面向对象笔记

Notes of Object Orient

theqiong.com

谨献给…。

λ

目　录

Preface iii

0.1 Objects iii

0.1.1 Specialized objects iv

0.1.2 Distributed objects iv

Overview vii

0.2 History viii

0.3 Fundamention x

0.3.1 Decoupling x

0.3.2 Encapsulation xi

0.3.3 Inheritance xi

0.3.4 Polymorphism xi

0.4 Formal Semantics xi

0.5 Languages xii

0.6 Design Patterns xiii

I Introduction 1

第 1 章 Overview 3

1.1 Primitive Type 4

II Design Pattern 5

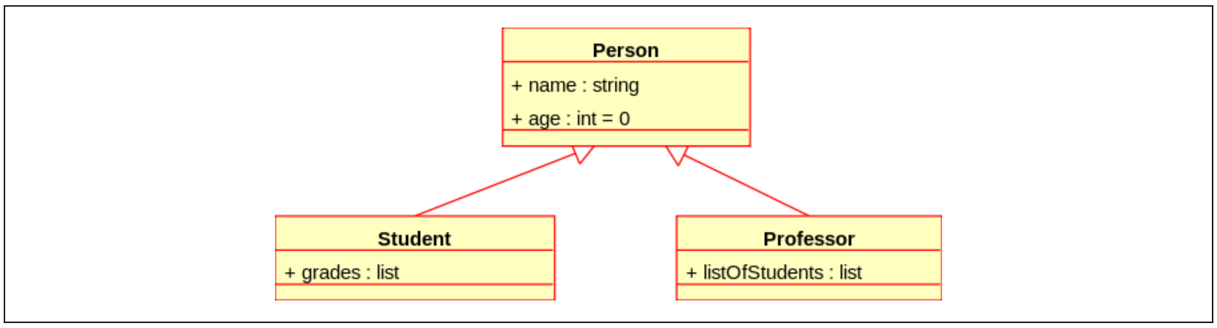
第 2 章 Introduction 7

索　引 5

Preface

0.1 Objects

Object-oriented or object-orientation is a software engineering concept, in which concepts are represented as "objects"[1].



In computer science, an object is a location in memory having a value and referenced by an identifier. An object can be a variable, function, or data structure. In the object-oriented programming paradigm,"object," refers to a particular instance of a class where the object can be a combination of variables, functions, and data structures. In relational Database management an object can be a table or column, or an association between data and a database entity (such as relating a person's age to a specific person).

An important distinction in programming languages is the difference between an object-oriented language and an object-based language. A language is usually considered object-based if it includes the basic capabilities for an object: identity, properties, and attributes. A language is considered object-oriented if it is object-based and also has the capability of polymorphism and inheritance. Polymorphism refers to the ability to overload the name of a function with multiple behaviors based on which object(s) are passed to it. Conventional message passing discriminates only on the first object and considers that to be "sending a message" to that object. However, some OOP languages such as Flavors and the Common Lisp Object System (CLOS) enable discriminating on more than the first parameter of the function. Inheritance is the ability to subclass an object class, to create a new class that is a subclass of an existing one and inherits all the data constraints and behaviors of its parents but also changes one or more of them.

Object-Oriented programming is an approach to designing modular reusable software systems. The object-oriented approach is fundamentally a modelling approach.[5] The object-oriented approach is an evolution of good design practices that go back to the very beginning of computer programming. Object-orientation is simply the logical extension of older techniques such as structured programming and abstract data types. An object is an abstract data type with the addition of polymorphism and inheritance.

Rather than structure programs as code and data an object-oriented system integrates the two using the concept of an "object". An object has state (data) and behavior (code). Objects correspond to things found in the real world. So for example, a graphics program will have objects such as circle, square, menu. An online shopping system will have objects such as shopping cart, customer, product,. The shopping system will support behaviors such as place order, make payment, and offer discount. The objects are designed as class hierarchies. So for example with the shopping system there might be high level classes such as electronics product, kitchen product, and book. There may be further refinements for example under electronic products: CD Player, DVD player, etc. These classes and subclasses correspond to sets and subsets in mathematical logic.

0.1.1 Specialized objects

An important concept for objects is the design pattern. A design pattern provides a reusable template to address a common problem. The following object descriptions are examples of some of the most common design patterns for objects.

- Function object: an object with a single method (in C++, this method would be the function operator, ”operator()”) that acts much like a function (like a C/C++ pointer to a function).
- Immutable object: an object set up with a fixed state at creation time and which does not change afterward.
- First-class object: an object that can be used without restriction.
- Container: an object that can contain other objects.
- Factory object: an object whose purpose is to create other objects.
- Metaobject: an object from which other objects can be created (Compare with class, which is not necessarily an object)
- Prototype: a specialized metaobject from which other objects can be created by copying
- God object: an object that knows too much or does too much. The God object is an example of an anti-pattern.
- Singleton object: An object that is the only instance of its class during the lifetime of the program.
- Filter object

0.1.2 Distributed objects

The object-oriented approach is not just a programming model. It can be used equally well as an interface definition language for distributed systems. The objects in a distributed computing model tend to be larger grained, longer lasting, and more service oriented than programming objects.

A standard method to package distributed objects is via an Interface Definition Language (IDL). An IDL shields the client of all of the details of the distributed server object. Details such as which computer the object resides on, what programming language it uses, what operating system, and other platform specific issues. The IDL is also usually part of a distributed environment that provides services such as transactions and persistence to all objects in a uniform manner. Two of the most popular standards for distributed objects are the Object Management Group’s CORBA standard and Microsoft’s DCOM.

In addition to distributed objects, a number of other extensions to the basic concept of an object have been proposed to enable distributed computing:

- Protocol objects are components of a protocol stack that enclose network communication within an object-oriented interface.
- Replicated objects are groups of distributed objects (called replicas) that run a distributed multi-party protocol to achieve high consistency between their internal states, and that respond to requests in a coordinated way. Examples include fault-tolerant CORBA objects.
- Live distributed objects (or simply live objects) generalize the replicated object concept to groups of replicas that might internally use any distributed protocol, perhaps resulting in only a weak consistency between their local states.

Some of these extensions, such as distributed objects and protocol objects, are domain-specific terms for special types of “ordinary” objects used in a certain context (such as remote invocation or protocol composition). Others, such as replicated objects and live distributed objects, are more non-standard, in that they abandon the usual case that an object resides in a single location at a time, and apply the concept to groups of entities (replicas) that might span across multiple locations, might have only weakly consistent state, and whose membership might dynamically change.

The Semantic Web is essentially a distributed objects framework. Two key technologies in the Semantic Web are the Web Ontology Language (OWL) and the Resource Description Framework (RDF). RDF provides the capability to define basic objects: names, properties, attributes, relations, that are accessible via the Internet. OWL adds a richer object model, based on set theory, that provides additional modeling capabilities such as multiple inheritance.

OWL objects are not like standard large grained distributed objects accessed via an Interface Definition Language. Such an approach would not be appropriate for the Internet because the Internet is constantly evolving and standardization on one set of interfaces is difficult to achieve. OWL objects tend to be similar to the kind of objects used to define application domain models in programming languages such as Java and C++.

However, there are important distinctions between OWL objects and traditional object-oriented programming objects. Where as traditional objects get compiled into static hierarchies usually with single inheritance, OWL objects are dynamic. An OWL object can change its structure at run time and can become an instance of new or different classes.

Another critical difference is the way the model treats information that is currently not in the system. Programming objects and most database systems use the "closed world assumption". If a fact is not known to the system that fact is assumed to be false. Semantic Web objects use the open world assumption, a statement is only considered false if there is actual relevant information that it is false, otherwise it is assumed to be unknown, neither true nor false.

OWL objects are actually most like objects in artificial intelligence frame languages such as KL-ONE and Loom.

The following table contrasts traditional objects from Object-Oriented programming languages such as Java or C++ with with Semantic Web Objects:

OOP Objects	Semantic Web Objects
Classes are regarded as types for instances.	Classes are regarded as sets of individuals.
Instances can not change their type at runtime.	Class membership may change at runtime.
The list of classes is fully known at compile-time and cannot change after that.	Classes can be created and changed at runtime.
Compilers are used at build-time. Compile-time errors indicate problems.	Reasoners can be used for classification and consistency checkingat runtime or build-time.
Classes encode much of their meaning and behavior through imperative functions and methods.	Classes make their meaning explicit in terms of OWL statements. No imperative code can be attached.
Instances are anonymous insofar that they cannot easily be addressed from outside of an executing program.	All named RDF and OWL resources have a unique URI under which they can be referenced.
Closed world: If there is not enough information to prove a statement true, then it is assumed to be false.	Open world: If there is not enough information to prove a statement true, then it may be true or false.

对象（object）是面向对象（Object Oriented）中的术语，既表示客观世界问题空间（Namespace）中的某个具体的事物，又表示软件系统解空间中的基本元素。在软件系统中，对象具有唯一的标识符，对象包括属性（Properties）和方法（Methods），属性就是需要记忆的信息，方法就是对象能够提供的服务。在面向对象（Object Oriented）的软件中，对象（Object）是某一个类（Class）的实例（Instance）。

面向对象程序设计是一种程序设计范型，同时也是一种程序开发的方法。对象指的是类的实例。它将对象作为程序的基本单元，将程序和数据封装其中，以提高软件的重用性、灵活性和扩展性。

面向对象程序设计可以看作一种在程序中包含各种独立而又互相调用的对象的思想，这与传统的思想刚好相反：传统的程序设计主张将程序看作一系列函数的集合，或者直接就是一系列对电脑下

达的指令。面向对象程序设计中的每一个对象都应该能够接受数据、处理数据并将数据传达给其它对象，因此它们都可以被看作一个小型的“机器”，即对象。

目前已经被证实的是，面向对象程序设计推广了程序的灵活性和可维护性，并且在大型项目设计中广为应用。此外，支持者声称面向对象程序设计要比以往的做法更加便于学习，因为它能够让人们更简单地设计并维护程序，使得程序更加便于分析、设计、理解。反对者在某些领域对此予以否认。

当我们提到面向对象的时候，它不仅指一种程序设计方法。它更多意义上是一种程序开发方式。在这一方面，我们必须了解更多关于面向对象系统分析和面向对象设计（Object Oriented Design，简称OOD）方面的知识。

Overview

Object-oriented programming (OOP) is a programming paradigm that represents concepts as “objects” that have data fields (attributes that describe the object) and associated procedures known as methods. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs. C++, Objective-C, Smalltalk, Java, C#, Perl, Python, Ruby and PHP are examples of object-oriented programming languages.

Object-oriented programming is an approach to designing modular, reusable software systems. Although discussions of object-oriented technology often get mired in the details of one language vs. the other, the real key to the object-oriented approach is that it is a modelling approach first. Although often hyped as a revolutionary way to develop software by zealous proponents, the object-oriented approach is in reality a logical extension of good design practices that go back to the very beginning of computer programming. Object-orientation is simply the logical extension of older techniques such as structured programming and abstract data types. An object is an abstract data type with the addition of polymorphism and inheritance.

Rather than structure programs as code and data, an object-oriented system integrates the two using the concept of an ”object”. An object has state (data) and behavior (code). Objects correspond to things found in the real world. So for example, a graphics program will have objects such as circle, square, menu. An online shopping system will have objects such as shopping cart, customer, product. The shopping system will support behaviors such as place order, make payment, and offer discount. The objects are designed as class hierarchies. So, for example, with the shopping system there might be high level classes such as electronics product, kitchen product, and book. There may be further refinements for example under electronic products: CD Player, DVD player, etc. These classes and subclasses correspond to sets and subsets in mathematical logic.

The goals of object-oriented programming are:

- Increased understanding.
- Ease of maintenance.
- Ease of evolution.

The overall understanding of the system is increased because the semantic gap—the distance between the language spoken by developers and that spoken by users—is lessened. Rather than talking about database tables and programming subroutines, the developer talks about things the user is familiar with: objects from their application domain.

Object orientation eases maintenance by the use of encapsulation and information hiding. One of the most common sources of errors in programs is when one part of the system accidentally interferes with another part. For example, in the very earliest days of programming, it was common for developers to use “go to” statements to jump to arbitrary locations within only a few routines and functions. Critics called this “spaghetti code” because it is disorganized. Structured programming addresses this by encouraging the use of procedures and subroutines. Appropriate usage sections off responsibility for individual blocks to implement separate functionality. So, for example, one would know that the square root function was separate from the launch missiles function, and a change to one could not affect the other.

Object-orientation takes this to the next step. It essentially merges abstract data types with structured programming and divides systems into modular objects which own their own data and are responsible for their own behavior. This feature is known as encapsulation. With encapsulation, not only can the “square root” and ”launch missiles” functions not interfere with each other, but also the data for the two are divided up so that changes to one object cannot affect the other. Note that all this relies on the various languages being used appropriately, which, of course, is never certain. Object-orientation is not a software silver bullet, and it is not magic that makes all development problems go away.[6]

In addition to providing ease of maintenance, encapsulation and information hiding provide ease of evolution as well. Defining software as modular components that support inheritance makes it easy both to re-use existing components and to extend components as needed by defining new subclasses with specialized behaviors. This goal of being easy to both maintain and reuse is known in the object-oriented paradigm as the ”open closed principle”. A module is open if it supports extension (e.g. can easily modify behavior, add new properties, provide default values, etc.). A module is closed if it has a well defined stable interface that all other modules must use and that limits the interaction and potential errors that can be introduced into one module by changes in another.

The object-oriented approach encourages the programmer to place data where it is not directly accessible by the rest of the system. Instead, the data is accessed by calling specially written functions, called methods, which are bundled with the data. These act as the intermediaries for retrieving or modifying the data they control. The programming construct that combines data with a set of methods for accessing and managing those data is called an object. The practice of using subroutines to examine or modify certain kinds of data was also used in non-OOP modular programming, well before the widespread use of object-oriented programming.

An object-oriented program usually contains different types of objects, each corresponding to a real-world object or concept such as a bank account, a hockey player, or a bulldozer. A program might contain multiple copies of each type of object, one for each of the real-world objects the program deals with. For instance, there could be one bank account object for each real-world account at a particular bank. Each copy of the bank account object would be alike in the methods it offers for manipulating or reading its data, but the data inside each object would differ, reflecting the different history of each account.

Objects can be thought of as encapsulating their data within a set of functions designed to ensure that the data are used appropriately, and to assist in that use. The object’s methods typically include checks and safeguards specific to the data types the object contains. An object can also offer simple-to-use, standardized methods for performing particular operations on its data, while concealing the specifics of how those tasks are accomplished. In this way, alterations can be made to the internal structure or methods of an object without requiring that the rest of the program be modified. This approach can also be used to offer standardized methods across different types of objects. As an example, several different types of objects might offer print methods. Each type of object might implement that print method in a different way, reflecting the different kinds of data each contains, but all the different print methods might be called in the same standardized manner from elsewhere in the program. These features become especially useful when more than one programmer is contributing code to a project or when the goal is to reuse code between projects.

0.2 History

Terminology invoking ”objects” and ”oriented” in the modern sense of object-oriented programming made its first appearance at MIT in the late 1950s and early 1960s. In the environment of the artificial intelligence group, as early as 1960, ”object” could refer to identified items (LISP atoms) with properties (attributes);[8][9] Alan Kay was later to cite a detailed understanding of LISP internals as a strong influence on his thinking in 1966.[10] Another early MIT example was Sketchpad created by Ivan Sutherland in 1960–61; in the glossary of the 1963 technical report based on his dissertation about Sketchpad, Sutherland defined notions of ”object” and ”instance” (with the class concept covered by ”master” or ”definition”), albeit specialized to graphical interaction.[11] Also, an MIT ALGOL version, AED-0, linked data structures (”plexes”, in that dialect) directly with procedures, prefiguring what were later termed ”messages”, ”methods” and ”member functions”.

The formal programming concept of objects was introduced in the 1960s in Simula 67, a major revision of Simula I, a programming language designed for discrete event simulation, created by Ole-Johan Dahl and Kristen

Nygaard of the Norwegian Computing Center in Oslo.[14] Simula 67 was influenced by SIMSCRIPT and C.A.R. ”Tony” Hoare’s proposed ”record classes”.[12][15] Simula introduced the notion of classes and instances or objects (as well as subclasses, virtual methods, coroutines, and discrete event simulation) as part of an explicit programming paradigm. The language also used automatic garbage collection that had been invented earlier for the functional programming language Lisp. Simula was used for physical modeling, such as models to study and improve the movement of ships and their content through cargo ports. The ideas of Simula 67 influenced many later languages, including Smalltalk, derivatives of LISP (CLOS), Object Pascal, and C++.

The Smalltalk language, which was developed at Xerox PARC (by Alan Kay and others) in the 1970s, introduced the term object-oriented programming to represent the pervasive use of objects and messages as the basis for computation. Smalltalk creators were influenced by the ideas introduced in Simula 67, but Smalltalk was designed to be a fully dynamic system in which classes could be created and modified dynamically rather than statically as in Simula 67.[16] Smalltalk and with it OOP were introduced to a wider audience by the August 1981 issue of Byte Magazine.

In the 1970s, Kay’s Smalltalk work had influenced the Lisp community to incorporate object-based techniques that were introduced to developers via the Lisp machine. Experimentation with various extensions to Lisp (such as LOOPS and Flavors introducing multiple inheritance and mixins) eventually led to the Common Lisp Object System, which integrates functional programming and object-oriented programming and allows extension via a Meta-object protocol. In the 1980s, there were a few attempts to design processor architectures that included hardware support for objects in memory but these were not successful. Examples include the Intel iAPX 432 and the Linn Smart Rekursiv.

In 1985, Bertrand Meyer produced the first design of the Eiffel language. Focused on software quality, Eiffel is among the purely object-oriented languages, but differs in the sense that the language itself is not only a programming language, but a notation supporting the entire software lifecycle. Meyer described the Eiffel software development method, based on a small number of key ideas from software engineering and computer science, in Object-Oriented Software Construction. Essential to the quality focus of Eiffel is Meyer’s reliability mechanism, Design by Contract, which is an integral part of both the method and language.

Object-oriented programming developed as the dominant programming methodology in the early and mid 1990s when programming languages supporting the techniques became widely available. These included Visual FoxPro 3.0,[17][18][19] C++[citation needed], and Delphi[citation needed]. Its dominance was further enhanced by the rising popularity of graphical user interfaces, which rely heavily upon object-oriented programming techniques. An example of a closely related dynamic GUI library and OOP language can be found in the Cocoa frameworks on Mac OS X, written in Objective-C, an object-oriented, dynamic messaging extension to C based on Smalltalk. OOP toolkits also enhanced the popularity of event-driven programming (although this concept is not limited to OOP). Some[who?] feel that association with GUIs (real or perceived) was what propelled OOP into the programming mainstream.

At ETH Zürich, Niklaus Wirth and his colleagues had also been investigating such topics as data abstraction and modular programming (although this had been in common use in the 1960s or earlier). Modula-2 (1978) included both, and their succeeding design, Oberon, included a distinctive approach to object orientation, classes, and such. The approach is unlike[how?] Smalltalk, and very unlike C++.

Object-oriented features have been added to many previously existing languages, including Ada, BASIC, Fortran, Pascal, and COBOL. Adding these features to languages that were not initially designed for them often led to problems with compatibility and maintainability of code.

More recently, a number of languages have emerged that are primarily object-oriented, but that are also compatible with procedural methodology. Two such languages are Python and Ruby. Probably the most commercially-important recent object-oriented languages are Visual Basic.NET (VB.NET) and C#, both designed for Microsoft’s .NET platform, and Java, developed by Sun Microsystems. Each of these two frameworks shows, in its own way, the benefit of using OOP by creating an abstraction from implementation. VB.NET and C# support cross-language inheritance, allowing classes defined in one language to subclass classes defined in the other language. Developers usually compile Java to bytecode, allowing Java to run on any operating system for which a Java virtual machine is available. VB.NET and C# make use of the Strategy pattern to accomplish cross-language inheritance, whereas Java makes use of the Adapter pattern.

Just as procedural programming led to refinements of techniques such as structured programming, modern object-oriented software design methods include refinements such as the use of design patterns, design by contract, and modeling languages (such as UML).

0.3 Fundamention

A survey by Deborah J. Armstrong of nearly 40 years of computing literature identified a number of fundamental concepts, found in the large majority of definitions of OOP.

Not all of these concepts appear in all object-oriented programming languages. For example, object-oriented programming that uses classes is sometimes called class-based programming, while prototype-based programming does not typically use classes. As a result, a significantly different yet analogous terminology is used to define the concepts of object and instance.

Benjamin C. Pierce and some other researchers view any attempt to distill OOP to a minimal set of features as futile. He nonetheless identifies fundamental features that support the OOP programming style in most object-oriented languages:

- Dynamic dispatch – when a method is invoked on an object, the object itself determines what code gets executed by looking up the method at run time in a table associated with the object. This feature distinguishes an object from an abstract data type (or module), which has a fixed (static) implementation of the operations for all instances. It is a programming methodology that gives modular component development while at the same time being very efficient.
- Encapsulation (or multi-methods, in which case the state is kept separate)
- Subtype polymorphism
- Object inheritance (or delegation)
- Open recursion – a special variable (syntactically it may be a keyword), usually called this or self, that allows a method body to invoke another method body of the same object. This variable is late-bound; it allows a method defined in one class to invoke another method that is defined later, in some subclass thereof.

Similarly, in his 2003 book, Concepts in programming languages, John C. Mitchell identifies four main features: dynamic dispatch, abstraction, subtype polymorphism, and inheritance.[22] Michael Lee Scott in Programming Language Pragmatics considers only encapsulation, inheritance and dynamic dispatch.

Additional concepts used in object-oriented programming include:

- Classes of objects
- Instances of classes
- Methods which act on the attached objects.
- Message passing
- Abstraction

0.3.1 Decoupling

Decoupling refers to careful controls that separate code modules from particular use cases, which increases code re-usability. A common use of decoupling is to polymorphically decouple the encapsulation (see bridge pattern and adapter pattern) – for example, using a method interface that an encapsulated object must satisfy, as opposed to using the object's class.

0.3.2 Encapsulation

Encapsulation refers to the creation of self-contained modules that bind processing functions to the data. These user-defined data types are called "classes," and one instance of a class is an "object." For example, in a payroll system, a class could be Manager, and Pat and Jan could be two instances (two objects) of the Manager class. Encapsulation ensures good code modularity, which keeps routines separate and less prone to conflict with each other.

0.3.3 Inheritance

Classes are created in hierarchies, and inheritance lets the structure and methods in one class pass down the hierarchy. That means less programming is required when adding functions to complex systems. If a step is added at the bottom of a hierarchy, only the processing and data associated with that unique step must be added. Everything else above that step is inherited. The ability to reuse existing objects is considered a major advantage of object technology.

0.3.4 Polymorphism

Object-oriented programming lets programmers create procedures for objects whose exact type is not known until runtime. For example, a screen cursor may change its shape from an arrow to a line depending on the program mode. The routine to move the cursor on screen in response to mouse movement can be written for "cursor," and polymorphism lets that cursor take simulating system behaviour.

OOP can be used to associate real-world objects and processes with digital counterparts. However, not everyone agrees that OOP facilitates direct real-world mapping (see Criticism section) or that real-world mapping is even a worthy goal; Bertrand Meyer argues in Object-Oriented Software Construction[28] that a program is not a model of the world but a model of some part of the world; "Reality is a cousin twice removed". At the same time, some principal limitations of OOP had been noted. For example, the Circle-ellipse problem is difficult to handle using OOP's concept of inheritance.

However, Niklaus Wirth (who popularized the adage now known as Wirth's law: "Software is getting slower more rapidly than hardware becomes faster") said of OOP in his paper, "Good Ideas through the Looking Glass", "This paradigm closely reflects the structure of systems 'in the real world', and it is therefore well suited to model complex systems with complex behaviours" (contrast KISS principle).

Steve Yegge and others noted that natural languages lack the OOP approach of strictly prioritizing things (objects/nouns) before actions (methods/verbs).[30] This problem may cause OOP to suffer more convoluted solutions than procedural programming.

0.4 Formal Semantics

Objects are the run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data, or any item that the program has to handle.

There have been several attempts at formalizing the concepts used in object-oriented programming. The following concepts and constructs have been used as interpretations of OOP concepts:

- coalgebraic data types;
- abstract data types (which have existential types) allow the definition of modules but these do not support dynamic dispatch

- recursive types;
- encapsulated state;
- inheritance;
- records are basis for understanding objects if function literals can be stored in fields (like in functional programming languages), but the actual calculi need be considerably more complex to incorporate essential features of OOP. Several extensions of System F<: that deal with mutable objects have been studied;[25] these allow both subtype polymorphism and parametric polymorphism (generics)

Attempts to find a consensus definition or theory behind objects have not proven very successful, and often diverge widely. For example, some definitions focus on mental activities, and some on program structuring. One of the simpler definitions is that OOP is the act of using "map" data structures or arrays that can contain functions and pointers to other maps, all with some syntactic and scoping sugar on top. Inheritance can be performed by cloning the maps (sometimes called "prototyping").

OOP was developed to increase the reusability and maintainability of source code. Transparent representation of the control flow had no priority and was meant to be handled by a compiler. With the increasing relevance of parallel hardware and multithreaded coding, developing transparent control flow becomes more important, something hard to achieve with OOP.

0.5 Languages

Simula (1967) is generally accepted as the first language with the primary features of an object-oriented language. It was created for making simulation programs, in which what came to be called objects were the most important information representation. Smalltalk (1972 to 1980) is arguably the canonical example, and the one with which much of the theory of object-oriented programming was developed. Concerning the degree of object orientation, the following distinctions can be made:

- Languages called "pure" OO languages, because everything in them is treated consistently as an object, from primitives such as characters and punctuation, all the way up to whole classes, prototypes, blocks, modules, etc. They were designed specifically to facilitate, even enforce, OO methods. Examples: Eiffel, Emerald, JADE, Obix, Ruby, Scala, Smalltalk, Self.
- Languages designed mainly for OO programming, but with some procedural elements. Examples: Delphi/Object Pascal, C++, Java, C#, VB.NET, Python.
- Languages that are historically procedural languages, but have been extended with some OO features. Examples: Pascal, Visual Basic, MATLAB, Fortran, Perl, COBOL 2002, PHP, ABAP, Ada 95.
- Languages with most of the features of objects (classes, methods, inheritance, reusability), but in a distinctly original form. Examples: Oberon (Oberon-1 or Oberon-2).
- Languages with abstract data type support, but not all features of object-orientation, sometimes called object-based languages. Examples: Modula-2, Pliant, CLU.

In recent years, object-oriented programming has become especially popular in dynamic programming languages. Python, Ruby and Groovy are dynamic languages built on OOP principles, while Perl and PHP have been adding object-oriented features since Perl 5 and PHP 4, and ColdFusion since version 6.

The Document Object Model of HTML, XHTML, and XML documents on the Internet has bindings to the popular JavaScript/ECMAScript language. JavaScript is perhaps the best known prototype-based programming language, which employs cloning from prototypes rather than inheriting from a class (contrast to class-based programming). Another scripting language that takes this approach is Lua. Before ActionScript 2.0 (a partial superset of the ECMA-262 R3, otherwise known as ECMAScript) only a prototype-based object model was supported.

0.6 Design Patterns

Challenges of object-oriented design are addressed by several methodologies. Most common is known as the design patterns codified by Gamma et al.. More broadly, the term ”design patterns” can be used to refer to any general, repeatable solution to a commonly occurring problem in software design. Some of these commonly occurring problems have implications and solutions particular to object-oriented development.

It is intuitive to assume that inheritance creates a semantic “`is a`” relationship, and thus to infer that objects instantiated from subclasses can always be safely used instead of those instantiated from the superclass. This intuition is unfortunately false in most OOP languages, in particular in all those that allow mutable objects. Subtype polymorphism as enforced by the type checker in OOP languages (with mutable objects) cannot guarantee behavioral subtyping in any context. Behavioral subtyping is undecidable in general, so it cannot be implemented by a program (compiler). Class or object hierarchies must be carefully designed, considering possible incorrect uses that cannot be detected syntactically. This issue is known as the Liskov substitution principle.

Design Patterns: Elements of Reusable Object-Oriented Software is an influential book published in 1995 by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, often referred to humorously as the ”Gang of Four”. Along with exploring the capabilities and pitfalls of object-oriented programming, it describes 23 common programming problems and patterns for solving them. As of April 2007, the book was in its 36th printing.

The book describes the following patterns:

- Creational patterns (5): Factory method pattern, Abstract factory pattern, Singleton pattern, Builder pattern, Prototype pattern
- Structural patterns (7): Adapter pattern, Bridge pattern, Composite pattern, Decorator pattern, Facade pattern, Flyweight pattern, Proxy pattern
- Behavioral patterns (11): Chain-of-responsibility pattern, Command pattern, Interpreter pattern, Iterator pattern, Mediator pattern, Memento pattern, Observer pattern, State pattern, Strategy pattern, Template method pattern, Visitor pattern

Responsibility-driven design defines classes in terms of a contract, that is, a class should be defined around a responsibility and the information that it shares. This is contrasted by Wirfs-Brock and Wilkerson with data-driven design, where classes are defined around the data-structures that must be held. The authors hold that responsibility-driven design is preferable.

Both object-oriented programming and relational database management systems (RDBMSs) are extremely common in software today. Since relational databases don’t store objects directly (though some RDBMSs have object-oriented features to approximate this), there is a general need to bridge the two worlds. The problem of bridging object-oriented programming accesses and data patterns with relational databases is known as object-relational impedance mismatch. There are a number of approaches to cope with this problem, but no general solution without downsides. One of the most common approaches is object-relational mapping, as found in libraries like Java Data Objects and Ruby on Rails’ ActiveRecord.

There are also object databases that can be used to replace RDBMSs, but these have not been as technically and commercially successful as RDBMSs.

面向对象程序设计是一种程序设计范型,同时也是一种程序开发的方法,这里对象指的是类的实例。

通常,OOP 被理解为一种将程序分解为封装数据及相关操作的模块而进行的编程方式。有别于其它编程方式,OOP 中的与某一数据类型相关的一系列操作都被封装到该数据类型当中,而非散放于其外,因而 OOP 中的数据类型不仅有着状态,还有着相关的行为。

OOP 思想被广泛认为是非常有用的,以致一套新的编程范型被创造了出来,而其它的编程范型例如函数式编程或过程式编程专注于程序运行的过程,而逻辑编程专注于引发程序代码执行的断言。

一项由 Deborah J. Armstrong 进行的长达 40 年之久的计算机著作调查显示出了一系列面向对象程序设计的基本理论。

对面向模拟系统的语言(例如 SIMULA 67)的研究及对高可靠性系统架构(比如高性能操作系统和 CPU 的架构)的研究最终导致了 OOP 的诞生。

面向对象程序设计的雏形早在 1960 年的 Simula 语言中便已出现,当时的程序设计领域正面临着一种危机:在软硬件环境逐渐复杂的情况下,软件如何得到良好的维护?面向对象程序设计在某种程度上通过强调可重复性解决了这一问题。20 世纪 70 年代的 Smalltalk 语言在面向对象方面堪称经典——以至于 30 年后的今天依然将这一语言视为面向对象语言的基础。

支持部分或绝大部分面向对象特性的语言即可称为基于对象的或面向对象的语言。早期的完全面向对象的语言主要包括 Smalltalk 等语言,目前较为流行的语言中有 Java、C#、Eiffel 等。随着软件工业的发展,比较早的面向过程的语言在近些年的发展中也纷纷吸收了许多面向对象的概念,比如 C->C++, C->Objective-C, BASIC->Visual Basic->Visual Basic .NET, Pascal->Object Pascal, Ada->Ada95。

近年来,面向对象的程序设计越来越流行于脚本语言中。Python 和 Ruby 是创建在 OOP 原理上的脚本语言,Perl 和 PHP 亦分别在 Perl 5 和 PHP 4 时加入面向对象特性。

面向对象程序设计可以看作一种在程序中包含各种独立而又互相调用的对象的思想,这与传统的思想刚好相反。传统的程序设计主张将程序看作一系列函数的集合,或者直接就是一系列对电脑下达的指令。面向对象程序设计中的每一个对象都应该能够接受数据、处理数据并将数据传达给其它对象,因此它们都可以被看作一个小型的“机器”,即对象。面向对象程序设计将对象作为程序的基本单元,将程序和数据封装其中,以提高软件的重用性、灵活性和扩展性。

计算机科学中对象和实例概念的最早萌芽可以追溯到麻省理工学院的 PDP-1 系统,该系统大概是最早的基于容量架构(capability based architecture)的实际系统。另外 1963 年 Ivan Sutherland 的 Sketchpad 应用中也蕴含了同样的思想。对象作为编程实体最早是于 1960 年代由 Simula 67 语言引入思维。

Simula 这一语言是奥利-约翰·达尔和克利斯登·奈加特在挪威奥斯陆计算机中心为模拟环境[1]而设计的,这种办法是分析式程序的最早概念体现。在分析式程序中,我们将真实世界的对象映射到抽象的对象,这叫做“模拟”。Simula 不仅引入了“类”的概念,还应用了实例这一思想——这可能是这些概念的最早应用。

20 世纪 70 年代施乐 PARC 研究所发明的 Smalltalk 语言将面向对象程序设计的概念定义为,在基础运算中,对对象和消息的广泛应用。Smalltalk 的创建者深受 Simula 67 的主要思想影响,但 Smalltalk 中的对象是完全动态的——它们可以被创建、修改并销毁,这与 Simula 中的静态对象有所区别。此外,Smalltalk 还引入了继承性的思想,它因此一举超越了不可创建实例的程序设计模型和不具备继承性的 Simula。此外,Simula 67 的思想亦被应用在许多不同的语言,如 Lisp、Pascal。

面向对象程序设计在 80 年代成为了一种主导思想,这主要应归功于 C++——C 语言的扩充版。在图形用户界面(GUI)日渐崛起的情况下,面向对象程序设计很好地适应了潮流。GUI 和面向对象程序设计的紧密关联在 Mac OS X 中可见一斑。Mac OS X 是由 Objective-C 语言写成的,这一语言是一个仿 Smalltalk 的 C 语言扩充版。面向对象程序设计的思想也使事件处理式的程序设计更加广泛被应用(虽

[1]奥利-约翰·达尔和克利斯登·奈加特为了模拟船只而设计的 Simula 语言,并且对不同船只间属性的相互影响感兴趣。他们将不同的船只归纳为不同的类,而每一个对象,基于它的类又可以定义它自己的属性和行为。

然这一概念并非仅存在于面向对象程序设计)。一种说法是,GUI 的引入极大地推动了面向对象程序设计的发展。

苏黎世联邦理工学院的尼克劳斯·维尔特和他的同事们对抽象数据和模块化程序设计进行了研究。Modula-2 将这些都包括了进去,而 Oberon 则包括了一种特殊的面向对象方法——不同于 Smalltalk 与 C++。

面向对象的特性也被加入了当时较为流行的语言:Ada、BASIC、Lisp、Fortran、Pascal 以及种种。由于这些语言最初并没有面向对象的设计,故而这种糅合常常会导致兼容性和维护性的问题。与之相反的是,"纯正的"面向对象语言却缺乏一些程序员们赖以生存的特性。在这一大环境下,开发新的语言成为了当务之急。作为先行者,Eiffel 成功地解决了这些问题,并成为了当时较受欢迎的语言。

在过去的几年中,Java 语言成为了广为应用的语言,除了它与 C 和 C++ 语法上的近似性。Java 的可移植性是它的成功中不可磨灭的一步,因为这一特性,已吸引了庞大的程序员群的投入。在最近的计算机语言发展中,一些既支持面向对象程序设计,又支持面向过程程序设计的语言开始出现,包括 Python、Ruby 等等。

正如面向过程程序设计使得结构化程序设计的技术得以提升,现代的面向对象程序设计方法使得对设计模式的用途、契约式设计和建模语言(如 UML)技术也得到了一定提升。

类

类(Class)定义了一件事物的抽象特点。通常来说,类定义了事物的属性和它可以做到的(它的行为)。举例来说,"狗"这个类会包含狗的一切基础特征,例如它的孕育、毛皮颜色和吠叫的能力。类可以为程序提供模版和结构。一个类的方法和属性被称为"成员"。

```
类  狗
开始
    私有成员:
            孕育:
    毛皮颜色:
    公有成员:
            吠叫():
结束
```

在这串代码中,我们声明了一个类,这个类具有一些狗的基本特征。

对象

对象(Object)是类的实例。例如,"狗"这个类列举狗的特点,从而使这个类定义了世界上所有的狗。而莱丝这个对象则是一条具体的狗,它的属性也是具体的。狗有皮毛颜色,而莱丝的皮毛颜色是棕白色的。因此,莱丝就是狗这个类的一个实例。一个具体对象属性的值被称作它的"状态"。(系统给对象分配内存空间,而不会给类分配内存空间,这很好理解,类是抽象的系统不可能给抽象的东西分配空间,对象是具体的)

假设我们已经在上面定义了狗这个类,我们就可以用这个类来定义对象:

```
定义莱丝是狗
莱丝.毛皮颜色:=棕白色
莱丝.吠叫()
```

我们无法让狗这个类去吠叫,但是我们可以让对象"莱丝"去吠叫,正如狗可以吠叫,但没有具体的狗就无法吠叫。

方法

方法(Method,可看成能力)是定义一个类可以做的,但不一定会去做的事。作为一条狗,莱丝是会叫的,因此"吠叫 ()"就是它的一个方法。与此同时,它可能还会有其它方法,例如"坐下 ()",或者"吃

()”。对一个具体对象的方法进行调用并不影响其它对象，正如所有的狗都会叫，但是你让一条狗叫不代表所有的狗都叫。如下例：

```
定义莱丝是狗
定义泰尔是狗
莱丝.吠叫()
```

则泰尔是会叫——但没有吠叫，因为这里的吠叫只是对对象“莱丝”进行的。

消息传递

一个对象通过接受消息、处理消息、传出消息或使用其他类的方法来实现一定功能，这叫做消息传递机制（Message Passing）。

继承

继承性（Inheritance）是指，在某种情况下，一个类会有“子类”。子类比原本的类（称为父类）要更加具体化，例如，“狗”这个类可能会有它的子类“牧羊犬”和“吉娃娃犬”。在这种情况下，“莱丝”可能就是牧羊犬的一个实例。子类会继承父类的属性和行为，并且也可包含它们自己的。我们假设“狗”这个类有一个方法叫做“吠叫 ()”和一个属性叫做“毛皮颜色”。它的子类（前例中的牧羊犬和吉娃娃犬）会继承这些成员。这意味着程序员只需要将相同的代码写一次。

```
类牧羊犬:继承狗

定义莱丝是牧羊犬
莱丝.吠叫()   /* 注意这里调用的是狗这个类的吠叫方法。 */
```

回到前面的例子，“牧羊犬”这个类可以继承“毛皮颜色”这个属性，并指定其为棕白色。而“吉娃娃犬”则可以继承“吠叫 ()”这个方法，并指定它的音调高于平常。子类也可以加入新的成员，例如，“吉娃娃犬”这个类可以加入一个方法叫做“颤抖 ()”。设若用“牧羊犬”这个类定义了一个实例“莱丝”，那么莱丝就不会颤抖，因为这个方法是属于吉娃娃犬的，而非牧羊犬。事实上，我们可以把继承理解为“是”。例如，莱丝“是”牧羊犬，牧羊犬“是”狗。因此，莱丝既得到了牧羊犬的属性，又继承了狗的属性。

```
类吉娃娃犬:继承狗
开始
   公有成员:
      颤抖()
结束
类牧羊犬:继承狗

定义莱丝是牧羊犬
莱丝.颤抖()   /* 错误：颤抖是吉娃娃犬的成员方法。 */
```

当一个类从多个父类继承时，我们称之为“多重继承”。多重继承并不总是被支持的，因为它很难理解，又很难被好好使用。

封装

具备封装性（Encapsulation）的面向对象程序设计隐藏了某一方法的具体执行步骤，取而代之的是通过消息传递机制传送消息给它。因此，举例来说，“狗”这个类有“吠叫 ()”的方法，这一方法定义了狗具体该通过什么方法吠叫。但是，莱丝的朋友蒂米并不需要知道它到底如何吠叫。从实例来看：

```
/* 一个面向过程的程序会这样写： */
定义莱丝
莱丝.设置音调(5)
莱丝.吸气()
```

```
莱丝.吐气()

/* 而当狗的吠叫被封装到类中，任何人都可以简单地使用： */
定义莱丝是狗
莱丝.吠叫()
```

封装是通过限制只有特定类的实例可以访问这一特定类的成员，而它们通常利用接口实现消息的传入传出。举个例子，接口能确保幼犬这一特征只能被赋予狗这一类。通常来说，成员会依它们的访问权限被分为 3 种：公有成员、私有成员以及保护成员。有些语言更进一步，例如：

- Java 可以限制同一包内不同类的访问；
- C# 和 VB.NET 保留了为类的成员聚集准备的关键字：internal（C#）和 Friend（VB.NET）；
- Eiffel 语言则可以让用户指定哪个类可以访问所有成员。

多态

多态（Polymorphism）是指由继承而产生的相关的不同的类，其对象对同一消息会做出不同的响应。例如，狗和鸡都有“叫 ()”这一方法，但是调用狗的“叫 ()”，狗会吠叫；调用鸡的“叫 ()”，鸡则会啼叫。我们将它体现在伪代码上：

```
类狗
开始
   公有成员：
       叫()
       开始
          吠叫()
       结束
结束

类鸡
开始
   公有成员：
       叫()
       开始
          啼叫()
       结束
结束

定义莱丝是狗
定义鲁斯特是鸡
莱丝.叫()
鲁斯特.叫()
```

这样，同样是叫，莱丝和鲁斯特做出的反应将大不相同。多态性的概念可以用在运算符重载上。

抽象

抽象 (Abstraction) 是简化复杂的现实问题的途径，它可以为具体问题找到最恰当的类定义，并且可以在最恰当的继承级别解释问题。举例说明，莱丝在大多数时候都被当作一条狗，但是如果想要让它做牧羊犬做的事，你完全可以调用牧羊犬的方法。如果狗这个类还有动物的父类，那么你完全可以视莱丝为一个动物。

目前已经被证实的是，面向对象程序设计推广了程序的灵活性和可维护性，并且在大型项目设计中广为应用。当我们提到面向对象的时候，它不仅指一种程序设计方法，更多意义上它是一种程序开发方式。在这一方面，我们必须了解更多关于面向对象系统分析和面向对象设计方面的知识。

Bibliography

[1] Wikipedia. Object. URL http://en.wikipedia.org/wiki/Object_(computer_science).

Part I

Introduction

Overview

面向对象思想诞生于 20 世纪 60 年代，并且最早被应用于 Simula 语言中，到 20 世纪 80 年代，面向对象编程语言才真正引起计算领域的普遍关注。

结构化程序设计基于任务的层次划分，而面向对象的设计则基于数据对象的层次划分。

从技术上讲，Java 语言具有如下优点：

- 简单

  Java 语言的语法类似于 C++，但摒弃了 C++ 中容易引发程序错误的地方，同时提供了丰富的类库。
- 面向对象

  Java 语言的设计是完全面向对象的。除了基本数据类型外，Java 的所有数据都是用对象表示的。
- 分布式

  Java 同时支持数据分布和操作分布。

  对于数据分布，Java 通过 URL 对象可以打开并访问相同 URL 地址上的对象，访问方式与访问本地文件系统相同。

  对于操作分布，Java 的 applet 可以从服务器下载到客户端，即部分计算在客户端进行，提高系统执行效率。
- 可靠性

  Java 是强类型的语言，要求显式的方法声明，这保证了编译器可以发现方法调用错误，保证程序更加可靠。

  Java 不支持指针，杜绝了内存的非法访问。

  Java 的自动单元收集避免了内存丢失等动态内存分配导致的问题。

  Java 解释器运行时实施检查，可以发现数组和字符串访问的越界。

  Java 提供了异常处理机制，可以把一组错误代码放在一个地方，这样可以简化错误处理任务，便于恢复。
- 安全性
- 可移植性

  大多数编译器产生的目标代码只能运行在一种 CPU 上，即使那些能支持多种 CPU 的编译器也不能同时产生适合多种 CPU 的目标代码。但是，Java 编译器产生的目标代码（J-Code）是针对 Java 虚拟机（Java Virtual Machine）的，而不是实际存在的 CPU。通过 JVM 可以掩盖不同的 CPU 之间的差异，使 J-Code 能运行于任何具有 JVM 的机器上。
- 多线程

  Java 通过两方面支持多线程。一方面，Java 环境本身就是多线程的，若干个系统线程运行负责必要的 GC、系统维护等系统级操作；另一方面，Java 语言内置多线程控制，可以简化多线程应用程序的开发。
- 动态

  Java 的动态特性允许程序动态地加载运行过程中所需要地类。
- 支持 Unicode

  Java 支持 Unicode，从而使得 Java 地程序能在不同语言地平台上编写和执行。

通过 Java Servlet 可以将使用 Java 开发的应用程序挂载在 Apache 或其他 Web 服务器上，Java applet 可以在网页浏览器中运行。

Java 与 Oracle 配合可以实现数据库的存储过程等。

0.7 Primitive Type

在计算机科学中，作为创建基础的原始类型（Primitive type）是由编程语言提供的数据类型，有别于复合类型，因此原始类型也有称作内置类型、基础类型或者基本类型。根据语言及其实现，原始类型在存储器中可能没有与对象一对一的对应。

原始类型的实际范围取决于所使用的特定编程语言。例如，在 C 语言中，字符串是一个复合类型，但是在后来的 Basic 中，字符串是原始类型。

具体来说，典型的原始类型包含字符、整数、浮点数、定点数、布尔类型和引用等。

- 字符（character、char），依字符集又分为 SBCS、DBCS、MBCS 这三大类；
- 整数（integer、int、short、long、byte）有各种精度，依是否可表示负数的数值又再各自区分有符号数与无符号数；
- 浮点数（float、double、real、double precision）；
- 定点数（fixed），其有各种精度，以及所选的数量级。
- 布尔类型有真和假两值。
- 引用（又称作指针或 handle），它是一个较小的值，指向其它可能大得多的对象所在的存储器地址。

在编程语言的发展过程中又出现了更多更复杂的原始类型，例如多元组、链表、复数、有理数和闭包等。

- 出现于 ML、Python 中的多元组。
- 出现于 Lisp 中的链表。
- 出现于 Fortran、C99、Python 中的复数。
- 出现于 Lisp 中的有理数。
- 出现于函数式编程语言（如 Lisp 和 ML）中的 first class function（首类）、closure（闭包）和 continuation 等。

通常在预期以基于原始类型的运算来构成较快速执行速度的语言中，整数加法可以运行单一的机器指令，且部分处理器提供特定的指令来以单一指令处理一系列的字符等。例如，C 语言标准中提到“一个‘简单’的 int 对象，具有原始的大小，其大小以运行环境的架构所暗示。”说明在 32 位架构下的 int，可能就是 32-bit 的长度。

大部分语言不允许以程序来修改原始类型的行为或性能。不过，例外的有 Smalltalk，其允许在程序内部扩展原始类型，因此可以在其上运行加入的运算，或者重新定义内置的运算。

Part II

Design Pattern

Introduction

继续提升抽象的层次,可以开始考虑类的更一般的关系,而不仅仅是父子关系。

在两个或更多个类(或者对象)中的相互作用,都是通过设计模式(design pattern)的形式来描述和定义的。