A Framework for the Simulation of Structural Software Evolution

BENJAMIN STOPFORD
School of Computer Science and Information Systems,
Birkbeck, University of London
and
STEVE COUNSELL
School of Information Systems, Computing and Mathematics, Brunel
University, Uxbridge, UB8 3PH.

As functionality is added to an aging piece of software its original design and structure will tend to erode. This can lead to high coupling, low cohesion and other undesirable effects associated with spaghetti architectures. The underlying forces which cause such degradation have been the subject of much research. However, progress in this field is slow as its complexity makes it difficult to isolate the causal flows leading to these effects. This is further complicated by the difficulty of generating enough empirical data, in sufficient quantity, and attributing such data to specific points in the causal chain. This paper describes a framework for simulating the structural evolution of software. A complete simulation model is built by incrementally adding modules to the framework, each of which contributes an individual evolutionary effect. These effects are then combined to form a multi-faceted simulation that evolves a fictitious code base in a manner approximating real-world behavior. We describe the underlying principles and structures of our framework from a theoretical and user perspective; a validation of a simple set of evolutionary parameters is then provided and three empirical software studies generated from Open-Source Software (OSS) are used to support claims and generated results. The research illustrates how simulation can be used to investigate a complex and an under-researched area of the development cycle. It also shows the value of incorporating certain human traits into a simulation - factors that, in real-world system development, can significantly influence evolutionary structures.

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. While the software engineering community has been studying software evolution for a considerable amount of time and made much progress [Leh80, Mens04], the increased importance and popularity of Open-Source Software (OSS) over the past few years have serious implications for both our analysis and understanding of evolutionary software forces. Data on the number of developers, discovered faults, severity of those faults and fixes are a standard part of an OSS resource such as sourceforge.net [Sou07]; moreover, OSS are increasingly becoming part of the development strategy of organizations and being downloaded in increasing numbers by those organizations. A series of recent empirical studies have investigated the features of evolving OSS both from an empirical and simulation perspective [Raffo00, Din04, Fer04, Smith06, Smith05]; some research on the criteria for when to adopt simulation (and why) has also been investigated [Kelln99]. One of the main drawbacks of these empirical studies is the relatively small amount of data that can be generated and analyzed by a single research project using standard statistical techniques. With the advent of OSS, an emerging challenge for the simulation community is to benefit from the vast amount of software project data available on-line and to use that data to inform the simulation modeling process. Equally important is for data used in the simulation modeling process to inform software development. This is one motivating factor for the research contained in this paper; the primary goal of the research is to develop a simulation model which investigates the macroscopic evolution of software as the policy by which it evolves, i.e., at the function level rather than the 'microscopic' code level.

Preventing the decay of systems as they evolve is a further, pressing challenge faced by the software engineering community. Techniques such as refactoring [Fow99] have been proposed as a way of reversing that decay; empirical studies are also a useful mechanism for establishing traits in evolving software and highlighting areas that may need maintenance attention. However, both techniques are time-consuming. While refactorings tool exist to aid the automation of certain refactorings, the amount of knowledge that can be generated about system evolution is limited. Equally, empirical studies require costly data collection (often manual) and complex statistical analysis. The framework we describe can be calibrated with a relatively small amount of empirical data and once calibrated its scope can then be extended with very little effort. Additionally, only a small set of empirical data is required as a seed. Thereafter, experiments can be run and re-run to generate further measurements; this is both cheaper in time and cost than using similar empirical methods or refactoring techniques and further motivates the research described in this paper. We therefore present a method for exploring software evolution from the 'inside out'. Individual rules can be proposed by the user, added to a simulation 'framework' and the effects of those rules then analyzed. For example, we could propose a rule imposing an upper limit on the number of methods that an Object-Oriented (OO) class comprises. When a class reaches a certain size, it is refactored [Opd92, Fow99] to move sections into delegate classes. An example experiment would involve investigating the impact of this principle as a system evolves. The role of evolving and measuring the system is left to an evolution policy which can be tailored to fit individual user aims.

The paper is organized as follows. In Section 2, related work is described. The simulation framework is presented including a description of its four basic elements. In Section 3, we elaborate on the basic components of the framework. We then discuss the use of the framework from a user's perspective, providing an insight into how the different policies and plug-ins are defined and configured, respectively (Section 4). The experiments using the simulation framework are then described in Section 5 supported by three previous empirical studies (Section 6). We then present a discussion of some of the issues raised by the research (Section 7) before concluding and pointing to further work in Section 8.

2. RELATED WORK

A variety of behavioral observations and heuristics that describe the evolution of software have been proposed. Examples vary from laws of software evolution proposed such as those by Lehman [Leh80] to more specific underlying behaviors such as coupling types of Briand et al. [Bria99a, Bria99b]. Simulating combined rules is within the bounds of a software model and provides an interesting basis for experimentation. 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 [Set06], defect levels and staffing profiles [Raffo96] as well as system size and effort trends [Wern99]. The aim of process simulation is 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 is on how code is structured as a physical entity and how this structure varies over the evolution of a project - it is the structure of software at a *source code* level that is under test. Kellner et al., [Kelln99] note that simulation can be used for modeling systems that display three behaviors; namely, system uncertainty and stochasticity, dynamic behavior and feedback mechanisms. Software is known to exhibit such behaviors and, hence, each one is incorporated into the simulation framework presented in this paper. An empirical study by Capiluppi et al., [Cap04] of the ARLA open source system found that the number of source files grew linearly as the project evolved. This is comparable to the simulated growth of class files being linear with a positive gradient. The same research also provided a study of the distribution of average lines of code per file over various releases. Their results showed the average number of lines per file increasing slightly as the system evolved. This increase was approximately linear with a small positive gradient.

We have supported the results of the framework with evidence from several empirical studies. At the heart of an empirical study is the use of software metrics [Fen98] that can be used to quantitatively and/or qualitatively assess a set of hypotheses. For example, in the case of our framework, a metric could be based on standard software metrics such as those suggested by Chidamber and Kemerer [Chid94] or Halstead [Hals77]; alternatively, a more specific metric that reflects the precise experimental aims could be used. The research in this paper draws on many software engineering disciplines. One area of direct relevance is that of the automation of program restructuring [Gris93]. There are also strong ties with attempts at modeling evolution through change history analysis [Bie03, Rob06, Gir06]. In our analysis, one guide to evolution of a system according to the simulation framework is through a metric and implementation cost breakdown; previous research has also addressed this as an important aspect of evolutionary processes [Snee04]. We note that the research described in this paper is a significant extension to the work first described in [Stop06]. In that paper, only the basic model was described and only a summary of initial, preliminary experimentation provided. In the next section, we describe an outline of the simulation framework and its constituent parts. The simulation model comprises four key elements and we describe each in detail.

3. THE SIMULATION FRAMEWORK

Simulations are designed to exhibit characteristics of real-world systems without replicating the complexity of the physical implementation. Thus, simulation is most applicable to systems which display a level of complexity beyond that which static models or other similar techniques can usefully represent. The simulation framework described in this paper reflects the growth of a fictitious system based on parameters and policies set up for each experiment. The framework allows the definition of basic rules of software development such as the existence of classes and methods and rules governing their relationship. Agents acting as simulated developers then evolve the code base through requirements specification detailed in a requirements policy. Of over-riding importance in the framework is the flexibility afforded to the user - evolving and measuring the system are left to customizable plug-ins which can be tailored to fit their individual aims.

3.1 Framework overview

The framework presents a controlled environment that enforces the evolution of the code base in a pseudo-realistic manner - the direction that this evolution takes rests entirely in the hands of the user. The framework follows a simple feedback network. Its four basic elements are requirements, evolution, measurement and code base. The latter three are connected in a feedback loop as shown in Figure 1. We describe each of these four elements in detail in Sections 3.2-3.5; for the purposes of Figure 1, we provide a précis of each:

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. Requirements are viewed as a set of tasks having a specific type and Agents are responsible for implementing those requirements.
2. Evolution: The Evolution Policy evolves 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. The evolution policy defines a set of rules dictating how to structure code as it is added. The evolution policy can also take account of information on the current state of the code base fed back to it from the code metrics.
3. Measurement: Metrics provide a means for the Agent to evaluate the code base prior to changing it and provides the closing section of the feedback loop in which agents can respond differently depending on their observation of the code base. The Agent will firstly note the cost incurred in creating code (the 'implementation cost'). The Agent will also estimate the cost of comprehending the relevant code before making the change (the 'comprehension cost'; n.b. in the current implementation of the tool, this is described as the 'metric cost', but was changed in this paper to clarify its purpose). These two measures reflect the tasks that a real developer would have to undertake as part of the real-world maintenance process.
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 the method level. The code base comprises: Classes, Functions, Events, Properties and References (each of these is discussed in Section 3.3).

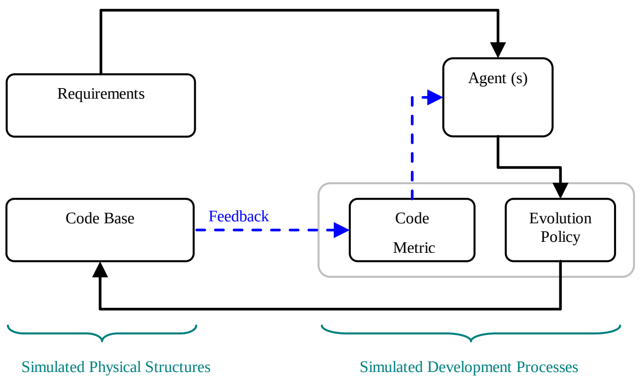


Fig. 1. An overview of the basic elements of the simulation framework.

A run of the simulation starts with the automatic generation of a set of requirements. These are then passed to an Agent to implement through the evolution policy specified by the user for the particular experiment. 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 an important aspect of the framework, since it allows the state of the code base to alter the evolutionary decisions made by Agents. Feedback loops, formed from simple concepts, are responsible for many of the processes observed in complex systems [For69]. As such, the simulation can create responses that are likely to differ significantly from those formed by static analysis alone. The framework only considers code at the method level and above. While it would have been feasible to have considered code at the statement level, simulation at the statement level requires a far more extensive level of computational analysis, introducing the problem of generating and tracking each statement (and the overhead that this carries). Method level analysis on the other hand provides a more manageable level of abstraction and a greater control over the simulation than a statement level analysis would afford. We note that the base version of the system is generated according to either the policies specified by the user *or* from the default policies. In other words, we are not transforming or parsing existing code. Evolution is based on the base version only and this is represented by the set of requirements and code generated according to that policy (user-defined or default).

3.2 Requirements

The simple framework views requirements as a set of tasks each having a specific type. Each task has a set of sub-tasks with a type of either: Entity, Operation or Data Entity:

1. Entities correspond to physical or conceptual units of the application (such as a web-based ‘shopping cart’) which are usually stateful but also contain functionality.
2. Operations correspond to processes that the application performs (such as the ‘checkout’ action on the shopping site).
3. Data Entities are entities that represent *only* data in the application (such as the ‘order’ data generated by the customer)

Each requirement also has a Change Operator: The Change Operator describes how the requirement will operate on the code base. For example, it could be a new piece of functionality or alternatively a bug fix. The change operators modeled in the simulation framework are 'New', 'Augment' and 'Change' and a description of each of these operators is given in Table 1.

Table 1. The change operators and their descriptions

Change Operator	**Operates On**	**Description**
New	Previous requirement or new system event	New functionality that is constructed either from an existing requirement that is to be extended or a new system event.
Change	Previous requirement	A change to the behavior of existing code (e.g., bug fix or clarification).
Augmentation	Previous requirement	An augmentation of existing behavior so that it performs a conditionally disparate function. The degree of difference is recorded as part of the requirement.

The Augmentation change operator is important because it represents one of the primary processes by which software degrades, i.e., classes are changed to perform a conditionally different function from that of their original design. Augmentation operates on a previous requirement, to change that requirement so that it performs some conditionally disparate function. This is different to just changing or extending code as augmentation implies that it will perform its original task whilst behaving differently under certain conditions. This results in the content of the new requirement being intertwined with the original content; this in turn acts to degrade the conceptual cohesion of the code block. Both Augmentation and Change are fundamental to understanding evolution as they represent the most basic forces that need to be harnessed if software evolution is to be controlled. An example software implementation that handles augmentation is proposed by Gamma et al. [Gam95] in their implementation of the Strategy pattern (amongst others). Gamma et al., use basic OO principles to encapsulate the section of a code module that is changing into an underlying strategy using polymorphism. Such a measure simplifies the primary code block as the augmented behavior is extracted to form a different strategy module for each behavior rather than being a complex set of conditionals in one module. This demonstrates a single real-world measure for reducing the detrimental effects of augmentation. The framework described in this paper is specifically designed to accommodate the simulation of such measures through its separate treatment of Augmentation and Change.

The separation of requirements from evolution is another important attribute of the framework's design. Running with different requirements allows the simulation to mimic different development environments (for example, green field developments versus mature products). Experiments can then either hold the requirements constant or deliberately vary them to explore how they affect the simulation. As tasks and requirements operate on one another (or on new system events) they create hierarchies in their structure as one requirement extends, changes or augments others. Notably, requirements can extend other requirements or tasks. An example of the structural relationships between requirements and tasks with example change operators is shown in Figure 2.

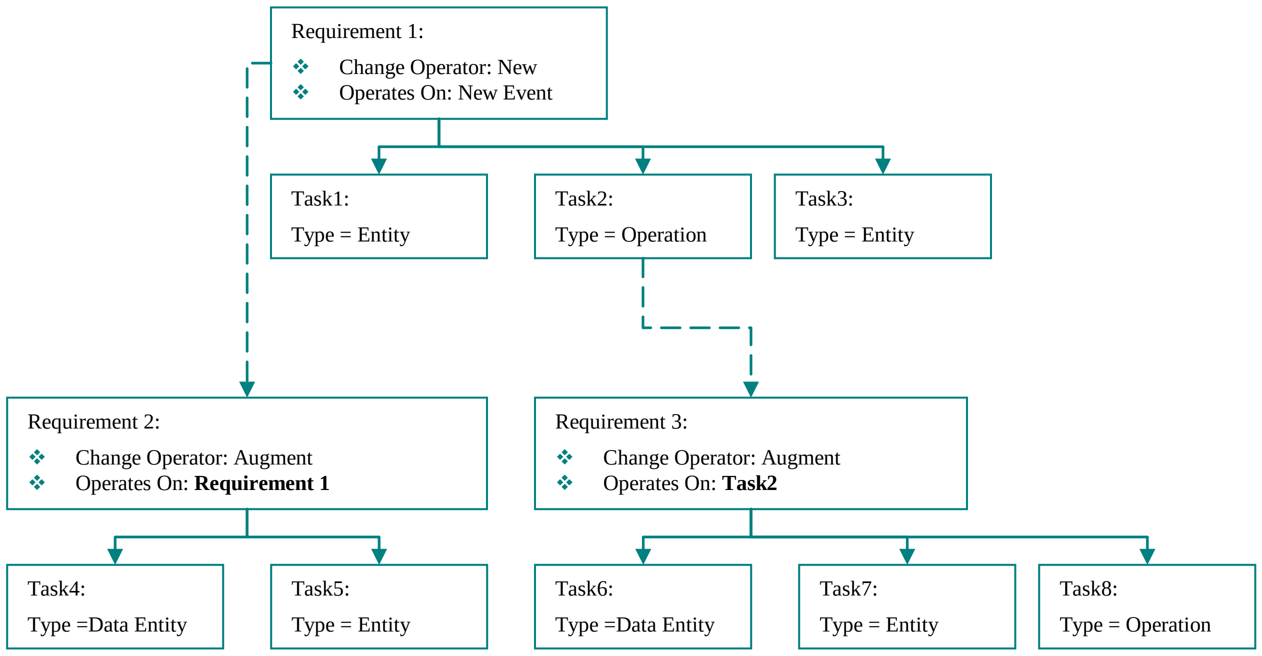


Fig. 2. The relationship between requirements, change operators and tasks.

3.3 The Code Base

The code base acts as a repository for different code constructs created and linked together by Agents. 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 those suggested by Kelsen [Kels04]:

1. Class: Classes represent a repository of functions and parameters and mirror Object-Oriented (OO) Classes.
2. Function: Functions represent the elementary units through which the code base is built and linked mirroring OO methods.
3. Event: Events denote an interaction with an event outside the system. This mirrors the Action Listener feature of OO where functions can react to outside events provided by a GUI interface.
4. Property: Properties represent the internal storage of state through local or global variables of a specified type; they mirror the data (attributes) used by (OO) classes. Properties are introduced to model the stateful linkages between methods and classes thus representing the encapsulation of OO state.
5. Reference: References link code constructs in a directional manner and mirrors OO coupling [Bria99a, Bria99b].

3.3.1 Relationships between code constructs. Four of these code base elements are inter-related, reflecting a typical OO system. Consider the example shown in Figure 3. Class C1 contains one Function F1. Property P1, available to Class C1 refers to a function F2 in Class C2; Property P1 has type C2.

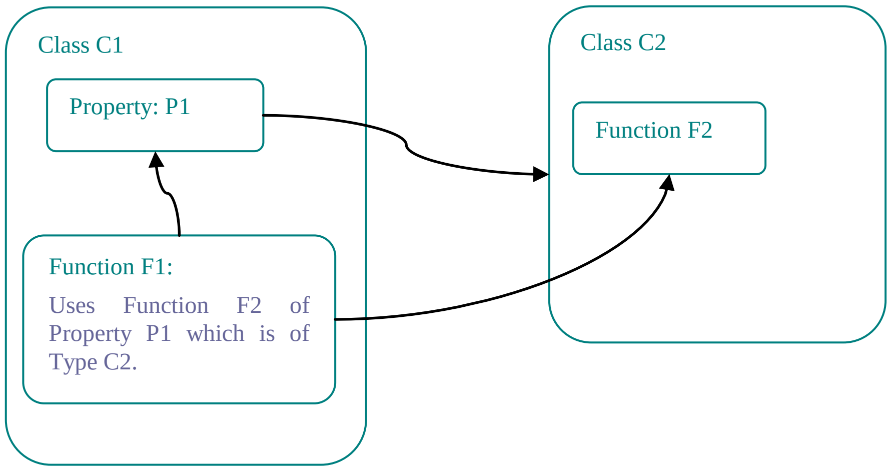


Fig. 3. Function and Reference Properties.

A specific coupling type is associated with each Reference as it is created. The specific variety of coupling type used is determined by the evolution policy (a basic coupling type is used by default).

3.4 An Agent

The Agent is a system concept that embodies the role of a developer in a real-world software project. Agents are stateful with the ability to 'learn' the system as they modify and add to it. The Agent's memory also leaks over time, reflecting the tendency of developers in a real-world development to forget code detail. The Agent's primary function is to facilitate the conversion of requirements into code using the stated Evolution Policy. The Evolution Policy is a plug-in that defines the fundamental operations turning requirements into code for a specific experiment. The Agent is responsible for facilitating this by locating the class to change. Agents are based on a model described by Gilbert [Gilb00] where each agent has a memory of the code constructs that they were responsible for implementing; this memory can be used by the Evolution Policy to improve the depth of the simulation, particularly when considering multiple agents acting on the code base. When multiple Agents are configured, each new generated requirement is implemented by an Agent selected randomly from the pool of currently active agents. The fading memory of each Agent is available to the user for reference as part of the Evolution Policy parameters (see Section 3.5). The action of the Agent is separated into sections based on the change operator of the requirement as the agent must respond differently to different change operators:

1. *New* functionality is added either to a system event of a task or requirement that is to be extended.
2. *Augmentation* can be applied to another Requirement or Task with the degree of augmentation being specified in the new requirement. The degree of augmentation controls how much of the original requirement will be altered when the new one is applied. This is performed by conditionally changing each code construct that is selected.
3. *Change* causes the new change task to be passed straight to the Evolution Policy for implementation.

3.5 The Evolution Policy

The Evolution Policy bears the most responsibility for evolving the code base and is thus a focal point for defining experiments. The user must implement three functions in the Evolution Policy in response to the major categories of requirement type: New, Change and Augment. The system implements a default Evolution Policy. This is of a basic form that only acts to evolve the code base in a random fashion with no overriding structure and not specifically tailored by the user. For each new requirement, whether New, Augment or Change, the actions taken as a result are listed in Table 2.

Table 2. The default Evolution Policy

Function	**Implementation**
Process New	If the Task Type is an Entity then create a new class. If the Task Type is an Operation then add, on average, three functions to the existing class. Each new function includes a property that is linked to it and two other existing functions in the class.
Process Augmentation	If the Task Type is an Entity then: ○ Create a new class for the entity. ○ Create 0-2 functions to the base class. ○ Add 0-2 references between base and Entity classes If the Task Type is an Operation then: ○ Create 0-2 functions to the base class. ○ Create 0-2 extra references from the base class to functions in other classes (selected at random).
Process Change	○ Add the new task to the function being changed. ○ Add a new function with properties linking them back to original function. ○ Add a reference to a random function (performed half the time)

Figure 4 expands the bounded box forming part of the simulated development processes in Figure 1, together with the cost components; it also illustrates how the Evolution Policy evolves the code base through feedback provided by the code metrics.

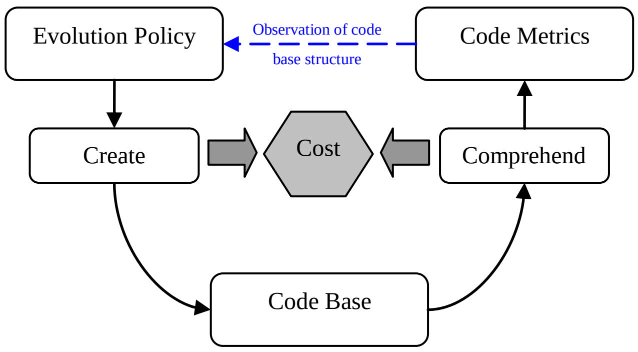


Fig. 4. The cost of a particular run of the simulation.

The user must implement four functions in the Evolution Policy to cause the code base to evolve. Each one corresponds to the conversion of a different task with a specific Change Type into code. The signatures on the interface, the Java plug-in code for which is in the supplement to this paper, are:

```
public Cost processNewTask(Class startingClass, Function startingFunction,Task task);
public Cost processAugmentation(Function startingFunction, Task task);
public Cost processChange(Task newTask, Task taskToExtend);
public CouplingType getCouplingType(CodeConstruct caller, CodeConstruct provider);
```

We note that in each case, the programmatic configuration of these plug-ins allow them to be extended to support more complex responses.

3.6 Complexity Injection and Evolution

Complexity Injection involves adding a random distribution of extra features 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. Complexity injection adds extra structural attributes (e.g., references, properties) whenever a new code construct is created. The Complexity Injector and the Evolution Policy have similar, but fundamentally different roles. The Complexity Injector, also a plug-in, is responsible for the monotonous detail added to all code constructs when they are created on a random basis (i.e., classes need functions and references). The Evolution Policy is responsible for shaping how the structures between classes and functions evolve. For example, returning to the example in the Introduction, when augmenting a class, the Evolution Policy enforces the maximum class size limit – when that is exceeded a new delegate class is created. Complexity is then injected into the new class at that point. The Evolution Policy retains responsibility for the more important and focused features such as tying the new class back to the original as a delegate. In the next section, we elaborate on the default implementations provided by the framework.

4. DEFAULT PLUG-IN IMPLEMENTATIONS

A default set of plug-ins are supplied with the simulation. They define a basic set of policies through which the code base can be evolved and are used in the experiments presented in Section 6. Each default plug-in is a basic implementation of the required functionality and does not necessarily represent an accurate depiction of the evolution of a real code base. Instead, it provides a starting point from which the basic behaviors of the framework can be validated. Each plug-in is represented as a policy which is a design mutated from the common design pattern structure known as the Strategy Pattern [Gam95]. Each implements an interface that defines the contractual obligations that the Policy must perform. The experimenter is then free to define how these obligations are fulfilled. The plug-ins are:

- `com.devsim.plugins.CodeMetrics`
- `com.devsim.plugins.EvolutionPolicy`
- `com.devsim.plugins.RequirementsPolicy`
- `com.devsim.plugins.ComplexityInjector`
- `com.devsim.plugins.EnvironmentVariable`

In the next section, we describe the two other aspects of the default implementation, namely the default Requirements Policy and the default Complexity Injection Policy.

4.1 The default requirements policy

The default Requirements Policy defines the various ‘ingredients’ used to create requirements. These generally represent ratios between the various types that are available. Table 2 shows the different settings making up the default Requirements Policy.

Table 3. The default requirements policy

Setting	**Description**
Calculate Task Type	25% of the time use Data Entity 10% of the time use Entity 65% of the time use Operation
Conceptual Type Reuse Percentage	3%
Calculate Task Revisit Type	25% of the time use Data Entity 10% of the time use Entity 65% of the time use Operation

Mean Number of Tasks per Requirement	Change Operator = New => 5 Change Operator = Augment => 3 Change Operator = Change => 2
Mean Task Size	2
Calculate Change Operator	15% of the time use Augment 40% of the time use New 45% of the time use Change
Augmentation %	A % selected at random

Each of the elements of Table 3 can be changed by the user to reflect different emphases. For example, the user may decide to modify the Change Operator percentages to reflect the fact that the experiment is for a system with very few additions to its functionality, but with a relatively large number of augments.

4.2 The default complexity injection policy

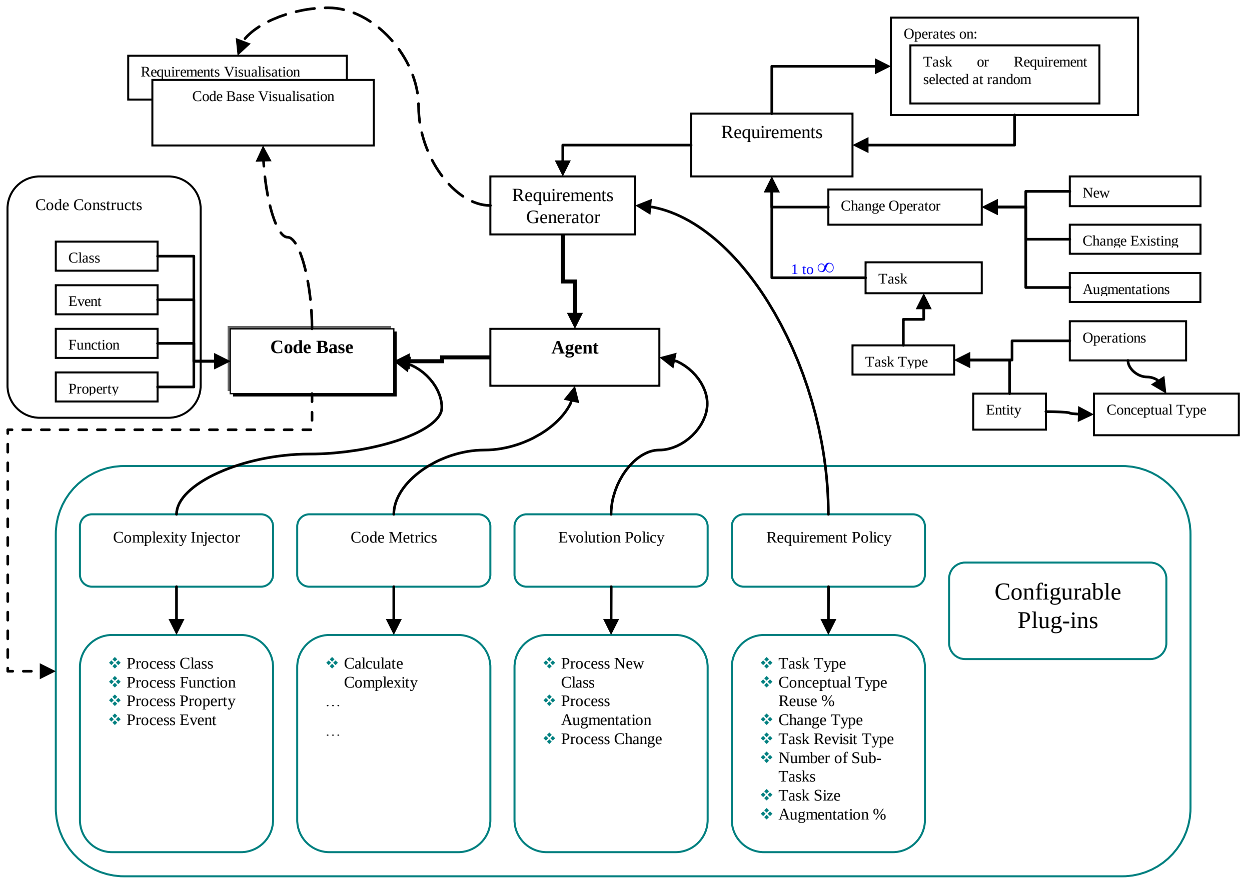
The definition of the default implementation for complexity injection is shown in Table 4. Default behavior for measuring the complexity of a function is defined by the number of requirement tasks that contribute to the function, the number of functions that refer to it (i.e., the amount that it is reused) and the number of other functions that it refers to (i.e., the number of outward references). Comprehension of a function is assumed to require knowledge of all functions in that class unless the Agent has an existing recollection of it. If the agent created the class, then they are assumed to have full knowledge of it and the complexity (to them) is zero.

Table 4. The default Complexity Injection Policy

Function	**Implementation**
Function Creation	Create references to, on average, 2 other functions selected at random.
Class Creation	• Create an average of 3 functions inside the new class (note that the creation of a Function will fire the complexity injection for Function creation). • Create an average of 3 properties inside the new class. • For each function link to 1 or 2 of the properties created.
Property Creation	No Implementation.

The Complexity Injector and Evolution Policies have similar, but fundamentally disparate roles. The Complexity Injector is used only when a new code construct is created and is triggered automatically. Its role is to add complexity to new constructs so that the code base evolves with sufficient detail to make it realistic. By doing so it takes the responsibility for this more mundane task away from the Evolution Policy. (For the most part, the user need only be concerned with the five plug-in classes.) The decisions on the characteristics of the default plug-ins were not made on a random basis. They reflect, in the views of the authors, and on the basis of their industrial development experience, the average type of ‘unit change’ that a system will undergo as it evolves. More specifically, it reflects the typical scale of requirements requests and subsequent complexities of code modification that a developer will typically have to overcome in industry. While the values in Tables 2, 3 and 4 are thus subjective, they do represent the behavior of what we consider to be an ‘average’ yet realistic system. We cannot guarantee that every system (or indeed any system) conforms to this template of evolution and that these values are definitive. However, there has to be some basis on which the user can calibrate (and later extend) the framework and we see the default values as a reasonably sound starting point for this. It is also entirely feasible that a user have in-house measurements on change frequency and complexity which would further add to the value of running experiments using the framework. The overall structure of the simulation model showing the inter-relationships between all the relevant components is illustrated in Figure 5. In the following section, we describe how the simulation application is used.

Fig. 5. Overall View of the Simulation Framework.

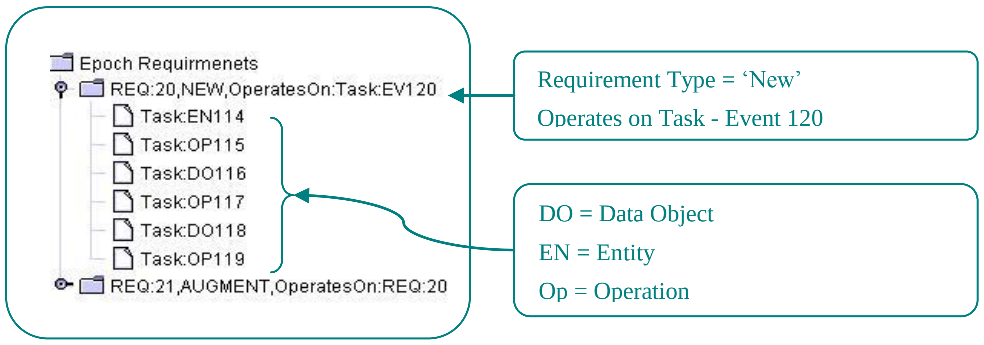


5. USING THE APPLICATION

One of the original aims of the framework was to provide user-friendly and informative feedback to the user of the state of an experiment. The application thus provides the user with a means of viewing the code base (at both a high and low level) and the current set of requirements; these views are now described.

5.1 Code base views

The Application GUI has three views, one for requirements and two for the code base. The code base views include a graphical representation of the class hierarchy and can be explored by the user to follow references made between classes. Figure 6 shows the structure of the requirements view. It shows the structure of a new requirement of type 'New' and the associated tasks generated, of which one is an entity, three are operations and two data objects. The user can click on Requirements to view the included Tasks.



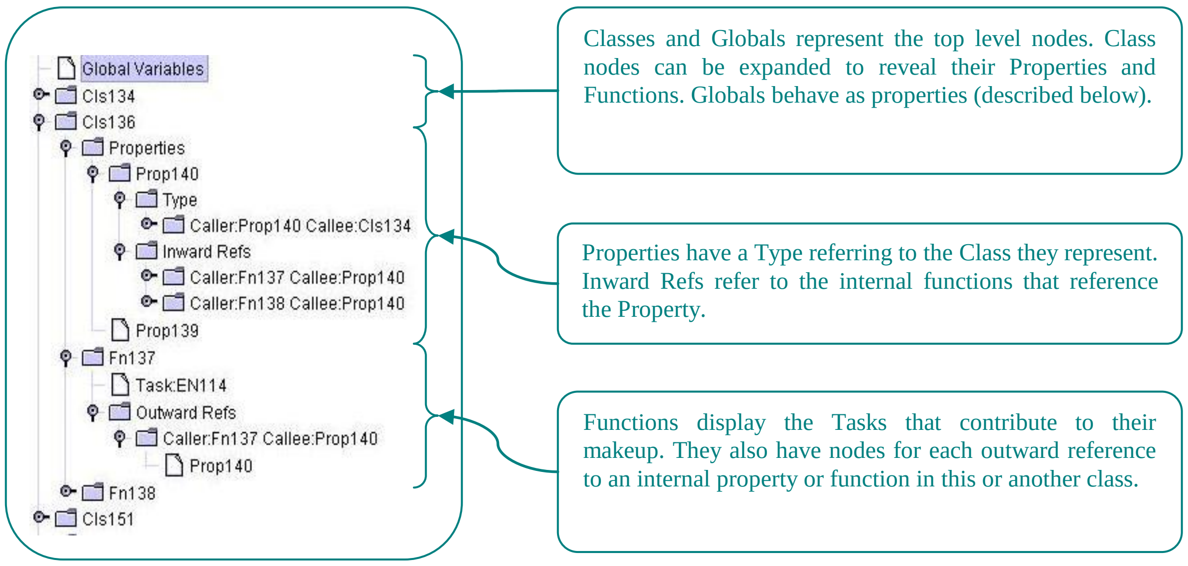
```
Epoch Requirmenets
  REQ:20,NEW,OperatesOn:Task:EV120
    Task:EN114
    Task:OP115
    Task:DO116
    Task:OP117
    Task:DO118
    Task:OP119
  REQ:21,AUGMENT,OperatesOn:REQ:20
```

Requirement Type = ‘New’
Operates on Task - Event 120

DO = Data Object
EN = Entity
Op = Operation

Fig. 6. Requirements view of the simulation.

The GUI supports a view of the code base that allows class hierarchies to be analyzed (Figure 7). The top level of the tree represents *all classes* and global properties in the system. Drilling to the next level displays functions and properties inside that class. Drilling further into a function reveals the tasks that contributed to it as well as all outward references that are made. References can also be drilled iteratively to view the whole call stack.



```
Global Variables
Cls134
Cls136
  Properties
    Prop140
      Type
        Caller:Prop140 Callee:Cls134
      Inward Refs
        Caller:Fn137 Callee:Prop140
        Caller:Fn138 Callee:Prop140
    Prop139
  Fn137
    Task:EN114
    Outward Refs
      Caller:Fn137 Callee:Prop140
        Prop140
  Fn138
Cls151
```

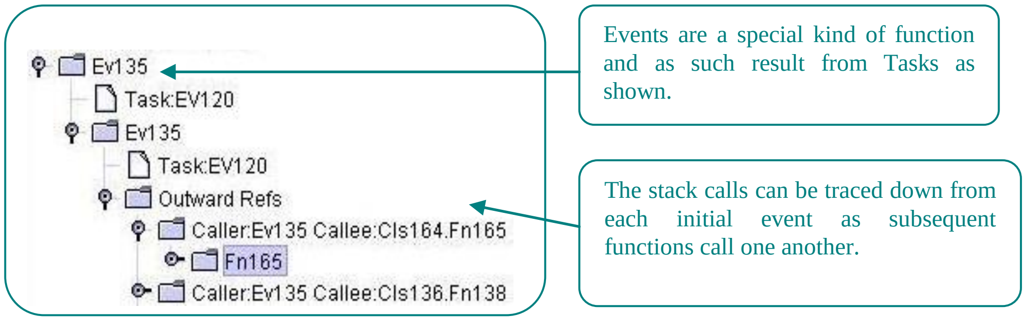
Classes and Globals represent the top level nodes. Class nodes can be expanded to reveal their Properties and Functions. Globals behave as properties (described below).

Properties have a Type referring to the Class they represent. Inward Refs refer to the internal functions that reference the Property.

Functions display the Tasks that contribute to their makeup. They also have nodes for each outward reference to an internal property or function in this or another class.

Fig. 7. High-level code view of the simulation.

The GUI supports a second view of the code base that allows system events to be traced through the resulting code that they execute (Figure 8). The top level of the tree represents all system events. Drilling to the next level displays classes and functions that are executed. References from these functions can then be drilled further as in the Class Drill View.


Ev135
Task:EV120
Ev135
Task:EV120
Outward Refs
Caller:Ev135 Callee:Cls164.Fn165
Fn165
Caller:Ev135 Callee:Cls136.Fn138

Events are a special kind of function and as such result from Tasks as shown.

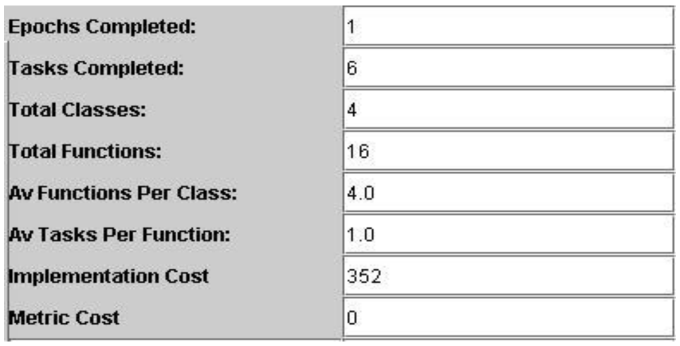
The stack calls can be traced down from each initial event as subsequent functions call one another.

Fig. 8. Low-level code view of the simulation.

5.2 Display panels

A further aim of the framework was to provide timely feedback to the user of the state of an experiment as it evolves. The application thus provides the user with a series of informative panels which allow the experiment to be tracked as it progresses; the readout and control panels are now described.

5.2.1 Readout panel. The simulation panel displays features of the evolution as the code base evolves (Figure 9). All readouts refer to the code base as a whole. ‘Epochs Completed’ refers to the number of requirements that have been added to the code base. ‘Tasks Completed’ refers to the number of tasks that have been added to the code base. As well as the Total Class and Function count, the panel also shows the Average number of functions that exist inside each class on average over the whole simulation run and the average number of tasks that exist inside each function on average over the whole simulation run. As previously described, the ‘Implementation Cost’ is associated with the creation of new code and the ‘Comprehension Cost’ associated with comprehension of the code base prior to a change.



Epochs Completed:	1
Tasks Completed:	6
Total Classes:	4
Total Functions:	16
Av Functions Per Class:	4.0
Av Tasks Per Function:	1.0
Implementation Cost	352
Metric Cost	0

Fig. 9. The readout panel.

5.2.2 The Control Panel. The Control Panel provides control for the simulation itself and is illustrated in Figure 10. The ‘Create Requirement’ function adds a new requirement to the requirements list so that it can be viewed (but will not add it to the code base). The ‘Expand All’ and ‘Collapse All’ functions expand and collapse all nodes visible in the tree pane. The ‘Clear Unimplemented Requirements’ function clears all requirements that have not been implemented in the code base and the ‘Clear Code Base’ function clears the code base but not the requirements; finally, the ‘Clear All’ function clears both the code base and requirements.

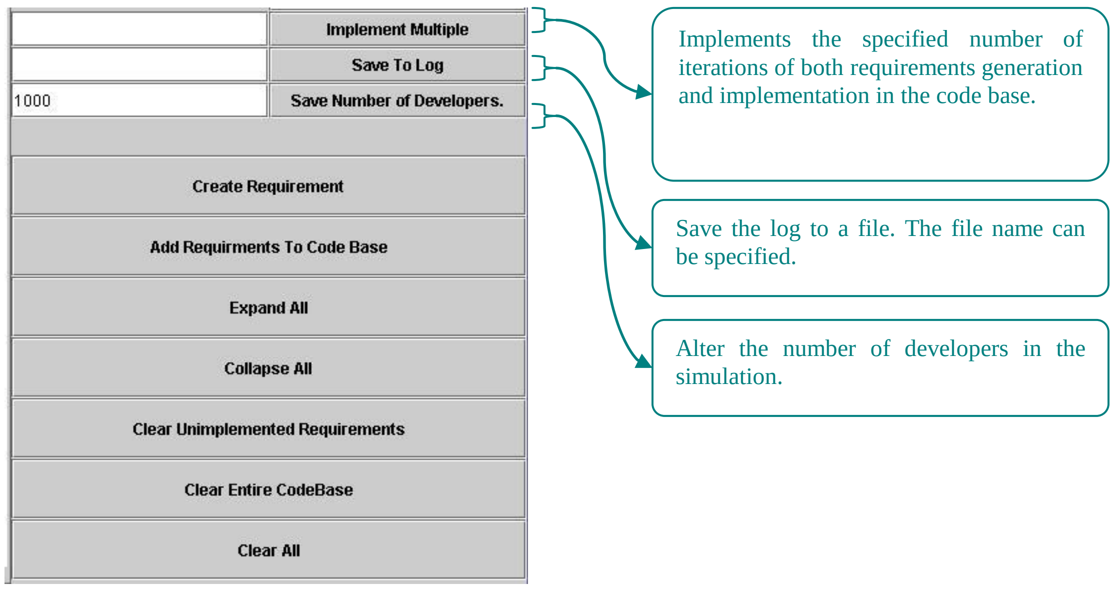

Implement Multiple
Save To Log
1000
Save Number of Developers.
Create Requirement
Add Requirments To Code Base
Expand All
Collapse All
Clear Unimplemented Requirements
Clear Entire CodeBase
Clear All
Implements the specified number of iterations of both requirements generation and implementation in the code base.
Save the log to a file. The file name can be specified.
Alter the number of developers in the simulation.

Fig. 10. The Control Panel.

The output of the simulation is sent to a file which is then saved in a tab-delimited format containing all empirical data provided in the GUI itself.

6. EXPERIMENTAL DESIGN AND ANALYSIS

Experiments can be designed to probe evolution of software structure though one of two primary methods. Firstly, through investigation of interaction between a specific evolutionary factor and other factors in the simulation (here, a ‘factor’ describes something that shapes evolution). Secondly, through investigation of the incremental effects which evolutionary factors have on themselves through repeated application. Alternatively, the reaction of the factor to different running conditions can be investigated such as environments with different proportions of new development, change and augmentation. Experimental designs must consider:

- Modeling of the Evolutionary Factor: How the new factor is to be modeled in the simulation. This means defining how the code base will be affected by the action of the new factor based on the input requirements and the existing code base.

- Measurement of the Evolutionary Factor: How the resulting evolution will be measured and how this measurement will feed back into future evolution cycles.

Both of these factors are required so that a feedback loop is set up between the code base and the Evolution Policy (Figure 1). The experimental method is defined according to the following steps:

1. Identify the problem to be investigated.
2. Develop a hypothesis that describes the cause of the problem.
3. Create an evolution policy plug-in that changes the code base according to the hypothesis.
4. Amend the code metrics plug-in to ensure that it is sensitive to the evolutionary changes expected.
5. Test the Evolution Policy and metrics in isolation to ensure that they reproduce real-world behavior.
6. Add the implemented policy to the full simulation model to allow investigation of the interactions between it and other simulated factors (note that all results are comparative rather than absolute measurements).
7. Devise and test alternative policies via previous steps.

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.

6.1 The Statistical Variance of Results

Due to the stochastic nature of many of the parameters used in the simulation, an underlying statistical variance makes each evolution of a system slightly different. It is therefore important to provide a measure of the implicit variation in results. Table 5 illustrates the standard deviations and means for the linearly evolving variables taken over three independent evolution default profiles, each consisting of three hundred epochs.

Table 5. Summary of statistical results for three independent, default evolution profiles

	Std. Deviation	**Mean**
Class	8.12	261
Function	86	1552
Tasks per Function	0.0187	1.1680
Functions per Class	0.3041	5.7414

6.1.1 Graphical variance. Figures 11, 12 and 13 show graphs for the function count over the course of the three runs, the tasks per function (Fn.) over the same three runs and a comparison between the Comprehension and Implementation Costs over the same three runs, respectively. In each case, there is strong correspondence between the lines plotted, suggesting (as per Table 5) a low variance in the values produced by the model. In terms of calibration of the model, the goal is to ensure that the simulation results agree with those expected, both through an intuitive understanding of the software engineering process and those provided by empirical observations.

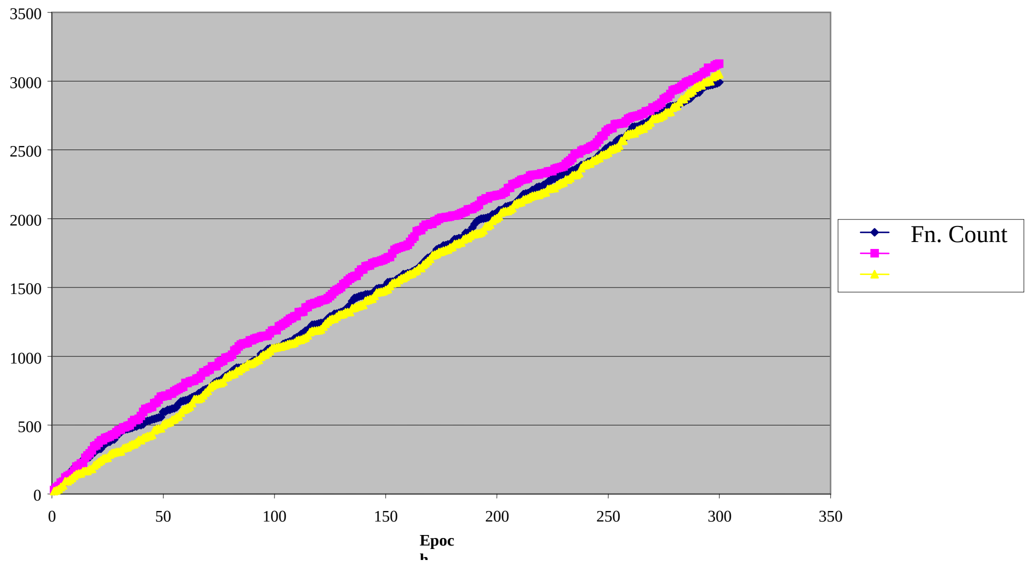


Fig. 11. Function Count for three separate but identically configured runs of the system.

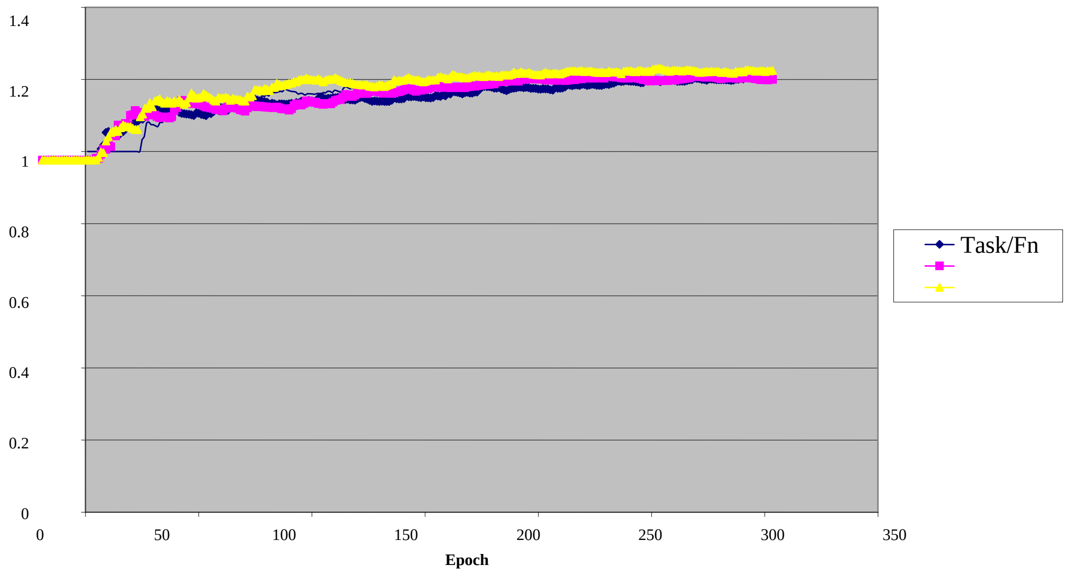


Figure 12. Tasks per Function for three separate but identically configured runs of the system

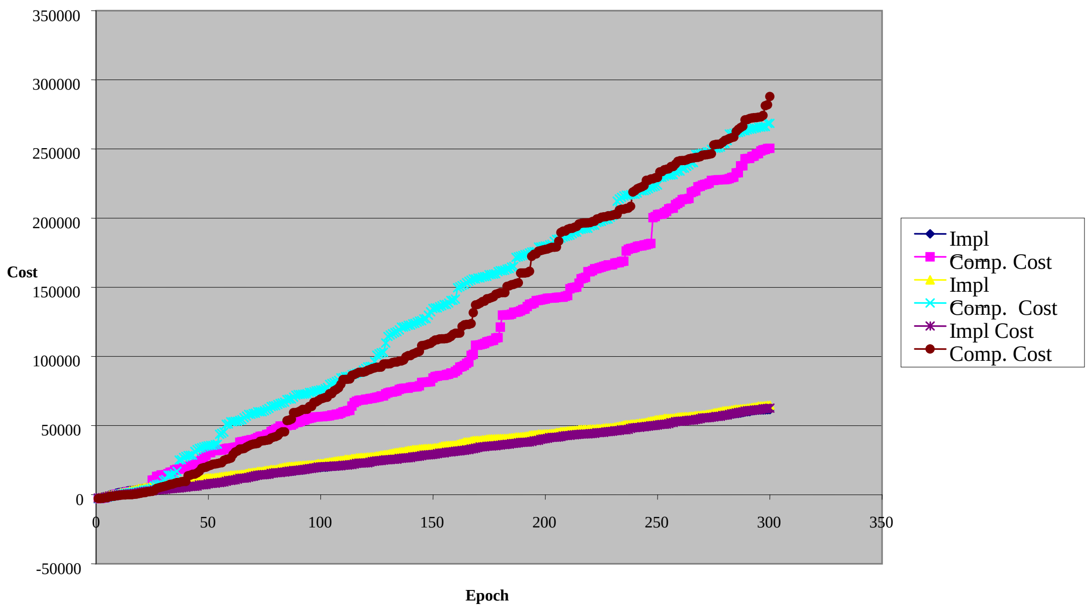


Fig. 13. Comparison between Metric and Implementation Costs in three separate runs of the simulation.

6.2 Experiment Two: Increase in Source Code Size under default settings

The aim of the second experiment is to ensure that running the simulation under default conditions (i.e., where the various experimental parameters are set to 'typical' values) causes the code base to expand in size in a manner that approximates what is considered to be a realistic software project. The experiment was run for 300 Epochs (requirement implementations) with the first 20 Epochs being exclusively 'New' requirements. The change operator ratios for this experiment were set to: Augment 15%, Change 45% and New 40%. Figure 14 shows the increases in tasks, classes and functions over the course of the simulation.

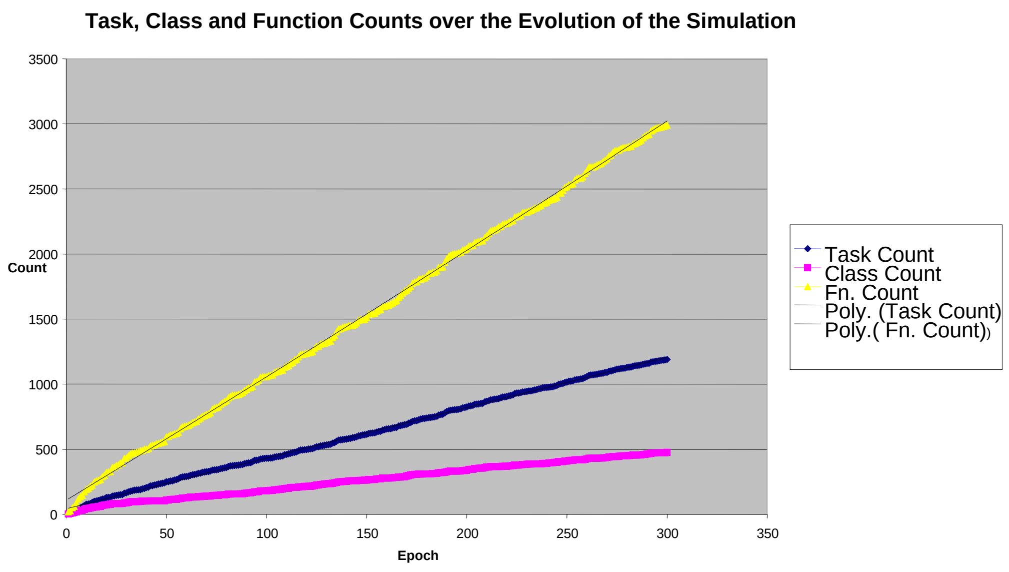


Figure 14. Increases in Task, Class and Function

The increase in task, class and function count are all linear over the life of the evolution which is to be expected since there is no modeled non-linearity between current size and size increase. All three measurements show slight variation due to the stochastic nature of requirements generation and complexity injection. There is also an observably increased gradient for the first twenty epochs in the Class and Function plots. This arises due to the first twenty requirements having a requirement type of 'New'. New requirements increase the probability of producing new code constructs and hence explain the increase in gradient. This behavior is corroborated by an empirical study made by Capiluppi et al., [Cap04] of the ARLA open source system. In the same study Capiluppi et al., found that the number of source files grew linearly as the project evolved. This is comparable to the simulated growth of class files being linear with a positive gradient. The same research also provided a study of the distribution of average lines of code per file over various releases. Their results showed the average number of lines per file increasing slightly as the system evolved. This increase was approximately linear with a small positive gradient. A comparable result was generated with our simulation framework and is displayed in the Figure 15. This chart shows the evolution of task density over the simulation where task density represents the average number of tasks per class and shows approximately linear growth with a slight positive gradient. The comparison assumes that, on average, the number of tasks is proportional to the number of lines of code in a real application.

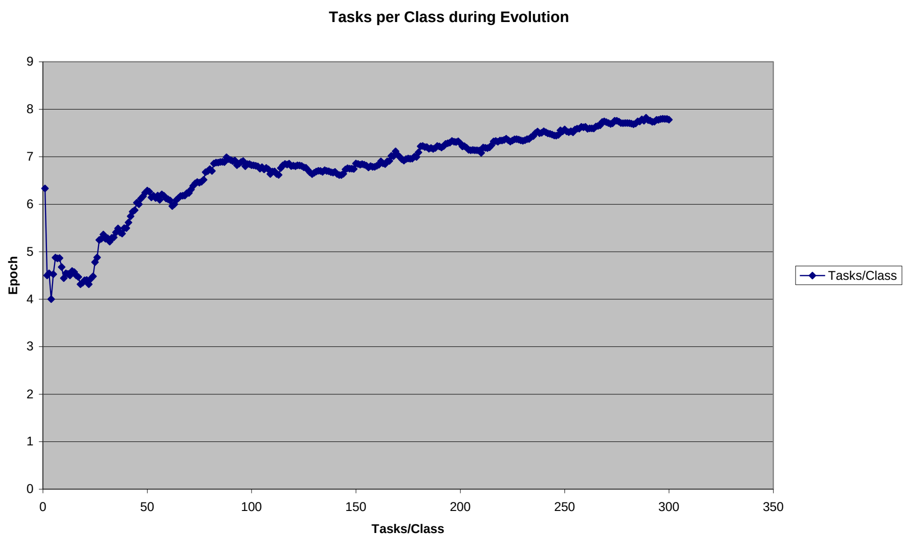


Fig. 15. Growth in tasks per class.

6.3 Experiment Three: 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. Validation of this aspect of the model is provided by analyzing the effect that requirement types have on the evolution of the code base. The default code metric is dependent on the density of requirements tasks in the code (the code base records which requirements contribute to each code construct). This system feature was validated by configuring an experiment with different distributions of requirement types. The experiment was run for 300 Epochs (requirement implementations) with the first 20 Epochs being exclusively 'New' requirements types as in previous experiments. The percentage distribution of each requirement type in each experiment is shown in Table 6 (configured in the Requirements Policy):

Change Type	Control (Run 1)	Run 2: High New	Run 3: Low New
Augment	15%	5%	25%
Change	45%	15%	70%
New	40%	80%	5%

Table 6. Percentage distribution for three different simulation runs

The experiment thus measures the evolution cost for each of the three different distributions of Requirement Types, averaged across three independent runs.

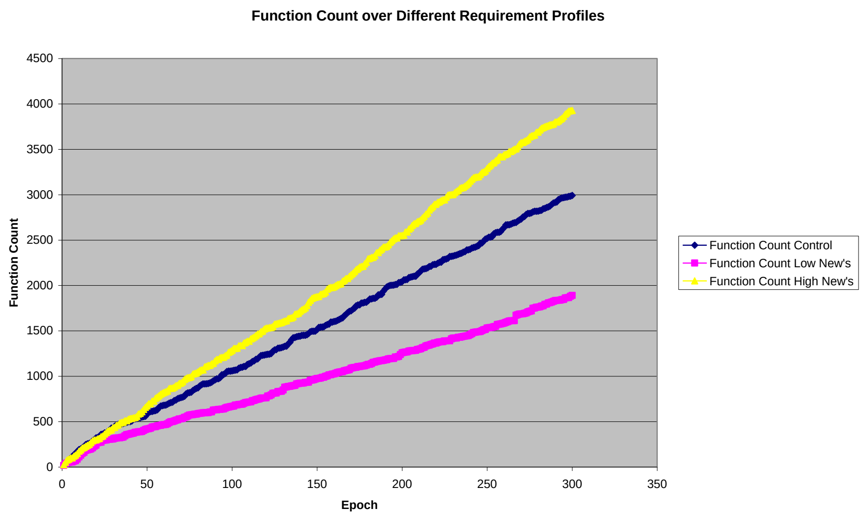


Fig. 16. Function change under different requirement profiles.

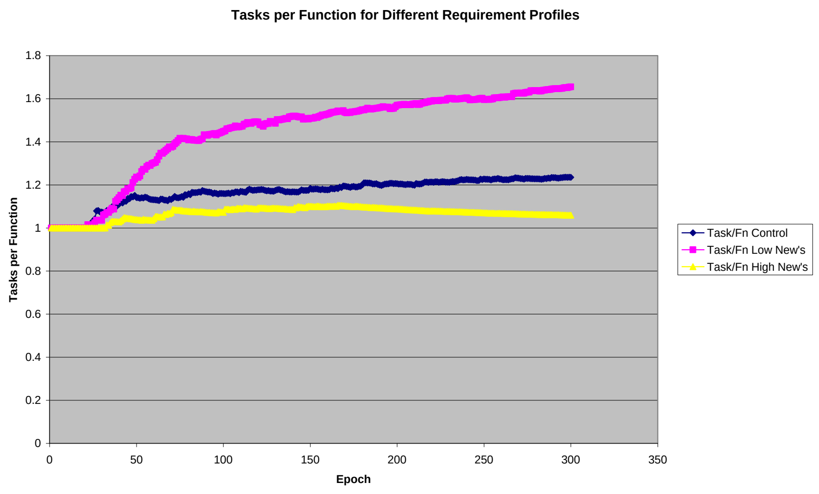


Fig. 17. Task changes under different requirement profiles.

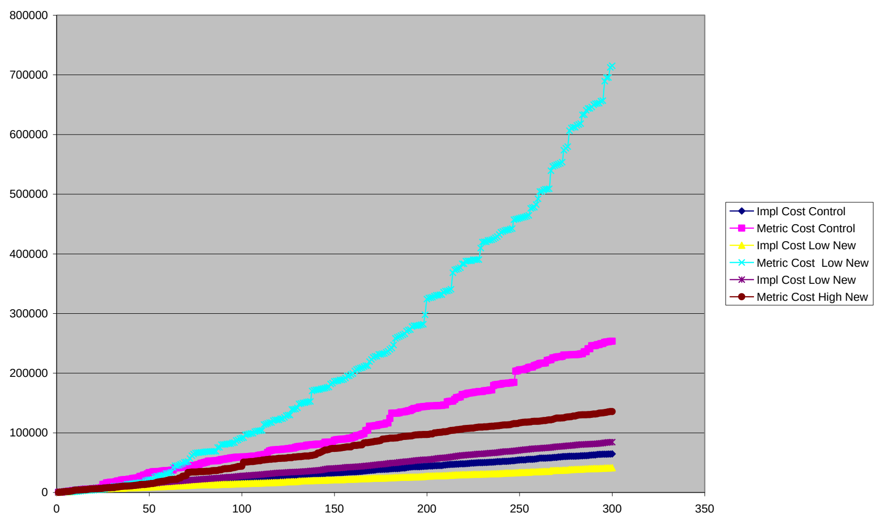


Fig. 18. Costs associated with different requirement profiles.

Requirements implying that old code must be changed (rather than new code being written) are expected to increase the requirement density and hence the metric value. This arises as changes to old code create, on average, fewer new code constructs per requirement than new developments. Conversely, large proportions of new development dilute the requirement density and this is reflected in lower metric values. In terms of the simulation, this corresponds to runs with a high proportion of the requirement type “New” having lower comprehension costs than those with a large proportion of requirement types “Change” and “Augmentation”. The results produced in this validation corroborate this hypothesis with an increase in curvature evident in runs with higher proportions of ‘Change’ and ‘Augmentation’ requirements (Figure 14). Run 2, predominantly the ‘New’ requirement type, has a near linear response as well as the lowest increase in comprehension cost, thus validating the expected behavior.

6.4 Experiment Four: Response to Different Numbers of Agents

The simulation provides a facility for specifying the number of agents that contribute to the evolution of code. Each agent “remembers” the code it created and this memory is taken into account by the default code metric. The metric value is dropped if the agent was responsible for creating the code under measurement. The hypothesis is that the metric value should evolve more slowly for low numbers of developers as they will each have been responsible for more code and hence have a broader memory of the code base. This was validated via the experiment displayed in Figure 19. The results show that development with two agents is most efficient and thirty 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 original hypothesis.

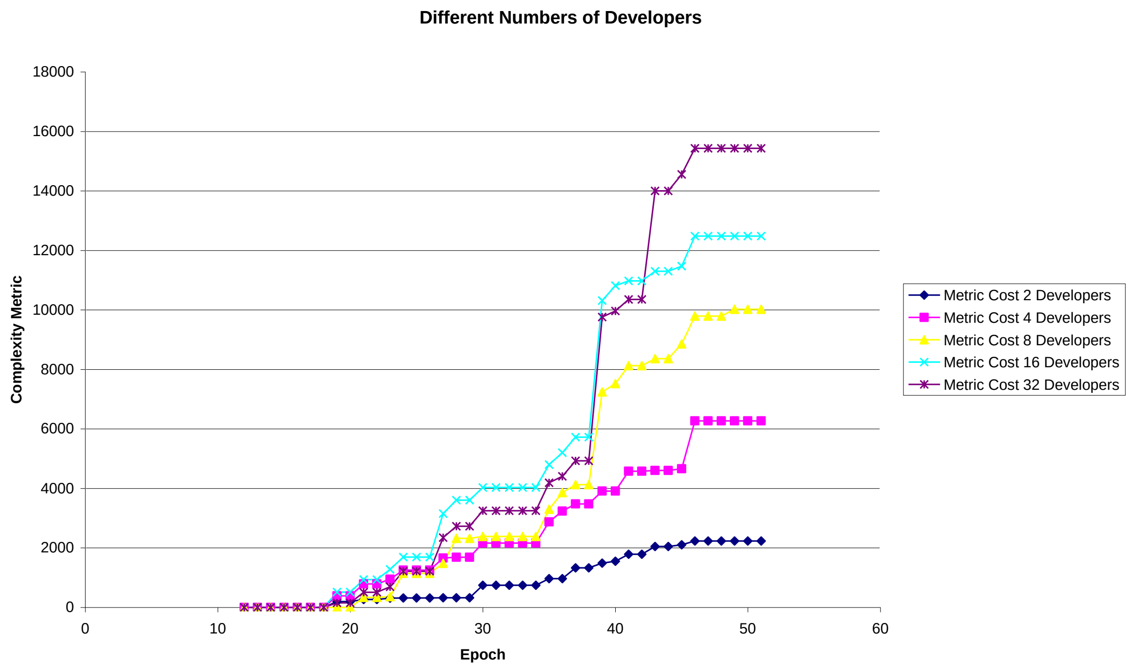


Fig. 19. Evolution with different numbers of agents

The complexity cost increases with the number of developers due to the increased cost associated with having to understand code prior to changing it. We might also expect a corresponding increase in faults in code as more developers are added to a project - more code is being changed by a developer who was not the author of the original code. With more developers, the likelihood of more faults may also arise due to the potential for lack of communication and/or miscommunication [Bro75]. The results presented in this section thus provide a level of confidence that the simulation performs in a manner approximating the features of a real-world application. This conclusion is corroborated by both intuitive expectations and empirical results presented in the next section.

7. EMPIRICAL SUPPORT

One important aspect of any simulation framework is to provide evidence from the real-world to support the results and assumptions of the simulation. In this section, we provide empirical evidence from three recent empirical studies carried out by the authors and collaborators, all of which used OSS as a basis. While none of those three studies used the simulation framework as such, they are used to support the claims we have made about that framework.

7.1 Empirical Support Study One

One major claim of the simulation framework is of linear growth in code size over the duration of the implementation. This was a feature we also observed of the study by Capiluppi et al [Cap04]. To support our claims about linear growth we provide evidence from a study of an OSS [Mub07]; the study used a Java system called ‘Velocity’ – a template engine allowing web designers to access methods defined in Java. For each of the nine versions of Velocity, we collected the number of added classes, added lines of code (LOC), added methods and added attributes (over the previous version). We define LOC as any non-comment, non-blank line. A bespoke tool was written to extract this data. One of the research questions posed in the same study was: *Does the number of new classes over the course of the nine versions increase constantly? This question is based on the notion that a system will grow over time in a constant fashion in response to regular changes in requirements.* We provide evidence from this study to support Experiments Two and Three (Sections 6.2. and 6.3, respectively).

7.1.1 Research Question Re-visited. Table 7 shows the number of new classes in total for each of the nine versions. The number of new classes varied significantly from one version to another. Between versions three and four and six and seven, relatively little change can be seen, while the peak of added classes is reached in the fifth version with 2032 new classes added. Clearly, the addition of classes to this system over the versions investigated is not constant. However, the Velocity system started with 224 classes and if we then add each of the values in column 2 of Table 7 cumulatively, we obtain the graph in Figure 20. While we cannot claim that increases in classes are constant on a version-by-version basis, the general trend for this system is an increasing one. Also of interest is the large number of new classes in the first and second versions of the system, suggesting a wide range of ‘new’ requests (or significant modifications to the existing) just after the system had been installed. Our framework was able to reflect this feature with the relatively higher percentage values of New requirements in early epochs (partly supporting Experiment Two). The large rise in new classes from version 4 to version 5 in Table 7 was unexpected, but not uncommon among the evolution of OSS. As part of a recent empirical study of the addition of classes to a system called JBoss [Nas08], over 4000 classes were added between one version (19 and 20). From version 20 to 21, over 4000 classes were removed from the system. Two versions later, the system had grown again by over 4000 classes. There are a number of explanations that may account for this trend. Firstly, the timing between the releases of versions may differ to the extent that proportionately, the change in classes may be equivalent. A more plausible explanation however, might be that there was a concerted effort to significantly enhance and re-engineer the system. One could ask why, if so many new classes were added, the existing system was not simply scrapped? Of course, one explanation may be that the existing set of classes contained key system functionality in which case they would have been of central importance to the next version of the system. The erratic nature of the evolution of OSS can be placed in context when we also consider that another system studied in [Nas08] saw no additions of any classes between versions 5 and 24. While the default policies of the framework described in this paper might not have anticipated such extremes, user-defined evolution policies might easily account for static or dramatic change in a system.

Version	**No. of new classes**

1st	788
2nd	1116
3rd	17
4th	11
5th	2032
6th	45
7th	297
8th	1274
9th	1386

Table 7: New classes over the course of nine versions

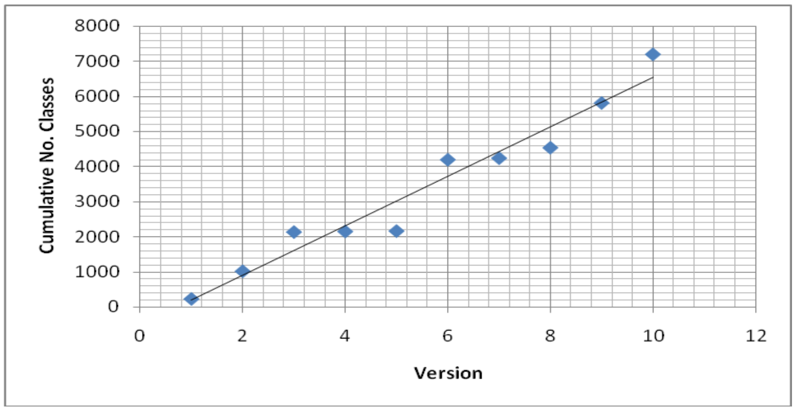


Figure 20. Cumulative No. classes

With each new class added, we could expect at least one method and one attribute to be added as well as significant numbers of LOC. Table 8 (taken from the same study) shows a positive significant (Spearman's) correlation at the 1% level (asterisked) between LOC, Attributes and Methods for the Velocity System. Table 9 shows the same correlations, but for new classes.

Correlation Coefficient	LOC	Attribute	Method
LOC	1.000	.917*	.950*
Attribute	.917*	1.000	.917*

Method	.950*	.917*	1.000

Table 8. Spearman correlation coefficients of the increases in LOC, Attributes and Methods

Correlation Coefficient	New Classes	Attribute	Method
New Classes	1.000	.833*	.950*
Attribute	.833*	1.000	.917*
Method	.950*	.917*	1.000

Table 9. Spearman correlation coefficients of new classes, increases in Attribute and Methods

The evidence from Tables 8 and 9 suggest that there is corresponding rise in the number of attributes and methods as classes are added to a system and that the three class features move in the same direction.

7.2 Empirical Support Study Two

The second study in support of our simulation framework sought to establish patterns of changes in Java classes taken from three libraries; the full results of this study were reported in Counsell et al [Cou03]. The change data for this study was collected manually from on-line source document representations of the *diffs* between successive versions of classes in three sub-libraries of the gnu GCC libjava library spanning a three year period [Gnu08]. A random sample of fifty classes from each of the IO, AWT and Lang. sub-libraries were used as a basis. Version numbers and lines of code added over the lifetime of each class were collected. Table 10 shows summary data for the number of added LOC over the period studied and shows the Min., Max., Mean and Standard Deviation values for each of the set of classes.

Library	**Min.**	**Max.**	**Mean**	**Std. Dev.**
AWT	0	4903	302.35	725.57
IO	-5	1031	139.29	178.53
Lang.	-15	1877	132.79	286.63

Table 10. Summary data for LOC added to classes

Table 11 shows the correlation values when we consider the changes in lines of code against the number of versions - every value is significant at the 1% level. In other words, with every new version of a class in each of the libraries, the number of LOC rises correspondingly.

Library	**Pearson's**	**Kendall's**	**Spearman's**
AWT	0.78**	0.74**	0.87**
IO	0.65**	0.49**	0.64**
Lang.	0.54**	0.66*	0.79**
Combined	0.57**	0.59**	0.74**

Table 11. Correlation of added LOC with number of versions for library and combined

Figure 21 shows the scatter plot for the AWT set of classes when number of added lines is plotted against number of versions.

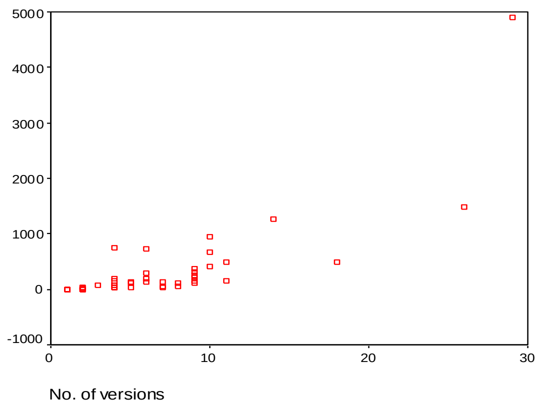


Fig. 21. No. of added lines versus number of versions: AWT

The same trend exhibited in Figure 16 was also found for the two other sets of classes, suggesting a common feature among the three library systems of consistently rising LOC as a class evolves.

7.3 Empirical Support Study Three

One claim made by the simulation framework and the subject of Experiment Four (Section 6.4) was that the when 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). The time taken to complete a change is therefore less. One realistic scenario that we can observe from the same supposition is that the more developers working on a project, the more that code will 'decay' because of that unfamiliarity; when code decays, faults in that software tend to start increasing, so causing the requirement for more maintenance. The net result is a circle of poor maintenance, faults, poor maintenance. Ultimately, the system has to be re-written or scrapped as those costs outweigh the benefits. (Experiment Four suggests, as more developers attempt to maintain code they didn't create themselves, maintenance costs rise.) There is a further aspect to the features of Experiment Four. As a system degrades, a common approach is to allocate more developers to a team. The theme of Experiment Four supports the view that allocation of more resources to a project may actually have the opposite effect to that intended. Our result also lends support to Brook's Law which suggests that adding staff to a late project makes it later [Bro75]. Of course, the case of OSS is different to in-house development of software upon which, at IBM, Brook's Law was originally based. For OSS, developers can join, contribute and leave as they wish. However, that doesn't necessarily invalidate claims we can make about the evolution of OSS.

Table 11 shows the data extracted from ten OSSs of varying application domains using a software tool called FindBugs [Find07]. FindBugs is a tool that will extract the number of 'potential' faults in Java code based on commonly identifiable and recurring patterns. The tool is used by companies and individuals alike and has a current user-base of over 300,000. Table 11 shows the total number of faults extracted by FindBugs, the total number of system classes, the number of developers on each system and the average faults per class. We note that all of this information was available from sourceforge.net for each of the projects in question.

System	**Total No. Faults**	**Classes**	**No. Developers**	**Average Faults per Class**
System 1	20	32	2	0.63
System 2	12	159	2	0.08
System 3	135	649	2	0.21
System 4	11	86	1	0.13
System 5	153	890	8	0.17
System 6	121	784	41	0.15
System 7	297	1253	7	0.23
System 8	131	102	1	1.28
System 9	14	82	1	0.17
System 10	770	914	13	0.84

Table 12: Data for ten Java OSSs

The largest numbers of faults (the top three) were from systems with 13, 7 and 2 developers, respectively. System 6 is the exception to this trend and has the second lowest fault per class rate. Equally, the lowest numbers of total faults were from systems with 1, 2 and 1 developers, respectively. While ten systems is only a relatively small representative sample of systems and we are looking at a single version of each system, there does seem some evidence to support the claim of Experiment Four; the more developers, the greater tendency for faults. Of course, maintenance comes in many shapes and forms. However, maintenance due to faults plays an important part in the evolution of a system and accounts for a significant chunk of maintenance effort. Table 12 shows the cross-correlation values for columns 1-3. It is interesting that while no statistical significance was found for the faults versus developers, the direction of the correlation is positive and high.

	Pearson's	**Kendall's**	**Spearman's**
Faults versus Classes	0.63	0.60**	0.79*
Faults versus Developers	0.22	0.50	0.62
Classes versus Developers	0.46	0.55**	0.75*

Table 13. Cross-correlation values

The significant relationship between faults and classes was not entirely unexpected. However, one threat to this result may be that the majority of faults were attributed to a relatively small number of classes. In other words, the number of classes is not a particularly good guide to fault distribution. The same principle would apply to the relationship between classes and developers, although we could easily justify the hypothesis that the number of classes rises as more developers are added to a system. Developers will always add new functionality rather than change old and there is only limited evidence of refactoring on a large-scale as a system evolves [Cou06]. In the next section, we discuss some of the issues raised from the simulation model and the experiments carried out.

8. DISCUSSION

This project has been an exploration into the simulation of software evolution from the perspective of the code and how it changes. The simulation framework allows experimenters to investigate two major relationships. Firstly, the mapping from a set of requirements to a structural implementation through an evolutionary policy and secondly, the feedback loop from the code base to the evolution policy via the code metrics. A number of key points emerged from development of the simulation framework and the subsequent experimentation.

- First, separating the requirements from the code base allowed the requirements to exist as a separate entity that could be grown independently. This had the important side effect that they could be validated prior to being implemented as code. Requirements also played a pivotal role in the structuring of any code base. Keeping a clear separation between these two made it easier to define their inter-relationships and allowed requirements to be held constant for experiments in which they were not part.

- Second, Forrester, a pioneer of System Dynamics, proposed that feedback loops, formed from simple concepts, create the foundations of a large percentage of the complexity observed in dynamic systems [For69, For71]. A vital attribute of our simulation is the feedback loop between the Evolution Policy and the code base. This provided a facility to model complex, non-linear behaviors generated from the interaction of simple concepts modeled in the system.
- Third, it is difficult to create a simulation that produces absolute results as each quantity must undergo careful and time consuming calibration. It was vital to validate all concepts thoroughly as any invalid assumptions would combine and scale to produce results that may not have been representative of real-life behavior.
- Fourth, the use of the simulation model as described only touches the surface of the potential for other, larger experiments to be undertaken. Future experiments could also be run to service more focused goals. These might include, for example, an investigation of whether evolution is affected by the starting structure of the code base and a study into the effects that iterative development cycles have on a code base vs. more traditional, lengthy cycles. Furthermore, there is plenty of scope for the effect of componentization and the effects that 'Separating Concerns' have on an evolving system; for example, when splitting GUI and business logic such as the MVC Pattern [Fow02], how does the overhead weigh against the benefit induced? To explore how the simulation framework might be used to investigate the separation of GUI and business logic we could consider the following set of six steps:

    1. The aim is declared to be an investigation of the effects of splitting GUI and business logic in an evolving system.
    2. The Requirements process is altered to incorporate the concept of a special "GUI requirement".
    3. A new Evolution Policy is created. This takes GUI requirements and implements them in separate classes to the business logic. As GUI and business tasks are added to the code base the evolution policy will create references between them. Different coupling types are used to link the GUI and business components.
    4. The code metric is altered to take into account the fact that separate concerns should be more comprehensible (this could result from some more fundamental metric such as Miller's magical number seven [Mil56]).
    5. A control experiment is run which mixes these new GUI requirements with the regular experimental parameters and policies.
    6. The final experiment is run to investigate the extra effort required to add such features and the effect it has on the structure when evolving in different environments.

In the case of point 4, care should be exercised in development of the framework to ensure that the separation of concerns does not have a negative impact on comprehensibility. While Miller's magical number seven [Mil56] might apply in the context of psychological experimentation, recent experience in a software engineering context suggests that too much decomposition (through the object-oriented concept of inheritance) can have the opposite effect to that intended [Harr99].

- Fifth, state evolution is a topic that has gained little light in recent research when compared to its counterpart, behavior. State adds complexity to the interaction of components at runtime. This is likely to have a detrimental effect on software as it evolves, but there is little data to corroborate this. Simulation would provide an ideal means for adding experimental data in this field as the runtime modification of state could be simulated in this framework with relative ease. A more far reaching goal would be to use the simulation to adapt the laws that bind the OO paradigm itself. Similar experiments could also investigate the effects that Aspect-Oriented Programming [Kicz97] has on the structure and evolution of code.
- Sixth, future experimental methods are likely to focus not only on the investigation of evolutionary concepts in isolation, but also on multiple concepts combined in the same simulation environment. This will allow the examination of more complex relationships that result from this mutual interaction. The following list enumerates additional features that could be added to the simulation to investigate the effect on structural evolution:

    - Coupling Types: Investigation into how different couplings between modules affect structural evolution.
    - Inheritance: The simulation already supports the concept of Abstraction Types in the Requirements section. This forms a basis for modeling inheritance and other forms of abstraction.
    - Design Principles: Adding new and validated design principles to the evolution policy would make evolution more realistic. Many principles could be introduced from basic ones such as encapsulation and reuse [Par72] to more complex design principles [Mey92], [Boo94], [Gam95].
    - Refactoring: Fowler [Fow99] presents a set of seventy-two refactorings that alter the structure of code. Simulating their effects would make an interesting experiment.

- Seventh, one approach that is often taken in the field of Software Process Simulation is to use multi-faceted models as decision support systems for managers. The process simulation approach has been pioneered by NASA and others, who utilize their wealth of empirical data from previous software projects to calibrate the simulations. Topics investigated include the defect detection efficiency of code inspections [Mun03]. The simulation model we present here could be used as the basis for a decision support tool. Such a tool could predict the effects that different environmental factors or coding practices have on a code base, allowing managers to tune their coding practices accordingly. It could also be used to test and experiment with different types of software evolution and maintenance models [Chap01].

- Finally, while use of simulation framework presented shows some promise modeling the evolution of systems, we stress that the utility of such a framework is only as ‘good’ as the set of metrics and parameters used to model that evolution. The conclusions we draw from a simulation are a reflection of those two elements. Care should thus be taken by potential readers and users to ensure that any conclusions and/or decisions made are on the basis of a sound set of underlying assumptions.

All the results presented in Section 6 and supported with evidence in Section 7 provide a level of confidence that the simulation performs in a manner that approximates the features of a real-world application in the areas tested. Confidence in the simulation results could be further increased through additional validation against further empirical sources as well as further experiments that investigate facets of the simulation not discussed in this section. In the following section, we draw some conclusions from the simulation framework presented.

9. 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 to the investigation of software evolution that could potentially aid such issues. A simulation can be calibrated with a relatively small amount of empirical data. Once calibrated, the scope can then be broadened to include different environments with little or no effort. This reduces the need for long and expensive empirical investigations as well as analysis that might not be practical through direct measurement. A significant problem faced in empirical studies of software evolution is the cost and time involved in collecting data. Software systems take a long time to build and hence so does studying them. It is also hard to find organizations that collect and retain either relevant software measurements or the software artifacts themselves [Kem96]. The simulation framework aids such issues by providing a means for extending results generated by standard empirical measurement. Using a small set of empirical data as a seed, experiments could be run to extrapolate further measurements. Such extrapolations would be far cheaper, both in time and cost than similar empirical methods. A second advantage is the increased flexibility offered by a simulated environment where settings can be changed and their effects replayed at will. This increases the efficiency of the method further.

From a theoretical standpoint our simulation model can be used to investigate the interaction of forces that contribute to software evolution by building a causal model from individual underlying behaviors. Software becomes increasingly complex as these simple forces interact with one another. The simulation allows these various complex parts to be investigated in isolation as well as part of a collective model. This facilitates a model of software evolution built from individually substantiated parts. Much as in other disciplines, simulation provides 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. As a final point, we note that the full source code for the simulation framework and a downloadable executable of the tool are freely available at: http://www.benstopford.com/devsim/devsim.shtml

Future work will consider a number of possibilities with respect to the tool. First, a refinement of the user interface to be more user-friendly with particular respect to the structure of the system being analyzed. A ‘zoom’ in and out facility providing the user with the ability to explore the system at different levels of abstraction, might be beneficial to their overall understanding.

In the same spirit, provision of more user information about the effects of each evolutionary step is another potential source of tool enhancement. Finally, converting the tool as it stands to become an Eclipse plug-in rather than a stand-alone tool is a possible avenue for future enhancement and dissemination.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the useful and valuable comments and suggestions provided by the anonymous reviewers. The paper has benefited significantly as a result.

REFERENCES

[Bie03] "Design Patterns and Change Proneness: An Examination of Five Evolving Systems", J. Bieman, G. Straw, H. Wang, P Munger and R. Alexander, Proceedings of the IEEE 9th International Software Metrics Symposium (METRICS 2003), Sydney, Australia, pages 40-49.

[Boo94] "Object-Oriented Analysis and Design with Applications", G. Booch, 2nd ed., Benjamin/Cummings, 1994.

[Bria99a] "An investigation into coupling measures for C++", L. Briand, P. Devanbu and W. Melo, Proceedings of the 19th International Conference on Software Engineering (ICSE 97), Boston, USA, pages 412-421, 1997.

[Bria99b] "A Unified Framework for Coupling Measurement in Object-Oriented Systems", L.C. Briand, J.W. Daly and J.K. Wust, IEEE Transactions on Software Engineering, 25(1), pages 91-121, 1999.

[Bro75] "The Mythical Man-Month", F. Brooks, Addison-Wesley, 1975.

[Cap04] "Studying the Evolution of Open Source Systems at Different Levels of Granularity", A. Capiluppi, M. Morisio and J. Ramil, Proceedings of the 12th International Workshop on Program Comprehension, Bari, Italy, pages 172-182, 2004.

[Chap01] "Types of Software Evolution and Software Maintenance", N. Chapin, J. Hale, K. Kham, J. Ramil and W. Tan, Journal of Software Maintenance: Research and Practice, 13(1), 2001, pages 3-30.

[Chid94] "A Metrics Suite for Object Oriented Design", S.R. Chidamber and C.F. Kemerer, IEEE Transactions on Software Engineering, 20(6), pages.476-493, 1994.

[Cou03] "Trends in Java Code Changes: the Key to Identification of Refactorings", S. Counsell, Y. Hassoun, R. Johnson, K. Mannock and E. Mendes, Proceedings of ACM 2nd International Conference on Principles and Practice of Programming in Java, Kilkenny, Ireland, June 2003, pages 45-48.

[Cou06] "Common refactorings, a Dependency Graph and some Code Smells: an Empirical Study of Java OSS", S. Counsell, Y. Hassoun, G. Loizou and R. Najjar, International Symposium on Empirical Software Engineering (ISESE), Rio de Janeiro, Brazil, pages 288-296, 2006.

[Din04] "Open Source Software Development: A Case Study of FreeBSD", T. Dinh-Trong and J. Bieman, Proceedings of 10th IEEE International Symposium on Software Metrics, Chicago, USA, 2004, pages 96-105.

[Fen98] "Software Metrics: A Rigourous and Practical Approach", N. Fenton and S. Pfleeger, PWS, 1998.

[Fer04] “Extracting Facts from Open Source Software” R. Ferenc, I. Siket and T. Gyimothy. Proceedings of 20th International. Conference on Software Maintenance (ICSM 2004), Chicago, USA, pages 60-69.

[Find07] http://findbugs.sourceforge.net/. Accessed 21/07/07.

[For69] “Urban Dynamics”, Forrester, J. W., Cambridge MA: Productivity Press. 1969.

[For71] "Counter-intuitive Behaviour of Social Systems", J. Forrester, Technology Review, 73(3), Jan. 1971, pages 52-68.

[Fow02] “Patterns of Enterprise Application Architecture”, M. Fowler, Addison-Wesley Professional, 1st edition, 2002.

[Fow99] “Refactoring: Improving the Design of Existing Code”, M. Fowler, Addison Wesley, 1999.

[Gam95] “Design patterns: elements of reusable object-oriented software“, E. Gamma, R. Helm, R. Johnson, J. Vlissides. Addison Wesley, 1995.

[Gilb00] “Computer Simulation in Science and Technology Studies”, P. Ahrweiler and N. Gilbert, Springer 2000.

[Gnu08] http://www.gnu.org/software/classpath/. Last accessed 02/05/08.

[Gris93] “Automated Assistance for Program Restructuring”, W. Griswold and D. Notkin, ACM Transactions on Software Engineering and Methodology, 2(3), 1993, pages 228-269.

[Gir06] “Modeling History to Analyse Software Evolution”, T. Girba and S. Ducasse, Journal of Software Maintenance and Evolution, 18(3), pages 207-236, 2006.

[Hals77] “Elements of Software Science”, M. Halstead, Elsevier Science Inc., New York, NY, 1977.

[Harr99] “Experimental Assessment of the Effect of Inheritance on the Maintainability of Object-Oriented Systems. R. Harrison, S. Counsell and R. Nithi, Journal of Systems and Software 52(2-3), pages 173-179, 2000.

[Kelln99] “Software Process Modeling and Simulation: Why, What, How”, M. Kellner, R. Madachy, and D. Raffo, Journal of Systems and Software, 46(2/3), pages 91-105, 1999.

[Kels04] “A Simple Static Model for Understanding the Dynamic Behaviour of Programs”, P. Kelsen, Proceedings of the 12th International Workshop on Program Comprehension, Bari, Italy, pages 46-51, 2004.

[Kem96] “Need for more Longitudinal Studies of Software Maintenance”, C.F. Kemerer and S. Slaughter, Report from the Proceedings International Workshop on Empirical Studies for Software Maintenance, Monterey, California., 1996, Empirical Software Engineering: An International Journal, 2(2), pages 109-118, 1999.

[Kem99] “An Empirical Approach to Studying Software Evolution”, C.F. Kemerer and S. Slaughter, IEEE Transactions on Software Engineering, 25(4), pages 493-509, 1999.

[Kicz97] “Aspect-Oriented Programming”, G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, J. Irwin, European Conference on Object-Oriented Programming (ECOOP 97), Jyvaskyla, Finland, pages 220-242.

[Leh80] "On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle”, Journal of Systems and Software, vol. 1, pages 213-221 (1980)

[Mens04] “Analyzing the Evolution of Large-Scale Software”, Introduction to Special Issue, T. Mens, J. Ramil and M. Godfrey, eds, Journal of Software Maintenance and Evolution: Research and Practice, 16(6), 2004, pp. 363-447.

[Mey92] “Applying design by contract”, B. Meyer, IEEE Computer, 35(10), pages 40-51, 1992.

[Mil56] “The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information”, G. A. Miller: Psychological Review, 63, pages 81-97, 1956.

[Mub07] “Package Evolvability and its Relationship with Refactoring”, A. Mubarak, S. Counsell, R. Hierons and Y. Hassoun, Proceedings of Third International ERCIM Symposium on Software Evolution 2007, Paris, France, October 2007.

[Mun03] “Using Empirical Knowledge from Replicated Experiments for Software Process Simulation: A Practical Example”, J. Munch and O. Armbrust: Proceedings of the 2003 International Symposium on Empirical Software Engineering (ISESE’03), Rome, Italy, pages 18-27, 2003.

[Nas08] “An Empirical Study of Evolution of Inheritance in Java OSS”, E. Nasseri, S. Counsell and M. Shepperd, to appear in Proceedings of the 19th Australian Software Engineering Conference, Perth, Australia, March, 2008.

[Opd92] “Refactoring Object-Oriented Frameworks”, Opdyke, W. Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1992.

[Par72] “Information Distribution Aspects of Design Methodology”, D.L. Parnas. IFIP Congress (1) 1971, pages 339-344. 1972.

[Raffo96] “Modelling Software Processes Quantitatively and Assessing the Impact of Potential Process Changes on Process Performance” Raffo, D. – PhD Dissertation. Carnegie Mellon University 1996.

[Raffo00] “Empirical Analysis in Software Process Simulation Modeling”. D. Raffo and M. Kellner, Journal of Systems and Software 53(1), pages 31-41, 2000.

[Rob06] “Beyond Source Code: The Importance of Other Artifacts in Software Development”, G. Robles, J. Gonzalez-Barahona and Juan Merelo, Journal of Systems and Software, 79(9), pages 1233-1248, 2006.

[Set06] “Planning and Improving Global Software Development Process Using Simulation”, S. Setamanit, W. Wakeland and D. Raffo, Proceedings of the 2006 ICSE International Workshop on Global Software Development for the Practitioner, Shanghai, China, pages 8-14.

[Smith06] “Agent-based Simulation of Open Source Evolution”, N. Smith, A. Capiluppi and J. Fernandez-Ramil, Journal of Software Process - Improvement and Practice, 11(4), July/Aug, pages 423 – 434, 2006.

[Smith05] “A Study of Open Source Evolution Data using Qualitative Simulation”, N. Smith, A. Capiluppi and J. Fernandez-Ramil, Journal of Software Process - Improvement and Practice, 10(3), pp. 287 – 300. 2005.

[Snee04] “A Cost Model for Software Maintenance & Evolution”, H. Sneed, International Conference on Software Maintenance 2004 (ICSM 2004), Chicago, Illinois, US, pages 264-273.

[Sou07] www.sourceforge.net

[Stop06] “Simulating the Structural Evolution of Software”, B. Stopford and S. Counsell. Proceedings of International Software Process Workshop (PROSIM 2006), May, 2006, Shanghai, China. Pages 294-301, (Springer LNCS 3966, Editors Wang et al).

[Wern99] “Software Process White Box Modeling for FEAST/1”, P. Wernick and M. Lehman, Journal of Systems and Software 46(2-3), pages 193-201, 1999.