Applying Recursion and Fractals As Design Patterns In Complex Systems

Steve Mazza
Naval Postgraduate School

The term paper must be at least ten pages in length (single-spaced, normal font size, graphics are included in page count). There is no maximum length, although a concise, well-crafted analysis is clearly superior to a lengthy, rambling narrative. The subject of the term paper will be the student's synthesis of how some aspect of complexity, chaos, and other related topics relates to engineering of complex systems. That is, how does each student think at least one of these topics can be integrated into a general approach to systems engineering. The term paper will be due in class during Week 12.

Introduction

I will show how applying recursive design patterns to complex systems reduces the complexity of their description. Furthermore, patterns for the recursive descriptions can be found in some classes of fractals. There exists an analog between certain fractals and the recursive descriptions for complex problems.

Background

The paper that follows is predicated on developing a working understanding of complexity in the context in which we intend. Furthermore, we arrive at a common understanding of how to describe complex systems. And lastly, we introduce the idea of recursion and show how it is able to simplify our description of systems (or problems) in many instances.

What Is Complexity?

What is complexity? What is the character that makes some systems complex and others not? Is there a hard line between complexity and non-complexity or do we just know it when we see it?[1]

At the root, problems are complex when they are difficult to understand or predict. Murray Gell-Mann proposed that systems should be considered complex when the difficulty in predicting their outcome is due, not to irregularities (or randomness), but to regularities in the system that are difficult to describe. This, he suggests, differentiates between purely random outcomes which are not necessarily complex and complex systems whose outcomes are difficult to predict. (Gell-Mann, 1995)

Miller counters that there should possibly not be any single unified definition of complexity, arguing that trying to unify them may be asking too much.

> Complexity can occur at many levels, including time, space, and interactions. Perhaps we are expecting too much if we want a single measure of complexity that captures all of our intuitions. (Miller, 2009, page 234)

Miller is, of course not wrong. But despite the fact that there are several ways in which we can think about complexity, we will still benefit from a working definition. So we will adopt the definition proposed by Mitchell in which, "large networks of components with no central control and simple rules of operation give rise to complex collective behavior, sophisticated information processing, and adaptation via learning or evolution." (Mitchell, 2009, page 13)

Many researchers will agree that it is the interactions among the moving parts of a system that are instrumental in giving rise to complexity. Langford characterizes interaction as, "the transfer of something from one object (sender) to another object (receiver)." (Langford, 2012, page 48) This is a transfer of information which can be used to influence behavior or decisions or to affect internal change (i.e., state change). Axelrod and Cohen argue that it is interactions that, "make a Complex Adaptive System come alive." (Axelrod & Cohen, 2000, page 63) They devote an entire chapter to their importance, arguing that interactions afford opportunities for information exchange, without which system complexity is significantly reduced.

The concept of complexity might be best illustrated with a counter-example. At first glance, one might consider a mechanical wrist watch to be a highly complex instrument. In fact, it is not complex at all; merely complicated. The interactions among its parts is so highly constrained (preordained) that complexity cannot arise. Robustness, a hallmark of complexity, is missing entirely. If one were to remove a piece of the instrumentation (e.g., a gear or spring) the wrist watch would cease to operate.

[1] This is a clear reference to Justice Stewart's description of pornography. The point is, can complexity be definitely identified (as in the legal sense) or is its character more elusive?

How to Describe Complex Systems

Formal methods are the basis by which we achieve repeatability in engineering. Without them we are doomed to fumble around in the dark, revisiting the same mistakes on successive efforts. Frameworks such as the Department of Defense Architecture Framework (DoDAF) and The Open Group Architecture Framework (TOGAF).

These provide the basis for structured, disciplined engineering and help guide and support the full life cycle of development. They help us to manage and understand our engineering design problems within the context of the broader support for their development, use, and disposal but they do not contain tools that are specific to decomposing the understanding of and communicating the complexity at the root of our design challenges.

The design patterns that we use are tools that help us visualize, understand, and communicate complexity in a relatively standardized manner. They describe recurring problems along with the core of their solution. Inasmuch they provide an easy way of rapidly transferring large amounts of information about a problem in an accessible way. In order to keep pattens useful in a standardized way, the Gang of Four[2] describe four essential elements of any design pattern: the pattern name, the problem, the solution, and the consequences. (Gamma, Helm, Johnson, & Vlissides, 1994, page 3)

Pattern Name. Providing a name for our pattern facilitates rapid communication of complex ideas by increasing our design vocabulary. Agreeing on a name means that engineers across many different problem domains can refer to the same problem and solution without ambiguity.

Problem. Very simply, the problem describes when and how we can apply the pattern. It provides the context for the pattern's use and may also set some conditions which must be met in order to do so. The problems to which patterns apply may occur at any level of design. The problem may be specific or may be descriptive of a whole class of design ideas.

Solution. The solution is sufficiently abstract such that it can be applied to a a range of implementations. The solution should describe the elements of the design and their interactions.[3] It may also describe the relationships and responsibilities of elements within the design.

Consequences. These are the critical basis for evaluating design decisions and involve trade offs such as flexibility, adaptability, speed, cost, and time. They are critical for choosing the correct alternatives when using patterns. Because consequences are not always listed explicitly within the description of a pattern, a thorough understanding of that pattern is necessary in order to evaluate its use.

The use of design patterns in software engineering is fairly ubiquitous. Design patterns in other disciplines varies, but is potentially equally as applicable. The emerging field of systems engineering has many analogs in software engineering and, consequently, may benefit equally from the establishment of a set of commonly accepted design patterns as the basis for a design vocabulary.

Design patterns provide a way of simplifying difficult problems by packaging the problem, solution, and consequences into a neat package and providing a name (a handle) by which to refer to it. This is a mechanism of abstraction and provides us with a very powerful tool for helping us deal with difficult and complex designs.

Methods Of Simplifying Complexity

There are two tools in our arsenal that gain us a lot of ground toward simplifying complexity. They are abstraction and encapsulation and we introduce and discuss them here, first individually, and then show how we can put them together to provide a powerful way of thinking about difficult and complex problems. The distinction between abstraction and encapsulation requires some working out since they are almost always used in conjunction.

Abstraction. Abstraction is the mechanism by which an interface to a potentially complex entity is simplified. The term *abstract* used as an adjective means existing in thought or as an idea but not having a physical or concrete existence. This separation between the idea (or essence) of something and its implementation gets at the core of why we care about abstraction at all.

It is very nice when trying to simplify a difficult problem to just concern ourselves with the interface, or contract, (see below) or even an idealized interface (at first) in order to gain an appropriate conceptual foothold on the complexity. Since abstraction refers to showing only the necessary details of an interface to the consumer, it also provides us a mechanism by which we can ratchet up the complexity as our understanding of the problem space increases.

As an example, consider the act of shipping a parcel to a relative in another state. Your responsibility as the consumer begins and ends with providing a valid shipping and return address and making payment. You need not concern yourself with the underlying implementation of *by what means* the package will arrive to your relative. It could be conveyed via ground freight or travel by air or even be strapped to a donkey. Assuming it arrives on time and in tact, you are afforded the luxury of not caring about the details... the implementation of its delivery. If the shipper invented a teleportation machine, you would be none the wiser.[4]. The idea of *shipping a parcel* is an abstraction on the mechanism of actually getting it to its destination.

[2] This is a commonly used reference to Gamma, Helm, Johnson, and Vissides and to their 1994 treatise on design patterns.

[3] Recall the importance of interactions in complex systems.

[4] Although one would assume that the rates would go up in order to cover the research and development

Encapsulation. In contrast, encapsulation hides data as well as complexity. However the emphasis is not on the interface, but on providing *data protection.* Encapsulation literally means *to surround.* While the complexity of an entity is a function of its implementation, it is not the implementation details that are important to encapsulation, rather the data and complexity. Protection in this case often refers to the prevention of access of data by users in unintended ways. The mechanism of encapsulation enforces integrity of the type, entity, or object.

Proper encapsulation enforces the contract with the user regarding how an entity or artifact will respond to interaction with its interface. Abstraction is the simplification of the interface; encapsulation is the hiding and protection of the data behind the interface. It controls the means by which the artifact may be used. Unlike abstraction, encapsulation often refers to a specific concrete implementation.

Consider the example of my morning coffee.[5] Since the installation of the Kureg machine in our break room, the production of coffee requires me only to select the type of coffee I want, insert the capsule into the machine, and press the large *Brew* button on the front. I need not care about how much water is required,[6] the temperature of that water, how to grind the coffee, or how long to steep the grounds. Fortunately for me, the button on the face of the machine is an encapsulation of how to brew coffee.

Putting Them Together. Abstraction and encapsulation are natural partners in simplifying our understanding of complexity. Taken together, they afford us a simultaneous simplification of the interface (i.e., contract with the user) and a protection of the implementation details. This allows us to focus on the *what* and temporarily ignore the *how* of an entity or artifact.

In a more practical manner, the use of abstraction and encapsulation together allows the distributed design and development of a solution to a complex problem in a way that accommodates the natural development life cycles of individual component parts to progress at their own pace and schedule. So long as the interface contract (abstraction) is maintained, the implementation details (encapsulation) are free to evolve as appropriate.

This leads us to a concept that is often referred to as *black box* architecture. It is a powerful mechanism for dealing with complexity and complex systems. Design patterns are used by engineers to create an understanding of difficult systems (or problems) and to create a conceptual framework around that problem space that is descriptive of a solution. The design patterns usually suggest natural divisions where interfaces can be created but remain agnostic of the details of implementation, leading to the black box architecture. Thus abstraction and encapsulation naturally flow from the effective use of design patterns, making them well suited to reducing complexity.

Introducing Recursion

The joke goes that in order to understand recursion[7], you must understand recursion.

Definition of Recursion.

> "A recursive process is one in which objects are defined in terms of other objects of the same type." (*http://mathworld.wolfram.com/Recursion.html*, 2014)

Very often the object of the same type is actually an exact copy of the first instance of the object but with a slightly different value, as in the case of a recursive algorithm in computer science. Another instance of a given function gets pushed on the call stack but is passed a different parameter.

The essence of recursion is that you may solve a large or difficult problem by first solving smaller versions of that same problem and applying those results to the larger problem. Developing a recursive solution to a problem hinges on identifying a base case so that you know where to start or end.

Recursion in the Classroom. This may be easiest to understand if we walk through an example. We will first select an algorithmic example so that we can keep our discussion tidy.

A classic example of recursion can be shown in the implementation of the factorial algorithm, which calculates $n!$ where n is some positive integer. The factorial of n is the product of n and all the positive integers below it such that

$$n! \equiv n(n-1)\cdots 2 \cdot 1$$

For example, $3! = 3 \times 2 \times 1 = 6$.

Implementing this algorithm in the C programming language, we have

```
unsigned int iter_factorial(int n) {
  int f = 1;
  int i;
  for(i = 1; i <= n; i++) {
    f *= i;
  }
  return f;
}
```

The implementation of this algorithm takes the integer argument n and uses the loop variable i to count from 1 to n, multiplying and collecting the loop variable in f.

Notice how much more streamlined the implementation and subsequent description are under a recursive version of the algorithm.

[5] The author presents this statement as if there is only one morning coffee. Subscription to this tenant is pure folly.

[6] The Kureg machine in our break room has a water line from the sink water supply so there is no reservoir to fill.

[7] See Recursion

```
unsigned int recursive_factorial(int n) {
  return n>=1 ? n * recr_factorial(n-1) : 1;
}
```

In this version, we are either multiplying n by the previous result of the function call or returning the value 1. The ternary operator looks menacing if you have not seen it, but what is happening (from left to right) is that n is compared against 1 and while it is still greater than or equal to 1, another copy of the function is pushed onto the call stack with the new value $n - 1$. In this way the values passed to the successive copies of the function march inexorably toward the terminal state, 1, at which point the final value is eventually returned.

Notice in this construction, we are calling the function from within the function, itself.[8]

Recursion in the Real World. While we will discuss the applicability of different proposed patterns later, we present here a common example in the interest of exposition. Consider the task of processing a bill-of-materials, as in the case of manufacturing. The cost of a manufactured assembly (component) is the sum of the cost of component parts plus any associated labor and other overhead. However, the component parts themselves may be manufactured assemblies. For example, an electric drill, made up of a motor, chuck, bearings, housing, switch, and a cord. A motor is made up of a frame, stator, rotor, and bearings. A rotor is made up of a shaft, stampings, wire, etcetera.

There are many examples of business processes that can be naturally expressed recursively. Organizational charts, directories, and call trees are some. The underlying implementations on which enterprise relational database management systems (RDBMS) are constructed are self-similar. While B-Trees and B+Trees[9] are overly simple for robust commercial RDBMS, they are exemplary of recursive constructs in the real world.

Fractals

Introduction

Introduce several classes of fractals (with figures).
1. Koch snowflake
2. Sieve

Consult Benoit Mandelbrot for examples and explanation.

Constructing Fractals

Describe and demonstrate their construction, emphasizing self-similarity and recursion.

Self-similarity and Fractals

Discuss self-similarity and its connection to fractals. Discuss how we can describe fractals at all levels of magnification by describing their construction at one given level. Show how simple rules can lead to complexity.

Design Patterns

Introduce the notion of using design patterns to describe solutions to problems. Point out that traditional patterns don't adequately encode recursion. Propose the use of some classes of fractals in describing recursive solutions to complex problems.

Use Of Design Patterns

Introduce the history and use of design patterns. (Gamma et al., 1994)

Shortcomings Of Traditional Design Patterns

Show the shortcomings of traditional design patterns in modeling complex systems.

http://sob.apotheon.org/?p=1814

> IMHO, the alleged "complexity" (or fragility) of many business programs is directly due to the attempt to solve the problem with non-recursive thinking.

Use of Fractals as Design Patterns

Propose the use of some classes of fractals as design patterns.

Example

Work through an example (with figures)

From Simplicity, Complexity

Discuss how simple rules can quickly lead to complexity.

Conclusion

Wrap up. Summarize what was covered. Restate the proposal to use certain classes of fractals to describe recursive solutions to complex problems.
1. Summarize and wrap up.
2. Restate the proposal

[8] Within a typical computing environment this is possible because each successive call to the function pushes the previous calls one level deeper on the stack and so variable name space is preserved.

[9] These are simpler constructs which are often introduced in undergraduate computer science curricula.

References

Axelrod, R. M., & Cohen, M. D. (2000). *Harnessing complexity: Organizational implications of a scientific frontier*. Basic Books.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Pearson Education.

Gell-Mann, M. (1995). What is complexity? remarks on simplicity and complexity by the nobel prize-winning author of the quark and the jaguar. *Complexity*, *1*, 16-19.

(2014, June). Retrieved June 13, 2014, from `http://mathworld.wolfram.com/Recursion.html`

Langford, G. O. (2012). *Engineering systems integration: Theory, metrics, and methods*. CRC Press.

Miller, . P. S. E., J. H. (2009). *Complex adaptive systems: An introduction to computational models of social life: An introduction to computational models of social life*. Princeton University Press.

Mitchell, M. (2009). *Complexity: A guided tour*. Oxford University Press.