A NEW SOFTWARE PROCESS MODEL DESIGNED FROM THE BASICS OF EVOLUTIONARY BIOLOGY AND SOFTWARE EVOLUTION

By

MURUGAPPAN RAMANATHAN

Master of Science in Computer Science

Oklahoma State University

Stillwater, Oklahoma

2007

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2007

A NEW SOFTWARE PROCESS MODEL DESIGNED

FROM THE BASICS OF EVOLUTIONARY BIOLOGY

AND SOFTWARE EVOLUTION

Thesis Approved:

Dr. Johnson Thomas

Thesis Adviser

Dr.Venkatesh Sarangan

Dr.Nophill Park

Dr. A. Gordon Emslie

Dean of the Graduate College

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

Software engineering can be defined as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software. This process of development of software is achieved by using different software life cycle models to design, code and test the software. The main purpose of the software process model is to create reliable and security oriented software.

The software process model consists of several steps like collection of requirements, designing the architecture, coding, testing and maintenance. Several process models like the water fall model, spiral model and prototyping are used by companies to create the software. But most of these models were designed for a single generation of software. This is a major drawback because most of the software's today have started to have several versions and generations and the present models do not support evolutionary software. For example software created today can be designed from different models like waterfall or spiral, written in different languages like Java or VC++; it can run on Windows or a Linux system and depends on the varied hardware environment it runs on. With so many varied methods of development the idea of software integration and quality in the software generations has become a major concern.

Some of the problems in the existing methods are that they are not designed to produce generations of software. Also a single model cannot be used by different types of companies as some are non iterative like the waterfall model and some are more ad hoc like the extreme programming method. Researchers such as Dr. MM Lehman have started to look into the degrading quality in software and have postulated the eight laws of software evolution [1] to help companies to understand the importance of evolution in software. Several challenges [2] have been found in the creation of evolutionary software such as the models used, the human resource involved, and the support available to the end user. From these challenges several new methods have been designed like the staged model and the PSPM model. But these models give more importance to the maintenance phase rather than the whole software life cycle.

In this paper, we propose solutions to problems in existing models by applying some of the principles of evolution in biology and biochemistry to software, and an abstract model has been generated. It is also a unification model of all the existing models and evolutionary principles. The basic building blocks in biology are the DNA, genotypes, phenotypes and enzymes. By altering these basic properties by the methods of mutation and selection, nature is able to create evolution in organisms. These basic principles of evolution were incorporated into the varying steps in the process model to generate an evolutionary process model. The model is called the Infinity Model. It is named so because its basic structure is based on the infinity symbol and it signifies the continuous iteration of software for several generations. It consists of a completely new design cycle with the importance given to both the creation of software and maintaining

the software. The main advantage of this model is that it is designed for evolutionary software. In this model, methods to correct the problems in the existing models like resource allocation, documentation and requirement updating have been incorporated. Moreover several case studies of large company software and the problems they faced were studied. From the case studies several methods like requirement evolution, consolidation and architectural evolution have been incorporated into the Infinity Model.

In the next chapter, the reasons to improve the software process model are assessed in detail. We then look into the various similarities between software and biology and the various levels at which they can be compared. In Chapter three the drawbacks in the existing software models are viewed and also the necessary improvements are studied. In Chapter four the Infinity Model is proposed and the different steps in the life cycle are looked at. In chapter five the first case study of the various software models available today are studied as well as the disadvantages in these models. The ideas and principles behind these models are explored. In the penultimate chapter case studies from different companies are studied and the changes and ideas from these case studies added into the Infinity Model. The thesis concludes in chapter 7.

CHAPTER II

REVIEW OF LITERATURE

Reasons to improve software development methods

As stated earlier software today can be written in different languages for varying hardware and run on various operating systems. Over a period of time the requirements and the expectations of the software seem to increase, but the quality of the software seems to decrease [1]. The initial problem here is the method followed by large companies to code their software. Even big operating system companies tend to release software with known bugs and errors [9]. If the quality is not achieved in the first generation of the code then it becomes tougher in the future generations. The process model used will define the whole lifetime of the system. If the model is not good then the system has to be reprogrammed from scratch once again, leading to a waste of human and economic resources.

If one takes into consideration operating systems (OS), each and every operating system has a different method of functioning and no two OS can communicate with each other directly. The problem here is the methods used in the development process. Small software companies therefore find it difficult to code software to run on all operating systems. This makes them code a lot of drivers to run in different systems and change it

for newer generations. Hence the companies tend to write drivers with less quality.

Another issue is the methods of software development followed by the companies. Even though there are so many models for creation, due to time and economical constraints companies tend to perform various parts of the upgrade using the extreme programming method. The main idea behind the extreme programming method is to write codes in a simple fashion for immediate concerns without thinking of the future [10]. Due to this the project goes to phase-out stage sooner.

When so many problems can occur in a single generation, the problem multiplies for multi generation software. The various problems have been defined by the eight laws of software evolution given by Dr. Lehman [1].

No.	Brief Name	Law
I 1974	Continuing Change	*E*-type systems must be continually adapted else they become progressively less satisfactory.
II 1974	Increasing Complexity	As an *E*-type system evolves its complexity increases unless work is done to maintain or reduce it.
III 1974	Self Regulation	*E*-type system evolution process is self regulating with distribution of product and process measures close to normal.
IV 1980	Conservation of Organisational Stability (invariant work rate)	The average effective global activity rate in an evolving *E*-type system is invariant over product lifetime.
V 1980	Conservation of Familiarity	As an *E*-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behaviour [leh80a] to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.
VI 1980	Continuing Growth	The functional content of *E*-type systems must be continually increased to maintain user satisfaction over their lifetime.
VII 1996	Declining Quality	The quality of *E*-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.
VIII 1996	Feedback System (first stated 1974, formalised as law 1996)	*E*-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

Table 1: Laws of Software Evolution [1]

The challenges that are present today for evolutionary software have been discussed in a workshop called Challenges on Soft-ware Evolution (ChaSE 2005) [2], which was jointly organized by the ESF Research Network RELEASE (Research Links to Explore and Advance Software Evolution) and the ERCIM Working Group on Software Evolution. The table presenting the challenges in software evolution is given below.

	Research target	Time horizon	Studied artifact	Support type	Stakeholder
A	preserving and improving quality	long	software system	tools, techniques, formalisms	developer, project manager, end user
B	analysing, managing, controlling	medium	programs	common appl. framework, exchange formats, interoperability standards	researcher
C	controlling, supporting	short	models	tools, techniques, formalisms	software engineer
D	controlling, supporting	medium	any pair of related artifacts	tools	software engineer
E	all types of research	medium-long	formalisms	formalisms	researcher
F	controlling, supporting	short-medium	languages	languages, programs	language designer, tool builder, researcher
G	controlling, supporting	medium-long	languages, software systems	tools, standards	tool builder
H	managing, controlling	medium	software process models	software process models	manager, software engineer
I	motivating	short	managers	metaphors	executives, managers
J	analysing	short	version control tools	tools	tool builder
K	analysing	medium	all information useful to get insight in a software system's evolution	statistical models, empirical studies	researcher
L	analysing	medium-long	release histories of long-lived, large, complex industrial software systems	techniques, tools	researcher
M	analysing	long	every kind of evolving artifact of a software system	empirical studies	researcher
N	analysing, predicting	short-medium	software systems	predictive models, measures, metrics	researcher
O	understanding, comparing	medium	evolving software systems	benchmarks, exemplars	researcher
P	teaching	short	formalisms, techniques, tools, theories	course material	teachers, students
Q	understanding, supporting	medium-long	everything	everything	researcher
R	controlling, supporting	short-medium	languages, execution platforms	languages, execution platforms, programs	tool builder, end user

Table 2: Classification of software evolution challenges [2]

Several different methods and solutions have been discussed to solve the challenges shown in the table. These challenges can be faced only by improving the process models, languages and human training [2]. Software engineers are now at a stage where they will have to rethink strategies to achieve better results and produce good software. To achieve this, one of the main requirements is to create a better process model to code the software. In order to achieve a better model, one does not have to develop new algorithms which have no base model or tested strategy. One just has to look into nature to see how a biological entity works and how ecology, even though being so diverse with all the living organisms, can control evolution by a common inbuilt code called the DNA. By looking into biology and software generation techniques, a better process model can be built.

Similarities between software evolution and evolutionary biology

Nature controls all the organisms with a single code called DNA {B}. DNA is a nucleic acid that contains the genetic instructions used in the development and functioning of all known living organisms**.** By changing the DNA sequence in a micro level the organism is able to perform drastic changes. For an example, although only 5% of the chromosomes differ between a chimpanzee and a human, the difference between these organisms is very huge. One has to look into nature at various levels (such as DNA level, enzyme level, phenotype level etc…) to achieve a pattern and correlate our software to a common pattern. By achieving this, software can be created in a more quality oriented way.

This does not imply one can only compare software at the coding level. Software also can be compared to evolutionary biology in the process model level. Several questions like why mutation {C} happens, how nature does natural selection {D} and the main characteristics of evolution that leads to the survivability of the organism can help in creating an effective process model. Software and evolutionary biology can be compared at two levels, the macro level and the micro level. This paper will investigate the macro level idea first because only after the macro level black box is opened, will researchers be able to open the micro level black box.

Macro Level Comparison

At the macro level comparison huge similarities can be found between evolution and software. For example, if the program code is compared to DNA, then a single installation of the code is a cell and the whole software base for that code installed in different systems is an organism (organism is similar to installed base). The survival and reproduction of the organism depends on the code and the environment it runs on (environment is the hardware and user) [12]. Looking closely, both software and biological organisms have many functions in common for example, both try to replicate, repair and upgrade for the given conditions.

The process of repair and upgrade is done by the methods of mutation and selection in nature (Mutation, Selection are similar to Software Life Cycle). Mutation occurs when a DNA gene is damaged or changed in such a way as to alter the genetic

message carried by that gene. Natural selection is the process by which favorable traits that are heritable become more common in successive generations of a population of reproducing organisms, and unfavorable traits that are heritable become less common. This is similar to the requirement collection and planning in software.

Comparing the relation between an OS and the other applications, one can find the similarity to the symbiosis {G} in nature [12]. The term symbiosis can be used to describe various degrees of the close relationship between organisms of different species. The various device drivers and the operating system works in the form of symbiosis. But the most important part for survival of both software and nature is co-evolution {H}. In biology, co-evolution is the mutual evolutionary influence between two species. Co-evolution in software is improving the dependent software together (such as the operating system and the drivers), so that the quality and security in both the software is maintained through the generations. When all these biological properties are incorporated into the corresponding steps in software, it leads to the creation of evolutionary software.

Micro Level Comparison

Even though theoretical micro level comparison is possible at this time, the methods to achieve software evolution at this level will not be possible, until the architecture to create the evolutionary cycle in the macro level is defined. By taking a closer look at the micro level comparison one can see that DNA and software have a lot of similarities. For example, DNA is made of four codes - A, T, C and G and software is

made of binary codes - 0 and 1. In DNA even though there are four codes, A is complementary to T and C to G. They always occur as pairs and that makes them more like the binary code. Moreover the white blood cells available in DNA for protection from viruses are similar to how an anti-virus tries to protect code. In nature when an organism gets hurt, the white blood cells immediately try to quarantine the bad cells and stop the bleeding and then try to kill the virus. Similarly in most of the anti-virus today the virus code after detection is quarantined and deleted.

In nature, genotype describes the genetic constitution of an individual that is the specific allelic makeup of an individual, usually with reference to a specific character under consideration. The phenotype of an individual organism describes one of its traits or characteristics that is measurable and that is expressed in only a subset of the individuals within. The genotype-phenotype distinction must be drawn when trying to understand the inheritance of traits and their evolution. This genotype {E} –phenotype {F} modularity {I} and inheritance present in nature can be compared to the module-function relation present in the software [12]. When the phenotype is changed the genotype changes, similarly when a function is changed in the code the corresponding module undergoes a change.

In nature, information is passed from DNA to RNA in the process of transcription and from RNA to protein by translation. In software the assembly code is compiled to an object code and that is executed to get the output. There are also similarities in the way mutations take place. In software a singe function is taken and is changed according to

the newer requirements; in biology, DNA tries to change the parts which can make the organism survive in the new environment. Even though the similarities can be seen in this level one still has to first get a high-level pattern to achieve greater understanding into how to convert a code to a DNA sequence.

Even though such similarities can be seen in evolutionary biology and software evolution, there are a few differences also [13]. In biological evolution the pace is slow, and the mutations that take place are random. However, in software the mutations are decided in the requirement phase itself. Software has these variations because of human involvement, but one does not have to copy nature completely to create the model, one just has to understand the principles and incorporate them into software.

CHAPTER III

METHODOLOGY

A new evolutionary biology based software process

Whenever a new idea is thought of, one will have to go back and look into other process models and see their working methods. That is the main idea of evolution and so that is also the first step for creating the Infinity Model. When one looks into the existing software one can get a better idea for the need of a new evolutionary software process model.

Existing Models

The most common place models available today are the waterfall model, spiral model and extreme programming method. However when one looks into these software models even though they may be useful in some projects, they will not be effective to create an iterative cycle of integrated system software. In future, software is not going to be a separate module or a part, each and every component is going to be virtualized and the main requirement for those systems are going to be intercommunication, inter-adaptability and security [14]. Many systems created today cannot run on other software based systems and also if they are able to communicate their performance is poor. For example even though the latest Apple systems allow Windows OS to run in their

hardware, the systems tend to overheat and some operations cannot be performed as done in the Windows based system. Another example is when a lot of audio and video formats are present with no particular media player to play, leading to usage of unsafe software packages to run these files.

Going back to the models, in the waterfall model, once a step is crossed one cannot go back to that step, and this makes it only usable in very simple software [15]. There is no iteration in the waterfall model and therefore developments and upgrades cannot be done before all the steps are completed. This leads to wastage of time, cost and human resource. The spiral model even though it is iterative, does not include cycles for maintenance of the software. In this model if an error is found or a new idea is to be incorporated, it cannot be done before going to the next cycle. The spiral models disadvantage is that it comes back in the next cycle to do the correction, taking up a lot of time and resources. Furthermore, there are no process steps for upgrades or patches, and this leads companies to use other methods to change the code, and the original architecture is lost. The extreme programming method even though effective for small companies has some major flaws like minimum or no documentation, and no group programming. The other models like prototyping are costly and can be used by only large companies.

There are two other models designed to solve the problem of software evolution. They are the staged model [11] and the PSPM software life cycle [16]. Both the models are new and have been designed with the software maintenance perspective. Ideas have

been taken from them and incorporated into the Infinity Model.

There can be many kinds of models for small and less costly products which do not go into an iterative process. However, for large projects with lot of iteration there seems to be no universal process model and companies tend to build their own model, which is not shared and which leads to software mismatch.

CHAPTER IV

PROPOSED MODEL

Infinity Model based on software biology

The model which is proposed in this paper is named the Infinity Model. The name is to signify that this model is for processes and projects which keep going for generations. Generation does not just mean a new version, but also upgrades and patches within a single version. The main idea behind this process is iteration, but in a varied style where both full cycles and half cycles of the model can be performed. The other important advantages of this model are natural selection and future mutation. The ideas that are going to be implemented from biology are genotype-phenotype hierarchy, gene-robustness, and principles of symbiosis and gene duplication. These basic principles are the building blocks in biology and this when incorporated into software gives a better pattern, a simple and more effective design.

As stated before biology and software do have their differences and this leads us to consider software evolution also. If a closer look is taken at the methods to develop software, most of the software products today go in for updates and maintenance till the phase-out of a generation. They then think about the next generation, and what changes can be done in that generation but this will not help companies in the long run.

In the Infinity Model, the cycle does not start from the beginning but from the middle. This can be better understood when the model is explained. The idea of starting the process in the middle is to achieve half cycles in the process. The advantage of this is if one sees a problem in the methods which have been used, even though the planning and development methods are finished, instead of going into coding and then coming back one can straight away go back to planning. Furthermore, when a new upgrade for a part or a patch has to be done, developers need not wait till all the steps are completed, but can go to coding with all the initial documentation they have and try to create the code. By this the company can have a continuous research and feedback cycle, and all the time somebody will be working in either cycle of the model. The proposed approach will therefore save time and resources, while generating better code.

Whenever a project is started there are the issues of cost. The companies tend to spend more time on coding rather than designing, and ultimately waste more time. It is better to have a slow and steady process than to go into an overnight finished product. This doesn't mean the company has to spend extra cost. They just have to plan to do the process in parallel.

The method to do this is during the requirement [16] and the planning steps more time is spent, and during the coding step project is divided in to effective modules and created in parallel. Moreover, testing is made a part of the whole project. When these steps are followed, initially the projects pace may seem slow but during the later steps the pace will be quicker and the programming will be very effective.

The various Steps of the Infinity Model are

1. Evolution Or Revolution
2. Requirements == Mutations
3. Planning == Selection
4. Development == Genetic Pattern Generation
5. Coding == DNA production
6. Testing == Repair
7. Packaging, Deployment and Feedback == DNA replication

The diagram given below gives the Infinity Model. The two cycles can be viewed, the first cycle is called the diplomatic cycle and the second cycle is the technological cycle.

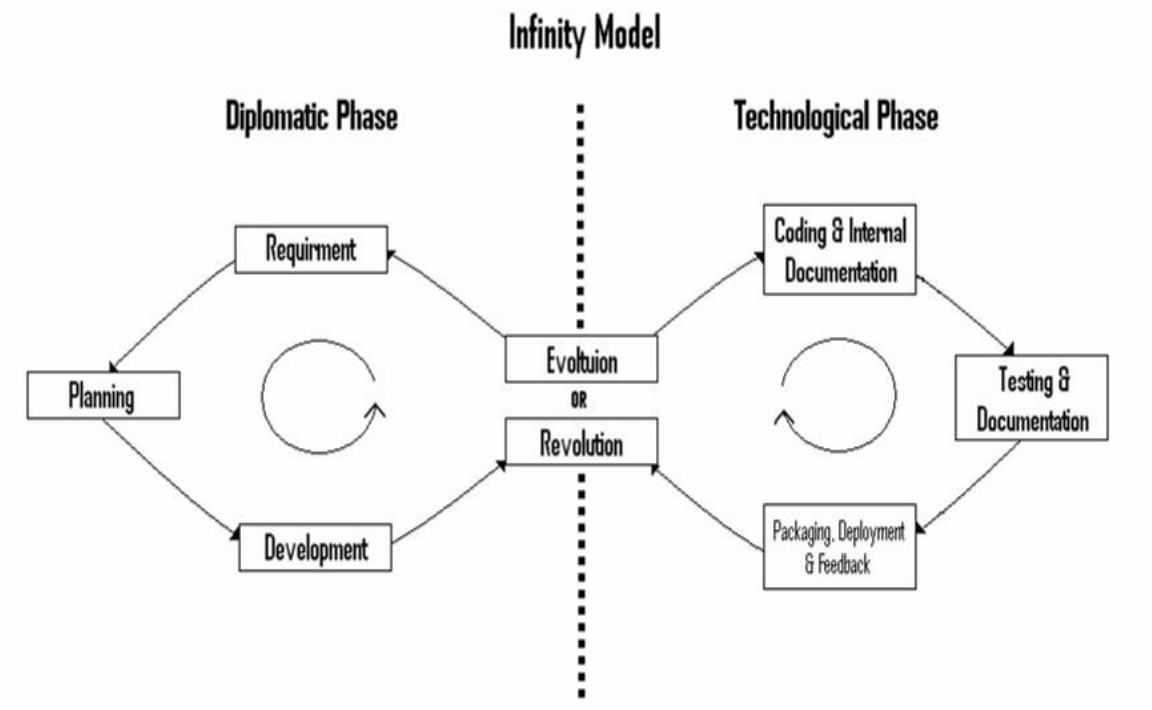

Figure 1: Infinity Model

Evolution or Revolution

The initial step is not planning but evolution or revolution. This is because the main idea of the project is selection, and this has to be done from the initial step. The selection that takes place here is, whether it is going to be creation of a next generation for existing software (evolution) or creation of completely new software (revolution). Even though evolutionary biology states that there is no macro evolution, in terms of software there is something that gives us a better understanding; both the creators and the methods of creation are known.

If the goal is going to be creating a new generation then it is an evolution, and documentation of the old project is taken into the diplomatic cycle. Each and every developer going to be involved has to work with the old product to get an idea of what they are going to do. The most important function to be performed is the collection of all feedback from previous users, and selection of properties which are going to be continued. If it is going to be a new project then all the modules created for other projects which can be used have to be collected. The competitor products available have to be thoroughly researched. The people involved in this step are based on the selection of evolution or revolution type.

<u>Evolutionary cycle personnel involved</u>

1. Project manager and programmers of existing generation

2. Second set of programmers to start work in new phase
3. Financial advisor (Experienced)
4. Client
5. Testers – Old generation and new recruits
6. Users of the Old generation
7. Private Reviewers

Revolutionary Cycle personnel

1. Experienced project manager
2. Quality oriented programmers
3. Financial advisor (committee)
4. Client
5. Private Reviewer
6. Testers (well experienced)

By doing the selection, a clear idea of how the project is going to be continued and also who are going to be involved is found. If the project is a continuation, the company already has the knowledge of the time and money that is going to be involved. The project should involve more experienced people to maintain quality. However if it is going to be a completely new project then more new developers can be used and fewer number of experienced people are enough.

The Infinity Model has internal evolutions also other than the overall software evolution, and these can be seen in the diagram below.

Evolution in Software

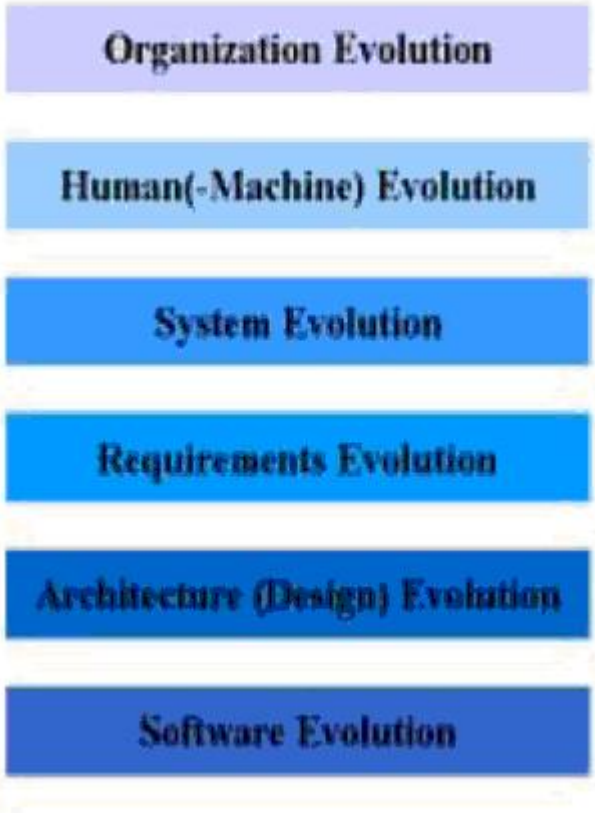

Figure 2: Evolution layers [17]

After the evolution or revolution step the various other evolutions like the requirement evolution and the architectural evolution take place. The necessity of these evolutions can be understood from the table below.

Evolution	Dependability Perspective
Software Evolution	Software evolution can affect dependability attributes (e.g., Reliability). Nevertheless software evolution can improve dependability attributes by faults removal and maintenance to satisfy new arising requirements.
Architecture Evolution	Architecture evolution is usually an expensive phenomenon. It does not affect directly dependability, but there is high risk if the evolution process is unclear and little understood. Architecture evolution may be needed to support specific system properties (e.g., redundancy, performance, etc.).
Requirements Evolution	Requirements evolution does not directly affect dependability, but non-effective management of the requirement process may allow undesired changes to fall down into the product affecting its dependability. On the other hand requirements evolution may enhance system dependability across subsequent releases.
System Evolution	System evolution may give rise to undependability. This is due to incomplete evolution of system resources. Evolution of some resources (e.g., software) should be taken into account by the other resources (e.g., liveware and hardware) in order to register a new configuration for the system. Hence the interactions among resources serve to effectively deploy a new system configuration.
Human (-Computer) Evolution	Human can react and learn how to deal with undependable situations, but continuous changes in the system configuration may give rise to little understanding about the system. Hence the human-computer interaction may become quite undependable as well.
Organization Evolution	Organization evolution should reflect system evolution. Little coordination between system evolution and organization evolution may give rise to undependability.

Table 3: Dependability perspective of Evolution [17, 20, and 21]

Requirements (Mutations)

The next step after making the decision of going into an evolutionary cycle or a revolutionary project is to get the requirements. In nature, the requirements that are collected are the change in the climate, predators and ecology. This leads to the

survivability and the adaptability of an organism. Nature takes in the input and does the process of mutation. It generates both good and bad mutations. Next using the process of selection, the good mutation (those that survive) are left and the bad mutations disappear. Even though the path followed by nature is at a slow pace, the important thing for software is, all the ideas generated during the requirement phase should be reviewed and documented. The most important concept in evolution is to understand that mutation is not the end of evolution but the beginning of a new one. Once a mutation takes place, a set of new mutations take place to support the change and standardize it. Similarly in software, the requirements are the starting step. Any new requirement is a starting point for a number of future requirements which will arise within the software life cycle or after a generation is released.

A continuous method of collection and implementation of the requirements is necessary. This is achieved by getting all the ideas (requirements) noted down and including them in the document. By this when the new project is going into generation all the old ideas can be looked up, and useful ones can be applied to the new phase. After getting all the requirements, they have to be arranged in the order of most interesting ideas to the least ones and the most applicable ideas to the least ones. This has to be documented and read by all the people involved in the project team before they come into the planning phase. By doing this, the groups when coding, can look up the other requirements and try to create modules in such a way that they can accommodate those requirements in the future.

The requirement maturity index [18, 19] for the software package is given by

$$RMI = RT - RC / RT$$

Where RMI is the requirement maturity index, RT is the total number of requirements and RC is total number of changes. Using this formula, engineers can decide the change in the size of the project and also the human resource involvement required. If the RMI increases then the complexity of the code will increase, and more testing will be required for the project.

In the Infinity Model, the requirement phase is performed with a different group of people to collect different types of data for both the cycles. The requirements will vary for both the cycles and the different requirements can be seen below.

Evolutionary requirement

The new set of requirements for this cycle is obtained from a number of people

1. Feedback from users
2. Requirements from the client
3. Ideas from the previous creators
4. Future changes that need to be done based on the software and hardware.

Checking into the future is required because the environment it is going to be deployed into may change before the project is released. The requirements collection will

have to proceed till the planned generation may exist theoretically in some form.

Revolution requirement

1. Complete requirement from the client
2. Questionnaire requirement from the future user
3. Future requirements if the product is going to go into cycles

The grouping of requirement engineering questions

1. Requirements Management Compliance
2. Business Tolerance Requirements
3. Business Performance Requirements
4. Requirements Elicitation
5. Requirements Analysis Negotiation
6. Requirements Validation
7. Requirements Management
8. Requirements Evolution & Maintenance
9. Requirements Process Deliverables
10. Requirements Description
11. System Modelling
12. Functional Requirements
13. Non Functional Requirements
14. Portability Requirements
15. System Interface
16. Requirements Viewpoints
17. Product-Line Requirements
18. Failure Impact Requirements

Figure 3: Requirement engineering questions [17, 20]

Even though the project may be new to the company, if the company is planning to venture into the market of the new software then a detailed study of how it is going to work in the future has to be made in this phase. By doing this the company can decide how to plan the human resource and the cost required for the project.

Planning (Selection)

The planning phase is the phase of selection. In evolution even though a number of mutants are created by nature, only those that survive are selected to be replicated. For example when organisms started the generation of an eye, initially most of the organism types did not have the biochemical components for an eye in them. When a change in the DNA {B} led for the creation of a single cell biochemical reaction, it was replicated in all the offsprings, and today almost all the organisms have some type of eye component in them. The eye of each and every organism has improved on the environment it lives.

In software there is a necessity to produce successful products, but how one can relate to evolution is by checking the survivability of software in the past and creating the new generation [22]. For example, in the past the graphical user interface for software was not available. All the computers worked on character based interface, but once the graphical user interface came into existence all software were made with visual interface. This leads to the point where during planning of software the visual aesthetics of the code have to be given a lot of importance. Furthermore, planning the look and feel of the software has to be discussed to get a successful product in the first release itself.

In the Infinity Model, during the planning phase the number of people should be high to perform a better selection of the requirements. They should be divided into groups. They will have to discuss about the various requirements to be selected for that generation. Then all the groups have to present their selection. The method of voting has

to be used to make the final selection of requirements from the different groups.

The groups should be allowed to make decisions on all parts of the project. The group should consist of all the people who are going to be involved in the process like the programmers, testers, hardware engineers, financial consultants, human resource managers and the clients.

Things to be planned

1. What kind of resources will be required
2. How the project is going to be performed
3. Language
4. Hardware Environment
5. Tools
6. The number of layers in the project
7. Future improvements and methods to allow them
8. Functions to be reused
9. Functions to be changed
10. What will be the size of the group
11. Time and cost analysis
12. People going to be involved
    a. Programmers
    b. Testers

Development (Genetic Pattern generation)

The development phase is where the real architecture is decided. The most important part in biological evolution is whenever a new DNA sequence helps the organism's survival it is incorporated into the existing DNA structure. When the organism reproduces, the basic DNA code of the organism has the instruction for the next generation DNA sequences also. The idea behind this is even if mutations are performed in the organism continuously only the code of the surviving pattern is replicated.

In software when creating an algorithm or a flow chart the most important thing to remember is that the properties of the old generation that survived and were liked by the users have to be repeated. The general structure should always be maintained to improve the success and security of the software.

In the development phase, a new evolutionary step starts which is the architecture evolution. Here all the requirements are designed into a formal architecture. This leads to new requirements and changes [17, 22] that have to be incorporated for standardizing the architecture. The development phase is never ending and will always have to be repeated to improve the system. The algorithms and the flowchart of the project are going to be developed here. During the development the number of layers in the project and the functions of the project have to be decided. For this group meetings have to be arranged and the full group has to meet at the starting and at the end of the project. This is to make sure that every one has an idea of the pattern that is going to be used in the project.

During the end meeting a person from an old project or some other project has to be called to inspect the phase, to make sure the planned components can be achieved with the architecture. The layers have to be decided to allocate coding groups according to each layer. The layers make it easy to calculate the time to be spent in the project. Examples for layers are the basic kernel level, the visual display level etc. The architectural properties which are decided here are functions, modules, GUI, partner software compatibility and security.

The biological concepts which are integrated in this phase are the gene-robustness {I}, genotypes {E}, phenotypes {F} and symbiosis {G}. By incorporating gene-robustness architectural stability is achieved. The core properties (inner most modules) of the code are well secured and changes to them is limited. This is done to make the base strong and secured. The internal kernel can undergo only minimal change in a generation to safeguard the quality of the software. The genotypes and phenotypes are the modules and functions that are going to be used in the project. In nature, when the phenotype is changed the genotype changes accordingly. The genotypes are modular, and this helps to reduce virus attacks. By making software more modular it is easier to make more changes and also remove modules if they are not working. The last is the symbiosis; while designing the system the architecture includes all the other codes which are going to survive on the main code (kernel). These codes also share the hardware environment with the main projects code. All these codes have to be collected and documented. This information is important to standardize the functionality and the reliability of the code in the hardware environment. By incorporating these biological concepts the survivability of

the software increases.

Group personnel

1. Project manager
2. Programmers
3. Testers
4. Client
5. Outside project manager
6. Financial advisor

Evolution OR Revolution

By coming back to this step the changes in the state of the project can be studied. For example different companies may have come up with similar products. New ideas which could get a breakthrough may have been found. Changes to the architecture to improve it or a new virus might have been found which could affect the code just decided. A project cannot be stopped for up-gradations, but the developers can start a new half cycle in the Infinity Model to look at methods to repair or improve the product [2, 17, and 25]. This cannot be done at the end of the project because of the time delay. Original ideas may be lost and changes that could have saved the system would ultimately lead to a huge loss. At the beginning of the project the ideas are varied, and groups involved want a lot of different things. Even though they may have sounded

promising at the beginning, when the development phase is reached people have more understanding to the project and realize its limitations. They are then in a better position to make better decisions on where improvements could be really made.

To achieve a better result from the project, the developers will have to redo the selection of evolution or revolution. By doing this a pattern can be generated, and if a new idea is found to improve the project or correct the problems found during the algorithm generation, it can be sent to the next evolutionary cycle. If a completely new requirement (an extra tool) outside the pattern is found then it goes to the revolutionary cycle. This recycle is not done by the old personnel who are going into the technological cycle, but by a new team who have been looking into the project from the outside from the beginning. This new team starts the work on the improvements and by doing this the company can have two teams who have a good idea of the project. The security and upgrading can be done with little problems as the product has been in a continuous improvement cycle.

Personnel Involved

1. Project manager (Two people)
2. Tester (One or Two teams)
3. Programmers (Two Teams)
4. Client
5. Financial consultant

The original group should be used in the initial stage along with the new group. The second group should take over and start the process again if there is place for development.

Technological Cycle

The second cycle in the Infinity Model performs the technical side of the project. This cycle is more private. It involves only company workers like programmers and testers. However in the final step, the client is brought back to perform problem solving and provide feedback. Software maintenance [2] is basically performed in this cycle. The main advantage of the Infinity Model is to allow software maintenance to be a cycle in the iterative model rather than a separate step.

Many companies perform updating and correction of large projects by a method called the extreme programming style. This is a separate method and is not a part of the original process model. The main difference in the Infinity Model is to standardize the methods and help companies to achieve better results than the methods they already employ. For example even though extreme programming [10] is used to reduce time, the amount of material the company has before starting coding is very limited and this leads to programming errors and loop holes.

In the Infinity Model this cycle is a repetition. By performing this part of the cycle

alone the architecture of the program is maintained, and newer changes are being incorporated in the documentation. This cycle performs the upgrading and error corrections. This cycle reduces the complexity and increases the quality of the software.

Coding and Internal Documentation (DNA production)

This is the start of the second cycle. The architecture is decided and algorithms are given to the various groups. The groups were decided by the project manager and the human resource personnel in the previous phase. The programmers are given the functions and layers which they are going to code. The standard methods for inducing security in the code [23, 24] are provided. A few years ago security in software [14] was not a big issue and programmers where coding using different methods. With the latest security threats, there is a need for programming methods with internal security. This can be found in biology were DNA {B} has an internal code for repair called the white blood cells [6].

The purpose of these cells is to quarantine and heel the parts which are affected and protect the parts which are not. Whenever a DNA is created the basic pattern of repair is also coded with it in all the repetitions, and it makes it easy for the organism to detect viruses. For example if humans were attached by a virus, the body tries to increase the body temperature and give symptoms to inform of the attack. It does not go straight down to shutdown mode. Similar methods have to be performed in the coding phase. The functions when joined have to coexist and protect themselves when attacked [16, 24].

Check points and internal testing have to be constructed within the program.

The programmers have to follow the algorithm created in the development phase seriously. No deviations are allowed from the main specs. Programmers also have to do some testing before giving it to the test group. The tests should be in the form of grey box testing [24] and should test the basic requirements. This has to be performed by the group which did that particular module of the program and also by the groups which performed the predecessor and successor modules.

Personnel Involved

1. Project manager
2. Programmers
3. Testers

This phase alone can be used to perform upgrades and maintenance. To do this the company will have to create an algorithm to allow the changes in the architecture. The algorithm should follow the pattern of the existing code and have the security measures inbuilt in code. The group which does the program has to see the issues which led to problems in the code, and try to create code patches without creating dormant code. The style of programming differes according to the project size; for large projects larger groups are used and a single group does a single module. For smaller projects, pair programming is used.

Testing and Documentation (Repair)

Testing [26] is the toughest part of the cycle. The testers will have to check if the product meets requirements, and if the quality and security issues are met. They have to look into the code without any prejudice and see if the code follows the check points, if the algorithm was followed and if any dormant code was created. The tester should be given permission to question the programmers on the parts which are doubtful. On the whole, the project depends more on the testers than on the programmers. If the tester misses an error, it leads to loss that cannot be corrected in that version.

Testing is the place where real mutations happen. The testers are the first users to find new requirements, changes for the next generation and the necessary updates. They give out not only the errors, but also the necessary first hand information on how the product works and also the parts which need change.

As the testers have been involved with the project from the start, they should create test modules before the programmers do the coding. They should also generate tests to check the check points and virus checking mechanisms constructed into the program. Extensive black box testing should then be performed and if the outputs are wrong then white box testing is done. Some level of mutation testing has to be performed to check for random errors that could have been created by the programmer. Testing should be done not only by the personnel involved, but also by the client at the end to make sure it meets the requirement. This is because, correcting a product already into

production is tougher and would be a lot easier if there is an unofficial check by the client beforehand. If it is a product like an operating system then the workers of the company have to be made to use the product. The inputs have to be used by the testers and programmers.

Documentation is an internal part of testing and the documents have to be updated continuously to report a success or a failure. If there is a failure then the reason for its occurrence and the corrective steps have to be documented. After the corrections are mode, the changed modules and results should be attached to the document.

Personnel Involved

1. Test Lead
2. Junior Tester
3. Client
4. Programmer if required

Testing is similar to planning, but here the selection takes place on the parts of the code to be upgraded and the changes that need be made.

Packaging, Deployment and Feedback (DNA Replication)

This is the last step in a single cycle of the Infinity model. This by itself is not a maintenance step like the waterfall model or the spiral model [23, 24, 28, and 29]. This is

because maintenance is not a single step. Based on the feedback the maintenance may require a full cycle or a half cycle repetition to get the required result.

The first part of this step is packaging. The overall packaging of the software and all the help utilities decides the survivability. In the finished product, the necessary help topics are added to the code from the documentation. The next part is deployment. It may be done in beta versions or as a full version. If it is released in beta then the feedback is initiated in a large scale. If it is the full version the maintenance phase of the project begins. The feedback is a multi-level, multi-loop and multi-user feedback. The feedback is the most important step for any evolutionary product. The general public or users tend to use the project in a way not decided by the creators and therefore are more likely to suggest new ideas and report errors.

This is not an end step but the start of evolution for this generation and the next [16]. The feedback from the help desk is the most important part of the documentation for the evolutionary process model. The company will have to document all the new ideas and errors without repetition. By doing this, when the cycle goes back to the evolution or the revolution step the teams can sit around and analyze the next generation.

Personnel Involved

1. Help topic documentation writers
2. Project manager

3. Programmer (Any of the Two groups)
4. Client

In the next step of Evolution or Revolution, the updates and changes from the feedback are done. The decision to go back to the diplomatic cycle is made. Consolidation of the changes in equal intervals of time is done based on the number of requirement changes (RMI). After all the parts are finished the company goes back to the initial step. Here decisions to improve the product are made. The economical gains achieved and the other clients for the product are explored. This is similar to nature where the survivability and adaptability of the new organism [12, 13] is tested continuously. All this leads to the next generation of the organism or software. The human resources involved in this step are the programmers, testers and the clients. They have to decide whether it is going to be a half cycle or the full cycle for upgrading the software. This is the last step in a single full cycle.

In the figure below, the various steps are divided into appropriate parts. There are two important cycles; they are the requirement – feedback cycle [17, 20] and the development - coding cycle. Both provide requirements to the software in different ways. The feedback – requirement cycle provides new ideas, consolidation and updates to be added to the generation. The development - coding cycle allows changes which are used to correct the requirements which are already available, and by natural selection the required properties are incorporated in to the final product.

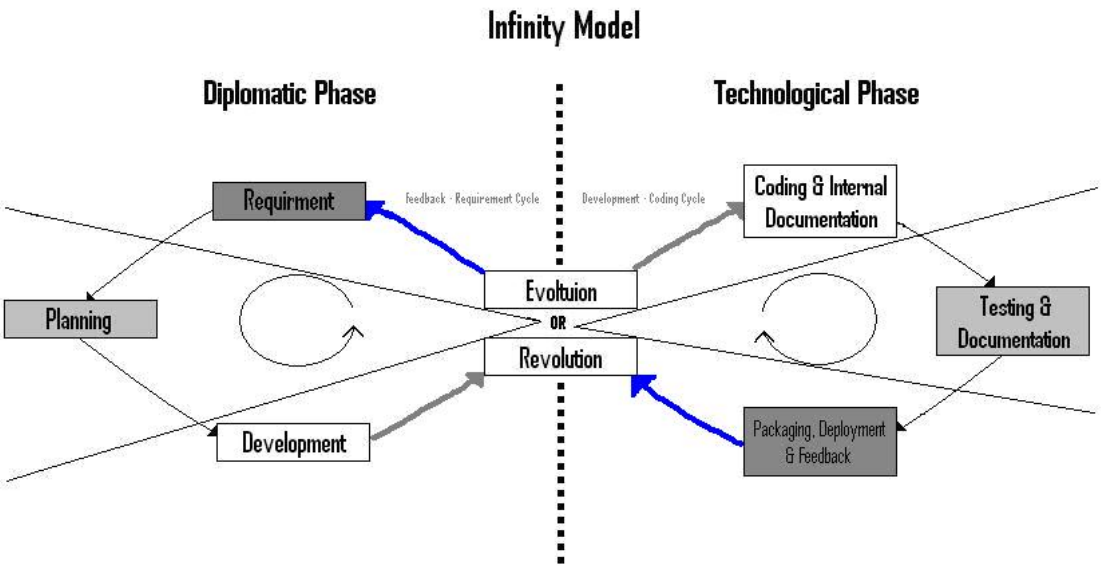

Figure 4: Loops in Infinity Model

The Infinity model is an abstract model designed to help evolutionary software and improve the methods of production of software. The Infinity model incorporates several evolutions like the requirement evolution [17], architectural evolution [20], system evolution [17, 21] and software evolution. It also contains the basic principles of evolutionary biology. The model is designed in a way that any kind of company, small or large could use it to design software

The Infinity model is designed to reduce the economical constraints [30] present in evolutionary and legacy software. This is done by giving methods and ideas to improve the human resource usage, time reduction and economic reuse of the functions in the previous generations. Any company can take up the model and customize it to incorporate the company policies and procedures. The process model is itself a starting step for mutations.

CHAPTER V

CASE STUDY: SOFTWARE PROCESS MODEL EVALUATION

In this chapter the methods followed by the existing process models are looked in depth. The models disadvantages for evolutionary software are given in the form of a critique. The models looked into are the waterfall model, spiral model, prototyping, extreme programming, staged model and the PSPM model.

Waterfall Model

The waterfall model [23] is a purely sequential method of performing software engineering. The first step is the requirement collection and after all the requirements are obtained the process moves to design. In the design stage the method of development of the software is planned and the architecture of the software is created. When the design is fully completed, an implementation of that design is made by coders. During the coding phase several programmers work in small teams and develop separate parts of the software. At the end of this phase all the parts are integrated. After the implementation and integration phases are complete, the software product is tested and debugged. Any faults introduced in earlier phases are removed here. Then the software product is installed, and maintenance is performed to introduce new functionality and remove errors.

Thus, in the waterfall model the team moves from one phase to the next only after the preceding phase is completed and perfected. Phases of development in the waterfall model are discrete, and there is no jumping back and forth or overlap between them. However, there are various modified waterfall models that may include slight or major variations upon this process.

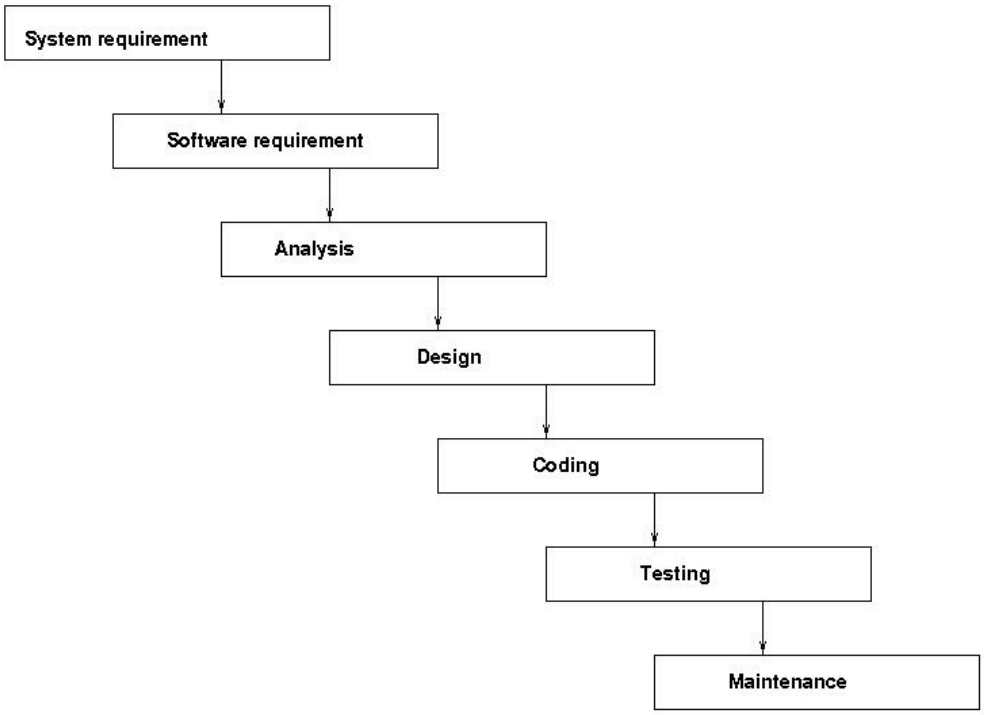

Figure 5: Waterfall Model [23]

Critique

The waterfall model is the classic model. All the steps of software development are defined in this model, but its major disadvantage is that it has no iteration [23, 31]. Unless those who specify requirements are highly competent, it is difficult to know exactly what is needed in each phase of the software process before time is spent in the

following phase [32]. The design phase may need feedback from the implementation phase to identify problem design areas. The main idea behind the waterfall model is that experienced designers may have worked on similar systems before, and so may be able to accurately predict problem areas. Because of this, the developers do not have to spent time in doing prototyping and implementing [32, 33]. Continous testing from the design, implementation and verification phases is required to validate the phases preceding them. Constant prototype design work is needed to ensure that requirements are non-contradictory and possible to fulfill. The implementation has to be performed continously to find and inform the problem areas to the design process. Constant integration and verification of the implemented code is necessary to ensure that implementation remains on track [33]. The counter-argument for the waterfall model is that constant implementation and testing to validate the design and requirements is only needed if the introduction of bugs is likely to be a problem. Frequent incremental builds are often needed to build confidence for a software production team and their client.

It is difficult to estimate time and cost for each phase of the process without doing some evaluation work in that phase, unless those estimating time and cost are highly experienced with the type of software product. The waterfall model brings no formal means of exercising management control over a project and planning. Moreover control and risk management are not covered within the model [31, 33]. Very specific skill sets are required for each phase; thus there is a requirement for multiple projects to run in sequence to optimize resource use. All members have to stay through the course of a given project, or the company will suffer skill levels by using inexperienced resources.

Spiral Model

The spiral model [24], also known as the spiral lifecycle model, is a systems development method (SDM). This model of development combines the features of the prototyping model and the waterfall model. The spiral model is intended for large, expensive, and complicated projects.

The working of the spiral model starts with collecting of requirements. The new system's requirements are defined in detail. This usually involves interviewing a number of users representing all the external or internal users and other aspects of the existing system. A preliminary design is created for the new system. A prototype of the new system is constructed from the preliminary design. This is usually a scaled-down system, and represents an approximation of the characteristics of the final product. A second prototype is evolved by a fourfold procedure: evaluating the first prototype; defining the requirements of the second prototype; planning and designing the second prototype; constructing and testing the second prototype. At the customer's option, the entire project can be aborted if the risk is deemed too great. Risk factors might involve development cost overruns, operating-cost miscalculation, or any other factor that could, in the customer's judgment, result in a less-than-satisfactory final product.

The existing prototype is evaluated in the same manner as was the previous prototype, and, if necessary, another prototype is developed from it according to the fourfold procedure outlined above. The preceding steps are iterated until the customer is satisfied that the refined prototype represents the final product desired. The final system is constructed, based on the refined prototype. The final system is thoroughly evaluated

and tested. Routine maintenance is carried out on a continuing basis to prevent large-scale failures and to minimize downtime.

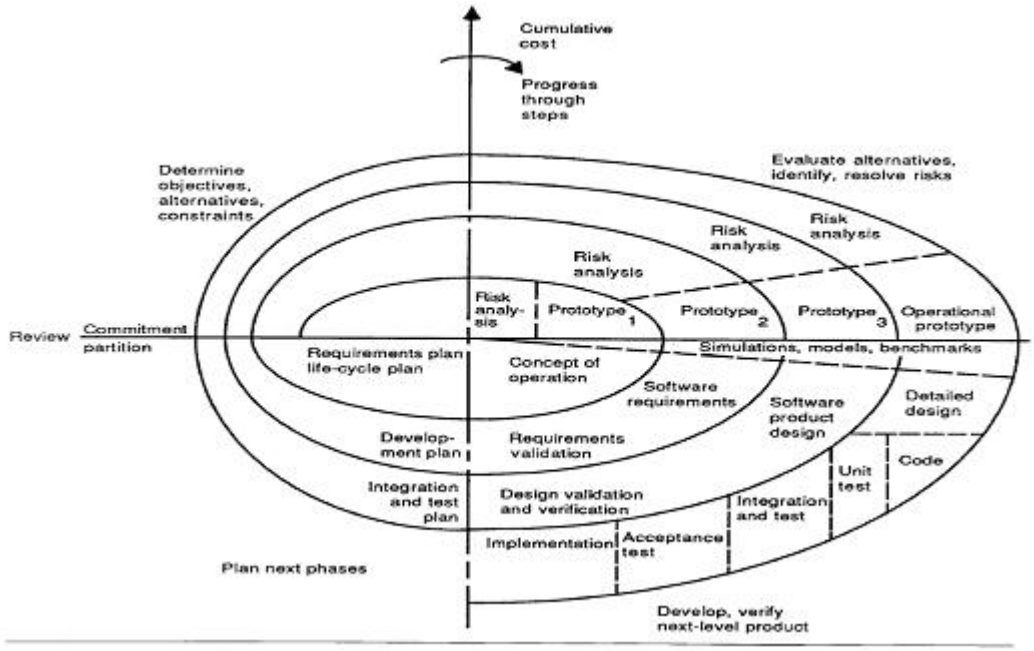

Figure 6: Spiral Model [24]

Critique

This is the first model which used iterative cycles to produce software and there are a few disadvantages with the model [34]. The model takes a lot of time to finish one cycle. The risk assessment needed by the model cannot be done by all companies in the beginning of the software development itself. Because of the risk assessment companies will not be able to use it in general software production [35]. The process guidance in determining objectives, constraints, and alternatives are not explicitly defined. Most of the companies lack risk assessment expertise.

The assessment of project risks and their resolution is not an easy task. A lot of experience in software projects is necessary to accomplish this task successfully [34, 35]. Because of the dynamic and risk driven approach of this model, the phase products and milestones are hard to define.

The main disadvantage from the point of view of software evolution is that it does not perform any cycle for the maintenance of the generation. The time spent for the single generation is good for real time products but cannot be used by commercial companies. It is also expensive and requires a lot of prototypes. The time consumption and human recourse distribution is not explained for the maintenance part of the software development. Due to the well defined structure of the spiral model all the companies cannot use it effectively. The time and cost do not allow small companies to perform such large procedures and expertise [35]. It is however the best model for projects which require reliability and quality at the highest standards.

Prototyping

Software prototyping [28] is the process of creating an incomplete model of the future software program. This model is used to let the users have a first idea of the completed program or allow the clients to evaluate the program. The main advantage of prototype is the software designer and implementer can obtain feedback from the users early in the project. The client and the contractor can compare if the software made matches to the software specification, according to which the software program is built. It

also allows the software engineer some insight into the accuracy of initial project estimates and whether the deadlines and milestones proposed can be successfully met.

The process of prototyping involves the following steps [28]

1. Identify Requirements: Determine basic requirements including the input and output information desired. Details, such as security, can typically be ignored.
2. Develop Prototype: The initial prototype is developed that includes only user interfaces.
3. Review: The customers, including end-users, examine the prototype and provide feedback on additions or changes.
4. Revise and Enhance the Prototype: Using the feedback both the specifications and the prototype can be improved. Negotiation about what is within the scope of the contract/product may be necessary. If changes are introduced then a repeat of steps three and four may be needed.

Critique

The focus on a limited prototype can distract developers from properly analyzing the complete project [36]. This can lead to overlooking better solutions, preparation of incomplete specifications or the conversion of limited prototypes into poorly engineered final projects that are hard to maintain. Further, since a prototype is limited in functionality it may not scale well if the prototype is used as the basis of a final deliverable. This may not be noticed if developers are too focused on building a prototype as a model [37]. Users can begin to think that a prototype, intended to be

thrown away, is actually a final system that merely needs to be finished or polished. This can lead them to expect the prototype to accurately model the performance of the final system when this is not the intent of the developers. Users can also become attached to features that were included in a prototype for consideration and then removed from the specification for a final system [36, 37]. If users are able to require all proposed features be included in the final system this can lead to feature creep. Developers can also become attached to prototypes they have spent a great deal of effort producing; this can lead to problems like attempting to convert a limited prototype into a final system when it does not have an appropriate underlying architecture. It cannot be used by small companies because of the cost. It is an expensive method of software development which includes several prototypes. The prototypes take a lot of time for creation and the company may skip important requirements to reduce the time. This leads to incomplete generations of software.

Extreme Programming

Extreme Programming [29] is the mostly widely used agile methodology to date. Originally formulated by Kent Beck with collaborators such as Ron Jefferies and Martin Fowler, XP consists of approximately twelve interconnected practices, making it the most well-defined agile process. It has been adopted by development groups around the world in a variety of different companies.

The twelve practices of XP are: [29]

A. Planning Game

B. Small Releases

C. Customer Acceptance Tests

D. Simple Design

E. Pair Programming

F. Test-Driven Development

G. Refactoring

H. Continuous Integration

I. Collective Code Ownership

J. Coding Standards

K. Metaphor

L. Sustainable Pace

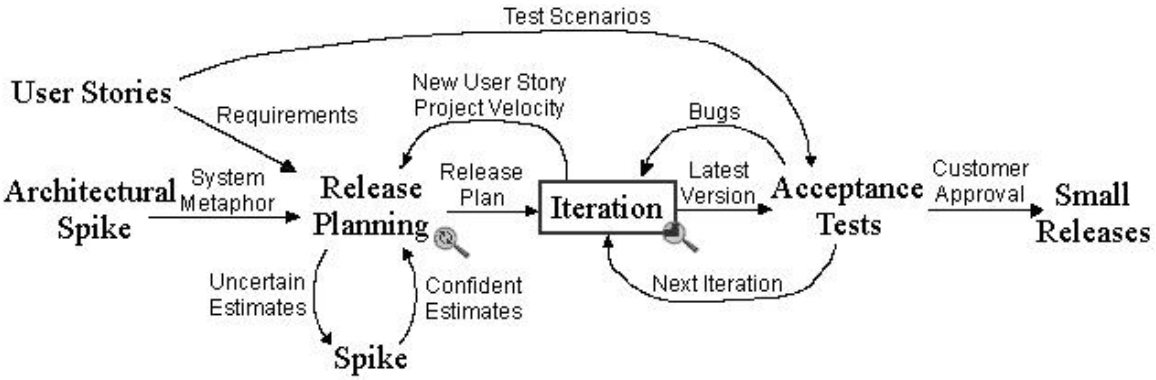

Figure 7: Extreme Programming [38]

Critique

The principles of extreme programming are to reduce cost and time and make it usable by everyone [39]. The biggest problem with this method is that it does not look for quality and reliability. The companies tend to use informal and flexible methods for

servicing which leads to lot of loop holes in the software, and people trying to service it in the future do not have a complete idea of what was done. Groups to control the changes in the code are being used by the companies using extreme programming and this is a sign that there are potential conflicts in project objectives and constraints between multiple users [40]. XP's expedited methodology is somewhat dependent on programmers being able to assume a unified client viewpoint, so the programmer can concentrate on coding rather than documentation of compromise objectives and constraints [39, 40]. This also applies when multiple programming organizations are involved, particularly organizations which compete for shares of projects.

The problems with extreme programming in case of quality are requirements are expressed as automated acceptance tests rather than specification documents. Requirements are defined incrementally, rather than trying to get them all in advance. Software developers are required to work in pairs.

There is no big design up front. Most of the design activity takes place on the fly and incrementally, starting with the simplest thing that could possibly work and adding complexity only when it's required by failing tests [40]. A customer representative is attached to the project. This role can become a single-point-of-failure for the project, and some people have found it to be a source of stress. There is also the danger of micro-management by a non-technical representative trying to dictate the use of technical software features and architecture.

Extreme programming can be used in small software which have minimal cost and time involved. This software tends to be the weak links for the virus to attack.

However, it has been claimed that XP has been used successfully on teams of over a hundred developers [39]. It is not that extreme programming doesn't scale, just that few people have tried to scale it, and proponents of XP refuse to speculate on this facet of the process.

The Staged Software Life Cycle Model

According to the staged model [11], the life cycle of a software system starts with initial development where a first functional version of the software is produced. Then the software moves on to evolution stage, during which the system's functionality is enhanced or adopted to satisfy the user's requirements. The servicing phase allows minor repairs and small functional changes only. From there, it is inevitable that the system eventually passes on to the phase-out stage where the system is being kept alive but is not changed any more. This is because no developer or maintainer dares to touch the system after that. Finally, the system is closed down and replaced by the next generation.

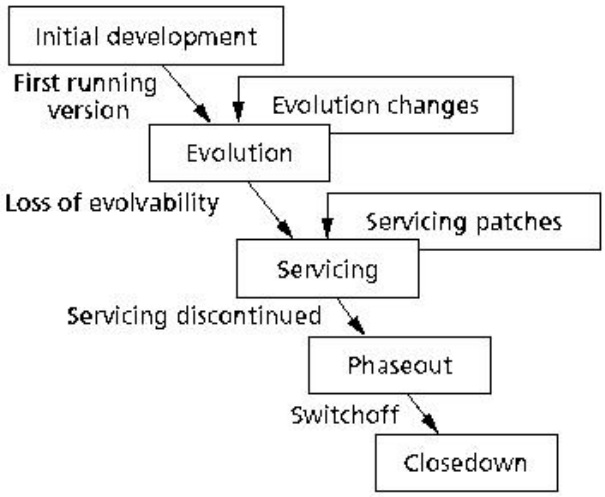

Figure 8: Staged Model [11]

Critique

This model was designed to improve the working of large systems [16]. It certainly helps in discussions between management and technical staff about the state of a system and necessary technical decisions, and their consequences. However, it is not well defined on issues that would be important for constructive improving of system evolution. The model does not give any ideas on how to stay in the evolution stage as long as possible. It uses, but does not define the term architectural integrity that according to the model, seems to be one of the major pillars on which evolution of the software relies [11].

It also states that systems can not return from servicing back into evolution. There are several counterexamples to this, if one thinks for example of Open Source Software such as Linux or commercial products, such as SAP, that were successfully serviced and evolved in several iterations over long periods of time [16]. A new user looking at the model may come to the conclusion that the initial development is viewed separately from the rest of the life cycle. The initial phase has a decisive impact on the life-time of the system. Long running initial developments also are themselves composed out of evolution and servicing steps.

Even though the model gives a good idea of the various maintenance phases it does not define how a company can move back into other phases and it also does not give a complete evolutionary model.

PSPM model

The main idea behind the PSPM model [16] is that from the starting of the life cycle the system enters a process that alternates between evolution and consolidation phases.

The consolidation phase constitutes the bottom-up portion of the process. The existing system is taken and modified according to technical aspects without actually adding new features but changing what is already there. The evolution phase is the top-down part of the PSPM. In this phase, requirements are elicited; refined and corresponding features are integrated into the system. The primary focus of this phase is to implement the requirements. The PSPM differs from other iterative processes models significantly by respecting the role of both activities at the process level.

This process spans the complete life cycle of the system until its phase out. Between the major phase's evolution and consolidation, the system is serviced, that is minor corrections or enhancements are performed. The other main idea is that the single servicing activity cannot degrade the quality of system but small changes over a long time tends to weaken the system and pushes it to a phase out stage.

Figure 9: PSPM Model [16]

Critique

This model is very basic and gives a method to perform evolution in a constructive way, but this model does not give the steps to be used for evolution or the time to be spent on the servicing and consolidation steps. The model just states that at equal intervals of time the whole code is to be consolidated. However when evolution occurs and a new phase is released consolidating the old phase will be of less usage to a company as they will be concentrating more on the new evolution [16]. The next biggest advantage in this model is it helps software to stay in the evolution stage till the company wants the software to phase-out, but this may lead to a legacy system.

After collecting all the advantages and disadvantages of the various models, we created a table comparing the existing models with the Infinity Model. In the table the important properties required for software evolution are compared for the different models. The comparison of the various models gives us the advantages of the Infinity model for creating evolutionary software. This can be noted from the table below.

Properties	Water Fall Model	Spiral Model	Prototyping	Extreme Programming	Staged Model	PSPM Model	Infinity Model
Iteration	No	Yes	Yes	Yes	N/A	N/A	Yes
Quality Oriented	Yes	Yes	Yes	No	Yes	N/A	Yes
Risk Assessment	Yes Partial	Yes Time Consuming	Yes	No	N/A	N/A	Yes
Human Resource	Experienced	Partial	Partial	Experienced	Partial	N/A	Partial
Evolutionary Maintenance	No	No	Yes	No	Yes	Yes	Yes
Time and Cost Analysis	Yes	Yes	Yes	No	No	N/A	Yes
Consolidation	No	No	No	No	No	Yes	Yes
Pair Programming	No	No	No	Yes	No	N/A	Yes
Types of Project	Small	Large	Large	Small	Large	Large	Small and Large
Model Type	Abstract	Well Defined	Abstract	Abstract	N/A	Abstract	Abstract
Servicing	No	No	No	Yes	Yes	Yes	Yes
Requirement – Feedback cycle	No	Yes	Yes	Yes	N/A	N/A	Yes
Evolutionary process model	No	No	Yes	No	Yes Maintenance	Yes Maintenance	Yes

Table 4: Comparison between different software life cycle models

CHAPTER VI

CASE STUDY: EXAMPLES FROM COMPANY SOFTWARE

In this chapter, various problems faced by different types of companies due to software evolution are studied. The case studies cover different types of software such as operating systems, embedded software, real time software and device drivers. The methods incorporated into the Infinity Model to improve the problems in these case studies are described at the end of each case study. The final case study is about evolution in a biological organism (lizard). From this case study the necessary mechanisms for survival of an organism are studied and the comparative software mechanisms are incorporated into the Infinity Model.

Avionics case study

In an avionics case study [41, 42], the evolution of requirement in a real time software environment is studied. In the case study, the authors have published the various stages the software goes through. They have showed that the requirement phase is not a single entity but takes place through out the life of the software. To understand this better the important points of avionics case study are shown in this paper. In the case study 22

releases [17, 41] of the different generations of the avionics software were taken and requirement changes that occurred in the generations of software were displayed.

A closer look into the study shows the requirements for software constantly change within a generation of the software and updating for the old software is required constantly. This can be better understood from the diagram below.

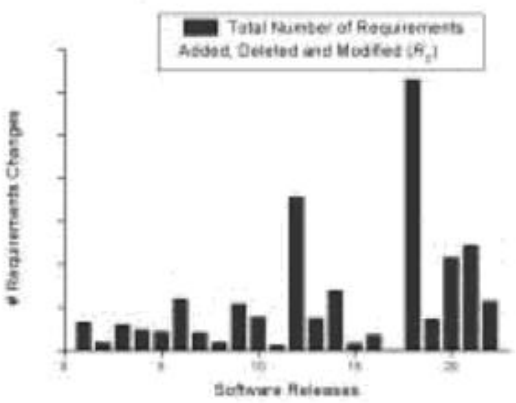

Figure 10: Number of requirement changes per software release [41]

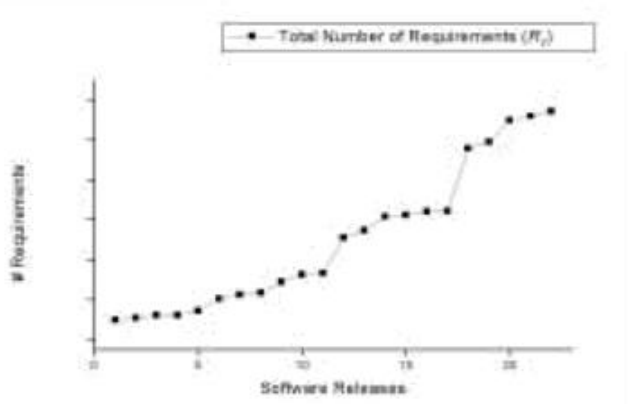

Figure 11: Total number of requirements per software release [17, 41]

From the picture above one can find that the number of requirement changes is very few when a completely new generation comes out. Moreover the requirements grew drastically [42] with succeeding generations. This shows that the rate of requirements goes up in comparison to the complexity of the software. The constant need for improvement leads to constant up-gradation of a generation.

The main idea behind requirements evolution is to improve the life cycle of the software. By better understanding and documenting the requirements they overall quality and reliability of the software can be improved.

Type of Change	Description
Add, Delete and Modify requirements	Requirements are changed due to the specification process maturity and knowledge.
Explanation	The paragraphs that refer to a specific requirement are changed for clarity.
Rewording	The requirements itself does not change, but it is rephrased for clarity.
Traceability	The traceability links to other deliverables are changed.
Non-compliance	A requirement that is not applicable for a new software package. This is the case when the requirements specification is based on that one of a previous project.
Partial compliance	A requirement that is applicable partially for a new software package. This is the case when the requirements specification is based on that one of a previous project.
Hardware modification	Several changes are due to hardware modifications. This type of change applies usually to hardware dependent software requirements.
Range modification	The range of the variables within the scope of a specific requirements is modified.
Add, Delete, Rename parameters/variables	The variables/parameters to which a specific requirement refers can change.

Table 5: Types of requirement changes identified in the case study [17, 41]

From the table above, one is able to find the various types of requirement changes that had occurred in the avionics software. These are the changes which occur in most of

the software. From this table one can find out the various changes to be noted down by the project team when they perform a requirement collection. It also gives an idea of where future changes could be found in the software.

This concept has been absorbed into Infinity Model and throughout the course of the software life cycle, the requirements of user, clients and coders are documented and evaluated for the next cycle [17]. By constantly reviewing the requirements in the model a company will be able ready for the future changes and the customer requirements.

Microsoft Software

The information for this case study was collected from the Microsoft case study [11, 43]. From this study sees that Microsoft does not use the traditional software maintenance followed in general by other software companies. They use the method of releasing their operating system to their users, and then starting to update the software from the errors and requirements reported by the user.

By this process they reduce the time in the development phase. This also helps to reduce the economical cost [11]. This process cannot be used for real time software but may be helpful for commercial software. Although the method has been successful for Microsoft, there may be huge problems if they have a competition in the operating system field [43]. If there is competition they will have to improve their software structure and also think of reliability during the production of the software than at the end

of the life cycle. The main principles they might have to incorporate to improve their software would be to start the whole process of software generations with multi user requirements. They will then have to create more modules and functions [43] to make the software more secured and reliable. Many of their competitors tend to use these methods and produce software which is far more reliable.

The Infinity Model incorporates ideas from the existing method used by Microsoft and as well the changes required. In the Infinity Model the versioning system is used to mark each and every cycle, it can be used for half cycles also. When a complete change is made it is incorporated into the next generation and released along with the older code. The process of programming used in the Infinity Model also gives an idea of the necessity for maximum modularization of the code.

The method used for the maintenance of the software leads to complete documentation of the changes, and leads to better consolidation and understanding of the changes. The most important thing to be understood from the Microsoft case study is the method used by them for evolution. They always start the next generation of the software once the older generation has reached a standard point. By this they do not become a legacy system and also the environmental changes are completely utilized by the later generations. This constantly keeps them up to date in the operating system software. The cycle model in the Infinity Model allows such a scheme.

Embedded System

In the embedded system case study [11, 44] one can see how small embedded system companies create their software, and also the problems they face with every new generation of the operating system. These small software companies tend to use little or no documentation during the development of their software. They use methods like extreme programming to perform the coding, and so they tend to create programs with a lot of errors even for a single generation. These drivers tend to be open links for the virus to attack the operating system. These codes with no quality or standards tend to waste the hardware resources and reduce the quality of the operating systems [44].

The case study shows that these companies use C, C++ or BASIC to code the software. Moreover consolidations [11] of the codes are not planned at any stage. With every new change or new generation of the OS the device drivers have to be rewritten or changed completely. There is no level of planning for the next generation and this leads to phase out the codes written for the embedded system. Considering all of this, traditional software maintenance offers little help. If the Infinity Model is used, the methods to collect the requirements from the partner company's on the changes in next generation are given. The model also gives the necessary consolidation techniques needed to improve the quality of software created [11]. If the companies understand the process of requirement - feedback cycle then they will be able to produce software with better quality, and the components created for a single generation can be used in the next generations also.

Open Source Software

Dr.Scacchi made a case study on open source software like Linux, Apache server and Mozilla Fire Fox [45, 46]. In this case study, several important lessons could be learned on how open source software develops and also what makes open source software successful.

In all open source software, programmers constantly change the code to adapt to new requirements. Each and every change, according to the grouped requirements is updated in to the code [46]. The decision of consolidation and selection of updating leads to the main survival of the system in the evolving hardware. As an example the number of changes Linux has gone through in the last 10 years can be seen from the figure below.

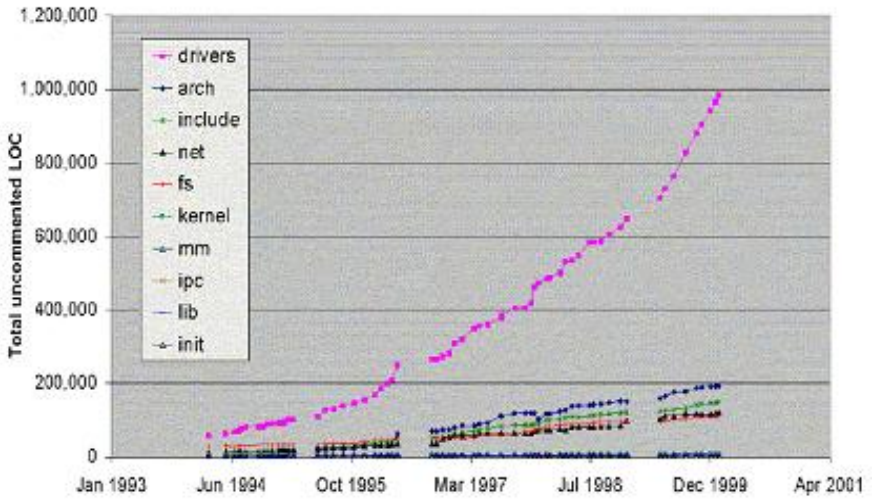

Figure 12: Data showing the size and growth of sub-systems in the Linux Kernel [45, 46].

In open source software, several number of programmers from different places tend to improve the way the single code works. There is also a downside to this method of development; as there no rules on the programming style if one of the programmers makes a mistake or creates junk code then the code will fail for a new user. The open source software will always be helpful only when experts use the code and have a through knowledge of the code [47]. If an inexperienced user tends to work on the code it will lead to errors.

The number of programmers who work in open source software is so large that the ideas that are input to open source software are vast. All these ideas may not be helpful and useful to all the users and this leads to wastage of memory and poor performance in the hardware. The number of people who have worked on the Linux kernel since 1993 can be viewed in the diagram below [45, 47].

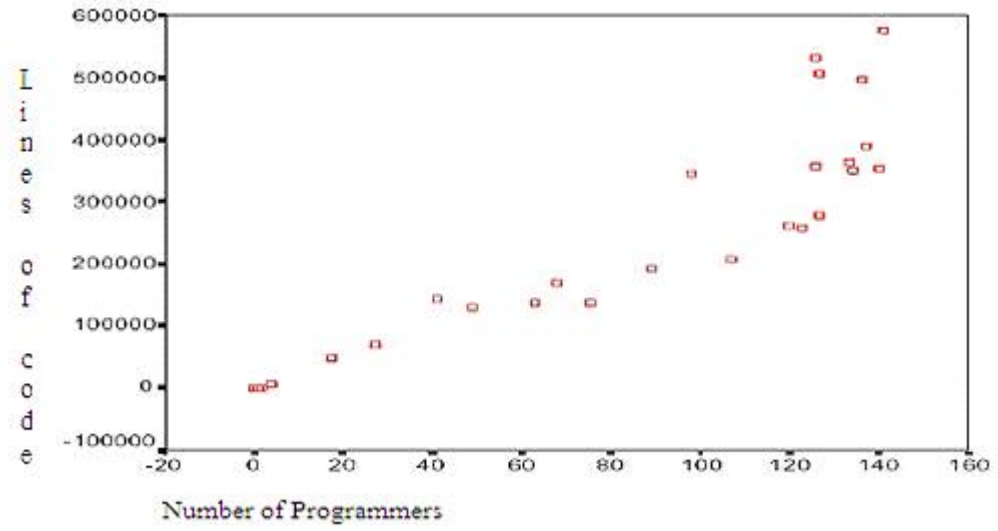

Figure 13: Growth of the lines of source code added as the number of software developers contributing code to the GNOME user interface grows [45, 47].

From this what one can infer for the Infinity Model is that a code developed for open source software should be created with maximum compatibility and modularity. This will help the programmers who tend to work on the code to be able to make changes easily and securely [48]. The diagram below shows us how updating and compaction takes place in the open source software, and how moving the selected updates to the next generation helps the survivability of the code.

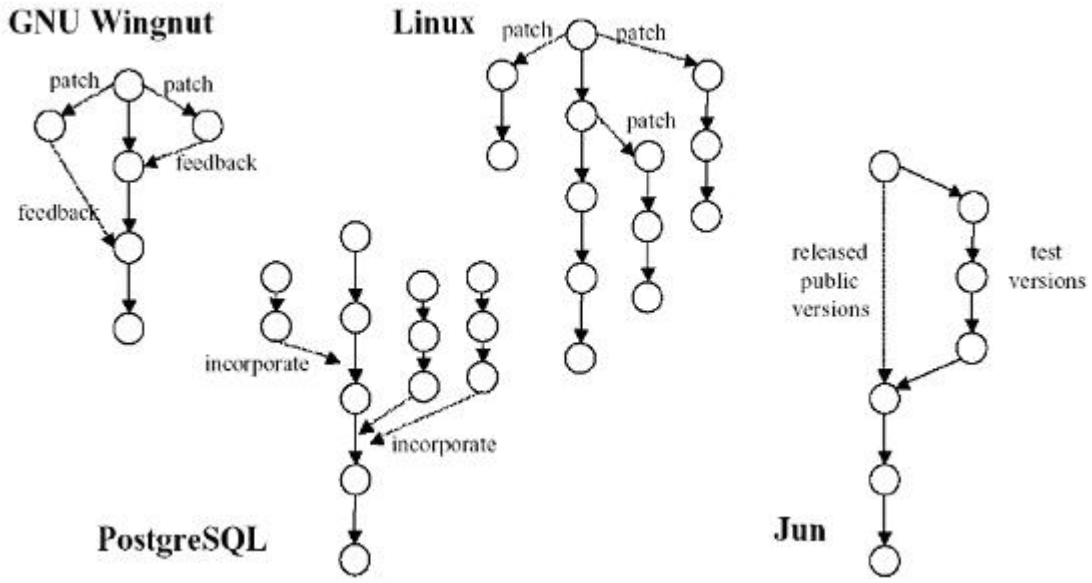

Figure 14: Patterns of software system evolution forking and joining across releases (nodes in each graph) for four different F/OSS systems [45, 48].

The ideas pf modularity and consolidation are added into the Infinity Model. The open source software developers can also use the Infinity Model for the development and consolidation of the software in a more professional and quality oriented way. When software is released in the open source, the documents of how it was developed and the methods of development should also be available so that the others can work on it.

Device Drivers

The device drivers and applications being developed for each and every generation of software are dependent on the OS completely. If the companies new generation do not allow old drivers to work or are not calibrated to accept future growth of drivers, then this leads to a lot of errors and virus attacks [49]. The Windows Vista released in 2007 has similar types of virus attack related problems in the new generation and they are listed below.

2007's Popular Applications with Critical Vulnerabilities

	Software	Version	Vendor's Solution	Nature of Vulnerabilities	CVE* Number(s)
1	Yahoo! Messenger	8.1.0.239 and earlier	Upgrade to 8.1.0.419	Buffer overflow allows remote attackers to execute arbitrary code via unspecified vectors.	CVE-2007-4515 CVE-2007-4391 CVE-2007-3148 CVE-2007-3147 CVE-2007-1680
2	Apple Quicktime	7.2	Patch	Multiple vulnerabilities allow remote attackers to execute arbitrary commands and code crafted URLs and Java applets.	CVE-2007-4673 CVE-2007-2397 CVE-2007-2396 CVE-2007-2393
3	Mozilla Firefox	2.0.0.6	Upgrade to 2.0.0.7 for some fixes	Allows remote attackers to execute arbitrary commands through specially crafted URLs.	CVE-2007-5045 CVE-2007-4841 CVE-2007-3845
4	Microsoft Windows Live (MSN) Messenger	7.0, 8.0	Upgrade to 7.0.08.20 or 8.1	Heap-based buffer overflow allows user-assisted remote attackers to execute arbitrary code via unspecified vectors involving video conversation handling in Web Cam sessions.	CVE-2007-4579 CVE-2007-2931
5	EMC VMware Player (and other products)	2.0, 1.0.4	Upgrade to 2.0.1 or 1.0.5	Allows remote attackers to execute arbitrary code via a malformed DHCP packet that triggers a stack-based buffer overflow or corrupt stack memory.	CVE-2007-0063 CVE-2007-0062 CVE-2007-0061
6	Apple iTunes	7.3.2	Upgrade to 7.4	Buffer overflow allows remote attackers to terminate the application or execute arbitrary code via a music file with crafted album cover art.	CVE-2007-3752
7	Intuit QuickBooks Patch	9 and earlier	Upgrade to 10	Multiple stack-based buffer overflows in the ActiveX control allow remote attackers to execute arbitrary code via unspecified vectors.	CVE-2007-4471 CVE-2007-0322
8	Sun Java Runtime Environment (JRE)	1.6.0_X	Not found	Buffer overflow in Java Web Start allows remote attackers to have an unknown impact via unexpected arguments to a method call.	CVE-2007-5019
9	Yahoo! Widgets	4.0.5 and previous	Upgrade	Stack-based buffer overflow allows remote attackers to execute arbitrary code via unexpected arguments to a method call.	CVE-2007-4034
10	Ask.com Toolbar	4.0.2.53 and previous	Not found	Stack-based buffer overflow in ActiveX control and Ask Toolbar allows remote attackers to execute arbitrary code.	CVE-2007-5107

Figure 15: Application with Critical Vulnerabilities for Windows Vista [49].

The drivers that are developed for the older generation should be allowed to work with the new generation and should not create errors. Windows Vista has a bug with the

drivers created for XP [2, 49]. Most of the drivers which are not preloaded in Vista are not allowed to install or are not saved in the hard disk, and are referred again and again with security violations being stated as the reason. A common issue with driver installation failures is associated directly with the driver package which lacks non-system driver files. In Windows Vista all the driver files have the INF reference. All the other driver files must be imported into the driver store before the package can be installed. Otherwise the files are not imported successfully and the installation fails.

There are problems concerning the installation of class installers and co-installers also. Some of the problems are related to the device installations that occur in an interactive system context [50]. Vista requires class installers and co-installers not to display a user interface with the exception of the finish-install action. Windows Vista also no longer attempts a client-side install [49, 50] in a scenario where the system-based install would return an error code.

These are problems in evolution when one of the companies does not share the information of the development of the new generation to the other partners. This leads the companies to use the old code or style of execution when the latest version of OS does not allow that. In a rush the driver companies tend to create new code with less quality to quickly supply a working driver for the new generation. Changes in generation should not affect people using other packages. In the Infinity Model the principles of symbiosis and co-evolution are used in the architecture evolution phase and the coding phase to avoid such problems.

Legacy System: Department of Defense

Rajlich and Bennett [11, 51] report on the method of software development used in the defense department in the USA, in particular, the problems faced by the company because of evolution and lack of expertise. This has led the company to rethink its strategies and may lead to a new generation developed from scratch. This has occurred due to the lack of change in the systems for a long time, and not deploying new members and techniques to change the code for the later generation.

The various findings of the paper are given below [11]:

1. The defense systems which have been in use for a very long period were developed in assembly language. They require continuous change to adapt to the new hardware.

2. The software is very important as they are all real time and errors or loss in software will lead to a disaster.

3. In the past, experts in both software and hardware had created the system and continuously worked on it. They were trying to improve the system and were trying to free it from ad hoc patches. They documented all the processes and tried to understand the impact of the software.

4. However, in recent times several of the experts has moved out, and there are a lot of decays in the old system when they are updated for the new hardware. There structural decays are a serous problem. The department feels it is impossible to reengineer the system as there are not enough experts and feels if the situation continues they will have to develop a whole new system from scratch.

This is because of the negligence of constant updating to the next generation and also the deployment of human resources [51]. From this case study it becomes clear that updating alone will not suffice, but migration and evolution are also needed for the survivability of a software system. The Infinity Model tries to involve new human resource in each and every cycle of the process. The Infinity model also incorporates the principles of migration whereby every project when it comes into the next cycle, the change in the environment and user requirements are studied.

Evolution in Nature: Lizard evolution

In nature, evolution takes place continuously in a slow but steady pace. All organisms have an inbuilt code called DNA, and all the organisms are constantly mutating at a very slow speed in the micro level. Here in this case study a particular organism is looked into and the mutations that occur on the organism in the given environment are studied.

The organism under study is the lizard and the different environment, in which it survives, differentiates the appearance of the lizards. The experiment [52], provided scientists with important information as they observed what they thought would be the extinction of the introduced lizards. But the lizards adapted to their new environments, and the focus of the experiment changed to studying this rapid evolution. An experiment with lizards in the Caribbean has demonstrated that evolution moves in predictable ways and can occur so rapidly that changes emerge in as little as a decade [52, 53]. The experiment bears on two theories of evolution; one is punctuated equilibrium and the next is gradualism. Gradualism states that evolution is a relatively slow, constant process, producing changes over millions of years [54]. Punctuated equilibrium states that environmental constraints hold species remain unchanged for millions of years, which then undergo rapid evolution when environmental changes demand it.

The experiment involved the introduction of one species of lizard to fourteen small, lizard-free Caribbean islands [52, 54]. The lizards were left for fourteen years. Lizards on Caribbean islands have been carefully studied by biologists for their adaptation to different conditions on different islands with corresponding changes in body shape. One of the important differences in the lizards noted by scientists over the years has been that lizards that inhabit large trees tend to have long legs, whereas those lizards that live on twig-like plants have short legs [53]. The more the vegetation differed from that of their original home the more the lizards should evolve. The scientists had predicted that evolutionary pressure would cause the long-legged lizards to produce short-legged forms as the Caribbean islands are almost treeless. Losos and his colleagues

report in the journal Nature, that the lizards evolved in the direction as predicted [52]. Those with the shortest legs are found on islands with the scrubbiest vegetation.

A long-standing issue in biology is whether micro small evolutionary changes are the same as macro evolutionary changes seen over millions of years. Douglas Futuyama of the State University of New York at Stony Brook, states that while there are many known instances of rapid evolution in biochemistry, such as evolving resistance to pesticide, there are fewer examples of bodily changes. One well known macro evolutionary event is the specialization of lizards on Caribbean islands. Lizards have evolved into 150 different species spread across these islands.

The rate of evolutionary change is measured in units called darwins [52]. Darwins provide a measure of the proportional change in a given organ over time. Changes typically seen over millions of years in the fossil record usually amount to 1 darwin or less. The transplanted lizards evolved at rates of up to 2000 darwins.

From the case study the main idea incorporated into Infinity Model is for the survival of an organism rapid mutation based on environmental conditions is required. Change in hardware should always be studied. Rate of mutation depends on rate of change of environment. In the software world, the environment is both hardware and user. Hence according to hardware updates or user requirements the next generation software should be made available.

The table below displays the various problems faced by the companies and the corresponding methodologies incorporated into the Infinity Model to reduce the problems. These problems can be defined as the requirements to build evolutionary software. If the solutions to these problems are incorporated into the process model then the software created will be more quality oriented. This has been done in the Infinity Model and this can be viewed in the table below.

Case Studies	Properties Incorporated into Infinity Model
Avionics Case Study: Increase in complexity due to increase in the number of requirements.	1. Requirement Evolution 2. Requirement Maturity Index 3. Requirement Collection and Documentation methods
Microsoft Case Study: Release of OS with known errors. Initiating of next generation when the previous generations is released.	1. Servicing and Consolidation techniques 2. Initiation of next generation when previous generation is released
Embedded Systems Case Study: Poor planning mechanisms. No process model or requirement collection methods were followed.	1. Partner or dependent company change collection mechanisms 2. Iterative feedback and requirement cycle
Open source Case Study: The methods of human resource distribution followed by open source programming. Servicing and consolidation techniques. Problems in sharing core parts of the code.	1. Human resource distribution according to projects 2. Core properties security 3. Consolidation in equal intervals of time
Device Driver Case Study: Problems in the partner softwares of Windows Vista.	1. Continuation of familiar techniques to keep user base. 2. Help partner software companies to create quality code to avoid virus attacks
Legacy Systems Case Study: Problems created due to non migration and non involvement of new developers.	1. Migration and consolidation in equal intervals of time 2. Requirements updating 3. Introduction of new human resource into projects to avoid loss of knowledge
Lizard Evolution Case Study: Rapid mutation in Lizards to survive in a complete new environment. Change of body parts in short intervals.	1. Rapid change to new requirements is necessary for survival 2. Change in requirement leads to several new requirements which need to be performed for the survival. 3. Environment is both hardware and use. Change in any one will directly affect the software.

Table 6: Properties incorporated into the Infinity Model from the different case studies

CHAPTER VII

CONCLUSION

By designing the new model we plan to start a new generation of process models. This is also an effort to make people look into nature to find different patterns and methods to create better software. When the understanding of the principles become clearer then the designing of better modes and projects will become more quality oriented.

The Infinity Model is an abstract model and also a unification theory of all existing models. It is designed from the basics of software evolution and also the important principles from evolutionary biology. The main purpose is to give an idea of the measures needed to make evolutionary software in the future.

From the various case studies of existing models the various advantages and disadvantages could be understood and also methods to decrease the disadvantages are tried in the Infinity Model. In the case studies of the various companies and projects the idea of the various changes needed in the existing methods used to design software could be found. The Infinity Model tries to improve the methods in those areas. From the case studies an idea of the advantages of the Infinity Model could be gathered.

The Infinity Model is a step towards creating methods and procedures to produce quality software. This model also includes evolution to be used in the future maintenance and development of the software. The model is a basic idea to create evolutionary software and a model on the time and cost involved to create the software is needed. The model also needs some improvements in the maintenance cycle to accommodate the requirements of different types of companies. A real-world software design is needed to test the effectiveness of the proposed model.

REFERENCES

1. M. M. Lehman, "Rules and Tools for Software Evolution Planning and Management", *Annals of Software Engineering*, Vol. 1, No. 6, pg. 15-44, 1997.

2. T. Mens, S. Demeyer, M. Wermelinger, S. Duccase, R Hirschfeld, M. Jazayeri, "Challenges of Software Evolution", *IEEE*, *Eighth International Workshop on Principles of Software Evolution* (IWPSE'05), pg. 13-22, 2005.

3. W. Scacchi, "Understanding open source software evolution", *Report*, Institute of Software Research, 2004.

4. M. Kim, "Understanding and Aiding Code Evolution by Inferring Change Patterns", *Proceedings IEEE International Conference on Software Engineering*, (ICSE 2007), pg. 101-102, 2007.

5. Luqi, "A graph model for software evolution", *IEEE Software Engineering,* Vol. 16, No.8, Pg. 918-927, 1990.

6. C. Corby, "Introduction to evolutionary biology", www.talkorgins.org, 1996. [Date last accessed : Dec 05, 2007]

7. E. Garfield, "Highly cited articles. 35. Biochemistry papers published in the 1940s". Current Contents No. 8, pg. 5-11, 1977,

8. "Molecular Evolution", http://en.wikipedia.org/wiki/Molecular_evolution , 2007. [Date last accessed : Dec 05, 2007]

9. "Windows Bugs", http://audacityteam.org/wiki/index.php?title=Windows_Bugs, 2007. [Date last accessed : Dec 05, 2007]

10. "Extreme Programming", http://en.wikipedia.org/wiki/Extreme_programming, 2007. [Date last accessed : Dec 05, 2007]

11. V. T. Rajlich, K. H. Bennett, "A Staged model for software life cycle", *IEEE Computer,* Vol. 33, No.7, 2000.

12. L. Yu, S. Ramaswamy, "Software and Biological Evolvability: A Comparison Using Key Properties", *Proceedings IEEE International Conference on Software Engineering*, pg.82-88, 2006.

13. C.L.Nehaniv, J.A.Hewitt, B.Christianson, P.D.Wernick, "What Software Evolution and Biological Evolution Don't Have in Common", *Proceedings IEEE International Conference on Software Engineering,* pg. 58 -65, 2006.

14. G. McGraw, "Software Security*", IEEE Security and Piracy*, Vol. 2, no. 2, pg. 80-83, 2004.

15. "A Survey of System Development Process Models", *CTG.MFA – 003*, Center for Technology in Government, 1998.

16. T. Seifert, M Pizka, "Supporting Software Evolution at the process level", *IEEE Software,* Vol. 20, No. 3, pg. 106-107, 2004.

17. Stuart Anderson and Massimo Felici. "Controlling requirements evolution: An avionics case study". In *Proceedings of SAFECOMP 2000, 19th International Conference on Computer Safety, Reliability and Security*, LNC S 1943, pg. 361–370, Rotterdam, The Netherlands, October 2000.

18. Stuart Anderson and Massimo Felici." Requirements engineering questionnaire", version 1.0, January 2001.

19. Tom Gilb. *Principles of Software Engineering Management*. Addison-Wesley, 1988.

20. IEEE. *IEEE Std 982.1 - IEEE Standard Dictionary of Measures to Produce Reliable Software*, 1988.

21. IEEE. *IEEE Std 982.2 - IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software*, 1988.

22. Ian Sommerville. *Software Engineering*. Addison-Wesley, sixth edition, 2000.

23. Royce, "Managing the Development of Large Software Systems", *Proceedings of IEEE WESCON* 26 (August): 1-9, 1970.

24. Barry W. Boehm. "A spiral model of software development and enhancement". *IEEE Computer*, Vol. 21, No. 2, pg. 61–72, May 1998.

25. Juha Kuusela. "Architectural evolution. In Patrick Donohoe", editor, *Software Architecture, TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, pages 471–478, San Antonio, Texas, USA, 1999. IFIP, Kluwer Academic Publishers.

26. G. T. Laycock: *The Theory and Practice of Specification Based Software Testing*. PhD Thesis, Dept of Computer Science, Sheffield University, UK, 1993.

27. Lawler LP, Pannu HK, Fishman EK, "MDCT evaluation of the coronary arteries, 2004: How we do it— Data acquisition, post processing, display, and interpretation", *AJR Am J Roentgenol*, Vol 184: pg. 1402-1412, 2005;

28. Joseph E. Urban, "Software Prototyping and Requirements Engineering", Report, Rome Laboratory, Rome, NY, 2003

29. Ken Auer and Roy Miller: *Extreme Programming Applied: Playing To Win*, Addison-Wesley. 2005

30. Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

31. Nilesh Parekh, "The Waterfall Model", http://www.buzzle.com/editorials/1-5-2005-63768.asp, 2005. [Date last accessed : Dec 05, 2007]

32. Adrian Als & Charles Greenidge, http://scitec.uwichill.edu.bb/cmp/online/cs22l/waterfall_model.htm , 2003. [Date last accessed : Dec 05, 2007]

33. Phillip A. Laplante and Colin J. Neill, "The Demise of the Waterfall Model Is Imminent and Other Urban Myths", ACM Queue, 2004

34. Adrian Als & Charles Greenidge, http://scitec.uwichill.edu.bb/cmp/online/cs22l/spiralmodel.htm , 2003. [Date last accessed : Dec 05, 2007]

35. ComTech, Disadvantages of Spiral Model, http://inhairstudio.blogspot.com/2007/10/disadvantages-of-spiral-model.html, 2007. [Date last accessed : Dec 05, 2007]

36. C. Melissa Mcclendon, Larry Regot, Gerri Akers, What is Prototyping, http://www.umsl.edu/~sauterv/analysis/prototyping/proto.html, 1999. [Date last accessed : Dec 05, 2007]

37. Keng Siau, www.ait.unl.edu/siau/mgmt454/Chapter6.rtf, University of Nebraska, 2000. [Date last accessed : Dec 05, 2007]

38. Don Wells, Extreme Programming, http://www.extremeprogramming.org/, 2000, 2001. [Date last accessed : Dec 05, 2007]

39. Matt Stephens, Disadvantages of Extreme Programming, http://www.softwarereality.com/lifecycle/xp/safety_net.jsp , 1998- 2007. [Date last accessed : Dec 05, 2007]

40. Matt Stephens and Doug Rosenberg , *Extreme Programming Refactored: The Case Against XP*, Apress, July 2003

41. Stuart Anderson and Massimo Felici, "Requirements changes risk/cost analyses: An avionics case study". In M.P. Cottam, D.W. Harvey, R.P. Pape, and J. Tait, editors, *Foresight and Precaution, Proceedings of ESREL 2000, SARS and SRA-EUROPE Annual Conference*, volume 2, pg. 921–925, Edinburgh, Scotland, United Kingdom, May 2000.

42. Stuart Anderson and Massimo Felici, "Controlling requirements evolution: An avionics case study". In *Proceedings of SAFECOMP 2000, 19th International Conference on Computer Safety, Reliability and Security*, LNC S 1943, pages 361–370, Rotterdam, The Netherlands, October 2000. Springer- Verlag.

43. M.A. Cusumano and R.W. Selby, *Microsoft Secrets,* Simon & Schuster, New York, 1998.

44. Michael A. Cusumano and Richard W. Selby, How Microsoft Competes, http://www.trudelgroup.com/bookr2.htm, Research Technology Management. Pg. 26-30, 1997

45. Walt Scacchi, "Understanding Open Source Software Evolution", *Proceedings IEEE Software Engineering Workshop '06,* Pg. 47-58, 2006

46. J. Erenkrantz, "Release Management within Open Source Projects", *Proc. 3rd. Workshop on Open Source Software Engineering,* 25th Intern. Conf. Software Engineering, Portland, OR, May 2003.

47. C. DiBona, S. Ockman and M. Stone, *Open Sources: Voices from the Open Source Revolution*, O'Reilly Press, Sebastopol, CA 1999.

48. M.W. Godfrey and Q. Tu, "Evolution in Open Source Software: A Case Study", *Proc. 2000 International Conference on Software Maintenance* (ICSM-00), San Jose, California, October 2000.

49. "Top Ten Worst Windows Applications", http://slashmyblog.blogspot.com/2007/11/top-10-absolute-worst-windows.html, 2007. [Date last accessed : Dec 05, 2007]

50. Ron Schenone, "Microsoft Windows Vista - Driver Problems Still a Problem", http://www.lockergnome.com/blade/2007/02/19/microsoft-windows-vista-drivers-problems-still-a-problem/, 2007. [Date last accessed : Dec 05, 2007]

51. George Stark, Al Skillicorn, and Ryan Ameele. "An examination of the effects of requirements changes on software releases", *CROSSTALK the Journal of Defense Software Engineering*, pg. 11–16, December 1998.

52. Dr. George Johnson, “DNA and Darwin: Evolution repeats itself in Caribbean lizards”, http://www.txtwriter.com/Onscience/Articles/losos.html, 2001. [Date last accessed : Dec 05, 2007]

53. Allen E Greer,”Lizards - How They Evolved and Lost Their Limbs”, http://www.amonline.net.au/factSheets/lizards.htm, 2003. [Date last accessed : Dec 05, 2007]

54. Jeff Poling, “Lizard experiment suggests rapid evolution”, http://www.dinosauria.com/jdp/evol/lizard.html, 1997. [Date last accessed : Dec 05, 2007]

APPENDIX

A. Biological Evolution: In biology, evolution is the change in the inherited traits of a population from generation to generation. These traits are the expression of genes that are copied and passed on to offspring during reproduction. Mutations in these genes can produce new or altered traits, resulting in heritable differences between organisms. New traits can also come from transfer of genes between populations, as in migration, or between species, in horizontal gene transfer. Evolution occurs when these heritable differences become more common or rare in a population, either non-randomly through natural selection or randomly through genetic drift.

B. DNA: Deoxyribonucleic acid, or DNA, is a nucleic acid that contains the genetic instructions used in the development and functioning of all known living organisms. The main role of DNA molecules is the long-term storage of information and DNA is often compared to a set of blueprints, since it contains the instructions needed to construct other components of cells, such as proteins and RNA molecules.

C. Mutation: A Mutation occurs when a DNA gene is damaged or changed in such a way as to alter the genetic message carried by that gene. A Mutagen is an agent of substance that can bring about a permanent alteration to the physical composition of a DNA gene such that the genetic message is changed.

D. Natural Selection: Natural selection is the process by which favorable traits that are heritable become more common in successive generations of a population of reproducing organisms, and unfavorable traits that are heritable become less common. Natural selection acts on the phenotype, or the observable characteristics of an organism, such that individuals with favorable phenotypes are more likely to survive and reproduce than those with less favorable phenotypes.

E. Genotype: Genotype describes the genetic constitution of an individual that is the specific allelic makeup of an individual, usually with reference to a specific character under consideration. It is a generally accepted theory that inherited genotype, transmitted epigenetic factors, and non-hereditary environmental variation contribute to the phenotype of an individual.

F. Phenotype: The phenotype of an individual organism describes one of its traits or characteristics that is measurable and that is expressed in only a subset of the individuals within that population. Examples include "blue eyes", or "aggressive behavior".
G. Symbiosis: The term symbiosis can be used to describe various degrees of close relationship between organisms of different species. Sometimes it is used only for cases where both organisms benefit; sometimes it is used more generally to describe all varieties of relatively tight relationships.

H. Co-Evolution: In biology, co-evolution is the mutual evolutionary influence between two species. Each party in a co-evolutionary relationship exerts selective pressures on the other, thereby affecting each others' evolution.

I. Genotype-Phenotype Mapping: The genotype-phenotype distinction must be drawn when trying to understand the inheritance of traits and their evolution. The genotype of an organism represents its exact genetic makeup, that is, the particular set of genes it possesses. The term "genotype" refers, then, to the full hereditary information of an organism. The phenotype of an organism, on the other hand, represents its actual physical properties, such as height, weight, hair color, and so on. It is the organism's physical properties that directly determine its chances of survival and reproductive output. The mapping of a set of genotypes to a set of phenotypes is sometimes referred to as the genotype-phenotype map.

VITA

Murugappan Ramanathan

Candidate for the Degree of

Master of Science

Thesis: A NEW SOFTWARE PROCESS MODEL DESIGNED FROM THE BASICS OF EVOLUTIONARY BIOLOGY AND SOFTWARE EVOLUTION

Major Field: Computer Science

Biographical:

Personal Data: Born in Coimbatore, Tamilnadu, India.

Education: Graduated from G.R.T Mahalakshmi High School, Chennai, India in May 2001; received Bachelor of Technology in Information Technology from Anna University, Chennai, India in May 2005. Completed the requirements for the Master of Science degree in Computer Science at Oklahoma State University, Stillwater, Oklahoma in December, 2007.

Experience: Graduate Assistant, Oklahoma State University, College of Engineering and Architecture, January to December 2006; Teaching Assistant, Oklahoma State University, Department of Computer Science, January to July 2006 and as a Graduate Lab Assistant, Oklahoma State University, Department of English, January 2007 to present.

Name: Murugappan Ramanathan Date of Degree: December, 2007

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: A NEW SOFTWARE PROCESS MODEL DESIGNED FROM THE BASICS OF EVOLUTIONARY BIOLOGY AND SOFTWARE EVOLUTION

Pages in Study: 78 Candidate for the Degree of Master of Science

Major Field: Computer Science

Scope and Method of Study: The process of software development is achieved by using different software life cycle models to design, code and test the software. Process models like the water fall model, spiral model and prototyping are used by companies. Most of these models were designed for a single generation of software. In this research, methods to correct the problems in existing models are proposed based on the principles of evolution in biology and biochemistry, and an abstract model has been generated. The model is called the Infinity Model. The basic principles of biological evolution have been incorporated into the varying steps in the Infinity Model to generate an evolutionary process model. It consists of a completely new design cycle which incorporates both the creation of software and the maintenance of software. In this model, methods to correct deficiencies like resource allocation, documentation and requirement updating in the existing models have been incorporated. Several case studies of large company software and the problems they faced were studied. From the case studies several methods like requirement evolution, consolidation and architectural evolution have been incorporated into the Infinity Model.

Findings and Conclusions: The Infinity Model is an abstract, unification model. It improves the quality and survivability of the software. It incorporates ideas from several models to create an evolutionary model for software. Using the Infinity Model different types of companies can create quality oriented evolutionary software.

ADVISER'S APPROVAL: Dr. Johnson Thomas