Introduction

Software Development as a Game
The Software Development Game
The Importance of Strategy
Modeling Uncertainty in Software Development
A Note About Chess and Go

The Chaos Strategy: Resolve the Most Important Issue First
Problems, Moves, and Issues
Issues Are Points of Resolution
Important Means Big, Urgent, and Robust
Phases of the Chaos Strategy

Devising a Specific Strategy
Trading Off Competing Goals
Thematic Problems and Issues
Addressing Hard Problems
Coping With a Crisis

Conclusion

Acknowledgments

Bibliography

INTRODUCTION

“What line of code should I write next?” This paper resulted from my search for an answer that reflects my experiences in writing real programs, and reflects both the Chaos model and life cycle. After trying to infer an answer directly from the Chaos life cycle, in fact from any life cycle, I ultimately gave up. I concluded that while life cycles describe what I can do in a project, life cycles do not describe what I should do in a project. For that I need another concept, such as strategy.

Strategies are idealized plans of how software development should work. Strategies help us to prioritize the things we do. A general strategy defines an overall approach for solving a problem that must be adapted to meet local circumstances. A specific strategy defines immediate goals. A good specific strategy balances short-term progress with long-term goals.

Applying the concept of strategy to software development may seem inappropriate, or even absurd, to software developers who normally worry about concepts like correctness, specification, and quality. Developers have been taught to think of software development as totally predictable, with each step a perfectly understood element. Strategy implies a contest which admits that a developer could lose. On the other hand, developers may be more familiar applying strategy to investments, marketing, elections, and war. People do meaningfully use strategy for complex military and business problems.

In The Chaos Model and the Chaos Life Cycle, I argued that each software development project is a chaotic, multi-level sequence of issues that arise and get resolved. Developers repeatedly select an issue to resolve, devise an approach, implement a solution, maintain the resulting program, and so forth. Software development is a project-to-project, situation-to-situation, minute-to-minute process, with many levels. Developers must deal with all levels from the “whole project” level down to the “one line of code” level. In The Complexity Gap, I argued that the middle levels of a project are not addressed by the traditional Stepwise Refinement and Object-Oriented Design strategies. The macro-process and top-down strategies address the top levels of a project. The micro-process and bottom-up strategies address the bottom levels of a project.

In this paper, I propose the Chaos strategy to guide development on the levels of a project within the Complexity Gap. I begin by describing parallels between software development and games of strategy, and by defining the Software Development Game. I then define elements of the Chaos strategy, a middle-out, technology-independent strategy. Finally, I show how to adapt the Chaos strategy to fit the various circumstances that arise in complex software development projects.

SOFTWARE DEVELOPMENT AS A GAME

In this section, I explore some of the ways that software development resembles a two-person game of strategy, such as Chess or Go, and some of the ways that writing one line of code in a software development project resembles playing one move in a game of Chess or Go. I model software development as a game to emphasize that strategy is essential to writing software productively. The Software Development Game model also reveals uncertainty within the software development process.
The Importance of Strategy

During each turn of the Software Development Game, developers want to select and make good moves. Moves should bring the project toward completion, though, the next line of code that a developer writes will probably not finish the program or even reach the next milestone. Because developers need to make good moves in the middle of complex projects, they need strategies to guide their choices.

Given the flexibility of the Chaos life cycle, it seems natural to ask, “what should a developer do first and what should he or she do next?” The Chaos life cycle allows me to ask the following questions about the design phase of a project, as well as the corresponding questions about the requirement analysis, implementation, and maintenance phases. “Which design issues must be resolved first and which can be left until later in the project?” “Which implementation, testing, and maintenance issues should be addressed as part of the design phase?” “Which implementation, testing, and maintenance issues are independent of the design phase, and should proceed in parallel with the design phase?”

In some projects, developers may decide to design everything before implementing anything. In other projects, developers may design and implement certain key components before designing and implementing any others. When making a medium-sized change to a program, developers could bury all of the changes in one module of the code, or they could allow the changes to affect many modules, or they could even make global changes. Whatever their options, developers make their choices explicit through their actions.

Different sequences of moves produce different results. Developers may work differently as deadlines approach and pressure to finish increases. Developers may lack the time to find the most elegant solutions or to implement the desired level of quality. Software has a lot of momentum, so it is important to build from the current state of the code. Developers cannot rewrite major sections of code without investing a lot of effort. The code written first often has the highest quality because developers have more opportunities to refine the code later. Therefore, developers should finish the most important parts of the project first.

Inconsistencies and inefficiencies can cause projects to be late or incomplete. In the middle of complex software development projects, developers have many opportunities to inadvertently make inefficient moves and delay the project. Poor developers, and even the worst developers, can sometimes write good solid code. And even the strongest developers can make mistakes. But poor developers must rewrite their code more often as circumstances change, because they are less able to build on their past efforts. Their code is not necessarily bad code. It just doesn’t fit the circumstances and mistakes cause them to consistently fall behind. Strong developers work efficiently and productively throughout the project and they consistently make good moves in complex positions without losing track of the final goals.

Modeling Uncertainty in Software Development
Many developers that I have met seem to think that they have total control over their choices and that they can easily write the proper code for any circumstance. However, I believe that writing the appropriate code is very difficult and that developers only have some control over their choices. I believe that during a project, developers must both plan and react. If we plan when we should be reacting, our actions will be irrelevant. If we react when we should be planning, our actions will be inefficient. The Software Development Game conveys this uncertainty. The Software Development Game explains the unknowns and strategy enables us to cope with them. In part, I want to emphasize uncertainty within the software development process, rather than the emphasis on certainty sponsored by the Structured Programming and Correctness movements for nearly three decades.

Smithson argues that wherever there are people, there are both uncertainty and ignorance. People can only handle a small amount of complexity at a time and easily forget details. People with incomplete knowledge can easily make mistakes. People with erroneous knowledge will probably make mistakes. Developers believe many things about their projects, beliefs which may be true, false, or even contradictory. Some of their beliefs are vague, fuzzy, and ignorant notions. Developers also have beliefs about what they believe that are subject to the same uncertainty and ignorance. It is easy to make good decisions for simple problems, but much harder for large or complex problems. Whether developers know their limitations or not, they make mistakes.

Uncertainty crops up in many ways. The definition of the project can be uncertain. Developers cannot really know ahead of time what will please the users. Users can change their minds and their expectations. The project can evolve unpredictably. In addition to changing design parameters, many things such as politics, time, and resource pressures can alter the evolution of the project. Changes in the underlying technology and competing products can affect how the project evolves. Even the current state of the project is uncertain. Most developers understand their own code pretty well, but have only a fuzzy notion about how other code in the project works. The true state of the code could be very different from what they expect.

All of the information about a project is subject to error. Developers know many things about the problems at hand and the current state of the solution as well as the traditions and values of their co-workers. To correct their mistakes and confusions, developers can talk to other developers on the project and read code and documentation. But since no developer knows where his or her own misconceptions lurk, errors are often exposed during a crisis. This explains why developers take so long to come up to speed on a project, and why they continue to make mistakes.

A Note About Chess and Go

I believe that producing software is much more intricate and difficult than playing Chess or Go, which we know is very hard to do well. Developers need strategy to write software for the same reason that Bobby Fisher needs strategy to play Chess and Kobayashi Koichi needs strategy to play Go. A small program consists of thousands of different lines of code as opposed to a few hundred moves or less in a game of Chess or Go. Developers may consider thousands of possible changes to a program at once, as opposed to a few dozen legal moves in a game of Chess or a few hundred legal moves in a game of Go. I believe that making good moves matters at least as much in programming as in Chess or Go.

I prefer to compare software development to Go because many relationships between software development and Go seem more obvious than in Chess. Go is a game of small differences, that get smaller and smaller toward the end of the game. These diminishing returns resemble the debugging and maintenance that occur at the end of real projects more accurately than the abrupt checkmate of the king in Chess. In a game of Go, players add stones to the board, one at a time, until they reach a final state. Players never move stones, though sometimes players remove captured stones from the board. Developers write code one line at a time until the program is finished. Sometimes developers change or delete sections of code, but mostly they add code. Though similar comparisons and interpretations can be found in Chess, the analogies seem less obvious to me.

In Go, players strive to surround as many points of territory as possible. The winner surrounds the most points of territory. Although many different moves can be played at any time, the sequence of moves played is very important. In a software development project, developers strive to complete the program as soon as possible with as high a level of quality as possible. Many different lines of code could be written first, but the sequence in which the code is written is very important. Go strategy is about consistent, efficient play. Inconsistent play, or even a single slack move, can cause players to fall behind. Once a player falls behind, he or she may never catch up.

Go players encode strategy as a set of actions plus advice on when to use each one. Much of Go strategy consists of proverbs and rules of thumb that enumerate “in this case do this, in that case do that.” Go proverbs include “Side attachment? Extend!” “Avoid empty triangles!” and “Hane at the head of two stones!” Proverbs may even contradict each other, such as: “Resign if you lose all four corners!” and “Resign if you win all four corners!” And, many principles of Go strategy are not explicitly stated. Go strategy has well-developed structures, rooted in time, which require study to discover and appreciate. There are many pieces of strategic advice to choose from. It takes years of hard knocks to learn when to heed, when to ignore, and how to apply each of these bits of wisdom.

Of course, the analogy between software development and the games of Chess and Go is not exact. Developers and Nature do not play symmetrical moves. Software development projects are clearly not zero-sum games. Complex projects may involve many independent players. And the uncertainty within software development projects implies that imperfect information games, such as Backgammon, may model software development more accurately than Chess or Go. I define the Chaos strategy largely in terms of Go, because Go is complex enough to carry an interesting comparison to software development and I understand it well enough to provide a strong foundation for this paper.

THE CHAOS STRATEGY: RESOLVE THE MOST IMPORTANT ISSUE FIRST
The Chaos strategy has one rule, “Resolve the most important issue first.” Though the meaning of first is clear, the meanings of resolve, important, and issue are more subtle. The Chaos strategy assumes that we begin with a list of issues to resolve. The most important issue on the list should be resolved first. After each step, developers reevaluate the list in light of the new state of the project and select the next issue. This process repeats until all important issues are resolved and the project ends.

Chaos is an appropriate name for this strategy because it is based on the Chaos model and the Chaos life cycle. Without concepts of software development as complex as the Chaos model and the Chaos life cycle, the sophistication of the Chaos strategy would be unnecessary.

In this section, I expand on the meanings of issue, resolution, and important. I also compare the phases of a game of strategy with the phases of the Chaos strategy to describe the evolution of a project. The concepts of opening, middle game, and endgame help explain how a project evolves in a different way than the phases of the life cycle do.

Problems, Moves, and Issues

I define the terms problem, move, and issue to distinguish between the elements that developers can control and those that developers cannot control. Problems are the tasks we want to accomplish that lie above the Complexity Gap. Moves are the pieces of a solution that lie below the Complexity Gap. Issues are the manageable units of work that lie within the Complexity Gap.

Problems are the tasks developers want to accomplish that are defined by the application. Problems correspond to activities within Booch’s macro-process and typically include: implementing features, fixing bugs, and revising specifications. Problems are the largest elements that developers address. Developers have little control over what the problems are, only over how they will solve the problems. Complex problems must often be broken into manageable subproblems to be resolved. These subproblems are the keys to completing complex applications. When problems are broken out, developers need to manage progress on each subproblem.

Moves are the atomic components of a solution that are defined by our technology. Moves correspond to activities within Booch’s micro-process and typically include: writing one line of code, changing one line of code, or restructuring one chunk of code in a language such as Fortran, Smalltalk, or Assembler. Moves also include using interactive tools, such as using a dialog editor to place one button. Developers have little control over what the moves are, only over how they will combine moves to build solutions. Since developers make moves one line of code at a time, developers must combine simple moves into manageable solutions.

Issues are the manageable units of work that lie on all levels between problems and moves. Issues correspond to activities within the Complexity Gap. Developers seldom resolve large problems directly and they seldom resolve anything with just one move. Typically, an issue consists of pieces of several different problems and consists of several different moves. Issues include writing functions, objects, and modules, creating scaffolding, and revising large blocks of code. For example, a function may be called from two different places to solve a part of two different problems. And a function combines many lines of code or, in other words, pieces of smaller solutions. Though each issue fits between a particular application and a particular technology, the concept of issue is independent of applications and technologies.

Before choosing the issue to resolve next, developers should enumerate each problem, move, and issue on a list, at least conceptually. Developers need not actually write down these lists. The To-Do list enumerates the problems of the project left to complete. The To-Do list should be comprehensive and identify the heart of each problem, because problems that are misidentified or left off the To-Do list will not be addressed. Note that during the maintenance phase of a project, a bug database may explicitly enumerate the To-Do list. The Can-Do list enumerates the valid technical solutions or moves available to us. The Can-Do list describes our immediate options. We should not underestimate the value of understanding our choices, because very often, poor developers simply fail to consider the best moves. The Issue list enumerates the middle-level options that developers use to construct pieces of problems using groups of moves. Given a more thorough description of the project, the developer can more effectively evaluate each issue.

Issues Are Points of Resolution

Issues are points of resolution. Resolving an issue does not mean completely finishing an entire problem. But, at the point of resolution, the unfinished parts of the problem should be relatively unimportant to the whole project and the finished parts of the problem should contribute to the rest of the project. When choosing an issue to resolve, developers should consider the level, sequence, and scope of each issue.

Level matters because each issue represents a different tradeoff of application concerns and technical solutions. The chosen issue could emphasize the upper levels, which gives a more top-down, problem-oriented flavor, or emphasize the lower levels, which gives a more bottom-up, solution-oriented flavor. Developers should carefully consider the relationships between the different levels. Issues that appear on one level may be best resolved on another level. I have often seen developers try to resolve an issue on one level when a better solution lies on another level, such as fixing a bug in an implementation by redefining the interface or vice versa.

Sequence matters because each issue represents a different tradeoff of short-term progress and long-term objectives. Each issue relates to both the issues already resolved and the issues that will follow. Often, work on a particular issue could stop at many different points. Work may proceed to the quickest point of resolution or to the most complete point of resolution. Developers should carefully consider the evolution of the whole project. Developers should plan where they want the project to be in one week, one month, etc., and they should keep possible changes in mind.

Scope matters because each issue encompasses a different amount of the uncompleted work. Developers should consider a scope that is both large enough to make real progress and small enough to be completed in the current circumstance. If developers choose too narrow a scope, their efforts may be short-sighted. If developers choose too broad a scope, their efforts will lack force.
After choosing an issue, the developer should apply reasonable effort, working to resolve that one issue before turning to the next. After resolving that issue, the developer deletes all resolved issues from the Issue list, adds any newly discovered issues to the Issue list, and likewise updates the To-Do and Can-Do lists. The developer then proceeds to resolve the most important issue that remains.

**Important Means Big, Urgent, and Robust**

All issues that a developer resolves should be *important*. The word *important* seems an intuitive, if imprecise, concept. In my experience, *important* is a very subtle concept that changes according to context. So to be slightly more rigorous, I define *important* as a combination of *big*, *urgent*, and *robust*. Note that we often use the words *efficient* or *productive* when we mean *important*.

**Big** issues make major progress. It is bigger to implement core features than optional features. It is bigger to implement new components than to finesse working code. Developers should resolve the issue that makes the most progress toward completion of the project. As the project evolves, smaller issues become the biggest ones left.

**Urgent** issues emphasize timeliness and respond to critical concerns and threats to hold up the project. Building a framework so that components can be tested and integrated is an urgent move. Input and output routines must often be finished before anyone else can test their code. Interfaces between components are usually urgent. Ideally, each piece of the project should be finished just before anyone needs it. Otherwise, a different issue is probably more urgent. As the project evolves, the remaining issues become more independent and less urgent.

**Robust** issues increase the simplicity and flexibility of the emerging system. Generalized and robust components cope with design changes and integration problems better. Building debugging aids usually slows development when viewed locally. However, robustness makes the project less susceptible to hidden bugs and integration problems later. Generalizing components is another robust move. But, developers should be realistic about robustness. If specific robustness is never used, the effort used to build the robustness is wasted. And developers should avoid complicating the project. As the project evolves, developers will create robustness less often and use the robustness that already exists more often.

In practice, each issue that we resolve should be big, urgent, and robust at the same time. If a particular issue lacks one or more of these properties, then a more important issue probably waits elsewhere. At the point of resolution, important aspects of a problem will be finished and any code should be reasonably robust. The final work on a particular problem often adds robustness so that the remaining work can be postponed. Work often stops when solving the rest of a problem is no longer urgent. As the project evolves, the issues become smaller, less urgent, and less likely to create robustness. The final issues addressed in the project should tie up minor loose ends. Each issue should be addressed in its turn. This is the natural evolution of a project.

**Phases of the Chaos Strategy**

The phases of strategy games shed a different light on the evolution of a project than the phases of life cycles do. Many games of strategy have three distinct phases: *opening*, *middle game*, and *endgame*. In Go, these phases reflect the strategic possibilities on the board as the game evolves. The Chaos strategy also has three distinct phases.

During the **opening** of a game of Go, players establish a framework from which to play the rest of the game. Moves must be very big and very flexible because they impact the entire board. During the opening of a software development project, developers define a framework or architecture into which the rest of the code will be placed. With a strong global framework, developers will have a stronger base from which to build, and the project will make major progress sooner. The framework may need to be changed, so it should be flexible. Developers will then share more code and throw away less scaffolding. Inexperienced developers often feel lost in the opening because everything is so fluid. They often attack any well-defined problem, as if the middle game had already started, because the goals of the middle game are more tangible. The many efforts to understand the role of architecture in a project suggest that researchers are trying to understand the possible openings of a project.

During the **middle game** of a game of Go, players resolve interconnected goals. Players may think in terms of setting and achieving specific goals, such as cutting off an opponent’s group or invading the opponent’s potential territory. Fighting leads to intricate positions. All of the goals in this phase are still interdependent, and must relate to each other and the board as a whole. During the middle game of a software development project, normal programming objectives are set and achieved. The middle game of a software development project corresponds to what most developers call real programming. Any code that deals with interfaces or local resources must be completed. All major components are written and tested. Major flaws, which are uncovered, must be corrected. The middle game concludes when all remaining work can be addressed locally.

During the **endgame** of a game of Go, players resolve all local positions. In principle, players know the result of a game when the endgame begins. In practice, players work very hard to win the last few points out of each local position. During a software development project, developers may think they are done when the program is code-complete; however they still must finish the last components, tweak the performance, finish the documentation, and fix minor bugs. Most of these tasks have little to do with bugs or correctness. These tasks concern the completion of local details. In complex projects there are many changes to make, so developers should make changes in order, the most important ones first. Work stops when no further changes are worth making. I prefer to call this final phase the *endprogram* phase rather than the *debugging* or *maintenance* phase because developers finish the program during the final phase. According to the Chaos life cycle, developers debug and maintain code throughout the entire project.

During all phases of a software development project, developers do the same thing, they resolve one issue after the next. Each resolution of an issue increases the constraints on a project and reduces the developer’s options. Therefore, each issue should be delayed until it is the most important issue remaining. Opening moves precede middle game moves and middle game moves precede
DEVISING A SPECIFIC STRATEGY

In this section I distinguish between general and specific strategies. General strategies define general principles, rather than specific details. The Chaos strategy is a general strategy because it describes the general principles rather than the specific actions to apply to a given move.

To actually use the Chaos strategy, developers must adapt it to their project. Developers must devise a specific strategy, a sequence of moves planned in advance, to fit their circumstance. The specific strategy defines the sequence of moves that he or she will play, such as, “implement $w$, test $x$, implement $y$, revise $z$, . . . .” It means, “do this now.” The specific strategy helps developers to order tasks and to put objectives in place. It keeps them from getting lost in the details of small problems or bouncing around inefficiently between large problems.

In this section, I discuss the devising of a specific strategy. I first discuss making tradeoffs between issues. I show that the Chaos strategy is consistent with thematic concerns that are not described by code structures, I discuss two local strategies for dealing with hard problems, and I show that the Chaos strategy works even during a crisis.

Trading Off Competing Goals

In Go, players often want to accomplish several different goals at the same time. For example, a player may strive to win more of the opportunities than his or her opponent, or a player may try to attack enemy stones while expanding his or her own territory. Players often trade off competing objectives. Software developers also trade off competing objectives. Developers usually want to accomplish several important goals, but they cannot do everything at once. When developers must accomplish several independent goals at the same time, then they must devise a common perspective or thread to unite the seemingly independent goals into one goal.

Some threads are based on program structures. Stepwise refinement suggests that we should write the high-level routines first or define the object interfaces first. This advice may be useful in many circumstances, yet may be misleading in others. For data-intensive applications, such as a database management system, designing the data structures first may be the best advice. For user-intensive applications, such as data entry systems, designing the user interface first may be the best advice.

Some threads are based on phases of the life cycle. The Waterfall life cycle suggests doing all specification first, then all design, and so on. The Chaos life cycle suggests that phases may overlap. If a specification is important, then specify until it is no longer important. A useful specification need not specify the entire program. Similarly, developers need not design, implement, or test anything completely, before moving on to the next issue. When a system easily decomposes into separate pieces, the implementation and testing of some pieces can often proceed in parallel with the specification and design of other pieces. Developers can decide when to proceed to the next phase for each piece of the project.

The specific strategy that a developer chooses should embody an appropriate thread. Perhaps certain problems should be resolved first. Perhaps certain pieces of certain problems should be resolved first. Usually, resolving some combination of pieces from different problems is best. Developers may need to work very hard to find a good thread. If a developer cannot find a common thread, then he or she must choose one goal or the other.

Developers should plan the sequence of moves most appropriate to their circumstances. To the extent that developers devise an appropriate strategy, they choose their results. A good specific strategy helps to avoid thrashing or switching back and forth inefficiently between two or more different goals. The best strategy for a given situation depends on the state of the project, the nature of the problem, and the skills and preferences of the developers, i.e. where we are today and where we want to go. Devising the best strategy for the current situation requires hard work and insight. The following table discusses some of the properties of good specific strategies.

<table>
<thead>
<tr>
<th>A Specific Strategy Should Have Some or All of These Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vitality:</strong> A good strategy identifies the heart of the current situation.</td>
</tr>
<tr>
<td><strong>Big Picture:</strong> Developers must look to the big picture and move the whole project toward completion. It doesn’t let developers get distracted by details.</td>
</tr>
<tr>
<td><strong>Relevance:</strong> A good strategy relates to the particular problem and the skills and preferences of the developers.</td>
</tr>
<tr>
<td><strong>Consistency:</strong> A good strategy pursues each issue to resolution, before turning to the next. Bouncing from one issue to another or changing the strategy invariably leads to inefficiency. The sequence of issues that gets resolved must flow.</td>
</tr>
<tr>
<td><strong>Flexibility:</strong> Flexibility enables programmers to cope with design changes and to recover from design and coding flaws.</td>
</tr>
<tr>
<td><strong>Patience:</strong> It is not possible to get ahead very fast. The difference between efficient moves and inefficient moves is often very subtle and will not show up for a long time.</td>
</tr>
<tr>
<td><strong>Pragmatism:</strong> A specific strategy should build effectively on previous work. It fits the occasion and predicts what really will happen rather than what we wish would happen.</td>
</tr>
<tr>
<td><strong>Opportuneness:</strong> When developers find an opportunity that fits with the big picture, they take it. If they wait, the opportunity may vanish. Opportunities which do not share the same big picture must be ignored. Good strategies create and exploit opportunities.</td>
</tr>
<tr>
<td><strong>Risk:</strong> Strong developers are aggressive and take risks. Developers cannot do their best work unless they risk failure once in a</td>
</tr>
</tbody>
</table>
**Thematic Problems and Issues**

Thematic problems and issues are the nonfunctional requirements and goals that do not appear explicitly in the program text. Problem-oriented themes include the high-level goals of usability, performance, reliability, correctness, and portability. Move-oriented themes include the low-level goals of locking policy, logging policy, constraint checking, exception handling, error recovery, coding styles, memory management, and documentation. Themes include everything not supported directly by code structures.

Themes are hard to implement because many instances occur scattered throughout the code and each instance may differ slightly from all other instances. Because thematic problems cannot be isolated within a single function or a single module, getting them right can be very difficult in a complex project. Consider the locking policy. To maximize efficiency and minimize deadlocks in a sophisticated real-time control system or database, each function may need to acquire and release a different set of resources in a very specific manner. Hundreds of functions may need to cooperate with each other according to very specific patterns. Consider constraint checking. Efficiency may dictate that different subsets of constraints be checked and propagated in many different places. And combinations of themes can increase the difficulties. Combining locking with exception handling and error recovery, especially in real-time and database management systems, exaggerates the difficulties.

Developers resolve themes using a variety of techniques, but most of these techniques boil down to establishing and enforcing policies. This means that developers identify and document the proper solutions, inspect the code for conformance, and correct any errors. Themes require finesse and consistency more than additional lines of code. Thus, themes often cannot be measured in terms of lines of code and therefore they often appear as unproductive work.

Unfortunately, nobody can expect high quality code, good documentation, or consistent error recovery without investing the appropriate time and resources. Without a push, I sometimes avoid resolving hard or unpleasant themes, such as quality control or documentation, until it is too late to do a proper job. So, developers need to account for the resources and the responsibility to resolve thematic problems, just as they do for functional problems.

If we define productivity in terms of issue resolution, rather than in lines of code, we can measure and account for nonfunctional requirements in exactly the same way as functional requirements. According to the Chaos strategy, themes are just another part of the standard software development process. In this sense, developers can (and should) treat themes just like any other functional requirement. Code reviews, test suites, and coding standards should be treated as part of normal software development, rather than as part of some special quality control process or review process. The Chaos strategy requires only that we identify the important themes as explicit problems and issues. Thematic problems may be broken into smaller thematic issues to get resolved. For example, to increase reliability we can break the reliability problem into a set of test suite and structured walk-through issues. The Chaos strategy handles themes because it is not defined with respect to any specific technology.

**Addressing Hard Problems**

There is a time and a place for solving each problem; this is especially true of hard problems. Few strategies distinguish between easy and hard problems or address the special considerations of hard problems. Hard problems can be inherently difficult, such as solving combinatorial problems fast, or difficult because a developer lacks specific skills, such as experience writing device drivers. Both types of difficulties put a project at risk. Hard problems should be addressed to minimize their adverse affects on the rest of a project, and should be treated with care.

The way that developers treat hard software development problems can be compared to the way that Go players treat moyos. Moyos are territorial frameworks or groups of stones that threaten to claim a large territory. A large moyo often influences much of a game, although dealing with it may account for only a few moves. Dealing with a moyo either first or last is often bad. But when the time comes, players consider two main approaches: reducing and invading. Ishida and Davies, Fujisawa, and Takemiya explore these two approaches in greater depth.

The first approach reduces the problem gently from the outside. By gently encroaching on the problem, sometimes the problem dwindles until it becomes manageable. The advantage is that a reduction strategy minimizes the risk to scheduling for the rest of the project. This approach maintains balance with global objectives. The disadvantage is that the problem may not be reduced enough. Reduction strategies seem appropriate when a partial solution may suffice.

The second approach invades the heart of the problem. Invasions fail in many ways so they must be carefully executed. To begin with, invasions require outside support. External approach moves give the attack something to work with on the outside as well as giving the developer a chance to probe the problem gently. Building scaffolding and other support code gives the developer support tools, as well as a chance to study the problem more closely. Then when the developer invades, he or she can devote his or her full energies to solving the problem. The disadvantage of invasion strategies is that a developer can get bogged down in a hopeless task. If the developer must also finish other work, then delays can adversely affect the rest of the project. Invasions are inherently all or nothing gambles that seem appropriate when partial solutions will not suffice.

When addressing a hard problem using either approach, developers should play lightly. Code may not work out, so the developer must be able to sacrifice it without qualms. Developers should create very light prototypes and should stabilize their code as quickly and as locally as possible. This keeps them from wasting time on attempts that do not work and keeps them from adversely affecting the rest of the project. The code that they produce will probably need additional work. If developers could have solved the problem correctly the first time, it wouldn’t have been a hard problem. Once the code becomes stable, it can be revised and
incorporated into the project.

**Coping With a Crisis**

Good strategies can lose. Sometimes developers uncover serious problems and then realize that they must make major revisions. When a crisis happens, we say that the specific strategy has failed. Crises can occur because developers make mistakes or because users change their minds. Developers may or may not have control over the circumstances. In either case, the only thing to do is to reevaluate the current state of the project, choose a new strategy, and carry on.

When there is not enough time to complete a project, something must change. If more time cannot be made available, then the project must change. When the developers must accomplish more with less time, they invariably implement only the most important goals and abandon everything else. By definition this is the Chaos strategy. No other solution makes any sense. Big, urgent, and robust remain the same concepts, though they may be applied in a different way. Thus, the principles of the Chaos strategy work whether there is a crisis or not.

When a project cannot be finished, the Chaos strategy insures that the most important aspects of the project will be complete. One useful result of following the Chaos strategy is that if development stops at any point, the project will be as complete as possible and the value of the program will be as great as possible. Unfortunately, this doesn’t tell you where you will be. The Chaos strategy doesn’t force projects to a point of completion. Nothing can force a project to completion. The Chaos strategy does not solve a crisis, but it can help you muddle through one.

**CONCLUSION**

Developers write programs, one line of code at a time. So to write programs productively, developers need guidance, one line of code at a time, throughout the software development process. Strategies help to balance short-term progress with long-term goals.

To motivate the need for strategy, I draw an analogy between software development and games of strategy. Life cycles resemble the rules of a game and describe what we can do during a project. Strategies describe what we should do during a project. The Software Development Game supports the ideas that the sequence of moves matters in software development and that developers only have some control over the software development process. Developers need a sophisticated concept of strategy to match the complexity of real programs and to guide their efforts throughout a project. The Software Development Game provides a framework for applying strategic advice from business, military, financial, and other fields.

I define the Chaos strategy as, “resolve the most important issue first.” Issues are points of resolution that are smaller than a problem and larger than a move and are independent of any particular application or technology. Important means a combination of big, urgent, and robust. To maximize forward progress and minimize rework, there is a best time to resolve each issue. I also show that the phases of the Chaos strategy resemble the opening, middle game, and endgame of games of strategy and that these phases describe the evolution of a project. The Chaos strategy avoids imposing any particular sequence of choice. The Chaos strategy concerns getting the work done, not how to do the work.

To use the Chaos strategy, developers must devise a specific strategy to meet the needs of their current circumstance. The specific strategy should reflect the applications, technologies, development environments, and experiences of the developers, as well as the various crises that may arise. Developers often struggle to find a thread to unite their varied goals.

I believe that developers already use many aspects of the Chaos strategy throughout the software development process. Developers use the “most important first” principle in software maintenance, when they explicitly prioritize their tasks. And developers who argue that they only use orthodox top-down strategies will often admit to considering middle-out issues when they design functions and modules. The Chaos strategy brings many common-sense ideas into one cohesive structure and legitimizes these middle-out strategic activities. The Chaos strategy advocates a very specific focus on line-by-line progress, the only thing that leads to success.

**ACKNOWLEDGEMENTS**

Over the last seven 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 J. Giancola, Joe Hill, Janice Kim, Arthur B. Maccabe, Puppydog, Gideon Shanaan, Michael Thompson, Charles Troup, Edwin E. Wing, Stephen P. Wing, Laura Yedwab, and colleagues at Electronic Data Systems Research, Vision Harvest, and WordPerfect, and Go enthusiasts in Albuquerque and Santa Fe. I would especially like to thank Mina Yamashita.

**BIBLIOGRAPHY**


Peter Coad and Edward Yourdon (1991) *Object-Oriented Design*, Yourdon Press.
Paper

- Describe and deal with uncertainty and choice throughout the software development process.
- Strategy is an important metaphor for consistently doing a good day-to-day job.
- Strategy allows me to pull in advice from other fields, specifically Chess and Go.
- Distinguish between *should* and *can*.

The Software Development Game

- We use strategies anyway.
- Two person games of strategy model uncertainty and choice.
- Software development is like a game of strategy.
- To develop software efficiently, we must use strategies.
- Strategies help us balance short-term progress and long-term goals.

The Chaos Strategy

- The Chaos strategy means, “resolve the most important issue first.”
- An issue is a mid-level concept, between a problem and a move.
- Issues are points of resolution. Sequences of moves must lead to resolution.
- Important means: big, urgent, and robust.
- The phases of software development when using the Chaos strategy resemble the phases of strategy games.
- The Chaos strategy is independent of application and technology.

Devising a Specific Strategy

- When many objectives are competing for resolution, we must devise a common thread.
- The Chaos strategy can handle thematic problems and solutions.
- The Chaos strategy does not change, no matter what our goals.
- We need special approaches to handle hard problems.
- The Chaos strategy does not change during a crisis.