JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 5, No. 1, January-February 2006

Complexity, its in the mind of the beholder

John D. McGregor, Clemson University and Luminary Software LLC, U.S.A.

Abstract

Complexity is a much analyzed, much debated, much measured property of software-intensive products. From a strategic point of view, complexity has implications for the development and evolution of software-intensive products. In this issue of Strategic Software Engineering I will consider multiple views of complexity, sources of complexity and actions that manage complexity.

1 INTRODUCTION

When my wife looks into my home office she sees a complex mass of interlocking towers of paper. When I look at it, I see the pile related to a consulting project and the pile related to a course I am teaching, and sometimes papers that relate to both. What is complex to her is not complex to me because I understand the structure. This is similar to the amateur chess player and the grand master. The grand master perceives large chunks of the board than the amateur who only sees a mass of individual pieces. The game is the same but it is perceived as more complex by the anateur.

The impetus for this column came from a discussion with a colleague. He believes that software engineering techniques are only needed for "large" projects. I believe that it depends upon the context in which the project is sited: how many people, how much time, and how much code to be produced.

My hypothesis is that "large" is relative just as is complexity. If I have to write a program that is estimated at 5000 LOC and I have a year to write it, I will perceive it to be much less complex than if I only have three days to complete the task. Similarly, if I must write the program by myself I will perceive it to be more complex than if I have two collaborators. There is less opportunity to analyze and understand the fundamental structures.

With time I can mitigate the effect of complexity by exploring the structure of the problem or solution. How much time is required to do this seems to be related to the size of the problem. Does this mean that my perception changes the complexity of the

Cite this column as follows: John McGregor: "Complexity, its in the mind of the beholder", in *Journal of Object Technology*, vol. 5, no. 1, January - February 2006, pp. 31-37, http://www.jot.fm/issues/issue_2006_01/column03

software or just that devices that alter my perception allow me to work in a more efficient way?

While that conversation was rattling around in my head I came across the following quote. “The complexity of software is an essential property, not an accidental one. Hence, descriptions of a software entity that abstract away its complexity often abstract away its essence.”[Brooks 95] Does this mean that efforts to apply abstraction to reduce complexity must lose valuable information?

Brooks is referring to an often used classification of complexity: accidental and essential. Essential compexity arises from the nature of the problem and how deep a skill set is needed to understand the problem. In some software development projects we really are dealing with rocket science. Accidental complexity is the result of poor attempts to solve the problem and may be equivalent to what some are calling complication. Applying the wrong design pattern or selecting an inappropriate data structure add accidental complexity to a problem.

Another classification of complexity defines the categories of psychological and structural complexity. The psychological complexity of software refers to those characteristics of software that make it difficult to understand [Curtis 79]. Structural complexity is partially described by Boydston, “Complexity of programming increases as the lines of code per module and the number of modules to interface increase.” [Boydston 84] These definitions point out two very different perspectives on complexity.

I spend a lot of time with clients taking unnecessarily complicated views of the world and simplifying them. Often I do this by “teasing apart” closely related but different concepts. I want to do this with “complexity” and “complicated” before addressing strategic issues related to complexity.

2 COMPLEX VS COMPLICATED

“This is a complex system that has been made needlessly complicated,” says the reviewer. A complex system is one that comprises many parts and whose parts interact with one another. It is desirable that the parts encapsulate portions of the system and often the parts will hide the complexity of the encapsulated functionality. Design rules drive us toward modularity and complex systems, i.e. lots of pieces. Some design rules even direct us to consider interactions as a primary focus for interface design. A complex system takes time to understand because of the many interfaces but has a rational structure that, given time, can be understood and exploited.

A complicated system is difficult to understand and analyze. The system may or may not be complex. A complicated system often has no architectural coherence so studying the system’s structure may not lead to any greater understanding of how it works. A complicated system may be more efficient than a complex one. That is, the seemingly incoherent structure may have been established by the piecemeal effort to engineer performance into a system after the fact.

A former collaborator of mine used to get a big smile after several refactorings of a problem and say “We are almost finished, its getting much simpler.” Most of us have experienced solving a problem that at first glance appeared to be simple but as we attacked it, the problem appeared more complicated as we discovered many different cases. After much work, the solution suddenly appeared to get simpler. Actually, we had reached the point at which we had a sufficiently complex solution that we understand sufficiently well to allow us to remove unnecessary complications. This refinement process looks something like Figure 1. Notice that the origin of both axes is “simple.”

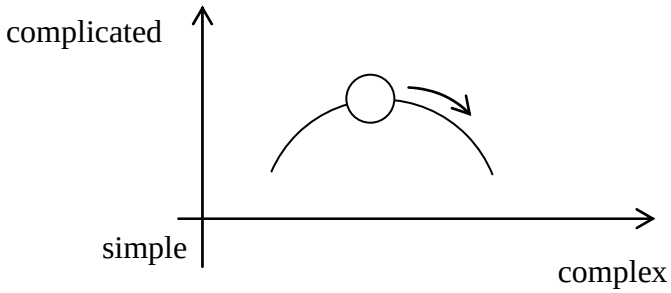


Figure 1 complexity vs complication

3 TOO MUCH OF A GOOD THING?

Being complicated is seldom a good thing, even in real time systems. Being complex may be necessary in order to achieve other qualities. Lets tease apart three more terms: encapsulation, information hiding, and modularity. A system is modular when there are pieces that can be removed and replaced with other pieces without disturbing th erest should be the rest of the system. Encapsulation is a technique for enclosing functionality within a construct that makes the functionality portable. For example, encapsulated pieces are structured so that their removal from a system involves only a single element. The package and class constructs are examples. Information hiding is a technique that limits access to the encapsulated functionality from outside the encapsulated bundle. Making only certain elements of a package or class public hides the rest.

Both encapsulation and information hiding do much to support modularity, but the design must still meet the needs of the product. Design is a process of trade-offs between qualities such as efficient performance and ease of maintenance. This is a balancing act. Conte et al. point out that “Over-modularization is as undesirable as under-modularization.” [Conte 86] Over-modularization leads to too many relationships to manage while under-modularization results in difficult maintenance of large pieces. Finding the right abstraction helps determine the optimum number and content of modules.

4 BUILDING COMPLEX, UNCOMPLICATED PRODUCTS

My original hypothesis was that the degree to which a problem is perceived to be complex is a function of the context in which it is to be solved. The number of people involved in the solution, their background and experience, and the time they are given to arrive at a solution affect their perception of the complexity of the problem.

Some problems are more complex than others to solve. Business problems are often complicated because of poor requirements analysis but seldom are very complex. Satellite communications systems may be complicated and usually are fairly complex. What techniques can we use to remove complications and manage the complexity?

Analysis

A thorough analysis of the problem, including a model of both the exact problem and the domain in which the problem is sited, provides a much less complicated path to a solution. The analysis allows us to understand the structure of the problem which often is a natural starting point for the structure of the solution. That is not to say that the requirements model has to be complete before other actions are taken nor does it prescribe a particular modeling approach. Test-first development uses test cases as the model of the problem.

As a professor I see students' answers on discussion questions on an exam that are long and rambling. They did not have an solution when they started and they wrote until they arrived at an answer. Attempting to implement the solution to a problem before the problem is understood usually results in an unnecessarily complicated solution. Refactoring can help with this provided there is sufficient time to do so. Refactoring requirements models as well as designs is essential to removing unnecessary complications.

Structure

A well-organized, clearly communicated software architecture, which achieves the desired qualities, is a first step toward a sufficiently simple structure for the product. The "clearly communicated" means that a rationale for each decision in the architecture has been provided. The deliberate structuring usually means that the architecture may be complex but not unnecessarily complicated.

The architecture is created by decomposing the monolithic solution into logical - using some definition of logical - pieces. This increases the complexity of the solution but can reduce unnecessary complication resulting in a solution that developers will be more able to understand.

Abstraction

Chose appropriate abstractions. Abstraction removes details from a view of the system but not from the system. As Brooks points out this approach can remove some of the

essence of the design. However, modern design techniques advocate using multiple views of a design, thereby preventing the loss of the "essence" of the design.

Abstraction is a useful technique for reducing the complexity of a design as perceived by the human viewing the design. It reduces complexity, not by reducing the number of elements, but by reducing the number of elements the human has to consider at any given moment.

The correct abstraction makes for a simpler design and uncomplicates the design. The wrong abstraction makes the design more complicated. In fact the wrong abstraction adds accidental complexity.

Interaction

Limit components in a complex system to local interactions. Complex systems are characterized by the interactions among system pieces. When these interactions begin to have global effects the system becomes unnecessarily complicated. The global effects are eliminated by the information hiding discussed in the previous section and by a high-quality software architecture discussed in the previous subsection.

The local interactions may result in emergent behavior. That is, in a complex system the totality of the system behavior is greater than the behaviors of the individual pieces. The well-defined architecture provides a basis for reasoning about the pieces and about how they interact.

Automation

Developers should operate on the appropriate representation. Automated generation of code from models usually results in complicated code and in many cases in a complex solution. The product users are unaware that the product is a more complicated solution than necessary, but the developers and maintainers will be acutely aware if they attempt to manipulate the generated code. This will not be a major concern if the evolution of automatically generated systems is carried out on the "source code", which in the case of automatic generation is the model from which a program is generated. Hacking the generated code is a tempting shortcut that has long-term negative impact. The maintenance personnel do not have to understand the structure of the generated code. In particular, if the solution is complex, modifying the pieces and their local interactions should be the major actions.

5 STRATEGIC IMPLICATIONS

Most pieces and systems are understandable and usable to the persons who constructed them. However, in today's development climate competitive advantage belongs to the group that can use pieces in multiple products and that can maintain products easily for a long time. This usually requires that many different people touch any given piece of a product.

This becomes a critical issue for a software product line. Core asset developers create many types of assets including code. These assets are then used by product builders to assemble products. The levels of reuse are very high. Each asset will be used by many product builders. No matter how complex the problem, the solution must not be unnecessarily complicated.

6 CONCLUSION

I have pulled apart the concepts of complicated and complex that are often lumped together. By considering these concepts in more detail we can identify techniques that enhance the design rather than degrade it. I have described some techniques that can be used to mitigate the effects of complexity on software development efforts. By producing designs that are more evolvable we achieve a competitive advantage over those who build products one stovepipe at a time.

REFERENCES

[Banker 93] Rajiv D. Banker, Srikani M. Datar Chris F. Kemerer and Dani Zweig. Software Complexity and Maintenance Costs, Communications of the ACM, v 36, n 11, Nov 1993.

[Boydston 84] Boydston, R.E. Programming cost estimate: Is it reasonable? In Proceedings of the Seventh International Conference on Software Engineering (1984), pp. 153 – 159.

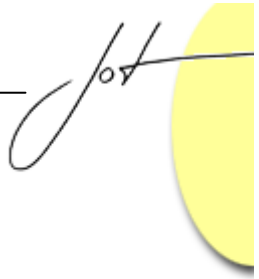
[Brooks 95] Brooks, Fredrick P. The Mythical Man Month: Essays on Software Engineering, Addison-Wesley, 1995.

[Conte 86] Conte, S.D., Dunsmore, H.E., and Shen, V.Y. Software Engineering Metrics and Models. Benjamin-Cummings, Reading, Mass., 1986.

[Curtis 79] Curtis, B., Sheppard, S.B., Milliman, P., Borst, M.A., and Love, T. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. IEEE Transactions on Software Engineering, SE-5, 2(1979), 96 – 104.

[UnivChicago 02] Univeristy of Chicago Magazine. The Complexity Complex. V95, no. 2, December 2002.

[Weissman 74] Larry Weissman. Psychological Complexity of Computer Programs: An Experimental Methodology, Software Engineering



About the author

Dr. John D. McGregor is an associate professor of computer science at Clemson University and a partner in Luminary Software, a software engineering consulting firm. His research interests include software product lines and component-base software engineering. His latest book is *A Practical Guide to Testing Object-Oriented Software* (Addison-Wesley 2001). Contact him at johnmc@lumsoft.com.