**The Object of Object Oriented Programming**

**Characteristics of OOP**

All design principles in programming are about managing complexity. Object Oriented Programming (OOP) tries to solve the complexity problem by subdividing a system into discrete, interchangeable, and manageable pieces called objects. Alan Kay, the pioneer behind the Smalltalk language, described OOP as “a successful attempt to qualitatively improve the efficiency of modeling the ever more complex dynamic systems and user relationships made possible by the silicon explosion.” [3] While OOP is a broad classification, it is generally attributed with a few core concepts and strategies.

Perhaps the most central idea to OOP is *encapsulation*. The idea is that we bundle related data and the functionality surrounding that data into one place: a class object. This goes hand-in-hand with another major OOP tenet—*data hiding*. Classes make their internal state information private and extend a well-documented, clean public interface which describes an object’s behavior. Encapsulation and data hiding affords us a way to achieve *data abstraction*, which is a major goal in OOP. We only need to worry about the public interface of a class to understand how it behaves. We don’t need to know the details of a class’s implementation to be able to use, interact with, and even potentially modify it.

Why does this help? For one, it helps to improve the *cohesion* of the system. Cohesion is how well the individual parts of our system fit together to make a sensible, understandable whole. [1] Secondly, encapsulation can help to limit unwanted *coupling*. Coupling describes the level of interdependence of modules on one another. [1] OOP strives to achieve loose coupling. It is easier to build a complex system piece by piece if we don’t have to worry about breaking low-level details in other parts of the system. The oft-cited software engineering principle known as “The Law of Demeter” takes this further by suggesting that modules should only interact with closely-related modules. Why? Thinking on many levels of abstraction at once is hard. When we can consider the rest of the system a new piece interacts with in an abstract way modeled only by its most related components, we can significantly reduce our cognitive load.

OOP is also well-known for the idea of *inheritance*. Inheritance allows newly designed objects to use existing objects as a model. First, this allows us to reuse code we’ve already written in new and specialized contexts. [4] Second, this groups related objects into an organized tree hierarchy which helps us understand their relationships to one another and to the larger project. *Dynamic binding* and *polymorphism* afford us another benefit from inheritance. We can use common methods on objects and enjoy either generic superclass behavior or the specialized subclass functionality without knowing the type of class object in advance. [4] This is yet another way to move toward the OOP goal of loose coupling.

**Simula and Smalltalk – The Roots of OOP**

The first language to utilize the ideas of objects, classes, and inheritance as central concepts was Simula, which debuted in 1967. It was designed to perform event simulations, so objects were used to model real world constructs for the simulations. [2]

Simula, however, was not directly responsible for the classification of these ideas as a popular design paradigm called “Object Oriented Design.” The Smalltalk language is generally credited with this, though it was influenced heavily by Simula. Smalltalk was part of an ambitious project by Alan Kay in 1972 called the Dynabook. The Dynabook was originally conceived as a sort of cross between a laptop and tablet controlled with a stylus with children as its target audience. Kay had a vision that “in the 1990's there will be millions of personal computers. They will be the size of notebooks of today, have high-resolution flat-screen reflective displays, weigh less than ten pounds, have ten to twenty times the computing and storage capacity of [the current models].”[3] He states that he wanted children to understand computing because “like literature, mathematics, science, music, and art, it carries special ways of thinking about situations that in contrast with other knowledge and other ways of thinking critically boost our ability to understand our world.”[3]

Though the Dynabook itself did not realize this vision in its entirety in the 70s, Smalltalk’s aim to simplify the daunting technical task of programming by making it easier, faster, and more accessible was a powerful transformative force on OOP languages to follow it.

**Modern OOP languages**

OOP languages in use today include C++, Java, Python, and Ruby. Though all support the primary OOP features, they have some deep and meaningful differences. Since our class focuses on Java, I’ll use that as a control and reference point for how the others differ. C++ was an adaptation of its predecessor, C, to the OOP paradigm. Indeed, its original name while under development was “C with Classes.” C++ supports too many features to discuss at length here, but notably it is not an automatically garbage collected language like Java, though it does use the keyword *new* for object instantiation. C++ also provides real generic types (rather than the syntactic sugar types of Java) and allows for direct pointer manipulation. Notably, C++ is compiled to architecture-specific machine code rather than using a virtual machine or interpreter. Python and Ruby differ from both C++ and Java by way of dynamic typing, sometimes referred to informally as *duck typing*. Data types exist but are handled implicitly at *runtime* unless an explicit cast is used. By contrast, Java is *statically* typed. Its type-checking is handled at *compile* time. In Python, whitespace is used by the compiler to determine blocks rather than braces. Ruby takes OOP to the next level such that everything is an object type (i.e., there are no primitive value types). [4] It follows the “Principle of Least Astonishment,” which is just that people should find the code obvious and intuitive rather than strange and archaic.

Though each language is general purpose and follows a reasonably similar overall paradigm, each has its own set of unique libraries which may be better suited for certain types of tasks. Python, for instance, has a robust selection of libraries for machine learning and data analytics. Ruby is utilized largely in tandem with the Rails framework for web development. Java is an older enterprise language and is used in a large variety of legacy applications as well as on the Android platform. C++ is extremely customizable and flexible, though this comes at the cost of complexity. It is often used in applications where performance matters like desktop application development, especially video game development.

**OOP Amongst other paradigms**

OOP falls into what is generally referred to as an *imperative* paradigm. Imperative languages focus on variables, which maintain state, and on procedures which modify the state. This contrasts with what is known as the *declarative* paradigm. Declarative languages describe logic without explicitly ascribing control flow to accomplish the task. SQL and Prolog are prototypically declarative languages. A subset of declarative languages often considered separately are *functional* programming languages. Functional languages treat functions (in the mathematical sense) as the primary building blocks of computation, and they are concerned primarily with inputs and outputs rather than state. Languages like Lisp and Haskell are often associated with this type of programming.

This distinction is not always a clean and neat divide. Many general-purpose, primarily imperative languages like C++ and JavaScript allow for declarative features. Java also introduced lambda expressions in version 8, which are a functional programming feature. Still, the ideas are worth keeping in mind when thinking about how our programming language of choice is asking us to think, what tradeoffs we are making, and what we are getting in return.

**References**

[1] Capretz, Luiz. (2003). A brief history of the object-oriented approach. ACM SIGSOFT Software Engineering Notes. 28. 6. 10.1145/638750.638778.

[2] Mitchell, J. (2002). History of Objects: Simula and Smalltalk. In Concepts in Programming

Languages (pp. 300-336). Cambridge: Cambridge University Press. doi:10.1017/CBO9780511804175.012

[3] Kay, Alan C. (1993) The Early History of Smalltalk. ACM SIGPLAN Notices Volume 28, No. 3, March 1993 Pages 69-95.

[4] Hristakeva, M. and Vuppala, R. (2009), A Survey of Object Oriented Programming Languages, Technical report, University of California, Santa Cruz, California.

URL: <http://users.soe.ucsc.edu/~vrk/Reports/oopssurvey.pdf> (Accessed 6.15.2018)