Growing Complex Software Systems

A Formal Argument for Piecemeal Growth in Software Engineering

Jerry Overton
Computer Sciences Corporation (CSC)
St. Louis, Missouri, USA
joverton@csc.com

***Abstract* – With piecemeal growth, complex systems are grown in a series of small steps rather than pieced together in one large lump. Although there are many specific examples (from agile methods) of piecemeal growth in software engineering; we argue that prior art has yet to produce general theoretical argument for building complex software systems this way. In this research, we propose a formal, theoretical argument for the general applicability of piecemeal growth to software engineering. As part of our argument, we infer both the requisites for piecemeal growth and some surprising connections between piecemeal growth and existing disciplines within software engineering.**

Keywords – Piecemeal Growth; Complex Systems; Software Engineering; Agile Methods; Software Design Pattern; Mathematics; Formal Method; POAD Theory

I. INTRODUCTION

A. The Nature of this Research

This paper presents the results of software engineering research. We begin with a brief discussion of the nature of this research to establish the proper paradigm for evaluating this work. While scientific problems are concerned with the study of existing artifacts and phenomena (the behavior of subatomic particles, the motions of planets, etc), engineering problems are concerned with how to construct new artifacts (bridges, buildings, and, in our case, software systems) [1]. While scientific research problems have an empirical nature, engineering (specifically software engineering) research problems do not. It is not possible to apply the same empirical validation methods used for scientific research to software engineering research [1].

Software engineering research is the study of processes by which people turn ideas into software [1]. Empirical data collected about these processes necessarily contain social and cultural aspects. Although empirical data may serve as an example to clarify the concepts presented here, it cannot objectively validate our results. Producing any such data and determining its correspondence with our results requires subjective interpretation.

In this paper, we advocate for the effectiveness of a particular software engineering approach using a structured argument. Ultimately our work is validated by whether or not the argument we present is convincing among practicing software engineers. To be considered convincing, the argument will have to generate interest and credibility. It will have to be circulated among a wider audience, polished and refined. Parts or all of the argument must be used by engineers to justify design processes of their own. We consider this work to be the first step in the process – we have recorded an argument so that it can be read, circulated, and scrutinized. For this paper, our goal is to produce an argument lacks identifiable errors or contradictions.

B. Piecemeal Growth and Software Engineering

Piecemeal growth is the process of building a complex system in small steps [2]; where nothing is ever completely torn down or erased. Additions are made, existing structures are embellished and improved [3]. This is different from modular design [4] where the system is composed from individual pieces snapped together. Consider the process by which the St. Mark's Square in Venice (Fig. **1**) was built – the example of piecemeal growth given in [5]. The process started in 560 A.D. with a small square basilica, where the castle of Doge (middle right in the picture) was built. In 976 A.D., two new buildings were added to the center of the basilica, including the tower shown in the middle of **Fig. 1**. By 1532 A.D., the tower became embedded in a rectangular building and the original basilica was extended.

 ISBN: 978-1-61208-221-9

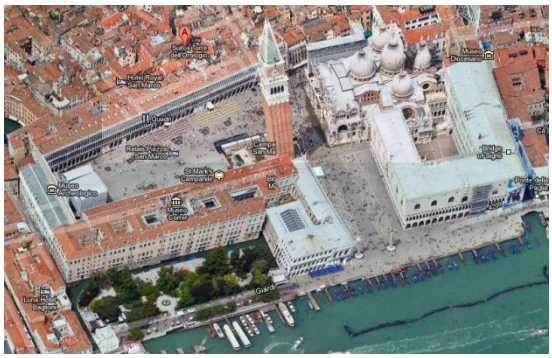

Fig. 1: St. Mark's Square in Venice [6].

St. Marks Square grew from a gradual sequence of changes rather than by assembling pre-fabricated parts. Each change mostly preserved the changes that came before. And each change contributed to the organic order seen in Fig. **1**. Because all acts of piecemeal growth have these characteristics in common [3]. We recognize a process as piecemeal growth, if it:

1. Specifies a sequence of operations
2. Each operation preserves the effects of all previous operations
3. Each operation solves part of a bigger problem
4. The sum effect of all operations solves the problem in its entirety

We will argue in the next section that although there are many specific examples of piecemeal growth in software engineering, there is nothing in the prior art that proposes an argument for the general applicability of piecemeal growth to building software systems. There is no theoretical argument that arbitrary, complex software systems can be built in a manner similar to the way St. Mark's Square was built. The general strategy of our argument is inspired by a technique used in mathematics – [6] argues that the Koch curve fractal exists by showing it to be the unique consequence of a particular equation. We argue that piecemeal growth is generally applicable in software engineering by showing it to be the unique consequence of a particular software engineering approach.

The rest of this paper is as follows. In Section II, we argue that software engineering is missing a general argument for the applicability of piecemeal growth to software engineering. In Section III, we introduce a system of mathematics needed to create that argument. In Section IV, we use our math to argue for the general applicability of piecemeal growth. In Section V and VI, we analyze our results and its significance.

II. STATE OF THE ART

The idea of piecemeal growth has made its way into software engineering practice through the adoption of agile methods [8] such as Extreme Programming (XP) [9], Scrum [10], and Crystal [11]. Agile methods assert that complex, well-designed software systems can be grown gradually through a process of continuous refactoring [12]. In this approach, software engineers do not put much emphasis on comprehensive analysis or design. Instead, they focus on building the highest-priority feature using the first reasonable approach that comes to mind. They refactor the results into a suitable design, and then repeat the process for the next highest-priority feature. The belief is that engineers can progress toward a solution piecemeal because refactoring makes it possible (and inexpensive) to make changes at any point.

Although the practice of piecemeal growth is known in software engineering as a part of agile methods, the actual idea of piecemeal growth is developed by prior works, such as [12], [13] and [14] that focus specifically on the practice of continuous refactoring. Instead of general arguments, these works all give detailed examples of how specialized refactoring techniques work to improve parts of a specific system. None of them propose an argument (although the premise is asserted) that, in general, continuous refactoring can be used to grow arbitrarily complex software systems.

For example, in [12], continuous refactoring is used as the basis for enabling piecemeal growth. The overall concept is developed using an introductory example. Although the example describes the basic idea of refactoring, it does not describe why this idea is useful beyond the specific example given. For the technique's broader application, the reader is asked to *"imagine [the example] in the context of a much larger system [12]."* In [13], continuous refactoring is asserted to be a proper basis for piecemeal growth in software engineering. The concept is illustrated by an example of evolving a new application framework for a legacy system, however, there is no argument for how to extend the techniques used in the example to the creation of systems not described in the example. In [14], the use of continuous refactoring for piecemeal growth is illustrated by an example of evolving a database for a simple financial institution. The work describes an example starting point for such a process, but

 ISBN: 978-1-61208-221-9

the process itself – and its applicability to systems other than what is exemplified – is asserted without further argument.

The goal of this research is to add to the current state of the art an argument for the general applicability of piecemeal growth to software engineering. Our approach is to come up with a mathematical model that characterizes the general existence of piecemeal growth in software engineering and a mathematical argument that piecemeal growth follows naturally as the result of a specific engineering approach. In the next section, we establish the system of mathematics needed to make our argument.

III. POSE AND POAD THEORY

The arguments in the next section will be based on a system of mathematics known as Problem-Oriented Software Engineering (POSE) [15] and Pattern-Oriented Analysis and Design (POAD) Theory [16]. In this section, we provide a summary of both.

In POSE, a software engineering problem has context, W; a requirement, R; and a solution, S. We write $W, S \vdash R$ to indicate that we intend to find a solution S that, given a context of W, satisfies R. Details about an element of the problem can be captured in a description for that element and a description can be written in any language considered appropriate. The problem, P_0 of designing a complex system can be expressed in POSE as:

$$CSystem: W, S \vdash R \qquad (1)$$

where W is the real-world environment for the system, S is the system itself and R are the requirements the system must meet. Equation (1) says that we can expect to satisfy R when the system S is applied in context W.

In POSE, engineering design is represented using a series of problem transformations. A problem transformation is a rule where a conclusion problem $P: W, S \vdash R$ is transformed into premise problems $P_i: W_i, S_i \vdash R_i, i = 1, \ldots, n\ (n > 0)$ using justification J and a rule named N, resulting in the transformation step $\frac{P_1 \ldots P_n}{P}\ \frac{[N]}{\ll J \gg}$. This means that S is a solution of $W, S \vdash R$ whenever $S_i, \ldots, S_n$ are solutions of $W_i, S_i \vdash R_i, \ldots, W_n, S_n \vdash R_n$. The justification J collects the evidence of adequacy of the transformation step. Through the application of rule N_i, problems are transformed into other problems that may be easier to solve. These transformations occur until we are left only with problems that we know have a solution fit for the intended purpose. POSE allows us to use one big-step transformation to represent several smaller ones. The progression of a software engineering solution described by a series of transformations can be shown using a development tree.

$$\dfrac{\dfrac{\overline{P_3: W_3, S_3 \vdash R_3} \quad P_4: W_4, S_4 \vdash R_4 \ \ [N_2]}{P_2: W_2, S_2 \vdash R_2 \ \ll J_2 \gg}\ [N_1]}{P_1: W_1, S_1 \vdash R_1 \ \ll J_1 \gg} \qquad (2)$$

In the tree, the initial problem forms the root and problem transformations extend the tree upward toward the leaves. There are four problem nodes in the tree: P_1, P_2, P_3, and P_4. The problem transformation from P_1 to P_2 is justified by J_1, the transformation from $P_2$to P_3 and P_4 is justified by J_2. The bar over P_3 indicates that P_3 is solved. Because P_4 remains unsolved, the adequacy argument for the tree (the conjunction of all justifications) is not complete, and the problem P_1 remains unsolved. A complete and fully-justified problem tree means that all leaf problems (in this case P_3 and P_4) have been solved.

For the sake of clarity, we will show the context, solution, and requirement of a problem only when necessary to understanding a given transformation. In many of the equations in section IV, these details are omitted and only the problem's name is shown. In general we adopt the practice of omitting any detail not required to support our argument. For example, we recognize that systems requirements often compete and designers must consider details such as how to trace from business requirements to system requirement to architectural choices. Although these considerations are important in the day-to-day practice of software engineering, they were not necessary to complete our argument for the general applicability of piecemeal growth in software engineering and were, thus, omitted from representation in subsequent formal models.

An $AStruct$ (short for Architectural Structure) [15] is used to represent an architecture in the solution. An $AStruct$, $AS\ [g\ ](c_1, \ldots, c_n)$ has a name, and combines a known structure, g (of arbitrary complexity), together with the c_i which are elements of the solution that are yet to be designed. Using the solution interpretation rule $SolInt$, we can modify the solution as follows:

$$\frac{W, AS[g](c_1 \dots c_n) \vdash R}{W, S \vdash R} \; [SolInt] \; \ll \; justify \;\; AS[\dots](\dots) \gg \tag{3}$$

Once an $AStruct$ has been applied, we can use the Solution Expansion transformation $SolExp$ to expand the problem context and refocus the problems to find the c_j that remain to be designed. For example, in the case where $n = 3$, we would have:

$$\frac{\begin{array}{c} W, g, c_2{:}\, null\,, c_3{:}\, null, c_1 \vdash R, \\ W, g, c_1{:}\, null, c_3{:}\, null, c_2 \vdash R, \\ W, g, c_1{:}\, null, c_2{:}\, null, c_3 \vdash R \end{array}}{W, AS[g](c_1, c_2\,, c_3) \vdash R}\, [SolExp] \tag{4}$$

where $null$ is used to indicate that nothing is known about that particular component. Using $null$ allows us to isolate a particular unknown element without making assumptions about any of the other unknown elements. The $SolExp$ transformation creates a number of premise problems. Each new premise problem requires solving and each premise problem contributes its solution to the other premise problems. Note that because the architecture being expanded has already been justified, the expansion of the architecture requires no further justification.

Software design patterns record the engineering expertise needed to justify the substitution of a complex, unfamiliar problems with simpler, more familiar one [17]. The basis of POAD Theory is that software engineering design can be represented as a series of transformations from complex engineering problems to simpler ones, with software design patterns used to justify those transformations:

$$\frac{SimplerProblem}{ConplexProblem} \; [SolInt] \; \ll Pattern_1, \dots, Pattern_n \gg \tag{5}$$

In (5) the engineering expertise in patterns $Pattern_1, \dots, Pattern_n$ are used to justify the replacement of the $ConplexProblem$ with the $SimplerProblem$.

In the next section, we used the mathematics of POSE and POAD Theory to argue that piecemeal growth can be used to create arbitrary complex software systems.

IV. THE EXISTENCE OF PIECEMEAL GROWTH

A pattern $Pattern_i$ tells us how to solve a problem by introducing an architecture and components modeled as follows:

$$\frac{W_i, AS_i[g_i](c_i) \vdash R_i}{Problem_i} \; [SolInt] \; \ll \; Pattern_i \gg \tag{6}$$

We could apply the solution expansion rule to the architecture introduced by $Pattern_i$, similar to what we did in (4). But suppose, instead, we were to study $Pattern_i$ and realize that there is a way to go about implementing the pattern's solution by breaking it into two problems: the problem of finding g_i (the problem of implementing the invariants of the pattern), and the problem of finding c_i (the problem of implementing the context-specific parts of the pattern). Suppose our research into $Pattern_i$ leads us to the engineering judgment (J_i) that there is a method for implementing the solution to the problem as follows:

$$\frac{Problemg_i\,, Problemc_i}{Problem_i{:}\, W_i, AS_i[g_i](c_i) \vdash R_i} \; [SolInt] \; \ll J_i \gg \tag{7}$$

Our research into $Pattern_i$ allows us to realize that we can solve $Problem_i$ by implementing $Pattern_i$ using a combination of $Problemg_i$ and $Problemc_i$. For the sake of clarity, we combine (6) and (7) into a single pattern of transformation.

$$\frac{\dfrac{Problemg_i, Problemc_i \; [SolInt]}{W_i, AS_i[g_i](c_i) \vdash R_i \;\; \ll J_i \gg}}{Problem_i} \; [SolInt] \; \ll \; Pattern_i \gg \tag{8}$$

is shortened to

$$\frac{Problemg_i, Problemc_i}{Problem_i} \; [SolInt] \; \ll J_i, \; Pattern_i \gg \tag{9}$$

In the original application of $Pattern_i$ to $Problem_i$, the component g_i acts as context for the component c_i. The solution to $Problemc_i$ will operate within the context of the solution for $Problemg_i$. This subtle relationship between the solution to $Problemg_i$ and the solution to $Problemc_i$ will be important later in our argument.

Suppose we had a $Complex\ Problem$ that we wanted to solve. Suppose that we found a set of transformations patterned after the one in (9) that we could apply in sequence to the $Complex\ Problem$ as follows:

$$\frac{Probg_1, \dfrac{\dots, \dfrac{Probg_n, Probc_n \;\; [SolInt]}{\dots \;\; \ll J_n, Ptn_n \gg}}{Probc_1} \dots \;\; [SolInt]}{Complex\ Problem{:}\, W, S \vdash R} \; \ll J_1, Ptn_1 \gg \tag{10}$$

We know that our completed solution will be composed of $n+1$ interrelated problems and n patterns. The solution to $Probg_{i+1}$ will operate within the context of $Probg_i$. The $Complex\ Problem$ is solved by finding solutions to all leaf-level problems $Prob\ g_1, \ldots, Prob\ g_n, Prob\ c_i$ For short, we can write (10) as the pattern sequence [17] :

$$[\ll J_1, Ptn_1 \gg, \ldots, \ll J_n, Ptn_n \gg] \qquad (11)$$

The sequence of (11) is a model of the analysis process required to find a solution to the $Complex\ Problem$ – the patterns Ptn_i needed to solve the problem, the implementation strategies J_i that must be used for each pattern, and the order in which each pattern and implementation strategy must be applied.

Suppose we completed our analysis by finding solutions to all leaf-level problems as follows.

$$\frac{\overline{Solng_1}\ [SolInt]}{Probg_1 \ll Jg_1 \gg'} \ldots, \frac{\overline{Solng_n}\ [SolInt]}{Probg_n \ll Jg_n \gg'} \qquad (12)$$

$$\frac{\overline{Solnc_n}'\ [SolInt]}{Probc_n \ll Jc_n \gg}$$

where$Solng_i$ is part of a specific implementation of the pattern Ptn_i, and Jg_i is convincing justification that $Solng_i$ is adequate to solve $Probg_i$. Just as we did with (10), we can (12) using the following sequence:

$$[\ll Jg_1, Solng_1 \gg, \ldots, \ll Jg_n, Solng_n \gg, \qquad (13)$$

$$\ll Jc_n, Solnc_n \gg]$$

Whereas (11) is a model of the analysis process needed to find a solution to the $Complex\ Problem$, (13) is a model of the design process required to realize the solution. It describes the specific implementation needed to solve each outstanding problem and justification for why each implementation works. The sequence of (13) can be interpreted as the ordered steps of piecemeal growth required to solve the $Complex\ Problem$.

Recall from Section I, the criteria 1-4 for recognizing a process as piecemeal growth. Equation (13) specifies a sequence of operations. Each step $\ll Jg_i, Solng_i \gg$ in the sequence results in $Solng_i$ – a partial solution to the $Complex\ Problem$. We know that step $i+1$ of (13) preserves step i because, from earlier analysis, we know that $Solng_{i+1}$ acts entirely in the context of $Solng_i$. We also know from Section III that the solution to the $Complex\ Problem$ is the collection of all solutions $Solng_1 \ldots Solng_n$. Thus, we have completed our argument that a piecemeal-growth solution to the $Complex\ Problem$ exists, and that the piecemeal-growth solution can be characterized as the sequence of steps given by (13).

In the last section, we analyze the significance of our efforts.

V. ANALYSIS

We showed piecemeal growth to be a consequence of the engineering strategy of (9), and that piecemeal growth requires the analysis process modeled in (11). We started by looking for a solution to the $Complex\ Problem$. This problem is an arbitrary software engineering problem in that the only assumption that we made was that the $Complex\ Problem$ has an arbitrarily large number of requirements. We made a single assumption (9) about the strategy for solving the problem and found a solution by working through the consequences of that one assumption. As a consequence, we satisfied the n requirements of the $Complex\ Problem$ with the $n+1$ problem-solving transformations represented by the sequence in (13) – which happened to properly characterize piecemeal growth. The progression we went through is an argument that piecemeal growth is applicable to arbitrary complex problems in software engineering. Piecemeal growth has some very specific characteristics (the criteria 1-4 from Section I). Yet, by starting only with an engineering assumption made independently of the decision to use piecemeal growth, we were able to derive an abstract mathematical representation that matched our characteristics of piecemeal growth.

We recognize that the formal models presented in Sections III and IV would be easier to understand if accompanied by a comprehensive example. However, creating such an example is outside the scope of the current research – our goal, here, was to record the argument in enough detail to allow it to be circulated and scrutinized. In future research, as the argument is polished and refined, it will become essential to supplement the formal model with a running example.

VI. CONCLUSIONS

In (13) $Solng_{i+1}$ acts entirely in the context of $Solng_i$ – a relationship that can be satisfied using abstraction and refinement [18] – note that a refinement works entirely

 ISBN: 978-1-61208-221-9

within the context of an abstraction. With this realization, we can begin to imagine specific tactics for general piecemeal growth: each step is a refinement of the previous step, and an abstraction for the next.

One of the more interesting questions in piecemeal growth is whether or not systems can be grown without the help of up-front planning [19]. Can we solve (or begin solving) the $Complex\ Problem$ without first completing some kind of detailed analysis? We know that the progression (13) from the $Complex\ Problem$ to its solution required the analysis shown in (11). If we were to proceed without up-front planning, we would have to derive the sequence of (11) as the result of progressing along the sequence of (13). As we implemented each of our interim solutions, we would have to be able to derive our next problem based on our current solution. More formally, we would need to be able to derive $Probg_{i+1}$ from $\ll Jg_i, Solng_i \gg$. Equation (10) implies that the minimum linking them is $\ll J_{i+1}, Ptn_{i+1} \gg$. That is, the minimum requisite for successful progression through the piecemeal growth of (13) is that one must be able to derive the $n+1^{st}$ step of the analysis (11) while one performs the n^{th} step of the design (13). This may be possible if one can anticipate how to structure $Solng_i$ so that it can act as the context for $Solng_{i+1}$. In other words, our argument implies that piecemeal growth without detailed planning is possible only if, at each step, one can successfully anticipate and accommodate the invariants of the next.

The idea that one must be able to anticipate future invariants suggests a potentially novel approach to piecemeal growth and a link between piecemeal growth and predictive analytics. Our argument suggests that all that is really needed to proceed with each step of piecemeal growth are the invariants of the next step. It may be possible predict all required invariants by performing a cluster analysis [20] on a complete set of system description documents. The resulting clusters and their dependencies may be interpreted as a map of the system's invariants. It may be interesting to explore whether or not it is practical to establish a community of software engineers that grow (piecemeal) complex software systems guided by architectures mined from collections of plain-text descriptions of what users would like the system to accomplish. For example, it may be possible to use crowdsourcing to efficiently produce a comprehensive set of description documents for a complex system, predictive and visual modeling to create a reliable map of that system's invariants, and piecemeal growth to build the system gradually over time using a long series of small, inexpensive acts of systems development guided by the derived map of invariants.

VII. REFERENCES

[1] M. Lázaro and E. Marcos. *Research in Software Engineering: Paradigms and Methods.* Advanced Information Systems Engineering, 17th International Conference. Proceedings of the CAiSE 05 Workshops, 2005, pp. 517-522

[2] C. Alexander. *A Timeless Way of Building.* Oxford University Press, 1979.

[3] C. Alexander. *The Oregon Experiment.* Oxford University Press, 1975.

[4] K. Sullivan, Y. Cai, B. Hallen, and W. Griswold, *The Structure and Value of Modularity in Software Design.* Proceedings, ESEC/FSE, 2001, ACM Press, pp. 99-108

[5] C. Alexander. *A Vision of A Living World.* The Center for Environmental Structure, 2005.

[6] Maps.google.com. Last Accessed: 03/15/2012

[7] H.O. Peitgen, H. Jurgens, D. Saupe. *Chaos and Factals: New Frontiers of Science.* Springer, 2004.

[8] J.O. Coplien, N.B. Harrison. *Organizational Patterns of Agile Software Development.* Prentice Hall, 2004.

[9] K. Beck. *Extreme Programming Explained: Embrace Change,* Second Edition. Addison-Wesley, 2004.

[10] K. Schwaber, M. Beedle. *Agile Software Development with SCRUM.* Prentice Hall, 2001.

[11] A. Cockburn. *Agile Software Development.* Addison-Wesley, 2001.

[12] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 2000.

[13] J. Kerievsky. *Refactoring to Patterns.* Addison-Wesley, 2005.

[14] S. W. Ambler, P. J. Sadalage. *Refactoring Databases.* Addison-Wesley, 2006.

[15] J. G. Hall, L. Rapanotti, and M. Jackson. *Problem-Oriented Software Engineering: Solving the Package Router Control Problem.* IEEE Trans. Software Eng., 2008. doi:10.1109/TSE.2007.70769

[16] J. Overton, J. G Hall, and L. Rapanotti. *A Problem-Oriented Theory of Pattern-Oriented Analysis and Design.* 2009, Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, pages 208-213, 2009.

[17] F. Buschmann, K. Henney, and D. Schmidt. *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages,* Volume 5. John Wiley & Sons, West Sussex, England, 2007.

[18] D. F. D'Souza, A. C. Willis. *Objects, Components, and Frameworks with UML.* Addison-Wesley, 1998.

[19] A. Dagnino, K. Smiley, H. Srikanth, A. I. Anton, and L. Williams, *Experiences in Applying Agile Software Development Practices in New Product Development*. In Proceedings of Proceedings of the Eighth IASTED International Conference on Software Engineering and Applications, Nov 9-11 2004 (Cambridge, MA, United States, 2004). Acta Press, Anaheim, CA, United States.

[20] I. H. Witten, E. Frank. *Data Mining Practical Machine Learning Tools and Techniques.* Elsevier, Inc. 2005.

 ISBN: 978-1-61208-221-9