- Merge two classes into one to combine the properties of the two classes into the new class.

However, neither solution is satisfied. In the first solution, all references to old objects need to be informed of changing. The second solution combines two different classes into one class. It violates the original intention of object-oriented class concept, which classifies objects by their attributes and behaviors.

### 2.2.2 Concept of Role

Role implies the character described in the drama or fiction. But it has a different meaning in computer area. In early 1977, Bachman et al. started to pay attention to the role and said the role is used with the relational model to describe the way in which an attribute relates to domains, the role type means the prototype of a class of roles with similar properties [BD77]. Kristensen et al. define the role of an object as a set of properties that are important for an object to be able to behave in a certain way expected by a set of other objects [KØ96].

An entity can play one or more roles. For example, a man is an entity, and his roles can be employee, manager, customer, student, and so on. Role is different from the entity. The role type is the prototype of the roles that share the same character, whereas the entity type is the prototype of the entities that play the same role. For example, one student entity is the type of people that possess the student role. The relationship between the role type and the entity type is many-to-many relation. An entity can play many roles and a role can be possessed by many entities. Therefore, both role types and entity types were viewed as object types without the distinction which is made in the role data model. Role has some characteristic features for analyzing the dynamic entities and plays various roles, particularly [GSR96] [Kri95]:

- Different roles of one entity may share the same structure and behavior. For example, the student role and employee role of a person can share the information of name, gender etc.
- Entities can add and delete roles dynamically. For example, a person acquired employee role when he is employed, then he abandons employee role when he quits the job.
- Roles can be added and deleted independently of each other. For example, an employee can become a project manager independently of being a department manager.

- **Dynamicity:** Entities exhibit role-specific behavior. For example, a person can have different phone numbers in their role of student and employee role.
- Roles restrict access to a particular feature. For example, a company's document can only be viewed by a department manager role, but not a general employee role.
- Entities may appear repeatedly in the same type of role. For example, a student may become a teacher assistant of several courses, and each of these courses may need different knowledge.

Another concept relates to the role is role model. Role model is the description of a set of object collaborations using role type [RG98]. Once role model is identified, it is easy to be recognized in the real world [BD77]. Roles and role models are abstraction and decomposition mechanisms. A role model identifies a prototype and a reoccurring structure of roles. It can be used for analysis and design [Ken99].

### 2.2.3 Patterns of Role

In [Fow97], Fowler divided roles into five analysis patterns to solve the problem, which are easy to represent many roles of the same object. The five patterns are single role type, separate role type, role subtype, role object, and role relationship. Figure 2.2.3.1 [Fow97] shows the structure of these analysis patterns.

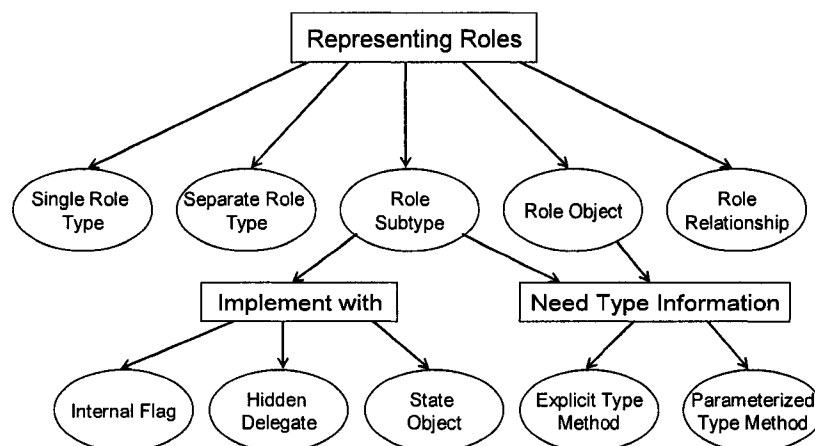


Figure 2.2.3.1 Structure of patterns

- **Single role type:** when the objects share most of the behavior but with few variations. It can use one type to describe all of them.
- **Separate role type:** when most of the objects share no common behavior.
- **Role subtype:** when some of the objects share common behavior while others cannot use the role subtype. The important key of this technology is that customers are convinced of dealing just with one single object, which has multiple types. The implementation of this technology uses three patterns: internal flag, hidden delegate, and state object. And there are two methods involved to get the relevant information.
- **Role object:** There are many discussions about role objects in relevant articles [Sch96] [Fow96] [Gam96] etc. Role object means each role has its own separate object, links to a basic object that includes common features. The user accesses to basic object for relevant role to use a role's features. The essential of both role object and state object are identical. Figure 2.2.3.2 [Fow97] shows how to use role objects for person, and each person has a set of role objects for its various roles.

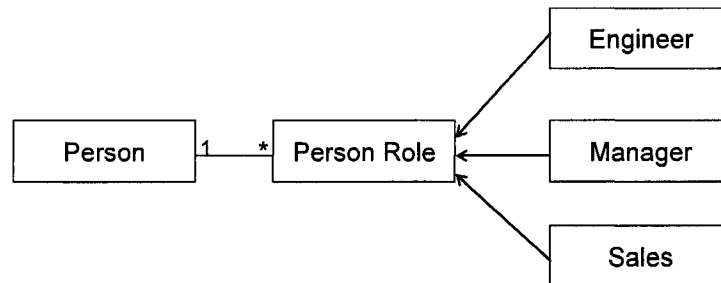


Figure 2.2.3.2 Using role objects for person

- **Role relationship:** It means that the role acts as a relationship between two objects. It is required when you consider an organization with several different groups. For example an engineer can begin work with one group then change into another group. Figure 2.2.3.3 [Fow97] shows treating role as a relationship.

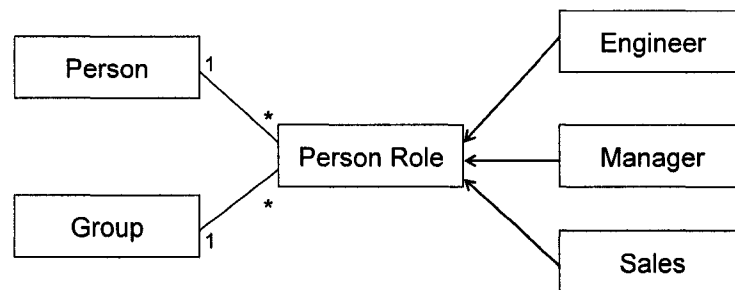


Figure 2.2.3.3 Treat the role as a relationship

Many researchers are dedicated to implementing dynamic classification by considering patterns for role. Bäumer et al. processed the research on role object pattern [BRS+97]. They show that role objects can be dynamically attached to and removed from the core object. For example, two different customers can separately play the role of borrower and investor, and a single customer object can play both roles. They point out that the implementation of the role object pattern must involve the consideration of two aspects: transparently extending key abstraction with roles and dynamically managing these roles. Many problems need to be considered when it is implemented in the real world. In particular, as described in following [BRS+97]:

- Providing interface conformance
- Hiding the role object creation process
- Decoupling role classes from the core
- Choosing appropriate specification objects
- Managing role objects
- Maintaining consistent core and role object state
- Maintaining role attribute constraints by using Property and Observer
- Maintaining conceptual identity
- Maintaining constraints among roles
- Maintaining constraints among roles by recursively applying the role object pattern



Research on analyzing patterns is based on the conceptual points, rather than implementation point [Fow97]. All the approaches can be used to design and implement the dynamic classification. So there are numerous works to be done on how to implement these approaches in the real computer languages [Ken99].

### 2.2.4 Class Hierarchy

In object-oriented system, class hierarchy is used to denote the relationship between classes. The class hierarchy has the following characteristic features [GSR96]:

- Class hierarchy supports sharing of structure and behavior of several classes due to inheritance.

Figure 2.2.4.1 models a typical class hierarchy. The rectangle represents class and the solid arrow represents the relationship between superclass and subclass. In this figure the class Person is a superclass that contains the common features of entities, the class Student and Employee are subclasses that are the extension of class Person and inherit the attributes and methods from class Person.

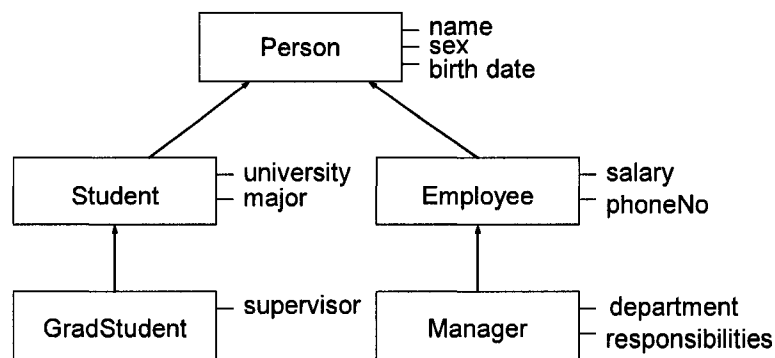


Figure 2.2.4.1 A class hierarchy

The entity in the real world can be represented as the instance of specific class. Figure 2.2.4.2 shows how to represent an instance object in a class hierarchy. Mr. Jason is an instance of class `Employee`, and it inherits some attributes from the class `Person` such as `name`, `birthDate`, and `privatePhoneNo`.

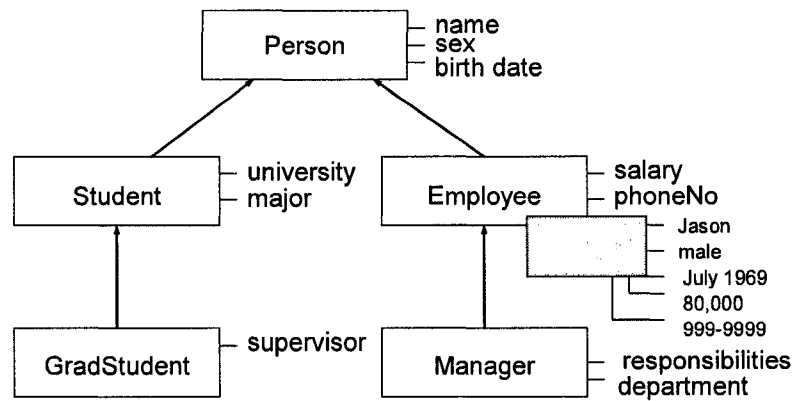


Figure 2.2.4.2 Class instantiation in a class hierarchy

- Evolving objects tracking is a tedious task.

In traditional object-oriented programming, the procedure for an entity changing its role is as follows: create a new class, copy the attributes of old class to new one, references to old objects need to be reset to new one, and finally delete the old one. Figure 2.2.4.3 shows the procedure of object Jason changed from student to employee.

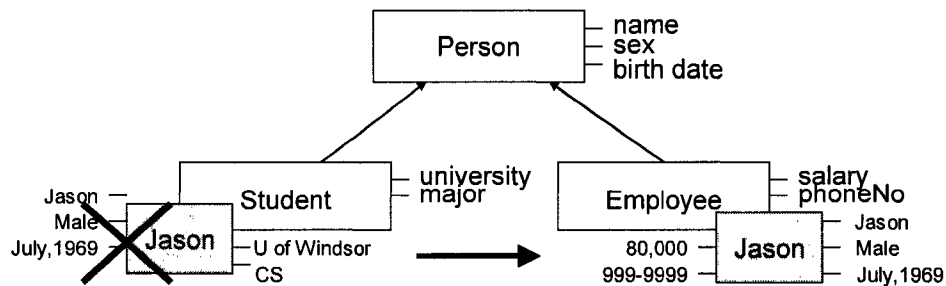


Figure 2.2.4.3 Evolving an object in class hierarchy

- Class hierarchies must be planned carefully or may grow exponentially when entities take on several roles.

We need to consider the relationship between different classes in dynamic classification description. Through the concept introduction on class hierarchy, problems arise on how to describe object dynamic classification with class hierarchy. We will discuss it on the following chapter.

## Chapter 3 Related Researches

In 1977, Bachman and Daya [BD77] first began the study on the dynamic classification. They build the concept of role model that is the extension of network model with role concept. In this model, the required data description and data manipulation language integrate the concept of record and role segment. The role model shows the data description with existing data can be provided to a high level. It is richer than all other earlier models and provides a good start for further study. A lot of researchers start to study on this model, and many different approaches were provided.

Based on the research method, there are foundational perspective, database perspective and programming perspective [DDD00]. Based on the way of how to realize the possibility, there are role type, pattern for role modeling, prototypical and aspect programming with roles [Li02]. We will explain the implementation of dynamic classification in different programming languages by groups.

### 3.1 Extended Smalltalk

Gottlob et al. [GSR96] presents how class-based object-oriented systems can be extended to handle evolving objects. Since original class hierarchy has serious difficulties in modeling the evolving objects, the combining role hierarchies can complete it. The difference between these two hierarchies is that the role hierarchy doesn't need to inherit the definition from supertype. An entity can be represented by an instance of the root and an instance of role type. So they compare the representing roles by class hierarchy, role hierarchy, and combined two.

Figure 3.1.1 [GSR96] shows the combination. The class hierarchy includes three classes: LegalEntity, Person, and Company. The role hierarchy includes: Customer, Student, Employee, Department Manager and Project Manager. Classes may have subclasses and may function as the root of role hierarchies. For example, class Person is the leaf of class hierarchy (class LegalEntity) and root of role hierarchy (role Student, Employee, Department Manager and Project Manager).

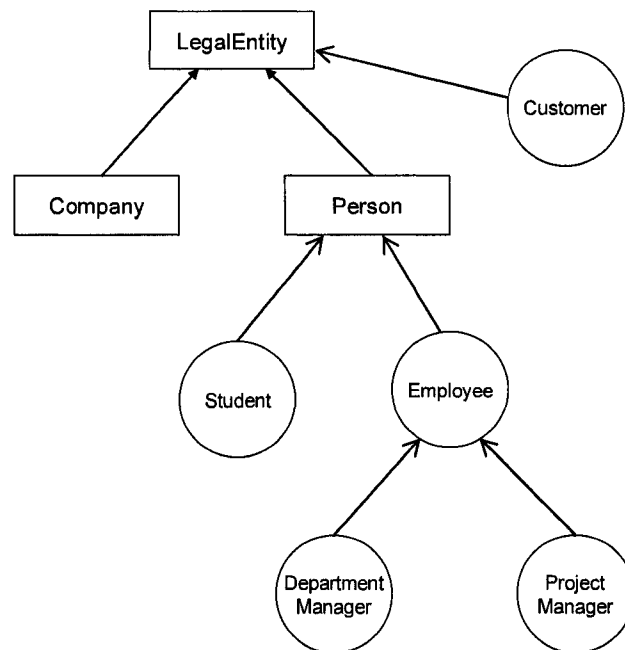


Figure 3.1.1 A class and role hierarchy

To implement this approach, they extend object-oriented language Smalltalk by adding a few classes to support the role mechanism and dynamic classification. By adding three classes, roles can easily be implemented as an additional feature in Smalltalk. Smalltalk can handle the evolving objects, and do not need to modify its definition. The three classes are RoleType, ObjectWithRoles, and QualifiedRoleType. Their functions are discussed in detail as follows:

- Class `ObjectWithRoles`: Used to define the behavior and structure of objects which may take on roles and roles may be roots of role hierarchies.
- Class `RoleType`: Used to define the behavior and structure of objects that are inner nodes of a role hierarchy. Table 3.1.1 [GSR96] shows the methods for defining role type and instances.

Class Method of RoleType	Purpose
<code>defRoleType: roleName</code> <code>instanceVariableNames: stringOfInstVarNames</code> <code>classVariableNames: stringOfClassVarNames</code> <code>poolDictionaries: stringOfPoolNames</code> <code>Category: categoryNameString</code> <code>roleSuperType: nameOfRoleSuperType</code>	Define role type <code>roleTypeName</code>
<code>newRoleOf: anObject</code>	Create new role of <code>anObject</code>

Table 3.1.1 Class methods of `RoleType`

- Class `QualifiedRoleType`: Used to define additional structure and behavior of qualified roles. It is a subclass of class `RoleType`. Table 3.1.2 [GSR96] shows the methods for defining qualified role types and instances.

Class Methods of QualifiedRoleType	Purpose
<code>defQualifiedRoleType: roleName</code> <code>instanceVariableNames:stringOfInstVarNames</code> <code>classVariableNames: stringOfClassVarNames</code> <code>poolDictionaries: stringOfPoolNames</code> <code>Category:categoryNameString</code> <code>roleSuperType:nameOfRoleSuperType</code> <code>classOfQualifyingObj: aClass</code>	Define qualified role type <code>roleTypeName</code>
<code>newRoleOf: anObject qualifiedBy:qualifyingObj</code>	Create new qualified role of <code>anObject</code>

Table 3.1.2 Class methods of `QualifiedRoleType`

### 3.2 DoR

Drossopoulou et al. [DDD01] introduce a new approach to resolve the object mutation from the programming perspective. They define an operational semantics for re-

classification operation to change the class membership of an object. In this approach, two additional classes are introduced: `state` classes are used to describe the object which may be reclassified, and `abstract state` classes are used to describe the abstract superclasses of state classes. It is implemented through language DoR. They develop a type and effective system for DoR and use the operational semantics to prove its soundness. DoR is an imperative, class-based and Java-like language, which allows object's dynamic reclassification by changing the object's class. In language DoR, they present the relevant syntax, operational semantics and typing. The benefits of this approach are that it is more liberal than most others from the programming perspective and it allows the object to be changed from class to class and it can change back to original class.

### **3.3 Fibonacci**

Albano et al. [ABG+93] present a new mechanism that shows how to use the existing object-oriented features as inheritance and late binding to solve strictly related problems. They introduce the Fibonacci features that are used in database problems from construction to properties. Fibonacci is a new strongly typed and object-oriented database programming language. It has a mechanism to model objects with roles to resolve the challenge of changing the object type dynamically at runtime.

### **3.4 Aspect Programming**

Based on the Object-Oriented Database System (OODBs), Richardson et al. [RS91] studied the perspective of aspect programming, and it extends objects to support multiple and independent roles. They add additional states and behaviors to an existing object while sharing the same object identity. This object model includes three parts: `abstract types`, `implementations` and `conformity rules`. The abstract type is independent of an implementation and may be conformable with multiple implementations.

Kendall [Ken99] studies mostly on the perspective of aspect-oriented programming which is based on the role object pattern. She points out that role object pattern have some problems in implementation level, which are object schizophrenia, significant interface maintenance and no support for role composition. She uses a conceptual model [Kri95] for an object and its roles to resolve some of these problems. In this model, an object is used to keep the intrinsic properties of an entity, roles are used to keep extrinsic properties of an entity and provide interface for other entities or roles to view and access it.

For example: Figure 3.4.1 [Ken99] shows the relationship between an object and its roles. A worker from the view of boss is subordinate, so he/she has three intrinsic members and three extrinsic. However, from view of customer, the worker plays two roles provider and subordinate, so he/she has three intrinsic members and five extrinsic.

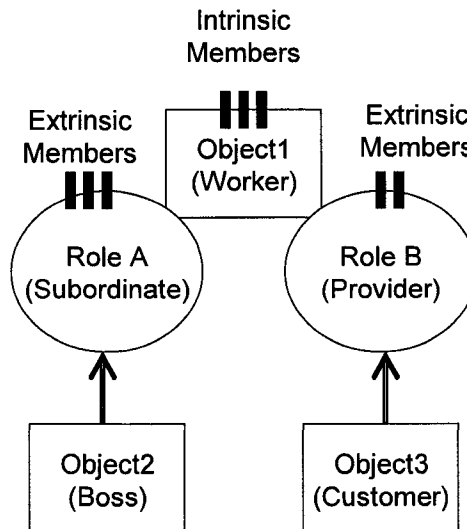


Figure 3.4.1 An Object and its Roles: Intrinsic and Extrinsic Members

For implementation view, Kendall describes how to implement role models in aspect-oriented language AspectJ and compare the difference with object-oriented language. Based on the five options of aspect-oriented designs and the suggestion of G. Kiczales,



she utilized the subject-oriented programming features to present the hybrid approach for role models. The five options are shown as follows [Ken99].

- Option 1: Static aspect introduces extrinsic role members to core class.
- Option 2: Aspect instance implements role behavior by advising role members that already exist in a core instance.
- Option 3: Aspect instance contains role members separate from a core instance. Two separate entities are used.
- Option 4: Aspect instance filters out invalid role members from a core instance with advice weaves. The core instance contains members for all roles.
- Option 5: Role and core are objects. Static aspect integrates or composes them using introduces weaves.

### **3.5 Prototype-Based Language**

Sciore [Sci89] discussed the specialization at the level of object in the prototype-based language. In his approach, objects are allowed to define their own inheritance part. The object specialization transfers the specialization hierarchy from type-level to object-level. It can create a more flexible and functional hierarchy. Base on the concept of the class-based system, each object is assigned to a class and has the same variables and methods. New methods let the object itself to choose whether inheritance is needed, as with prototype-base systems. Objects of real-world entity are contained in an object hierarchy. Each object in object hierarchy represents a role played by entity. An object can be added or removed from an object hierarchy.

The prototype-based system will be slower than the class-based system, which is suitable for management over large number of objects. This approach focuses on how to combine both the efficiency of the class-based system and the flexibility of the prototype-based system to reach the desired result. It satisfies the following three dimensions: whether an

object can change its inheritance path at run time, whether the inheritance is implicit or explicit, and whether inheritance is each object or each group [LSU88]. The key feature is that object can inherit from other object without abandon the ability of managing large numbers of object efficiently by using the class hierarchy.

## Chapter 4 Programming with Dynamic Classification

Base on the early research, Li [Li02] presents a new approach to object-oriented dynamic classification, which combines the concept of object and role. The traditional class hierarchy has a problem to represent the dynamic classification relation of classes. It does not support the specific behavior of role. To solve the problem, he extended class hierarchy with a dynamic classification relation between classes to improve it. He also unified the notion of role and object by allowing a primary object to represent the static (permanent) properties and a dynamic object to represent the dynamic (temporary) properties.

### 4.1 Extended Class Hierarchy

In the real world, some properties of the entities are permanent and some others keep temporarily. For example, one person Jason has permanent properties of human as primary property, and has temporary properties during different phases in his life. He possesses the property of student while he is a student in school and possesses the property of employee while he is employed later. Therefore we may create different classes (Person, Student and Employee) in object-oriented programming to describe the entity. The object generated by class Person describes the inherent and intrinsic properties. We call it *primary object*. The object generated by class Student or Employee describes the dynamic and extrinsic properties. We call it *dynamic object*.

For the relation between these objects, we use a *parent-child relation* to describe it and compose the objects and relation on an *object hierarchy*. For a real word entity, we create a primary object as the root of the object hierarchy, and create other dynamic objects as the leaf of the object hierarchy to describe the change of the status. A non-leaf object in

the hierarchy may have one or more children. An entity can be described by a unique primary object and other one or more dynamic objects. For example, Jason as an object of class `Person` is an entity in the real world. While he is a student, an object of class `Student` shows his property of student. While he is an employee, an object of class `Employee` shows his property of employee. Combination of different objects can adequately show the property of student Jason or employee Jason. A parent-child relation exists between these objects. Here we use an object hierarchy to describe it. It is shown in Figure 4.1.1a.

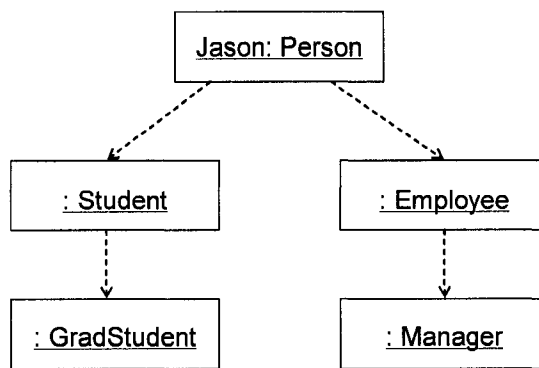


Figure 4.1.1a An object hierarchy

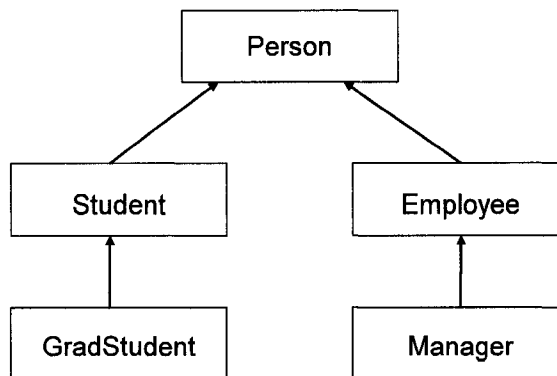


Figure 4.1.1b A class hierarchy

In object-oriented programming language, class as aggregation of the objects who possesses the same property describes the entities in the real world. We can define the inheritance relation between classes. A class may inherit the attributes and methods from another class if there is an inheritance relation between two classes. As shown in Figure

4.1.1b class `student` and `person` have inheritance relationship. The class `Student` inherits the attributes and methods from the class `Person`. It is a traditional class hierarchy.

Inheritance relation between classes is static, and subclass can statically inherit the attributes and methods from superclass. So we call the inheritance relation in class hierarchies as *static subclass-superclass relation*. The traditional class hierarchy can only describe the inheritance relation, not applicable for dynamic classification relation. Li improved class hierarchy by presenting an *extended class hierarchy*. In Figure 4.1.2 it shows an example of extended class hierarchy that combines Figure 4.1.1a and Figure 4.1.1b. He also defined a new relation combining these two relations (parent-child relation and static subclass-superclass relation) as *dynamic superclass-subclass relation* that satisfies the following conditions [Li02]:

- Condition 1: If class  $C_1$  is static subclass of class  $C_0$ , class  $C_3$  is a static subclass of class  $C_2$ , and class  $C_0$  is a dynamic parent of class  $C_2$ , we say that there is an implied dynamic parent-child relation between the classes  $C_1$  and  $C_3$ , and class  $C_3$  is a dynamic subclass of  $C_1$ .
- Condition 2: If class  $C_2$  is a dynamic subclass of class  $C_1$  and  $C_1$  is a dynamic subclass of class  $C_0$ , we call  $C_2$  a dynamic subclass of  $C_0$  as transitivity.

The concept of dynamic superclass-subclass relation is different to inheritance relation. The dynamic classification relation is dynamic, and it just likes the above definition of parent-child relation in object hierarchies. For example, we imagine two classes:  $C_0$  and  $C_1$ . The dynamic classification between these two classes is defined as: if the object of class  $C_0$  can dynamically acquire the attributes and method of  $C_1$ , we call the class  $C_0$  the *dynamic parent* of class  $C_1$ ,  $C_1$  is *dynamic child* of  $C_0$ .

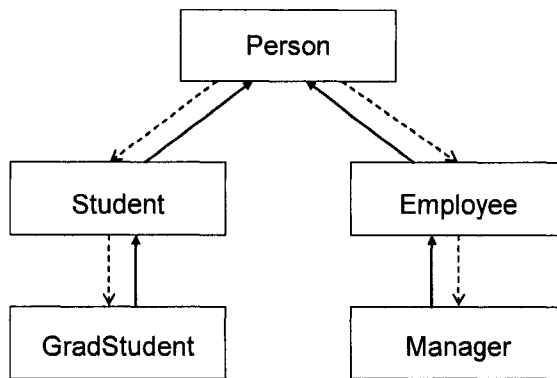


Figure 4.1.2 An extended class hierarchy

To better describe the approach, we show an example in Figure 4.1.3 which the class **Student** and **Employee** inherits from class **Person**, class **Manager** inherits from class **Employee** and the class **Person** is a dynamic parent of the class **Employee**. We denote the solid link as inheritance relationship, dashed arrow as dynamic classification relationship. Suppose  $C_0$  presents class **Person**,  $C_1$  as class **Student**,  $C_2$  as class **Employee**,  $C_3$  as class **Manager**. Based on the Condition 1, we can conclude an implied dynamic parent-child relation between the class **Student** and **Manager**.

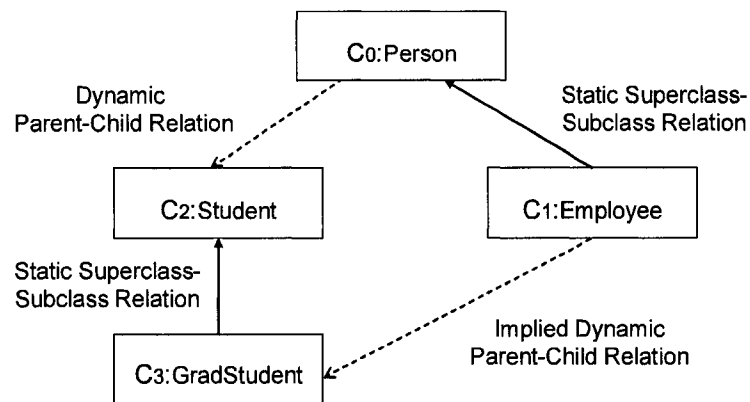


Figure 4.1.3 Implied dynamic parent-child relation

As a consequence, there is an implied dynamic classification between the class and itself, because the class is a static subclass of itself. For example,  $C_0$  is of class **Person**.  $C_1$ ,  $C_2$ ,

$C_3$  are of Student classes. We can get the implied dynamic parent-child relation between the class Student and itself.

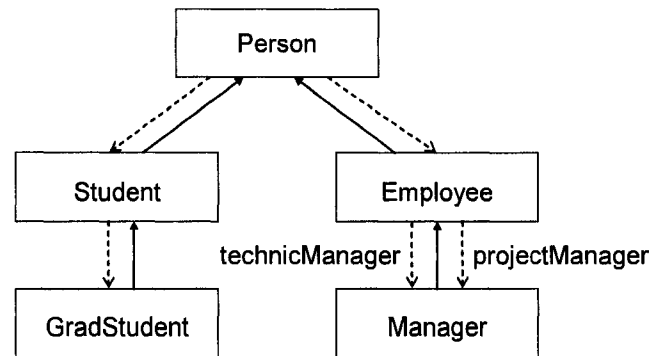


Figure 4.1.4 An extended class hierarchy with dynamic classification

In the real world, an entity may play different roles simultaneously. To model it, we extend the class hierarchy to support multiple dynamic classifications. It is showed in Figure 4.1.4. For example, we may create two instances of class **Manager** as dynamic children of an employee object and this object can play different roles that are technical manager and project manager. Now we define an operational semantic to illustrate this dynamic classification relation. We define dynamic classification relation between the class  $C_0$  and  $C_1$  as a triple  $\langle C_0, l, C_1 \rangle$ , the label  $l$  identifies the role in the dynamic classification relation. The dynamic classification relation illustrated in the Figure 4.1.4 can be presented with triples:

- $\langle \text{Person}, \text{employee}, \text{Employee} \rangle$
- $\langle \text{Person}, \text{student}, \text{Student} \rangle$
- $\langle \text{Student}, \text{gradStudent}, \text{GradStudent} \rangle$
- $\langle \text{Employee}, \text{technicManager}, \text{Manager} \rangle$
- $\langle \text{Employee}, \text{projectManager}, \text{Manager} \rangle$

Up till now we fully introduced and discussed the new approach of object-oriented dynamic classification which is presented by Li. Basically, Li summarized the new approach by the following definitions: extended class hierarchy, object hierarchy and class definition [Li02].

**Definition 1:** An *extended class hierarchy* is a triple  $(C, I, R)$  with finite sets  $C, I$ , and  $R$  such that

- $C$  is a set of classes;
- $I \subset C \times C$  is an inheritance relation that implies no cycle;
- $R \subset C \times N \times C$  is a dynamic classification relation such that for any dynamic classifications  $\langle C, l_1, C_1 \rangle$  and  $\langle C', l_2, C_2 \rangle$  in set  $R$ , if  $C = C'$  and  $l_1 = l_2$ , then  $C_1 = C_2$ . The symbol  $N$  is the set of all identifies in the object-oriented programming language.

**Definition 2:** For an extended class hierarchy  $(C, I, R)$ , an *object hierarchy* is a quadruple  $(Q, P, \mathcal{F}, \mathcal{E})$  with finite sets  $Q, P$  and functions  $\mathcal{F}, \mathcal{E}$  such that

- $Q = \{\text{obj}_0, \text{obj}_1, \dots, \text{obj}_n\}$  is a non-empty set of objects;
- $P \subset Q \times Q$  is a parent-child relation between objects in set  $Q$  such that graph  $(Q, P)$  is a tree;
- $\mathcal{F}: Q \rightarrow C$  maps objects in  $Q$  to classes in  $C$ ;
- $\mathcal{E}: P \rightarrow R$  maps the parent-child relation  $P$  between objects to dynamic classification relation  $R$  such that for each  $\langle \text{obj}, \text{obj}' \rangle \in P$ , if  $\mathcal{E}(\langle \text{obj}, \text{obj}' \rangle) = \langle C, l, C' \rangle \in R$ , we have  $\langle \mathcal{F}(\text{obj}), C \rangle \in I^*$  and  $\langle \mathcal{F}(\text{obj}'), C' \rangle \in I^*$ .

The symbol  $I^*$  is reflexive and transitive closure of inheritance relation  $I$  that is static subclass-superclass relation.

**Definition 3:** Assume an extended class hierarchy  $(C, I, R)$ . The *definition of a class*  $C \in C$  specifies a pair  $\langle C^\alpha, C^\beta \rangle$  of finite partial functions such that

- $C^\alpha: N \rightarrow C$  declares each attribute name  $a \in \text{dom}(C^\alpha)$  with a type  $C^\alpha(a) \in C$ ;



- $C^{\beta}: (\mathbb{N} \times C^+) \rightarrow C$  specifies a return type  $T = C^{\beta}(m, (C_1, C_2, \dots, C_n)) \in C$  for each pair  $(m, (C_1, C_2, \dots, C_n)) \in \text{dom}(C^{\beta})$ . We say that class  $C$  defines return type  $T$  for operation  $m(C_1, C_2, \dots, C_n)$ .

## 4.2 Syntax of Dynamic classification

Based on the above theory, Li defined the syntax to implement dynamic classification in object-oriented programming language. Here we follow the presented syntax in [Li02]. In that paper, it discusses dynamic classification, syntax structure, especially the dynamic classification on objects of class. It can be used to extend any object-oriented languages, such as C++, Java, or C#.

In this paper we use the syntax of extended BNF (Backus-Naur Form) to illustrate this approach. A BNF grammar is composed of a set of production rules. Each production rule has two sides left and right and separated by the symbol ‘ $:: =$ ’. The right side contains a non-terminal symbol. And the left side is consisted of one or more alternative specifications [Rag81]. For more specific, the syntax meaning is as follows:

- Symbol ‘|’ is used for separated alternative specifications
- Symbol ‘<’ and ‘>’ is used for enclosing a string of one or more characters
- Square bracket ‘[’ and ‘]’ is used for surrounding optional.
- Symbol ‘{’ and ‘}’ indicates repetition.
- Suffix ‘\*’ is used for a sequence of zero or more of an item.

### 4.2.1 Class Definition

First, we introduce the definition of class with dynamic classification. The following syntax can be used to define the static and dynamic superclass-subclass relation in object-oriented programming language. Keyword `class` is used to define the class name, and symbol ‘:’ is used to define the inheritance relationship. The symbol ‘:’ is followed by the class name of static parent. Keyword `dynamic` is used to define dynamic superclass-

subclass relationship. It is followed by the class name of dynamic child. Keyword `label` is used to define the role of the object in that class, and it introduces identifier lists of dynamic classification relation. By this syntax we can implement the dynamic classification relation (C, I, R) defined in the Definition 1.

The following syntax extended the grammar in C# language. The keyword `class` and symbol `'::'` are standard C# sentence, and keyword `dynamic` and `label` are not included in C# language. Here is the definition of class syntax.

```

<class_definition> ::= class <class_name>[<static_parents>]*
                        [<dynamic_classification>]*<class_body>
<class_name> ::= <identifier>
<static_parents> ::= : <class_names>
<dynamic_classification> ::= dynamic <class_names>
                        [<label_lists>]
<class_names> ::= <class_name>{,<class_name>}*
<label_lists> ::= label [<label_list>]
<label_list> ::= <identifier>{,<identifier>}*

```

Next, we discuss the class body. In class body, we can define the attribute and method in its class. The method can be composed of a signature and a method body that include one or more statements. Here is the syntax of class body.

```

<class_body> ::= [{<property>;}]*
<property> ::= <attribute>|<method>
<attribute> ::= <class_name><attribute_name>
<attribute_name> ::= <identifier>
<method> ::= <class_name><method_name>([<parameter_list>]*)
                        [{<statement>}*]
<method_name> ::= <identifier>

```

```

<parameter_list> ::= <class_name><identifier>{,<class_name>
                    <identifier>}*

```

In the example of company employee that is forenamed, we can construct three classes Person, Employee and Manager. Class Employee takes the class Person as its static parent, and the class Employee can inherit all the attributes from class Person. Class Employee takes the class Manager as a dynamic child and class Employee can obtain the attributes dynamically from class Manager. The class Manager plays two roles: projectManager and technicManager. Figure 4.2.1.1 shows the relationship between three classes.

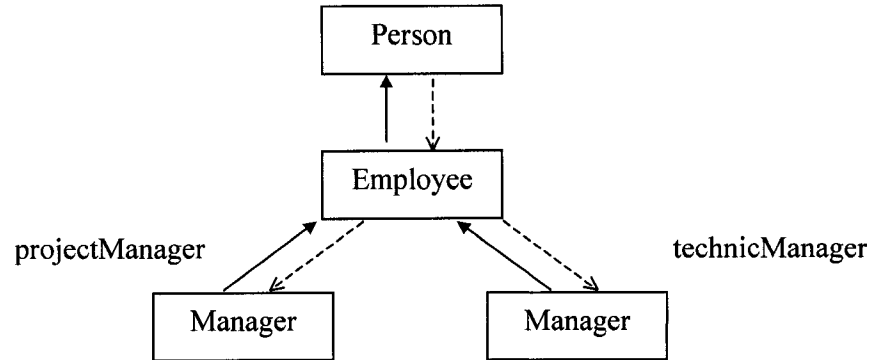


Figure 4.2.1.1 Relations of three classes

The followings are the demo codes, and it shows how to define classes to implement dynamic classification relation like  $\langle \text{Employee}, \text{projectManager}, \text{Manager} \rangle$  and  $\langle \text{Employee}, \text{technicManager}, \text{Manager} \rangle$  in Figure 4.2.1.1. Meanwhile we define the attribute serviceLength and method setServiceLength, getServiceLength for this class.

```

// class Employee
Class Employee:Person dynamic Manager label[projectManager,
                                         technicManager]
{
    Year serviceLength;
    setServiceLength(Year year)
    {

```

```

        this.serviceLength = year;
    }
    Year getServiceLength()
    {
        return this.serviceLength;
    }
}

```

#### 4.2.2 Statement Definition

We now discuss the syntax for *statement* definition, which is divided into three types. An *expression* ends with a semicolon. An *assignment* statement assigns the parameter and attribute to an object. The third type is *conditional* statement such as if-else statement.

```

<statement> ::= <expression>; | <assignment> | <conditional> |
              (<statement>)
<assignment> ::= <parameter> = <expression>; |
                 <express>. <attribute_name> = <expression>;
<conditional> ::= if <expression> then <statement>
                 else <statement>

```

We define the *expression* as follow:

```

<expression> ::= <object_creation> | <declassification> |
                 <field_access> | <method_invocation> |
                 <dynamic_child> | <dynamic_parent> |
                 <constant> | <parameter> | (<expression>)

```

An *expression* principally performs the following operations:

- Create primary and dynamic object of class. Detailed syntax is as follows:

```

<object_creation> ::= <primary_object_creation> |
                    <dynamic_object_creation>

```

```

<primary_object_creation> ::= new <class_name>()
<dynamic_object_creation> ::= <expression> newChild(
    <class_name>, <identifier>, <class_name>, <class_name>)
    |<expression> newChild(<class_name>, <identifier>)
    |<expression> newChild(<identifier>)

```

Here is the example:

```

Student Jason = new Student ();
eJason = Jason newChild(Person, employee, Employee);
eJason newChild(projectManager);

```

The example describes the process of Jason from being a student to a manager. First we create object Jason in class Student as a primary object with the expression.

```

Student Jason = new Student ();

```

Then, we create object eJason of class Employee as a dynamic object with the expression.

```

eJason = Jason newChild(Person, employee, Employee);

```

By building the parent-child relationship between object Jason and eJason through dynamic classification <Person, employee, Employee>, the parent Jason has the ability to access the attributes and methods to child eJason. Same way, the following expression creates a new object of class Manager and builds the parent-child relation with object eJason through dynamic classification <Employee, projectManager, Manager>. Obviously, object Jason can be student, employee, or manager.

```

eJason newChild(projectManager);

```

Similarly, expression

```

newChild(projectManager);

```

is the abbreviation of the expression.

```

newChild(Employee, projectManager, Manager);

```

- Declassification object. Using these expressions to remove the dynamic objects and delete the parent-child relation between two objects. Among these expressions `removeChild` is a Keyword, acting as remove the appointed relations separately. The detail syntax is as follows:

`<declassification> ::= <expression> removeChild`

- Store and acquire operations on data of object. The detail syntax is as follows:

`<field_access> ::= <expression>. <attribute_name>`

`<method_invocation> ::= <expression>. <method_name>  
 ([<argument_list>]*)`

`<argument_list> ::= <expression>{, <expression>}*`

Herein, we comprehensively discussed the new approach presented by Li from the motivation, definition to the implementation syntax. Next we will focus on the detailed implementation of dynamic classification in object-oriented programming languages.

## Chapter 5 Implementing Dynamic Classification in C#

This chapter works on how to extend the object-oriented programming language C# to implement the functions of dynamic classification. We will present a preprocessor, by which a C# code including the extendable dynamic classification functions can be translated to standard C# code. The translated code can be compiled by a C# compiler. This chapter is divided into four sections: feature of the C# language, implementation of the preprocessor, the restrictions and assumptions of syntax, and how to use the preprocessor.

### 5.1 Features of the C# Language

C# language is an object-oriented programming language. It can be used to create applications that run in the .NET CLR. C# is an evolution of C and C++ language and it is created by Microsoft especially to work with the .NET platform. C# can be used for more common application types: windows applications, web applications and web services. As it is a recent language, it has been designed with foresight and contains many good features from other languages while clearing up their problems [Wat02]. The C# language has the following features [Msd04].

- C# is a simple but powerful programming language, which is intended to write enterprise applications.
- C# uses many C++ features in the areas of statements, expressions, and operators.
- C# has much improvement and innovation in the area of type safety, versioning, events, and garbage collection.
- C# provides common API styles: .NET Framework, COM, Automation, and C-style APIs. C# supports unsafe mode which let us use pointers to read and write memory that is not under control of the garbage collector.

Reflection is also a feature of the C# language. It is the process by which a program can read its own metadata. It is a powerful mechanism, which allows program to inspect type information and invoke methods on those types at runtime. Reflection is generally used for any of the following four tasks [Lib01]:

- Viewing metadata: Type's metadata can be explored with reflection.
- Performing type discovery: Performing type discovery: Reflection allows program to examine the types in an assembly and interact with or instantiate those types. It is used in creating custom scripts.
- Late binding to methods and properties: Reflection allows the programmer to invoke properties and methods on objects instantiated dynamically based on type discovery. It is also called dynamic invocation.
- Creating types at runtime: Reflection can be used to create new types at runtime and then to use those types to perform tasks. We can use it when a custom class is created at runtime, and it runs significantly faster than more generic code is created at compile time.

In our program, we use the `System.Reflection` namespace and functions as follows:

- **MethodInfo**: Used to discover the attributes of a method such as the name, return type, parameters, and it provide access to method metadata.
- **GetType**: Used to get the runtime type of the current instance.
- **GetMethod**: Used to explore method of a Type object to invoke a specific method.
- **Invoke**: Used to invoke the method or constructor reflected by the MethodInfo instance.

In the following example, we will show how to use the functions of reflection in a program.



```

using System;
using System.Reflection;

namespace TestReflection
{
    class Class1
    {
        public static void Main(string[] args)
        {
            try
            {
                //Initialize input parameters
                Class2 testObject=new Class2();
                Object input1=testObject;
                string input2="JoinName";
                object[] input3=new object[2];
                input3[0]="Jason";
                input3[1]="Wang";

                //Invoke method MethodCheck in Class1
                Class1.MethodCheck(input1, input2,
                                   input3);
            }
            catch(Exception e)
            {
                Console.WriteLine("Exception:"
                                   +e.Message);
            }
        }

        //Check and invoke the specified method
        public static void MethodCheck(Object obj, string
                                       methodName, Object[] methodArguments)
        {
            //Get the types of method arguments
            Type[] arguTypes=new Type
                [methodArguments.Length];
            for(int i=0; i<methodArguments.Length; i++)
                arguTypes[i]=
                    methodArguments[i].GetType();
        }
    }
}

```

```

        //Evaluate whether the method has
        //been find from the specified object
        if (methodInfo != null)
        {
            //If found it, invoke the method of
            //the specified object
            Console.WriteLine("The method has been
                               found.");
            methodInfo.Invoke(obj,
                              (Object[])methodArguments);
        }
        else
        {
            Console.WriteLine("Can not find the
                               method");
        }
    }
}

class Class2
{
    public void JoinName(string FirstName,
                        string LastName)
    {
        //Join fist and last name, and show them
        Console.WriteLine("The first name is: {0}",
                          FirstName);
        Console.WriteLine("The last name is: {0}",
                          LastName);
        Console.WriteLine("The full name is: {0}",
                          FirstName+LastName);
    }
}
}

```

This program includes two classes: `Class1` and `Class2`. The method `MethodCheck` in class `Class1` is used to check and invoke a specified method of an object. The method `JoinName` in class `Class2` is used to join the first and last name and then display them. In method `MethodCheck`, the input parameter `obj` is an object name, the `methodName` is a

method name, and the `methodArguments` is the input parameters of method `methodName`. In this method, we use the following syntax to bring the types of `methodArguments` into an array `arguTypes`, the function `GetType()` is used to get the runtime type of the current object `methodArguments[i]`.

```
Type[] arguTypes=new Type[methodArguments.Length];
for(int i=0; i<methodArguments.Length; i++)
    arguTypes[i]=methodArguments[i].GetType();
```

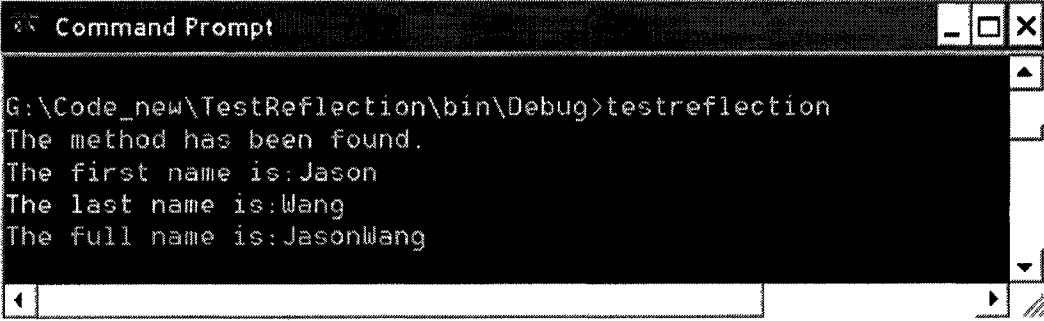
Then we use the following syntax to get the information of method `methodName` from the specified object `obj`, and keep the information in `MethodInfo`. The function `GetMethod()` is used to explore method of a specified object.

```
MethodInfo methodInfo=obj.GetType().GetMethod(methodName,
        (Type[]) arguTypes);
```

Finally we evaluate the value of `MethodInfo`. If it is not null, it means the method has been found from the specified object. We use the function `Invoke()` to invoke this method reflected by the `MethodInfo` instance. The syntax is shown as follows:

```
if(methodInfo!=null)
{
    Console.WriteLine("The method has been found.");
    methodInfo.Invoke(obj, Object[])methodArguments);
}
else
{
    Console.WriteLine("Can not find the method");
}
```

The result of this program is now:



```
Command Prompt
G:\Code_new\TestReflection\bin\Debug>testreflection
The method has been found.
The first name is:Jason
The last name is:Wang
The full name is:JasonWang
```

## 5.2 Implementation of Preprocessor

The preprocessor includes two parts: Process1 and Process2. It completes the translation of the input program. Process1 is used to analyze and record the information of input program. The recorded information will be used for Process2. Process2 is used to translate input program to standard C# code. A detailed discussion to program structure will be given at follows.

### 5.2.1 The First Process of the Preprocessor

The first process for an input program is the method Process1 of class Process that mainly focuses on the forepart preparation for the program translation. The functions of Process1 are presented as follows.

- By analyzing the sentence that contains the keyword `dynamic` in the input program, preprocessor creates the dynamic classification relation table and saves the table in the specified file `dynamic_class_info.dat`. This file includes the information of class name, label and dynamic child class. It will be used to translate input program in Process2. For example, we create the content in Table 5.2.1.1 from the following dynamic classification definition sentence.

```

// class Person
public class Person dynamic Student, Employee
{
    public string name, sex, address;
}

// class Employee
class Employee:Person dynamic Manager label [technicManager,
                                             projectManager]
{
    public string employeeNumber;
    public string salary;
    public string GetSalary()
    {
        return this.salary;
    }

    public void SetSalary(string s)
    {
        this.salary = s;
    }
}

```

Class Name	Label	Dynamic Child Class
Person	student	Student
Person	employee	Employee
Employee	technicManager	Manager
Employee	projectManager	Manager

Table 5.2.1.1 dynamic\_class\_info. dat structure

- By analyzing the sentence containing the keyword `new` in the input program, preprocessor saves the name of new object and its class name in global variable `newObjectArray`. The `new` is a keyword in standard C# language, which is used to create objects and invoke constructors. The information in variable `newObjectArray` will be used for `Process2` later. For example, we analyze the following sentence and save the name of object `jason` and class

GraduateStudent in variable newObjectArray as (jason, GraduateStudent).  
It shows that object jason is an instance of class GraduateStudent.

```
GraduateStudent jason = new GraduateStudent();
```

- By analyzing the sentence containing the keyword `newChild` in the input program, preprocessor saves the name of new object and its class name in global variable `newObjectArray`, which is used for the `Process2` later. The `newChild` is a keyword in dynamic classification grammar. It is used to create primary and dynamic objects from classes. In the following demo codes, the first statement creates object `e` of class `Employee` that depends on dynamic classification relation `(Person, employee, Employee)`, and lets the object `e` as a dynamic child of object `jason`. For the first line, preprocessor saves the name of object `e` and class `Employee` in variable `newObjectArray` as `(e, Employee)`. The second line doesn't define a new object name, so preprocessor will create the object name automatically. For the second line, preprocessor records nothing in variable `newObjectArray`.

```
e = jason newChild (Person, employee, Employee);  
e newChild ( deptManager);
```

- Analyzing each class definition and save the class information in the specified file `class_info.dat`. It includes the class name, superclass name, variable number and variables, method number and methods. This file will be invoked in `Process2`. For example, we can create the table in the Table 5.2.1.2 from the following class definition sentence:

```

// class GraduateStudent
class GraduateStudent : Student
{
    public string professorName = null;
    public string thesisSubject = null;
    public void show()
    {
        Console.WriteLine("Professor Name: {0}",
                           professorName);
        Console.WriteLine("Thesis Subject: {0}",
                           thesisSubject);
    }
}

```

<b>Class Name</b>	GraduateStudent
<b>Superclass Name</b>	Student
<b>Variable Number</b>	2
<b>Variables</b>	professorName, thesisSubject
<b>Method Number</b>	1
<b>Methods</b>	show()

Table 5.3.1.2 class\_info.dat structure

## 5.2.2 The Second Process of the Preprocessor

The method Process2 of class Process implements the second process for input program. It uses the information from Process1 and focuses on the program transition. The functions are shown as following.

- Translate dynamic classification definition sentence.  
Searching the keyword `dynamic` to find the dynamic classification definition sentence and translate the input program to an output file. In chapter 4.2.1, we introduced how to use the keyword `dynamic` and `label` in object-oriented programming language to define dynamic classification relation( $C, I, R$ ). As

these keywords are not recognized by standard C# but defined by ourselves, so a transition is required before compiling the input program. In preprocessor, both the first and second process are responsible for analyzing the dynamic classification relation, but with different purpose. The relationship analyzed in Process1 is used for recording the relationship information of the input program. In Process2, it is used for the translation of input program. We explain it by the example below.

```
// class Person
public class Person dynamic Student, Employee
{
    ...
}

// class Student
class Student:Person dynamic GraduateStudent
{
    ...
}

// class Employee
class Employee:Person dynamic Manager label
                                [technicManager, projectManager]
{
    ...
}

// class GraduateStudent
class GraduateStudent:Student
{
    ...
}

// class Manager
class Manager:Employee
{
    ...
}
```



This demo code defines five classes: Person, Student, Employee, GraduateStudent and Manager. The dynamic classification relations between them are illustrated in the Figure 4.1.4. Before we explain how to translate this code, we consider how to record relations between these classes in run time first. Therefore we define a new class DynaClassification and add it to the output file. The methods of class DynaClassification is shown in Table 5.2.2.1.

Methods	Purpose
DynaClassification()	Record the dynamic classification relation
Equals()	Compare different dynamic classification relations
MethodSearch()	Search for maximal subobjects of specified object that include the invoked method
FieldSearch()	Search for maximal subobjects of specified object that include the invoked field
ExecuteMethod1()	Invoke method (no output parameter)
ExecuteMethod2()	Invoke method (with output parameter)
ExecuteField1()	Invoke field (no output parameter)
ExecuteField2()	Invoke field (with output parameter)

Table 5.2.2.1 Class methods of DynaClassification

Here we just discuss the method DynaClassification and Equals, and the other methods will be explained in the following part. The method DynaClassification is used to record the dynamic classification relation between the classes, with variables C, l and D separately represent the dynamic parent, label and dynamic child in the dynamic classification relation. The method Equals is inherited from class Object and used to compare different dynamic classification relations. The method DynaClassification and Equals are defined as:

```

using System;
using System.Reflection;
using System.Collections;

// class DynaClassification
class DynaClassification
{
    public string label;
    public Type parent, child;

    // Used to record the dynamic classification relation
    public DynaClassification(Type C, string l, Type D)
    {
        parent = C;
        label = l;
        child = D;
    }

    // Used to compare different dynamic classification
    // relations
    public bool Equals(DynaClassification dynac)
    {
        If (parent.Equals(dynac.parent)&&label.Equals
            (dynac.label)&&child.Equals(dynac.child))
            return true;
        return false;
    }
    ...
}

```

So we can use an instance of class `DynaClassification` to record the dynamic classification relation of input program. The way for recording is to define an `ArrayList dynamics` for dynamic classification relation and save each dynamic superclass-subclass relation in it. If a class does not have a superclass, which means the class is the root node in the dynamic classification relation. We define a new `ArrayList dynamics` in the class and define an interface `Parent` as the superclass of this class. Whereas if a class has a superclass, which means the class is the leaf node in the dynamic classification relation, the class can inherit the

variable `dynamics` from its superclass. Therefore we can get all the dynamic classification relations from the variable `dynamics`. Therefore, the dynamic classification definition sentence listed above can be translated to the following C# code:

```
using System.Collections;

// interface Parent
interface Parent
{
    ICollection GetChildren();
}

// class Person
public class Person:Parent
{
    internal Parent parents;
    internal DynaClassification dyna;
    internal Hashtable children= new Hashtable();
    public static ArrayList dynamics;

    static Person()
    {
        dynamics = new ArrayList();
        dynamics.Add(new DynaClassification(
            typeof(Person), "student", typeof(Student)));
        dynamics.Add(new DynaClassification(
            typeof(Person), "employee", typeof(Employee)));
    }
    public ICollection GetChildren()
    {
        ICollection children_values = children.Values;
        return children_values;
    }
    ...
}

//class Student
class Student : Person
{
    static Student()
    {
```

```

        dynamics.Add(new DynaClassification(
            typeof(Student), "graduateStudent",
            typeof(GraduateStudent)));
    }
    ...
}

// class Employee
class Employee:Person
{
    static Employee()
    {
        dynamics.Add(new DynaClassification(
            typeof(Employee), "technicManager",
            typeof(Manager)));
        dynamics.Add(new DynaClassification(
            typeof(Employee), "projectManager",
            typeof(Manager)));
    }
    ...
}

// class GraduateStudent
class GraduateStudent:Student
{
    ...
}

// class Manager
class Manager:Employee
{
    ...
}

```

- Translate the `newChild` statement sentence.

Searching the keyword `newChild` to find the `newChild` definition sentence and translate it correspondingly. In chapter 4.2.2 we introduced how to use the

keyword `newChild` in object-oriented programming language to define a dynamic object. Now preprocessor opens file `dynamic_class_info.dat` that is created in `Process1`. From this file, preprocessor can get dynamic classification relation between classes. Depending on the relation, a `newChild` definition sentence will be translated into some standard `C#` sentences which include creating object and describing dynamic classification relation. For Example, the expression

```
obj' = obj° newChild (C, l, C' );
```

creates a dynamic object `obj'` with class `C'`. Object `obj'` is a subobject of object `obj°`, and they are related with dynamic classification relation `(C, l, C')`. In this expression, we have defined the subobject name `obj'`. If it is not defined, the preprocessor will automatically create a new object name. The following input code is an example.

```
using System;
using System.Reflection;
using System.Collections;

public class DemoCode
{
    public static void Main(String[] args)
    {
        ...
        // Sentence 1
        GraduateStudent jason = new GraduateStudent();
        // Sentence 2
        e = jason newChild (Person, employee, Employee);
        // Sentence 3
        e newChild (technicManager);
        ...
    }
    ...
}
```

In the above codes, object `jason` is created by class `GraduateStudent`, object `e` is created by class `Employee`, and there exists a dynamic parent-child relation between the object `e` and `jason`, where object `jason` is a primary object and object `e` is a dynamic object, and they are related with dynamic classification relation (`Person`, `employee`, `Employee`). The relationship between object `jason` and `e` will be recorded and used for the following program at runtime. Here we put the relation in Hashtable `children` which is defined in the root class of the dynamic superclass-subclass relation. The following is a translation C# code of the input code above.

```
using System;
using System.Reflection;
using System.Collections;

public class DemoCode
{
    public static void Main(String[] args)
    {
        ...
        // Sentence 1
        GraduateStudent jason = new GraduateStudent();

        // Sentence 2
        Employee e = new Employee();
        DynaClassification temp_Dyna_0 = new
            DynaClassification (typeof(Person),
                "employee", typeof(Employee));
        jason.children.Add(temp_Dyna_0, e);

        // Sentence 3
        Manager temp_deptManager0 = new Manager();
        DynaClassification temp_Dyna_1 = new
            DynaClassification(typeof(Employee),
                "technicManager", typeof(Manager));
        e.children.Add(temp_Dyna_1, temp_deptManager0);
        ...
    }
}
```

- Translate the `removeChild` sentence.

Searching keyword `removeChild` to find the `removeChild` sentence and translate it correspondingly. The `removeChild` sentence is used for declassify operation of the dynamic classification relation. The following sentence is an example:

```
jason removeChild;
```

This input code can be translated to the following C# code. It removes the object `jason` from Hashtable `children`, and disconnects the relation between the object `jason` and its dynamic child object. The parameter `temp_Dyna_0` is an instance of class `DynaClassification` that recorded the dynamic classification relation between the classes.

```
jason.children.Remove(temp_Dyna_0);
```

- Method and field operate and access

In this step, preprocessor gets a list of `new` objects and `newChild` object from global variable `newObjectArray` which is created in `Process1` and use the list to analyze input program. When preprocessor find a method that invoked sentence of an object, it opens file `class_info.dat` that is created in `Process1` to analyze the class of this object that belongs to a system class or user created class. If preprocessor cannot find the invoked method from this class, it will check the superclass of this class depending on recorded information in file `class_info.dat`. Finally, a result will be returned, which shows whether the class is found or not. Preprocessor that depends on the result translates the input code.

In translation codes, we use the reflection function of C# language to implement the operation of method and field at run time. We define other three methods in the class `DynaClassification` for method operation: `MethodSearch`, `ExecuteMethod1` and `ExecuteMethod2`, which are responsible for method check and method access. The `MethodSearch` use the reflection function to search whether an object or subobjects of this object includes the specific method in real time. The `ExecuteMethod1` and `ExecuteMethod2` are used for setting the operation and getting the values from the methods. We also define three methods in the class `DynaClassification` for field operation: `FiledSearch`, `ExecuteField1` and `ExecuteField2`, which are responsible for field check and access. The operation of field is similar to method operation. The methods of method operation in class `DynaClassification` are shown as follows.

```
using System;
using System.Reflection;
using System.Collections;

// class DynaClassification
class DynaClassification
{
    ...
    // Search for maximal subobjects of specified object
    // that include the invoked method
    public static Object MethodSearch(Object objectName,
                                     string methodName, Object[] arguments)
    {
        Type[] arguTypes = new Type[arguments.Length];
        for(int i=0; i<arguments.Length; i++)
            arguTypes[i] = arguments[i].GetType();
        ArrayList objReturn = new ArrayList();
        ArrayList objList = new ArrayList();
        objList.Add(objectName);
        While(objList.Count != 0)
        {
            ArrayList S=new ArrayList(objList);
```



```

objList = new ArrayList();
for(int i=0; i<S.Count; i++)
{
    try
    {
        MethodInfo SearchMethodInfo=
            S[i].GetType().GetMethod
            (methodName, (Type[]) arguTypes);
        if(SearchMethodInfo!=null)
        {
            objReturn.Add(S[i]);
        }
        else
        {
            objList.AddRange(((Parent)
                S[i]).GetChildren());
        }
    }
    catch(MethodAccessException MAE)
    {
        Console.WriteLine("MAE:"+
            MAE.Message);
    }
    catch(Security.SecurityException SE)
    {
        Console.WriteLine("SE:"+
            SE.Message);
    }
}
}
if(objReturn.Count==1)
    return objReturn[0];
return null;
}
// Invoke method (no output parameter)
public static bool ExecuteMethod1(object objectName,
    string methodName, object[] arguments)
{
    try
    {
        Object obj=DynaClassification.MethodSearch(
            objectName, methodName, arguments);
    }
}

```

```

        if(obj!=null)
        {
            Type[] arguTypes = new Type
                [arguments.Length];
            for (int i=0;i<arguments.Length;i++)
                arguTypes[i]=
                    arguments[i].GetType();
            ((MethodInfo) obj.GetType().GetMethod(
                methodName, (Type[]) arguTypes)).
                Invoke(b, (Object[]) arguments);
        }
        return true;
    }
    catch (Exception e)
    {
        Console.Out.WriteLine("Error:" + e.Message);
        return false;
    }
}

// Invoke method (with output parameter)
public static bool ExecuteMethod2(object objectName,
    string methodName, object[] arguments,
    out Type outType, out string outValue)
{
    outType = null;
    outValue = null;
    try
    {
        Object obj=DynaClassification.MethodSearch(
            objectName,methodName, arguments);
        if(obj!=null)
        {
            Type[] arguTypes = new
                Type[arguments.Length];
            for(int i=0;i<arguments.Length;i++)
                arguTypes[i]=arguments[i].
                    GetType();
            OutType = ((MethodInfo) obj.GetType().
                GetMethod(methodName,
                    (System.Type[]) arguTypes)).
                Invoke(obj, (System.Object[])
                    arguments).GetType();
        }
    }
}

```

```

        outValue = ((MethodInfo) obj.GetType().
            GetMethod(methodName, (System.Type[])
            arguTypes)).
            Invoke(obj, (System.Object[])
            arguments).ToString();
    }
    return true;
}
catch (Exception e)
{
    Console.Out.WriteLine("Error:" + e.Message);
    return false;
}
...
}

```

The following is an example, it shows how to translate method operation sentence to C# code.

Example 1:

```

using System;
using System.Reflection;
using System.Collections;

public class DemoCode
{
    public static void Main(String[] args)
    {
        ...
        string salary_set="$8000";
        jason.SetSalary(salary_set);
        ...
    }
}

```

Translate to C# code:

```
using System;
using System.Reflection;
using System.Collections;

public class DemoCode
{
    public static void Main(String[ ] args)
    {
        ...
        string salary_set="$8000";
        string methodname="SetSalary";
        object objectname=jason;
        object[] arguments = new object[1];
        arguments[0] = salary_set;
        DynaClassification.ExecuteMethod1(objectname,
                                          methodname, arguments);
        ...
    }
}
1
```

Example 2:

```
using System;
using System.Reflection;
using System.Collections;

public class DemoCode
{
    public static void Main(String[] args)
    {
        ...
        string salary_get = jason.GetSalary();
        ...
    }
}
```

Translate to C# code:

```
using System;
using System.Reflection;
using System.Collections;

public class DemoCode
{
    public static void Main(String[] args)
    {
        ...
        string methodname = "GetSalary";
        object objectname = jason;
        object[] arguments = new object[0];
        Type outputType;
        string outputArgument;
        DynaClassification.ExecuteMethod2(objectname,
                                         methodname, arguments, out outputType,
                                         out outputArgument);
        string salary_get = outputArgument;
        ...
    }
}
```

### 5.3 Restrictions and Assumptions of Syntax

This project demonstrates how to realize the function of dynamic classification with the existing object-oriented programming language C#. It is an implementation of the new approach mentioned in chapter 4. For easily understanding the program, some restrictions are defined as following:

- Extend C# with keywords `dynamic`, `label`, `newChild` and `removeChild`.
- The variables in C# language can be classified as two types: simple variable type and complex variable type, as shown in Table 5.3.1. Our program only uses the simple variable types by now.

<b>Simple Variable Types</b>	int,uint,long,ulong,short,ushort,byte,sbyte bool,char,string,float,double,decimal
<b>Complex Variable Types</b>	Enumerations, Structures, Arrays

Table 5.3.1 Variable types

- The elements in class include behaviors and attributes. In C# we use methods to define behaviors of a class, fields and properties to define the attributes. The fields are defined using standard variable declaration format. The properties are defined in a similar way to fields, but they act like methods. In our program, the variables can only be defined in the class body and out of the method body, as fields in the C#. The following program describes the difference between the fields and properties.

Example (Field):

```
class testField
{
    public int testInt;
    public char testChar;
}
```

Example (Property):

```
class testProperty
{
    private int testInt;
    public int TestInt
    {
        get
        {
            return testInt;
        }
        set
        {
            testInt=value;
        }
    }
}
```

- The use of the dynamic classification object is only limited to the basic expression which include the = assignment operator.

```
<variable> = <object>.<field | method>;
<object>.<field> = <variable>;
```

## 5.4 How to Use the Preprocessor

The usage of the preprocessor consists of the following three steps:

- Create C# source code files. We can compose a C# code which includes the dynamic classification feature in a text editor. The code can be saved as any file format.
- Translate the C# source code. We use the preprocessor to translate the source code to a standard C# code which can be recognized by a C# compiler. The preprocessor is a C# code translation tool in DOS command window. There are several ways to execute preprocessor. exe, correspondingly get different results as show below.

a) Generate the help information:

```
C:\>preprocessor -?
Usage: Preprocessor [-f<output-pathname>] input-pathname...
-? Show this usage information
-f Send output to specified pathname instead of the console
```

b) Run preprocessor. exe with input file name:

If the input file is formatted as \*.cs, it will generate an output file test\_dc. cs under the current directory, e.g.

```
C:\>Preprocessor testcode.cs
```

If the input file is formatted as any other file formats, it will also generate output file test\_dc.cs, e.g.

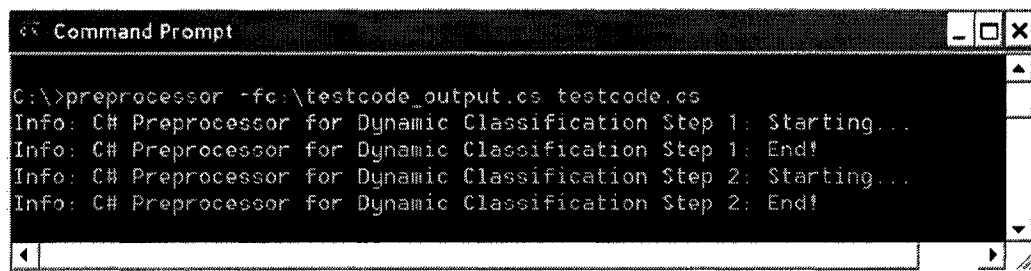
```
C:\>Preprocessor testcode.txt
```

c) Run preprocessor with input and output file names:

```
C:\>preprocessor.exe -fc:\testcode_output.cs testcode.cs
```

The preprocessor will process the input file testcode.cs and the output file testcode\_output.cs will be put into c:\.

If the preprocessor runs successfully, the following results will be shown:



```
Command Prompt
C:\>preprocessor -fc:\testcode_output.cs testcode.cs
Info: C# Preprocessor for Dynamic Classification Step 1: Starting...
Info: C# Preprocessor for Dynamic Classification Step 1: End!
Info: C# Preprocessor for Dynamic Classification Step 2: Starting...
Info: C# Preprocessor for Dynamic Classification Step 2: End!
```

That means the input code has been processed two times and an output file has been generated.

- Compile the output source code. We use the compiler provided by C# language to compile .cs file to generate the executed file.



csc testcode\_dc.cs

## Chapter 6 Conclusions

### 6.1 Conclusions

In this paper, we introduce a new approach and discuss the theory of dynamic classification in detail based on the knowledge of precious research. The research of the dynamic classification can be divided into two major parts: from theory perspective and from implementation perspective.

- From theory perspective

Researchers present many approaches for dynamic classification. Bachman and Daya [BD77] first extended the network data model with a role concept. Wieringa et al. [WJS94] propose an order-stored logic dynamic database to implement class migration and role-playing. Sciore et al. [Sci89] presents the object specialization based on prototype. Richardson et al. [RS91] works on the aspect of programming. Fower [Fow97] presents the five types of analysis pattern, etc.

- From implementation perspective

Many papers discuss the concrete implementation of dynamic classification in existing computer language. Gottlob et al. [GSR96] researches on the implementation of roles in Smalltalk and Fibonacci [ABG+93] [AGO95], Kendall [Ken99] implement the role model in the AspectJ. Depke et al. [DEK00] conducts the research on how to implement dynamic classification in UML. Drossopoulou et al. [DDD+01] develops a type and effect system for language DoR and Fickle to testify their approach, etc.

Based on the existing research achievement, we study object-oriented programming language with dynamic classification, and get a totally comprehensive understanding on this area. I focus on my thesis from two parts:

- From the aspect of theory: do more research on object-oriented programming with dynamic classification, and complete the theory by a new approach.
- From the aspect of implementation: base on the complete understanding of existing theories, implement dynamic classification by object-oriented programming language C#. I develop a preprocessor, by which a C# code including the extendable dynamic classification functions can be translated to standard C# code. The export code can be compiled by a C# compiler.

## **6.2 Future Work**

Up till now, a lot of research achievements of dynamic classification have improved the implementation, but still there are some problems, such as:

- To improve the existing theory.
- To implement the theory in programming language.
- To study on the system efficiency, which is one of the reasons that many existing programming languages do not support dynamic classification?

## Reference

1. [ABG+93] *A. Albano, R. Bergamini, G. Ghelli and R. Orsini. An Object Data Model with Roles. Proceedings of the Nineteenth International Conference on VLDB, pages 39-51, Dublin, Ireland, 1993.*
2. [BD77] *C. Bachman, M. Daya. The Role Concept in Data Models. In Proceedings of the 3rd International Conference on VLDB, pages 464-476, 1977.*
3. [BRS+97] *D. Bäumer, D. Riehle, W. Siberski and M. Wulf. The Role Object Pattern. Proceedings of the 4th Annual Conference on the Pattern Languages of Programs, 1997.*
4. [DDD+00] *S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, P. Giannini. Dynamic Object Reclassification. October 30, 2000.*
5. [DDD+01] *S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, P. Giannini. Fickle: Dynamic Object Reclassification. In Electronic Proceedings of The Eighth International Workshop on Foundations of Object-Oriented Languages (FOOL 8), London, England, January 20, 2001, Available at <http://www.cs.williams.edu/~kim/FOOL/sched8.html>.*
6. [DH98] *E. R. Doke, B. C. Hardgrave. An Introduction to Object COBOL. John Wiley & Sons, Inc. 1998.*
7. [Fow97] *M. Fowler. Dealing with Roles. Proceedings of the 4th Annual Conference on the Pattern Languages of Programs, Monticello, Illinois, USA, Sept. 1997.*
8. [Gam96] *E. Gamma. The Extension Objects Pattern. Submitted to PloP 1996.*
9. [GSR96] *G. Gottlob, M. Schrefl, B. Rock. Extending Object-Oriented Systems with Roles. ACM Transactions on Information Systems, Vol. 14, No. 3, pp 268–296, July 1996.*
10. [KA95] *S. Khoshafian, R. Abnous. Object Orientation - second edition. John Wiley & Sons, Inc. 1995.*

11. [Ken99] *E. A. Kendall*. Role Model Designs and Implementations with Aspect-oriented Programming. In *OOPSLA'99*, pp.353-369, Denver, CO, USA, November 1999.
12. [Kri95] *B. B. Kristensen*. Object-Oriented Modeling with Roles. *OOLS'95, Proceedings of the 2nd International Conference on Object-oriented Information Systems, 1995*.
13. [Kri96] *B. B. Kristensen*. Roles: Conceptual Abstraction Theory & Practical Language Issues. *Theory and Practice of Object Systems*, 2:143-160, 1996.
14. [Li02] *L. W. Li*. Extending Object-Oriented Programming Languages with Dynamic Classification. *Submitted to Journal, 2002*.
15. [Lib01] *Jesse Liberty*. Programming C#. July 2001.
16. [MO98] *J. Martin, J. J. Odell*. Object-Oriented Methods: A Foundation. *Prentice Hall, 1998*.
17. [Msd04] *Msdn*. Visual C# Language Concepts: C# Language Tour. *Microsoft Corporation 2004*.
18. [Nor96] *R. J. Norman*. Object-Oriented Systems Analysis and Design. *Prentice Hall, 1996*.
19. [Rag81] *F. G. Ragan*. Formal Specification of programming language. *Prentice – Hall, Inc., 1981*.
20. [RG98] *D. Riehle, T. Gross*. Role Model Based Framework Design and Integration. In *Proceedings OOPSLA '98 ACM SIGPLAN Notices*, pages 117--133, Oct. 1998.
21. [RS91] *J. Richardson, P. Schwarz*. Aspects: Extending Objects to Support Multiple, Independent Roles. In *Proc. Intl. Conf. on Management of Data, ACM*, pp 298-307, 1991.
22. [Sch96] *A. Schoenfeld*. Domain Specific Patterns: Conversions, Persons and Roles, and Documents and Roles. [www.cs.wustl.edu/~schmidt/Plop-96/schoenfeld.ps.gz](http://www.cs.wustl.edu/~schmidt/Plop-96/schoenfeld.ps.gz), 1996.
23. [Sci89] *E. Sciore*. Object Specialization. *ACM Transactions on Information Systems*, 7(2): pp 103-- 122, April 1989.
24. [Wat02] *Karli Watson*. Beginning Visual C#. *Wrox Press Ltd., 2002*.

25. [WJS94] R. Wieringa, W. de Jonge, P. Spruit. Roles and Dynamic Subclasses: A Modal Logic Approach. *Proceedings of the 8th European Conference on Object-Oriented Programming, ECOOP'94, Springer-Verlag, Bologna, Italy, July 1994.*
26. [Wu99] C. Thomas Wu. *An Introduction to Object-Oriented Programming with Java. McGraw Hill 1999.*

## **VITA AUCTORIS**

Wenjiang Wang was born in 1969 in Tianjin, P.R.China. He graduated from Beijing Technology and Business University where he obtained a B.E. in Computer and Application in 1992. He is currently a candidate for the Master's degree in Computer Science at the University of Windsor and hopes to graduate in Summer 2004.