OBJECT MIGRATION IN A CLASS-BASED OWNERSHIP ENVIRONMENT

A THESIS

Submitted by

D.S. PRADEEP KUMAR

in fulfilment for the award of the degree

of

MASTER OF SCIENCE (BY RESEARCH)

FACULTY OF INFORMATION AND COMMUNICATION ENGINEERING

ANNA UNIVERSITY CHENNAI

CHENNAI - 600 025

JUNE 2009

ANNA UNIVERSITY CHENNAI
CHENNAI - 600 025

BONAFIDE CERTIFICATE

Certified that this thesis titled **"OBJECT MIGRATION IN A CLASS-BASED OWNERSHIP ENVIRONMENT"** is a bonafide work of **Mr. D.S. PRADEEP KUMAR**, who carried out the research work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

Place : Chennai 600 025
Date :

Dr. SASWATI MUKHERJEE
(SUPERVISOR)
Assistant Professor
Department of Computer Science and Engineering
College of Engineering
Anna University
Chennai 600 025.

ABSTRACT

A widely followed approach for designing a secure, structured environment is by using a combined technique of static typing and modularity. In practice, programmers mostly prefer class-based programming languages like C++, Java, etc., for designing such an environment.

Class-based languages are good in (1) Encapsulation, which helps in separating components of composite objects and hence components can be accessed only by feature calls to these composite object, (2) Class inheritance, which helps in modifying the implementation being reused from the parent classes.

On the other hand, the approach for designing dynamic evolution environment is object composition, an alternative to class inheritance, where new functionality is obtained by assembling or composing objects to get more complex functionality.

In object-oriented programming languages, aliasing is considered as a major problem, which permits unauthorized access to the data structure nodes. Thus aliasing breaks encapsulation and information hiding principle of object oriented programming environment. In this thesis, ownership is used for encapsulation. In ownership encapsulation model, the owner gives a logical boundary specifying how communication should take place between objects inside the owners' encapsulation boundary and objects outside the

owners' boundary. In particular, ownership allows one to confine an object inside a data structure and to prevent representation exposure through leaking thereby solving the problem of aliasing between ownership contexts.

This, we expect, will help in mapping the flux of the real world under the technique of secure programming where both encapsulation and object composition will be provided. This thesis exploits the concepts of ownership types along with object migration to provide for such an environment.

ACKNOWLEDGEMENT

I express my profound gratitude to my revered supervisor **Dr. Saswati Mukherjee,** Assistant Professor, Department of Computer Science and Engineering, CEG, Anna University, Chennai, for her valuable guidance, suggestions, constant support and tireless efforts to complete my research work.

I would like to thank **Dr. C. Chellappan**, Head, Department of Computer Science and Engineering for their kind cooperation to carryout the research work at Anna University Campus.

I wish to thank my Monitoring Committee members **Dr. T.V. Geetha,** and **Dr. C. Pandurangan** for their remarkable suggestions and motivation in this work.

I thank my beloved parents, brothers, sisters, relatives and friends for their moral support and encouragement to complete this research work successfully.

D.S. PRADEEP KUMAR

TABLE OF CONTENTS

CHAPTER NO. TITLE PAGE NO.

LIST OF FIGURES

LIST OF SYMBOLS AND NOMENCLATURE

Δ	-	Delta
Γ	-	Domain
Jm	-	Jmigrate
κ	-	Kappa
OTM	-	Ownership Transfer Model
θ	-	Theta

CHAPTER 1

INTRODUCTION

1.1 PROGRAMMING METHODOLOGY

Every programming environment can be classified into two depending on software engineering and security concern. The software engineering concentrates on building large software system that maps the real world situation. Their focus is mainly on s*eparation of concern and reusability*, thereby helping in the development of independent modules and sharing components. Separation of concern is a powerful abstraction mechanism that helps in analyzing problems within separate context. The concept of reusability helps in developing independent reusable component that can be easily used in more than one context. On the other hand, security mechanism segregates the runtime environment into domains, where granting permissions or other means of access control policies can facilitate secure inter and intra domain accesses. However, in general, it becomes important to choose a proper programming language environment to map the real world concepts into practice. It is observed that the current object-oriented language environment reveals an apparent dichotomy between class-based programming language system and prototype-based programming language system. These classic mechanisms differ significantly in flexibility, robustness, and in providing safety guarantees.

A real-world entity is modeled by a single object, which is assigned to a class when it is created. Each object in a class has exactly the same set of variables and methods. An object cannot individually define other variables or

methods, nor can it change the way in which variables and methods are inherited from other classes. One consequence of this approach is that all members of a class have a uniform structure. Thus the objects in a class can be stored efficiently through a shared representation, and the set-oriented access of class members is made more efficient. Parallel processors can also take advantage of this uniform structure, further enhancing efficiency. This efficiency becomes increasingly important as classes contain larger numbers of objects.

1.2 TRADITIONAL SOFTWARE DESIGN QUALITY

A fundamental goal of software design is to structure product in order to reduce the complexity of interconnections between modules. Hence, for that purpose, designers follow different criteria that allow them to reduce the complexity and mutual dependencies between cooperating parts of the code. The software design qualities used traditionally for this purpose are *coupling* and *cohesion* as given by Stevens et al in (Stevens et al 1974).

Coupling refers to the interrelated aspects of different parts of code and it is desirable to have coupling at its minimum. When the system has high coupling, one module modifies or relies on the internal workings of another module. On the other hand, in case of low coupling modules are not dependent on each other; instead they use a public interface to exchange parameterless messages (or events).

Whenever we make one object dependent on another for its operations - or one system dependent on another for its operations - they are coupled. One problem with coupled systems or objects is that, rather than using defined public interfaces, when one object looks inside of another object for its operation, changes in the internal operation of that object can make the other object operate incorrectly.

Cohesion describes relatedness of the steps that the designer puts into the same module. Cohesive implies that a certain class performs a set of closely related actions. Lack of cohesion, on the other hand, means that a class is performing several unrelated tasks. Cohesion and reusability are studied in detail in (Bieman and Kang 1995). Bieman and Byung-Kyoo Kang treat the method and instance variable class components as the key class units that may or may not be connected. A method and an instance variable are related by the way that an instance variable is used by the method. Two methods are related (connected) through instance variable(s) if both methods use the same instance variable(s). Using this orientation, class cohesion can be measured by the relative connectivity (through instance variables) of the methods.

Class cohesion refers to the relatedness of visible components of the class which represent its functionality. Class cohesion is the measure of the degree of relatedness of these components. Thus it becomes clear from the above context that, in a highly-cohesive system, code readability and the likelihood of reuse is increased, while complexity is kept manageable.

There are various levels of coupling and cohesion. In practice, it may be difficult to decide exactly which levels of coupling or cohesion are exhibited by various segments of a system. Nevertheless, the concepts of coupling and cohesion provide a valuable intellectual framework for thinking about software modules and software modularity.

Improving software design quality implies making program modules more independent, making code more self-documented, and making the intent of the designer easily understood. The two characteristics, maintainability and reusability, are the most important characteristics of software quality. In the context of object-oriented (OO) software development, combination of data and operations is represented in larger units

called a class. In OO scenario, cohesion means relatedness of the public functionality of a class whereas coupling stands for the degree of dependence of a class on other classes in the same OO system. An improvement over cohesion and coupling measures of modular design is achieved in object oriented scenario based on "*information hiding* and *encapsulation*".

Encapsulation means that the components of a composite object cannot be modified except by feature calls to the composite object, or by calls from the composite object to its components. Encapsulation limits and controls aliasing and interference, thereby simplifying reasoning and improving understanding of the object-oriented environment. Information hiding, on the other hand, means that the components of a composite object and their states cannot be accessed by clients. Information hiding limits and controls the dependence of clients on the suppliers' (composite objects') internal representation, thereby localizing the effects of changing this representation during system maintenance (Kent and Maung 1995).

A program with high cohesion and low coupling exhibits good encapsulation. In other words, encapsulation is the goal achieved when we are able to reduce coupling. A class with good encapsulation, in turn, lends itself to being cohesive.

Object-oriented programming has two main objectives: to maintain loose coupling between the classes and to build highly cohesive classes. High-cohesion means well-structured classes and loose coupling means more flexible, extensible class.

Following subsection discusses the object-oriented approaches by highlighting their various advantages.

1.3 PRESENT OBJECT-ORIENTED PROGRAMMING METHODOLOGIES

In the global computing scenario, the programming mechanisms can be classified under the following two key properties: secure, structured environment, and dynamic evolution environment.

A widely followed approach for designing the secure, structured environment is by using a combined technique of static typing and modularity. In practice, programmers mostly prefer class-based programming languages like C++, Java, etc., for designing such an environment.

Observations in favour of class-based languages are:

1. Encapsulation (which combines typing and modularization techniques) is strongly supported in static languages than in dynamic languages (Stein et al 1988)
2. Class inheritance (also called implementation reuse or white-box reuse as defined by (Gamma et al 1994)) helps modify the implementation being reused from the parent classes

However, the problem with the class inheritance is its static nature, which makes the inheritance hierarchy to be fixed for the lifetime of object and hence cannot be changed during run-time.

On the other hand, the approach for designing dynamic evolution environment is object composition, an alternative to class inheritance, where new functionality is obtained by assembling or composing objects to get more complex functionality. Object composition helps in designing environments with changing or unknown requirements or one that interacts with other systems that change unpredictably (Gamma et al 1994).

An observation in favor of object composition is:

It has the ability to facilitate anticipated (when entity relationships are statically known) and unanticipated (when entity relationships can be updated dynamically) evolution environment by changing the behavior being composed at runtime (Gamma et al 1994, Kniesel 1999).

The following two sections discuss in greater details the properties of class-based and object-based programming languages.

1.3.1 Class-based Languages

Class-based languages, as the name suggests, are based on classes. In this language model, classes are the fundamental building blocks of objects. In class-based programming languages, the objects can be distinguished into two kinds: classes and instances of classes (a.k.a. objects). A class itself doesn't do anything - it is a blueprint, a general description of objects to be created from that class. Once a class is instantiated, objects are obtained. Although the variables and methods (also called as signature or type or an interface describing how to use the class), that an object must have is defined by their classes, each object vary from each other in the values of these variables, also called the object's state or instance variables.

The instances of a class are the objects created from the classes. Objects are behavioral units that a programmer can manipulate in order to achieve the intended behavior. Object creation is usually done either by calling a ***new*** method of the class (e.g. in Smalltalk) or by using the ***new*** operator in conjunction with a constructor method of the class (e.g. in C++ and Java). More precisely, ***new*** allocates an attribute record and returns a reference to it. The attribute record contains the initial values and the method code specified by the class. Objects are constructed of fields and a set of

methods to manipulate those fields. An object, IP phone for example, might have fields like phone number, person name, detail status, service provider, etc. and methods such as dialing using name/ID, call transfer, call hold, conference, and multiparty call. An object's state at a certain point in time is the set of values of its variables.

Inheritance

In class-based languages only the class is the reusable part. The class anticipates the structure of the objects generated from the class. This anticipation makes the class-based languages more static. The user defines new classes of objects. The class acts as types for the objects.

A class can inherit behavior from another class, which is called its superclass. Inheritance is the sharing of attributes between a superclass and its subclass. By subclassing a class, the new class automatically inherits its behavior from the superclass, but it can add its own behavior on top of this. It can also override the inherited features if it so wishes. As an example, class IPv6 might inherit behavior from class IPv4, but can add new features in the new class. Both classes will therefore share many features, but the class IPv6 will have more functionality than IPv4. The notion of *subclass* and inheritance is a very important concept. Thus subclass is a vehicle that describes incrementally the extensions and changes to its *superclass*.

A class can have infinitely many subclasses, which can in turn be subclassed. This creates a hierarchy of classes, a taxonomy starting from the topmost class or classes and expanding like a tree. A subclass is said to be more specific, while the generic description can be found in the superclass. A language can have one class from which all classes must initially be subclassed (e.g., the Object class in Java). This results in a taxonomy where there is only one tree.

Subsumption

Subsumption is the ability to use a subclass object where an object of its superclass is expected. As all subclasses of a given class share the same methods and fields, i.e. the same type, similar behavior can safely be accepted. The opposite is not possible, since the superclass does not (necessarily) have all the same methods and fields. If a class IPv4 defines a method network_address, then any subclasses of it, e.g. IPv6, will have network_address and hence an object of a class IPv6 can be used where an object of the class IPv4 is expected. An IPv6 packet is thus subsumed by an IPv4 packet.

The major advantage of the class-based approach is its class-based static type mechanism, the encapsulation model and inheritance mechanism. Static typing is helpful in compile time analysis for errors, and encapsulation is a technique for minimizing interdependencies among separately-written modules by defining strict external interfaces (Snyder 1986). A class is encapsulated if the access to the internal state of the class or its objects is restricted by the definition of the programming language. Such accesses can be made only via the defined external interface. Encapsulation has many advantages in terms of improving the understandability of programs and facilitating program modification. Minimizing the exposure of implementation details in the external interface provided will maximize the advantages of encapsulation. Inheriting the superclass methods and instance variables helps in reusing the classes.

As a general philosophy, static typing ensures that well-typed programs help in detecting programming mistakes early and hence has fewer bugs. In addition, the modularity helps in dividing the environment conceptually into manageable parts, each of which owns separate internal

resources. The inheritance helps in static reusability of the software at the level of classes.

1.3.2 Object-based Languages

The object-based languages, also called prototype-based language, evolved in Lisp, Smalltalk, and artificial intelligence communities. The philosophy of object-based language is to satisfy the extreme flexibility in deciding the object hierarchies. This ensures that an object can evolve dynamically by modifying its lookup path and hence are not fixed as done in class-based inheritance (Borning 1986) (Lieberman 1986). As a consequence, little attention has been given in designing typed object-based languages. Some recent languages like Emerald (Hutchinson 1987), Cecil (Litvinov 2003) and Omega (Blaschek 1994) are the simple typed object-based languages.

The distinction between classes/instance is not needed if the alternative of using *prototypes* is adopted (Lieberman 1986). A prototype represents the default behavior for a concept, and new objects can reuse part of the knowledge stored in the prototype by indicating how the new object differs from the prototype. In class-based approach, the class objects must be created before their instances can be used, and behavior can be associated only with classes. Inheritance based on classes fixes the communication patterns between objects at instance creation time itself. Designing a system representing knowledge incrementally and dynamically modifying concepts is one of the advantages of the prototype-based approach.

Reuse at the object level needs behavior to be shared between objects that already exist. The disadvantage of object-based language is that it does not have the facility of encapsulation that is provided in the class-based languages, or support in a limited manner.

1.3.3 Observations

When specifying and reasoning about real world entity modelling like web development, network programming, AI Robotics, etc, we need both dynamic change in behavior as well as secure structured environment. In such design environment, it is reasonable to expect both class composition (a.k.a. class inheritance) and object composition in a single programming model to map the flux of the real world (object composition) within a secure language property (encapsulation).

However, most of the widely used general purposes object oriented languages like C++ and Java lack the above mentioned flexibility. Attempts to merge the two widely recognized properties have led to weak support either by restricting the users with some form of anticipation or with rigid coding convention (Kniesel 2000, Kniesel 1999).

It is desirable that both encapsulation and object composition be provided to map the flux of the real world under the technique of secure programming. This presents a great challenge for the research world to combine class-based techniques such as encapsulation and inheritance, with prototype based technique such as object composition. Encapsulation mechanism fixes the call graphs and encapsulation policies (method implementation) and on the other hand in the case of dynamic object composition we can not determine the encapsulation policies.

In object-oriented programming languages, aliasing is considered as a major problem (Minsky 1996), which permits unauthorized access to the data structure nodes. (Noble et al 1998) forms the basis for ownership model. In the ownership model (Clarke et al 1998, Clarke and Drossopoulou 2002, Boyapati 2003), the owner gives a logical boundary thereby specifying how communication should take place between objects inside the owners'

encapsulation boundary and objects outside the owners' boundary. In particular, ownership allows one to confine an object inside a data structure and to prevent representation exposure through leaking thereby solving the problem of aliasing between ownership contexts. This has motivated us to use ownership type for encapsulation. However, since ownership fixes the owner of an object for its lifetime and is static, it cannot be changed dynamically. To map the requirements of change in the real world dynamically, we also use object migration or ownership transfer.

Before discussing the focus of our thesis in subsection 1.6, we discuss the ownership type and ownership transfer in the next two subsections.

1.4 OWNERSHIP TYPE ENCAPSULATION MECHANISM

Ownership type is one of the recent techniques that enforces notion of object-level encapsulation (Clarke et al 1998, Clarke and Drossopoulou 2002). In ownership type every object has an owner. The owner is either another object or the predefined constant world for objects owned by the system.

An *ownership context* represents a set of objects with the same owner. There is also a *root ownership context*, which is the set of all objects that have no owner. Each object thus belongs to exactly one ownership context. The contexts form a hierarchy, with the root ownership context at the top.

Aggregate objects are containment constructs that group other objects organized in some manner (Clarke 2001). Ownership types forms an aggregate of objects with owner-as-dominator property. Aggregates typically

support operations to access individual members, and to iterate over all members, as in queries.

The ownership types enforce the property called the owners-as-dominators, where that objects are encapsulated by their owners. The domination property says that every path from the root to an object will pass through its owner in the object graph. The ownership type systems are used to structure the object store into contexts and to restrict references between contexts. Thus it forms a logical boundary which protects internal objects from direct accesses from outside objects. As a result, objects with the same owner object X are in one context, Γ, and X is called the owner of Γ.

The ownership containment is termed as the property where an object is considered to be inside its owner. The ownership gains a strong notion of encapsulation, by preventing access to an object from objects outside its owner and thereby giving encapsulation at the object reference level. This is also called as per-object based encapsulation.

However, the exception with the ownership is its static property, where the owner is fixed for the entire lifetime of an object. Static ownership has the advantage of static predictability of the runtime object graph and hence can be proved safe for runtime deployment. However, they have the disadvantage of fixed ownership for life time of the object. This does not allow the flexibility of object composition.

1.5 OWNERSHIP TRANSFER

Ownership transfer is an important property to map the flux of the real-world entity modeling, but problematic issue which forms a new spectrum of research at present.

Some ownership type systems such as SafeJava (Boyapati and Rinard 2004) support ownership transfer based on unique variables (Boyland and Retert 2005). However, the existing ownership types do not provide any facilities to change the owner of an individual object in unanticipated manner. Instead it supports transfer of externally unique object (Clarke and Wrigstsad 2003) which will transfer context of the owner instead of individual per-object based.

Ownership transfer has many applications.

The owner of an object is first determined when the object is created, yet an object needs to be changed transferred from one owner to another. The need for this *ownership transfer* is illustrated by the following examples:

1. *Merging data structures*: data structures such as lists are merged efficiently by transferring the internal representation of one structure to the context of the other
2. *Work flow system*: tasks in work flow systems are transferred repeatedly from processor to processor.
3. *Object initialization*: constructors often take an existing object as parameter and then capture this object, that is, transfer the ownership to the object being constructed. A special case of object initialization is the Factory pattern, where product objects are created in the context of a global factory and then transferred to the client.

However, to make ownership systems to be practical, they must allow objects to transfer ownership from one owner to another.

1.6 THESIS FOCUS

The discussion of traditional technique suggests that an ideal software engineering model should have all the qualities of both class-based and object-based design. This way we can avoid the overhead in deciding the language to be used for designing any specific applications. With the consolidation of the broad language model, the dynamic state can be predicted easily during static checking itself.

The main focus of this thesis is that ideas and advantages from both the class-based world and prototype-based world be used in programming languages to ensure that both the properties of the software engineering are made available in programming languages, without sacrificing the security and flexibility of either language model. This is achieved under the name of *object migration* where an object can dynamically move from one specialized ownership domain to another related ownership domain at runtime, thereby facilitating structural evolution and behavioral evolution. Structural evolution refers to changes in class relations, while behavioral evolution refers to the dynamic variations of behavior an object may exhibit.

Thus, object migration helps the movement of objects from one environment to another dynamically. This change in the environment dynamically will help in modifying the lookup hierarchy, identity, and the encapsulation properties of an object. We propose to use a combination of ownership with ownership transfer to obtain such an environment. We also look at the problems and use mechanism to avoid some of the security breaches which may happen in object migration like dangling pointer and multiple-class effect. Dangling pointer is the problem of having unknown reference to an object location even after the object is physically removed either by garbage collection or moved physically, and multiple-class problem is the problem associated with class-based language model where classes are

the shared entity for every objects of that class and hence object migration will create side-effects in both these cases.

To this end, we propose a model. Our proposed model, namely Ownership Transfer Model (OTM) exploits this combination and shows how to achieve the flexibility of prototype-based systems without abandoning the advantages of the class-based paradigm along with possibilities of solution of above mentioned problems. The evidence of safety in OTM has been illustrated using a language called Jmigrate (Jm), which is based on the Featherweight Java (FJ) (Igarashi et al 2001).

1.7 OUTLINE OF THE DISSERTATION

The thesis is organized as follows:

The **Chapter 1** details about the motivation behind this research followed by related research survey in the **Chapter 2**.

The concept of ownership transfer and type specifications has been dealt with in **Chapter 3**, which provides a reference model to represent ownership types in a class-based programming language and its encapsulation properties.

Following this, the problems associated with class-based ownership types and the ownership transfer are detailed in **Chapter 4**. **Chapter 5** illustrates the proposed model through the design of a new language named Jmigrate (Jm), which is followed by formal definitions in **Chapter 6**.

Finally, in **Chapter 7**, we summarize the contributions of this dissertation and suggest some future directions for research.

CHAPTER 2

RELATED WORKS

2.1 INTRODUCTION

Change and diversity are intrinsic to the real world which continually evolves in ways that cannot be anticipated.

Often the requirements that a system must meet can change in fully unanticipated ways due to a variety of possible factors ranging from company-internal decision to new legislation. Unanticipated requirements change lead to unanticipated behavior evolution. Such changes must be equally well supported by programming languages and tools just like changes that can possibly be anticipated.

Nevertheless current object oriented languages and methods fall short of allowing unanticipated changes/reuse. Modeling of behavior evolution is mostly inhibited by the inability to reuse and evolve existing software in unanticipated ways.

Generally the object-oriented world is classified broadly as class-based languages (for example Java, C++) and prototype based languages (for example Cecil (Litvinov 2003), Self (Ungar and Smith 1987)). Class-based languages are able to statically enforce invariants, and hence rule out many common errors during compile-time. This makes them to be widely accepted for production programming.

In spite of inherent limitations of class-based model there are ways to express the dynamics of the real world with class-based programs. This is done by following various language independent design patterns (Gamma et al 1994), and language specific solutions given in various research work (Meyer 1992, Meyers 1996) condensed by various authors to frequent problem. In these approaches, the functionality missing from the language is simulated by a set of cooperating classes. However this is not a satisfactory solution (Yu 2001).

This insight led to a variety of proposals for an extension of the class-based model. Some of the works include specific extension for objects that can be regarded from different perspectives at the same time (Wieringa and de Jonge 1991, Wieringa et al 1994), or objects that can change roles dynamically (Richardson and Schwarz 1991, Pernici 1989).

It is desirable to have a minimal kernel model that offers maximal expressiveness. The prototype based languages is the exemplary model that offers maximal expressiveness with self-contained, concrete objects. The prototype based languages can directly express changes of structure and behavior of objects.

To build a balanced design environment, the simplicity and flexibility of prototype-based systems and the high abstraction and rigidity of class-based systems are the key properties to have a good software development environment.

However in addition to these above mentioned software qualities, it is important to consider the basic property of object-oriented programming method, called the encapsulation and information hiding. Comparatively, the class based systems has good encapsulation property when compared to prototype-based systems which lacks the property of encapsulation (Scharli et

al 2004). The proposal of Object Oriented Encapsulation (OOE) (Scharli et al 2004) gives a new model for defining encapsulation policies in dynamically typed languages.

Hence in a balanced design environment combining prototype-based systems and class-based systems, it is important to provide encapsulation. We are motivated by the recent advancement in ownership type encapsulation (Clarke et al 1998, Clarke and Drossopoulou 2002, Boyapati et al 2003) which provides encapsulation at the level of object reference. However, this ownership mechanism too has a serious limitation of being static; hence every object owners will be decided at compile time itself and cannot be changed at runtime. Thus ownership types can be adopted for encapsulation only if there is a method for providing ownership transfer, through which we can change the owners of an object dynamically.

In the following sections, we have grouped the discussion of the related work in the research world under four headings: the class-based and prototype-based languages, the ownership model of encapsulation, the ownership transfer and the dynamic object based design techniques.

2.2 CLASS VS PROTOTYPES

There is a rich body of literature on type system for class-based languages given in (Cardelli and Wegner 1985, Danforth and Tomlinson 1988, Ghelli and Orsini 1991, Palsberg and Schwartzbach 1992, Palsberg and Schwartzbach 1994, Pierce and Turner 1994, Nierstrasz 1995, Bruce 1996, Bruce 1995a, Bruce 1995b, Abadi and Cardelli 1996b).

Class-based systems differ from object-based systems by allowing groups of objects with uniform structure to be created via instantiation and the

structure and behavior of instantiated objects to be incrementally specified via inheritance.

In a class-based language, it is the classes that explicitly specify how objects are to be created. An object can access its own instance properties and the class properties of its class(Abadi and Cardelli 1996a).

Inheritance is a relation between classes. Given a method invocation of the form o.m(…), a language-dependent process called method lookup is responsible for identifying the appropriate method m of the object o that has to be executed. Class-based systems generally follow the standard storage model for method lookup. In this model, methods are packed into method suites and they are shared by objects of the same class. Method lookup must access these method suites. In the presence of inheritance, method suites are organized as a tree, and method lookup will follow the chain of method suites. Within the methods, the identifier *this* (in languages like Java and C++) refers to the host objects that originally received the invocation of the method m(…).

Subclass describes the structure of a set of objects in an incremental manner, by offering extensions and changes to its direct superclass. Inheritance is the sharing of attributes between a class and its superclass. Without subclasses, an occurrence of *this* in a class declaration refers to an object of that class. With subclass, *this* refers to an object of the subclass, not to the superclass, and hence dynamically bound at run-time i.e. the code to be executed is determined dynamically, depending on the object which received the message. Thus from subclass to access definitions of superclass a special identifier *super* is used. The *this* and *super* are pointer references that are internally maintained by the language definition.

Prototype-based languages offer maximum expressiveness (Lieberman 1986, Ungar and Smith 1987, Taivalsaari 1996, Chambers 1993). Prototype-based languages focus on working with self-contained concrete objects instead of abstract classes. They give up the notion of class and hence are more dynamic. In the Treaty of Orlando (Stein et al 1988), the differences between prototype-based and class-based languages are analyzed. Class-based language is known for its type soundness, while the prototype-based language is well known for its object level specifications (Sciore 1989, Borning 1986).

However, prototype-based systems have been criticised for their lack of static type system. In prototype-based languages changing parent object can be done dynamically at the level of objects (called delegation) rather than static class based inheritance. Generally the prototype-based languages concentrate at object level and not at the module level (Snyder 1986), where they severely lack in the encapsulation policy defined by the class-based languages.

Cecil (Chambers 1993), and Omega (Blaschek 1994) restricted delegation to be static and hence delegation parents to be known statically. They also eliminated any form of individual behavior change and hence there is not much difference to a class-based environment.

Types are invariants that put a constrain to the range of values that can be stored in variables, passed as parameters or returned as method results in any state of an object (Palsberg and Schwartzbach 1994).

A type system defines a set of rules that allows us to infer types for every expression within a program. The static type-checking guarantees the type correctness property for an expression at compile-time and helps to ensure that errors will not occur at run-time.

The power of object-oriented type systems is their notion of subtyping. An expression of a subtype may safely be used in any place where an expression of a supertype is expected. Subtyping in conjunction with dynamic binding lets the same message have different effects at different stages of execution. Abadi and Cardelli proposed the first type systems for prototype-based language in (Abadi and Cardelli 1996a).

Some of the type systems for prototype based programming environment include (Fisher and Mitchell 1994, Katiyar et al 1994, Fisher and Mitchell 1995, Abadi and Cardelli 1996a, Riecke and Stone 2002). In (Fisher and Mitchell 1994), Fisher and Mitchell assumed that method addition or update and subtyping are mutually exclusive, i.e., their object types allow either extension and update without subtyping or subtyping without extension and update.

Another more general approach is proposed by Riecke and Stone (Riecke and Stone 2002). Their system combines unrestricted width subtyping and unrestricted method addition; it is considered the first object calculus with object extension and full width subtyping.

Deciding object behavior based on classes fixes subtype relationship. Thus objects cannot evolve by changing its hierarchy dynamically. Enforcing static relationship does not describe the dynamic sharing between objects (Stein et al 1988, Snyder 1986).

The inheritance hierarchy helps in deciding subtyping relationship, method or behavior inheritance from parents, and the lookup path which helps in forwarding unknown messages to its parent.

Delegation is an act of passing unknown messages to the delegatee parent, which has a greater capability of handling that message on behalf of

the delegator child (Lieberman 1986, Stein et al 1988). Delegation is the special property of the prototype-based languages which replaces the static class-based inheritance.

F. J. Hauck in (Hauck 1993a, Hauck 1993b) dealt with typed inheritance based on typed interfaces. His aim is to change the base class of a class that is fixed during inheritance. The pointer binding in defining the inheritance/aggregation relation is made explicit and it is a kind of stored pointer model. The properties of subtyping are discussed in detail by (Stein 1987), where object types and class types are differentiated.

According to (Bardou and Dony 1996, Chambers et al 1991, Dony et al 1992), an object and its delegation parents form one conceptual entity: a split object.

According to (Kniesel 1999, Kniesel 2000), every class is provided with a mandatory delegatee field that refers to the parent (static parent) and for dynamic delegation the parent field is changed with new delegatee object provided the type of the new parent is subtype of the mandatory delegatee's type.

2.3 OWNERSHIP TYPES

In object-oriented programming languages, aliasing is considered as double-edged knife with its advantage of the creation of advanced data structure, and disadvantage of object's reference leakage (Minsky 1996), which permits unauthorized access to the data structure nodes.

John Hogg et al, recognized object aliasing as a major problem in their work presented in (Hogg et al 1992). The Islands (Hogg 1991) and Balloon (Almeida 1997) present research on full alias encapsulation, which is

considered as less flexible for working with advanced data structure. Safe alias mechanism is given by Olivier Zendra and Dominique Colnet (Zendra and Colnet 1999) based on Eiffel language.

The proposal by Noble et. al. (Noble et al 1998) forms the basis for ownership model. In the ownership model (Clarke et al 1998, Clarke and Drossopoulou 2002, Boyapati C. 2003), the owner gives a logical boundary thereby specifying how communication should take place between objects inside the owners' encapsulation boundary and objects outside the owners' boundary.

In object-oriented programs, an object can potentially reference any other object in the object store and read and modify its fields through direct field accesses or through method calls. Such programs with arbitrary object structures are difficult to understand, to maintain, and to reason about.

The ability of an object to access another object's fields can be achieved more easily by enforcing that only certain objects can modify the object store directly whereas the rest of the objects have no direct reference to the object store at all.

Ownership has been applied successfully to structure the object store and to restrict reference passing and the operations that can be performed on references. In particular, ownership allows one to confine an object inside a data structure and to prevent representation exposure through leaking (Noble et al 1998).

The restrictions on references simplify reasoning about programs: they enable modular verification (Leino and Muller 2004, Muller and Poetzsch-Heffter 1999), facilitate thread synchronization (Boyapati et al

2002), and allow programmers to exchange internal representations of data structures (Banerjee and Naumann 2005).

Ownership models usually enforce the owner-as-dominator property (Clarke et al 2002). This restriction allows an owner object to control how the objects it (transitively) owns are accessed.

The verification of functional correctness properties such as object invariants, a weaker ownership model suffices: an object X can be referenced by any other object, but reference chains that do not pass through X's owner must not be used to modify X (Lieno and Muller 2004). This model distinguishes among read, write and read-only references, and enforces the owner-as-dominator property only on read-write references. Owners can control modifications of owned objects. This property is called as owner-as-modifier (Dietl and Muller 2005).

Ownership properties can be checked statically by type systems. Most existing work focuses on parametric ownership type systems that enforce the owner-as-dominator property (Clarke and Drossopoulou 2002). The ownership type systems by Boyapati (Boyapati and Rinard 2004, Boyapati et al 2003) weaken the owner-as-dominator property by allowing instances of inner classes to access the representation of the instance of the outer class they are associated with. Thus, they can handle iterators, but not more general forms of sharing. While parametric ownership type systems describe ownership properties accurately and guarantee a strong type invariant, ownership parametricity increases the complexity of the type system and the annotation overhead (Dietel and Muller 2005).

In Universes (Mueller and Poetzsch-Heffter 1999), authors relax the restricted nature of the existing ownership types by permitting passing of references (on condition) outside the boundary and also defining invariants on

objects. The Universe type system organizes objects into ownership contexts (Dietl and Müller 2005, Mueller and Poetzsch-Heffter 1999). Each object has 0 or 1 *owner* objects. The owner of an object (or the absence of an owner) is determined by the new expression that creates the object. Once determined, the owner of an object cannot be changed.

The Universe type system enforces the "owner-as-modifier" property. Thus in this situation, if one looks at all the references from outside an ownership context into objects within the context, all of these references must be read-only references, with the exception of any references from the context's owner.

In proposals by (Potanin et al 2004, Potanin et al 2006), approaches have been made towards making ownership more practical for the purpose of programming languages with parametric polymorphic type system. Effect based encapsulation mechanism is proposed by Yi Lu and John Potter (Lu and Potter 2006).

The other alternative encapsulation technique for ownership type is by providing restriction to access certain objects based on their type annotations (Zhao et al 2003, Vitek and Bokowski 1999, Aldrich et al 2002).

Confinement properties impose a structure on object graphs which can be used to enforce encapsulation properties essential to certain program optimizations, modular reasoning, and software assurance.

In (Vitek and Bokowski 1999) Bokowski and Vitek proposed a lightweight notion of encapsulation for Java called confined types. The idea is to use Java’s notion of software module (packages) as an encapsulation boundary. A class is termed confined if references to instances of the class may not leak out of the class defining package. In other words, a confined

object can only be stored in fields of objects defined in the same package and manipulated by code of classes belonging to its package. This approach requires very few annotations (one annotation per confined class, and some extra annotation for inherited method) and that conformance to the confinement rules can be checked in a module-wise manner. Confinement, as defined in (Clarke 2001, Gordon 2007, Gordon and Noble 2007), enforces the informal soundness property that an object of confined type is encapsulated in its defining scope. Two drawbacks of the work of confinement types are: (1) classes can only be confined within a single package and (2) standard collection classes (such as vectors, lists) can not be used to hold confined objects.

Following this in (Zhao et al 2003) the authors tried to resolve the issues by finding solution to the above mentioned two problems. The idea is that modules are composed of two distinct software layers: an interface composed of public classes and a core consisting of confined classes. Confinement adds to the visibility rules provided by the language by guaranteeing that subtyping can not be used to 'leak' reference to core classes. Furthermore confinement annotations make the programmer's intent explicit and allow for automated checking.

2.4 OWNERSHIP TRANSFER

Ownership transfer is the property of changing the ownership of an object at run time. Generally, the existing ownership system fixes the owner of an object statically and hence the owner cannot be changed dynamically (Cameron et al 2007, Muller and Rudich 2007, Bornat et al 2005). The support for ownership transfer based on uniqueness (Boyland and Retert 2005) increases the complexity of the program understanding.

The other method of ownership transfer is the External Uniqueness (Clarke and Wrigstsad 2003), used to remove the problems faced by unique reference called the abstraction and the orthogonality problems. The idea is that the externally unique reference is the only active reference into the aggregate object from outside and hence it is unique.

The problem with external uniqueness is that both, the movement and the borrowing (temporary transfer, existing for the scope of the function) cause the entire aggregate to transfer (for example, transferring an entire list) and hence it is not per-object based ownership transfer (transferring particular node within the list). The other approach is based on object invariants (Lieno et al 2004, Lieno et al 2005) where the ownership transfer is appealed between owners only after both the owners are unpacked. Work in (Bornat et al 2005) deals with ownership transfer using separation logic, where the permission is transferred between concurrent threads.

The work in (Pradeep 2006) aims at consolidating three worlds namely, class-based, prototype-based, and role-based language model using ownership encapsulation and modal logic. In this work ownership transfer is given as dynamic role modification (also known as mode-switching (Abadi and Cardelli 1996a)). The work in (Tamai et al 2005) gives dynamic adaptive environment which forms major inspiration for the present work.

All the above ownership-based system do not support proper ownership transfer or provide support based on uniqueness or provided only with migration.

2.5 DYNAMIC OBJECT-BASED DESIGN

Dynamic object re-classification is a feature which allows an object to change its class while retaining its identity. Thus, the object's behavior can

change in fundamental ways (*e.g.*, non-empty lists becoming empty, iconified windows being expanded, *etc.*) through re-classification, rather than replacing objects of the old class by objects of the new class. Lack of re-classification primitives has long been recognized as a practical limitation of object-oriented programming.

A distinguished feature of *Fickle*, with respect to other proposals for dynamic object reclassification (Ancona et al 2001, Drossopoulou et al 2001), is that it is type-safe, in the sense that any type correct program is guaranteed never to access non-existing fields or methods. In *Fickle* class definitions may be preceded by the keyword state or root with the following meaning: *state classes* are meant to describe the properties of an object while it satisfies some conditions; when it does not satisfy these conditions any more, it must be explicitly re-classified to another state class.

The other form of dynamic object modification (i.e. dynamic inheritance) is the delegation. Delegation is an act of passing unknown messages to the delegatee parent, which is more capable of handling that message on behalf of the delegator child (Lieberman 1986, Stein et al 1988), where the automatic forwarding of messages to the parent will internally binding *this* reference of the receiver object.

According to (Fisher and Mitchell 1994, Abadi and Cardelli 1996a), delegation cannot be safely combined with static typing and subtyping. The proposal from Riecke and Stone avoided this restriction in their proposal (Riecke and Stone 2002); it is considered the first object calculus with object extension and full width subtyping. According to (Bardou and Dony 1996, Chambers et al 1991, Dony et al 1992), an object and its delegation parents form one conceptual entity, a split object.

According to (Kniesel 1998, Kniesel 1999, Kniesel 2000), every class is provided a mandatory delegatee field that refers to the parent (static parent). In case of dynamic delegation, if the type of the new parent is subtype of the mandatory delegatee's type, the parent field is changed with new delegatee object.

From this concept of unanticipated delegation in object-oriented programming language, we are motivated to with our idea of unanticipated ownership transfer by letting dynamic binding between ownership domains and hence we have more dynamic object migration environment.

CHAPTER 3

OWNERSHIP TRANSFER IN CLASS-BASED OWNERSHIP LANGUAGES

3.1 INTRODUCTION

In this chapter, we present our model for ownership transfer, which helps in adding the facility of ownership transfer in typical class-based programming languages such as Java, C++, etc. First, we discuss our base model which is the representation of the external owner through which object migration is carried out. This is called the dominant ownership. Next we formulate a schematic model to represent classes, objects, and owners, which will help us analyze the reference relationship between classes and objects in a class-based ownership language environment with their encapsulation and relationship properties. In this model, we exploit the classification of encapsulation as is used in class-based ownership environment. We shall also discuss the problems associated with having ownership transfer (a.k.a. object migration) in class-based systems.

First we discuss the modified ownership type, called the dominant ownership domain in the next section 3.2, followed by a discussion on our proposed model in Section 3.3.

3.2 THE DOMINANT OWNERSHIP DOMAIN

Aggregate objects as used in class based languages are containment constructs that group other objects organized in some manner, e.g., sets, bags,

lists, tuples, arrays, etc. Aggregates typically support operations to access individual members, and to iterate over all members, as in queries. Ownership types forms an aggregate of objects with owner-as-dominator property. Aggregates maybe "homogeneous", containing only objects from the same class or from classes inheriting from the same class, or they may be "heterogeneous", containing objects from many classes (Kent and Maung 1995)

An aggregate object's representation encapsulation is violated when the mutable objects making that aggregate object's representation are accessed directly by other objects in the system. As an example, accessing mutable node objects that are part of a linked list by objects other than the linked list of which those nodes are a part would be a violation of the list's representation encapsulation.

In our approach, we use the ownership type as proposed by the authors in (Clarke et al 1998, Clarke and Drossopoulou 2002, Boyapati 2003) albeit in a customized form where the owner is capable of holding dynamic collections of aggregate objects and is also capable of establishing unanticipated relationship with other such owners. These owners are called the *dominant-owner*s. Dominant ownership domain will encapsulate the entire aggregate object. Figure 3.1 below shows the dominant ownership domain and their encapsulated aggregate objects. Here Elvis_Center is called the dominant owner. We propose a mechanism of establishing both anticipated and unanticipated relationship between dominant owners called neighbourhood. The neighbourhood is unidirectional, *i.e.*, for example, $\Delta \kappa \Omega$ implies that Δ has Ω as a neighbor but Ω doesn't have neighbor Δ. The Greek letter Kappa (κ) is used to represent the neighborhood between owners. In Figure 3.1, the dominant owner Ounda_Center is the neighbour of Elvis_Center but the reverse is not true i.e. Elvis_Center is not the neighbour

of Ounda_Center. Hence the neighborhood relationship as represented in the figure is Elvis_Center κ Ounda_Center.

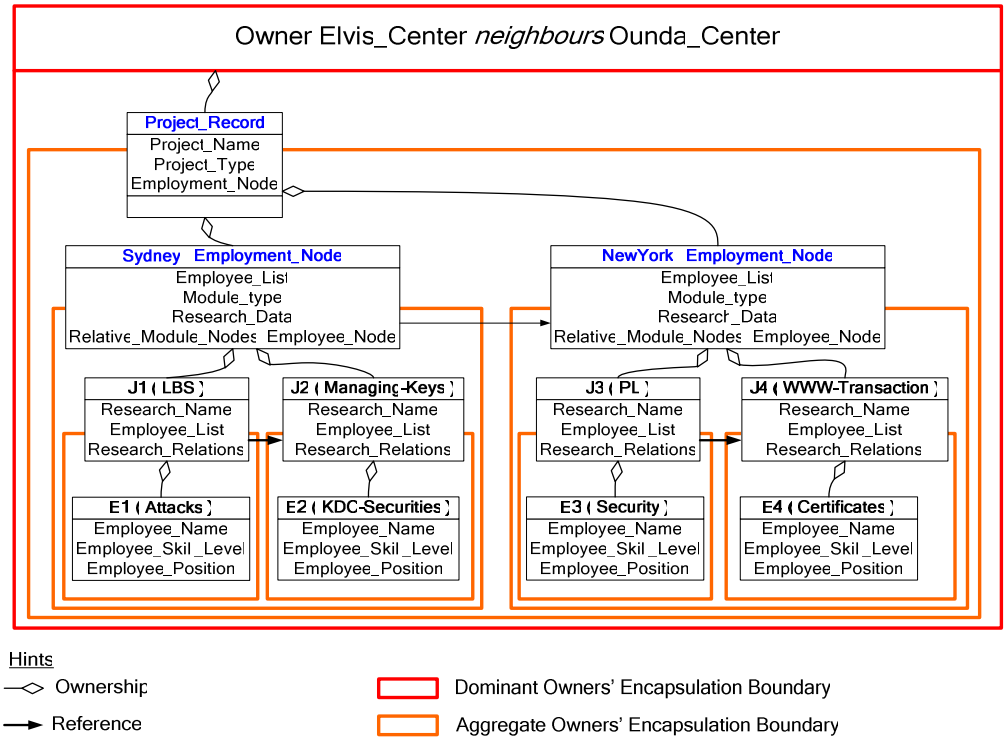

Figure 3.1 Dominant ownership domain.

Neighborhood helps to restrict ownership transfer only to a neighbour dominant ownership domain. In our design, neighbourhood is not fixed throughout the lifetime of the dominant owner, *i.e.*, it can dynamically bind with other dominant owners. It can also unbind from existing neighbours, thereby modifying the neighbourhood scenario at runtime. Thus the property of dynamic modifications to neighborhood helps in unanticipated ownership transfer.

As already mentioned, the objects within the dominant owners are called the aggregate objects. Aggregate objects will have their own ownership boundary. As an example, in Figure 3.1, the Project_Record is an aggregate

object encapsulating the other objects within it. The object Sydney:Employment_Node (object Sydney of the type Employment_Node) and the object NewYork:Employment_Node are encapsulated by Project_Record, *i.e.*, Project_Record is the owner of these two objects. These objects in turn have their contexts encapsulated.

By default, the objects at the same level, e.g, Sydney:Employment_Node and NewYork:Employment_Node are assumed internally as neighbours. However, internal neighbourhoods like this are fixed statically and hence can not be modified at runtime dynamically. Thus the objects Sydney:Employment_Node and NewYork:Employment_Node are having fixed neighbourhood, and similarly the objects J1(LBS) and J2(Managing_Keys) are at the same level and hence are neighbours internally.

Ownership encapsulation is an important property that defines restrictions at the reference level. In the next section the schematic representation is discussed followed by a detailed discussion on encapsulation properties as is applicable in our model.

3.2.1 Object Migration

Object migration is the property which permits an object to dynamically transfer its ownership from one ownership domain to another neighborhood ownership domain. In Object migration the migrated object will change its ownership domain. After migration, the migrated object will have multiple owners, *i.e.*, the current owner to which the object gets migrated and the original owner from where the object is instantiated. The objects can migrate only between neighborhood domains. Thus the objects are free to migrate between dominant owners or between aggregate owners present at the same level. However, object migration causes side effects when added in a class-based scenario. In the next subsection we shall formulate a schematic

model to capture the key properties to be noted, which will help to analyze the side-effects in object migration.

In Figure 3.2, the owner dominant Elvis_Center has a neighbor Ounda_Center. The objects inside the Elvis_Center are allowed to migrate from Elvis_Center based on the neighbourhood of the object. The object J1(LBS) is permitted to migrate from the aggregate owner Sydney:Employment_Node to NewYork:Employment_Node which is at the same level of Sydney:Employment_Node, and can also migrate from Project_Record to other aggregate owner present at the same level of Project_Record. Also the object J1(LBS) can migrate from dominant owner Elvis_Center to its neighbor Ounda_Center, however it is not permitted to migrate to Runa_Center which is neighbor of Ounda_Center.

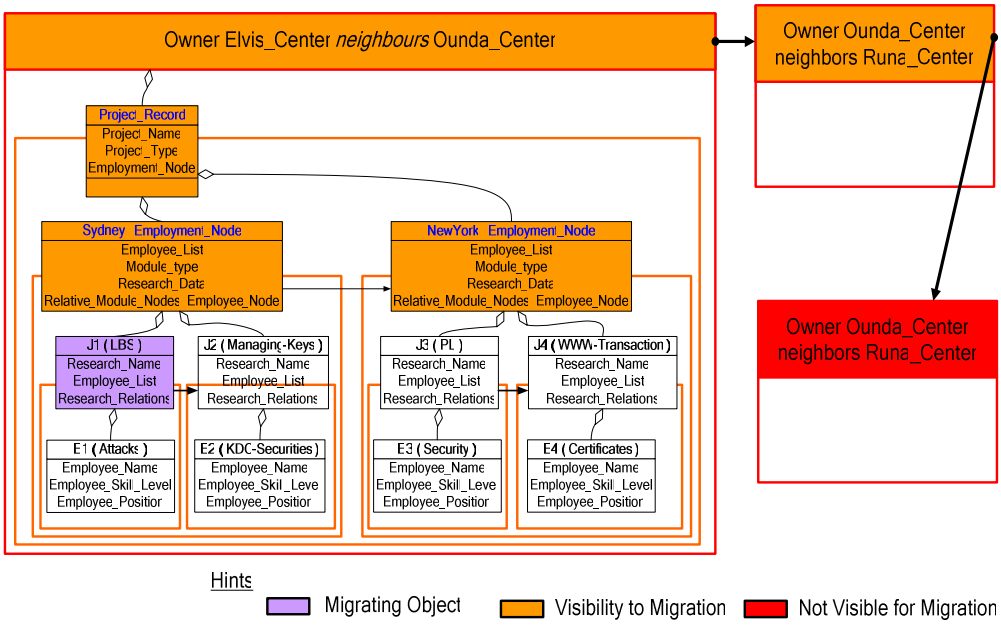

Figure 3.2 Object Migration

In the next section we shall analyze the standard storage model of a class-based programming language in representing the relationship between

classes and objects in the heap. And in the later part of this subsection we shall formulate a schematic representation of classes, objects and ownership representation using our model called the reference model.

3.3 CLASSES, OBJECTS AND DOMINANT OWNERSHIP ENVIRONMENT

Adding object migration to a class-based object oriented languages is not free of hazards. However, before studying the problems associated with ownership transfer, it is important to study the core part of dominant ownership, *i.e.*, relationship. Let us first model the run time heap memory lay-out for object to class relationship within the dominant ownership environment, which can help us understand the encapsulation breaches that may occur when we add object migration to a class-based object oriented languages.

The model we propose would capture the general representation of classes and objects in a class-based programming language like C++ and Java. Creating a simple model to represent the relationship between classes, objects, owners and neighbors would help us to analyze the side-effects. The proposed model gives the runtime layout and hence clearly shows the reference relationship between objects and classes in an ownership environment.

The simplicity of the proposed model is due to the fact that in general, to capture the side-effects happening at the reference level of a class-based systems we need only few properties to get highlighted instead of every internal details of a class or object. The details we are more concerned about are the reference relation between classes and objects in a class-based ownership system, the encapsulation boundary, and the presence of static variables. Thus this model represents the reference relationship between

objects, classes at language implementation level in a class-based ownership environment and helps us to analyze the side-effects directly at the language implementation level.

The next subsection explains the standard storage model of generic class-based programming languages, followed by our reference model that captures the representation of classes, objects, owners and neighbors in such an environment in the next subsection. This subsection is followed by the classification of encapsulation property in a class-based ownership environment in subsection 3.3.3.

3.3.1 The Standard Storage Model

In class-based languages, methods are not directly embedded into objects; instead they are factored into method suites that are shared by the objects of the same class. Method lookups access these method suites associated with the corresponding classes. In the presence of inheritance, method suites are organized as hierarchical tree, and method lookup may require examining a chain of method suites. The storage model of object is important to understand the semantics of programming language, the reason is driven as follows from (Abadi and Cardelli 1996a):

> *(…) it is usually designed to produce the illusion that methods are, after all, embedded directly into object, as in naïve storage model. When this illusion fails, confusion may result in both language semantics and programming.*

Another important field to be noted is ***this*** and ***super*** identifier; *this* identifier is used by the method to refer to the object that originally received the invocation of that method, and the *super* identifier is used by the subclass methods to invoke the old version of the method from a superclass. In the

presence of inheritance, most practical programming languages follow hierarchical method suites. In hierarchical method suites, the method lookups search these suites from subclasses to superclasses until an appropriate method is found.

Thus the hierarchical method suite helps us to organize the lookup hierarchy into well defined structure. In our design we are particularly interested in the reference relationship between two entities like classes to objects, or subclass to superclass etc., and thereby analyzing the side-effects in object migration in the presence of inheritance. In the next section, we will discuss the reference model.

3.3.2 The Reference Model

Figure 3.3 shows our reference model. We use sphere to represent object, rectangle to represent classes and oval to represent owners, and in addition we use two small circles, hollow circle and filled circle, to represent the properties of the corresponding entities. The hollow circle indicates the property of an individual entity, i.e., the presence of the variables like, instance variable in the case of objects and static or class variables in the case of classes. We represent the relationship between two entities using presence of filled circle. As an example the type of an object can be represented by showing a reference pointing from an object to its corresponding class, and the subclass to superclass relationship can be represented by showing a reference pointing from a subclass to its superclass. Similarly the dominant ownership relations can be represented using a reference pointing from classes/objects to the corresponding dominant owner. The reference present in the class will say the current owner, and the presence of reference in the object will represent the original owner from where the object is created. And the relationship between dominant owners (neighborhood) can be represented

by the presence of a reference from one dominant owner pointing to another dominant ownership domain.

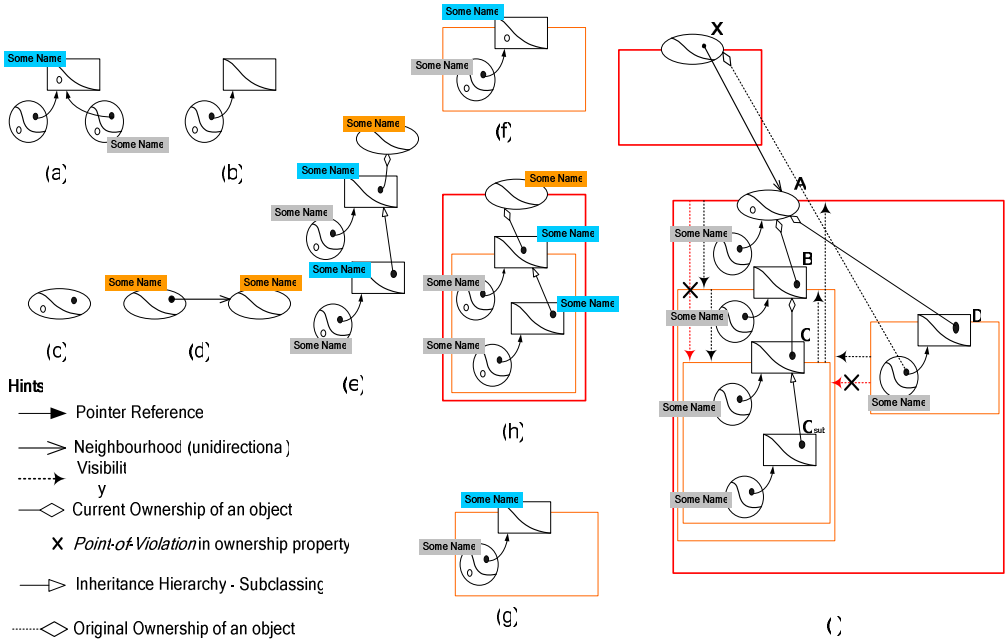

Figure 3.3 Reference Model - capturing Classes, Objects, Owners and Neighbors representation in class-based ownership type systems

In addition, there is another relation called aggregate owner relationship, which represents the ownership of aggregate objects present within the dominant owner. As an example in Figure 3.3(i) X and A represent the dominant owners, and B (encapsulates C) and D (encapsulates its own context) are the aggregate object. The aggregate objects B and D are considered to present at the same level within the dominant owner A since both aggregate has the same dominant owner A as their current owner. C is the aggregate owner encapsulating its own context.

Figure 3.3(a) shows two objects to a class (sibling objects) and the presence of class variable indicate that it is shared by the objects belonging to

this class. The hollow circle in the class represents the presence of class instance or static variables, and hollow circle in the object instance represents object instance variables or non-static variables. Filled circle in the object instances shows the type of object, i.e. the class to which the object belongs. The Figure 3.3(b) shows empty classes with an empty object. These are represented with the usual notation of sphere and rectangle without a name associated with these entities. These are objects and classes that are deleted/migrated without revoking the memory through garbage collection. It is possible for a system to contain empty objects/classes/dominant owners. This class does as shown in this example (Figure 3.3(b)) shows a class without static variables present in it (indicated by the absence of hollow circle). Figure 3.3(c) shows empty dominant owner. Figure 3.3(a) also shows one deleted object. In Figure 3.3, "*Some Name"* indicates the name (identifier) given to an entities (like classes/objects/dominant owner) which are unique inside the environment.

Figure 3.3(c) shows the dominant owner, while Figure 3.3(d) shows the representation for neighbourhood of the dominant owner. The Figure 3.3(e) shows the presence of objects and classes within the dominant owners' environment and also the presence of subclasses. The presence of filled circle in the dominant owner indicates the presence of the neighbourhood of the owner.

Object migration is the property which permits an object to dynamically transfer its ownership from one ownership domain to another neighborhood ownership domain. The presence of hollow circle in the dominant owner represents the presence of migrated object inside the dominant owner. Figure 3.3(i) shows the migrated object D within the domain A with the presence of hollow circle inside the dominant owner A. After migration the class will have a reference to its current owner (in this case

domain A) and the object will point to the original owner from where it is migrated (in this case X).

In Figure 3.3(h) we show the presence of inheritance relationship in the dominant owner. In Figure 3.3(i) the dominant owner A encapsulates the object B and B in turn encapsulates object C. There exists a subclass of C, the C_{sub}. D is an object migrated to dominant owner A. Thus in Figure 3.3(i) we are having a hollow circle in the dominant owner which says the presence of migrated object within this dominant ownership domain A.

3.3.3 Classes, Object and Owner Encapsulation

Our next step is to define the encapsulation boundary in a class-based programming language model in the presence of ownership. We classify the encapsulation under the following three heads:

1. Class Encapsulation
2. Object Encapsulation
3. Owner Encapsulation

These three kinds of encapsulations would lead to the distinction of the encapsulation boundary clearly and will also help in identifying the side-effects that may affect the encapsulation boundary during delegation.

In Figure 3.2, the encapsulation boundaries for the three kinds are shown. Figure 3.2(f) shows the class encapsulation boundary, and Figure 3.2(g) shows the object encapsulation. The difference between the above two kinds of encapsulation is the presence of instance variables, *i.e.*, the presence of class-variables impose class encapsulation and its absence impose object encapsulation. In class encapsulation the class is shared among the objects belonging to the corresponding classes. And in object

encapsulation, each object has its own separate class properties and hence has separate encapsulation boundary. Figure 3.2(h) shows the dominant owner encapsulation boundary. The dominant owner acts as a package or container encapsulating both classes and the objects within its boundary. Figure 3.2(i) shows the aggregate owners' encapsulation boundary and the visibility rules in the presence of ownership encapsulation, where the aggregate owner is the class or object encapsulating the objects it contains within its encapsulation boundary. In Figure 3.2(i), we have shown a class D as a migrated object to the dominant owner A, which implies that the class D and its object will follow all the rules that are applicable to the aggregate present within the dominant owner A.

Until now we have defined a reference model that captures the reference relationship and encapsulation property of objects, classes in a class-based ownership environment. In the rest of this chapter, we would develop a scenario that will help us to understand the problem clearly. We use our reference model to depict the situation and later we shall analyze the side-effects and how object migration will happen in a class-based ownership environment.

3.4 SCENARIO

Let us assume that we are having a research organization called *Elvis_Center*. The Elvis_Center forms an encapsulation boundary to the inside objects as per the ownership property. Let us take the scenario given in Figure 3.4. In this situation the employees can migrate from one company to other company.

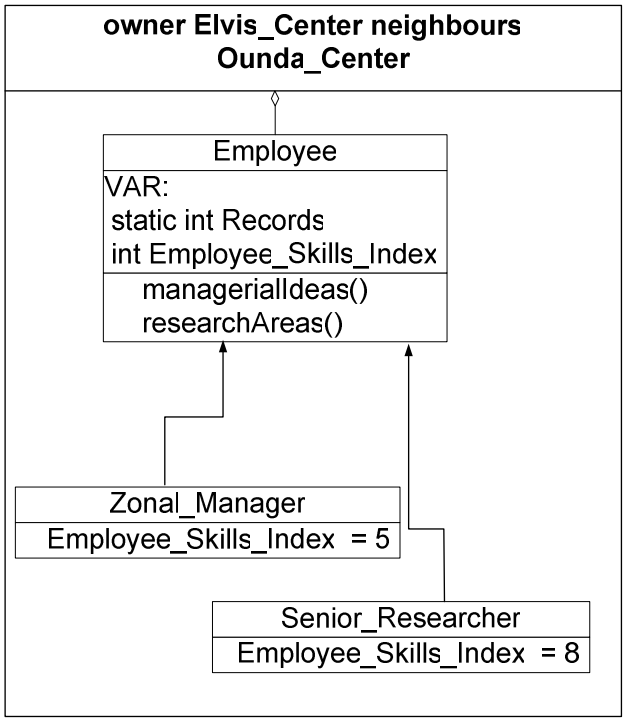

Figure 3.4 Scenario showing Object Migration

In Figure 3.4, we have a class Employee within the owner Elvis_Center. It maintains details about the employees. This implies the objects created from the class Employee will have the Elvis_Center as their default owner. The class Employee has two variables, the static class variable and the object-instance variable. The object-instances Zonal-Manager and the Senior-Manager have their own copy of the variable, integer, and share the static variable, Records. As the design is based on the class-based language, the objects will have a pointer to the class from which it is created.

In addition to the encapsulation boundary the owner Elvis_Center has declared its static relation to other owner Ounda_Center, as shown in the Figure 3.4.

Figure 3.5 is the reference model for the scenario given in the Figure 3.4. In Figure 3.5 we have only captured the necessary property to be

highlighted, like ownership encapsulation, dominant owners, neighborhood, presence of static variables and the object migration. Hence, this model will help us to analyze the side-effects due to object migration in a class-based ownership environment clearly.

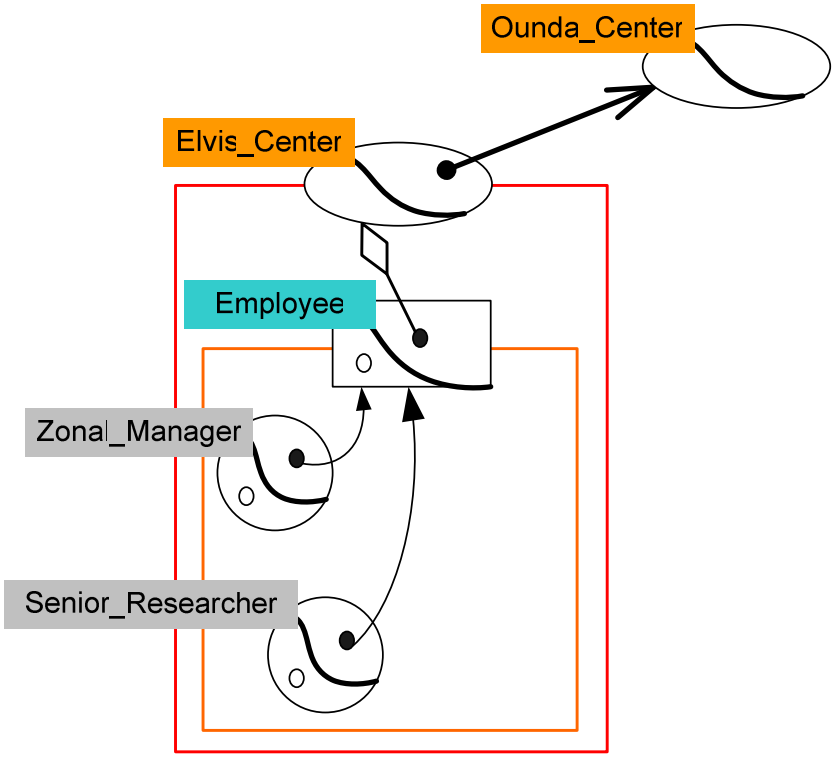

Figure 3.5 Reference Model of the Scenario

In Figure 3.5, the hollow circle is absent in Ounda_Center representing the absent of migrated object inside the Ounda_Center.

3.5 CONCLUSION

In this chapter we have analyzed the fundamental concepts associated with the ownership transfer in a class-based ownership environment. We have also developed a reference model that captures certain minimal properties like classes, objects, owners and neighborhood

representation in a class-based ownership environment with its encapsulation properties. The reference model will be helpful in understanding possible side-effects that may occur in object migration in such an environment.

In the next chapter, we shall discuss the problem associated with object migration and we shall represent them using our reference model.

CHAPTER 4

THE OWNERSHIP TRANSFER PROBLEMS AND SOLUTION

4.1 INTRODUCTION

Class-based language has features like the presence of permanent pointer from object pointing to the class it belongs to. In such programming environment, the type of an object is determined by the class to which the object belongs. At runtime this relationship is represented using a pointer between the object instance and the class to which the object belongs. Similarly the is-a relationship between subclass and superclass is also represented using a pointer from subclass to the superclass.

However, having these pointers create side-effects to dynamic object migration. Various problems may arise out of the situation. Here we consider two main problems, viz., (1) existence of multiple classes, and (2) the dangling pointer problem. These effects are discussed in the next two sections of this chapter. Section 4.4 discusses the proposed model called the Ownership Transfer Model (OTM), where we also present a solution for the problems of multiple class and dangling pointer. We have analyzes the solution is given in 4.5.

4.2 MULTIPLE-CLASS AND ENCAPSULATION BREACH

The property of ownership type is that it does not permit reference leakage outside the boundary, which may break encapsulation. However

when an object is transferred between owners, the ownership transfer necessitates pointer exposure.

Let us reconsider the scenario given in the earlier chapter. In this scenario, an instance of the class Employee can migrate from one company to other company. Figure 4.1 block A reproduces the scenario given in the earlier chapter. Here blocks B and C give the situation after an ownership transfer.

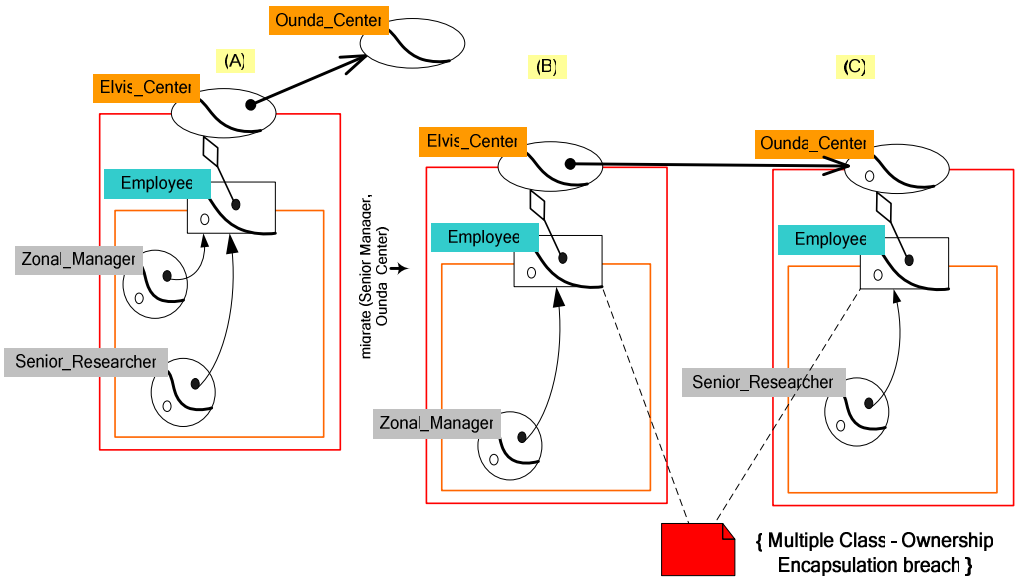

Figure 4.1 Scenario showing breaching effect due to Multi class existence

As per the class-based system multiple objects of same type will be created from a single class, which creates the requirement of maintaining a common class instance for its entire set of object instances in the whole system. In our example, the object Zonal_Manager and the Senior-Researcher are from the class Employee. Thus after the migration of Senior_Researcher from owner Elvis_Center to the neighbour owner Ounda_Center there exists two copies of the class-instance Employee one in the owner Elvis_Center and the other in the new owner Ounda_Center. However, class Employee contains

static variable called Records. Thus with multiple copies present in different owner, multiple copies of the static variable will also be present. Hence, altering the static class-instance variable present in one of the owners (Ounda_Center or Elvis_Center) will need the other copy of the class-instance present in the other owner (Elvis_Center or Ounda_Center respectively) to be modified indirectly. Thus the multiple-class is a serious side-effect of the ownership transfer where we cannot guarantee the safety properties.

4.3 EXISTENCE OF DANGLING POINTER

The existing ownership model allows the alias existence within the ownership boundary. This flexibility is to relax the model to adapt various data-structures. However, this flexibility acts as a constraint for adapting ownership model into the dynamic environment.

As per our scenario in Section 4.2, Figure 4.1A in this chapter, the owner Elvis_Center has two managers Zonal_Managers and the Senior_Researcher that are having the same class type Employee. Let us add a superclass to the class Employee called Projet_Documents to this scenario; this superclass is used to maintain the number of employees within the company. Class Projet_Documents is maintained by its object instances Coordinator X and Coordinator Y. Coordinator Y is an alias of Coordinator X. This is shown in Figure 4.2 block A. Coordinator X is now migrated to related owner Ounda_Center as shown in Figure 4.2B. However, since Coordinator Y is unaware of this migration, it continues pointing to Coordinator X, now an empty object. Thus migration leaves an empty object in owner Elvis_Center to which Coordinator Y continues to point. Thus the problem of dangling pointer is created in this situation.

The existence of alias within an owner to a deleted object location will remains as a residue alias. These kinds of residue aliases are not desirable since these will create adverse side effects on the software design.

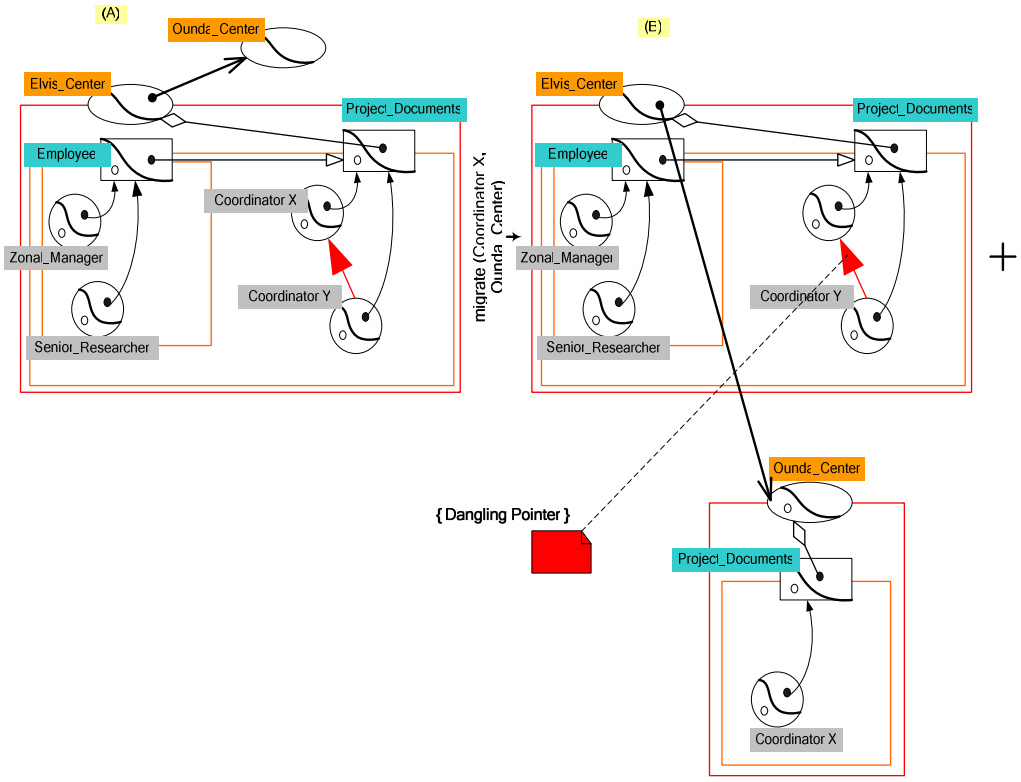

Figure 4.2 Scenario showing breaching effect due to dangling pointer existence

4.4 THE OWNERSHIP TRANSFER MODEL (OTM)

Although a desirable property in a class based scenario is to have object migration, from the discussions of the earlier sections, it is clear that simple modifications of the static properties like encapsulation (ownership encapsulation) in an aliasing environment to achieve object migration will create serious side-effects. To this end, we propose to modify the present type system annotations by adding type annotations that capture alias information which is used to control sharing and dynamic movement of an object.

This section describes our proposed model Ownership Transfer Model. It discusses how it is possible to move objects across the encapsulation boundary in a class-based language model. We have also proposed the design of a language called Jmigrate (Jm) that illustrates the model OTM. This will be discussed in the next chapter.

In OTM we classify aliases in three categories, viz.

1. unknown aliases
2. permitted references
3. known aliases

as shown in Figure 4.3.

Unknown aliases are references that do not allow the host object to know about its presence and hence may be harmful. The permitted references are references or aliases that are part of the software design criteria. The known aliases are the references which are present due to the language design model. Both the permitted references and the known alias are considered to be harmless. As an example, in a class-based programming language the reference between objects to classes are known aliases.

In Figure 4.3, the dotted line represents known aliases, the dashed line represents the unknown aliases and the solid line represents the known permitted references. Class A is the class instance, A1 and A2 represent the object instances and x represents the (nonstatic) variable present within each of the object instances. By the language design, the object instance will have a reference pointing to the class to which it belong and similarly the object will have a *this* reference to refer the variables present within itself (in this case only x). These are the known alias references since they are part of the language design itself.

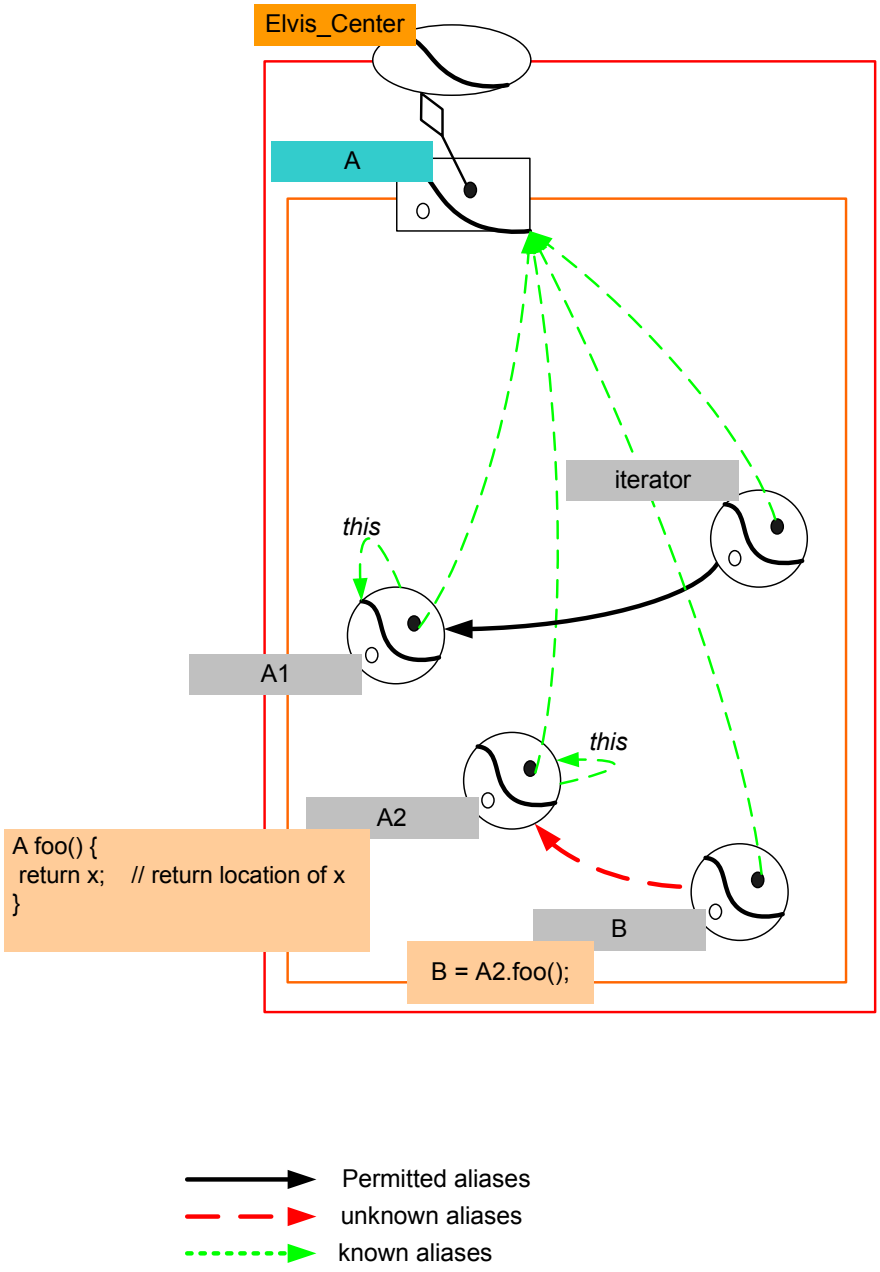

Figure 4.3 The Presence of Aliasing in a Class-Based Programming Environment

Object B belonging to some class type other than A, is acquiring the reference of x belonging to A1 by calling the method foo() which will return the reference of x. Such references are unknown aliases and they may

be harmful, since they can alter the content of variable x without passing through the interface provided by the object A1.

The iterator object belongs to class A and acquires a reference to the object A2 explicitly. This is a known alias. Such structure facilitates various data structural properties.

First, the skeleton of OTM and its relation to traditional object models is described in section 4.4.1. Section 4.4.2 discusses how problems like dangling pointers and multi-class are taken care of in the proposed model.

4.4.1 The Ownership Transfer Model: Structure

The Ownership Transfer Model has the following features:

1. It has an owner, similar to package (as in Java) or container (as in C++ / Java) that is capable of holding other objects.
2. Each owner is mapped to a memory segment that does not interfere with other owners
3. Ownership transfer is achieved using two methods: anticipated and unanticipated design.
4. The objects are created using *new* operation as per the class-based language.
5. Neighborhood defines the relationship between owners
6. The neighborhood can be established as either anticipated or unanticipated in OTM
7. Objects can migrate only to the neighborhood owners

These seven features establish the skeleton of the Ownership Transfer Model (OTM). To have a more general presentation of the

ownership transfer idea, it is useful to introduce in the system the *bind establisher* (discussed below), in addition to the owner environment.

Bind Establisher: The bind establisher is a collection of methods that is used to establish unanticipated relationship with other owners. Anticipated ownership relations can be decided at the compile time by programmers. The bind establisher is used to extend the neighborhood of an owner dynamically using the methods provided within it. Thus we can achieve unanticipated owners relationship where the objects can evolve in an unanticipated manner by acquiring properties dynamically from the neighborhood owners. From programmer's perspective, the bind establisher and the owner environment are the same.

In OTM, ownership transfer is possible only to the related owners. Let us now assume that an owner A has an anticipated relationship with other owner B. In this case, an ownership transfer can happen from A to B. Dynamically the owner A can bind to any other owner C in an unanticipated manner to have unanticipated ownership transfer.

4.4.2 OTM Approach to Dangling Pointer and Multiple-Class Problems

As the dangling pointer can create adverse side-effects, our primary challenge is to remove the possibility of the existence of dangling pointer. The dangling pointer can be removed only if we have restriction in transfer of the objects. This can be achieved by having object annotations specifying the aliasing property of the objects. Our approach in OTM differentiates between the class-type and the object-type. The class type of the corresponding objects is the class name from which they have been formed by using new operation. On the other hand, the object type specifies the object's alias property. In OTM we classify the object property as confined and non-confined. The

confined objects are objects which can be aliased only by confined objects from within the owner's boundary. No non-confined objects are permitted to alias with a confined object. On the other hand, a non-confined object can be aliased by both confined and non-confined objects. This is shown in Figure 4.4 and Figure 4.5.

In Figure 4.4, we have shown two neighborhood dominant owners Elvis_Center and Ounda_Center, where the <R> modifier represents the confined objects and the <Y> modifier represents the non-confined objects.. The class Employee has four objects namely, Scientist A, Scientist B, Scientist C and Scientist D. Among these object instances Scientist A and Scientist D are of type <R> and the objects Scientist B and Scientist C are of type <Y>. Solid line refers to aliases between two entities, the dashed line refers to unknown aliases (which are not permitted), the dash-dot line refers to read-only reference and the dotted line refers to known aliases. In Figure 4.4, Scientist A is an alias to Scientist B and Scientist B is an alias to Scientist C. It is also shown that Scientist B cannot refer to Scientist A since their object types are different, i.e. Scientis B is of object type <Y> and Scientist A has object type <R>.

The object Senior_Researcher is the migrated object, from Elvis_Center to Ounda_Center. This is shown with a dashed arrow with diamond head, which is pointing to the original owner of the migrated object Senior_Researcher, the Elvis_Center. Since the migrated object's class Employee has a static variable, any modification in one copy of the static variable must be reflected to other copies of the class too. This is shown with dotted arrow between the class Employee in owners Elvis_Center and Ounda_Center. Within the migrated current owner Ounda_Center, the object Senior_Researcher can have only read-only references to other <Y> typed objects present within the Ounda_Center.

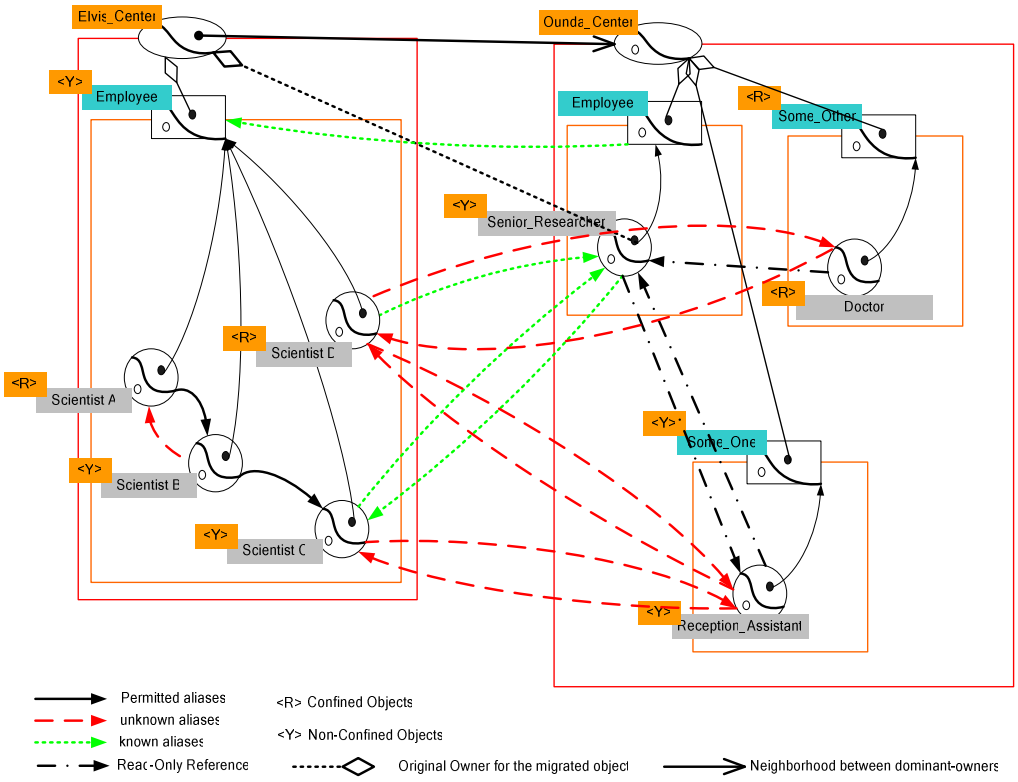

Figure 4.4 The Reference Diagram Representing the Aliasing Properties

With this design we permit only the known alias reference across the dominant owners' boundary. Thus our system is statically checkable.

In OTM, only the safe aliases are permitted, thereby providing a mechanism for static verification of any possible side-effects due to object migration. Hence, the permitted aliases and known aliases can be classified as safe aliases for whom we can predict any possible side-effects at compilation time. In OTM, we permit only the non-confined objects to migrate between owners. This is because the non-confined objects cannot expose the owners' contents. The permission to have alias reference between different object types is given in Figure 4.5.

Within Ownership Domain

From \ To	R	Y
R	Permitted	Permitted
Y	Not Permitted	Permitted

Between Ownership Domain

From \ To	R	Y
R	Not Permitted	Permitted-Alias
Y	Not Permitted	Permitted-Alias

Form Migrated Objects to Originator Ownership Domain

From \ To	R	Y
R	Can Not Migrate	Can Not Migrate
Y	Not Permitted	Permitted-Alias

Form Migrated Objects to Migrated Ownership Domain

From \ To	R	Y
R	Can Not Migrate	Can Not Migrate
Y	Not Permitted	Read-Only

Form Originator Ownership Domain to Migrated Objects

From \ To	R	Y
R	Can Not Migrate	Permitted-Alias
Y	Can Not Migrate	Permitted-Alias

Form Migrated Ownership Domain to Migrated Objects

From \ To	R	Y
R	Can Not Migrate	Read-Only
Y	Can Not Migrate	Read-Only

Figure 4.5 Aliasing Properties for Objects

In the Figure 4.5, we design a matrix diagram for specifying the permitted alias properties between object types. The column From specify the

object type that can be alias to object types specified in the To column. As shown in the diagram, within ownership domain we are permitting alias reference between two <R> typed objects, and we are not permitting alias from <Y> typed object to <R> typed object. We can build similarly for all the other possible cases as shown in Figure 4.5.

During object migration, the migrated object is permitted to refer the originator owner's contents through known aliases. Hence after object migration, the migrated object has multiple owners, i.e., the previous owner (originator) and the current owner. The confined objects within the previous owner of the migrated object can have safe aliases to migrated objects and also a migrated object can have safe aliases to its previous owners' objects. However, inside the current owner (to which the object has migrated), a migrated object can have permitted read-only references, which means that only the values can be obtained. It is not allowed to establish a reference within the owner to which the object has migrated.

4.5 ANALYZING THE SOLUTION

The aim of this research is to provide ownership transfer mechanism by conforming to the ownership type system. Our challenge is to find a system that takes care of the following problems:

1. The multiple-class effect and
2. Avoiding dangling pointer statically

To the success of the model, these two challenges should be satisfied without affecting the ownership encapsulation model in class-based ownership programming language. In the above section we have formulated a solution to these two problems by designing a type system. In this section we

shall validate the solution by analyzing the type mechanism designed in the previous section with the scenario modeled in the previous chapter.

4.5.1 Multiple-Class Existence

The problem due to the presence of the multiple-class across various owners may create ownership breaches. The problem is approached in two ways in the OTM:

1. It allows multiple existence of the class-instance, if and only if there are no static variables present in the class-instance.
2. It provides the facility of message passing between owners whenever the class-instance variables get altered.

These typing characteristics can be implemented in any language that is based on OTM. In Chapter 5 of the thesis, we exhibit a new language Jmigrate (Jm) based on OTM.

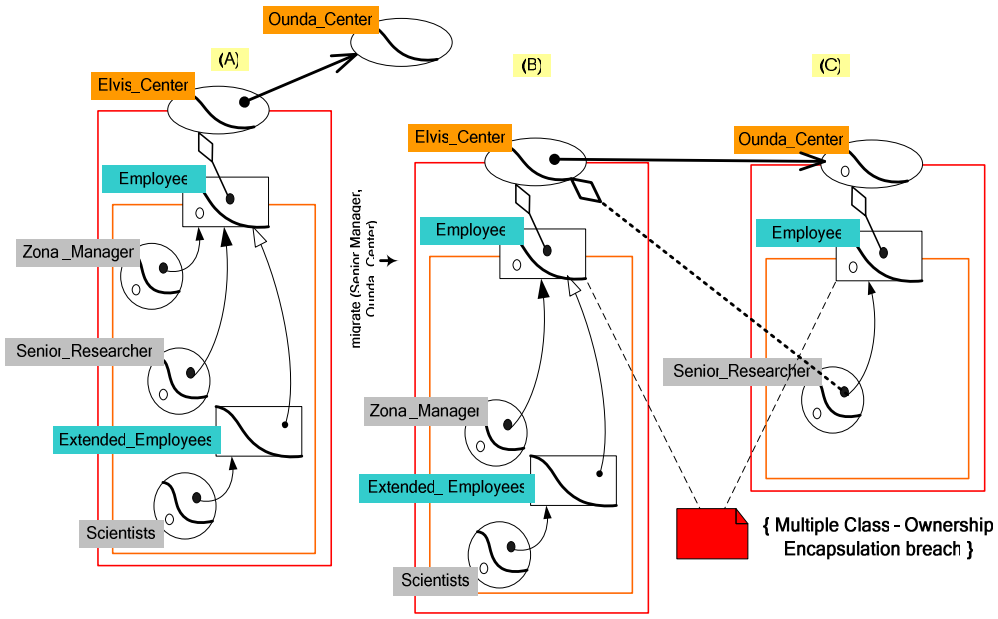

Figure 4.6 Multi class existence

Example

The following example illustrates the scenario where an object is migrated between two neighborhood owners and we analyze the references in various cases as given in Figure 4.6.

1. After migrating the object Senior_Researcher from owner Elvis_Center to the neighbor Ounda_Center, we have multiple existence of the class Employee.
2. Any change to the static variable in the Employee class, either in Ounda_Center or in Elvis_Center will affect all other objects depending on these shared class static variables.
3. Also if we have a subclass Extended_Employee to the class Employee, as shown in Figure 4.6, any changes to class Employee will also get affect the subclass Extended_Employee

4.5.2 Dangling Pointer Problem

As the dangling pointer can create adverse side-effects, our primary challenge is to remove the possibility of the existence of dangling pointer. The dangling pointer can be removed only if we have restriction in transfer of the objects. This can be achieved by having object annotations specifying the aliasing property of the objects.

In OTM, we move the alias pointer to the location of the migrated object, i.e. we will update the alias pointer address information whenever the aliased object is migrated.

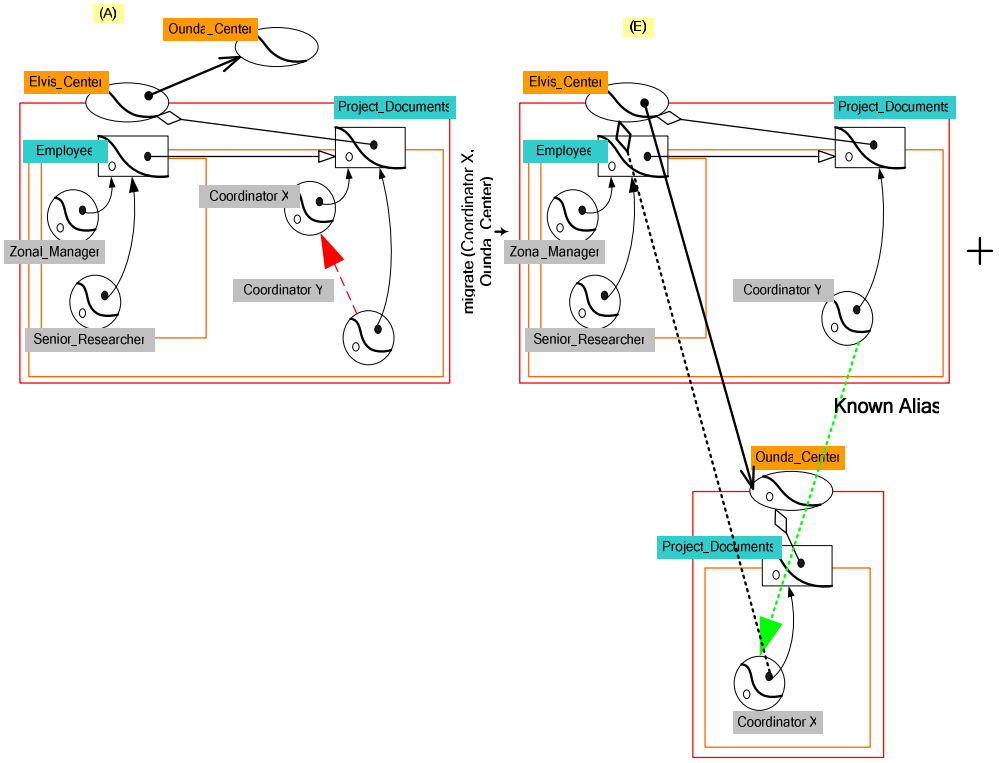

Figure 4.7 Dangling pointer existence

Example

The following example illustrates the scenario where an object is migrated between two neighborhood owners in the presence of alias inside the originator owner. The following points can be observed from the Figure 4.7.

1. The Coordinator X object is aliased by Coordinator Y object within the owner Elvis_Center
2. After Coordinator X object gets migrated to Ounda_Center, the alias from Coordinator Y will be converted into known alias, and hence permitted to refer to the migrated object Coordinator Y present within the owner Ounda_Center.
3. Thus we can avoid dangling pointer, as well as we can restrict confined object to migrate outside the owners boundary.

4.6 CONCLUSION

This chapter has presented the basic aspects of the OTM model. It has been shown how the problems like dangling pointer and the multi-class existence can be avoided statically in such a class-based environment.

The main focus of the chapter has been the discussion of safety problems that can arise from such an extension:

- The risk of “transferring ownership” of an object in a class-based programming model which rise to problem with multi-class existence
- The risk of “transferring ownership” of an object in the presence of any unknown aliases

A solution for the problem of multi-classes has been presented, which consists of the rule that “allow multi-class existence / message passing”. This seems to be obvious from the existing mechanism followed in practice.

Regarding the second problem it has been shown that upon transferring the non-confined objects the reference that are present within the originator owners boundary will follow the migrated object towards the migrated domain. And identifying confined and non-confined objects are done statically. So, every transfer that is statically safe is dynamically safe.

The unanticipated extension that can be achieved via dynamic binding between owners enables transfer of objects to the dynamic bound owners. This is a powerful technique but it entails the risk that components which have not been designed to work together might interact in undesirable ways. The next chapter details a new language called Jmigrate (Jm) which is based on OTM.

CHAPTER 5

THE JMIGRATE (Jm) LANGUAGE

5.1 INTRODUCTION

The previous Chapter has introduced the OTM model facilitating dynamic object migration. We also discussed the problems like (1) existence of multiple classes, and (2) the dangling pointer problem.

In this chapter we describe the design, implementation and applications of the language Jmigrate (Jm). Jmigrate has Java like syntaxes facilitating dynamic object migration following the model of OTM. Before a detail discussion on the language Jmigrate we shall discuss the keywords used in our language and its properties.

5.2 TYPE MODIFIERS

<R>	<Y>	class	owner
static	int	extends	neighbor
new	this	super	

In Jmigrate we classify the keywords into three broad categories, viz., object modifiers, class modifiers and owner modifiers. Jmigrate contains eleven keywords.

Object modifiers consist of keywords like *<R>, <Y>, new, this, int, static*. On the other hand, class modifiers consist of keywords like *extends, class*. The owner modifier consists of keywords like *owner* and *neighbor*.

Apart from these modifiers every entity in the system will be identified using an unique identifier name.

The <R> and <Y> annotations are type modifiers (Section 4.4.2) that specify objects alias property.

The keyword *owner* is used to represent the dominant ownership domain. The keyword *neighbor* is used to represent the anticipated relationship between dominant owners. The keyword *extends* is used to represent the subclass relationship between two classes.

The keyword *static, new, int, this* are similar to Java and C++. The *static* keyword is used to specify the static property of the class-instance. The keyword *new* is used to instantiate objects from a class. The keyword *int* is used to specify the integer variable. More importantly, the keyword *this* is used to represent the current object or the method receiver and is used to access the objects fields from inside. The keyword *super* is used to access the fields in the hierarchy that is to access fields present in the superclass from the subclass.

5.3 OBJECTS WITHIN OWNERS BOUNDARY

In Jmigrate every object will have 0 or 1 aggregate owners, and 1 dominant owner. In Jmigrate, the dominant owners are declared using the keyword *'owner'*. The aggregate owners are assumed by the presence of has-a relationship between objects.

```
…
owner Elvis_Center {};
…
```

In the above, we have created a dominant owner Elvis_Center. The dominant owner cannot have objects, i.e. we can not create objects for Elvis_Center using *new* constructor.

In Jmigrate, the keyword *extends* is used to specify two key properties i.e.

1. The subclass and the subtype properties.
2. Specifying the aggregate objects within the dominant owners

The class extending the dominant owner will make its objects to be within the dominant owner by default. Following example program shows how the inside objects are created.

```
[1].    owner  Elvis_Center  {};
[2].    class Employee extends Elvis_Center  {
[3].    Properties      property = new Properties();
[4].    static int Records;
            int Management_Skills;
[5].    Employee Skill_legal (int skill) {
                    if (this.Management_Skills == skill)
[6].                   return this;
            }
        int RecordNo(int number) {
[7].                this.Records = number;
        }
[8].    };
        …
[9].    Employee J1 = new Employee();
        …
```

The class Employee that extends the owner Elvis_Center comes under the boundary of the owner, shown in Line[1]. Hence all the objects created for the class Employee will have their default dominant owner as Elvis_Center. Line[9] shows the Employee object J1, and hence the object *property* (Line[3]) within the aggregate object *J1* will have *J1* as its aggregate owner and Elvis_Center as its aggregate *J1's* dominant owner. The static keyword before the variable *Records* (Line[4]) specifies the shared entity and hence in Line[7] modifying the static variable will affect all the objects belonging to this class. Line[6] shows the function returning the this reference of the object which will create alias to that object. In Section 5.6 we shall see how we use our type system to restrict this kind of aliases.

5.4 ANTICIPATED OWNERS

In anticipated declaration, the related owners are declared statically by the owner using the keyword *neighbor*.

owner *Elvis_Center neighbor Ounda_Center* { };
owner *Ounda_Center neighbor Runa_Center* { };

The *neighbor* keyword is used to declare the related owners. The related owners are those to which the ownership transfer can happen. In the above code the owner Elvis_Center is related to Ounda_Center and Runa_Center . Ounda_Center, in turn, is related to the Runa_Center. The relation gives a directional graph, which means that objects from Elvis_Center can migrate/transfer to Ounda_Center or Runa_Center however the other direction is not possible, i.e., objects from Ounda_Center/Runa_Center cannot migrate/transfer to Elvis_Center. Thus neighbor forms a directional relationship graph between dominant owners.

5.5 UN-ANTICIPATED OWNERS

In unanticipated model the owner is free to bind dynamically to any other owner using the method *bind()* present in every owner. The following is an example of unanticipated ownership transfer:

```
class Related_Accounts extends Elvis_Center  {
static int No-of-Workers;
    record-Update() {
                        ---
    }
};
…
Ounda_Center  :: Assessor = new Related_Accounts() ;
…
```

To transfer objects between the owners the *migrate()* function has to be called. The function is present within every owner and it takes two arguments, *migrate(object, target-owner)*. In the above example, Assessor is the object created from the class Related_Accounts within the owner *Ounda_Center,* (i.e. Creating a new object Assessor for the class Related_Accounts within the dominant owner Ounda_Center)

```
…
Elvis_Center .migrate(Assessor, Runa_Center );
            // Error. The Elvis_Center  has no relation to
            Runa_Center
…
```

In the example, the dominant owner *Elvis_Center* has no anticipated relationship with *Runa_Center* . Thus trying to transfer the object Assessor to *Runa_Center* will result in error.

Thus to transfer the Assessor to *Runa_Center* , we need to establish dynamic relationship between the two dominant owners *Elvis_Center* and *Runa_Center*. This relationship can be established using the *bind()* function, that is present within every dominant owners.

```
…
Elvis_Center .bind(Runa_Center );     // Establishes directional
                                              //relationship
…
```

Thus in the above line, unanticipated neighborhood relationship has been established between the dominant owner *Elvis_Center* and the dominant owner *Runa_Center* .

Thus after establishing the unanticipated neighborhood relationship between the dominant owners it becomes now possible to transfer the object Assessor from *Elvis_Center* to *Runa_Center* as shown below.

```
…
Ounda_Center .migrate(Assessor, Elvis_Center );
            // OK. The Elvis_Center  is neighbor to Runa_Center
…
```

5.6 JMIGRATE TYPES

To indicate the object as confined in Jmigrate objects are annotated with the keyword <R> and non-confined objects with the keyword <Y>. The annotations are explained using the following example:

```
<R> class Some_Records extends Elvis_Center  {
static int Records;
int Personal_ID;
```

```
    userInfo( ) {
            /* gives details about the Employees */
    }
};

<Y> class Some_Related_Accounts  extends Ounda_Center  {
static int No-of-Managers;
static int No-of-Workers;
int Management_Skills;
    record-Update() {
                /* update the Accounts */
    }
};
```

Since the design is based on a class-based language, the annotation is given based on the class. In the above example the class Some_Records is represented as confined by annotating it with <R> type, and the class Some_Related_Accounts is represented as nonconfined by annotating it with <Y> type.

As per the property of the OTM the object type is restricted based on the class annotations. Thus during object creation the object can have its own type, either <R> or <Y> depending on the class annotations. This is because; during object creation we must have a facility to specify the alias property, like confined or non-confined, so that we cannot fix the alias type statically by class annotations. As an example,

```
Elvis_Center  :: <R> Coordinator = new Some_Records();

Elvis_Center  :: <Y> Another_Coordinator = new Some_Records();
```

In the above example, we are creating a new confined object Coordinator for the confined class Some_Records within the dominant owner Elvis_Center. Similarly we also try to create a new non-confined object called Another_Coordinator for the confined class Some_Records. But creating non-confined objects from a confined class is not permitted in Jmigrate. Thus the class annotated with the keyword <R> can produce only <R> typed objects. And on the other hand, creating new objects from non-confined classes are as follows:

```
…
Ounda_Center  :: <Y> Handle = new Some_Related_Accounts() ;
Ounda_Center  :: <R> Another_Handle = new
                                        Some_Related_Accounts() ;
…
```

In the above example, we are creating a new non-confined object Handle for the non-confined class Some_Related_Accounts within the dominant owner Ounda_Center. Similarly we also try to create a new confined object called Another_Handle for the same non-confined class Some_Related_Accounts. This is possible and allowed in Jmigrate, since during object creation there won't be any possible alias existence to the object.

Thus the class annotated with the keyword <Y> can produce both <R> typed objects and <Y> typed objects.

5.7 INHERITANCE AND SUBSUMPTION PROBLEM

Inheritance and subsumption are the two key properties of object-oriented programming systems. Subsumption is the ability to use a subclass object where an object of its superclass is expected. Substituting a subclass

object in place where superclass object is expected makes the object oriented systems more advantage and flexible for future design.

However, in Jmigrate we cannot permit subsumption as present in the literature of programming language. The problem here is with between the confinement and migration of an object. As an example consider a class Some_Records as follows:

```
Elvis_Center neighbor Ounda_Center {};
<Y> class Some_Records extends Elvis_Center  {
static int Records;
int Personal_ID;
    userInfo( ) {
            /* gives details about the Employees */
    }
};

Elvis_Center  :: <R> A = new Some_Records();
Elvis_Center  :: <Y> B = new Some_Records();

<Y> class Some_Inherited_Records extends Some_Records  {
…
};

Elvis_Center  :: <R> C = new Some_Inherited_Records();
Elvis_Center  :: <Y> D = new Some_Inherited_Records();
```

Thus in the above example the Some_Inherited_Records class inherits the class Some_Records. We have also created objects for the classes as above, where the object A and C are defined as confined objects and B and D are declared as non-confined objects. Thus if we are going to follow

subsumption property between these classes and then permitting subclass objects to substitute in place where we need superclass objects then we will get problems. i.e. subsuming C in place of B or D in place of A will create problems, where after subsuming C in place of B will allow C to migrate which is undesired property. Also subsuming D in place of A will also remove migration of the object D.

Usage of subsumption relationship may create problems in confinement and problems after ownership transfer. As an example, let us see how subsumption relation will weaken the confinement:

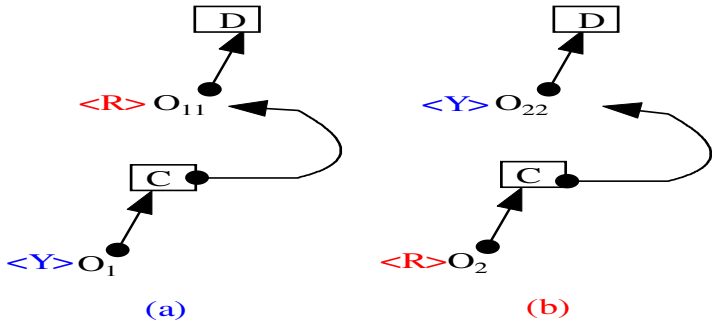

Figure 5.1 Subsumption Problems

In the Figure 5.1(a) that the object O1 is of type <Y> and if we are using subsumption relation then the O1 object will have the type D and can be substituted in place of O_{11} : D, but the expected object type for D is <R>. Similarly, in Figure 5.1(b) subsumption of the object O_2 typed <R> as type D, where the expected object type of D is <Y>. In the case of Figure 5.1(b) the problem is not with confinement breaching as done in (a) but this subsumption relation will create problem in expected result, due to migration property of <Y> objects.

Generally, the subtype relationship is decided from the subclass relationship based on hierarchical structure. However, in Jmigrate we also

consider the object types, hence we can restrict subsumption relation between objects present in the subclass to subsume to superclass objects if and only if their object types are same. i.e. <R> object of a subclass can subsume only another <R> objects of its superclass and similarly <Y> object of a subclass can subsume only another <Y> objects of its superclass. T

5.8 MULTIPLE-CLASS PROBLEM

The problem of multiple-class can be approached in two ways as per the OTM,

1. Allowing multiple existence of the class-instance, if and only if there is no static variables present in the class-instance.

As per the above point, multiple existence of the class-instance across the ownership boundary is possible as they won't affect other objects. This can be understood in the following example:

```
<Y> class Records extends Elvis_Center{
    // without static variables ;
}
Elvis_Center  :: <R> Coordinator = new Records;
Elvis_Center  :: <Y> Non_Confined_Coordinator = new Records;
…
Elvis_Center.diffuse(Non_Confined_Coordinator, Ounda_Center );
```

In the above example the class Records has no static variables. Two objects called Coordinator and Non_Confined_Coordinator are created. In the final line of the example the Non_Confined_Coordinator is migrated to the related owner Ounda_Center. Hence after ownership transfer we will have multiple existences of the class-instance Records in the owners *Elvis_Center*

and *Ounda_Center*. However, since there are no static variable, there is no multiple class existence problems.

The other point as per the OTM specifications is as follows:

2. Providing the facility of message passing between owners whenever the class-instance variables get altered.

As per the above point, multiple existence of the class-instance across the ownership boundary is allowed, provided there exists a facility of message passing. This can be explained in the following example:

```
<Y> class Records extends Elvis_Center  {
static boolean Employee_Permission_Status;
int Personal_ID;
    userInfo( ) {
            /* gives details about the Employees */
    }
};
Elvis_Center  :: <R> Coordinator = new Records;
Elvis_Center  :: <Y> Non_Confined_Coordinator = new Records;
Elvis_Center.diffuse(Non_Confined_Coordinator, Ounda_Center );
```

In the above example the class Records has static variable boolean Employee_Permission_Status. There are two objects called Coordinator and Non_Confined_Coordinator for this class. In the final line of the example we are transfering the Non_Confined_Coordinator to the related owner. Hence after ownership transfer we will have multiple existences of the class-instance Records in the owners *Elvis_Center* and *Ounda_Center*. Thus updating the static variable from Non_Confined_Coordinator present within the

Ounda_Center will indirectly affect the Confined object present within the Elvis_Center.

This modification is obvious from the scenario and is also common in the present software environment. However, in Jmigrate modifying the static variables through aliases is the problem we have to avoid. Thus with this motto we make that the Non_Confined_Coordinator within Ounda_Center can be accessed by other objects within the same owner Ounda_Center using read-only references alone. Hence this will not break the encapsulation and information hiding of the Non_Confined_Coordinator object.

5.9 DANGLING POINTER PROBLEM

The problem of dangling-pointer can be approached as follows,

1. Updating the migrating objects location to all its aliases.

Updating the migrating objects location to all its aliases it self a huge task, since the present programming languages object representation does not permit such updating and also updating will be costlier that leaving alias as such and go for garbage collections. In Jmigrate we follow indexed object representation which will permit us to handle the problem easier. As an example:

```
<Y> class Records extends Elvis_Center{
    int value = 10;
}
Elvis_Center  :: <R> Coordinator = new Records;
Elvis_Center  :: <Y> Non_Confined_Coordinator = new Records;
…
```

```
// Alias Creation
Coordinator = & Non_Confined_Coordinator;
// Non_Confined_Coordinator's owner is Elvis_Center

Coordinator.value = 15;          // change value to 15 within
                                 //Non_Confined_Coordinator
…
// Non_Confined_Coordinator's owner is Ounda_Center

Elvis_Center.diffuse(Non_Confined_Coordinator, Ounda_Center );
Coordinator.value = 20;          // change value to 20 within
                                 //Non_Confined_Coordinator
```

In the above example the class Records has two objects called Coordinator and Non_Confined_Coordinator. The Coordinator is alias to Non_Confined_Coordinator. In the final line of the example the Non_Confined_Coordinator is migrated to the related owner Ounda_Center. Hence after ownership transfer we will have dangling pointer from Coordinator pointing to the empty memory location previously acquired by Non_Confined_Coordinator.

As per the above point, migration of Non_Confined_Coordinator will create dangling pointer. Hence in Jmigrate during migration the pointer from Coordinator will also gets updated by permitting it to point to the new location of the Non_Confined_Coordinator within the Ounda_Center. This reference is termed as known reference, since it is already known in the previous originator owner Elvis_Center and hence part of the software design.

5.10 CONCLUSION

This chapter has presented the Jmigrate (Jm) language and how we are programming in Jmigrate are seen. We have also presented the syntax and types of the programming language Jm.

This chapter shows how we program the class relationship, dominant owners and aggregate owners using Jmigrate. We have also analyzed the side-effects that will occur due to object migration and the solution provided in the Jmigrate using various examples and Jmigrate codes.

CHAPTER 6

FORMAL DEFINITIONS

6.1 INTRODUCTION

The Jmigrate (Jm) language model is implemented using Featherweight Java (FJ) (Atsushi et al 2001). We shall discuss the syntax of the language in the following section and proceed with semantics and the type system of our language Jmigrate.

6.2 SYNTAX

Figure 6.1 presents the syntax of our language Jmigrate. The metavariables I, J, G and Z range over the owners names; A, B, C, D, S and T ranges over the class names; f and g range over the field names; m ranges over method names; x, y ranges over the parameter names; e, t ranges over terms; u, v ranges over values. CT range over class declarations, GT range over dominant owner declarations; M and K ranges over methods and constructor declarations; e_r range over certain restricted terms and e_r is formed as a logical group from term e; to represent the restricted expressions that are possible for classes between dominant ownership domain. *this* is a special variable and we consider it not to be used as parameter to a method, and is bound implicitly to every method. ξ represents the confinement property of the object and in syntax Figure 6.1 we have represented it as a pair; where $\xi.1$ represents the <R> type and $\xi.2$ represents the <Y> type. We use the bar over

the names to represent the sequences of that name. As an example $\bar{f}$ is a short hand notation for the sequence $f_1, f_2, \ldots, f_n$.

In our language we have used *extends* keyword to specify the relationship between two classes. The declaration *owner G neighbor Z* introduces the dominant ownership domain named G that has neighborhood relationship with other dominant owner called Z.

GT(G) ::= owner G neighbor Z

CT(C) ::= ξ class C extends D { ξ: $\overline{C}\ \bar{f}$; $\overline{M}$ K}

ξ ::= {<R> , <Y> }

e ::= x | e.f | e.m ($\bar{e}$) | new C ($\bar{e}$) | C (e)

e_r ::= x | e.f | e.m ($\bar{e}$) | C (e)

v ::= new C ($\bar{v}$)

K ::= C ($\overline{C}\ \bar{f}$) { Super ($\bar{f}$) ; this. $\bar{f}$ = $\bar{f}$ }

M ::= ψ B m ($\overline{B}\ \bar{x}$) { return e; }

Subclassing, Subtyping, and Matching

$\xi.0 = \bullet = <R>$ $\qquad$ $\xi.1 \rightarrowtail \xi.2$ $\qquad$ $C \cup \xi.2 <\# C \cup \xi.2$

$$\frac{B \cup \xi.1 <_c C \cup \xi.2}{B <: C} \qquad \frac{B \cup \xi.2 <_c C \cup \xi.2 \quad C \cup \xi.2 <_c D \cup \xi.1}{B <\# D \ \wedge\ C <\# D}$$

Θ - Type environment: mapping from variables to External types

Γ - Type environment: mapping from variables to class-is- types

Δ - Type environment: mapping from variables to nonvariable types

δ - Type environment: mapping from nonvariable to nonvariable types

δ_Δ - Type environment: mapping from nonvariable type to variable types

Φ_D - corresponds to dominant owner to which D belongs (i.e. current owner)

ρ_D - corresponds to neighborhood owenr (relative owners) of D's owner

Figure 6.1 Syntax of Jmigrate

ξ class C extends D $\{\xi: \overline{C}\ \bar{f};\ \overline{M}$ K$\}$ introduces a class named C with confinement property ξ which has the superclass D. The class C has fields $\bar{f}$ with types $\overline{C}$ and external types ξ, a single constructor K and methods $\overline{M}$. The class table CTD(C) is a mapping from class names C to class declarations CT, and the ownership table GT is a mapping from owner names G to the owner declarations GT. By default, external type is <R>. The relation between external type is that, <R> is highly coarse than <Y> type. Therefore, <R> lies within the owner. Here, *subclassing* and *subtyping* are differentiated using *matching_like* relation between objects. So **<:**$_C$ used to represent the subclassing and **<:** used to represent subtyping. To have proper typing relation that represent either subtyping or subclassing the *matching_like* relation (**<#**) is used. The union ($\cup$) symbol represent the combination of class name and the external type. The lookup functions are given in Figure 6.2, Figure 6.3.

Field Lookup

GT(G) = owner G neighbor Z

CT(D) = ξclass D extends G $\{\xi: \overline{D}\ \bar{g};\ \psi.1\ \overline{M}$ K$\}$

CT(C) = ξ class C extends D $\{\xi: \overline{C}\ \bar{f};\ \psi.1\ \overline{M}$ K$\}$

fields(C) = $\overline{C}\ \bar{f};\ \overline{D}\ \bar{g} \in \xi.2$ fields(D) = $\overline{D}\ \bar{g}$

Method Lookup

GT(G) = owner G neighbor Z

CT(D) = ξ class D extends G $\{\xi: \overline{D}\ \bar{g}\ ;\ \psi.2\ \overline{M}_1K\}$

B m($\overline{B}\ \bar{x}$) {return e;} $\in \overline{M}_1$

methods(G) = $\overline{M}_1$

mtype(m,G) = (D, $\overline{B} \rightarrow$ B, this)

Figure 6.2 Field and Method Lookup in Jmigrate

Method Type Lookup

$$\frac{\text{CT(C)} = \xi\ \text{class C extends D}\ \{\xi:\ \overline{C}\ \bar{f};\ \psi.1\ \overline{M}\ \text{K}\} \qquad \text{B m}(\overline{B}\ \bar{x})\ \{\text{return e;}\} \in \overline{M}}{\text{mtype(m, C)} = (\text{C},\ \overline{B} \rightarrow \text{B},\ \text{this}_{\text{C}})}$$

$$owner_{\Delta}(\text{C}) = owner_{\Delta}(\text{D}) = \text{G}$$

$$\frac{\text{CT(D)} = \xi\ \text{class D extends G}\ \{\xi:\ \overline{D}\ \bar{g};\psi\ \overline{M}\ \text{K}\ \} \qquad \text{CT(C)} = \xi\ \text{class C extends D}\ \{\xi:\ \overline{C}\ \bar{f};\psi.1\ \overline{M}_1\ \text{K}\} \qquad \text{m not defined in } \overline{M}_1}{\text{mtype(m, C)} = \text{mtype(m, D)}}\ \text{M-DELEGATION}$$

$$\frac{\textbf{But IF}\ \text{m defined in}\ \psi.2\ \overline{M}}{\text{mtype(m, C)} = \text{mtype(m, D)} = \text{mtype(m, G)}}\ \text{M-MIGRATION}$$

$$\frac{\text{CT(D)} = \xi\ \text{class D extends Z}\ \{\xi:\ \overline{D}\ \bar{g};\psi\ \overline{M}\ \text{K}\ \} \qquad owner_{\Delta}(\text{C}) = \text{G} \quad \Re_{\delta(owner_{\Delta}(C))} = \text{Z} \qquad \text{CT(C)} = \xi\ \text{class C extends B}\ \{\xi:\ \overline{C}\ \bar{f};\psi.1\ \overline{M}_1\ \text{K}\} \qquad \text{migrate(new C}(\text{v}_0), \text{G, Z, } \xi.2\ ) \qquad \text{m not defined in } \overline{M}_1 \quad \text{and} \quad \text{m defined in } \psi.2\ \overline{M}}{\text{mtype(m, C)} = \text{mtype(m, Z)}}$$

Figure 6.3 Method Type Lookup in Jmigrate

The auxiliary definitions are listed in the Figure 6.4. The definition method(G) returns the method present in the owner G, and the mtype returns the type signature of a method. Here in order to have a clear idea about *this* (*this*$_D$) we have included *this* definition along with the methods signature.

Method Body Lookup

$$\frac{CT(C) = \xi\ \text{class C extends D}\ \{\xi:\ \overline{C}\ \overline{f};\ \psi\ \ \overline{M}\ K\} \qquad B\ m(\overline{B}\ \overline{x})\ \{\text{return e;}\} \in \overline{M}}{mbody(m, C) = (\overline{x}, e)}$$

$$\frac{CT(C) = \text{class C extends D}\ \{\xi:\overline{C}\ \overline{f};\ \psi\ \ \overline{M}\ K\} \qquad M\ \text{not defined in}\ \overline{M}}{mbody(m, C) = mbody(m, D)}$$

$$\frac{CT(D) = \xi\text{class D extends G}\ \{\xi:\overline{D}\ \overline{g};\ \psi.1\,\overline{M}_1\ \ \psi.2\ \ \overline{M}_2\} \qquad CT(C) = \xi\ \text{class C extends D}\ \{\xi:\ \overline{C}\ \overline{f};\psi\ \overline{M}\ K\} \qquad m\ \text{not defined in}\ \overline{M}\ \ \wedge\ B\ m(\overline{B}\ \overline{x})\ \{\text{return e;}\} \in \overline{M}_2}{mbody(m, C) = mbody(m, D) = mbody(m, G)\ [\ \text{where mbody(m, G) = mbody(m, D)}\ ]}$$

$$\frac{CT(D) = \xi\text{class D extends G}\ \{\xi:\overline{D}\ \overline{g};\ \psi.1\,\overline{M}_1\ \ \psi.2\ \ \overline{M}_2\} \qquad CT(C) = \xi\ \text{class C extends D}\ \{\xi:\ \overline{C}\ \overline{f};\psi\ \overline{M}\ K\} \qquad visible_{\Delta,\delta}(Z, G) \qquad migrate(\text{new C}(v_0), G, Z,\ \xi.2\ ) \qquad m\ \text{not defined in}\ \overline{M}}{mbody(m, v_0) = mbody(m, Z)}$$

Figure 6.4 Auxiliary Definition in Jmigrate

The predicate override is overriding normal methods present in the superclass. In override, *this* in the parameter used to specify *this* pointer of the method after overriding. It is used to specify the combined usage of delegation and inheritance. Observe that $this_D$ says, whenever this method is invoked it depends on fields of the class D; while the similar thing with this, which depends on the contents in migrated owner. By the rule of D_{obj} during delegation the objects clone will be there in the migrated owner, which says a secure delegation occurs here.

6.3 EVALUATION

Jmigrate has a call-by-value semantics (Figure 6.5). The three evaluation rules E-NFIELD, E-NCAST, E-NINVK represents the basic evaluation rules. The evaluation rules deal with field access, type casting and the method invocation by an object. The object can externally access the fields only when it belongs to $\xi.2$ and similarly method invocation is possible only when the method belongs to normal functional property.

Valid Method Overriding

$$\frac{\text{mtype(m, C)= (C , } \overline{B} \rightarrow \text{B, this}_C) \Rightarrow (\text{D <: C , } C_0 = \text{B} \wedge \overline{C}_0 = \overline{B}\text{)}}{\text{override(m, } C_0, \overline{C}_0 \rightarrow C_0, \text{this}_D)}$$

$$\frac{\text{mtype(m, G) = (D, } \overline{B} \rightarrow \text{B, this)} \Rightarrow \text{ (C <\# D, } C_0 = \text{B, } \overline{C}_0 = \overline{B}\text{)}}{\text{override(m, D, } \overline{C}_0 \rightarrow C_0, \text{this)}}$$

$$\frac{\text{fields}(C_0) = \overline{C}\ \overline{f}}{\text{new } C_0(\overline{v}).\text{f}_\text{i} \rightarrow \text{v}_\text{i} \Rightarrow f_i \in \xi.2} \quad \boxed{\text{E-NFIELD}}$$

$$\frac{\text{C <: D}}{\text{(D) new C}(\overline{v}) \rightarrow \text{new C}(\overline{v})} \quad \boxed{\text{E-NCAST}}$$

$$\frac{\text{mbody(m, C) = } (\overline{x}, \text{e})}{new\ C(\overline{v}).m(\overline{u}) \rightarrow \left[\overline{u}/\overline{x}, new\ C(\overline{v})/this_C \right] e \Rightarrow m \in \psi.1} \quad \boxed{\text{E-NINVK}}$$

$$\frac{\text{e} \rightarrow \text{e'}}{\text{new C(e)} \rightarrow \text{new C(e')}} \qquad \frac{\text{e} \rightarrow \text{e'}}{\text{(C)e} \rightarrow \text{(C)e'}} \qquad \frac{\text{e} \rightarrow \text{e'}}{\text{e.f} \rightarrow \text{e'.f}}$$

$$\frac{\text{e} \rightarrow \text{e'}}{\text{e.m}(\overline{e}) \rightarrow \text{e'. m}(\overline{e})} \qquad \frac{\text{e} \rightarrow \text{e'}}{\text{e}_0\text{.m(e)} \rightarrow \text{e}_0\text{.m(e')}}$$

$$\text{migrate}(\text{v}_0, \text{G}, \text{Z}, \xi) \quad \text{where } \text{v}_0 = \text{C} \cup \xi.2$$

Figure 6.5 Evaluations Rules in Jmigrate

6.4 VISIBILITY RULE

The visibility rule shown in Figure 6.6, Figure 6.7 and Figure 6.8 is the foundation of the Jmigrate in determining the owners, types and terms that are visible to the particular class for using it within the class.

Basic Rules

$owner_{\Delta}(T)$	; determines the owner of type T
$\Re_{\delta}(owner_{\Delta}(T))$	; determines the neighborhood owners
$\Delta \vdash$ T OK	; type T is OK
$visible_{\Delta}$ (O, D)	; owner O is visible in class D
$visible_{\Delta,\delta}(\tilde{O}, O)$	; relative owners $\tilde{O}$ is visible to owner O
$equal_{\delta}(\mathring{O}, \H{O})$	; both owners $\mathring{O}$ and $\H{O}$ are equal
$visible_{\Delta}$ (T, D)	; type T is visible in class D
$\Theta;\Gamma;\Delta \vdash visible$ (e, D)	; expression e is visible in class D
$\Theta;\Gamma;\Delta \vdash Nvisible$ (e, D)	; expression e is not visible in class D
$\Theta;\Gamma;\Delta;\delta \vdash visible_{\delta_{\Delta}}$ (e_r, D)	; e_r is visible in class D
$\Theta;\Gamma;\Delta;\delta \vdash visible_{\delta_{\Delta}}$ (T, D)	; type T is visible in class D

Indirect Visibility

$\Theta;\Gamma;\Delta;\delta \vdash ivisible_{\delta_{\Delta}}$ (e, D)	; expression e is visible for class D
$\Theta;\Gamma;\Delta;\delta \vdash ivisible_{\delta_{\Delta}}$ (T, D)	; type T is visible for class D
$\Theta;\Gamma;\Delta;\delta;\delta_{\Delta} \vdash Nivisible$ (e, D)	; expression e is not visible in class D

Figure 6.6 Judgments in Jmigrate

Visibility rules divided into three sets of rules: determining ownership visibility, type visibility and term visibility. Here, Φ_D function is used to return the owner of the class D; and the function ρ_D that returns the

relative owners of the owner of the class D. The owner visibility checks that the given owner is either the owner of the current class, representing the ownership; and checks that the relative owners related to the current owner of the current class determining the related ownership domain. The type visibility allows the usage of types based on the owner visibility. The type visibility rule is also applicable between the owners, stated through the predicate *ivisible*. The indirect visibility of the type represents through the delegation relation. Here the type refers to the class name and are not referring the external types.

Owner Visibility

$$visible_{\Delta}(O,D) = O \ \in \ \Phi_{\mathrm{D}}$$

$$where \qquad owners(D) = \left\{ \begin{array}{l} J' \in J \quad \wedge \quad \Re(J') \\ if \\ CT(D) = \xi \ class \ D \ extends \ J' \{\ldots\} \\ and \quad GT(J') = owner \ J' \{\ldots D \ldots\} \end{array} \right\}$$

$visible_{\Delta,\delta}(O, D) = O \in \Phi_D \wedge \rho_D$ where,

$$owners(D) = \left\{ \begin{array}{l} J' \in J \quad \wedge \quad \Re(J') \\ if \\ CT(D) = \xi \ class \ D \ extends \ J' \{\ldots\} \\ and \quad GT(J') = owner \ J' \ neighbor \ I' \{\ldots D \ldots\} \end{array} \right\}$$

$$\Re(\Phi) = \left\{ \begin{array}{l} I' \in \overline{I} \\ if \\ GT(\Phi) = owner \ \Phi \ \ neighbor \ I' \{\ldots\} \end{array} \right\}$$

Figure 6.7 Owner Visibility

Term visibility rules states that for each subexpression, the rules determine whether the type of that subexpression is visible according to the type visibility rules. Extending term visibility between owners through delegation relation gives indirect visibility. The predicate *Nvisible* checks whether the respective term is visible for a particular class based on the external type. Similarly, the predicate *Nivisible* checks term visibility between the owners.

During object migration the terms present in neighbors owner may be visible to the class present in other owner.

The predicate $\Theta;\Gamma;\Delta;\delta \vdash visible_{\delta_\Delta}(e_r, D)$ determines the term visibility. Here we have specified the restricted term visibility because *new* is not available after object migrated to the neighbor owner.

Type Visibility

$$visible_\Delta(T, D) = visible_\Delta(owner_\Delta(T), D)$$

$$\Gamma; \delta; \delta_\Delta \vdash ivisible_\Delta(T, D) = \Gamma;\Delta;\delta_\Delta \vdash equal_\delta(\Re_\delta(owner_\Delta(D), owner_\Delta(T)))$$

Term Visibility

$$\frac{\Theta;\Gamma;\Delta \vdash x : T \cup E \quad visible_\Delta(T, D)}{\Theta;\Gamma;\Delta \vdash visible(x, D) \Rightarrow E \in \xi.2} \quad \boxed{\text{V - VAR}}$$

If $D \cup \xi.1$ then $\Theta;\Gamma;\Delta \vdash visible(x, D) \Rightarrow E \in \xi$

$$\frac{\Theta;\Gamma;\Delta \vdash x' : T \cup E \quad visible_\Delta(T, D)}{\Theta;\Gamma;\Delta \vdash Nvisible(x', D) \Rightarrow (E \in \xi.1) \wedge (D \cup E \text{ where } E \in \xi.2)} \quad \boxed{\text{V - NVAR}}$$

Figure 6.8 Type and Term Visibility in Jmigrate (Continued)

$\Theta;\Gamma;\Delta \vdash visible(e, D)$ $\quad \Theta;\Gamma;\Delta \vdash e.f_i : T \cup E'$

$\Theta;\Gamma;\Delta \vdash e.f_j : T \cup E''$ $\quad visible_{\Delta}(T, D)$ $\quad i \neq j$ $\quad E', E'' \in E$ $\quad$ V - FIELDS

If $D \cup \xi.1$ then $\Theta;\Gamma;\Delta \vdash visible(e.f, D) \Rightarrow E \in \xi$

If $D \cup \xi.2$ then $\Theta;\Gamma;\Delta \vdash visible(e.f_i, D) \Rightarrow E' \in \xi.2$

and $\Theta;\Gamma;\Delta \vdash Nvisible(e.f_j, D) \Rightarrow E'' \in \xi.1$

$\Theta;\Gamma;\Delta \vdash e.\ m(\bar{e}) : T \cup E$ $\quad visible_{\Delta}(T, D)$ $\quad$ V - MINVK

$\Theta;\Gamma;\Delta \vdash visible(e, D)$ $\quad \Theta;\Gamma;\Delta \vdash visible(\bar{e}, D)$

$\Theta;\Gamma;\Delta \vdash visible(e.m(\bar{e}), D) \Rightarrow e.m(\bar{e}) : T \cup E$, where $E \in \xi.2$

and If $D \cup \xi.1$ then $\Theta;\Gamma;\Delta \vdash visible(e.m(\bar{e}), D) \Rightarrow E \in \xi$

$\Theta;\Gamma;\Delta \vdash visible(\bar{e}, D)$ $\quad visible_{\Delta}(T, D)$ $\quad T = T \cup E$ $\quad$ V - NEW

$\Theta;\Gamma;\Delta \vdash visible(new\ T(\bar{e}), D) \Rightarrow \bar{e} : T \cup E$, **If** $E \in \xi.2$

Else $\bar{e} : T \cup \xi.1$**, If** $E \in \xi.1$

$\Theta;\Gamma;\Delta \vdash visible(e, D)$ $\quad visible_{\Delta}(T, D)$ $\quad$ V - CAST

$$\Theta;\Gamma;\Delta \vdash visible((T)e, D) \Rightarrow \left\{ \begin{array}{l} e : \_ \cup \xi.2 \wedge T \in E \\ \vee \\ e : \_ \cup \xi.1 \wedge T \in \xi.1 \end{array} \right\}$$

Figure 6.8 Type and Term Visibility in Jmigrate (Continued)

$\Theta;\Gamma;\delta;\delta_\Delta \vdash$ y: S ∪ E $\Theta;\Gamma;\Delta;\delta \vdash ivisible_{\delta_\Delta}$ (S, D)

V_R - VAR

$\Theta;\Gamma;\delta;\delta_\Delta \vdash$ x: S ∪ E

$\Theta;\Gamma;\Delta;\delta \vdash ivisible_{\delta_\Delta}$ (y, D) ⇒ (E∈ ξ.2)

V_R - INVK

$\Theta;\Gamma;\delta;\delta_\Delta \vdash Nivisible(x, D) \Rightarrow$ (E∈ ξ.1)

$\Theta;\Gamma;\Delta \vdash$ e.m($\bar{e}$) : S ∪ E $\Theta;\Gamma;\Delta;\delta \vdash ivisible_{\delta_\Delta}$ (S, D)

$\Theta;\Gamma;\Delta \vdash visible\ (e, D)$ $\Theta;\Gamma;\Delta \vdash visible\ (\bar{e}, D)$

$\Theta;\Gamma;\Delta \vdash Nvisible(e.m(\bar{e}), D)\ \ e = this_D \mid this\_\ ;\ where\ \ equal_\delta(owner(\_), owner(D))$

$\Theta;\Gamma;\Delta;\delta \vdash visible_{\delta_\Delta}(e_1.m(\bar{e}_1), S)\ \ mbody(m, S) = (\bar{u}, e'')$

$\Theta;\Gamma;\Delta;\delta \vdash ivisible_{\delta_\Delta}(e_1.m(\bar{e}_1, D))\ \ ;where\ \ \left[\bar{e}_1/\bar{u} \wedge this_{fp}/e_1\right] e''\ \ \wedge\ \ e_1.m(\bar{e}) : S \cup \psi.2$

Figure 6.8 Type and Term Visibility in Jmigrate

The V_R–INVK rule specifies the reference relation between the owners. The Figure 6.9 says about the class and method type. The IN-METHOD and DEL-METHOD specifies functions visibility within and between owners respectively. Similarly, the CLASS+I and CLASS+D says about class visibility within and between owners respectively.

$$\frac{\Gamma;\Delta;\bar{x}:\bar{B},this:C \vdash t_0 : D_0 \cup E \qquad \Delta \vdash \bar{B}, B, D_0 \qquad visible_{\Delta}(t_0, C) \quad visible_{\Delta}(B, C) \quad visible_{\Delta}(\bar{B}, C)}{B\ m(\bar{B}\ \bar{x})\ \{return\ t_0;\}\ OK\ in\ C}$$

IN-METHOD

$$\frac{\Gamma;\delta;\delta_{\Delta};\Delta,\bar{x}:\bar{B},this:this_{fp} \vdash t_0 : D_0 \cup E \quad \delta;\delta_{\Delta};\Delta \vdash \bar{B}, B, D_0 \qquad ivisible_{\Delta}(\bar{B}, C) \quad ivisible_{\Delta}(B, C) \quad ivisible(D_0, C)}{B\ m(\bar{B}\ \bar{x})\ \{return\ t_0;\}\ OK\ in\ C}$$

DEL-METHOD

$$\frac{\Delta \vdash \bar{S} <: \bar{N}\ \text{OK} \quad visible_{\Delta}(\text{N, C}) \quad \Delta \vdash \text{N}, \bar{N}, \bar{T}\ \text{OK} \qquad \bar{M}\ \text{OK in C} \quad visible_{\Delta}(\bar{T}, \text{C}) \quad visible_{\Delta}(\bar{N}, \text{C}) \quad visible_{\Delta}(\bar{S}, \text{C})}{\text{Class C extends N } \{\xi : \bar{S}\ \bar{f};\psi\ K\ \bar{M}\}OK}$$

CLASS+I

$$\frac{\Theta;\Gamma;\Delta;\delta \vdash ivisible_{\delta_{\Delta}}(\bar{T}, \bar{N}) \quad \Theta;\Gamma;\Delta;\delta \vdash ivisible_{\delta_{\Delta}}(\bar{T}, \text{C}) \qquad \Delta \vdash \text{N}, \bar{N}\ \text{OK} \quad \Delta;\delta;\delta_{\Delta} \vdash \bar{S}, \bar{T}\ \text{OK} \qquad \Theta;\Gamma;\Delta;\delta \vdash visible_{\delta_{\Delta}}(\bar{S}, \text{C}) \quad visible_{\Delta}(\bar{N}, \text{C}) \quad visible_{\Delta}(\text{N, C})}{\text{Class C extends N } \{\xi : \bar{S}\ \bar{f};\psi\ K\ \bar{M}\}OK}$$

CLASS+D

Figure 6.9 Class and Method Type in Jmigrate

6.5 PROPERTIES

In this section, the properties of the language Jmigrate, and the nature of confined objects based on ownership criteria are discussed. The proof says that during execution, all expression result in an instance of a class is visible within the current context, and also after the migration, the visible portion for the object will be reversed but will be provided through delegation.

LEMMA 1. **Subject reduction**

If $\Gamma;\Delta \vdash e : C \cup E$ and $e \to e'$ then $\Gamma;\Delta \vdash e' : C' \cup E$ for some C' **<#** C

LEMMA 2. **Hierarchical Objects-Ownership Invariance**

If $\Delta \vdash$ S **<:** T; $\Delta \vdash$ T **<:** G then $owner_{\Delta}(S) = owner_{\Delta}(T) = G$ where G is the dominant ownership domain.

Proof. By induction on the depth of the subtype hierarchy and by the property of the language P1 every class must be either within some space (global or dominant ownership domain). By CLASS + I a class extends from a ownership domain and all the subclasses of this class will have the same owner which is the ownership domain.

LEMMA 3. **Ad hoc objects-Ownership Invariance**

If $owner_{\Delta}(T) = G$ and $visible_{\Delta}(T, S)$, mtype (m, S) = mtype (m, G) where mtype(m,G) = $(T, \overline{B} \to B, this)$ then $owner_{\Delta}(S) = owner_{\Delta}(T) = G \ \wedge \ \Delta \vdash$ S **<#** T

Proof. By induction on the relationship between ad hoc objects within the ownership domain. Where T provides the method required by S indirectly through *finger* function. This implies that there is a delegation relation between S and T through owner G. hence there is no subclassing but subtyping relation between them. By CLASS+D a class have visibility of type in other domain which implies movement of object of the particular type from that related domain to its native..

LEMMA 4. **Ownership Invariance in Object Migration**

If $\Gamma;\Delta \vdash u : S \cup \xi.1$, $\Gamma;\Delta \vdash v : S \cup \xi.2$, $\Delta \vdash$ S **<#** T ;where $owner_{\Delta}(S) = O \ \wedge \ owner_{\Delta}(T) = O'$ and $visible_{\Delta,\delta}(O', O)$ is the initial location of the types and ,If

move(v, O', E) $\Rightarrow$ $v \to v'$ for $E \in \xi.2$ then $\Gamma;\delta;\delta_\Delta \vdash owner_\Delta(S) = O'$ and $owner_\Delta(S) = O$

Proof. By induction based on the objects migration between ownership domain saying that objects moving between domains will also carry the class-is-type property but prevails based on the objects type for its rights determination. It also implies that the persistence of the class within the previous owner helps in creation of further objects from the same class.

THEOREM 1. **Confinement Invariant (Inside Owner)**

Let e be a subexpression appearing in the body of a IN-METHOD of a well formed class C defined by CLASS+I. Then: If $e \to^* new\, D(\bar{e})$, then $visible_\Delta(D, C)$.

Proof. Since the class is well formed, its methods are also well formed. In the environment $\Gamma;\Delta \vdash$ e:T and $\Gamma \vdash visible_\Delta(e, C)$ holds, which implies that $visible_\Delta(T, C)$ and hence $visible_\Delta(owner_\Delta(T), C)$. Then by subject reduction property, there is a T' such that $\Gamma;\Delta \vdash$ new D($\bar{e}$): T', where $\Gamma;\Delta \vdash$ T' **<#** T means there is another possibility as $\Gamma;\Delta \vdash$T'**<:** T. Specifically Ad hoc Objects-Ownership Invariance possibly has Hierarchical Objects-Ownership Invariance in it. Therefore by LEMMA 2 and LEMMA 3 $owner_\Delta(D) = owner_\Delta(T)$ which implies $visible_\Delta(owner_\Delta(D), C)$ and hence $visible_\Delta(D, C)$. □

THEOREM 2. **Confinement Invariant (Between Owners in Object Migration)**

Let e be a subexpression appearing in the body of a DEL-METHOD of a well formed class C defined by CLASS+D. Then: If $e \to^* new\, D(\bar{e})$, then $visible_\Delta(D, C)$.

Proof. Since the class is well formed, its methods are also well formed. In the environment $\Gamma;\Delta;\delta;\delta_\Delta \vdash e_r:T$ and $\Theta;\Gamma;\Delta;\delta \vdash visible_{\delta_\Delta}(e_r, C)$ holds, which implies that $\Theta;\Gamma;\Delta;\delta \vdash visible_{\delta_\Delta}(T, C)$ and hence $visible_{\Delta,\delta}(\Re_\delta(owner_\Delta(T)), owner(C))$. Then by subject reduction property, there is a T' such that $\Gamma;\Delta \vdash$ new D($\bar{e}$): T', where $\Gamma;\Delta \vdash$ T' <# T. Therefore by Ad hoc Objects-Ownership Invariance $owner_\Delta(D) = owner_\Delta(T)$ which implies $visible_\Delta(owner_\Delta(D), C)$ and hence $visible_\Delta(D, C)$. □

THEOREM 3. **Confinement Invariant (After Object Migration)**

Let e be a subexpression appearing in the body of a DEL-METHOD of a well formed class C defined by CLASS+D. Then: If $e \to^* new\, D(\bar{e})$, then $visible_\Delta(D, C)$ and If $visible_\Delta(S, D)$ and $v \to v'$ for $E \in D \cup \xi.2$ then $\Theta;\Gamma;\Delta;\delta \vdash ivisible_{\delta_\Delta}(S, D)$.

Proof. Since the class is well formed, its methods are also well formed. In the environment $\Gamma;\Delta;\delta;\delta_\Delta \vdash e:T$ and $\Theta;\Gamma;\Delta;\delta \vdash ivisible_{\delta_\Delta}(e, C)$ holds, which implies that $\Theta;\Gamma;\Delta;\delta \vdash ivisible_{\delta_\Delta}(T, C)$ and in the environment $\Gamma;\Delta \vdash$ S OK and $\Gamma;\Delta \vdash visible_\Delta(S, D)$ with $visible_{\Delta,\delta}(\Re_\delta(owner_\Delta(T)), owner(C))$. Then by subject reduction property, there is a T' such that $\Gamma;\Delta \vdash$ new D($\bar{e}$): T' $\cup E$, where $\Gamma ; \Delta \vdash$ T' <# T. Therefore by ownership variance $owner_\Delta(D) = owner_\Delta(T)$ for some $v : D \cup \xi.2$ and $owner_\Delta(D) \neq owner_\Delta(T)$ for some v : D $\cup E$ therefore according to the language properties P5, which implies that $\Theta;\Gamma;\Delta;\delta \vdash ivisible_{\delta_\Delta}(S, D)$. Finally based on ownership variance which implies $\Gamma;\Delta \vdash visible_\Delta(D, C)$ $visible_\Delta(owner_\Delta(D), C)$ and hence $visible_\Delta(D, C)$. □

CHAPTER 7

CONCLUSIONS

7.1 PROPOSED WORK

In this thesis, we propose ownership types as the encapsulation policy that will be useful to predict the object encapsulation statically and facilitate local reasoning about program correctness in object-oriented languages, and the ownership transfer (anticipated and un-anticipated) that is helpful in designing the system for future changes.

7.2 CONTRIBUTIONS

Our proposed model, namely Ownership Transfer Model (OTM) exploits this combination and shows how to achieve the flexibility of prototype-based systems without abandoning the advantages of the class-based paradigm. However, as we propose to combine the two advantages, we have shown that the method is not out of hazards. Problems arising out of such combination have been discussed and solutions in OTM have been proposed. The evidence of safety in OTM has been illustrated using a language called Jmigrate (Jm), which is based on the Featherweight Java (FJ).

7.3 FUTURE WORK

As a future enhancement, we are planning to add delegation and subtype mechanism which will help us to relate objects after migration. Thus delegation will help us to modify the inheritance hierarchy for an object dynamically. Secure delegation with ownership encapsulation will help us to have a better system permitting dynamic evolution and secure static typing.

REFERENCES

1. Abadi M. and Cardelli L. (1996a), 'A Theory of Objects', Springer-Verlag.

2. Abadi M. and Cardelli L. (1996b), 'On Subtyping and Matching', ACM Transactions on Programming Languages and Systems (TOPLAS), ACM Press, Vol. 18(4), pp. 401-423.

3. Aldrich, Kastadinov V. and Chambers C. (2002), 'Alias annotations for program understanding', in Proceedings of the 17th ACM SIGPLAN conference, Object Oriented Programming Systems Languages and Applications (OOPSLA), ACM Press, Vol. 37(11), pp. 311-330.

4. Almeida P.S. (1997), 'Balloon Types: Controlling Sharing of State in Data Types', in Proceedings of European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag, pp. 32-59.

5. Ancona D., Anderson C., Damiani F., Drossopoulou S, Giannini P. and Zucca E. (2001), 'An Effective Translation of Fickle into Java', in Proceedings of the 7th Italian Conference on Theoretical Computer Science, Lecture Notes In Computer Science (LNCS), Springer-Verlag,Vol. 2202, pp. 215-234.

6. Banerjee A. and Naumann D.A. (2005), State Based Ownership, Reentrance, and Encapsulation, in Proceedings of European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science (LNCS), Springer Berlin/Heidelberg, Vol. 3586, pp. 387-411.

7. Bardou D. and Dony C. (1996), 'Split Objects: a Disciplined Use of Delegation within Objects', in Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA), pp. 122-137.

8. Bieman J.M. and Kang B.K. (1995), 'Cohesion and reuse in an object-oriented system', in Proceedings of ACM Symposium on Software Reusability (SSR), ACM Press, pp. 259-262.

9. Blaschek G. (1994), 'Object-Oriented Programming with Prototypes', Springer-Verlag.

10. Bornat R., Calcagno C., O'Hearn P. and Parkinson M. (2005), 'Permission Accounting in Separation Logic', in Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), ACM Press, pp. 259-270.

11. Borning A.H. (1986), 'Classes Versus Prototypes in Object-Oriented Languages', in Proceedings of ACM/IEEE Fall Joint Computer Conference, IEEE Computer Society Press, pp. 36-40.

12. Boyapati C., Lee R. and Rinard M. (2002), 'Ownership Types for Safe Programming: Preventing Data Races and Deadlocks', in Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA), ACM Press, pp. 211-230.

13. Boyapati C., Liskov B. and Shrira L. (2003), 'Ownership Types for Object Encapsulation', in ACM SIGPLAN Notices, ACM Press, Vol. 38(1), pp. 213-223.

14. Boyapati C. and Rinard M.C. (2004), 'Safejava: a unified type system for safe programming', Massachusetts Institute of Technology.

15. Boyland J. and Retert W. (2005), 'Connecting Effects and Uniqueness with Adoption', in Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), ACM Press, pp. 283-295.

16. Bruce K. (1996), 'Typing in object-oriented languages: Achieving expressibility and safety', Technical Report, Williams College.

17. Bruce K., Cardelli L., Castagna G., The Hopkins Objects Group, Leavens G.T. and Pierce B.C. (1995a), 'On Binary Methods', in Theory and Practice of Object Systems, John Wiley & Sons Inc, Vol. 1(3), pp. 221-242.

18. Bruce K., Schuett A. and Gent R.V. (1995b), 'PolyTOIL: A type-safe polymorphic object-oriented language', in Proceedings of European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science (LNCS), Springer-Verlag, Vol. 952, pp. 27-51.

19. Cameron N.R., Drossopoulou S., Noble J. and Smith M.J. (2007), 'Multiple Ownership', in Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA), ACM Press, pp. 441-460.

20. Cardelli L. and Wegner P. (1985), 'On understanding types, data abstraction, and polymorphism', ACM Computing Surveys, Vol. 17(4), pp. 471-522.

21. Chambers C. (1993), 'The Cecil language: Specification and rationale', Technical Report tr 93-03-05, Department of Computer Science and Engineering, University of Washington.

22. Chambers C., Ungar D., Chang B.W. and Holzle U. (1991), 'Parents are Shared Parts of Objects: Inheritance and Encapsulation in SELF', in Journal on Lisp and Symbolic Computation, Kluwer Academic Publishers, Vol. 4(3), pp. 207-222.

23. Clarke D.G. (2001), 'Object Ownership and Containment', PhD Thesis, University of New South Wales.

24. Clarke D.G. and Drossopoulou S. (2002), 'Ownership, Encapsulation and the Disjointness of Type and Effect', in Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA), ACM Press, pp. 292-310.

25. Clarke D.G. and Wrigstsad T. (2003), 'External Uniqueness is Unique Enough', in Proceedings of European Conference on Object-Oriented Programming (ECOOP), Springer Berlin/Heidelbergm, Vol. 2743, pp. 59-67.

26. Clarke D.G., Potter J., and Noble J. (1998), 'Ownership Types for Flexible Alias Protection', in Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA), ACM Press, pp. 48-64.

27. Danforth S. and Tomlinson C. (1988), 'Type theories and object oriented programming', ACM Computing Surveys, Vol. 20(1), pp. 29-72.

28. Dietl W. and Müller P. (2005), 'Universes: Lightweight Ownership for JML', in Journal of Object Technology (JOT), Vol. 4(8), pp. 5-32.

29. Dony C., Malenfant J. and Cointe P. (1992), 'Prototype-Based Languages: From a New Taxonomy to Con-structive Proposals

and Their Validation', in ACM SIGPLAN Notices, ACM Press, Vol. 27(10), pp. 201-217.

30. Drossopoulou S., Damiani F., Ciancaglini M.D. and Giannini P. (2001), 'Fickle: Dynamic Object Re-classification', in Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP), Lecture Notes In Computer Science (LNCS), Springer-Verlag, Vol. 2072, pp. 130-149.

31. Fisher K. and Mitchell J.C. (1994), 'Notes on typed object-oriented programming', in Proceedings of the International Conference on Theoretical Aspects of Computer Software (TACS), Lecture Notes In Computer Science, Springer-Verlag, Vol. 789, pp. 844-885 .

32. Fisher K. and Mitchell J.C. (1995), 'A delegation-based object calculus with subtyping', in Proceedings of the 10th International Symposium on Fundamentals of Computation Theory (FCT), Lecture Notes In Computer Science, Springer-Verlag, Vol. 965, pp. 42-61.

33. Gamma E., Helm R., Johnson R. and Vlissides J. (1994), 'Design Patterns: Elements of Object-Oriented Software Architecture', Addison-Wesley Publishing Company, Reading, Massachusetts.

34. Ghelli G. and Orsini R. (1991), 'Types and subtypes as partial equivalence relations', in Maurizio Lenzerini, Daniele Nardi and Maria Simi, editors, Inheritance Hierarchies in Knowledge Representation and Programming Languages, Wiley, pp. 191-210.

35. Gordon D. and Noble J. (2007), 'Dynamic Ownership in a Dynamic Language', in Proceedings of symposium on Dynamic languages (DLS), ACM Press, pp. 41-52.

36. Gordon D. (2007), 'Encapsulation enforcement with dynamic ownership', Master's thesis, Victoria University of Wellington.

37. Hauck F.J. (1993a), 'Inheritance Modeled with Explicit Bindings: An Approach to Typed Inheritance', in Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications (OOPSLA), ACM Press, pp. 231-239.

38. Hauck F.J. (1993b), 'Class-Based Inheritance is Not a Basic Concept', in Friedrich-Alexander-University Erlangen-Numberg, Computer Science Department, IMMD IV, Technical Report TR-14-6-93.

39. Hogg J. (1991), 'Islands: aliasing protection in object-oriented languages', in ACM SIGPLAN Notices, ACM Press, Vol. 26(11), pp. 271-285.

40. Hogg J., Lea D., Wills A., deChampeaux D. and Holt R. (1992), 'The Geneva convention on the treatment of object aliasing', in ACM SIGPLAN OOPS Messenger, ACM Press, Vol.3(2), pp. 11-16.

41. Hutchinson N.C. (1987), 'EMERALD: An object-based language for distributed programming', PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle, WA.

42. Igarashi A., Pierce B.C. and Wadler P. (2001), 'Featherweight Java: a minimal core calculus for Java and GJ', in ACM Transactions on Programming Languages and Systems (TOPLAS), ACM Press, Vol. 23(3), pp. 396-450.

43. Katiyar D., Luckham D. and Mitchell J. (1994), 'A type system for prototyping languages', in Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), ACM Press, pp. 138-150.

44. Kent S. and Maung I. (1995), 'Encapsulation and aggregation', in Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS), Prentice Hall.

45. Kniesel G. (1998), 'Delegation for Java: API or Language Extension?', Technical Report IAI-TR-98-5.

46. Kniesel G. (1999), 'Type-Safe Delegation for Run-Time Component Adaptation', In Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP), Lecture Notes In Computer Science (LNCS), Springer-Verlag , Vol. 1628, pp. 351-366.

47. Kniesel G. (2000), 'Dynamic Object-Based Inheritance with Subtyping', Ph.D. Thesis, Universitat Bonn, Institut fur Informatik III, D-5311 Bonn.

48. Leino K.R.M. and Müller P. (2005), 'Modular verification of static class invariants', in FM 2005: Formal Methods, International Symposium of Formal Methods Europe, John Fitzgerald, Ian J. Hayes and Andrzej Tarlecki (eds.), Lecture Notes in Computer Science (LNCS), Springer, Vol. 3582, pp. 26-42.

49. Leino K.R.M. and Muller P. (2004), 'Object invariants in dynamic contexts', in Proceedings of European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science (LNCS), Springer-Verlag, Vol. 3086, pp. 491-516.

50. Lieberman H. (1986), 'Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems', in proceedings on Object-oriented programming systems, languages and applications (OOPSLA), ACM Press, pp. 214-223.

51. Litvinov V. (2003), 'Constraint-Bounded Polymorphism: an Expressive and Practical Type System for Object-Oriented Languages', Ph.D. thesis, Department of Computer Science and Engineering, University of Washington, Seattle, WA.

52. Lu Y. and Potter J. (2006), 'Protecting Representation with Effect Encapsulation', in Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), ACM Press, pp. 359-371.

53. Meyer B. (1992), 'Eiffel: the language', Prentice-Hall Object-Oriented Series.

54. Meyers S. (1996), 'More Effective C++', Addison-Wesley.

55. Minsky N.H. (1996), 'Towards Alias-Free Pointers', in Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP), Lecture Notes In Computer Science (LNCS), Springer-Verlag, Vol. 1098, pp. 189-209.

56. Muller P. and Poetzsch-Heffter A. (1999), 'Universes: A type system for controlling representation exposure', in Programming Languages and Fundamentals of Programming, Poetzsch-Heffter A. and Meyer J. (eds.), Fer-nuiversitat Hagen, Technical Report 263.

57. Muller P. and Rudich A. (2007), 'Ownership Transfer in Universe Types', in proceedings on Object-oriented programming systems, languages and applications (OOPSLA), ACM Press, pp. 461-478.

58. Nierstrasz O. (1995), 'Regular types for active objects', In Object-Oriented Software Composition, Prentice Hall, pp. 99-121.

59. Noble J., Vitek J. and Potter J. (1998), 'Flexible Alias Protection', in Proceedings of the 12th European Conference on Object-Oriented

Programming (ECOOP), Lecture Notes In Computer Science (LNCS), Vol. 1445, pp. 158-185.

60. Palsberg J. and Schwartzbach M.I. (1992), 'Three discussions on object-oriented typing', ACM OOPS Messenger, Vol. 3(2), pp. 31-38.

61. Palsberg J. and Schwartzbach M.I. (1994), 'Object-Oriented Type Systems', John Wiley.

62. Pernici B. (1989), 'Objects with roles', Technical Report, Centre Universitaire d'Informatique, University of Geneva.

63. Pierce B.C. and Turner D.N. (1994), 'Simple type-theoretic foundations for object-oriented programming', Journal of Functional Programming, Vol. 4(2), pp. 207-247.

64. Potanin A., Noble J. and Robert Biddle (2004), 'Generic Ownership: Practical Ownership Control in Programming Languages', in Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA), ACM Press, pp. 50-51.

65. Potanin A., Noble J., Clarke D.G. and Biddle R. (2006), 'Featherweight Generic Confinement', in Journal of Functional Programming, Cambridge University Press, Vol. 16(6), pp. 793-811.

66. PradeepKumar D.S. and Saswati Mukherjee (2006), 'Modal Logic and Ownership Types: Uniting Three Worlds', in Doctoral Symposium Proceedings of OOPSLA, ACM Press.

67. Richardson J. and Schwarz P. (1991), 'Aspects: Extending objects to support multiple, independent roles', In Clifford J. and King R. (eds.), Proceedings of the ACM SIGMOD International Conference on Management of Data, ACM Press, pp. 298-307.

68. Riecke J.G. and Stone C.A. (2002), 'Privacy via Subsumption', in Information and Computation, Academic Press, Inc, Vol. 172(1), pp. 2-28.

69. Scharli N., Black A.P. and Ducasse S. (2004), 'Object-oriented Encapsulation for Dynamically Typed Languages', in Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA), ACM Press, pp. 130-149.

70. Sciore E. (1989), 'Object Specialization', in ACM Transactions on Information Systems (TOIS), ACM Press, Vol. 7(2), pp. 103-122.

71. Snyder A. (1986), 'Encapsulation and Inheritance in Object-Oriented Programming Languages', in proceedings on Object-oriented programming systems, languages and applications (OOPSLA), ACM Press, pp. 38-45.

72. Stein L.A. (1987), 'Delegation Is Inheritance', in proceedings on Object-oriented programming systems, languages and applications (OOPSLA), ACM Press, pp. 138-146.

73. Stein L.A., Lieberman H. and Ungar D. (1988), 'A Shared View of Sharing: The Treaty of Orlando', in Object-oriented concepts, databases, and applications, ACM Press, pp. 31-48.

74. Stevens W., Myers G. and Constantine L. (1974), 'Structured Design', in IBM Systems Journal, Vol. 13(2), pp. 115-139.

75. Taivalsaari A. (1996), 'On the notion of inheritance', ACM Computing Surveys, Vol. 28(3), pp. 438-479.

76. Tamai T., Ubayashi N. and Ichiyama R. (2005), 'An Adaptive Object Model with Dynamic Role Binding', in Proceedings of the 27th International Conference on Software Engineering (ICSE), ACM Press, pp. 166-175.

77. Ungar D. and Smith R.B. (1987), 'Self: The power of simplicity', in Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA), ACM Press, pp. 227-242.

78. Vitek J. and Bokowski B. (1999), 'Confined Types', in Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA), ACM Press, pp. 82-96.

79. Wieringa R. and de Jonge W. (1991), 'The identification of objects and roles', Technical Report, Faculty of Mathematics and Computer Science, Vrije Universiteit, De Boelelaan 1081a, 1081 HV, Amsterdam, Netherlands.

80. Wieringa R., de Jonge W. and Spruit P. (1994), 'Roles and dynamic subclasses: A modal logic approach', In Tokoro and Pareschi [TP94], pp. 32-59.

81. Yu S. (2001), 'Class-is-type is inadequate for object reuse', in ACM SIGPLAN Notices, ACM Press, Vol. 36(6), pp. 50-59.

82. Zendra O. and Colnet D. (1999), 'Towards Safer Aliasing with the Eiffel Language', in Proceedings of the Workshop on Object-Oriented Technology, Lecture Notes In Computer Science (LNCS); Springer-Verlag, Vol. 1743, pp. 153-154.

83. Zhao T., Palsberg J. and Vitek J. (2003), 'Lightweight Confinement for Featherweight Java', in Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA), ACM Press, pp. 135-148.

LIST OF PUBLICATIONS

1. **PradeepKumar D.S.** (2008), 'Alias Count Facilitate Ownership Transfer', in Poster Session, in ACM symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), ACM Press.

2. **PradeepKumar D.S.** and Saswati Mukherjee (2007), 'Jmigrate (Jm): An object-oriented language with Object specialization and Ownership transfer', in IADIS International Conference Applied Computing, (Accepted but not published due to financial constraint).

3. **PradeepKumar D.S.** and Saswati Mukherjee (2006), 'Modal Logic and Ownership Types: Uniting Three Worlds', in Doctoral Symposium Proceedings of OOPSLA, ACM Press.

VITAE

D.S. Pradeep Kumar, was born on 10th March 1982 at Sivakasi in Tamil Nadu. He graduated in Computer Science Engineering in the year 2004, from Arignar Anna Institute of Science and Technology, Madras University, Chennai, India.

At present he is working as Research Assistant in the Project Collaboration Research Project in Smart and Secure Environment, Department of Computer Science and Engineering, CEG, Anna University, Chennai. His field of interest is Programming Languages, Security, Semantics and Logics, Language-based security, Software engineering, Dynamic objects behavior and type evaluation, Distributed objects and Mobile objects, environmental security analysis and behavioral monitoring, Natural Language Processing and Semantic analysis.

He has participated and presented one ACM Doctoral Symposium paper and one ACM Poster paper. One paper got selected in International Conference and not published due to financial situation.