An Experimental Simulation of the Evolution of Software

Benjamin Stopford

MSc Advanced Information Systems project report,
School of Computer Science and Information Systems,
Birkbeck College,
University of London

19th September 2005

Abstract

This dissertation presents a simulation framework through which experiments into the evolution of software systems can be studied. Specifically, the evolution of a code base itself. The simulation framework is constructed in java and can be customised to specific experimental goals through a set of configurable plug-ins. When configured, the experiment is run to investigate the effects of different evolutionary behaviours in the resulting code base. The framework for the simulation is presented along with description of the pluggable extensions (plug-ins) and discussion of what it can and cannot perform.

A simple, default implementation of the plug-ins is provided which simulate the basic behaviour. The system, with the default plug-ins is then validated to ensure that they work correctly and that the system evolves the code base in a manor that is plausible based on current empirical observations.

Finally experiments are performed to validate this default behaviour. A specific experiment is also run to probe the effects that agent memory have on development in a multiple agent system. The results confirm expectations within acceptable error margins.

Acknowledgements

I'd like to thank my tutor, Steve Counsell, who has dealt admirably with the profusion of ideas from which this project was formed. He has been both a guide and a mentor and without his relentless help I may never had completed. Thank you Steve.

Glossary of Terms

Term	Definition
Agent	The concept of a developer in the system
Augment (Change Operator)	A Change Operator that represents an augmentation of existing behaviour so that it performs a conditionally disparate function.
Augmentation Ratio	This defines the proportion of code constructs in an augmentation that will be changed. For example if the augmentation ratio is 50% then half the code constructs that correspond to the Task will be augmented with the new Requirement.
Change (Change Operator)	A Change Operator that represents a change to the behaviour of existing code (e.g. bug fix or clarification).
Class	Represents a repository of functions and parameters.
Complexity Injection	Complexity Injection involves adding a random number of extra features to a code construct when it is created.
Conceptual Types	Tasks are related to one another through Conceptual types. This is similar to an IS-A relationship where the Conceptual Type is used to link two tasks that are similar.
Data Entity	An entity that has no functional content. It only holds state.
Entity	Represents a type of Task in a Requirement. An Entity infers that the Requirement Task represents a conceptual entity of the system that combines both state and behaviour. An example might be a 'Shopping Cart' in a online shopping application.
Epoch	One run of the simulation defining a single development cycle through the implementation of a set of requirements. .
Event	An action outside the system that causes a process to be initiated. Events exist as requirements and code entities.
Evolution Policy	The Evolution Policy is the plug-in that determines how the code base evolves (along with the Complexity Injector).
Function	Functions are analogous to Operations in the Requirements section. They represent the elemental unit through which the code base is built and linked.
New (Change Operator)	A Change Operator that represents new functionality that is constructed either from an existing requirement that is to be extended or a new system event.
Operation	Represents a type of Task in a Requirement. An Operation infers that the Requirement Task represents an operation that must be performed such as the "Check Out" function in a online shopping application.
Property	Properties represent local variables of a specified type.
Reference	A reference describes one Function or Property calling another. It also has a Coupling Type, determined by the evolution policy. Coupling types determine whether data is retrieved or changed in the reference.
Task	A subtask of a Requirement. Each task has a type which is used to indicate the fact that tasks within a requirement can differ. The types used currently in the simulation are Entity, Operation and Data Entity.

Task Type	This describes what the content of a task is. It can be one of Entity Operation or Data Entity.

Roadmap

Sections 1–4: Introduce the concept of simulation and its application in the context of software evolution.
Sections 5-9: Discuss the different aspects of the simulation implemented as part of this dissertation.
Sections 10-11: Discuss the results of the experiments performed with the simulation.
Sections 12-14: Discuss further experiments and associated work.
Sections 15-16: Closing remarks.

1 Background

1.1 Introduction: Simulation as a Tool for Experimentation

As a first year undergraduate studying mechanics the rules that underlie the basic operations of the universe began to crystallise with an attractive simplicity. Newton's laws of motion epitomise this by modelling the motion of all moderately sized bodies through a handful of remarkably simple laws. However the simplicity of such laws form a smokescreen that covers the layer cake of interactions that really exist in most real world systems. Seemingly simple structures move rapidly towards chaotic behaviour when combined. The predictable motion of a pendulum is a good example. On its own it moves in a predictable motion described by Newton's laws. However combine two pendulums together and rapidly the motion becomes random. Static analysis of such systems becomes impossible as the non-linearity that characterises their interaction quickly dominates.

The apparently random behaviour that results from the interplay of such forces is still predictable, at least statistically. Thus constructing a computer simulation of the duel pendulum model is a relatively simple task. It is simple because each law that drives the system is simple. Only the resulting behaviour, the interaction between the laws, is complex.

The study of software is fundamentally different to physics as with software there is no thorough understanding of the underlying behaviour, at least not as a cohesive and substantiated theory. However the software engineering field have produced many behavioural observations and rules that describe software evolution. Simulating these rules individually as well as combining their effects is well within the bounds of a software model. Simulation requires the definition of static rules that describe the system i.e. what exists and how those features behave individually. Interactions between these rules and features are then added. In the pendulum example this would be the modelling of rules such as force, mass and separately their interaction via Newton's laws.

In software simulation individual models are used in an analogous manor. First the laws which are known and understood are modelled in the simulation. Their interaction produces a complex system from the individually validated features. The research interest is then gained either from the repeated application of one law upon itself or the effect single laws have on the system as a whole.

Simulation is used extensively in the physical and social sciences, engineering, economics and finance to construct models of complex systems. Obscure and complicated problems that have been explored including commodity supply and demand [Med70], models of urban growth and decay [For69], Micro and Macro Economic models [For89] and amongst others, model of environmental damage [Ster00]. Such simulations, which are used to model real world systems,

are designed to display significant characteristics without replicating all the complexity that exists in the physical implementation. Thus simulation is most applicable when considering systems that display a level of complexity beyond that which static models or other similar techniques can usefully represent. Kellner et al [Kelln99] note that simulation can be used for modelling systems that display the three following behaviours:

1. System uncertainty and stochasticity.
2. Dynamic behaviour.
3. Feedback Mechanisms.

Systems which exhibit such behaviours, but whose underlying forces are at least partially understood make good candidates for simulation. The aim of the experimenter is to take those concepts that are understood and apply them collectively to investigate, quantify and promote their understanding.

1.2 The Application of Simulation in Software Engineering

Software is subject to a variety of factors that result in its degradation over time. This was noted in the early 1980's by Lehman in his law of Increasing Complexity [Leh80]:

"As an evolving program is continually changed its complexity, reflecting a deteriorating structure, increases unless work is done to maintain or reduce it."

This deteriorating structure is also associated with the concept of increasing entropy [Bian00]. As a response to such issues experts in the Software Engineering community have come up with a plethora of 'best practices' that denote how software should be constructed to best defeat such undesirable outcomes.

However such counsel is usually offered in isolation taking the form of simple generalisations that rarely account for the subtle interactions between forces that shape a piece of software as it evolves.

The evolution of structure in software systems displays all three of the complexities enumerated by Kellner. They result from the action of a variety of forces that mould evolution. Simulation provides a unique opportunity to model and analyse these forces as they act upon one another in differing situations. The simulation model can also be extended to evaluate the collective effects of the various mechanisms that have been suggested for managing the ever deteriorating structure. This provides validation of their intrinsic and collective benefit as well as offering the possibility of an enhanced insight into the relationships that govern this complex field.

1.3 Previous Work on Simulation

The majority of research using simulation in software engineering is concerned with the simulation of software process. Prominent examples of this include the modelling of project planning [Kelln91], defect levels and staffing profiles [Raffo96] as well as system size and effort trends [Wern99]. These differ from the simulation model presented here in that they investigate the processes by which people, technology and practices are organized to transform information, materials and energy into a piece of software.

Conversely, the proposal discussed in this paper looks only at the effect that evolution has on the structure of software at a source code level. There may appear to be overlaps here with process simulations, however the key difference lies in the structures that are under study. The aims of process simulations are to investigate and optimise the process through which code is constructed as part of multifaceted methodologies stretching over the project lifecycle. Instead the development simulation presented here focuses on how code is structured as a physical entity and how this structure varies over the evolution of a project. It is about the code structure rather

than the process through which it is generated. The processes that are modelled are incorporated so as to include their effect on the code structure.

2 An Introduction to the Simulation Model

This simulation models the growth of a fictitious system based on parameters and policies set up for each experiment. The research gain is achieved by changing the rules by which the simulation evolves and measuring how this affects the resulting code base.

This is best elaborated with a simple example: Let us assume that a principal is proposed that states that, as a system evolves, there is benefit in imposing a hard limit on the number of methods that a class can have. Thus when a class reaches a certain size it is refactored [Fow99] to move sections into delegate classes. An experiment is to be conducted that will investigate whether this proposition remains true as a system evolves.

Initially this could be investigated with a static analysis. Such an analysis might conclude that benefit would be gained as smaller classes are, on the whole, easier to understand than larger ones due to design principles such as information hiding [Par72].

Verifying this assumption in a traditional fashion would likely involve a long and potentially expensive empirical study. However useful insights can be gained by applying the proposition to an existing simulation and observing how it changes the evolution of the code base.

To do this a simulation experiment would be configured such that when a class evolves to a certain size it is refactored to include a delegate. The simulation would then be run, combining this new policy with a set of default policies that define how systems are believed to operate in general. The resulting combination will measure the interaction of this newly proposed evolution policy with those that are already modelled.

Such an experiment would not necessarily answer the question explicitly but would certainly provide potentially useful data from which other observations about the conjecture can be made. For example different distributions of results are likely to arise as the maximum class size threshold is varied. Forcing class evolution to divide at smaller thresholds is likely to result in many distributed classes which would reduce comprehensibility as structures become fragmented. If true, this would add a second, lower tail to produce a two tailed distribution of comprehensibility against maximum class size whose profile would be of interest.

As well as performing validation in the presence of other evolutionary factors, the simulation facilitates the investigation of repeated experiments within different environments. Such environments might include small vs. large projects or green field projects vs. ones that contain much rework etc.

3 Aims

The primary aim of this project is to develop a workable simulation model to facilitate experiments that investigate the macroscopic[1] evolution of software as the policy by which it evolves is changed. This is realized through the development of a simulation framework. The framework defines the basic rules of the system and facilitates the generation of an evolving code base. These basic rules are configurable through system settings. The role of evolving and measuring the system is left to customizable plug-ins which can be tailored to fit individual experimental aims.

[1] The divide between macroscopic and microscopic is taken at the level of functions i.e. code that exists inside functions is considered microscopic.

A default set of these plug-ins is provided and experiments are run to validate them against expected results. Care is taken to make sure that the simulation provides modelling of a sufficient number of attributes of the system to ensure that interesting experiments can be run. Conversely the simulation does not model so many attributes as to make it unwieldy and overly complex to operate.

4 Experimental Design: How to Simulate the Development Process

Kellner enumerated three complexities each of which is incorporated in the framework design (see section 2). The incorporation of each of these properties is discussed in the following sections:

4.1 System Uncertainty and Stochasticity

The process of software development is non-deterministic as ultimately it is a product of human endeavour. As a result any attempt to model it requires a stochastic basis. This stochasticity is injected into the simulation in two ways. Firstly the requirements generation process that feeds the simulation has stochastic behaviour. Secondly complexity is randomly injected into the code base as a final stage of the evolution process. Extra stochasticity can also be added by experimenters when implementing the various plug-ins.

4.2 Dynamic Behaviour

Dynamic behaviour in software evolution results from the complex interaction of the many factors that affect the transformation of requirements into code. The simulation provides a pluggable evolution policy in which the evolution of the code base can be controlled on an experiment by experiment basis. This ‘evolution policy’ can be altered to take into account a variety of factors when evolving the code. These will be discussed further in a later section.

4.3 Feedback Mechanisms

Feedback in the simulation is provided between the code base and the evolution policy (i.e. the plug-in responsible for evolving the code). This allows the simulation to be configured to change the way it evolves code based on feedback about the structure of the code being changed. For example the probability of refactoring a piece of code is likely to be reduced if that code is highly coupled. Thus a measure of the coupling of the function in the code base could be used as feedback into the evolution policy to moderate refactoring decisions.

5 Elements of the Simulation

5.1 An Overview of the Simulation Model

Simulations of any type must take an abstracted view on the environment of interest. Inside this view a selection of variables must exist whose individual action is predictable, but whose combined effect is not.

The simulation of the software construction process presented here takes a view that is limited to entities greater than or at method level. Manipulation of the model is performed at this level with the facility for finer granularity to be added stochastically.

The basic configurable elements of the framework are:

- **Requirements:** The Requirements Generator provides a utility that generates requirements through a stochastic but configurable process.
- **Evolution:** The Agent and Evolution Policy evolve the code base based on requirements they are given to implement.

- **Measurement:** Code Metrics provide a means for the Agent to evaluate the code base prior to changing it.

The fourth element of the simulation is the Code Base but this is considered a static entity in the framework rather than a configurable element (although potential for alteration is provided).

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 an evolution profile based on a set of predefined circumstances that the simulation framework supports. The evolution policy also takes into account feedback information from the Code Metrics applied as the code is changed. Feedback loops, formed from simple concepts, are responsible for many of the complex processes observed in dynamic systems [For69]. This is demonstrated in the figure 1.

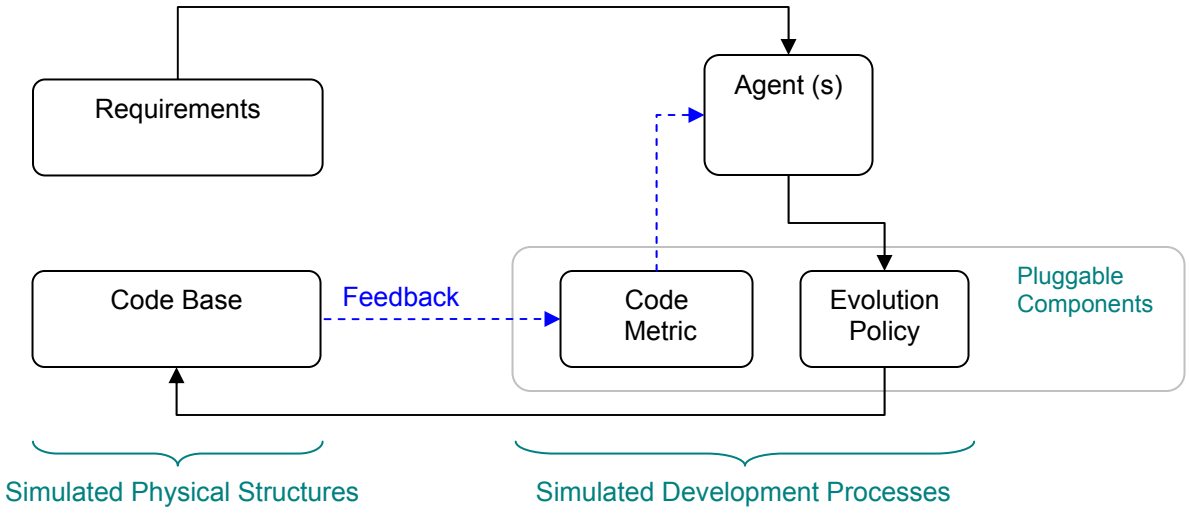


Figure 1: An overview of the basic elements of the simulation framework including the flow of data from Requirements to the Code Base via the Agent. It also demonstrates the feedback loop that exists from the Code Base back to the Agent via the Code Metric. The complexity metric and the Evolution Policy are pluggable units with the other roles being handled by the framework (configurable through the environment variables).

The simulation measures the code evolution process 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 by default. The costing model is split into implementation cost and comprehension cost (but this can easily be extended). The former being the cost associated with creating and altering code. The latter being the cost associated with comprehending the code base as it is changed. Thus the feedback from the code base to the costing model is installed.

The following sections provide an overview of the various elements that contribute to the simulation as denoted in figure 2.

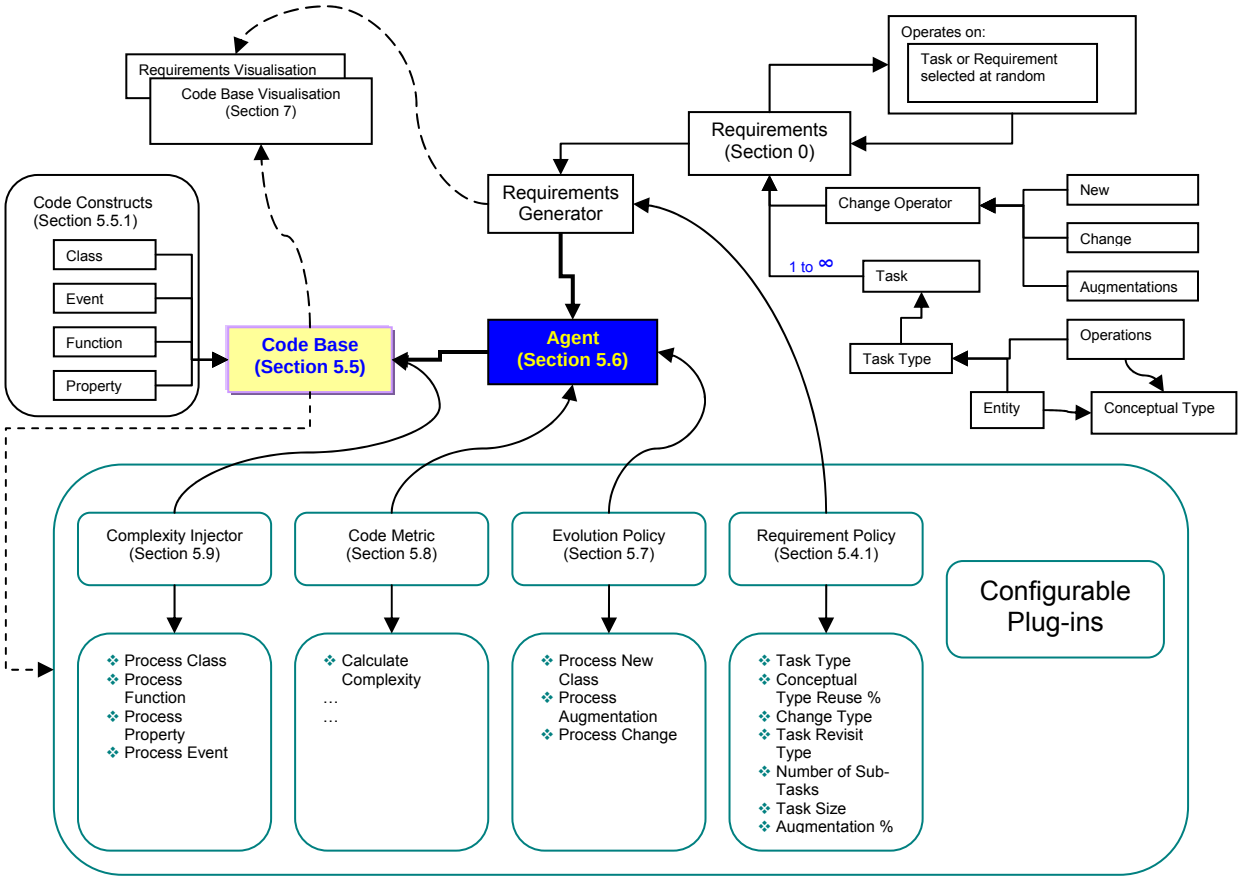


Figure 2: System Overview. Diagram of the main system concepts including how they connect together. Each of the major elements are discussed in the appropriate sections which follow.

5.2 Plug-Ins

The simulation framework is designed to facilitate experiments in software evolution. As such the majority of the framework is presented to the experimenter as a utility. This utility has specific sections, known as plug-ins, which are available for custom configuration. Plug-ins allow the experimenter to specify the logic through which the code base is evolved, within the context of a specific experiment, without having to worry about the larger concerns of the simulation itself. Physically the plug-ins are simple classes, written in Java™, which implement the appropriate interface. The experimenter supplies the Java™ implementation that most aptly fits their experimental goals.

5.3 Requirements

The simulation process commences with the generation of requirements. Requirements are structured collection of tasks that will be later turned into code by an Agent. The structure of requirements is deliberately kept simple by only including those concepts that are fundamental to the later alteration of code. This results in a requirements model that does not include all the complexities of a rigorous model.

The simple model views requirements as a set of tasks each having a specific type. Requirements themselves have a change operator that describes how the requirement will operate on the code base when it is implemented. Finally requirements have a starting point. Thus a Requirement contains:

- **A Set of Tasks**: Each sub-task has a **Task-Type** of either Entity, Operation or Data Entity.
- **A Change Operator:** The Change Operator describes how the requirement will operate on the code base. For example it could indicate that the requirement is for new functionality or a bug fix.
- **A Starting Point:** A Task or Requirement that represents a starting point for the piece of work. In the case of completely new functionality this takes the form of a system event.

5.3.1 Task Types – Entities and Operations

Each task has a type which indicates that tasks within a requirement can differ. The types used currently in the simulation are **Entity**, **Operation** and **Data Entity**. **Entities** correspond to physical or conceptual units of the application (such as a Shopping Cart) which are usually stateful but also contain functionality. **Operations** correspond to processes that the application performs (such as the Checkout action). Data Entities are entities that represent only data in the application. Tasks are related to one another through **Conceptual types**. This is similar to an IS-A relationship where the Conceptual Type is used to link two tasks that are similar.

5.3.2 Change Operators

Entities and Operations define the content of Tasks in a Requirement i.e. they say something about what that task will entail. Change Operators however determine how the task will actually operate on the code base. For example it could be a new piece of functionality or alternatively a bug fix. The change operators modelled in the simulation are New, Augment and Change.

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 behaviour of existing code (e.g. bug fix or clarification).
Augmentation	Previous requirement	An augmentation of existing behaviour so that it performs a conditionally disparate function. The degree of difference is recorded as part of the requirement.

Augmentation and Change

Augmentation is important as 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 base 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 which in turn acts to degrade the conceptual cohesion of the code block.

Augmentation and Change are fundamental to understanding evolution as they represent some of the most basic forces that must be harnessed if software evolution is to be controlled. An example software implementation that handles augmentation is put forward by Gamma et al [Gam97] in their implementation of the Strategy pattern (amongst others). Gamma et al use basic OO principals to encapsulate the section of a code module that is changing into an underlying strategy using polymorphism. Such a measure simplifies the primary module as the augmented behaviour is extracted to the strategy class rather than being a complex set of conditionals inside the primary module itself.

5.4 The Requirements Generator

The Requirements Generator is responsible for creating a requirement for implementation in the code base. In doing this it performs the following steps.

- Determines the Change Type. The first Requirement is always a New.
- Creates a requirement.
- Assigns new Tasks to the Requirement, Each task being assigned a Task Type.
- Sets the Change Type of the task.
- Selects an existing task or system event for the task to operate on.

Each of these steps is driven by a stochastic process. However the distribution of the selections that are made can be controlled via the Requirement Policy.

5.4.1 The Requirement Policy

The Requirement Policy allows control over a number of the facets of the requirement generator. These are enumerated in the table below.

Setting	Description
Task Type	Determine the distribution of Task Types that are created.
Conceptual Type Reuse Percentage	% of overlap between Task Types assigned to tasks.
Task Revisit Type	Tasks are revisited by new requirements (unless they are of type 'New'). This setting allows control over the distribution of change by task type (i.e. Entity, Operation etc). So for example it could be configured so that Operations are changed more often that Data Entities.
Mean Number of Tasks per Requirement	Determine the mean number of tasks in a requirement. The more tasks the larger the impact on the code.
Mean Task Size	Provide and average size indicator for each task. The larger the value the greater effect it will have on the resulting code base.
Calculate Change Type	Determines the Change Type that will be applied to each new Task in a Requirement.
Augmentation %	Determines the degree of change that will occur during an Augmentation. This is interpreted as the number of existing tasks in the requirement that will be augmented. For example an augmentation percentage of 20% would result in changes in one in five of the functions in the class that is being augmented.

The default implementation provided in this dissertation is discussed in section 8.1.

5.4.2 Requirement Chaining

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. This structure is summarized in figure 3.

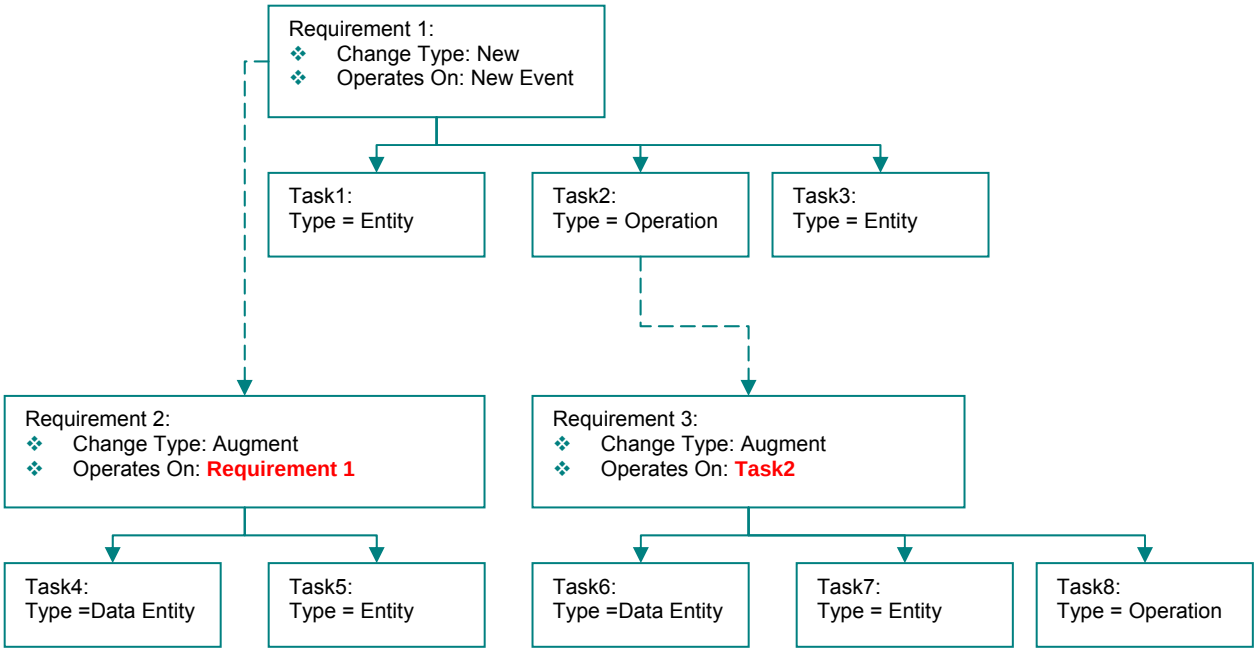


Figure 3: The hierarchical relationship between tasks and requirements. Only requirements have operators but these can operate on either a task or another requirement.

5.5 The Code Base

The code base exists as a singleton in the simulation and acts as a repository for different code constructs that are created and linked together by Agents (simulated developers). The code base allows the modelling of a set of code constructs. These constructs can then refer to one another via the various calls made open to them by the code base such as “Reference” or “Create Function”. The code base encapsulates calls that create code constructs and references so that it retains responsibility for enforcing integrity within the structure that is created. The rules enforced are comparable to those found in a real life code base and are discussed in the following sections.

5.5.1 Code Constructs

The code constructs are based on the work suggested by Kelsen [Kels04].

- Class: Represents a repository of functions and parameters.
- Function[2]: Functions are analogous to Operations in the Requirements section. They represent the elemental unit through which the code base is built and linked.
- Event: Events also exist inside classes and denote an interaction with an event outside of the system.
- Property: Properties represent local variables of a specified type.
- Reference: A reference describes one Function or Property calling another. It also has a Coupling Type, determined by the evolution policy. Coupling types determine whether data is retrieved or changed in the reference.

Properties represent the data manipulated by functions and are stored at class or global scope. Properties allow the simulated code base to model state as references from functions to properties. These references can have different Coupling Types which signify whether data is

[2] Kelsen [Kel04] actually uses the term Operation rather than Function. Function is used in this context due to the earlier naming clash.

being used or changed, providing a facility for the analysis of state changes. The intention is for knowledge of such stateful changes to be utilised in the code metrics and evolution policy. As an example the code complexity might be considered to increase if a function uses a property that is changed by a number of other functions in that class.

5.5.2 The Execution Path

The execution path of the simulation starts at one of the system events (there is a specific GUI view for analysing the simulation from this point of view, described in section 7.4). It then flows through all references to that initial system event (which is a type of function) recursively until they complete.

The code base in the simulation is constructed in such a way that this execution path is always enforced (creation of a new code construct requires the specification of the calling one). The various plug-ins always have access to the calling function so that it is easy to extend them within different experiments.

Properties do not participate in execution. Instead they are bypassed with a reference directly to the function in the property that they utilise. This is explained further in section 5.5.4.

5.5.3 Functions

Functions are the basic unit for the simulation. Functions extend their influence by coupling themselves to other functions either in the same class or a different one. In this manner the code executed by the function is increased. They can also reference Properties.

The model focuses only on structures at or greater than a function level. Provision for extending this further has been made in the Complexity injector (section 5.9).

5.5.4 Properties

Properties are the least intuitive construct as they do not participate in the execution path of the simulation. They are introduced to model the stateful linkages between methods in classes (or globally), representing the real world encapsulation of state observed in fields. Possible uses of Properties could include the evolution of Cohesion.

During the creation of a Property the following events occur:

- The property is created either in a class or globally.
- It is assigned a type. This corresponds to the type of object that it represents.
- A reference is made from the utilising Function to the Property.
- A reference is made from the Property to the class that describes its type.
- A third and final reference is made from the calling Function to the Function that it intends to use in the property's type. This is described in the figure 4.

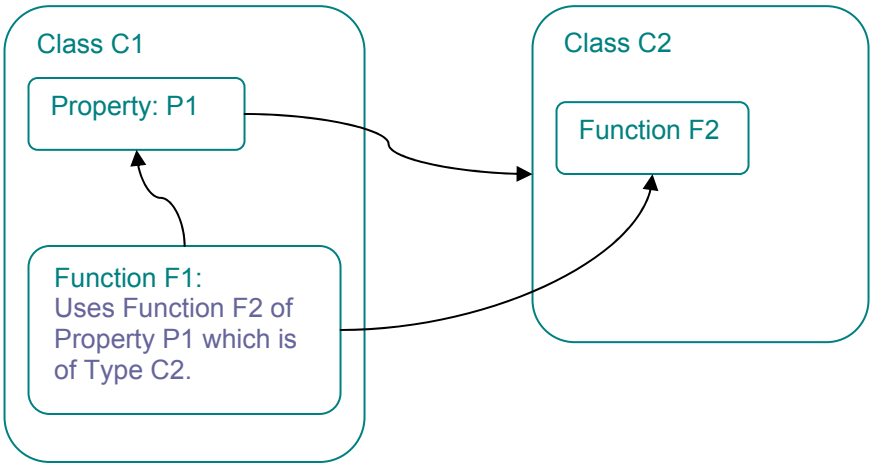


Figure 4: Demonstration of the various references that are created when a Function F1refers to a Property P1 inside its own class. The property is of type C2 i.e. the property is an instance of C2.

5.5.5 Rules of the Code Base

All creational routines are performed inside the code base class. This ensures that the rules are complied with and that storage constraints are obeyed. In addition the following rules are applied:

- All code is ultimately connected to an initial system event. This arises as the Code Base enforces that all Code Constructs must lie on a run. A run is defined as an execution path that originates from a System Event, then continues iteratively through all connected functions until they complete.
- Code constructs must be added to classes (with the exception of global properties which are stored in the code base).
- Properties can be class or global scope.
- Functions can refer to Properties in their local class or refer directly to Functions in the current or in other Classes.

5.6 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' 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 a plug-in that defines the fundamental operations that turn a single new Task into Code in a specific experiment. The agent is responsible for facilitating this (locating the class to extend if required etc). Details of the agent are based on the model described by Gilbert [Gilb00].

5.6.1 Multiple Agents and Agent Memory

Each agent has a memory of the code constructs that they were responsible for implementing in the code base. This can be used in 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 a randomly selected agent taken from the pool. Agent memory is built as agents create code. In the current implementation agent memory is only incremented for code that is created. However provision has been made for this to be extended to include code that is changed or viewed by the agent.

The agent's memory also leaks over time[3] based on an environment variable that controls the number of epochs (implementations of a complete requirement) before the agents memory fades completely. The fading memory of each agent is available to the experimenter in the various plug-ins.

The agent's memory is incremented using a code construct called the Costing Proxy. This provides a layer between the API, used by the various plug-ins, and the code base itself. All creation events are intercepted and increment the agent's memory.

5.6.2 Agent Response to different Change Types

The action of the agent is separated into sections based on the change type of the requirement as the agent must responds differently to different Change Types. The various behaviours are enumerated in the following three sections.

Change Type of New

New functionality is added either to a system event of a task or requirement that is to be extended:

System Event Creation:	If the requirement operates on a system event then this is effected in the code base. This involves the creation of a new class and function for the event.
Establishing a Starting Point:	The starting point is either a new system event (as above) or the first class/function in the task/requirement that is being changed/added to.
Entity Creation:	The introduction of a new entity always results in the creation of a new Class. Classes always contain a stateful structure.
Function Creation:	Functions are added to the starting class.

Change Type of Augmentation

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.

System Event Creation:	N/A
Establishing a Starting Point:	If augmenting a Task then the starting point is all classes and functions that correspond to that task. The selection is reduced by the Augmentation Ratio.
Augmentation:	The augmentation of each function deemed valid by the agent is sent to the Evolution Policy for augmentation.

Change Type of Change

The task to change and the new change task are both passed straight to the evolution policy for implementation.

5.7 The Evolution Policy

The Evolution Policy is the plug-in that bares most responsibility for evolving the code base and thus is a focal point defining experiments. The experimenter must implement four functions in the Evolution Policy to cause the code base to evolve. Each one corresponds to the conversion of a

[3] It is assumed that implementation cost is proportional to time. Thus time is taken from the total Cost of implementation so far in the code base.

different Task with a specific Change Type into code. The signatures on the interface are displayed below:

```
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);
```

The evolution policy also provides a set of utilities that allow the experimenter to customise their implementation. These include:

- 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 (see section 5.6.1).
- 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 which is used as a measure of the experiment (see section 5.8).
- Coupling Type: A specific coupling type is associated with References as they are created. The specific variety of coupling type used is determined by the evolution policy (a basic coupling type is used by default). It is proposed that future experiments will define custom coupling types which can subsequently be used in the evolution policy and code metric to provide feedback on the coupling of structures that exist in the code base.

Additional utilities are available but have not been validated with experimental results. These are discussed in section 12. The default implementation provided in this dissertation is discussed in section 8.2.

5.8 Code Metric

The Code metric is a facility by which the complexity of a code construct can be measured prior to it being extended, augmented or changed. The Code metric is one of the key plug-ins that would likely be customized in experiments. A basic implementation for a complexity metric is provided as part of this dissertation. This is discussed further in section 8.4.

5.9 Complexity Injection

Complexity Injection involves adding a random distribution of extra features to a code construct when it is created. This is the fundamental means through which low level complexity (references, properties etc) is added. Complexity injection occurs whenever a new code construct is created. It exists as a plug-in with a simple signature so that it is easy to alter the level of complexity injected into any specific experiment. The default implementation provided in this dissertation is discussed in section 8.3.

The Complexity Injector provides the possibility for increasing the level of granularity that the simulation supports. Whilst not implemented in this version, the complexity injector could be changed to inject lower level code constructs such as 'if' statements, loops and local variable assignments. This would provide a finer level of granularity as well as allowing more realistic code metrics and evolution processes to be applied to the code base.

5.10 The Difference between the Evolution Policy and the Complexity Injector

The Complexity Injector and the Evolution Policy have similar, but fundamentally disparate roles. The Complexity Injector is used only when a new Code Construct is created (It is triggered automatically from the constructor). Its role is to add complexity to new constructs so that the code base evolves with sufficient detail to make it realistic. Thus it takes the responsibility for this more mundane task away from the Evolution Policy.

The Evolution policy is left to add more complex and purposeful features to the code base. Such features tend to arise from the larger evolutionary goals of the experiment. For example whenever augmenting a class the Evolution Policy may dictate the creation of a new Class. This would have complexity injected into it like any other class. However the Evolution Policy would then add more interesting and focussed features such as tying the new class back to the one being augmented with various references and parameters.

In summary, the Complexity Injector is responsible for the monotonous detail that must be added to all code constructs when they are created (classes need functions and references etc). The Evolution Policy is responsible for shaping how the structures between classes and functions evolve, beyond the 'random' growth imposed by the Complexity Injector.

6 Measuring the Simulation

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

- **Implementation Cost**: The cost incurred in creating new code. This is proportional to the number and type of code constructs created (the cost for each construct is configured in the Environment Variables).
- **Metric (Comprehension) Cost**: The metric cost emulates the action of an agent comprehending the code base prior to making a change. The term metric cost is used as the cost can be based on any 'metric' devised from the measures available in the simulation.

The 'Implementation Cost' is calculated by the Costing Proxy. This class sits between the Evolution Policy and the Code Base and increments the cost of code units as they are created. In its default implementation the overall implementation cost is proportional to the number of code constructs that have been created. The cost weighting of each type of code construct is defined in the Environment Variables.

The cost incurred by comprehending code prior to making a change is more complex. It is calculated through feedback from the CodeMetrics class which takes into account various parameters such as the agent's knowledge and apparent complexity the source code itself. The metric implementation exists as a plug-in so can easily be varied.

Cost also provides a notion of time to the simulation. The basic unit of time is the average costing for an epoch. The total costing in the simulation run is also tracked (in the AgentController class).

7 The Graphical User Interface

The GUI has three views, one for the Requirements and two for the Code Base. It also displays summary data concerning the progress of the simulation. The GUI is designed to facilitate comprehension of the code base structure as the simulation evolves. This allows the experimenter to get a feel for how different factors that have been introduced affect the evolution.

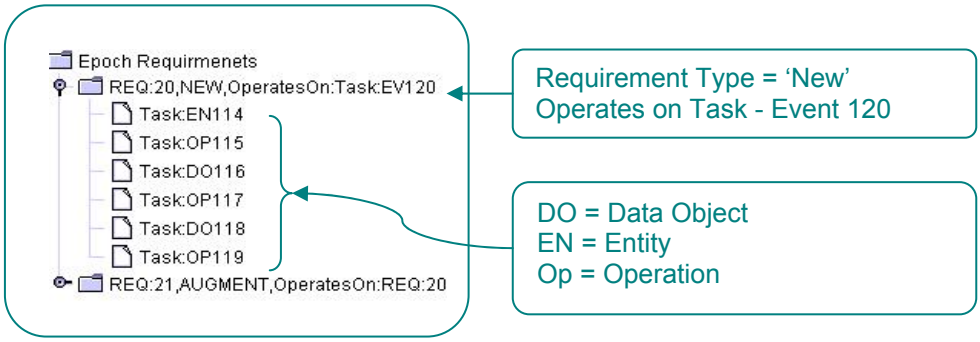
7.1 Running the Application

The application can be downloaded and run from the web page detailed below. A copy of the source code is also available so that the various plug-ins can be altered.

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

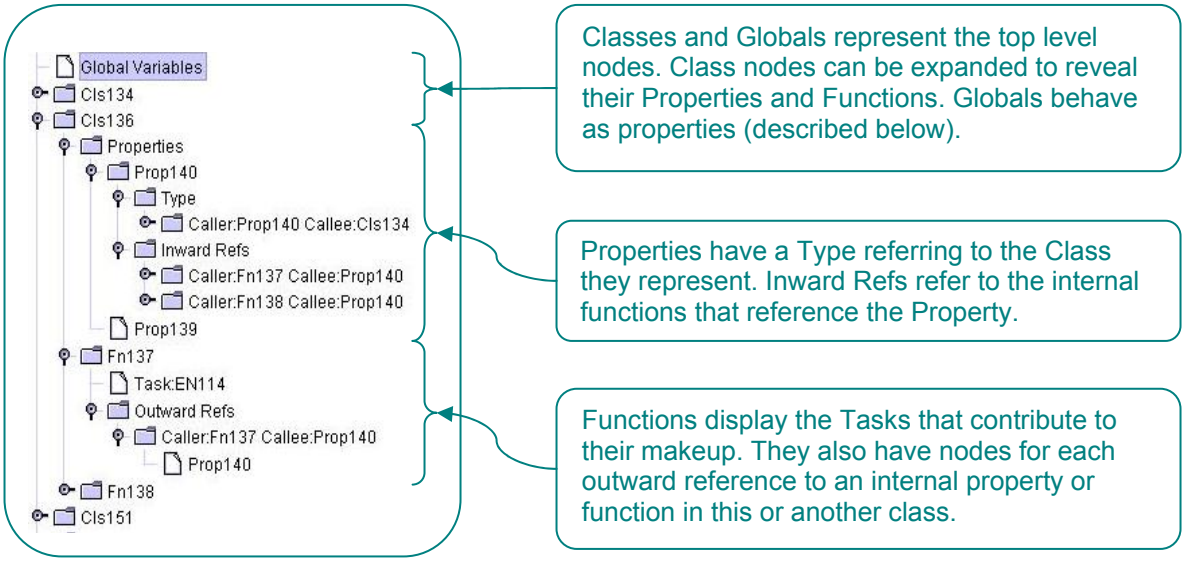
7.2 The Requirements View

The Requirements view presents a tree view of the requirements that have been generated. When the application starts a single requirement (of type 'New') is added. The user can click on Requirements to view the included Tasks.



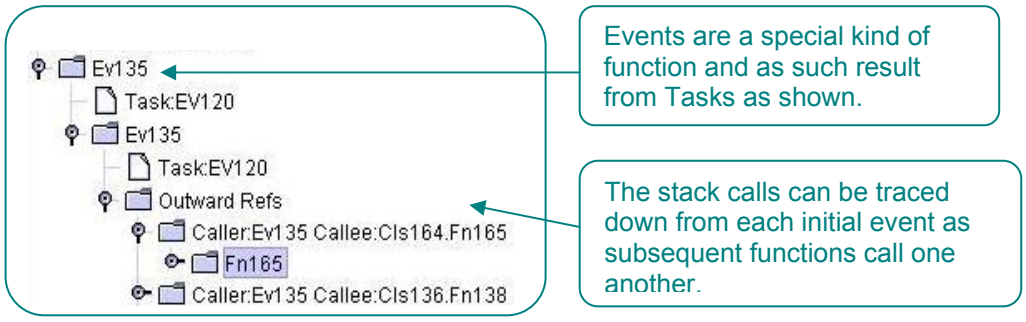
7.3 The Code Base Class Drill View

The GUI supports a view of the code base that allows class hierarchies to be analysed. 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 be drilled iteratively to view the whole call stack.



7.4 The Code Base Event View

The GUI supports a second view of the code base that allows system events to be traced through the resulting code that they execute. 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.



7.5 The Readout Panel

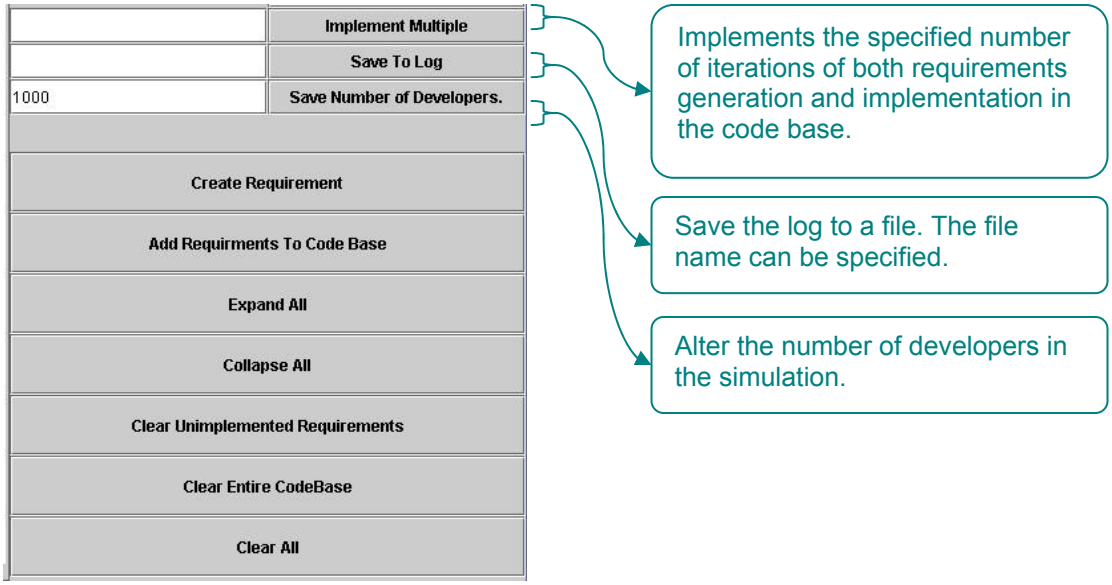
The control panel displays features of the evolution as the code base evolves. All readouts refer to the code base as a whole.

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

Readout	Description
Epochs Completed	The number of Requirements that have been added to the code base.
Tasks Completed	The number of Tasks that have been added to the code base.
Total Classes	The total class count.
Total Functions	The total function count.
Av Functions Per Class	The average number of functions that exist inside each class one average over the whole simulation run.
Av Tasks Per Function	The average number of tasks that exist inside each function on average over the whole simulation run.
Implementation Cost	The cost associated with the creation of new code.
Metric Cost	The cost associated with the metric output for classes that are extended and changed (or other implementation configured by the user in the evolution policy).

7.6 The Control Panel

The Control Panel provides control for the simulation itself. The buttons provide the following functions:-



Function	Description
Create Requirement	Add a new requirement to the requirements list. This will add the requirement so that it can be viewed but will not add it to the code base.
Add Requirements to Code Base	Adds the created requirements to the code base.
Expand All	Expands all nodes in the visible tree pane.
Collapse All	Collapses all nodes in the visible tree pane.
Clear Unimplemented Requirements	Clears all requirements that have not been implemented in the code base.
Clear Code Base	Clears the code base but not the requirements.
Clear All	Clears Code Base and requirements.

7.7 The Data File Output

The output of the simulation is sent to a file. This file is saved in a tab delimited format and contains all empirical data provided in the GUI itself. The path to the output file is defined in the EnvironmentVariable class and the name can be overridden in the GUI.

8 Default Plug-in Implementations

This section discusses the workings of each of the four default plug-ins supplied and shipped with the simulation. They define a very basic set of policies through which the fictitious code base can be evolved. They are estimates and as such do not necessarily represent an accurate depiction of the evolution of a real code base. Instead they provide a starting point from which to validate basic behaviours of the framework. It is anticipated that future experiments will improve on the basic assumptions they make, incorporating more realistic evolution policies and metrics. It should also be noted that in each case the programmatic configuration of these plug-ins allow them to be easily extended to support more complex responses.

8.1 The Default Requirements Policy

The Requirements Policy defines the various ‘ingredients’ used by the Requirements Generator to create requirements. These generally represent ratios between the various types that are available.

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

8.2 The Default Evolution Policy

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. This should be extended in specific future experiments.

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 function 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: o Create a new class for the entity. o Create 0-2 functions to the base class. o Add 0-2 references between base and Entity classes ❖ If the Task Type is an Operation then: o Create 0-2 functions to the base class. o 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)

8.3 Default Complexity Injection

Default implementation for complexity injection is:

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.
Event Creation	No Implementation.

8.4 Complexity Metric

The default behaviour for measuring the complexity of a function is defined by:

- The number of Requirement Tasks that contributed to the function.
- The number of functions that refer to it (i.e. the amount that it is reused).
- 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) drops to zero. The concept of memory leaks is available but is not utilised in this implementation.

9 The Software Implementation of the Simulation

The framework for the simulation consists of 42 java classes including the four plug-ins. This section documents some of the major constructs in the system as well as commenting on the software architecture implemented.

9.1 Design Considerations

Care was taken in the simulation to ensure that the code is structured in a way that maximises extensibility whist minimising the quantity of code that the experimenter need be concerned with. This is achieved through the architectural aim of separating mechanism and policy[4]**.** In this context the ***mechanism*** is the simulation framework with its infrastructure for the creating and extending a factious code base. The ***policy*** is then the various plug-ins that define how the code base is extended, measured etc. Thus, for the most part, the experimenter need only be concerned with the five plug-in classes, delving into the rest of the code base only when the fundamentals of the framework need to be changed.

An important design consideration is that almost system settings result from programmatic policies. These either implement some function, such as evolving a particular type of requirement or return a distribution of values such as the different Requirement Types used to generate requirements. Keeping these policies code based rather than simple constants or buttons in the GUI retains a high level of extensibility in the framework.

As the simulation grows its complexity will increase as additional simulated factors are added. It is anticipated that this increase in the complexity of the system will require a flexibility that can only be offered by programmatic policies. Simple constants could not deal with the various conditions that are likely to ensue as the simulation size increases. The drawback is that a dynamic class loader is required to make changes whilst the simulation is running. However free class loading utilities such as Eclipse [Eclip] this is not considered to be too much of a hindrance.

9.2 The com.devsim.plugins Package

Each plug-ins represented as a policy which is a design mutated form 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:

[4] The idea of separating Mechanism and Policy comes from the notion that the human mind can only deal with a finite number of concepts [Mil56]. This concept has been adapted and reused by numerous authors including Parnas [par72] often going under the names such as Abstraction and Design By Contract [Mey92], [Fow97], [Fow02].

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

These provide access to customisable features for the simulation as discussed in the previous sections.

9.3 The Code Base (com.devsim.code.CodeBase)

The code base is the class responsible for holding all the simulated code constructs and the relationships that tie them together. In particular all construction of code constructs is performed in the CodeBase class via the “create…” methods.

9.4 The Code Base API (com.devsim.code.CodeBaseAPI)

The code base API interface provides an API to external uses (such as the plug-ins package).

9.5 The Costing Proxy (com.devsim.code.CostingProxy)

The Costing proxy implements the API interface so that it can act as a proxy between plug-ins and the code base. It calculates the cost of a specified set of calls. The costs for the creation of each type of code construct are defined in the EnvironmentVariable class. The costing proxy increments the agent’s memory only when code constructs are created.

9.6 The Agent (com.devsim.evolution.Agent)

The agent facilitates the evolution performed by the Evolution Policy. It defies the starting class and function to use based on the Task and mediates the appropriate executions of the evolution policy.

9.7 The AgentFactory (com.devsim.evolution.AgentFactory)

The Agent Factory is responsible for creating and supplying the multiple agents that may exist in the system. The number of agents in the simulation is controlled by a setting in the file EnvironmentVariable.java in the plug-ins package. When multiple agents are specified a different Agent will be selected from the pool for each new requirement that is implemented.

9.8 Memory (com.devsim.evolution.Memory)

The Memory class holds the agents memory. In particular it allows the calculation of a discount factor which simulates the agent’s loss of memory over time (The basis for Time in the simulation is taken from the average effort required to implement a requirement). Memory is currently not set to leak but this can easily be changed to leak by altering the CodeMetrics class to use the Memory.remembersPortion() method in the Memory class.

9.9 Random Number Generator (com.devsim.utils.RandonGen)

The random number generator provides all stochasticity to the simulation. It has a range of utility functions that perform useful tasks other than just generating random numbers. These include selecting random values from lists, perform a desired function a random number of times and generate random Booleans and percentages.

10 Experimental Method

10.1 Experimental Design

Experimenters must design experiments that investigate features to which the simulation is sensitive. These are typically different methods by which code can evolve at a macroscopic level (i.e. at the level of functions, parameters and their interaction). The design must follow one of the following two approaches:

- Investigate the interaction between a specific evolutionary factor and other factors which are installed in the simulation (Where a 'factor' describes something that shapes evolution such as refactoring).
- Investigate the incremental effects of evolutionary factors on themselves through repeated application of the specific factor under investigation. Alternatively the reaction of the factor to different running conditions could be investigated (such as environments with different proportions of new development, change, augmentation etc).

Experiment designs must consider:

- **Modelling of the Evolutionary Factor**: How the new factor is to be modelled 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, via the metric, as described in Figure 1.

10.2 Experimental Method

The experiment method is defined by the following steps:

- Identify the problem to be investigated.
- Develop a dynamic hypothesis to explain the cause of the problem.
- Map each part of the dynamic hypothesis to its appropriate plug-in in the simulation framework.
- Make the relevant plug-in alterations required to fulfil the hypothesis.
- Tests the simulation model to ensure that it reproduces real world behaviour.
- Add the implemented property 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).
- Devise and test alternative policies that will also solve the problem and test via previous steps.

10.3 Tuning the Experiment

The second phase involves tuning the experiment so that it behaves in an expected manor. Initially this is best performed with the user interface views (see section 7), supplying small numbers of requirements so that the code base does not bloat. Tuning is useful for ensuring that the changes made to the evolution policy have the required effect and the code base structure is evolving correctly.

When tuning the simulation it is useful to break it down into its simplest units so that each section can be tested separately. This can be done by altering the various settings (particularly in the Requirements Policy) so that each feature can be tested in isolation. For example testing, with the requirements policy configured to only allow one Change Type at a time, allows each section

of the evolution policy to be tested independently of the others. Further tuning can then be performed through analysis of the output file.

10.4 Running the Experiment

Experiments are configured based on the required plug-ins and the associated simulation environment. Nominally experiments take the form of several runs in which a single parameter is varied in either the simulation environment or one of the plug-ins.

In the results presented in this dissertation each set is taken as the average of three separate runs. The output is then exported for graphical analysis.

11 Results: Experiments and Calibration of the Framework

The simulation framework is calibrated through the execution of a set of experiments that analyse performance over different variations on the standard settings. 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.

Calibration (1): The Statistical Variance of Results

Many of the parameters used in the simulation have an underlying statistical variance making each evolution of the system slightly different. It is therefore important to provide measure of the implicit variation in results. The below table demonstrates the average standard deviations for the linearly evolving variables taken over 900 measurements of three evolution profiles. The results for these measurements are shown in full in Appendix (B).

	Std Dev	Mean
Class	8.12	261
Function	86	1552
Tasks per Function	0.0187	1.1680
Functions per Class	0.3041	5.7414

Experiment (1): Increase in Source Code Size under Standard Settings

Aim

The aim of the first experiment is to ensure that running the simulation under standard conditions (i.e. where the various experimental parameters are set to ‘typical’ values) causes the code base to expand in size in a manor that approximates a real software project.

Method

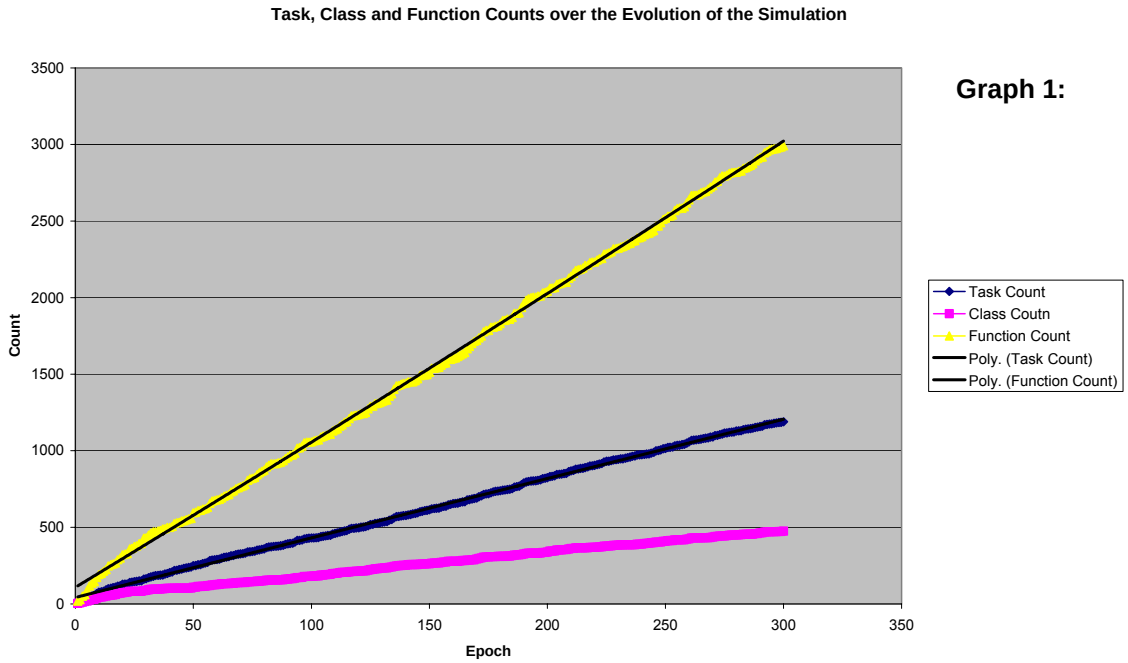
The experiment was run for 300 Epochs (requirement implementations) with the first 20 Epochs being exclusively ‘New’ requirements. The change type ration for this experiment was set to:

Change Type	Ratio
Augment	15%
Change	45%
New	40%

And each result represents the average formed over three separate runs.

Results

The results of the class, Task and Function counts over the evolution is shown in Graph1.



Graph 1:

The increase in task, class and function count are all linear over the life of the evolution which is expected as there is no modelled 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 behaviour is corroborated by an empirical study made by Capiluppi et al [Cap04] of the ARLA open source system. In this 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 results for Capiluppi's experiment are shown in Graph 2 for comparative purposes.

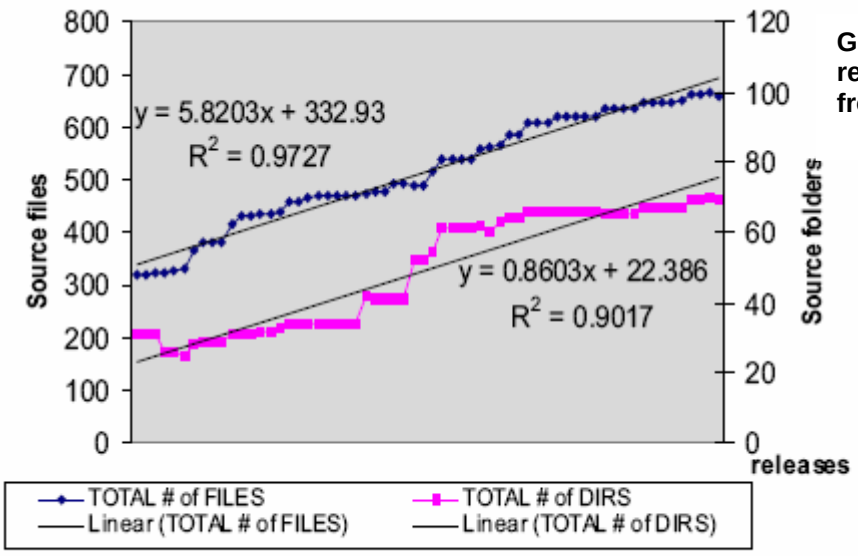


Figure 3 - Files and folders growth

Graph 2: reproduced from [Cap04]

Capiluppi et al also provide a study of the distribution of average lines of code per file over various releases. Their results show the average number of lines per file increasing slightly as the system evolves. This increase is approximately linear with a small positive gradient as shown in Graph 3.

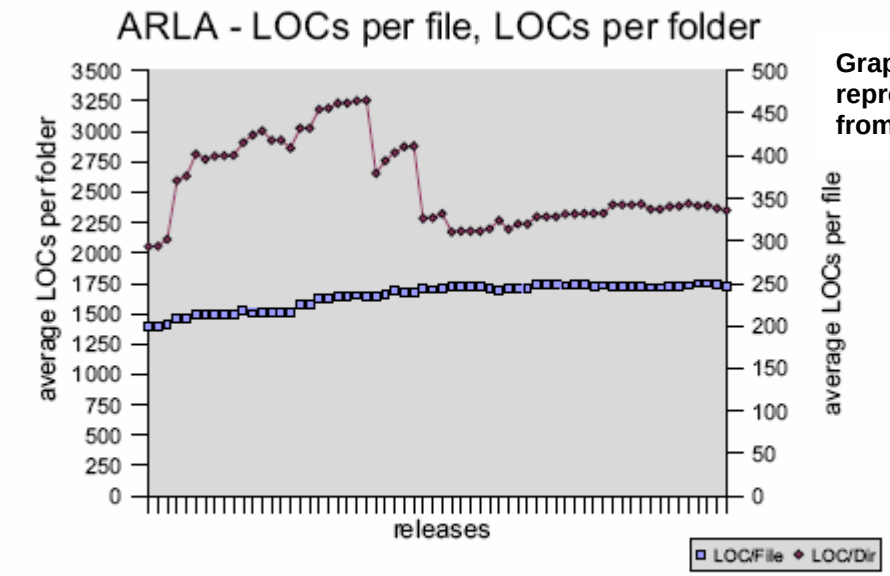
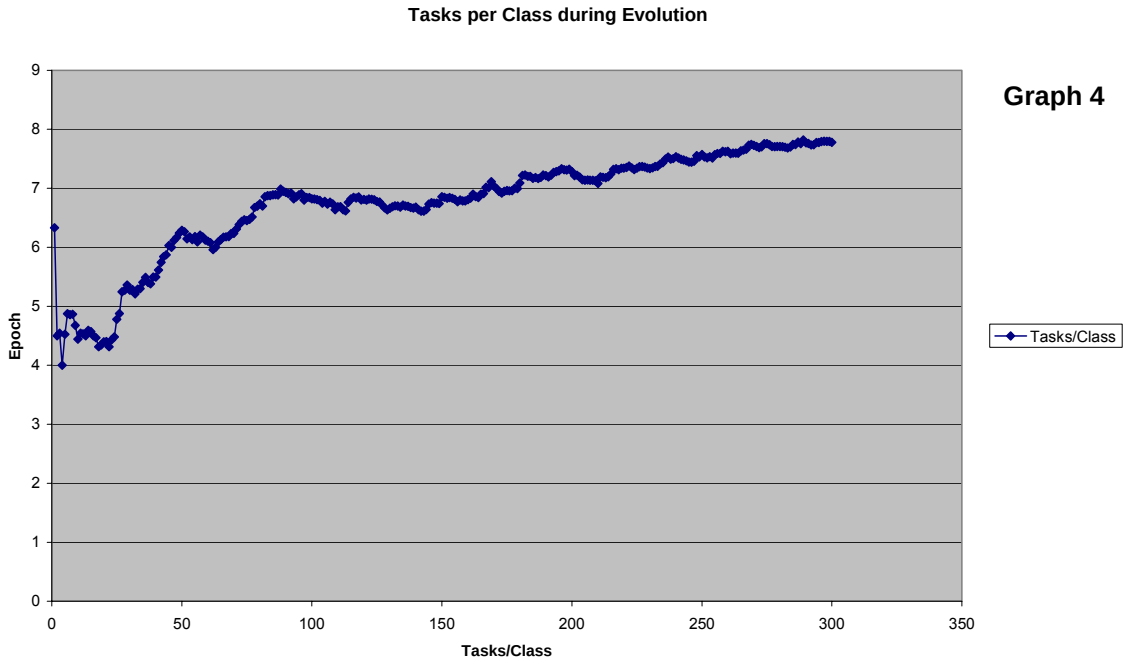


Figure 7 - Average size in LOC for folders (Y axis) and files (Y secondary axis)

Graph 3: reproduced from [Cap04]

A comparable result was generated with the simulation framework and is displayed in the Graph 4. This chart shows the evolution of task density over the simulation where task density represents the average number of tasks per class. The comparison assumes that on average the number of tasks will be proportional to the number of lines of code in a real application.



Graph 4

Both Capiluppi's results and those formed through the simulation framework are approximately linear with a small positive gradient. The initial readings show some variation from this trend but this is considered to be normal when the simulation is stabilising. The functions per class and tasks per function (which theoretically would combine to give the tasks per class) are shown in Appendix (A). These results validate the growth of the simulation in terms of number of files and file size through replication of a profile that is empirically substantiated.

Experiment (2): The Effect of Requirement Type on Code Comprehensibility

Aim

The aim of this experiment is to investigate whether the Requirement Type has an effect on the comprehensibility of the code base that is produced. The expectation is that this link should hold. The default code metric is dependent on the Task density in the code. Thus Change Types such as New, which are associated with the creation of high numbers of classes, should result in higher comprehensibility as the density of Tasks overall is lower. Conversely Change Types such as Change or Augment will operate on existing code, increasing the task density and this should be measured in the metric.

Thus it is expected that increasing the proportion of the requirement type "New" would reduce the metric count as more classes would be created, decreasing the task density and coupling. Conversely, increasing the proportion of "Changes" and "Augmentations" is expected to increase the code metric value.

Method

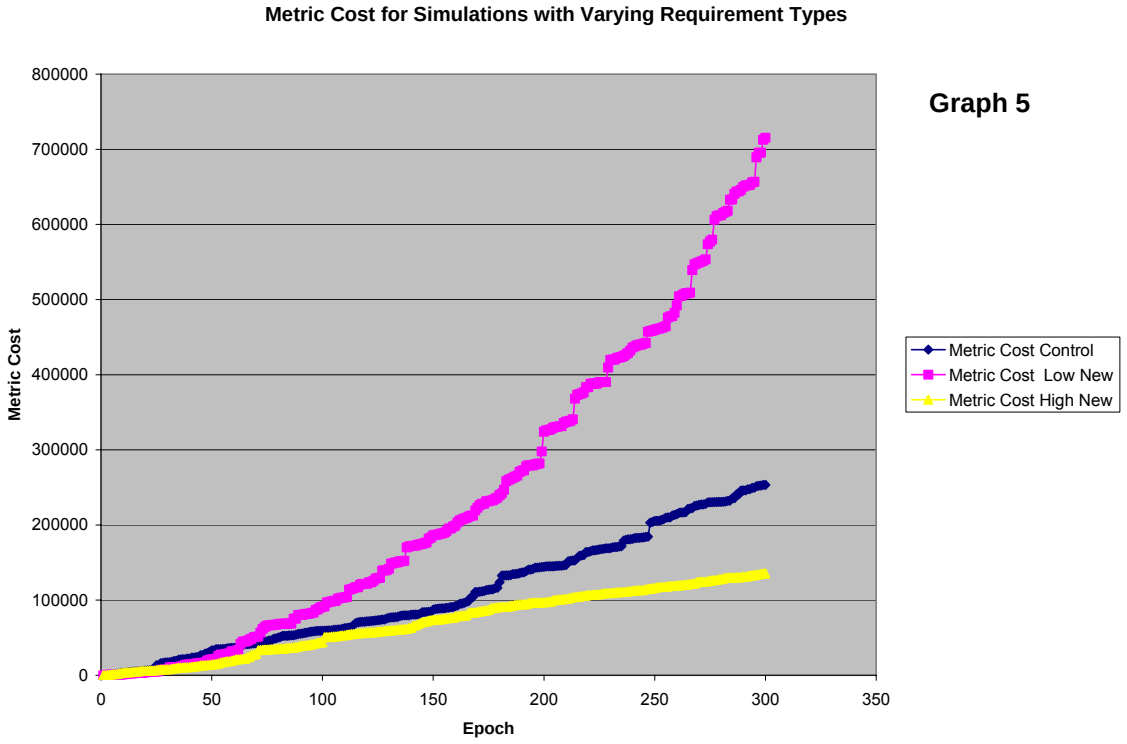
This proposal was validated by configuring an experiment with different distributions of requirement types. The experiment was run for 300 Epochs (requirement implementations) with the first 20 Epochs being exclusively 'New' requirements types as in previous experiments. The statistical distribution of each requirement type in each experiment is shown in the below table (configured in the Requirement Policy):

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

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

Results

The results, shown in Graph 5, demonstrate a distinct increase in curvature as the 'Change' and 'Augment' requirement predominate with the run that favours 'New' requirement types having the lowest metric cost increase. This validates the expected behaviour with the new requirement types creating a higher proportion of classes thus reducing the class density and hence the code metric value. Full results for this experiment are available in Appendix (C).



Graph 5

Experiment (3): Varying the Number of Agents

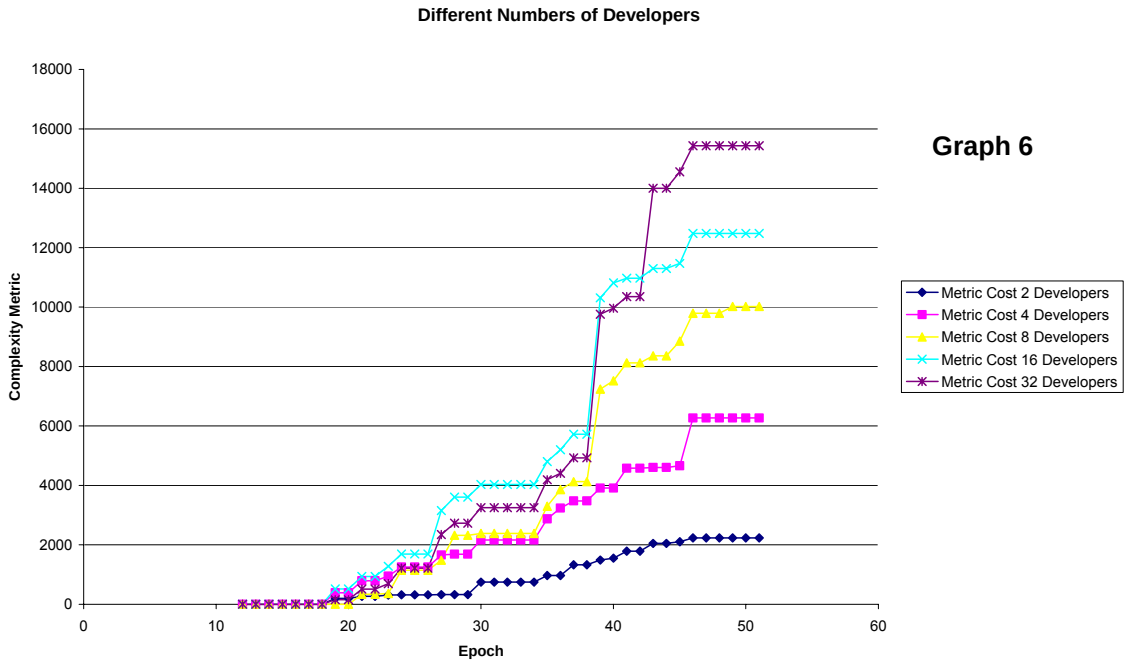
Aim

The simulation provides a facility for specifying the number of agents that contribute to the evolution of code. Each agent has a memory of all code they created (see section 5.6.1) and the default code metric takes this into account dropping the metric value immediately to zero if the agent created the code that is being measured.

These factors imply that the code metric value should evolve more slowly for low numbers of developers as they will each have been responsible for more code and hence have more memory coverage of the code base. The aim of this experiment is to determine if this assumption holds.

Method

The experiment was configured to measure this proposition over five different runs with five different numbers of agents contributing to evolution of the code base. Each experiment was run over 50 epochs with the results displayed in the Graph 6.



Graph 6

Results

The results show that development with two agents is most efficient and thirty is least. The complexity cost increases with the number of developers due to the increased cost associated with having to understand coded prior to changing it. When there are fewer developers this cost is lower as each developer was responsible for the original construction of a higher proportion of the code base and thus has less to learn. This validates the assumed behaviour stated above. In addition implementation cost stayed approximately constant in all experiments (see Appendix (D) for full results).

11.1 Analysis and Further Measurements

The results presented in this section provide confidence that the simulation performs in a manor that approximates the features of a real world application in the areas tested. Such a conclusion is corroborated by intuitive expectations as well as empirical results.

Confidence in the simulation results could be increased further through additional validation against other empirical sources as well as further experiments that investigate facets of the simulation not discussed in this section. Details of the unimplemented features are documented in section 12.

12 Other Framework Utilities

The framework provides a host of utilities that have not been included in the basic experiments performed here. These are summarised below:

Requirements Generator	The requirements generator has a variety of settings that have not been investigated fully. These are detailed in section 5.4.

Advancing the Evolution Policy	As previously discussed the evolution policy can be extended to include any number of extra features. Most notably the introduction of feedback from the code base to the evolution policy via the code metrics would facilitate the modelling of complex non-leaner behaviours.
Coupling Type	The coupling type is defined in the evolution policy but is not implemented in the default behaviour. This could be included in future experiments.
Task Size	Each task has an impact level or size associated with it. This could be used in the code metrics to provide a further level of granularity.
Abstraction Type and Task Type	The abstraction type is modelled in Tasks and Requirements and acts as a conceptual link between disparate code entities that represent similar concepts. The Task type denotes whether the requirements are Operations, Entities or Data Entities. These segregators are designed to be used to model Object Orientated concepts in the evolution policy and code metrics.
Leaking Memory	The ability for Memory to leak over a defined period of time is provided in the framework but not implemented in the default behaviour.
Global Variables	Global variables are currently not implemented in the default evolution policy. They can easily be added to the code base via the API.

13 Further Experiments

The aim of this framework is to allow experimentation and validation of evolutionary theories in software engineering. To this aim the simulation provides four basic elements, each of which can be varied independently to explore different experimental aims. These basic elements are Requirements, Evolution, Measurement and the Code Base (assumed to be constant) as shown in Figure (5).

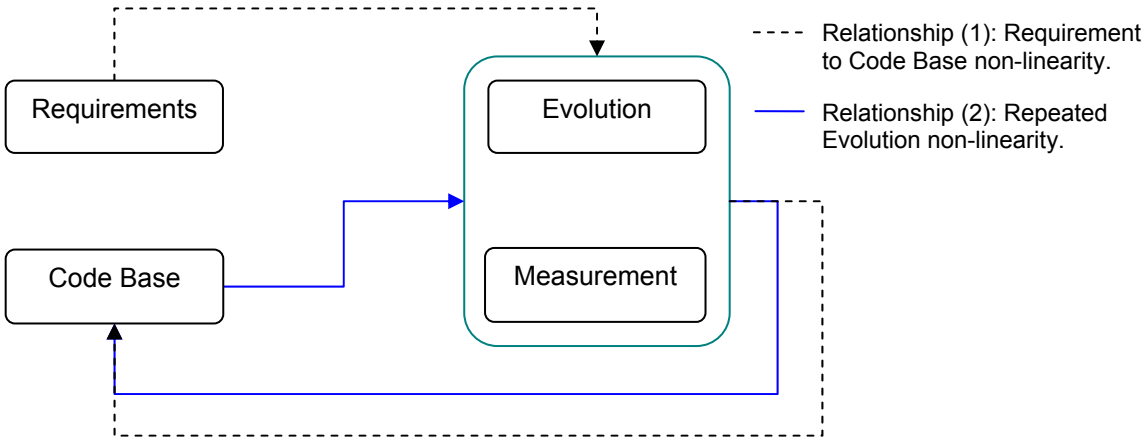


Figure 5: The two major configurable relationships in the system. The dashed arrows represent the non-linear relationship between requirements and the code base. The solid line represents the feedback circuit by which the evolution can alter based on the state of the code base itself.

Exploring the simulation process further involves investigation of both of the two major relationships that the framework exposes. The experiments presented in this dissertation only explore Relationship (1), the non-linearity between requirements and their resulting code

implementation. Relationship (2), as yet unexplored, results from the feedback loop that exists between the code base and the evolution policy via the code metrics. This loop feeds back information to the evolution policy regarding the structure of the code base at the time that the change is being made. This allows agents to react to the differently according to the state of the code structures being changed. This feedback loop is fundamental to the modelling of the underlying complex processes that are generally evolve in dynamic systems [For69].

As well as the relationships between the four major entities in the simulation, future experiments will explore evolution of software though one of two experimental methods. The first is the simplest and involves investigation of single evolutionary concepts in isolation or within different environmental setups with the repeated effects of application being measured. The second involves combining multiple concepts within the same simulation environment and examining the complex relationships that result from their interaction. In reality the second method will often evolve from the first. These two methods are discussed in the next sections.

13.1 Investigating Repeated Evolution of Single Evolution Concepts

Experiments of this type investigate single factors, applied repeatedly under different environmental conditions, extending the approach taken in this dissertation. For example an experiment might investigate the effect content coupling has on evolution over a number of different types of project i.e. small/large, high change / Greenfield etc.

Initially the simulation might proceed by adding the following topics to the model:

- **Adding Design Principals:** Adding new and validated design principals to the evolution policy is fundamental to making the simulation more realistic. Many principals could be introduced from basic ones such as simple reuse to more complex examples such as ‘encapsulate what varies’.
- **Adding Refactoring:** The introduction of refactoring methods to the evolution policy to be triggered by code base feedback.
- **Adding Coupling Types:** An investigation into how coupling types affect evolution both in terms of output and code base structure.
- **Adding Global Data:** The use of global data and its evolution.
- **Adding Inheritance:** The simulation already supports the concept of Abstraction Types in the Requirements section which would form a basis for modelling inheritance and other forms of abstraction.

More complete experiments could also be run to service more focussed goals. These might include:

- **Starting Structure:** An investigation of whether evolution is affected by the starting structure of the code base.
- **Iterative Development:** A study into the effects iterative development cycles have on a code base vs. more traditional, lengthy cycles.
- **The Effects of Componentisation on Evolution:** Experiments on this topic might address whether componentising a system as it grows makes it easier to extend and maintain?
- **The Effects of Inheritance:** Is inheritance a help or a hindrance? Do application structures really benefit from its existence? Does inheritance degrade an applications structure and if so what can be done to reduce this effect over the long term evolution of a system?
- **Separating Concerns in a System:** An investigation into the effects ‘Separating Concerns’ has 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? A worked example of this is presented in the next section (13.2).

- **Analysis of the Evolution of State:** State evolution is a topic that has gained little light in recent research when compared to its counterpart, behaviour. State adds complexity to the interaction of components at runtime. This is likely to have a detrimental effect to software as it evolves but there is little data on the evolution of this. Simulation would provide an ideal means to increase experimental data in this field as the runtime modification of state could be simulated in this framework with relative ease.
- **The Rules of OO Software Languages:** A more far reaching goal would be to use the simulation to adapt of the laws that bind the OO paradigm. Similar experiments could also investigate the effects Aspect Orientated Programming [Kicz97] has on the structure and evolution of code.

13.2 Separating GUI and Business Logic: A Thought Experiment

This section describes a brief "thought experiment" which explores how the simulation framework might be used to investigate the separation of GUI and business logic from both code structure and productivity standpoints? The stages of the experiment would be:

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 on based on Millers 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.

13.3 Investigating Multi-Faceted Evolution

The most interesting experimental results may arise through measuring the interaction of different features, such as those suggested in the previous section, on one another. Such interactions can quickly create complex non-linear responses which can only be observed either through simulation or empirical analysis and thus are a primary goal for any experiment in this area.

14 Further Work

Forester's analysis of social systems [For71] investigates various factors such as population trends and the quality of urban life etc. He utilises computational models developed in the field of system dynamics [SysD] with some interesting conclusions. In particular he found several cases where practitioners installed unsuccessful solutions problems they had. Analysis with a simulation model revealed that although a static analysis indicated the solution should be beneficial, when interacting with the surrounding system the solution actually made the situation worse. For example he notes:

"In many instances it emerges that it is the known policies applied to aide a system which actually causes the troubles." [For71]

Forrester is pointing at his observations that many of the processes put in place to help address various problems often end up making them worse. The real response of the 'remedy' is to worsen the 'condition'. This counter intuitive conclusion arises as the complexity of the system may act to mask the true action of an attribute. Whether similar situations exist in the field of

software evolution is unknown but the simulation of software evolution offers a unique opportunity to investigate whether or not such relationships do exist in our field.

Another approach and one 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 largely been pioneered by NASA who utilise their wealth of empirical data on previous software projects to fuel the software process simulations. Topics investigated include the defect detection efficiency of code inspections [Mun03]. This approach could be beneficial in the simulation of software evolution, applied as a form of decision support tool. Such a tool could be used to predict the effects that different environmental factors or coding practices have on a code base, allowing managers to tune their coding practices accordingly.

14.1 Improving the Simulation Program

Other than the various extensions to the simulation model a number of issues could be improved in the simulation program itself to make it easier to experiment with:

- A method for analysing the structure of the code base at a single point in time. This might allow the structure of the code base to be measured rather than the cost alone. This would likely take the form of a suite of custom metrics that would analyse the entire code base at set points in the experiment.
- Allow the definition of multiple plug-ins. This would be a relatively simple task and would allow different policies to be set up for different behaviours of the system. For example there might be different metrics plug-ins for different standard software metrics.
- Alteration of the stochastic elements so that they are configured via standard distributions. For example rather than specifying the Augmentation Percentage it would be better to provide a standard deviation for the distribution of Augmentations.
- Incorporate a dynamic class loader into the simulation so that plug-ins can be changed while the application is running without the need for an external device.

14.2 Alternative Simulation Techniques

The simulation model presented here is a custom implementation but there are a number of other frameworks and modelling tools available. Vensim [Vensim] provides a rigorous approach that should be considered where the extraction of mathematical relationships between simulated features is of primary interest. Vensim produces high quality dynamic feedback models with a host of additional mathematical tools for problems such as regression, optimisation etc. Another simulation framework is produced by the Arizona Centre for Integrative Modelling and Simulation named DevsJava [DevsJ]. These tools were not used in the context of this dissertation as they are geared towards heavier, more industrial simulation models and thus were not practical for this proposal given the time scale.

15 Reflections on the Simulation and Experiences Gained

This project has been an exploration into the simulation to software engineering taken from the perspective the code and how it changes. This section summarises the key principals that I have learnt in constructing the experimental framework.

- **Separating Requirements and Code**. Separating the requirements from the code base allows the requirements to exist as a separate entity that can be grown independently. This has the important side effect that they can be validated prior to being implemented as code. Requirements also play a pivotal role in the structuring of any code base. Keeping a clear separation between these two makes it easier to define relationships between them. It also allows the requirements to be held constant for experiments in which they are not involved.

- **Feedback**. Jay Forrester, the father figure of System Dynamics, proposes that feedback loops, formed from simple concepts, create the foundations of a large percentage of the complexity observed in dynamic systems [For69]. A vital attribute of the simulation is the feedback loop between the evolution policy and the code base. This provides a facility to model complex, non-linear behaviours which are generated from the interaction of simple concepts modelled in the system.
- **The Importance of Minimising the Complexity of the Model.** The model as a whole must be kept simple. If the model is too complex then it becomes difficult to distinguish the cause of different observed effects. This is particularly true if they are attributable to several of the variables of the system. This point of view is elaborated further in [Gilb00].
- **The Importance of Comparison as Apposed to Absolute Measurements**. It is hard to create a simulation that produces absolute results as each quantity must undergo careful and time consuming calibration. Instead results are best obtained by comparing the performance of one experiment with another (or a control).
- **The Importance of Validating each Added Element.** It is vital to validate all concepts thoroughly as invalid assumptions will combine and scale to produce results that may not be representative of real live behaviour.

16 Conclusions

The evolution of software, in particular its structural erosion over successive generations, represents one of the primary problems in software engineering today. Methods for combating such issues have been proposed on mass but few possess empirical substantiation. This dissertation presents a novel approach to the investigation of software evolution that could potentially aid these issues.

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 as well as opening the doors to investigations that might not be practical by direct empirical measurement.

From a theoretical standpoint the simulation can be used to investigate the interaction of forces in an attempt to determine the underlying principals of software evolution. As with the pendulum example, software becomes increasingly complex as the simple forces that shape it interact with one another. It is the interaction of these forces that make combating software degradation such a difficult problem. The simulation framework presented here allows the exploration of these issues in way that cannot be performed by other means.

However this dissertation is only intended as a proof of concept. The depth to which the simulation model has been calibrated is constrained by the time scales of the project and thus further investigation and optimisation of the framework would likely be needed before it could be used to generate any significant research data.

Simulation, as a mechanism for deriving rules of behaviour from complex systems, has a rich and longstanding background. The field of System Dynamics started applying computational simulations to the study of complex systems as far back as the 1960's and there is a rich set of research work spanning multiple disciplines from which knowledge can be leveraged. However Software Engineering is a young discipline and the laws of software evolution are still forming. Much as in other disciplines, simulation may provide a valuable window into a world otherwise inaccessible to current research, expediting the crystallisation laws as well as opening the doors to new insights.

References

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

[Cap04] “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

[DevsJ] http://www.acims.arizona.edu/SOFTWARE/software.shtml Site for DevsJava

[Eclip] http://www.eclipse.org Free development environment that includes a dynamic class loader.

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

[For71] Jay W. Forrester, "Counterintuitive Behaviour of Social Systems", Technology Review, Vol. 73, No. 3, Jan. 1971, pp. 52-68.

[For89] “The System Dynamics National Model: Macrobehavior from Microstructure”, Forrester, J. W. Computer-Based Management of Complex Systems: International System Dynamics Conference, ed. P. M. Milling & E. O. K. Zahn. Berlin: Springer-Verlag. 1989.

[Fow02] “Patterns of Enterprise Application Architecture”, Martin Fowler, Addison-Wesley Professional; 1st edition (November 5, 2002) p330

[Fow97] “UML Distilled: Applying the Standard Object Modelling Language”, Martin Fowler with Kendall Scott, Addison-Wesley Object Technology Series, 1997.

[Fow99] “Refactoring: Improving the Design of Existing Code” (Object Technology S.) Martin Fowler: Addison Wesley, 1999

[Gam95] “Design patterns : elements of reusable object-oriented software“: Erich Gamma, Richard Helm, Ralph Johnson, John Vissides Addison Wesley March 14, 1995

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

[Kelln91]“Software Process Modelling Support for Management Planning and Control”. Kellner, M Proceedings of the first international conference on the software process 1991.

[Kelln99] “Software Process Modelling and Simulation: Why, What, How”: Kellner, Madachy, and Raffo, Journal of Systems and Software, Vol. 46, No. 2/3 (15 April 1999)

[Kels04] “A Simple Static Model for Understanding the Dynamic Behaviour of Programs.” Kelsen: International Workshop on Program Comprehension 2004

[Kem99] “An Empirical Approach to Studying Software Evolution”: Kemerer, IEEE Transactions on Software Engineering 1999

[Kicz97] “Aspect Orientated Programming”: Kiczales, Lamping, Menheker, Maeda, Lopes, Loingteir, Irwin: European Conference on Object-Orientated Programming (ECOOP 97)

[Leh80] "Program, Life Cycle and the Law of Program Evolution", M. Lehman, Proceedings of the IEEE, 68, 1060-1078, 1980

[Med70] “Dynamics of Commodity Production Cycles”. Meadows, D. L. Cambridge MA: Productivity Press. 1970.

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

[Mil56] “The magical number seven, plus or minus two: Some limits on our capacity for processing information”, G. A. Miller: Psychological Review, 63:81-97, March 1956.

[Mun03] “Using Empirical Knowledge from Replicated Experiments for Software Process Simulation: A Practical Example”. J. Munch, O. Armbrust: Proceedings of the 2003 International Symposium on Empirical Software Engineering (ISESE’03) 0-7695-2002-2/03

[Par72] “Information distribution aspects of design methodology”. D.L. Parnas. IFIP Congress (1) 1971: 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

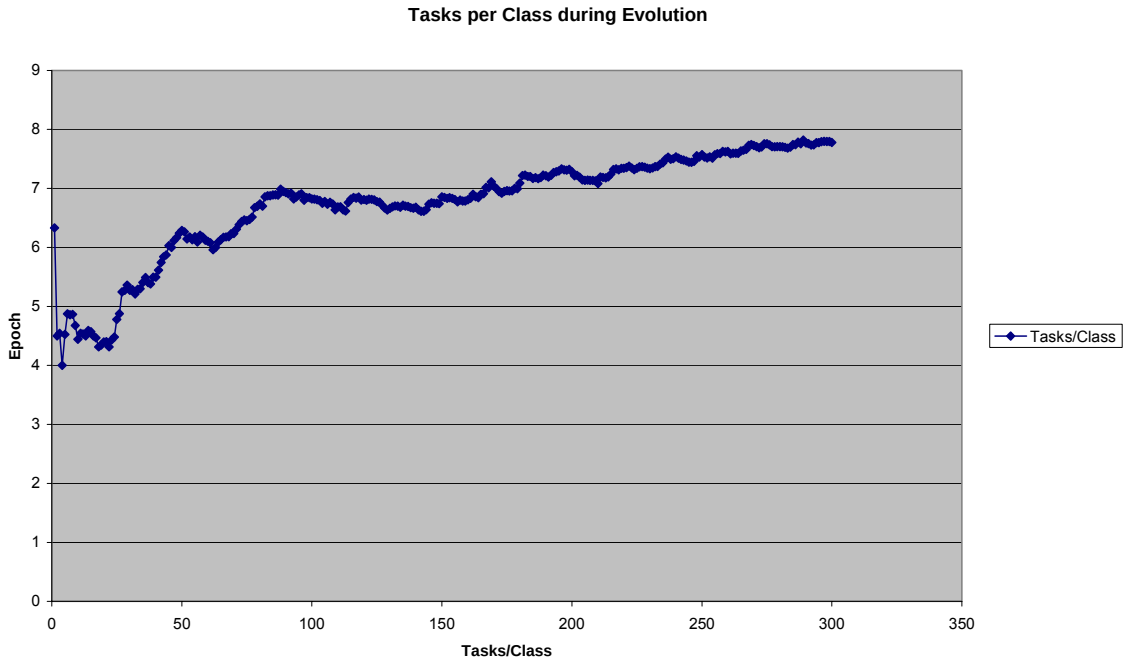
[Ster00] “Business Dynamics: Systems, Thinking and Modelling for a Complex World” John Sterman, (Irwin/McGraw-Hill,2000)

[SysD] http://www.systemdynamics.org The home page for the Systems Dynamics Group at MIT.

[Vensim] http://www.vensim.com/software.html

[Wern99] “Software Process White Box Modelling for FEAST/1”, Wernick and Lehman: Journal of Software Systems 1999

Appendix A: The Evolution of Tasks per Class



Tasks per Class over a run of 300 epochs with standard settings (20 new requirements enforced initially). This can be compared with the results collected analysing the ARLA system [Calp4] reproduced below (the lower plot). Both show approximately linear growth with a slight positive gradient.

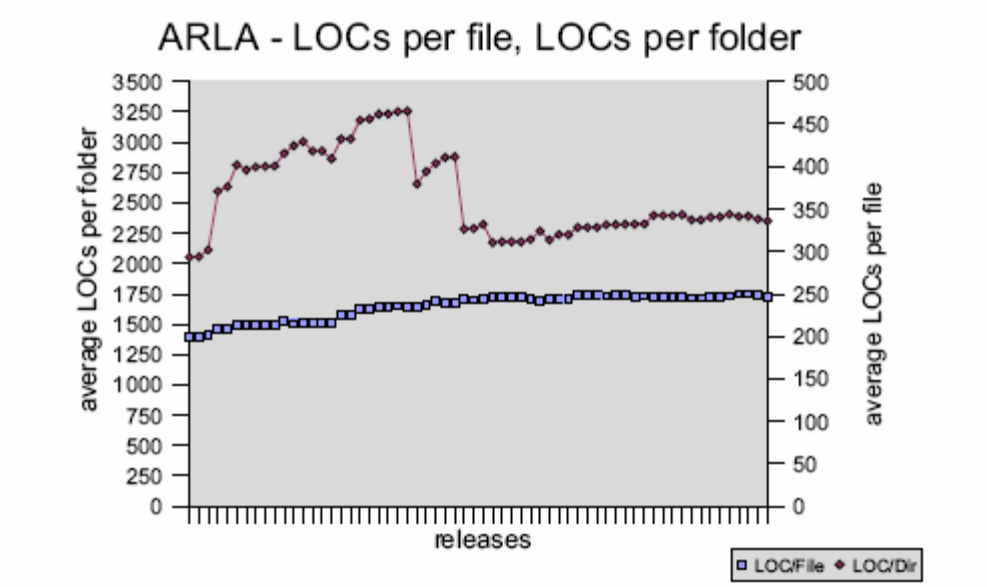
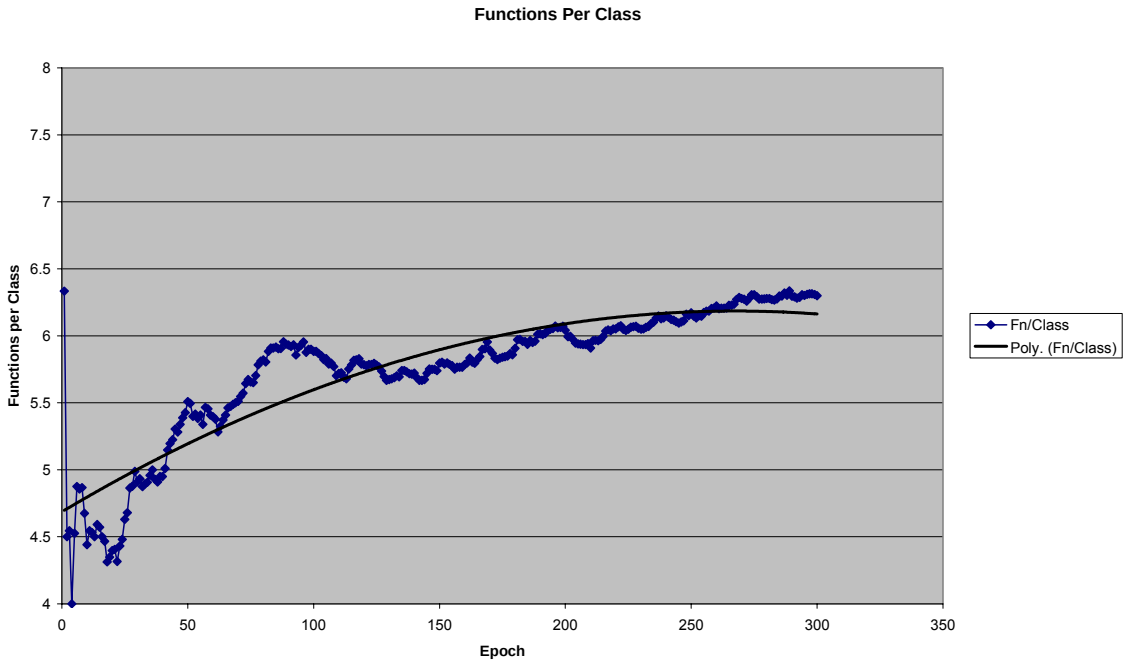
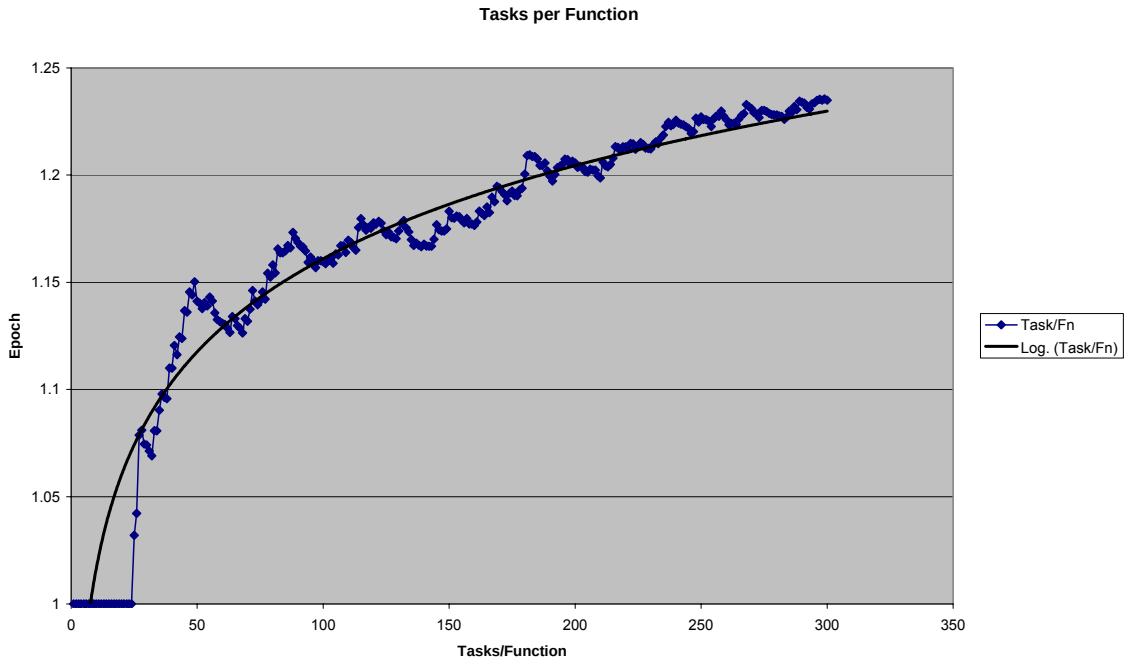


Figure 7 - Average size in LOC for folders (Y axis) and files (Y secondary axis)

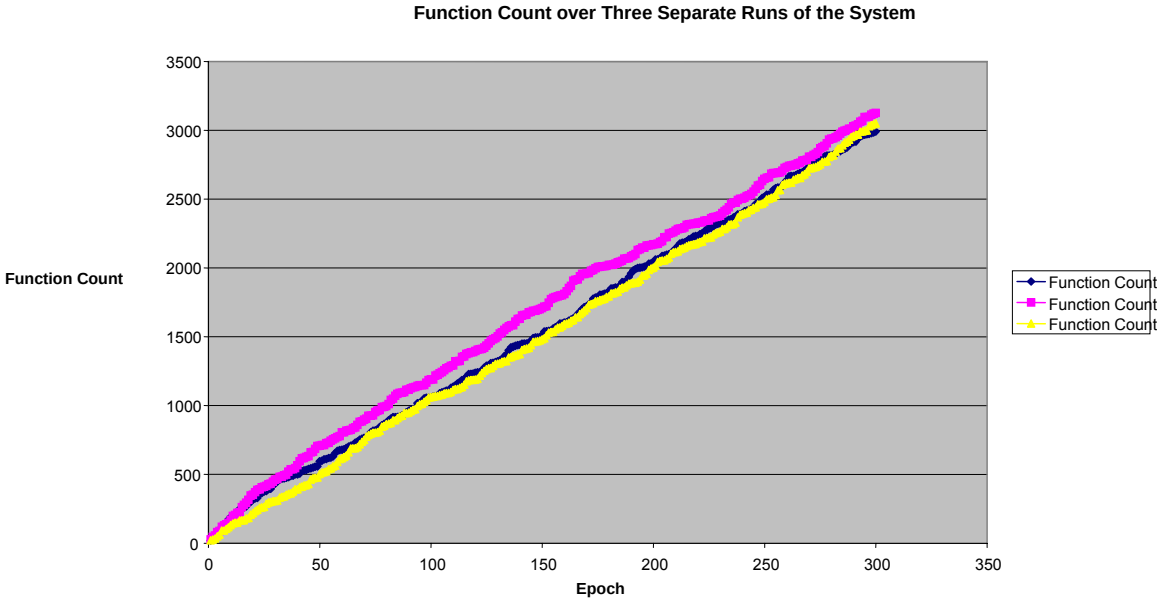
Other data included in this experiment was the number of functions per class and the number of tasks per function. These are shown below.

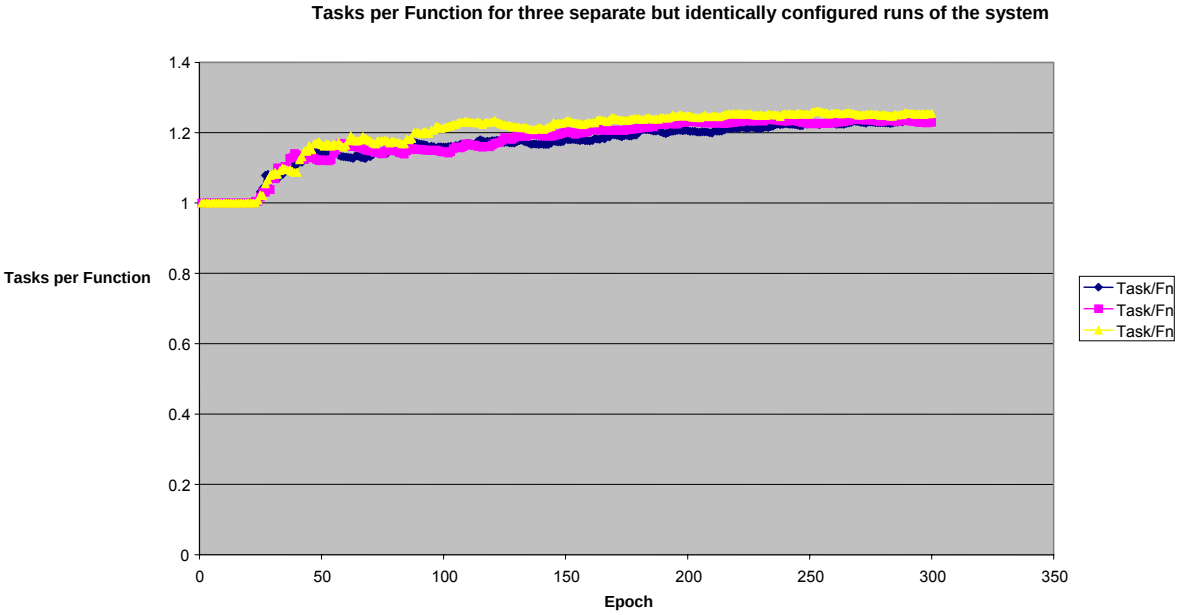


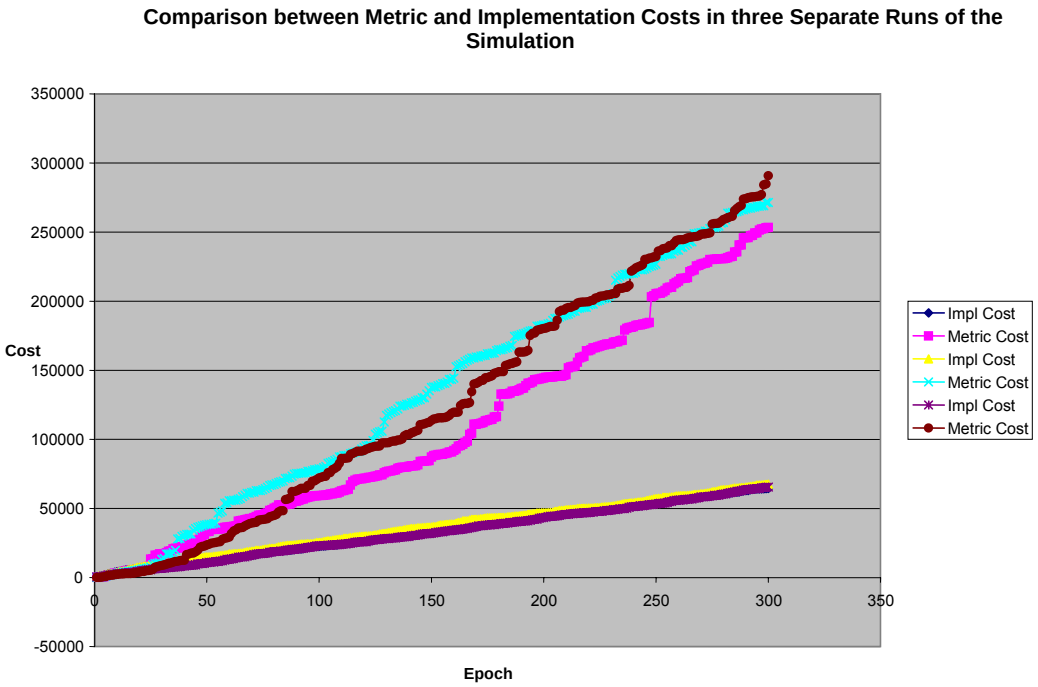

Tasks per Function
1.25
1.2
1.15
1.1
1.05
1
Epoch
0
50
100
150
200
250
300
350
Tasks/Function
Task/Fn
Log. (Task/Fn)

Appendix B: Variance over Separate Runs

Variance in system output for various measures over three extended runs

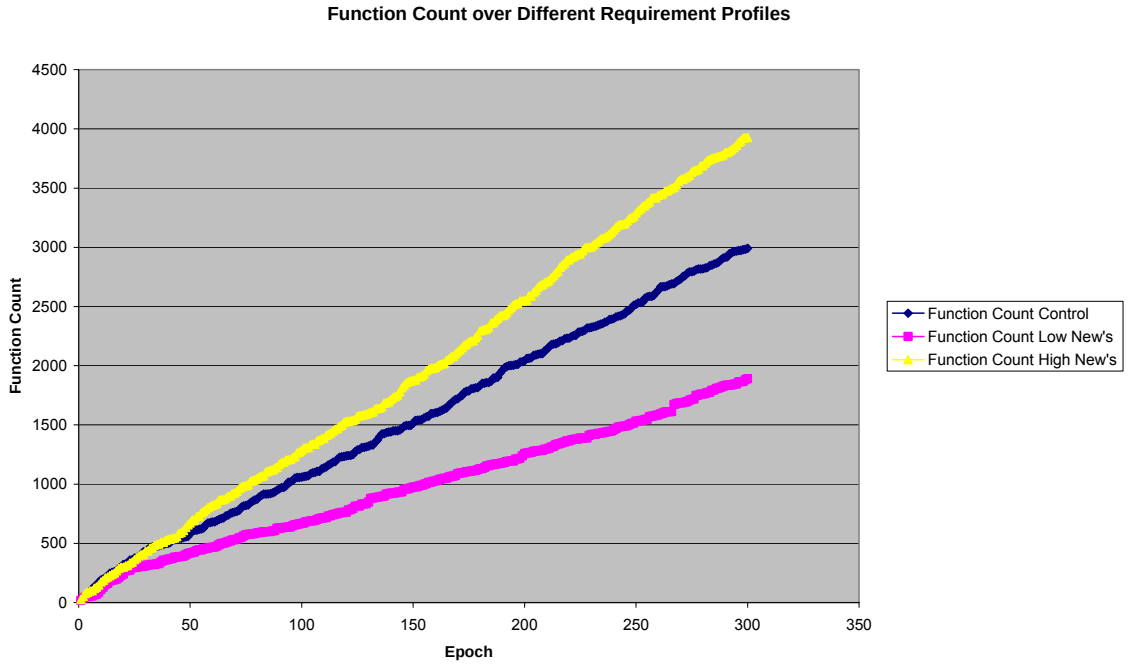




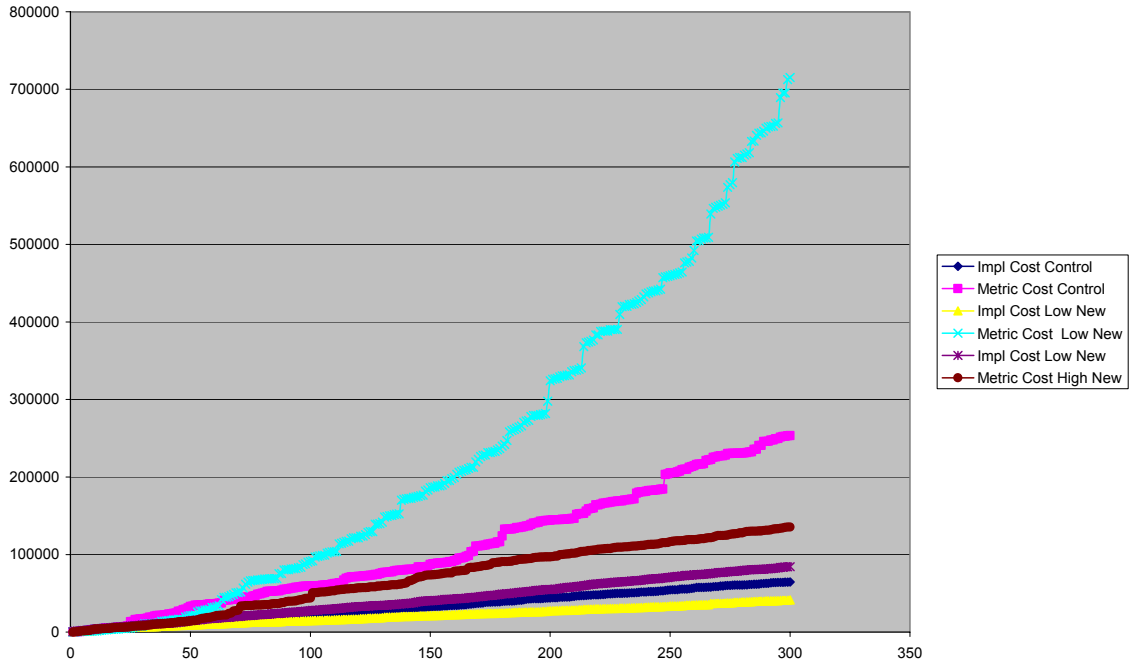

Comparison between Metric and Implementation Costs in three Separate Runs of the Simulation
350000
300000
250000
200000
150000
100000
50000
0
-50000
Cost
0
50
100
150
200
250
300
350
Epoch
Impl Cost
Metric Cost
Impl Cost
Metric Cost
Impl Cost
Metric Cost

Appendix C: Evolution with Different Requirement Profiles

Three experimental runs with different requirements type profiles are shown below for different measures. The control run is performed with standard parameters. Low New's refers to the fact that the Requirements had a disproportionately low proportion of 'New' Requirement Types in the requirements. Conversely High New's corresponds to a disproportionately high number of 'New' Requirement Types used in the requirements generation process.



Implementation and Metric (Complexity) cost over different requirement type profiles.

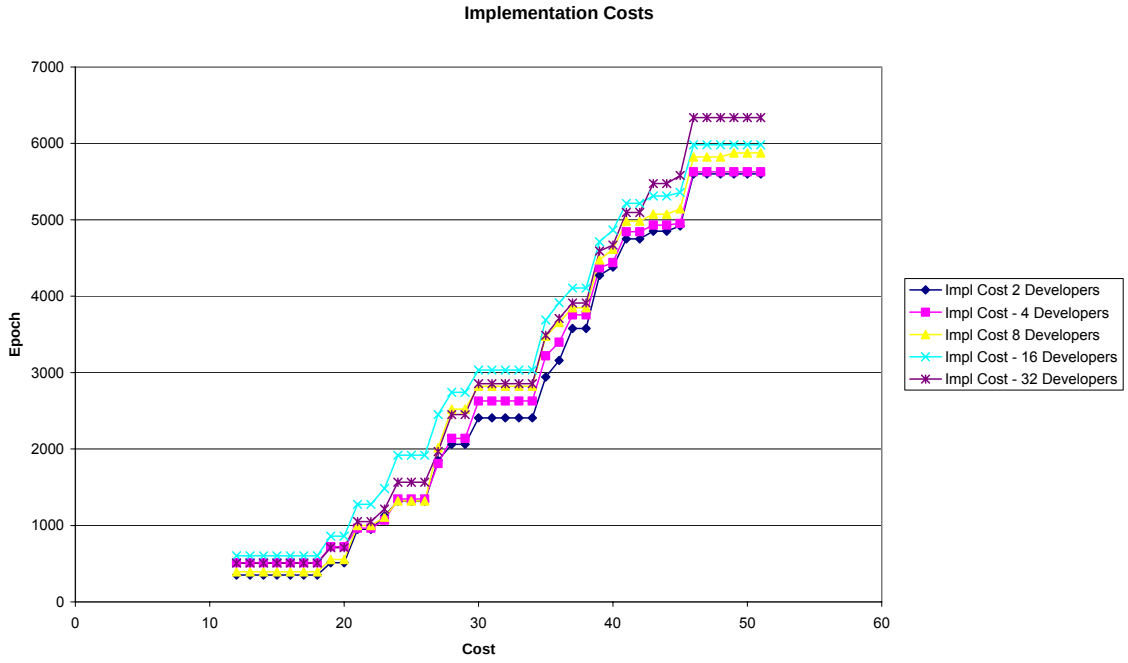


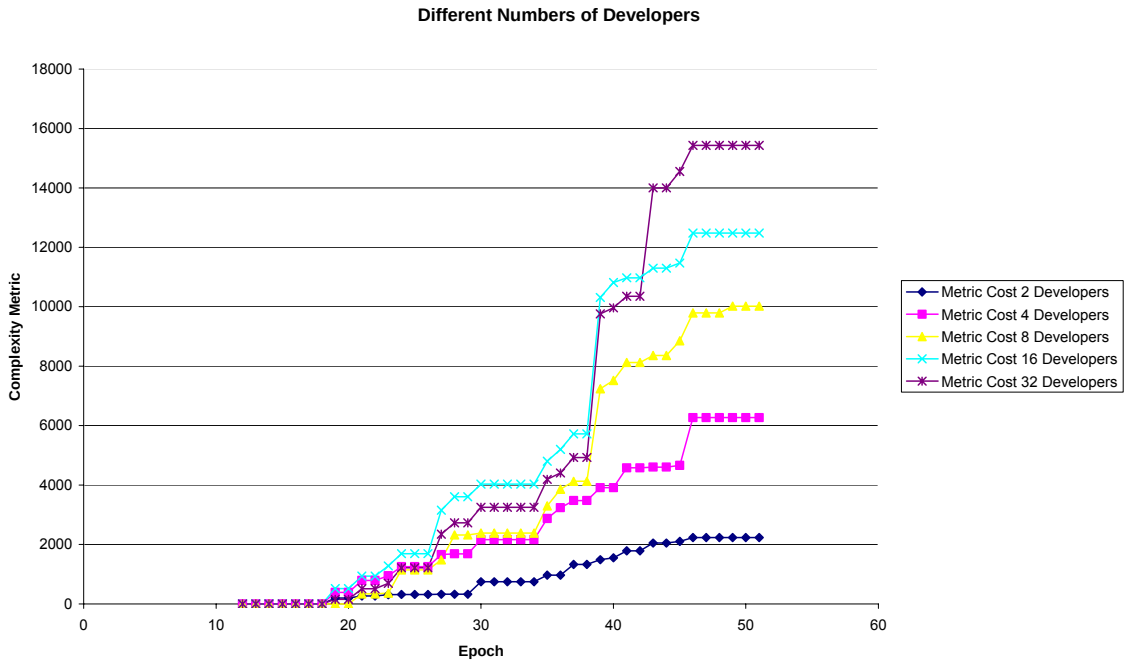
Appendix D: Evolution with Different Numbers of Agents

This section presents results that document the affect that different numbers of Agents (developers) have on the cost incurred in implementing a system. Two plots shown here are for Implementation cost and Complexity cost.

Implementation cost is approximately constant between experimental runs as the number of developers has no effect on how long it takes to develop the required software in this version of the simulation.

However the second plot shows the complexity cost which increases with the number of developers. This occurs due to the increased cost associated with having to understand coded prior to changing it. When there are fewer developers this cost is lower as each developer was responsible for the original construction of a higher proportion of the code base and thus has less to learn.



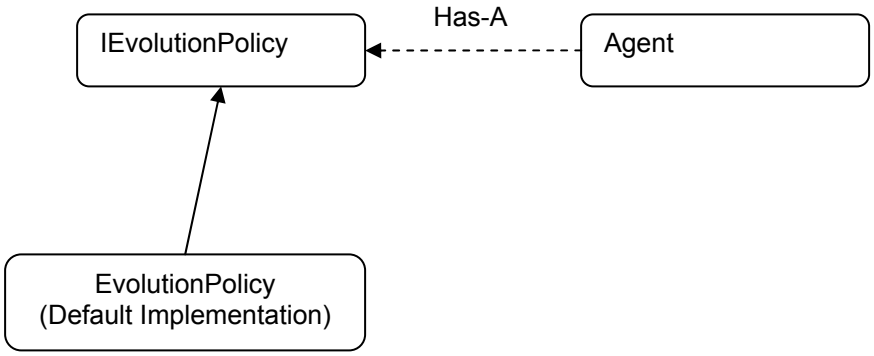

Different Numbers of Developers
Complexity Metric
Epoch
0
2000
4000
6000
8000
10000
12000
14000
16000
18000
0
10
20
30
40
50
60
Metric Cost 2 Developers
Metric Cost 4 Developers
Metric Cost 8 Developers
Metric Cost 16 Developers
Metric Cost 32 Developers

Appendix E: Sample Code

This section contains sample code for three important classes in the code base:

- The Evolution Policy Interface
- The Evolution Policy
- The Agent

The entire code base is available for download at
http://www.benstopford.com/devsim/devsim.shtml



```
/*
 * Created on 27-Aug-2005
 */
package com.devsim.evolution;

import com.devsim.code.CodeConstruct;
import com.devsim.code.CouplingType;
import com.devsim.code.Function;
import com.devsim.requirements.dataobjects.Task;

/**
 * The Evolution Policy is the plug-in that determines
 * how the code base evolves (along with the Complexity Injector).
 * This is the fundamental place where experiments are configured.
 * The evolution policy has access to a variety of attributes of
 * the simulation that can be used to alter the way that the
 * simulation evolves.
 *
 * In each case the evolution policy implementation must be responsible
 * for the turning of
 *
 * @author Ben
 *
 */
public interface IEvolutionPolicy {
	/**
	 * Create the code associated with the new task passed which is of
	 * requirement type NEW
	 *
	 * @param startingFunction - the function to start processing from
	 * @param task - the task to process
	 */
	public Cost processNewTask(Function startingFunction, Task task);

	/**
	 * Augment the base function with the new task of requrement type
	 * AUGMENT
	 *
	 * @param startingFunction  - the function to start processing from
	 * @param task - the task to process
	 */
	public Cost processAugmentation(Function startingFunction, Task task);

	/**
	 * Change code from an exising Task as specified in the new Task
	 * that is of requirement type CHANGE
	 *
	 * @param startingFunction - the function to start processing from
	 * @param newTask - the task to process
	 */
	public Cost processChange(Function startingFunction, Task newTask);

	/**
	 * Determine the coupling type that should be used for this caller and provider
	 * @param caller
	 * @param provider
	 * @return
	 */
	public CouplingType getCouplingType(CodeConstruct caller, CodeConstruct provider);

}
```

```
/*
 * Created on Aug 12, 2005
 */
package com.devsim.plugins;

import com.devsim.code.Class;
import com.devsim.code.CodeBase;
import com.devsim.code.CodeBaseAPI;
import com.devsim.code.CodeConstruct;
import com.devsim.code.CouplingType;
import com.devsim.code.Function;
import com.devsim.code.Property;
import com.devsim.evolution.Cost;
import com.devsim.evolution.IEvolutionPolicy;
import com.devsim.evolution.Memory;
import com.devsim.requirements.dataobjects.RequirementType;
import com.devsim.requirements.dataobjects.Task;
import com.devsim.utils.RandomGen;

/**
 * The Evolution Policy is the plug-in that determines
 * how the code base evolves (along with the Complexity Injector).
 * This is the fundamental place where experiments are configured.
 * The evolution policy has access to a variety of attributes of
 * the simulation that can be used to alter the way that the
 * simulation evolves.
 *
 * @author Benjamin Stopford
 */
public class EvolutionPolicy implements IEvolutionPolicy {
	private Memory _agentMemory;

	/**
	 * Constructor with memory from agent instance that is
	 * using this evolution policy
	 * @param agentMemory
	 */
	EvolutionPolicy(Memory agentMemory){
		_agentMemory = agentMemory;
	}

	/* (non-Javadoc)
	 * @see com.devsim.evolution.IEvolutionPolicy#processNewTask(com.devsim.code.Function,
com.devsim.requirements.dataobjects.Task)
	 */
	public Cost processNewTask(final Function startingFunction, final  Task task) {
		RequirementType newReqType;
		newReqType = task.getRequirementType();
		if(newReqType.isEntity() || newReqType.isDataEntity()){
			//New enties result in new classes being created
			final Class c = getAPI().createClass(task, startingFunction);

			//Create refeerences to the new class
			RandomGen.performRandomTimes(new RandomGen.Repeater(){
				public void run() {

	getAPI().createReference(startingFunction,(Function)RandomGen.getRandom(c.getFunctions()));
				}
			}
			,c.getFunctions().size());
		}
		else if(newReqType.isOperation()){

			//Create average of three functions in the existing class
			//Each new class with have a property connected to a couple of other functions in the class
			RandomGen.performRandomTimes(new RandomGen.Repeater(){
				public void run() {
					addFunctionToClassWithRefAndProp(startingFunction, task,getAPI());
				}
			}
```

```
                        ,6);
                }
                else{
                        throw new RuntimeException("This should never happen!?!?!");
                }

                return CodeMetrics.calculate(startingFunction,_agentMemory);
        }

        /* (non-Javadoc)
         * @see com.devsim.evolution.IEvolutionPolicy#processAugmentation(com.devsim.code.Function,
com.devsim.requirements.dataobjects.Task)
         */
        public Cost processAugmentation(final Function baseFunction, final Task task){

                final Class baseClass = baseFunction.getClazz();
                RequirementType newReqType;
                newReqType = task.getRequirementType();
                if(newReqType.isEntity() || newReqType.isDataEntity()){
                        //Add the new entity class to the to the code base
                        final Class entityClass = getAPI().createClass(task, baseFunction);

                        //Create average of 1 EXTRA function to the base class
                        RandomGen.performRandomTimes(new RandomGen.Repeater(){
                                public void run() {
                                        Function f = getAPI().createFunction(task,baseClass, baseFunction);
                                        getAPI().createReference(baseFunction,f);
                                }
                        }
                        ,2);

                        //Randomly add 0-2 references between base and enitty classes
                        RandomGen.performRandomTimes(new RandomGen.Repeater(){
                                public void run() {
                                        Function randomBaseClassFunction =
(Function)RandomGen.getRandom(baseClass.getFunctions());
                                        Function randomEntityClassFunction =
(Function)RandomGen.getRandom(entityClass.getFunctions());

        if(randomBaseClassFunction!=null&&randomEntityClassFunction!=null)

        getAPI().createReference(randomBaseClassFunction,randomEntityClassFunction);
                                        }
                        }
                        ,2);
                }
                else if(newReqType.isOperation()){
                        //Create average of 1 EXTRA function to the base class
                        RandomGen.performRandomTimes(new RandomGen.Repeater(){
                                public void run() {
                                        Function f = getAPI().createFunction(task,baseClass, baseFunction);
                                        getAPI().createReference(baseFunction,f);
                                }
                        }
                        ,2);

                        //Create 0-2 extra references from the base class to randomly selected functions
                        RandomGen.performRandomTimes(new RandomGen.Repeater(){
                                public void run() {
                                        Function f = getAPI().createFunction(task,baseClass, baseFunction);

        getAPI().createReference(baseFunction,getAPI().getFunctionAtRandom());
                                        }
                        }
                        ,2);

                }
                else{
                        throw new RuntimeException("This should never happen!?!?!");
                }
```

```
                return CodeMetrics.calculate(baseFunction,_agentMemory);
        }

        /* (non-Javadoc)
         * @see com.devsim.evolution.IEvolutionPolicy#processChange(com.devsim.code.Function,
com.devsim.requirements.dataobjects.Task)
         */
        public Cost processChange(Function funcToExtend, Task newTask) {
                Cost cost = new Cost();

                //add the new task to all functions that use the current one
                funcToExtend.addTask(newTask);

                //add a new function with properties linkign them back to original fucntion
                addFunctionToClassWithRefAndProp(funcToExtend,newTask,getAPI());

                //half the time add a new ref
                if(RandomGen.getBool()){
                        getAPI().createReference(funcToExtend,getAPI().getFunctionAtRandom());
                }
                cost.add(CodeMetrics.calculate(funcToExtend,_agentMemory));

                return cost;
        }

        /**
         * @return Returns the _api.
         */
        CodeBaseAPI getAPI() {
                return CodeBase.getAPI();
        }

        /**
         * Adds a new function with a reference and
         * properteis from the creating function
         * @param startingFunction
         * @param task
         */
        static Function addFunctionToClassWithRefAndProp(Function startingFunction, Task task,CodeBaseAPI api) {
                Class startingClass = startingFunction.getClazz();
                Function f = api.createFunction(task,startingClass, startingFunction);
                api.createReference(startingFunction,f);

                //create a property
                Property p = api.createProperty(api.getClassAtRandom(),startingClass,task);
                api.createReference(f,p);

                //link the new property to the new function and 2 other random functions
                Function f1 = (Function)RandomGen.getRandom(startingClass.getFunctions());
                Function f2 = (Function)RandomGen.getRandom(startingClass.getFunctions());
                api.createReference(f1,p);
                api.createReference(f2,p);

                return f;
        }

        /* (non-Javadoc)
         * @see com.devsim.evolution.IEvolutionPolicy#getCouplingType(com.devsim.code.CodeConstruct,
com.devsim.code.CodeConstruct)
         */
        public CouplingType getCouplingType(CodeConstruct caller, CodeConstruct provider){
                return EnvironmentVariable.getCouplingType(caller,provider);
        }

}
```

```
package com.devsim.evolution;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.logging.Logger;

import com.devsim.code.Class;
import com.devsim.code.CodeBase;
import com.devsim.code.Event;
import com.devsim.code.Function;
import com.devsim.plugins.PolicyFactory;
import com.devsim.requirements.RequirementRepository;
import com.devsim.requirements.dataobjects.Extension;
import com.devsim.requirements.dataobjects.Requirement;
import com.devsim.requirements.dataobjects.Task;
import com.devsim.utils.RandomGen;

/**
 *
 * The agent facilitates the evolution performed by the Evolution Policy.
 * Primarily it determines the starting class and function from the Task
 * and mediated the appropriate executrions of the evolution policy.
 *
 * @author Benjamin Stopford
 *
 * TODO To change the template for this generated type comment go to
 *
 */
public class Agent extends AgentController {
	private static final Logger LOGGER = Logger.getLogger("com.devsim.evolution");
	private Memory agentMemory = new Memory(this);
	private final IEvolutionPolicy policy = PolicyFactory.getEvolutionPolicy(agentMemory);
	private static int maxid;
	private int id;

	//AGENT CAN ONLY BE CONSTUCTED AT PACKAGE LEVEL (I.E. BY THE FACTORY)
	Agent (){
		maxid++;
		id = maxid;
	}

	public String toString(){
		return "Agent:"+id;
	}

	/* (non-Javadoc)
	 * @see
com.devsim.evolution.EvolutionTemplate#comprehend(com.devsim.requirements.dataobjects.Requirement)
	 */
	protected Cost comprehend(Requirement r) {
		//dependent on the task size, the number of existing functions,
		//their complexity and whether the agent has seen it before.
		return Cost.NO_COST;
	}

	/* (non-Javadoc)
	 * @see com.devsim.evolution.EvolutionTemplate#design(com.devsim.requirements.dataobjects.Requirement)
	 */
	protected Cost design(Requirement r) {
		//opertunity to switch evolutoin policy based on code base
		return Cost.NO_COST;
	}

	/* (non-Javadoc)
	 * @see
com.devsim.evolution.EvolutionTemplate#implement(com.devsim.requirements.dataobjects.Requirement)
	 */
	protected Cost implement(Requirement newReq) {
```

```
		getApi().startCosting(this);
		Cost totalCost = new Cost();

		List taskList = newReq.getTasks();
		Extension unitToExtend = newReq.getOperatesOn();
		RequirementRepository reqRepo = RequirementRepository.getInstance();
		Function startingFunction=null;

		//if it is a new event then add it in a new class with a new function
		if(unitToExtend.getRequirementType().isSystemEvent()){
			//startingClass = ;
			Event ev = getApi().createEvent(unitToExtend.getTask());
			startingFunction = ev;

		}else{
			//operating on an exisiting task so check it exists
			if(!reqRepo.exists(unitToExtend.getParent())){
				throw new RuntimeException("The task that is being changed has not been
coded yet. Extendion:"+unitToExtend.toString());
			}
		}

		if(newReq.getChangeType().isNew()){

			// New functionality may have a specified starting Task or Requirement.
			// New functionality however is only only ever started from one class
			// chosen at random if the task or requirement that is being extended
			// has several to choose from.
			if(unitToExtend.isRequirement()&&startingFunction==null){
				Requirement r = unitToExtend.getRequirement();
				Task firstTask = (Task)r.getTasks().iterator().next();
				//startingClass = getApi().getFirstClassForTask(firstTask);
				startingFunction =
(Function)RandomGen.getRandom(getApi().getFunctionsForTask(firstTask));
			}
			else if(unitToExtend.isTask()&& startingFunction==null){
				Task t =unitToExtend.getTask();
				Class startingClass = getApi().getFirstClassForTask(t);
				startingFunction = getApi().getFirstFunctionForClass(startingClass);
			}
			if (startingFunction==null){
				throw new RuntimeException("The starting function needs to have been found by
this point.");
			}
			Iterator tasks = taskList.iterator();
			while(tasks.hasNext()){
				Task task = (Task)tasks.next();
				totalCost.add(policy.processNewTask(startingFunction, task));
			}
		}
		else if(newReq.getChangeType().isAugment()){

			Iterator tasksToAugment;
			Iterator tasksToImplement;

			// AUGMENT functionality will always be changing a Requirement.
			if(unitToExtend.isRequirement()){
				Requirement reqToAugment = unitToExtend.getRequirement();
				if (reqToAugment==null){
					throw new RuntimeException("Requirement is null but needs to be
supplied for an augmentation.");
				}

				//Augmentation operates on all the requirements
				tasksToAugment = reqToAugment.getTasks().iterator();
			}
			else{
				Task task = unitToExtend.getTask();
				if (task==null){
```

```
                                        throw new RuntimeException("Task is null but needs to be supplied for
an augmentation.");
                                }

                                //Augmentation operates on all the requirements
                                List tasksToAugmentList = new ArrayList();
                                tasksToAugmentList.add(task);
                                tasksToAugment = tasksToAugmentList.iterator();
                        }
                        tasksToImplement = newReq.getTasks().iterator();

                        //loop through each existing task that must be augmented
                        while (tasksToAugment.hasNext()){
                                Task taskToAugment = (Task)tasksToAugment.next();
                                Iterator functionsToAugment =
getApi().getFunctionsForTask(taskToAugment).iterator();

                                //loop through each task that must be implemented as new functionality
                                while (tasksToImplement.hasNext()){
                                        Task taskToImplement = (Task)tasksToImplement.next();
                                        if(RandomGen.getPercentage() <
newReq.getAugmentationPercentage()){

                                                //loop through each of the functions for the top level task that
is being augmented
                                                while(functionsToAugment.hasNext()){
                                                        Function f = (Function)functionsToAugment.next();
                                                        //proceed with augmenting the class

        totalCost.add(policy.processAugmentation(f,taskToImplement));
                                                }
                                        }
                                }
                        }
                }
                else if(newReq.getChangeType().isChange()){

                        Iterator tasks = newReq.getTasks().iterator();

                        while(tasks.hasNext()){
                                Task newTask = (Task)tasks.next();
                                //this will only be a change to a task
                                Task taskToExtend = unitToExtend.getTask();

                                Iterator functions =
CodeBase.getAPI().getFunctionsForTask(taskToExtend).iterator();
                                Function f=null;
                                while(functions.hasNext()){
                                        f = (Function)functions.next();
                                        totalCost.add(policy.processChange(f, newTask));
                                }
                        }
                }
                totalCost.add(getApi().getCost());
                return totalCost;
        }

        /* (non-Javadoc)
         * @see com.devsim.evolution.EvolutionTemplate#test(com.devsim.requirements.dataobjects.Requirement)
         */
        protected Cost test(Requirement r) {
                //assume testing is proportional to the number of functions that were changed


                return Cost.NO_COST;
        }

        /* (non-Javadoc)
         * @see com.devsim.evolution.EvolutionTemplate#complete()
         */
```

```
	protected void complete(Requirement r) {

		//all the functions created as part of the
		//tasks implemented by this agent in this epoch
		//are added to the agents memory

		Iterator tasks = r.getTasks().iterator();
		while (tasks.hasNext()){
			Task t = (Task)tasks.next();
			Iterator functions = getApi().getFunctionsForTask(t).iterator();
			while (functions.hasNext()){
				Function f = (Function)functions.next();
				agentMemory.add(f);

			}
		}
	}

	public Memory getMemory() {
		return agentMemory;
	}
}
```