[PD-01-021] Object-Oriented Programming in GIS Applications

Abstract

Object-Oriented Programming (OOP) is a paradigm that abstracts everything in a program to objects, providing a template for varying classes. Offering flexibility through subclasses while maintaining consistency in the main codebase, it has revolutionized code structuring and significantly influenced today's popular programming languages. Concurrently, GIS leverages OOP to model geographic entities as models and encapsulate their attributes. This seamless integration highlights the enduring connection between Computer Science and GIS. OOP allows for different collaborative patterns to organize geospatial data. However, there are limitations in representing vector or raster data within this framework, which can be addressed with Machine Learning frameworks.

Keywords: encapsulation, Java, object-oriented

Author & citation

Cornejo, W. (2025). Object-Oriented Programming in GIS Applications. The Geographic Information Science & Technology Body of Knowledge (Issue 1, 2025 Edition), John P. Wilson (Ed). DOI: 10.22224/gistbok/2025.1.15.

Explanation

1. Principles of Object-Oriented Programming (OOP)
2. GIS Introduced to OOP
3. Object-Oriented Programming Patterns
4. Challenges and Limitations
5. Future Developments

1. Principles of Object-Oriented Programming

1.1 What is Object-Oriented Programming?

In programming, various methods exist to structure code efficiently. For small-scale tasks, defining variables directly in a file or command line may be sufficient. However, as the complexity grows, a need arises to automate certain actions and entities in a program. Functions can store actions that can be applied multiple times to different variables for the sake of memory and code efficiency. Although they help with code organization, functions were still just a set of actions, the data was always separate. This naturally led to Object-Oriented Programming (OOP), a paradigm that organizes software design around objects rather than functions and logic. An object represents a single unit that encapsulates data and behavior, often modeling real-world entities. This abstraction enhances code reusability and has influenced many modern programming languages (Stroustrup, 1993).

Everything in OOP can be made into a class, which serves as a blueprint for a type of

object, with each instance sharing the methods and characteristics defined in the class. For example, if we define a class "Phone" with attributes "color," "model," and "memory," methods for this class could be "print_color" to display the phone's color or "call" to initiate a phone call. Any "Phone" created will inherit these attributes and methods. If we want more specific objects, subclasses "Home Phone" or "CellPhone" will inherit the characteristics of their parent class. A subclass can also add to its unique objects and functions, as well as override methods from the parent for specialized functionality. For example, a "setup" method for the "Phone" class might involve different steps for a home phone versus a cell phone, reflecting their unique setup processes.

1.2 History

The first iteration of OOP was with Simula, a programming language developed in the 1960s by Ole-Johan Dahl and Kristen Nygaard. Originally developed to simulate discrete events that Nygaard encountered with Monte Carlo Systems when working at the Norwegian Defense Research Establishment, Simula introduced dynamic binding, which refers to executing method calls at runtime rather than at compile time. A function could then be referenced at execution of a file. Through the encapsulation of functions, Simula allowed for the modeling of early complex systems and significantly influenced the way programmers approached software design (Kroghdahl, 2003).

Smalltalk was developed in the 1970s at Xerox PARC by Alan Kay, Dan Ingalls, and Adele Goldberg. It expanded the concept of what could be an object, including even primitive data types and structures. This approach was carried into many common languages like Java and Python making OOP more mainstream and practical (Kay, 1993). Smalltalk provided an integrated development environment (IDE) which allowed for interactive and flexible programming (Goldberg & Robson, 1983). With a browser-based interface for easy navigation, as well as tools and libraries for testing and deploying code, Smalltalk's IDE allowed for rapid development of programs. Modern IDEs are built on the core principles of data sharing from Object-Oriented Programming. JupyterLab, for example, enables web-accessible code to be run in blocks, a popular format for data science. GitHub has become a key platform for enhancing collaboration among developers by offering repositories for their projects. The ideas and innovations introduced by Smalltalk continue to shape modern programming paradigms and tools, reinforcing its legacy in the evolution of software development.

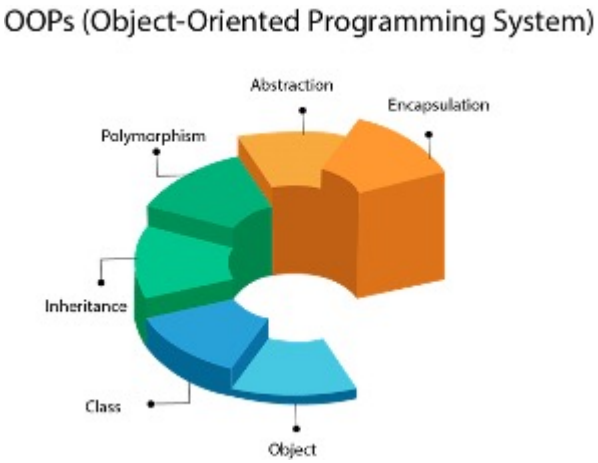


Figure 1. An Object-Oriented Programming System. Source: author.

In the early 1980s, Bjarne Stroustrup revolutionized programming with the development of

C++, a language that significantly advanced the object-oriented programming paradigm by combining the efficiency and flexibility of C (C++'s predecessor) with the powerful principles of OOP. C++ introduced classes along with access specifiers, constructors, destructors, and operator overloading, providing a robust framework for building complex software systems with portable components (Stroustrup, 1993). Its support for multiple inheritance and templates enabled developers to write more generic and reusable code, favorable for large-scale software engineering projects. C++ also emphasized performance using a compiler, ideal for real-time applications where efficiency was necessary (Stroustrup, 1993). C++ played a pivotal role in popularizing OOP in both academic and industrial settings, bridging the gap between high-level design and low-level system programming. Java, created by James Gosling in 1995, further propelled object-oriented programming by offering a platform-independent environment, introducing a pure object-oriented approach and emphasizing strong memory management with automatic garbage collection. **Java** is known for its verbose way of programming, having descriptive names for functions and other operations, and has become one of the most popular programming languages, profoundly influencing software development practices and the design of modern software architectures (Arnold et al., 2000).

2. GIS Introduced to OOP

2.1 How does OOP Apply to GIS

Integrating OOP with GIS offers significant benefits. OOP allows for the creation of modular extensible GIS applications, where complex geographical features and their behaviors can be modeled as objects. A "River" object in a GIS application could have attributes like length, flow rate, and pollution levels, along with methods to simulate seasonal flow variations or to calculate the impact of pollutants. This modularity not only simplifies the development process but also enhances the ability to manage and update geographic models (Zeiler, 1999).

Encapsulation ensures that the internal state of objects can only be changed by the user, promoting robust and reliable GIS applications. Inheritance allows for the creation of specialized objects from general ones, enabling the development of complex hierarchical relationships within geographic data.

2.2 History of Object-Oriented GIS

During the 1990s, object-oriented principles began to integrate into GIS to address limitations of earlier GIS technologies, which revolutionized the way spatial data was modeled, stored, and processed. Traditional GIS systems were based on relational databases, which handled spatial data as tables, with rows and columns. While effective for certain types of data management, this approach struggled with the complex relationships and behaviors found in geographic data. The rigidness of relational databases made it difficult to represent real-world entities and their interactions dynamically.

Object-oriented GIS emerged as a solution to these challenges by leveraging OOP to model geographic features as objects. Each object encapsulated both the attributes and methods relevant to real-world entities, such as roads, rivers, or parcels of land. With the large amounts of geographic data being gathered, creating an ontology for varying types of data and linking them with object-oriented mapping ensured interoperability (Fonseca and Egenhofer, 1999). This allowed for more natural and intuitive representations of geographic

phenomena, facilitating more sophisticated analyses and simulations.

A key development in object-oriented GIS during the 1990s was the introduction of object-oriented databases specifically designed for spatial data. These databases (e.g., the ESRI Geodatabase) allowed for the seamless integration of spatial and non-spatial data, supporting complex data types and relationships (Zeiler, 1999). The Geodatabase model provided a framework for storing geographic data in a hierarchical structure, where feature classes and relationships could be defined and managed in an object-oriented manner.

ESRI's geodatabase is not object oriented, at best it applies a limited set of OO principles on top of a classic relational database. The hierarchy is limited to datasets, classes and subtypes. There is no provision of specialized methods, only for setting attribute constraints. It is now downplayed by ESRI in its latest versions of ArcGIS Pro/Online/Enterprise. ESRI's geodatabase is defined as a top-level unit of geographic data, managing datasets, feature classes, and object classes. Being built on top of a classic relational database, it is limited in the set of OO principles it can apply.

The incorporation of inheritance and polymorphism further enhanced the flexibility and efficiency of GIS applications. A "Transit Network" class could serve as a base class for "Road Network" and "Railway Network," each inheriting common attributes and methods while introducing unique characteristics. Complex data structures became more compatible, and clients could aggregate data from different sources (Fonseca and Egenhofer, 1999). In an Ontology-Driven Geographic Information System, objects are referenced by their own ontologies and each stored in containers, where a user can retrieve and manipulate its methods.

During this period, several influential GIS software products, like ArcInfo and Smallworld, followed suit. These systems demonstrated the practical benefits of object-oriented GIS, such as improved data integrity, enhanced modeling capabilities, and greater extensibility. The ability to create custom objects and extend the core functionality of GIS software empowered users to develop tailored solutions for specific applications, ranging from urban planning to environmental monitoring. Smallworld was developed by General Electrics for the telecommunication and utilities industry and is the mechanism for how the National Gird manages electricity and gas networks. Smallworld formatted data as multiple types of geometry and allowed versioning work for backup.

3. Object-Oriented Programming Patterns

The continued application of OOP techniques to spatial entities has led to the emergence of design patterns tailored to GIS applications. These patterns provide structured solutions to common problems, promoting code reusability, scalability, and maintainability. This section explores key design patterns in the context of GIS.

3.1 Design Patterns in Object-Oriented GIS

Design patterns serve as reusable frameworks for solving recurring problems, helping developers efficiently create GIS applications. As guiding principles, each offers a unique balance between flexibility, reusability, and scalability in software design. By promoting collaboration among developers, they foster a culture of shared knowledge and collective problem-solving. These patterns empower developers to address complex challenges with generalized solutions (Prakash et al., 2018). Each pattern prioritizes one of the key aspects

of OOP mentioned earlier.

3.2 Types of Patterns

The Singleton Pattern ensures only one instance exists for class at a time with global access. This is ideal for simple scenarios that do not require flexibility, like for enterprises that require centralized control and want to avoid parallel processing conflicts while keeping critical code protected.

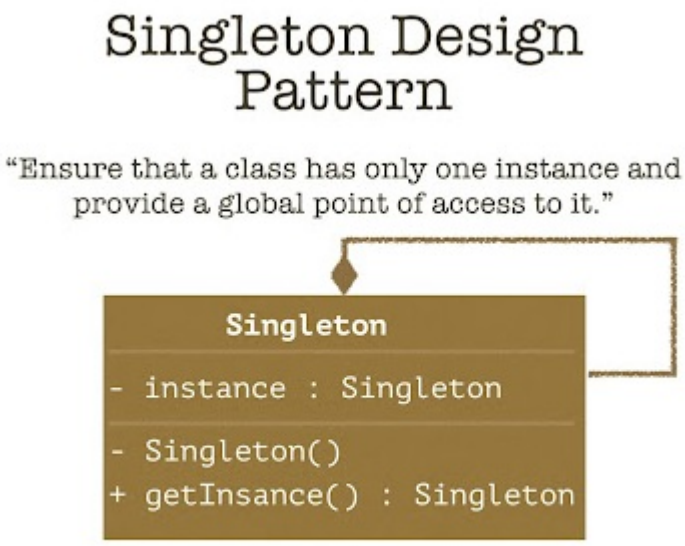


Figure 2. The Singleton pattern. Source: author.

The Factory pattern defines a class that allows its subclasses to represent different types. Being a popular option for patterns, it allows for creating a family of related objects without specifying the exact class of each object. This responsibility is taken care of by a factory class which determines the appropriate class based on input.

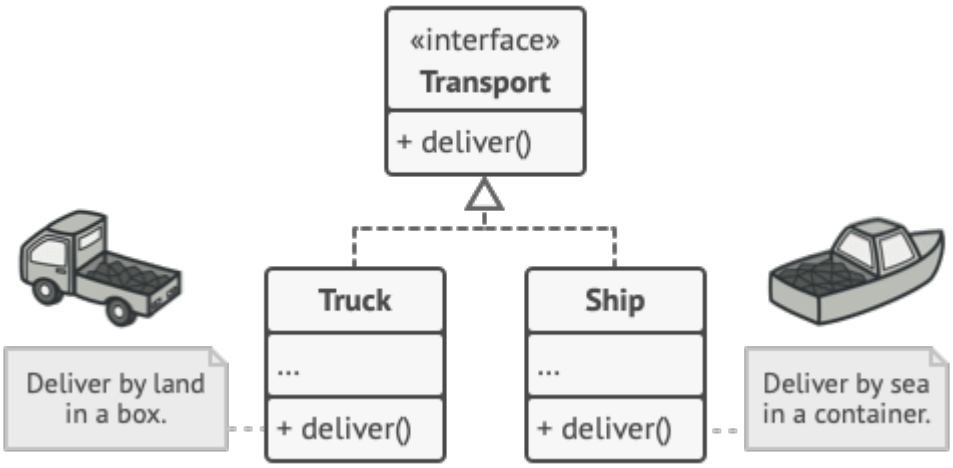


Figure 3. The Factory pattern. Source: author.

Singleton and Factory fall under the umbrella of creational patterns, which have a goal of making a system independent of how its objects are created, composed and represented, ensuring the system remains adaptable to how objects are created. On the other hand, Behavioral design patterns focus on how objects interact in a system, ensuring good communication. The following two are examples of behavioral patterns.

The Observer Pattern allows for objects to have one-to-many dependencies so that when the state of one object changes, its dependencies are updated automatically. This is ideal for event handling systems, like an emergency, where change can be communicated efficiently. A real-world example of an Observer pattern would be a weather reporting system which offers real-time information through various sensors, user interfaces, and one central server. The central server acts as the parent class which maintains its list of user interfaces, each acting as class instances. Once a sensor receives alarming data, the central server updates its state and notifies all connected interfaces.

The Composite Pattern can be thought of as a parent class to classes as well as objects. If there are multiple subclasses, the composite class can group and access these ‘grandchild’ classes. It is useful in GIS when a map has multiple layers, and the programmer wishes to establish a hierarchy.

Finally, the Strategy Pattern is best for when a specific problem can be solved with multiple algorithms, which offers the user control over the system's behavior in choosing the appropriate algorithm (Prakash et al., 2018). The pattern also supports software augmentation by adhering to the Open-Closed Principle. This rule defines two modes for a class: open for extension or closed for modification.

4. Challenges and Limitations

4.1 Data Integrity

OOP seems ideal for vector-based GIS data, where there are clear distinctions between different types of data. However, maintaining the spatial relationships between objects adds complexity to the framework. This ramps up in a dynamic and fast paced environment where new data is constantly introduced. Raster data behaves differently than its vector-based counterpart, as there are multiple cells to be categorized as objects. Defining the boundaries of raster data can be difficult and varies case by case.

There is also a learning curve to viewing data in an OOP perspective, which speaks to the growing connection between GIS and Computer Science. While GIS is rooted in geography, it has been in a transitional phase, attempting to construct the theory behind spatial analysis and develop the software (Bowlick et al., 2019). Today, GIS is expanding into many other fields and growing alongside Computer Science. Modern GIS curricula introduce students to popular GIS software (ArcGIS and QGIS) and also introduce programming concepts that enhance GIS capabilities (Bowlick et al., 2019). College programs now emphasize proficiency in web-based GIS, Python, and geodatabase design, integrating core Computer Science topics.

5. Future Developments

5.1 GIS and Machine Learning/AI

Technological advancements have significantly increased the volume of data collected from

the internet and other sources, creating a growing need for efficient organization. Tools like Docker and Kubernetes address this challenge by providing containerization platforms that package projects into containers, which can be deployed to clusters. These clusters allow multiple containers to operate in an environment where they are both accessible and isolated from one another. This technological evolution has also driven the development of advanced spatial analysis methods, greatly enhancing the capabilities of GIS.

Similar to GIS, certain branches of Computer Science are expanding into various fields. Machine Learning and AI are being developed at a fast pace and becoming more integrated in our society, like social media. Current resources allow for complex modes that can revolutionize spatial analysis by enabling sophisticated pattern recognition, predictive modeling, and automated decision-making processes. Machine learning algorithms can analyze vast amounts of spatial data to uncover insights that traditional methods might lack. When combined with object-oriented programming (OOP), the development of GIS applications becomes even more powerful. Various Machine Learning models can be encapsulated in a class object with a portable interface, making it easier to integrate into large systems and update class methods (Roy et al., 2024). For example, Support Vector Machines, a type of Supervised Learning, are already used in land cover analysis to distinguish water and land. Defining a class for the model would abstract its functionality and have it ready to use by others. A GIS application can employ OOP to create classes representing different spatial entities and integrate ML models to predict changes or classify features based on spatial attributes. This synergy enhances the flexibility and maintainability of GIS software, allowing developers to build more efficient and adaptive systems that can evolve with advancing ML techniques.

References

Arnold, K., Gosling, J., and Holmes, D. (2005). THE Java Programming Language, Third Edition. Addison-Wesley Professional.

Bowlick, F. J., Bednarz, S. W. and Goldberg, D. W. (2020), Course syllabi in GIS programming: Trends and patterns in the integration of computer science and programming. The Canadian Geographer / Le Géographe canadien, 64: 495-511

Fonseca, F. T. and Egenhofer, M. J. (1999). Ontology-driven geographic information systems. In Proceedings of the 7th ACM International Symposium on Advances in Geographic Information Systems (GIS '99). Association for Computing Machinery, New York, NY, USA, 14–19.

Goldberg, A. and Robson, D. (1983). Smalltalk-80: the language and its implementation. Addison-Wesley Longman Publishing Co.

Kay, A. C. (1993). The early history of Smalltalk. In The second ACM SIGPLAN conference on History of programming languages (HOPL-II). Association for Computing Machinery, New York, NY, USA, 69–95.

Krogdahl, S. (2005). The Birth of Simula. In: Bubenko, J., Impagliazzo, J., Sølvberg, A. (eds) History of Nordic Computing. HiNC 2003. IFIP International Federation for Information Processing, vol 174. Springer, Boston, MA.

Prakash, A., Ansari, M. Y., and Karri, N. A. (2018). Design Patterns in GIS Application Design. 2018 4th International Conference on Computing Communication and Automation (ICCCA), Greater Noida, India.

Roy, P. P., Abdullah, M.S., and Siddique, I. M. (2024). Machine learning empowered geographic information systems: Advancing Spatial analysis and decision making. World Journal of Advanced Research and Reviews, 22(01), 1387–1397

Stroustrup, B. (1993). A history of C++: 1979–1991. In The second ACM SIGPLAN conference on History of programming languages (HOPL-II). Association for Computing Machinery, New York, NY, USA, 271–297.

Zeiler, M. (1999). Modeling Our World: The ESRI Guide to Geodatabase Design. Redlands, CA: Esri Press.