Simulating the Structural Evolution of Software

Benjamin Stopford[1], Steve Counsell [2]

[1] School of Computer Science and Information Systems,
Birkbeck, University of London
[2] School of Information Systems, Computing and Mathematics,
Brunel University, London

Abstract. As functionality is added to an ageing piece of software, its original design and structure tends to erode. The underlying forces which cause such degradation have been the subject of much research. However, progress in this field is slow due to the difficultly faced in generating empirical data [6] as well as attributing observed effects to the various points in the causal chain [7]. This paper tackles these problems by providing a framework for simulating the structural evolution of software. A complete model is built by incrementally adding modules to the framework, each of which contribute an individual evolutionary effect. These effects are then combined to form a multi-faceted simulation that evolves a fictitious code base approximating real world behavior. Validation of a simple set of evolutionary parameters is provided, demonstrating agreement with current empirical observations.

1 Introduction

Software evolution is a complex phenomenon and deriving formulations for the interactions that make up its whole is a significant challenge. In fact, software science possesses no theoretical framework to describe its evolution. There are nevertheless, a variety of behavioral observations and heuristics that describe the evolution of software. Examples vary from laws of software evolution, such as those proposed by Lehman [8], to more specific underlying behaviors such as the coupling types of Briand et al. [1]. Simulating rules individually is within the bounds of a software model and combining such effects would provide an interesting basis for experimentation.

This paper presents a method for exploring software evolution from the inside out. Individual laws can be proposed and added to a simulation framework. The effects of these laws can then be measured in isolation, under different environmental conditions and against other proposed laws.

The majority of research in software engineering simulation is concerned with the simulation of software process. Prominent examples of this include the modeling of project planning [4], defect levels and staffing profiles [9] as well as system size and effort trends [10]. The aims of process simulations are to investigate the processes by which people, technology and practices are organized to transform information, materials and energy into a piece of software. Conversely, the focus herein addresses the effect that evolution has on the structure of software at a source code level and how this structure varies over the evolution of a project. It is thus the code structure that is under analysis rather than the process through which it is generated.

2 The Simulation Model

The framework is based around a fictitious code base which defines the basic rules of software development such as the existence of classes and methods as well as their means of interaction. Agents (simulated developers) then evolve this code base through the addition and processing of requirements. The specifics of evolving and measuring the system are left to customizable plug-ins which can be tailored to fit individual experimental aims. The framework thus presents a controlled environment that enforces the evolution of the code base in a realistic manner - the direction that this evolution takes rests in the hands of the experimenter. The proposed model follows a simple feedback network. Its four basic elements are requirements, evolution, code metrics and the code base. The latter three are connected in a feedback circuit as shown in Figure 1:

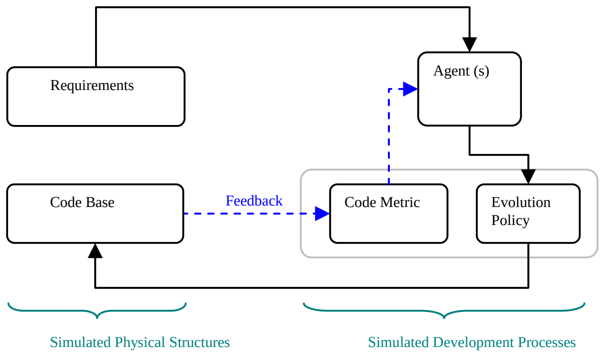

Figure 1. An overview of the basic elements of the simulation framework and the data flows between them. The feedback loop from the Code Base back to the Agent via the Code Metric is also shown.

1. Requirements: Requirements are generated through a stochastic, configurable process and can be reused across experiments or created afresh. Requirements control the conceptual content of the simulation to later be turned into code constructs.
2. Evolution: The Agent and Evolution Policy evolve the code base using the requirements specified. Evolution is concerned with turning the hierarchy of requirements of different types into a structure of code constructs.
3. Measurement: Code metrics provide a means for the Agent to evaluate the code base prior to changing it. This supplies the closing section of the feedback loop in which agents can respond differently depending on their observation of the code base.

4. Code Base: The evolution of the relationships between physical code constructs is modeled inside the Code Base. The simulation considers only entities greater than, or at method level.

A run of the simulation starts with the generation of a set of requirements. These are then passed to an Agent to implement. The Agent implements the requirements through the use of an evolution policy specified for the particular experiment. The evolution policy defines a set of rules dictating how to structure code as it is added. The evolution policy can also take into account information on the current state of the code base fed back to it from the code metrics.

The evolution of the code base is measured using a cost function. The cost function is an arbitrary measure that can be used to compare the relative costs of different runs of the simulation. The costing model is split into implementation cost and metric cost. The former is the cost associated with creating and altering code. The latter is the cost output by the metric.

The code metrics allow experimenters to model the cost of comprehending different structures as well as a means for agents to observe the code base. The act of observing the code base through metrics causes information to be fed back from the code base into the evolution policy so that the code base structure can influence how it is evolved. This is important since it allows the state of the code base to alter the evolutionary decisions made by agents. Such feedback loops, formed from simple concepts, are responsible for many of the processes observed in complex systems [3]. As such, the simulation can create responses that are likely to differ significantly from those formed by static analysis.

2.1 Requirements

Requirements control the conceptual content of the simulation to later be turned into code constructs; the separation of requirements from evolution is important. Running with different requirements allows the simulation to model different development environments (for example green field developments vs. mature products). Experiments can then either hold the requirements constant or deliberately vary them to explore how they effect the simulation. Requirements are generated though a stochastic process and can be serialized and reused across different experimental runs.

2.2 The Code Base

The code base acts as a repository for different code constructs created and linked together by Agents. These constructs can then refer to one another via the various calls made open to them by the simulation framework such as “Reference” or “Create Function”. The code base encapsulates all creational calls and references so that responsibility for enforcing integrity within the resulting code is retained.

The code constructs used by the simulation are based on the work suggested by Kelsen [5]. They include Classes, Functions, Events, Properties and References. Classes and Functions represent standard classes and functions that might be encountered in a real code base. Events denote an interaction with an event outside of the system. Properties represent the internal storage of state through variables of a specified type. References link code constructs in a directional manor. References also specify a Coupling Type, determined by the evolution policy. Coupling types define the effect that a reference has on the code construct it operates upon, for example describing whether data is retrieved or changed through the coupling.

The execution path of the simulation starts at one of the system events. The code base in the simulation is constructed in such a way that this execution path is always enforced.

2.3 The Agent

The Agent is a system concept that embodies the role of a developer in a real software project. Agents are stateful with the ability to 'learn' about the system as they modify and add to it; the agent's primary concern is to facilitate the conversion of requirements into code using an "Evolution Policy". The Evolution Policy is the plug-in responsible for turning requirements into code. The agent is responsible for facilitating this (for example, by locating the class to change).

Each agent has a memory of the code constructs that they were responsible for implementing. This memory dissipates as time elapses in the simulation. An agent's memory can be accessed from the evolution policy or complexity metric to improve the depth of the simulation, particularly when considering multiple agents acting on the code base. When multiple agents are configured, each new requirement is implemented by an agent selected randomly from the pool.

2.4 The Evolution Policy

The Evolution Policy is the plug-in that bears responsibility for evolving the code base and is thus a focal point for defining experiments. The experimenter must implement three functions in the Evolution Policy in response to the major categories of requirement type: New, Change and Augment. In addition, the evolution policy also provides a set of utilities that allow the experimenter to customize their implementation. These include:

Code Metric: The evolution policy uses a code metric to retrieve feedback from the code base before making a change. The code metric also records a cost used as a measure of the experiment (see section 2.5).

Memory: The memory of the agent is accessible from the evolution policy. This provides feedback on the agent's recall of the various code constructs that they created.

Coupling Type: A specific coupling type is associated with References as they are created detailing the nature of interactions made through references.

2.5 Measurement

The simulation is measured via a cost function that provides a measure for comparing different runs. The total cost is split into two different sections that indicate the separation between the cost of creating and the cost of understanding code.

Metric Cost is calculated by the code metric plug-in. This provides the means for customizing experiments by allowing an experimenter to specify how the evolutionary factors modeled in the experiment should be measured.

Implementation Cost is that incurred through the physical creation of code. This is calculated automatically and is proportional to the number and type of code constructs created.

2.6 Complexity Injection

Complexity Injection is a feature of the framework that allows a random distribution of extra features (references, properties etc) to be added to a code construct when it is created. This allows the complexity of the simulation to be controlled without altering the logic in the evolution policy.

The Complexity Injector and the Evolution Policy have similar, but fundamentally different, roles. The Complexity Injector is responsible for the monotonous detail added to all code constructs when they are created (classes need functions and references, etc). The Evolution Policy is responsible for shaping how the structure between classes and functions evolve.

2.7 Default Plug-In Implementations

A default set of plug-ins are supplied and shipped with the simulation. They define a basic set of policies through which the code base can be evolved and are used in the validation experiments presented in this paper. It is anticipated that future experiments will improve on the basic assumptions they make, incorporating more realistic evolution policies and metrics. To this end, they are created in an extensible manner.

3 Using the Framework to conduct Experiments

The Framework includes a GUI designed to ease the comprehension of the code base structure during a simulation run. The GUI has three views: one for the requirements and two for the code base. The code base views include a graphical representation of the class hierarchy and can be drilled into by the user. A second view represents the set of execution paths. More structured analysis can be performed using data provided through an output data file.

The method for conducting an experiment is:

1. Identify the problem to be investigated and develop a dynamic hypothesis that describes its cause.
2. Create an evolution policy plug-in that changes the code base according to the dynamic hypothesis.
3. Amend the code metrics plug-in to ensure that it is sensitive to the evolutionary changes expected.
4. Test the evolution policy and metric in isolation to ensure that provide the expected behavior.
5. Add the implemented policy to the full simulation model so that it can be investigated in conjunction with other existing simulated factors.

4 Validation of the Simulation Framework

The simulation framework is validated through a suite of tests that analyze performance over different experimental conditions. The aim of each test is to validate a basic behavior of the system against an intuitive understanding or empirical observation.

4.1 Validation (1): Linear Evolution of Code Base Size

Empirical observations of the increase in size of an evolving code base, as measured by Capiluppi et al [2], show a linear increase in size over time. The simulation framework was used to reproduce this behavior using the default plug-ins. Both the original and simulated results show a linear increase corroborating this basic behavior.

Capiluppi et al. also provide a study of the distribution of average lines of code per file over various releases stating that the average number of lines per file should increase slightly as the system evolves. A comparable result was generated with the simulation framework.

4.2 Validation (2): The Effect of Requirement Type

An important function of the simulation is its ability to respond to different types of requirement in a distinct manner. This aspect is validated by measuring how Requirement Types affect evolution of the code base and ensuring that this agrees with expected behavior. The proposition is that developments simulated from requirements that include a high degree of re-visitation will cost more to develop. This assumption is made from the real world observation that existing code is harder to change (as it must be understood). The results in Figure 2(a) corroborate this hypothesis with the cost being significantly higher for simulations that have to revisit code.

4.3 Validation (3): Response to Different Numbers of Agents

The simulation provides a facility for specifying the number of agents that contribute to evolution. Each agent “remembers” the code they created and this memory is taken into account by the default code metric. The effect of this is to drop the associated metric cost incurred by an Agent that is changing code they were responsible for creating (and that they therefore remember). This effect was validated via the experiment results displayed in Figure 2(b). These show that development with two agents is most efficient and the one with thirty agents is least efficient. Where there are fewer developers, the cost is lower as each developer is responsible for the original construction of a higher proportion of the code base (and thus has less to learn). This validates the simulation through observation of an expected behavior.

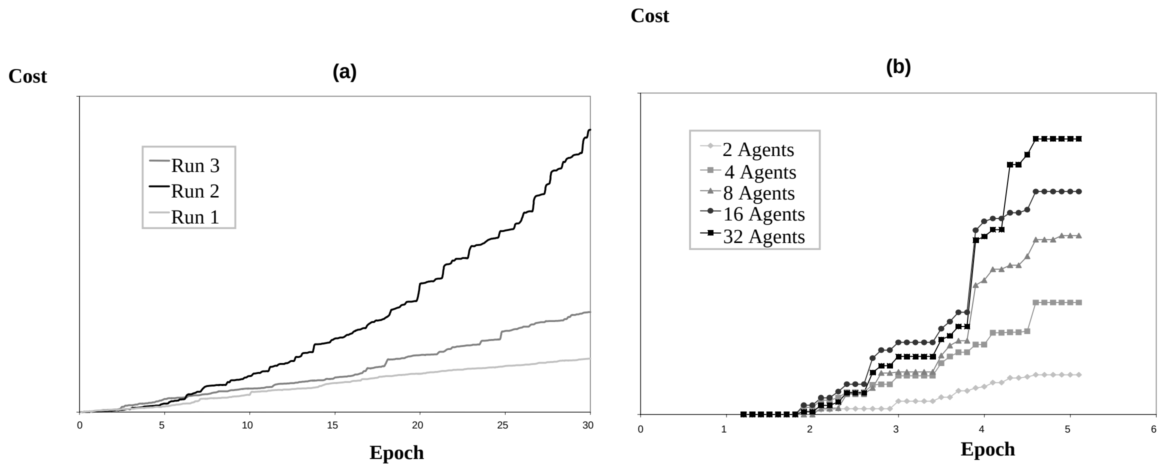

Figure 2: (a) Metric Cost for Simulations with varying requirement types. Run (3) is the control, Run (2) represents requirements that incorporate a large proportion of changes and Run (1) is predominantly new requirements. **(b)** Evolution with different numbers of agents. The different profiles result from the effect of agent memory.

The results presented in this section provide a level of confidence that the simulation performs in a manner approximating real world behavior. This conclusion is corroborated by both intuitive expectations and empirical results.

5 Conclusions

The evolution of software, in particular its structural erosion over successive generations is a primary concern of software engineering today. This paper presents a novel approach for investigating this problem. From an empirical standpoint, the simulation can be calibrated with a relatively small amount of empirical data. Once calibrated, the scope can be broadened to include different environments with little or no effort. This reduces the need for long and expensive empirical investigations.

From a theoretical standpoint, the simulation can be used to build a causal model of software evolution from individual behaviors. These behaviors can be investigated in isolation as well as part of a collective model. Such a bottom-up approach cannot easily be replicated by any other method.

Much as in other disciplines, simulation may provide a valuable window into a world otherwise inaccessible to current research, expediting the crystallization of laws as well as opening the doors to new insights. Full source code for the simulation framework can be found at:

http://www.benstopford.com/devsim/devsim.shtml

6 References

[1] “A Unified Framework for Coupling Measurement in Object-Oriented Systems”: L.C. Briand, J.W. Daly, J.K. Wust: IEEE Transactions on Software Engineering, Vol 25, No 1, Jan/Feb 1999

[2] “Studying the Evolution of Open Source Systems at Different Levels of Granularity”: Capiluppi, Morisio and Ramil: Proceedings of the 12th International Workshop on Program Comprehension

[3] “Urban Dynamics” Forrester, J. W. Cambridge MA: Productivity Press. 1969.

[4] “Software Process Modeling Support for Management Planning and Control”. Kellner, M. Proceedings of the first international conference on the software process 1991.

[5] “A Simple Static Model for Understanding the Dynamic Behavior of Programs.” Kelsen: International Workshop on Program Comprehension 2004

[6] “Need for more Longitudinal Studies of Software Maintenance”: C.F. Kemerer, S. Slaughter, Proc. Int’l Workshop Empirical Studies Software Maintenance, Monterey Calif., 1996

[7] “An Empirical Approach to Studying Software Evolution”: C.F. Kemerer, S. Slaughter, IEEE Transactions on Software Engineering, Vol. 25, No. 4 July/August 1999

[8] "Program, Life Cycle and the Law of Program Evolution", M. Lehman, Proceedings of the IEEE, 68, 1060-1078, 1980

[9] “Modeling Software Processes Quantitatively and Assessing the Impact of Potential Process Changes on Process Performance” Raffo, D. – PhD Dissertation. Carnegie Mellon University 1996

[10] “Software Process White Box Modeling for FEAST/1”, Wernick and Lehman: Journal of Software Systems 1999