Programming and Computing: Design Patterns, Binary Systems, and Program Synthesis

Anthony J. Coots

National University

TIM-8102: Principles of Computer Science

Dr. *******

May 18, 2025

Programming and Computing: Design Patterns, Binary Systems, and Program Synthesis

Computer programming is inherently established in its theoretical foundations and practical applications. Procedural and object-oriented programming approaches, computing design patterns, the binary number system, programming theory, and program synthesis have all played a significant role in the evolution of programming from its early beginnings to today's more complex systems. The goal of this paper is to explore these foundational elements, compare key programming paradigms, describe various design patterns, explain the principles of binary systems in computing, and analyze the significance of programming theory and program synthesis in real-world applications.

Procedural vs. Object-Oriented Programming Approaches

The choice between procedural and object-oriented programming paradigms is made before developing a software program, either explicitly or implicitly. If a software program should follow routines or a series of actions, the approach will likely use the procedural programming paradigm. On the other hand, object-oriented programming uses concepts such as encapsulation, modularity, inheritance, and cooperation across other objects to combine data and behavior.

Commented [JL1]: Great introduction to the section. You might consider including a real-world scenario or example for stronger context.

Procedural programming languages like C, Pascal, and Fortran follow a linear and sequential approach, and data is global, while functions run independently. Object-oriented languages like Java, C++, C#, etc., organize programs within a hierarchy, where objects, their functions and data are encapsulated, form relationships through paradigm-specific attributes, inheritance and polymorphism. One important difference between these paradigms is code reusability; procedural programming allows for much less reuse, seen primarily at the function

level, while object-oriented programming is renowned for its reusability, and the code is often shared with developers worldwide.

Commented [JL2]: Good point—consider briefly explaining how inheritance and polymorphism support reusability for clarity.

While object-oriented programming is often associated with better maintainability since it provides a modular structure, there is research that challenges this assumption. Brborich et al. (2020) compared maintenance effectiveness and speed across procedural and object-oriented program versions, which procedural programming demonstrated higher maintainability statistics. Specifically, across a subject group of over 90 university students, those maintaining procedural code performed maintenance tasks more effectively and faster than those maintaining object-oriented code, suggesting that under certain conditions, procedural programming can offer advantages in maintainability which reinforces the statement that the paradigm used in an approach should be based on the multiple factors. These factors include the nature of the problem, system complexity, scalability requirements, expertise of the development team, and general intuition of expected outcomes.

Computing Design Patterns

Overview

Design patterns are typical solutions to recurring problems in software development. They were first formally introduced by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, known as the "gang of four," in their 1994 book *Design Patterns: Elements of Reusable Object-Oriented Software*. This book helps standardize design decisions and improve software's maintainability, scalability, and readability. Design patterns are categorized as creational, structural, and behavioral, each of which is significant. Creational patterns (Singleton, Factory Method, and others) manage an object's instantiation (i.e., creating objects from classes). Structural patterns (Adapter, Bridge, Composite, etc.) address how classes and objects are

Commented [JL3]: Excellent breakdown! You might want to include a brief example for each category to ground the theory in practice.

composed. Behavioral patterns (Chain of Responsibility, Command, Observer, etc.) focus on object communication.

Theoretical Perspective

From a theoretical perspective, a design pattern fully encompasses the principle of abstractions, the principle of computer science that allows developers to conceptualize a program's structure without being overwhelmed by implementation details. Design patterns encourage modularity. Nabi et al. (2016) applied the Information Foraging Theory to create community-based design patterns for programming tools to frame developer optimization of information-seeking behaviors. The patterns draw from empirical evidence on how developers navigate, search for, and process code when debugging, refactoring, and reusing. The information foraging theory helps identify not only which design patterns are effective but also why they are effective.

Applied Perspective

From an applied perspective, developers apply design patterns to more effectively complete development tasks. Supekar and Khande (2024) explore how artificial intelligence may help bridge the gap between knowledge and practice in design pattern adoption. Their study demonstrates how AI-powered tools such as code analyzers, intelligent code suggestion systems, and automated refactoring engines can be used to identify, implement, and learn design patterns. For example, AI can recognize when the Factory Method pattern is appropriate based on a developer's code context and provide it in an integrated development environment as needed. Thus, these tools can reduce barriers for junior developers or effectively assist in development.

Binary System and Data Storage/Transmission

The binary system, which is absolutely foundational to all of modern digital computing, operates on a base-2 representation. In binary, all data is expressed using only two symbols: 0 and 1, which are known as binary digits, or bits; the term was created by John W. Tukey in 1946. Unlike analog systems, which represent information through continuous signals or physical analogies, digital systems rely on discrete binary states. By the mid-20th century, digital computers began to replace these analog systems by using signal propagation to represent two distinct states, where typically a high voltage represented a "True" state or "1" and a low voltage a "False" or "0". These binary states form the basis for all logic gates, memory storage, and data transmission across these computing systems.

Arguably, the most influential application of encoding binary in computing is using the American Standard Code for Information Interchange, otherwise known as ASCII. ASCII is the map between 7-bit binary numbers and text characters, and the representation thereof. For example, the letter "A" is 01000001 in binary, the (lowercase) letter "j" being 01001010. This allows computers to store, process, and transmit text reliably across different systems, as long as those systems follow the same standard. Internally, these characters are stored in memory as sequences of binary and can be interpreted or manipulated by programs depending on the machine's architecture, operating system, and software. When combined, sequences of ASCII characters, really binary values with a common representation, can represent source code, configuration files, and so on, forming the basis of computer programs and system interactions.

Commented [JL4]: Well-written historical and technical overview.

Synthesizing Programming Theory

Programming theory encircles the principles and concepts that construct computer programs' writing, analysis, and understanding. It informs the practice of programming by explaining how and why programs function to create software programs. A pertinent example of a field that relies on programming theory in computer science is Human-Computer Interaction, where the insights guide the development of meaningful and applicable software programs/systems.

Alzubi et al. (2025) show a relationship through the design of a multimodal deep learning-based human-computer interaction system for smart education environments. Their system, albeit it does not explicitly mention programming theory, integrates programming theory with user-focused design principles from human-computer interaction. The result is an adaptive learning platform that uses diverse input such as touch, voice, or gestures, and a multilayer Convolutional Neural Network to improve learning outcomes. This illustrates that understanding the high-level concepts of programming theory, like how an algorithm must translate user input to system responses, is important for the correct implementation and for creating pedagogically effective systems. The theory is not confined to abstract logic; it derives from the systems users interact with daily.

Commented [JL5]: Nice integration of current research!

Program Synthesis: Types and Relevance

Program synthesis refers to the automated construction of software programs from a given specification, described as the creation of “self-writing code” (David & Kroening, 2017). Instead of designing algorithms manually, a developer would describe desired behavior, and the program synthesizer would then generate an implementation that would satisfy that specification.

Commented [JL6]: Very informative and technically sound. You could highlight an industry application of each synthesis type to show real-world relevance.

David and Kroening (2017) identify three types of program synthesis: deductive synthesis, inductive synthesis, and counter example-guided inductive synthesis (CEGIS).

Deductive synthesis starts from complete formal specifications and derives programs that are correct by construction through logical deduction. However, it requires precise and usually complex specifications. Inductive synthesis, in contrast, works from partial specifications, such as sets of input-output examples, and infers programs that generalize from them. It is more flexible and adaptable to common development scenarios, but may not guarantee complete correctness. Lastly, Counter example-guided inductive synthesis combines the two approaches. With this, candidate programs are generated from partial examples and then tested against a verification oracle; if a counter example is found that the candidate fails on, then the synthesizer refines its search iteratively until a program is produced satisfying all cases.

Program synthesis and the aforementioned techniques have growing relevance and importance across application domains such as bug fixing, program repair, controller synthesis for embedded systems, end-user programming in spreadsheets, and education platforms to generate exercises and feedback. Program synthesis increases developer productivity and access to computational problem-solving without the generally expected programming expertise.

Though program synthesis has significant potential, it does have its challenges. One current challenge is the difficulty of creating complete and accurate specifications. Further than how deductive reasoning requires precise formal specifications, which is time-consuming and sometimes impractical, another prominent example involves the scalability of synthesis tools. Though CEGIS improves the overall efficiency by iteratively refining candidates, performance bottlenecks may occur, especially when codebases are large and the needed system behavior is

complex. David and Kroening (2017) mention in their work that optimized solver techniques remain an open area for research and innovation.

Hypothetical Programming Problem and Solution

Use Case

The following information demonstrates a pure hypothetical programming problem and its solution. The scenario involves developing a graphics tool that processes an image stored as a 32-bit RGBA array of values, 8 bits per Red, Green, Blue, and Alpha channel (for transparency). A user would upload an image represented in this 2-dimensional array, and the task would be to rotate it 90 degrees clockwise. Each pixel in this hypothetical problem and solution is a 32-bit integer.

Commented [JL7]: Clear and creative scenario.

Requirements

Input	2-dimensional matrix of 32-bit integers representing RGBA pixels.
Data Processing	Programming to preserve original values and perform specified rotation.
Output	2-dimensional matrix of 32-bit integers, rotated 90 degrees clockwise.

Programming Theory

The programmer must understand how 2-dimensional arrays map to memory to access data and correctly rotate positions. Modular functions should separate the logic into reusable parts, such as a function with a naming convention “rotate” or “rotate_image.” The rotation function abstracts implementation from the user. Though not the scope of this paper, the solution should involve the use of bit manipulation theory, as each pixel is a 32-bit value, the manipulation focuses on the position of the pixel itself, not the individual RGBA channels.

Figure 1

Main Flowchart and Python Code.

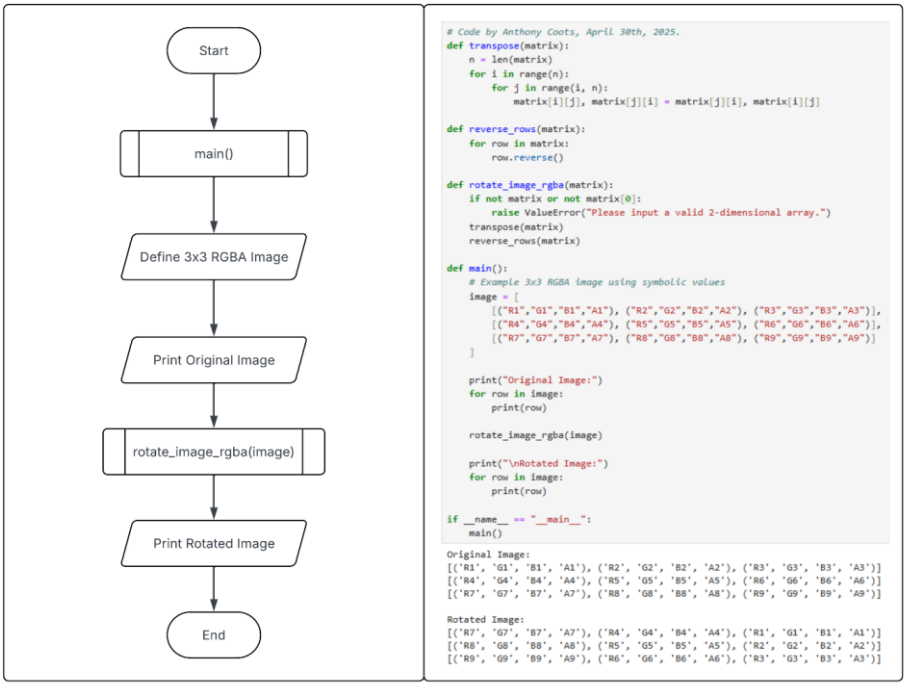


Figure 2

Rotate the RGBA image.

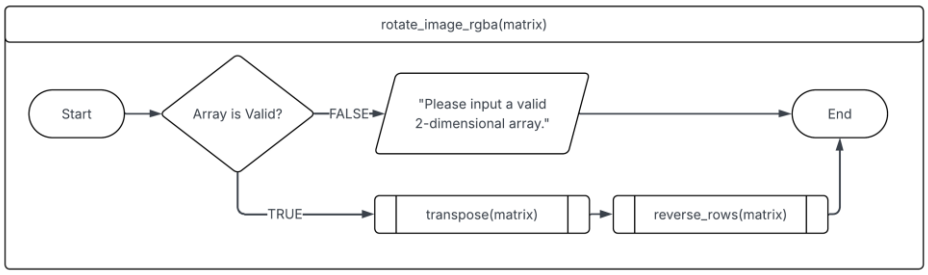


Figure 3

Transposing the values.

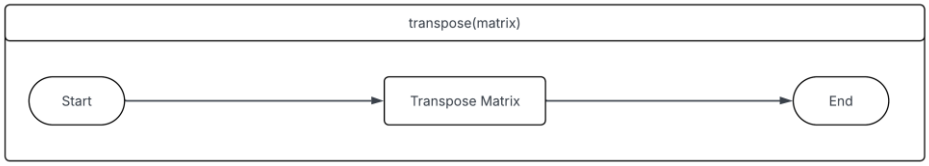
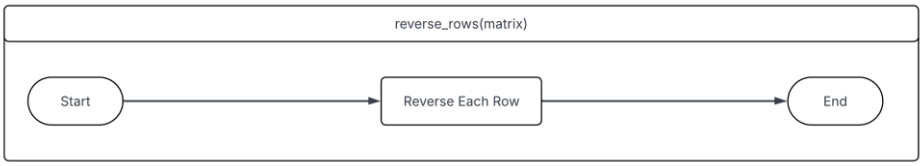


Figure 4

Reversing the rows.



Evaluation and Critique

The 2-dimensional 32-bit RGBA image rotation program separates the required functionality into three functions, “transpose,” “reverse_rows,” and “rotate_image_rgba” where each performs the desired task. The functions create a modularization of each respective task and thus abstract the logic. This algorithm, in general, is correct by its construction. The transpose function flips the matrix on its main diagonal, and hence rows are converted to columns, then “reverse_rows” mirrors the new rows to formally complete the 90-degree clockwise rotation. A Boolean condition exists within the “rotate_image_rgba” function to check if the input is valid. However, this is a very high-level algorithm; no GUI has been engineered, and all activity has been assumed in a Jupyter Notebook environment.

Commented [JL8]: Strong evaluation. You might want to discuss performance considerations or memory use, especially for large images.

Conclusion

It is clear that software development is both a technical discipline and a theoretical craft. Whether through applying design patterns to simplify complex architectures, using binary to encode and transmit data, or adopting program synthesis to assist or automate the development process, theory consistently reinforces programming and its practice. Though not all programs explicitly use these theoretical frameworks, their application only strengthens the foundation of programming and computing.

References

Alzubi, T. M., Alzubi, J. A., Singh, A., Alzubi, O. A., & Subramanian, M. (2025). A Multimodal Human-Computer Interaction for Smart Learning System. *International Journal of Human-Computer Interaction*, *41*(3), 1718–1728. https://doi.org/10.1080/10447318.2023.2206758

Brborich, W., Oscullo, B., Lascano, J. E., & Clyde, S. (2020). An Observational Study on the Maintainability Characteristics of the Procedural and Object-Oriented Programming Paradigms. *2020 IEEE 32nd Conference on Software Engineering Education and Training (CSEE&T), Software Engineering Education and Training (CSEE&T), 2020 IEEE 32nd Conference On*, 1–10. https://doi.org/10.1109/CSEET49119.2020.9206213

David, C., & Kroening, D. (2017). Program synthesis: Challenges and opportunities. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, *375*(2104), 20150403. https://doi.org/10.1098/rsta.2015.0403

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.

Nabi, T., Sweeney, K. M. D., Lichlyter, S., Piorkowski, D., Scaffidi, C., Burnett, M., & Fleming, S. D. (2016). Putting information foraging theory to work: Community-based design patterns for programming tools. *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium On*, 129–133. https://doi.org/10.1109/VLHCC.2016.7739675

Supekar, V., & Khande, R. (2024). Improving Software Engineering Practices: AI-Driven Adoption of Design Patterns. *2024 Second International Conference on Advanced Computing & Communication Technologies (ICACCTech), Advanced Computing & Communication Technologies (ICACCTech), 2024 Second International Conference on, ICACCTECH*, 768–774. https://doi.org/10.1109/ICACCTech65084.2024.00128