Using Planning Techniques in Object-Oriented Design

Milko Marinov

Abstract: *In the object-oriented paradigm, a real-world entity is modeled as an instance and operations applicable to the objects. In this paper the major features of subtyping mechanism and the main type of hierarchies in the Object Model are presented. The application of planning techniques in object-oriented design is specified and further developed by defining a generalized algorithm of the* ***Planner*** *and* ***Executor*** *at conceptual level.*

Key words: *Object-oriented design; Class hierarchy; Inheritance; Planning techniques.*

INTRODUCTION

Object-oriented analysis and design cannot be cleanly separated. Design is architectural modeling. It adds detail, precision and implementation-dependent features to the analysis models. The original object-oriented design method was Booch's (1986) [2]. The main object-oriented and object-based design methods and notations in current use are the Booch (1991) method [2], OODLE (Object-Oriented Design LanguagE), HOOD (Hierarchical Object-Oriented Design), OOSD (Object-Oriented Structured Design) and UML (Unified Modeling Language) [2,3,4,5,8]. In all of these methods there are many types of diagrams used to present and to investigate the inheritance between classes. In the object-oriented paradigm, a real-world entity is modeled as an instance (object) and operations (methods) applicable to the objects. A class, and therefore an object, may inherit properties and methods from other classes. Thus the fundamental data modeling concepts instantiation (an object is an instance of a class) and generalization (a class inherits properties and methods from other classes) are built into the object-oriented paradigm. Further, the inheritance mechanism makes it possible for applications to define new classes and have them inherit properties from existing classes; this makes the applications easily extendible [3,4].

Object-oriented databases comprise the definition of the structure and the behaviour of objects, where behaviour is defined by activities as well as by object life-cycle, which show in which order activities can be invoked on an object. Large database schemas are often not defined by a single person but by a group of people that comprises future users of the new database. The result of first design phase is a set of view schemas, each one containing a part of the whole conceptual database schema. In the second phase, the system integrator collects the views and defines the conceptual database schema by integrating the view schemas. The integrated schema can be implemented in a database system in subsequent design steps [5,7]. Therefore, from this point of view, the investigation of the inheritance is very important in the object-oriented database design.

In this paper an attempt is made to incorporate the artificial intelligence techniques into the design of hierarchies of object-oriented databases. We propose to use planning techniques to create large, complete hierarchies without conflict inconsistencies between classes. Most existing plan-execution techniques were developed to plan the physical actions of real or simulated robots. For example, STRIPS, HACKER and NOAH were designed to plan robot construction and movement tasks, and ELMER was designed to plan the route of a simulated robot taxi [1,5]. The creating and execution of an inheritance plan poses an unusually difficult test for planning techniques. Most planners create plans which are designed to change the state of the observable, physical world, or a simulation of it. The current state in the inheritance plan is presented in the form of the Object Model contents and there are some problems to define the initial and goal state.

The paper is structured as follows. Section 2 provides some specifications of the general characteristics of the inheritance and type hierarchies in object model. Section 3 describes the plan representation of inheritance structure in object model. Section 4 summarises the author's contributions and his future research intentions.

INHERITANCE AND TYPE HIERARCHIES IN OBJECT MODEL

Instances inherit all and only the feature of the classes they belong to, but it is also possible in an object-oriented system to allow classes to inherit feature from more general superclasses. In this case inherited features can be overridden and extra features added to deal with exceptions. Inheritance describes many different mechanisms in which type definitions or implementations can be related to one another through a partial order. The basic notion is that we can modify type definitions incrementally by adding subtype definitions that somehow modify the original type. The combination of the supertype definition and the subtype modifications produces a completely defined new type. The inheritance allows for incremental modification of type definitions, thereby providing capability for such things as extension of previous definitions and reuse of code. These modifications to types are limited to those that can be made without disturbing dependencies between the preexisting types and programs that use them. The features of a subtyping mechanism are as follows [2,4]:

- ***Substitutability:*** To say that B is a subtype of A typically means that any context that is expecting an instance of type A must also accept an instance of type B. This principle is called substitutability.
- ***Static type checking:*** It means that all reasoning based on information expressed on types is checkable at compile-time. There is no need to check type compatibility at run-time. This property is clearly desirable, since it means that there is no need to insert expensive run-time checks in the resulting code.
- ***Mutability:*** It is possible to interact with an object only via its available messages or operations. An object supports a state that can be observed by some subset of its operations. The operations that report on an object's state are called *reporters*, and the set of such operations is $O_r(T)$. The operations that will alter the state of its instances are called mutators, and the set of mutators is $O_m(T)$. An operation m is a mutator for type T if for some instance x of T and for some r in O_r, it is possible to execute the following code fragment such that, at the end, a is not equal to b.

  $a = r(x);$ $m(x);$ $b = r(x);$

  If m is to be a mutator, it must be possible to observe its effect on some object. A type system incorporates mutability if it is possible to construct a type T with some operation that is a mutator.
- ***Specialization via constraints:*** Assume that $Ops(T:Type)$ is a function that returns the legal operations (i.e. messages) that are defined on the given type T. Specialization via constraints occurs whenever the operation redefinition on a subtype constrains one of the arguments to be from a smaller value set than the corresponding operation on the supertype.

The object model distinguishes three separate hierarchies:

⇒ ***Specification hierarchy*** (*compatibility of predicates*): By hierarchy we mean a partial order (e.g. a directed acyclic graph represents this). The specification hierarchy expresses consistency among type specifications in such a way as to allow for substitution of instances of a subtype in contexts that are expecting an

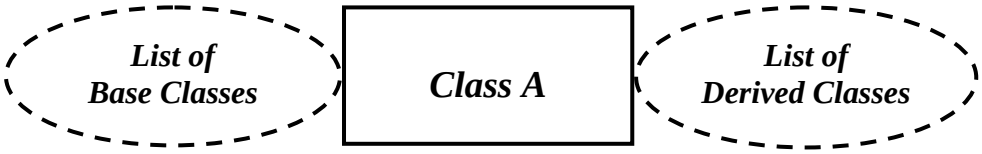
instance of a supertype. Subtypes add behavior only to their supertypes. They cannot restrict the values of the input arguments to any of the methods of the supertype. In this way, the specification hierarchy supports static type checking.

⇒ ***Implementation hierarchy*** (*code sharing*): provides for code sharing among types. It allows some of the operations on a type to be inherited (i.e. reused) and others to be redefined. The redefined methods overload the name of the operations and the proper code body is selected at run-time based on the type of a distinguished argument. Redefinition of operations allows to reuse what is similar between an old type and a new type, and to modify the behavior of that part of an old type definition that must be different. This hierarchy need not mirror the specification hierarchy.

⇒ ***Classification hierarchy*** (*subset and constraints*): describes collections of objects and the containment relationship among these collections. The collections can be defined by enumeration or by a predicate.

PLAN REPRESENTATION OF INHERITANCE STRUCTURE IN OBJECT MODEL

Inheritance is defined for classes. The concept implies that a derived class inherits data and operations from a base class. The derived class may itself be a base class for another layer of inheritance. The system of classes that use inheritance forms a class hierarchy.



A derived class is often referred to as subclass with a corresponding base class described as a super class.

Developing a plan is defined as finding a sequence of actions to accomplish a specified goal. A planning problem must first have a vocabulary of symbols and notion in which the initial state, goal conditions and operators may be specified. The elementary objects of a class hierarchy are the classes. Each class corresponds to some unit of the domain. The goal of the planning problem will be specified as an expression consisting of logical connectives (AND) and instances of classes.

The ***Planner*** constructs the plan which in itself is a sequence of steps aimed at achieving the global goal. In the domain of class inheritance, the global goal can be defined as the list of all leaf nodes (the classes without derived classes). Each step is connected with corresponding actions. In our case the meaning of 'step' is to be connected with a corresponding class for which the preconditions (base classes) and the expected results (derived classes) have been formed. The plan is developed by the ***Planner*** through simulation of the derived classes of the current class over the Object Model. Starting with the current state of the Object Model, the ***Planner*** is trying to find such a sequence of classes through which the global goal will be achieved. By using the base classes (preconditions) and the derived classes (expected results) of the classes, the ***Planner*** can implement different branches in the plan. A successful plan is the one which can be applied to the current state of the Object Model and can achieve the global goal. The plan is a directed acyclic graph consisting of a set of nodes and a set of arcs. An arc from node ***A*** to node ***B*** means that the fulfillment of ***A*** has to precede the fulfillment of ***B***. The nodes of the graph are two types: ***classes*** and ***goals***. The goal node which precedes a particular

class node corresponds to the base classes (preconditions). The goal node which follows a particular class node corresponds to the derived classes (expected results) which are the base classes for the following class node.

Let's discuss in greater detail the summarized algorithm of the ***Planner***. The algorithm uses the following local data structures: ***CurClass*** – the current class from the class hierarchy; ***BaseList(CurClass)*** – a list of the ***CurClass*** base classes; ***BaseClasses*** – a temporal list structure; ***InheritTable*** – a table with the following structure:

InheritTable:

ClassName	BaseClasses	DerivedClasses
[*String*]	[*List*]	[*List*]

In the initial state the ***DerivedClasses*** list is empty, i.e. for each class only the base classes are defined.

Planner *algorithm*

```
Let BaseClasses be empty;
Let BaseList(CurClass) be empty;
Select CurClass from the set of classes, incorporated into the Object Model (OM);
        CurClass has to be with at least one base class;
Copy InheritTable.BaseClasses[CurClass] to BaseList(CurClass);
Copy BaseList(CurClass) to BaseClasses;
While BaseClasses is not empty Do
{
        While BaseList(CurClass) is not empty Do
        {
                Add CurClass to the InheritTable.DerivedClasses[x]
                                        of the correspondent base class X;
                Delete the base class from BaseList(CurClass);
        }
        Set a class from BaseClasses to CurClass;
        Delete CurClass from BaseClasses;
        An attempt is made by continuing the planning in depth;
        Copy InheritTable.BaseClasses[CurClass] to BaseList(CurClass);
        Copy BaseList(CurClass) to BaseClasses;
}
```

The ***Executor*** uses a plan created by the ***Planner***. If serious problems arise during the execution of the initial plan, the ***Executor*** may reinvoke the ***Planner*** to revise the plan. If the plan fails then there are integrity constraints in the class inheritance. The ***Executor*** algorithm is based on a procedure for producing a linear or total ordering of the nodes of a directed graph whose arcs represent a partial ordering relation of the nodes. The ***Executor*** algorithm uses the following local data structures: ***GoalClasses*** - the global goal is defined

as a list of all classes with base classes, but without derived classes in the Object Model; ***InitList*** - a list of classes which defines the initial state; ***DeriveList(CurClass)*** – a list of the ***CurClass*** derived classes.

Executor *algorithm*

```
While GoalClasses is not empty Do
{
        Planning is done following the algorithm described above, depending on
        the initial state of the Object Model or the state at which the plan has failed
        having in mind the global goal. A new plan P is generated;

        For each class X of P
                If InheritTable.BaseClasses[x] is empty
                        Then add X to InitList;
        While InitList is not empty Do
        {
                If GoalClasses is empty, TERMINATE;
                Select and remove a class (CurClass) from InitList;
                Copy InheritTable.DerivedClasses[CurClass] to
                                                        DeriveList(CurClass);
                While DeriveList(CurClass) is not empty Do
                {
                    Select and remove a class (CurClass) from DeriveList(CurClass);
                    If  CurClass is in GoalClasses Then
                         remove CurClass from GoalClasses;
                }
        }
}
```

The above described generalized algorithms of the ***Planner*** and ***Executor*** do not include all possible branches in them. The author provides this part of the algorithms which might be of interest to a wider range of readers.

CONCLUSIONS AND FUTURE WORK

Inheritance is the ability to deal with generalization and specialization or classification. Subclasses inherit attributes and methods from their superclasses and may add others of their own or override those inherited. Multiple inheritance occurs when an object is an instance from more than one class. Inheritance delivers extensibility, but can compromise reusability. Multiple inheritance is powerful but introduces additional problems concerning object model integrity.

The main advantage of using the planning techniques to investigate the inheritance between classes in the object model is that the class hierarchies is guided by the global structure of the plan which is produced automatically by the ***Planner*** based partly on the information about the object model.

The author's further efforts will be aimed at building the ***Planner*** and ***Executor*** algorithms into a knowledge-based CASE tool for object-oriented design to support the meta-knowledge about the object model.

REFERENCES

[1] Barbehenn, M., S. Hutchinson. Efficient search and hierarchical motion planning by dynamically maintaining single-source shortest paths trees, *IEEE Transactions on Robotics and Automation*, Vol. 11, No 2, pp. 198-214 (1995).

[2] Booch G., Object-oriented design with applications, *Redwood City, CA: Benjamin Cummings*, (1991).

[3] Booch G., J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, *Addison-Wesley* (1998).

[4] Graham I., Object-oriented methods, *Addison-Wesley*, pp. 193-224 (1993).

[5] Gorla N., An object-oriented database design for improved performance, *Data & Knowledge Engineering*, Vol. 37 (2), pp. 117-138 (2001).

[6] Marinov M., Analysis of planning techniques used in intelligent systems, *Automatica & Informatics*, Sofia, Vol. 5-6, pp. 25-33 (1997).

[7] Preuner G., S. Conrad, M. Schrefl, View integration of behavior in object-oriented design, *Data & Knowledge Engineering*, Vol. 36 (2), pp. 153-183 (2001).

[8] Lee R., W. Tepfenhart, UML and C++: A Practical Guide to Object-Oriented Development, Prentice-Hall (2001).

ABOUT THE AUTHOR

Assist. Prof. Milko Marinov, PhD, Department of Computer Systems & Technologies, University of Rousse, Phone: (+359 82) 888 356, E-mail: MMarinov@ecs.ru.acad.bg.