The Chaos Model and the Chaos Life Cycle

L. B. S. Raccoon

INTRODUCTION

I believe that to truly understand software development, we must not only understand the flow of an entire project and how to write each line of code, we must also understand how one line of code relates to the whole project. It seems to me that we have studied each aspect of software development in isolation, not how all aspects fit together. The Waterfall model, defined by Royce, and the Spiral model, defined by Boehm, discuss management-level issues, such as phases and deadlines, rather than how to write one line of code or fix one bug. Programming methodologies show us how to solve technical problems, rather than how to solve users' problems or to meet deadlines. In this paper, I use the principles of chaos (or fractals) as a metaphor to bridge the gap in our understanding of the relationship between one line of code and the entire project.

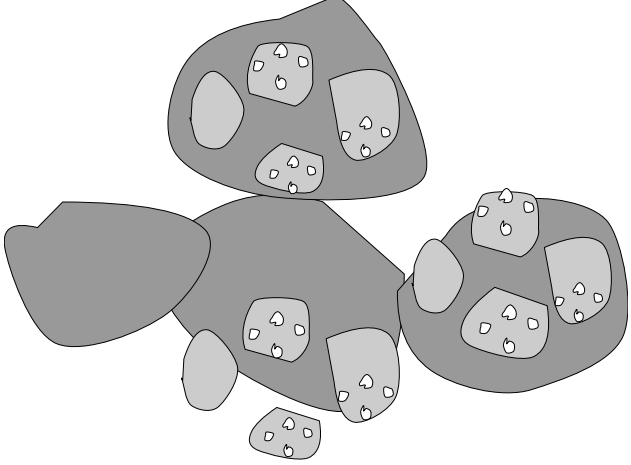
Throughout this paper, I describe software development from the developer's point of view. If we want to understand software development, we must describe what developers do. After all, developers do the work. We know that large programs consist of many lines of code and that large projects consist of the daily efforts made by individual developers. We know that the large scale and the small scale somehow relate to each other. Yet most models of software development seem to focus on one extreme or another, ignoring the role of developers.

In the first section, I define the Chaos model which combines a simple, people-oriented, problem-solving loop with fractals to describe the structures within a project. I believe that software development is a human activity: people write the software, use the solutions, and experience the problems. I believe that creating software is very complex; we cannot simplify software development by imposing simple models on it. The Chaos model uses fractals to describe a cohesive structure which encompasses many of the issues actually encountered during software development. This structure helps to explain the influences within a project and the roles that developers play.

In the second section, I interpret the Chaos model to reveal the meaning behind the structure. I show that users, developers, and technologies form a continuum throughout software development. They all interact in a complex dance. This interpretation improves our understanding of the contribution and limitations of users, developers, and technologies.

In the third section, I define the Chaos life cycle to describe how a project evolves over time. Life cycles are essentially the top-level perspectives of software development. In light of the Chaos model, I define the phases of the life cycle in terms of fractals and show that all phases occur throughout the life cycle. These chaotic definitions suggest that I can interpret the complete life cycle in terms of each phase, and conversely, I can interpret each phase in terms of a complete life cycle. The phases of the life cycle show our perspectives on the state of a project, rather than what the state of a project really is. Thus, developers need many skills to be able to understand

and respond to situations that arise throughout a software development project.



THE CHAOS MODEL

The Chaos model combines a linear problem-solving loop with fractals to describe the complexity of software development. The linear problem-solving loop involves four different stages: problem definition, technical development, solution integration, and status quo. Fractals describe the structure between different parts of a project. The Chaos model differs from other models in that it imposes little organization on the development process, rather, it allows many organizations to evolve. This allows the Chaos model to apply in many complex situations.

The structure of a simple problem is different from the structure of a more complex problem. In general, we break complex problems into simpler subproblems. We use this reductionist approach to deal with problems that are too large to handle otherwise. Yet, stating that the structure is recursive is too simple. What is the relationship between the different components? Fractals require that the subproblems of any one problem have approximately the same size and value. By arguing that good solutions must have a fractal structure, I show several relationships within projects, as well as explain some of the complexity that we encounter in real projects.

The Chaos model describes a flexible structure which reflects the intricate patterns that occur in real projects. The chaotic patterns between the levels of a project explain the complexity of software development. Software development is a continuum from the whole project down to each line of code and involves both human and technical issues on all levels.

The Linear Problem-Solving Loop

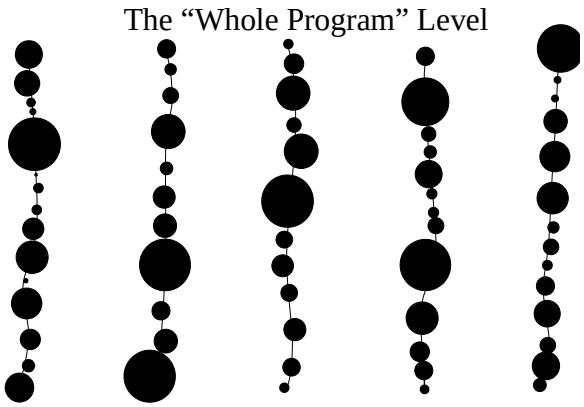
The Linear Problem-Solving Loop, shown in Figure 1, has four stages: status quo, problem definition, technical development, and solution integration. Each trip around the loop begins with a Status Quo and returns to a new Status Quo. This loop shows the difference between solving problems in the big sense and solving problems in the little sense. Technical development is only one aspect of problem solving. The problem-definition and solution-integration stages emphasize the human aspects of software development.

During **problem definition**, developers choose a specific problem to solve and determine its solution constraints. Sometimes users know exactly what they want and sometimes users are frustrated and have no idea what they want. Solving their problems may or may not be possible. But the ability of people to describe their problems is independent of whether their problems should be solved. If the problem is to port a program to a new platform, the problem definition can be very simple. If the problem requires development of a new program to solve a new application or use new technology, the problem definition can be very complex. Developers must decide whether a new user interface would suffice, what to do about compatibility with previous systems, and where the system should be in five and ten years. Sometimes politics plays a big role, especially when a decision requires a consensus among many people. In this case, bad politics can sink a good solution. There are often many reasonable solutions and no easy way to choose among them.

During **technical development**, developers do the work. If a problem can be solved by technical means, developers will solve it. Professionals use technical tools and methodologies when appropriate. But developers cannot use technology to solve nontechnical problems nor can they ensure that the right problem gets solved nor that a good solution will get used. Inappropriate use of technology is destined to fail. Placing technical development in the context of the linear problem-solving loop emphasizes these limitations.

During **solution integration**, programmers incorporate the results of technical development into the world at large. Integration involves advertising, selling, and disseminating the solution to users. Users reject or ignore many results because they find that the wrong problem was solved or the improvement is inadequate. Graphical user interfaces languished for years, mostly due to nontechnical concerns, such as cost and compatibility.

The **status quo** represents the current state of affairs in the world. This includes the technical state-of-the-art, as well as the economic and social circumstances of the participants. As the new technical solution gains acceptance, a new status quo emerges and the cycle repeats. Depicting the loop, rather than the stages of problem solving, points out that problem solving improves the status quo. Figure 2 shows that the linear problem-solving loop has no starting or stopping points. The status quos are points of stability within a stream of change.



The Fractal Problem-Solving Loop

The linear problem-solving loop applies on many levels of a project. It applies to entire projects, as well as subgroups, and individual developers. Large problems are made of many smaller subproblems. The obvious fractal generalization of the linear problem-solving loop is shown in Figure 3. However, Figure 4 depicts the erratic patterns that occur in real projects.

Fractals imply a very specific scaling relationship in the recursion. The same expansion applies to each level. All levels of software development have the same value to the project as a whole. Each level is composed of all of the levels below it and so each level repeats the structure of the next level down. The top levels deal with a few large issues, the middle levels deal with more mid-size issues, and the bottom levels deal with many small issues. The total impact — the sum of the impacts of each issue — is the same on each level.

The linear problem-solving loop also applies to levels above and below the project boundaries. Above the project level, companies embark on projects to gain strategic advantages and make money. Companies solve economic and productivity problems. Below the programming level, individuals solve various psychological and physiological problems. The brain is a compact neural problem solver, addressing many issues per second. While these extreme ranges may represent valid problem solving, they go beyond the scope of software development per se, so I will set cutoffs at the project boundaries as suggested by Mandelbrot.

Example: Consider a company which wants to forecast and manage its growth. If the company predicts that it will outgrow its internal accounting system and it decides to develop a new accounting program, these are a few of the many distinct levels the project contains:

- At the **program level**, the company must handle projected growth. The technical solution is to create a new program. Integration involves putting the system on line and educating the users.
- At the **component level**, a team creates the user interface. During problem definition, the team members define how the interface will work and how each member will contribute. During technical development, they create the user interface. Integration involves selling the finished code back to the whole project through code reviews, code integration, and system testing.
- At the **function level**, developers work on their own parts of the project. A developer first decides what function to create. During technical development, he or she edits and compiles the function. During solution integration, he or she tests and integrates the results.
- At the **“one line of code” level**, a developer decides which line of code to edit next and why. Technically, he or she makes changes using an editor and then decides whether the result looks correct and elegant enough to keep.

Influences within a Project

Levels are not independent. All levels of a project are connected by a web of influences that stretches between the “whole program” level and the “one line of code” level. Adjacent levels influence each other very strongly. Distant levels influence each other very weakly. For example, the meaning of a function depends very much on the exact position and meaning of each line of code. Yet the meaning of a program depends very little on the position and meaning of any one line of code. One line of code changes or disappears completely, depending on the algorithms and data structures used and the developer’s programming style.

To understand software development effectively, we must distinguish between those issues that we both can and want to control and those issues that we either cannot or do not want to control. The state of a project is the subset of the status quo of interest to developers. The distinction between the state of a project and the status quo often seems arbitrary. In Figures 5, 6, 7, and 8 the states of a project are represented as vertical strings of dots. Each dot represents a different logical level of the project state. The dots oversimplify the depiction of the levels, which are multidimensional and quite complicated.

The state of a project separates naturally into distinct conceptual levels. The top level represents the concept of what the whole program should accomplish. The bottom level represents what we can accomplish with one line of code. The levels in between represent what we can accomplish using mid-level structures, such as

functions and modules. The middle levels do not refer to the physical structure of the project, the call structure of the code, the inheritance structure of the code, the size of the code, or any other explicit structure in the code. Code structures often mimic the conceptual structures, but seldom exactly. While many conceptual components are implemented as separate functions in separate files, some conceptual components may be spread across many different functions in many different files.

Software development is the flow from one project state to the next. Normally, each level of a project proceeds in loose coordination with the other levels, with no single level dominating or controlling the whole project. Changes can occur at any level during a state transition and one change can affect many other levels.

Software Development is a Human Activity

Software development is much more than technical development. The linear problem-solving loop shows that many nontechnical issues affect software development and the fractal problem-solving loop points out that these nontechnical issues affect all levels of a project. Thus, things can go wrong in the problem-definition and solution-integration stages of problem-solving loop on any level, from the whole program level down to the "one line of code" level.

Developers can solve the wrong problem. Developers can mistakenly identify the problem or ignore the problem definition and solve a different problem. Developers can fail to sell and support their technical solutions. And good technical solutions can be rejected because they don't satisfy the ulterior objectives of a group. These problems occur when developers fail to work with other people effectively. Developers within a group may not agree on the goals of the project or the scope of the solution. Developers may miscommunicate with users, which prevents them from understanding what the program should accomplish. And a developer may miscommunicate with his or her co-workers, which prevents him or her from solving the right piece of the problem.

Developers can misuse technology. They can over-engineer or under-engineer a solution which either wastes resources or doesn't solve the problem. Some developers insist on using state-of-the-art technology, instead of more appropriate conventional technology. And some developers refuse to use technology that they didn't invent themselves, the NIH syndrome. We don't normally worry about these problems because we assume that developers don't make technical mistakes.

Many developers seem to assume that cooperation within a group is easy and that conflicts within a project can be resolved by technical authority or leadership or some new technology. But tools and methodologies cannot overcome nontechnical problems.

All of these problems have a strong human element. Unfortunately, people problems are not normally considered a development issue. Yet, I think that these human issues cause as much or more trouble than technical issues. These problems must be resolved by direct communication and cooperation between everyone involved. Good management can mitigate some of these problems, but probably cannot eliminate them. Given the hands-off attitude many managers take, nontechnical problems often go unaddressed.

AN INTERPRETATION OF THE CHAOS MODEL

In this section, I show how we can interpret the Chaos model to reveal the meanings and purposes behind software development. Describing the structure within a project is fine, but how does the project relate to users and technologies? The Chaos model defines a structure on which we can hang these concepts.

We can reinterpret the meaning of the "whole program" level and the "one line of code" level in terms of users and technologies. The "whole program" level represents the users' needs or the goals of the project. The goals of the project are defined by the users at the top level, so the goals must trickle down to the bottom level. The "one line of code" level represents our technical resources or the smallest pieces of the solution. Developers write code one line at a time using established techniques on the bottom level, so the solutions must trickle up to the top level. In the middle levels, developers match up the users' needs with the technical resources to satisfy them.

This interpretation of the Chaos model shows that software development is a continuum from the user to the developer to the technology. We need to understand how users, developers, and technologies contribute to a project to understand their possibilities and their limitations.

Users' Needs

Software is valuable because it helps businessmen to manage finances, helps students to write reports, helps

publishers to lay out graphics, and helps workers to run factories. The goal of software development is to improve users’ productivity. Application software makes the general public more productive. Systems software makes other developers more productive.

The top levels of the project are defined by the needs of the user and tend to be outside of the developers’ realm of control. While developers do help users understand their needs, developers do not determine what the users need. Note that the users’ needs can change over the course of a project.

Users depend on developers to create robust and effective programs to meet their needs. Users would like to decide what a program should do, but they cannot make all decisions which matter. If users could make these decisions, they would write their own software. But, few users possess the development skills necessary to adequately specify, implement, and maintain a large program. Software developers have the skills and resources to decipher the users’ needs and create effective technical solutions.

Unfortunately, many traditional models decouple software development from the users during requirements analysis, and then they ignore the users during the rest of development. Developers become responsible to the specification and are discouraged from interacting with users. If the specification is inadequate or changes arise, the users have little recourse and the decoupling prevents anyone from clearing up the problem. Developers do currently recognize the importance of user participation during User Interface design. However, user participation is still not considered part of normal software development.

Developers must understand the application and the needs of users. It often takes a long-term relationship between users and developers, with a lot of dialog, to uncover what users really need. If the needs of users change, we must accept those changes. If users are slow to understand their needs, then developers must wait for users to clarify their needs. As software engineering becomes more applications-oriented, developers must work more effectively with users. Solving the wrong problem is very unproductive for both users and developers.

Technical Resources

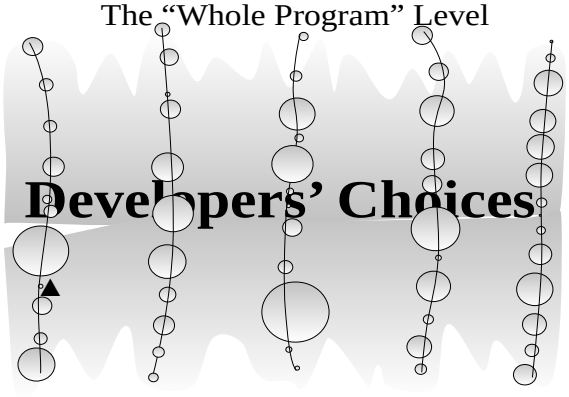
Technology is the collection of tools and methodologies that enable developers to solve specific technical problems. We use technology to build programs for nontechnical users. Most tools and methodologies apply only to the lowest levels of a project. At the bottom levels, editors help developers to change one line of code at a time and debuggers help developers to watch one line of code execute.

The bottom levels tend to be outside of the developers’ realm of control. The bottom levels are defined by what we can achieve with one line of code. For example, programming languages determine how functions are built, but not how programs should be constructed for the user. Note that the technical resources can change over the course of a project.

Software developers often seem to think that technology can do anything, that technology suffuses our environment. In fact, when problems easily reduce to technical solutions, developers get predictable results. But, developers must construct everything that technology does not implement directly. Most of what developers do, at the middle and upper levels of a project, remains unsupported by technology. In the middle levels, “make” allows developers to update a project with one command and revision control systems allow developers to manage groups of changes with one command. Libraries and class hierarchies fit between the bottom and middle levels. One function call or object invocation can stand for one idea, which is a bottom-level interpretation, or can stand for a lot of code, which is a mid-level interpretation.

Few tools apply to the middle and upper levels of a project. Most high-level tools, such as formal specification languages and test case generators, are really projects, not tools. Formal specifications and test suites are usually many lines long. Thus these are not technologies; they are complete development projects in their own right. These projects can help the whole software development project, but they are not part of the finished program. Developers creating formal specifications and automatic test systems use most of the same technologies that other developers use: editors, compilers of various sorts, “make,” and revision control systems.

While developers do use the programming technologies, we are not responsible for creating new programming technologies. Viewing technical resources as part of a chaotic structure reveals that developers need tools and methodologies to work on higher and higher levels of the project.



Developers Solve Mid-Level Problems

Developers work on all levels of a project, but spend most of their time working on the middle levels. In the middle, developers match the pieces of a problem with chunks of code. The problems are small enough to be solved and the solutions are big enough to be useful. Every level of the project, every size of component, and every scope of decision is caught in the web of influences stretching between the users’ needs and the technical resources available to satisfy the users’ needs. Because the needs of the users strongly influence the upper levels of the project and the technical resources strongly influence the lower levels of a project, developers have the most influence in the middle levels.

To solve a mid-level problem, developers must isolate a solvable problem from the rest of the project. They must distinguish between the issues that are important to the current problem and those that are unimportant. They must then consider the lower-level issues of how to technically implement the solution, and the higher-level issues of what their solution means to the user. They must consider enough information that the result is meaningful and the solution is achievable. Figure 8 shows how developers might work on several adjacent levels at once to solve one mid-level problem.

Conflicts between levels occur all the time and are very normal. Decisions made on one level often seem wrong on other levels. Changes on one level can delay or undo progress on other levels. For example, building program scaffolding can help to complete certain components; but it also adds extra work to other components. Sometimes conflicts between levels show that one or more issues are being handled on the wrong level. High-level decisions can reveal unreasonable assumptions and be too difficult to implement. Low-level decisions can overly constrain a solution and lead to bad functionality or useless code. The right-level decision avoids these problems.

Matching complex problems to complex solutions is very difficult. Consider the Traveling Salesperson Problem. From a given city, we cannot efficiently determine which city to visit next without knowing the entire tour. The fact that software development is much more complicated than the Traveling Salesperson Problem explains why strict top-down and bottom-up approaches to software development do not work. In programming, we cannot efficiently determine the best answer to a problem on any one level, unless we know the answer to all problems on all levels. Developers must make reasonable tradeoffs.

THE CHAOS LIFE CYCLE

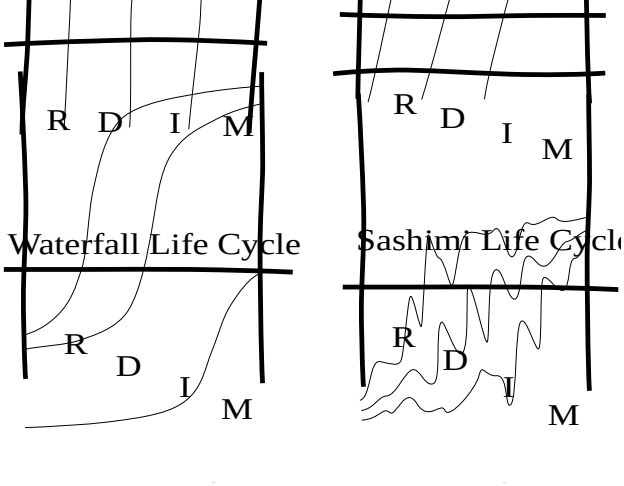
In this section, I introduce the Chaos life cycle which views requirements analysis, design, implementation, maintenance, and prototyping — the phases of the life cycle — in terms of fractals. I include prototypes in this list because prototypes are identifiable parts of a software development project. A life cycle shows how phases change over the course of a project. While we usually view these phases as flexible concepts, defining them as fractals leads to some refinements and generalities that we normally overlook and shows that all phases occur throughout all of software development.

I use fractal phase definitions to show that the phases resemble each other. I show that we can view the whole life cycle in terms of each phase and view each phase in terms of the whole life cycle. By transitivity, the phases are essentially identical. So, our assessment of where a project is shows our perspective about the project, rather than any essential truth about the project.

Developers use all skills throughout a project. Developers often get caught up in small problems and forget how they relate to other problems that come before and after, or above and below the focus of attention. Developers need to keep the whole flow of software development in mind.

A Note on Life Cycles

A life cycle is not a model. To greatly simplify the distinction between them, a life cycle depicts the sequence of events within a project, while a model depicts the structure within a project. Figure 9 shows one set of variations on the Waterfall and Sashimi life cycles (as defined by DeGrace and Stahl) and the Regression and Chaos life cycles (as defined here). These life cycles depict four different ways that programming flows from requirements analysis to design to implementation to maintenance. Each life cycle differs in how the phases change throughout



the project.

Simple life cycles help us understand projects by emphasizing simple parts of software development and ignoring complex, obscure details. The Waterfall life cycle depicts software development as a fixed sequence of distinct phases. The Sashimi life cycle allows phases to overlap, though not to get mixed up. These first two life cycles suggest that projects follow rigid patterns. They suggest that to write the best program possible, developers should plan to meet specific schedules, and that changes and problems that unexpectedly arise probably reflect bad planning.

Complex life cycles help us understand the more sophisticated facets of projects by revealing complexity. The Regression life cycle allows each phase to linger throughout the entire project. Maintenance starts on “day one” and requirements analysis finishes at the end of the project. The Chaos life cycle breaks the rigid flow of phases and allows them to come and go as the project evolves. Problems and changes may surface and then get resolved. These last two models suggest that projects are adaptable. They suggest that planning and scheduling can only be so useful, and that to write the best program possible, developers must respond to the various problems and opportunities that will undoubtedly arise.

Fractal Phase Definitions

The normal definitions of life cycle phases focus on the top, project-level concepts. In light of the Chaos model, I generalize these phase definitions and show how they apply to all scales, from large to small, between the users’ needs and the technical resources. The following definitions can refer to the activities on any one scale, on all scales together, or on any subset of scales. I define five representative terms here, leaving the definitions of specification, testing, and other life cycle concepts to the reader.

Fractal Phase Definitions
Requirements Analysis: To analyze requirements commonly means to identify what the user needs or to define the goals of the project. The resulting specifications are documents of varying degrees of formality (from sticky note reminders to user manuals) which are written in languages using varying degrees of precision (from shorthand English to formal specification languages). By treating a to-do list written on a note card as an informal specification, we can see that requirements analysis occurs throughout all levels of a project. ● At the top levels, we identify what the users want the program to do. ● In the upper levels, we identify what a program needs the components to do. ● In the lower levels, we identify what a component needs lines of code to do. ● At the bottom levels, we identify what a line of code needs the compiler to do.
Design: To design means to plan ahead, to bridge goals and actions. This means a variety of things, from laying out the structure of the code, to establishing coding standards. Coding standards include plans for how lines of code should work together. Whenever developers think about how they will do something, before they do it, they are designing. ● At the top levels, we design how the program will work. ● In the middle levels, we design how components will work. ● At the bottom levels, we design how individual lines of code will work.
Implementation: To implement means to carry out or perform an action. Developers create pieces of code of various sizes and scopes. ● At the top levels, we implement projects by writing programs. ● In the middle levels, we implement pieces of the project by writing components. ● At the bottom levels, we implement pieces of components by writing lines of code.
Maintenance: To maintain means to fix or enhance a program. Developers could fix a small bug or rewrite the entire program. All changes that improve flexibility and generality of a program count as maintenance, too. Maintenance is more than just the dregs of the previous project. As the needs of the users change, the project must adapt. Developers normally think of maintaining entire projects, though they can also think of maintaining modules or lines of code. Developers fix and improve modules and lines of code throughout the project. ● At the top levels, we maintain entire programs. ● In the middle levels, we maintain components. ● At the bottom levels, we maintain lines of code.
Prototyping: Prototypes are experiments to determine how well a particular approach solves a problem. Prototypes can be created at any point during a project. Very often, developers simply try something. If the result works well enough, developers use the results to guide subsequent efforts, otherwise they look for another solution. When developers don't know how to solve a problem, prototypes can show them what a solution looks like. Prototypes can be built one after another, or several different prototypes can be built at the same time. Since prototypes are not final products, developers should try to maximize the useful information that results and minimize the resources invested. Prototypes help us understand all levels of a project. ● At the top levels, we prototype entire programs. ● In the middle levels, we prototype components. ● At the bottom levels, we prototype single lines.

Phase Means Perspective

I believe that all phases occur throughout software development and that we can view our project in terms of any phase we choose. The phases of the Chaos life cycle are essentially the same in at least two distinct ways: horizontal slices (or levels of a project) and fractal scaling (or sub phases). When we treat a problem as being in one phase, it shows our perspective on the project rather than any essential distinctions between phases of the life cycle. The belief that specification normally precedes design and design normally precedes implementation, is too simple. Specification, design, and implementation are fractal variations of the same thing. Software development is the same throughout: high to low, front to back, all scales, and all levels. Developers interleave design, implementation, and testing as they work on different pieces of the project. It is not possible to isolate these phases from each other.

The definitions of phases in the previous table show that all phases apply to every sized piece of code. On every level of the project: components must be specified, designed, and implemented. We can also interpret

horizontal slices in terms of levels. Every level acts as a specification for the level below it, an implementation for the level above it, and a design that connects the level above with the level below. Even the top level acts as an implementation of the user's needs and the bottom level acts as a specification to the compiler.

In terms of fractal scaling, each phase resembles the whole life cycle and the whole life cycle resembles each phase. The first column of the following table shows that each phase occurs in all phases. Each phase involves setting goals, carrying out the goals, and maintaining the results. The second column of the following table shows that the complete life cycle can be viewed in terms of a single phase. We can view all of software development in terms of design, in terms of implementation, or in terms of maintenance. To read the table, I find it helpful to keep the following three points in mind. First, the empty program is an infinitely buggy version of a real program. Anything will improve it. Second, the bulk of standards committee work involves maintaining, improving, and correcting the specification. For example, the C++ standard changes every three months as it is corrected and refined. Third, even the smallest bug must be identified, and therefore specified, before it can be fixed.

So by transitivity, each phase is identical to every other phase. The phases blend into each other and the life cycle dissolves into an amorphous flow of emphasis. The distinctions that we make between phases become arbitrary and show our perspective on the project, rather than any essential truth about software development. When we say that a project is in one phase or another, it shows where we think we are, more than where we actually are.

Each Phase Occurs in All Phases	Each Phase Is a Complete Life Cycle
Requirements analysis emphasizes deciding what to create. We analyze the requirements of: ● specification documents ● pieces of code ● enhancements and bug fixes	Requirements analyzers create documents which specify the program. We: ● plan how we will produce the specification ● implement the specification ● fix and enhance the specification
Design emphasizes planning how the code will work. We design: ● specification documents ● pieces of code ● enhancements and bug fixes	Designers create documents which describe how the code will work. We: ● plan how to create the design ● implement the design ● fix and enhance the design
Implementation emphasizes creating the program. We implement: ● specification documents ● pieces of code ● enhancements and bug fixes	Implementors create code. We: ● specify what code to implement ● design how the code will work ● implement the code ● fix and enhance the code
Maintenance emphasizes improving the program. We fix and improve: ● specification documents ● pieces of code ● enhancements and bug fixes	Maintainers create modifications of code. We: ● specify the correction in a bug report or change order ● design how to make the correction ● implement the correction ● fix and enhance the correction
Prototyping emphasizes learning the problem. We prototype: ● specification documents ● pieces of code ● enhancements and bug fixes	Prototypers create experiments. We: ● specify what the prototype will accomplish ● design the prototype ● implement the prototype ● fix and enhance the prototype

Everyone Needs All Skills

Developers need experience in specification, design, implementation, maintenance, and prototyping throughout every software development project. Because of the fractal similarity between the phases, proficiency in any one phase requires proficiency in all phases. Developers bring specific skills and points of view to their work. They often view the entire life cycle in terms of their specialty. Strong developers understand a variety of perspectives, so they can approach their current situation in any terms they choose. Developers make better decisions when their perspective matches the actual circumstances of the project, rather than the preconceived

notions of progress defined by life cycles.

Problems encountered in one phase of development might best be understood in terms of another phase. For example, is a discrepancy between a specification and an implementation, a problem in the specification or a problem in the implementation? Small bugs can be symptoms of specification errors and omissions. Improving the simplicity and generality of a bug fix can force a specification change. Developers must often choose at what level to resolve a discrepancy. Many minor user concerns are often left out of a specification and must be filled in during implementation. Mid-level discrepancies may force developers to maintain a specification, to design a mid-level component, and to implement a few lines of code, all at the same time.

By understanding the flow of change within a project, developers can better cooperate with other developers working on other steps. Those who write specifications should understand the abilities and limitations of those who will implement and maintain the results. Those who implement code should cooperate with those who specified and will maintain the program. Those who maintain code should fix and improve what has been specified and implemented, rather than just replacing known bugs with different bugs. This is true, even if the whole project will be completed by the same developer.

CONCLUSION

In this paper, I have shown how the principles of chaos can lead us to a better understanding of software development. I use chaos as a metaphor to define both the Chaos model and the Chaos life cycle, emphasizing the developer's point of view. The Chaos model and Chaos life cycle define a concise framework for exploring, interpreting, and assessing software development.

The Chaos model combines a linear problem-solving loop with fractals to suggest that a project consists of many interrelated levels of problem solving. The top levels of a project consist of resolving a few large problems and the bottom levels of a project consist of resolving many small problems. Developers solve problems at all levels between the "whole project" level and "one line of code" level. The Chaos model expresses the humanness and the uniform complexity of software development.

Interpreting the Chaos model shows how users, developers, and technologies interact during a project. The users' needs define the goals of the project at the upper levels, and must trickle down to the bottom levels. The technical resources define the solutions at the bottom levels, and must trickle up to the top levels. Developers match the users' needs with the technical solutions in the middle levels of a project.

The Chaos life cycle reveals complexity in the evolution of a project. Defining the phases of the life cycle in terms of fractals shows that all phases of the life cycle occur within all other phases, throughout the life cycle. Thus, how we interpret the phases of the life cycle shows our perspective on the state of a project, rather than any essential truth about the state of the project.

Many authors have made similar points and created models that are both iterative and flexible. But, the perspective of the Chaos model allows us to combine many of these issues and arguments into one framework. The power of the Chaos model is that it concisely unifies so many facets of software development.

ACKNOWLEDGMENTS

Over the last six years, many people helped me to develop, critique, and expand on the ideas expressed in this paper. I would like to thank Bear, Bunnyrabbit, Anne Cable, K. C. Cress, Anthony Giancola, Joe Hill, Janice Kim, Arthur B. Maccabe, Puppydog, Gideon Shaanan, Charles Troup, Edwin E. Wing, Laura Yedwab, and colleagues at Electronic Data Systems Research, Vision Harvest, and WordPerfect. I would especially like to thank Mina Yamashita.

BIBLIOGRAPHY

Boehm, Barry W. (1986) A Spiral Model of Software Development and Enhancement, ACM Software Engineering Notes, August 1986, pages 14-24.

Booch, Grady (1991) *Object-Oriented Design*, Benjamin Cummings.

Borenstein, Nathaniel S. (1991) *Programming as if People Mattered*, Princeton University Press.

Brooks, Frederick P. Jr. (1975) *The Mythical Man Month*, Addison Wesley.

Brooks, Frederick P. Jr. (1987) No Silver Bullet, in *Computer*, April 1987, IEEE Computer Society.
Charette, Robert N. (1986) *Software Engineering Environments*, McGraw-Hill.
DeGrace, Peter and Leslie Hulet Stahl (1990) *Wicked Problems, Righteous Solutions*, Yourdon Press.
Gries, David (1981) *The Science of Programming*, Springer Verlag.
Mandelbrot, Benoit B. (1983) *The Fractal Geometry of Nature*, Freeman.
Marcus, Clare Cooper and Wendy Sarkissian. (1986) *Housing As If People Mattered*, University of California Press.
Royce, W. (1970) Managing the Development of Large Software Systems: Concepts, WESCON Proceedings (Aug.)

Paper

- The Chaos model is a developer-oriented model that relates the whole project to lines of code.
- Express the complexity and humanness of software development.
- The Chaos model combines the linear problem-solving loop with fractals.
- Understand and improve the role of the developer in software development.
- The Chaos model is about levels. The Chaos life cycle is about scaling

The Chaos Model

- Define the Chaos model.
- Software development requires both human and technical skills.
- States of a project are a subset of the status quo.

Interpreting the Chaos Model

- The Chaos model expresses continuity between users, developers, and technologies.
- Users' needs define the goals at the top level. The goals trickle down.
- Technology defines the solutions at the bottom level. The solutions trickle up.
- Developers match the users's needs with technical resources in the middle levels.

The Chaos Life Cycle

- Define the Chaos life cycle.
- Fractals define the life cycle phases in a simple, yet general, way.
- Phase indicates our perspective rather than project's status.
- Everybody needs all skills throughout the life cycle.