CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

Computer applications over time have evolved to play a progressively significant role in individuals’ lives. Predictably, there is always a demand by end-users for bigger, better and faster applications. This invariably has caused an increase in application complexity, which is rather unfortunate for application developers. This rise in complexity in turn has led to an increase in the time to develop these applications. But in an unwanted irony, these demands for improved applications come with a very short deadline and require that the applications be launched in the market at earliest (Yang & Tian 2012).

Evidently, the market forces, which drive the early deadlines for these ameliorated applications, find themselves a challenge in the development teams’ need for longer time frames to create them. This necessitates tools to assist a developer or development teams in shortening the development cycle. An example of such an application development tool is known in the trade as computer code generator. These generators are capable of generating source codes, which can then be compiled or interpreted as per operating environment requirements (Nelson 2005).

Reduction of development and maintenance time sits right at the top among other crucial goals for software developers. Automatic code
generation, Generative Programming and Machine Learning are the approach to achieve this goal.

This chapter is a review on automatic code generation, which talks about how it facilitates the software development process and the current drawbacks one faces in it. In addition, this chapter also introduces generative programming, presents its needs and compares it with object oriented programming, component-based programming. Furthermore, this chapter reviews machine learning and its demands in software engineering. Lastly, the motivation and objectives of the thesis are elaborated in this chapter.

1.2 AUTOMATIC CODE GENERATOR

1.2.1 Automatic Code Generator at Present

In the present scenario, an automatic code generator may facilitate development by capturing knowledge and business rules for an enterprise software application and then generating the millions of lines of codes within seconds in the desired computer language for the desired platform (Macley 2000). Example of automatic code generator include GeneXus, a product of ARTechConsultores S.R.L.

These automatic code generators rely on visual representations of the desired output fed as input to them, based on which they generate the code. For instance, writing computer codes for a Graphical user interface (GUI) such as those found common in most of the Microsoft Windows operating system applications is time consuming. To relieve developers of this mundane task, automatic code generators have been created. These GUI code generators provide a graphical, drag and drop environment to the developers to create GUIs using various built-in interface controls (e.g. radio buttons, drop down list boxes, selection boxes, etc.). After placing all the
required features and controls imitating the final appearance on the graphical canvas, the developer instructs the tool to render the designed GUI and the tool thus generates the necessary code. The code generated may be in any of the various languages supported by the generator (e.g. C, C++, Java, etc.). As is appreciated by those lacking a bit in the tradecraft, the code actually generated comprises of several files with each file containing one or more sub-component. In structured languages, these sub-components may take form of functions, procedures, subroutines et al. Yet in object oriented languages these sub-components may shape up as classes, objects, fields, methods, method bodies (Nirav&Sanjib 2007).

Code generators can also be found in use with Enterprise java beans (EJB). EJBs are objects, usually describing some business logic. An automatic code generator applied to the EJB generates source code objects for the specific environment for which the source code is being deployed. A single EJB input into a source code is capable of spawning several object classes (depending on, for instance, the deployment environment and the bean type) with each object class describing one or more objects and each object comprising of various fields, methods and method bodies(Maki 2007).

1.2.2 Problems with current Automatic Code Generators

In another unwanted irony, automatic code generators, being enterprise-level applications themselves, however, are generated from scratch using standard development tools and programming methodologies and not using automatic code generation. Thus its own development fails to enjoy most of the benefits it aims to provide(David & Shawn 2004).

Another drawback, which current automatic code generators inevitably induce, is the discontinuity in development and maintenance of enterprise software. The generators produce the source code based on the
knowledge and business rules fed to it with no insight about bugs, new features etc. These are then rectified and added manually by the developer resulting in a break between generator output and final application. This discontinuity enforces that subsequent development or maintenance on the enterprise software be done in the traditional manner, without any support from the code generator.

The discontinuity problem is aggravated in the scenario where an automatic code generator is used to produce several different applications for an enterprise, all of which then undergo modifications and changes falling out of the generator’s context. Then, if a new feature is later desired inside each application, it has to be coded independently into each individual application. This results in increased development time and a high possibility of bugs and incompatibility issues between applications (Kenneth & Ogami 2011).

Thus, a desire, for a way for the automatic code generator, to enjoy the very same benefits it aims to provide, stems from these problems. On fulfillment, any enterprise software application generated thereby can safely rely on the code generator to regenerate any computer code that it might require. Furthermore, the ability to supply a new feature added by a developer, automatically across all applications of enterprise software, will be handy indeed.

Presently, code generators exist as a single application and are provided to developers as such, so developers are able to generate codes for the sub-components based on the input they feed. However, the argument, that a single code generator often does not provide optimal code for each sub-component, can be held in agreement (Waqar 2007). Developers often face the problem of unwanted and environment – inappropriate code being generated by the single code generator. The generation of these irrelevant blocks can be traced to identifiable portions within the source code of the generator itself.
But this solution requires that the source code for the generator should be provided to the developer which the code generator vendor would be unwilling to deliver for various reasons. This puts the developer into an unsatisfactory situation (Nelson 2005).

Furthermore, existing code generators’ functionalities cannot be easily modified or extended by the developers unless they are made privy and given access to the source code itself. This, as before, is another difficult situation for the developer.

1.3 GENERATIVE PROGRAMMING

Among the most recent advances in the field of software development, the movement towards automating the software development process has gathered considerable momentum. One particular area of promise and interest is the field of Generative software development (GSD), which is defined in the layman’s terms as “a software development paradigm based on modelling software system families such that, given a particular requirement specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge” (Czarnecki&Eisenecker 2000). GSD is a paradigm whose objective is to automate the software development process, such that software products can be manufactured by component assembly, like other core infrastructure industries producing consumer, mechanical, electrical and other goods have achieved successfully.

Looking back through software development history, software engineering has been aimed at delivering solutions to some specific problems. Hence, dominant methods and tools for software engineering have been designed and implemented for developing single software systems. Recent
developments such as Object-oriented programming (OOP), Model driven architecture (MDA), Object relational mapping (ORM), and code generators have been brought in implementation to tame to growing complexity and scope of single software systems. Predictably, with these resources there has been a tremendous rise in productivity. But alas, it still has failed to catch up with growing expectations and thus developers still struggle under pressure.

In contemporary times, the significance of, and dependence upon software has grown. There have been increasingly varied demands – from entirely new systems to variants of old systems. The rate in which these demands present themselves is on a continuous rise. Thus, even with all the staggering accomplishments, software engineering has been unable to bridge the gap between demands and supply successfully.

The goals of GP are:

- increasing reusability and adaptability,
- improving complexity control,
- managing a large number of variants,
- increasing efficiency (space and execution time).

These goals are achieved through the application of a number of fundamental principles:

**Opening the model implementation:** Each component provides access to its implementation strategies as aspects.

**Providing alternative aspects for a concern:** An aspect is a component implementing a concern of another component. Relevant concerns of a component are modeled as its parameters.
Propagation of aspects: Aspects can be passed from one component to another. They can be propagated from containers to their parts and from parts to their containers. Propagation extends their scope to a set of components, reduces redundancy and facilitates global domain specific optimizations (Eisenecker 1997).

Hiding complexity using layering and configuration rules: Components have some external aspects (features) and a usually larger number of internal aspects. The internal aspects can be derived automatically from the external using configuration rules. Configuration rules are also used to express constraints (i.e. defining which configurations are valid and which are not).

Zero-overhead rule: Overhead in the final product has to be avoided. Domain-specific and general-purpose optimization techniques are used in order to avoid efficiency penalties for good design. The time of binding can be controlled; so static binding is used instead of dynamic binding, if sufficient.

1.3.1 Generative Programming vs. Object Oriented Programming

The object-oriented paradigm marks a significant improvement in software engineering. Some of its beneficial donations include:

Improved comprehensibility and complexity management: Some cognitive science theories establish that humans organize knowledge into sets of interrelated concepts (which are modeled as semantic nets or frames (Minsky 1975). Since such constructs can be morphed into models quite adequately, the object-oriented paradigm facilitates us in building “natural” and better comprehensible models. Moreover, encapsulation aids in hiding unnecessary details (i.e. it supports abstraction). Lastly, objects can be arranged in inheritance and hierarchical constructs which demonstrate the relationships between corresponding concepts (Collins 1995).
**Extensibility, adaptability, and reusability:** Inheritance and overriding facilitate extension and modification of objects. Polymorphism provides the capability for rendering alternative implementations of components.

**Maintainability:** Encapsulation helps to reduce redundancy and polymorphism enables us to avoid error-prone type switches.

A notable shift in focus in the object-oriented community has been seen where developing a reusable solution for families of applications is being targeted instead of developing single applications from scratch. Class libraries comprising sets of classes to be reused in a larger application are being morphed into frameworks, which allow us to reuse their components, flow of control, and the overall application structure. In addition, frameworks provide sets of substitute components in order to allow for axes of variation (Pree 1995).

It is believed that a number of issues concerning the development of reusable models for application families (including frameworks) are yet to be addressed adequately within the object-oriented community:

- Refined modeling approach for application families until now, only inductive approaches have been proposed, e.g. patterns (Gamma et al 1995), evolution of frameworks (Roberts & Johnson 1996)).
- Adequacy of domain concepts decomposition and specification
- Management of family-oriented development induced additional complexity
- Balancing efficiency penalties and improved comprehensibility, adaptability, reusability, maintainability of design
1.3.2 Generative Programming vs. Component Based Software Engineering

One particular solution, which has gained acceptance throughout industry, is the concept of software reuse. Specifically, the practice of reuse has moved from the low-level (e.g. methods, procedures, etc.) to a higher abstracted level (e.g. reusing objects in OOP). The paradigms of Component-based software engineering (CBSE) and Service-oriented architecture (SOA) mark the latest trends in software reuse strategy. These paradigms have distanced from the traditional notions of software engineering where solution components are tightly integrated and too rigid for rapid change. CBSE’s components and SOA’s services are basically software units equipped with well-defined interfaces and standard communication methods. A component based system, as the name suggests, is made up of several components which themselves may be broken down into sub-components and so forth. These components interact via interfaces, thereby working in tandem and enabling the system to function as a coherent whole. Reusing components from one system to the next however poses a problem when system architectures are dissimilar (Frank & Yannis 2003). Different components across both architectures may share common functionality, yet differ enough to make reuse of one in another impossible. SOA faces the same pitfalls. Although CBSE and SOA provide for software reuse, they are still insufficient on their own to solve the problem of increasing rate of change, system variations and requirement complexity faced by the software industry.

1.3.3 Domain Engineering

Domain engineering (DE) consists of systematic development of a domain model Domain analysis (DA) and its implementation (Domain Implementation). A domain model is the representation of common and variant aspects of a number of representative systems of a domain and the
rationale for variations. Examples of DA and DE methodologies include Organization and domain modeling (ODM), Feature-oriented domain analysis (FODA) (Kang 1990), Domain analysis and reuse environment (DARE) (Frakes et al 1995), and Domain specific software architecture (DSSA) (Taylor et al 1995). A typical DA process comprises of:

**Scoping the domain of interest:** The scoping process deals with business goals, stakeholders, and setting (i.e. legacy systems and standards, existing domains etc).

**Domain concept modeling along with their defining features and typical feature variations:** where a feature is a statement or assertion about a concept or set of concepts.

DA provides a systematic methodology for framework development; however, much work is left to be done in integrating existing DA with OOAD methodologies.

A domain comprises of various types of knowledge, including chiefly:

**Decision knowledge (i.e. competence knowledge):** knowledge defining the domain. All domains have a decision procedure, which utilizes this knowledge (e.g. a set of rules) and determines what is included in the domain and what is not.

**Operational knowledge:** knowledge delineating the base concepts and aspects of the domain.

**Configuration knowledge:** knowledge setting valid combinations of concepts and aspects and delineating aspects, which imply other default aspects in a given configuration. This type of knowledge can be interpreted as a set of *configuration rules.*
**Design knowledge:** has two distinguishable components in it as:

**Simple design rationale:** configuration rules with conflict resolution (e.g. priorities)

**Design history:** a complete mapping between end-user requirements and the system present as a series of intermediate designs obtained by application of formal transformation sets including the record of the used transforms.

**Advanced expertise:** knowledge dealing with heuristics, fuzzy logic and several levels of meta-knowledge.

A domain model can be classified according to the kind of knowledge it contains. Typical object oriented frameworks model the operational knowledge (in form of base-level classes). They might also contain some configuration knowledge, which is often represented implicitly in some methods. In some cases, the configuration knowledge is represented more explicitly in specially designated metacomponents.

In purview of domain engineering, the paradigm of GSD attempts to surmount the limitations posed of traditional software engineering practices. A domain is defined as a set of software systems and features that, as a whole, can be considered a family. Domain scope is defined in terms of technical, marketing, and economic requirements. Domain engineering involves domain analysis, in which a domain scope is established (i.e., a family of systems with common and variable features and dependencies between the variable features); domain design whose purpose is to develop a common architecture for a system family; and domain implementation which involves implementing domain components, generators, and the reuse infrastructure.
The paradigm of GSD targets domain implementation, such that systems or components may be fashioned automatically from domain-specific knowledge supplied by domain analysis and design. GSD endeavors to enable this automation through problem space definition, configuration knowledge specification and defining a solution space. The problem space is concerned with problem domain. The problem space is a set of domain-specific abstractions that enables engineers to communicate in a way deemed natural to define the domain. The problem space contains the components available to define a domain and all their various combinations of systems. The solution space represents a specific solution implemented via domain specifications. Configuration knowledge, which is domain-specific, maps the correct combination of components and features from the problem space to the solution space. In the purview of GSD, generative tools such as code generators ideally utilize the domain-specific configuration knowledge and automate the production of components within a solution space. However, in real time, prior art code generators provide such automation only on a small scale (i.e., at the solution level) and are inextensible to a family of solutions such as a domain.

Where conventional software engineering endeavors to satisfy the requirements for a single system, domain engineering instead, strives to provide reusable components and solutions for entire families of systems within a problem domain. A domain can be seen analogous to a product, and domain engineering to product-line development. In this context, a product line is a family of systems.

Variance and the rate of change in requirements is understandably a core concern when speaking of product lines and product families. The ability to spawn diverse systems from a common parent domain demands effective management and realization of variations. Degree of variability measures the capability to support changes in a system. A low degree of variability
indicates a rigid system, while a high degree of variability indicates a more nimble system. Variations between products in a product line can be determined throughout the software lifecycle, which typically includes requirements definition, analysis, design, implementation, compilation, linking, installation, and execution. A software development process where variations are dealt with late in the software lifecycle is successful in achieving nimbleness. The later a variation is acted upon, more the nimbleness of the system. Accounting for the demand to satisfy the requirements of increasing software variations and ever higher rates of change, defining system variations as far back in the software lifecycle as possible is now an accepted norm in the industry.

Domain engineering is a paradigm that seeks to exploit commonalities among a family of systems while managing the variations among them in a systematic way. When applied to product-line engineering, this paradigm endeavors to create new product variants based on a common set of reusable assets such as a common architecture, components, models, and an effective development processes. These reusable assets may be deployed within the purview of CBSE or SOA, thus providing reusability within a domain. By applying the paradigm of GSD, reconfigured components can be manufactured based on domain-specific configuration information, where the configuration information serves to configure the variation points in a system. Table 1.1 shows the classifications of various generative approaches and Figure 1.1 portrays the spectrum of domain engineering technologies.
Table 1.1 Classification of generative approaches

<table>
<thead>
<tr>
<th>Approach</th>
<th>Configuration Time</th>
<th>Kind of Design Knowledge</th>
<th>Kind of Optimization</th>
<th>Concerns</th>
<th>Coordination Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aspect-Oriented Programming</td>
<td>Static</td>
<td>???</td>
<td>local and global static optimizations</td>
<td>any</td>
<td>Any</td>
</tr>
<tr>
<td>Subject-Oriented Programming</td>
<td>static and dynamic</td>
<td>constraints, configuration rules, configuration rules</td>
<td>??</td>
<td>any</td>
<td>object identity and composition rules</td>
</tr>
<tr>
<td>Metaobject protocols</td>
<td>dynamic (and static)</td>
<td>not specified</td>
<td>local and (global ??) static and dynamic optimizations</td>
<td>any</td>
<td>Any</td>
</tr>
<tr>
<td>(Chiba 1995)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generative Programming</td>
<td>static &amp; dynamic</td>
<td>constraints, configuration rules</td>
<td>local and global, static and dynamic optimizations</td>
<td>Any</td>
<td>type parameterization (static) and delegation (dynamic)</td>
</tr>
</tbody>
</table>
1.3.4 Code Generation in GSD

GSD derives much of its power from code generation technology. It is through implementation of domain-specific configuration knowledge via generators, that the full potential of GSD is unleashed. Unfortunately, the craft of code generation has been limited to single system engineering and thus, in its current state, generates specific systems to solve specific problems. Resultantly, these methods and techniques when used in GSD provide solutions in terms of software components, which are limited in flexibility and variance (Maki 2005).

1.4 MACHINE LEARNING

Machine learning deals with issues of how to build programs that undergo self-improvement through experience at some task. Machine learning algorithms have been effective in solving problems in a variety of application domains. They are particularly useful for (a) poorly understood problem domains where little knowledge exists for the humans to develop effective algorithms; (b) domains where there are large databases containing valuable implicit regularities to be discovered; or (c) domains where programs must adapt to changing conditions. So it hardly comes as a surprise that the field of software engineering is a fertile ground where several development and
maintenance tasks can be fashioned as learning problems and can be solved through learning algorithms (Zhang & Jeffrey 2003).

The amount of literature published on challenges faced during developing and maintaining large software systems in a transient environment is quite elaborate (Brooks 1987). Even in the contemporary scenario, in developing large software, one faces difficulties in these four quintessential areas – complexity, conformity, changeability, and invisibility. There have been various attempts, with the goal of evolutionary or incremental improvements, which addressed some aspect of the mentioned difficulties (Boehm 2000; Ernst et al. 2001; Lowry 1992; Parnas 1979).

However, to overcome the difficulties in these quintessential areas, formalization and automation of both the processes and products of software development has been zeroed in, and in achieving this, AI especially comes in handy. Specifically, in the past 20 years, Machine-learning (ML) methods have been integrated in software development. ML addresses the issue of developing computer programs that self-improve at some task through experience (Mitchell 1997a). Its commitment lies in the creation and compilation of verifiable knowledge related to artifact design and construction (Provost & Kohavi 1998). ML algorithms pose viable substitutes to resolve several software development issues as opposed to existing approaches.

Notwithstanding the various results in ML & Software Engineering (SE) published in the past decades, there have been very few efforts at summarizing the state of implementation and its issues in applying ML to SE (Zhang 2000). Menzies (2001) narrows down on decision tree based learning methods to SE issues. Another sketch is put up from the perspective of data mining techniques being applied to software process and product.
1.4.1 Overview of Machine Learning

ML algorithms have been employed to solve many problems across different domains. Some typical applications include: data mining problems where large database exhibit valuable implicit regularities that can be identified automatically; sparsely understood domains where lack of knowledge hinders development of effective algorithms; or domains where programs need to continually morph to cater to changing conditions (Mitchell 1997b). Following list of publications and web sites offers a good initiation to the interested reader to study the current state of integration of ML applications (Aha 2013; Bergadano&Gunetti 1995; Bratko&Muggleton 1995; Langley & Simon 1995; Menzies 2001; Michalski et al 1998; Mitchell 1997a; Mitchell 1997b; Mitchell 1999; Quinlan 1990; Sutton &Barto 1999; Green 1986).

For efficient use of ML methods as tools to resolve real world SE issues, a clear understanding of both the problems, and the tools and methodologies utilized is required (Jonathan 2002). It is imperative that we know (1) available methods, which are applicable, (2) features of those methods, (3) situations in which these methods are most effective, and (4) their theoretical corroborations. Since solutions to a given problem can often be evinced (or approximated) as a target function, the problem solving process (or the learning process) comes down to determine how to find a function which best describes the known and unknown cases or phenomena for a given problem domain. When learning a target function (or a set of possible target functions) from training data, the following issues are encountered:

- Characteristics of learning process. Learning can occur in a supervised or unsupervised fashion. Different methods may
contain different inductive bias, search scheme, guiding factor in search, and need with respect to the availability of a domain theory. A target function’s generalization can be either eager (at learning stage) or lazy (at classification stage), and its approximation can be incurred either locally or globally with respect to a set of training cases. Learning can produce two outcomes — knowledge augmentation or knowledge (re)compilation. Based on the kind of interaction between a learner and its environment, learning is differentiated as query learning and reinforcement learning.

- Representations. Various learning methods may adopt different representation formalisms for the data and knowledge (functions) to be learned. In some methods, the target function is not explicitly defined.

- Properties of training data and domain theories. Data accumulated for the learning process differs in terms of quantity as being small or large, and in terms of random errors as being noisy or accurate, and also varies in valuations.

Learning methods can differ in criteria regarding training data, with some methods demanding huge data, others highly sensitive to quality of data, and still others requiring both training data and a domain theory. Additionally, the domain theories’ quality (correctness, comprehensiveness) will directly impact the outcome of analytical learning methods. Finally, depending on the way by which training data generation and its provision to learner occurs, there are batch learning and on-line learning.

- Theoretical corroborations and practical considerations. Corroborating learning methods can be done using various justifications: statistical, probabilistic, or logical.
• Target function output. With respect to the output, learning problems can be classified into binary, multi-value and regression.

There are numerous types of learning methods with their own characteristics and their effectiveness limited to certain learning problems. Mitchell (1997a) categorizes the major types of learning methods into the following: Concept learning (CL), Decision tree (DT) learning, Artificial neural network (ANN), Bayesian learning (BL), Reinforcement learning (RL), Genetic algorithms (GA) and Genetic programming, Instance-based learning (IBL), of which Case-based reasoning (CBR), is a popular method), inductive Logic programming (ILP), Analytical learning (AL), of which Explanation-based learning (EBL), is a method, and combined Inductive and analytical learning (IAL) and each of these has an associated algorithm.

1.4.2 Current usage of Machine Learning in Software Engineering

Many areas in software engineering development have already exacted benefits of machine learning algorithms. This section firstly examines the reported results and offers classification of the existing work, secondly analyses the current state of practice in this niche area, and lastly discusses some general issues in ML & SE.

1.4.2.1 Classification of existing work

The classification done below is based on activity types to group ML method applications in SE tasks.

**Prediction and estimation:** In this group, ML methods are used to predict or estimate: (1) software quality, (2) software size, (3) software development cost, (4) project or software effort, (5) maintenance task effort, (6) software
resource, (7) correction cost, (8) software reliability, (9) software defect, (10) reusability, (11) software release timing, and (12) testability of modules.

**Software size estimation:** Neural Network (NN), used by Dolado (2000) to validate the component-based method for software size estimation.

**Software quality prediction:** NN is used by Evett et al (1998) to generate software quality models that take software metrics collected earlier in development as input, and goes on to predict the number of faults that will be discovered later in development or during operations for each module. A comparative study to evaluate several modeling techniques for predicting quality of software components is done by Lanubile&Visagio (1997). Among them is the NN model. Another NN based software quality prediction work, as reported by Hong & Wu (1997), is language specific, and in which design metrics for SDL (Specification and Description Language) are first defined, and then used in building the prediction models for identifying fault prone components. NN based models facilitate faults’ and software quality measures’ prediction (Khoshgoftaar et al 1995; Khoshgoftaar et al 1997). CBR is the learning method used in two separate software quality prediction efforts. Porter & Selby (1990) used a DT based approach to generate measurement-based models of high-risk components. Another DT dependent approach is used to build models for predicting high-risk Ada components (Briand et al 1992). Another comparative study result is reported by Cohen &Devanbu (1997) on using ILP methods for software fault prediction for C++ programs. Software quality prediction is formulated as a CL problem by Almeida &Matwin (1999). The proposed approach is applied to a set of COBOL programs.

**Software (project) development effort prediction:** IBL techniques used by Shepperd& Schofield (1997) predict the software project effort for new projects. Another CBR application in software effort estimation is reported by
Vicinanza et al (1990). DT and NN are used (Srinivasan & Fisher 1995) to help predict software development effort. Additional research on ML based software effort prediction contains: a genetically trained NN predictor (Shukla 2000), a comparative study of software effort estimation techniques in that is based on NN and CBR.

**Software cost prediction:** A general approach, called optimized set reduction based on DT, is described by Briand et al (1992) for analyzing software engineering data, and is shown to be an effective technique for software cost estimation. Also, a comparative study done by Briand et al (1999) includes a CBR technique for software cost prediction. The result reported by Chulani et al (1999) indicates that the improved predictive performance of software cost models can be obtained through the use of Bayesian analysis, which offers a framework where both prior expert knowledge and sample data can be utilized to obtain predictions.

**Software resource analysis:** DT utilized in software resource data analysis, identifies classes of software modules that have high development effort or faults (Selby & Porter 1988).

**Correction cost estimation:** An empirical study is done by Almeida et al (1998) where DT and ILP are used to generate models for estimating correction costs in software maintenance.

**Software reliability prediction:** Software reliability growth models can be made use of to characterize how software reliability changes with time and other factors. The models provide mechanisms for estimating current reliability measures and for predicting their future values. The work by Karunanithi et al (1992) reports the use of NN for software reliability growth prediction.
Defect prediction: BL was used by Fenton & Neil (1999) to predict software defects. Though the system reported is only a prototype, it exhibits the potential Bayesian belief networks have in assimilating multiple perspectives on defect prediction into a single, unified model.

Testability prediction: The work reported by Khoshgoftaa et al (2000) describes a case study in which NN is used to predict the testability of software modules from static measurements of the source code.

Software release timing: How to determine the software release schedule is an issue that concerns both the software product developer, and the user and the market. A method, based on NN, was proposed by Dohi et al (1999) for estimating the optimal software release timing.

Maintenance task effort prediction: Models are generated according to NN, DT, and regression methods, for software maintenance task effort prediction (Jorgensen 1995). The study measures and compares the prediction accuracy for each model.

Reusability prediction: Predictive models are built through DT (Mao et al 1998) to verify the impact of some internal properties of object-oriented applications on reusability.

1.4.3 Relating ML Algorithms to SETasks

This section, firstly offers a discourse on some general steps with respect to application of machine learning methods to the real world problems. Then, it goes on to provide a guideline on how to select the type of learning methods for a given task. Due to paucity of space, some examples on how software engineering tasks can be formulated as learning problems and approached using machine learning algorithms are skipped. For details, please refer to Zhang & Tsai (2002).
1.4.3.1 General procedure

**Problem Formulation:** The first step is to formulate a given problem conforming to the framework of a particular learning method selected for the task.

**Problem representation:** The next step is to select an appropriate representation for both the training data and the knowledge to be learned. The representation of the attributes and features in the learning task often tends to be problem-specific and formalism-dependent.

**Data collection:** The third step is to collect data needed for the learning process. The quality and the quantity of the data needed are dependent on the selected learning method.

**Domain theory:** Certain learning methods (e.g., EBL) rely on the availability of a domain theory for the given problem. Acquiring and preparing a domain theory, and working out its qualities, therefore becomes a significant issue that will affect the outcome of the learning process.

**Performing the learning process:** Once the data and a domain theory (if needed) are ready, the learning process can be performed. The data will be divided into a training set and a test set.

**Analyzing and evaluating learned knowledge:** Analysis and evaluation of learned knowledge is an integral part of the learning process. Various practical problems are encountered in many learning methods such as overfitting, local minima, or curse of dimensionality that occur due to either data inadequacy, noise or irrelevant attributes in data, nature of a search strategy, or incorrect domain theory.
**Fielding the knowledge base:** This step is concerned with the usage of learned knowledge (Langley & Simon 1995). The knowledge could be embedded in a software development system or a software product, or used without embedding it in a computer system.

As observed in Langley & Simon (1995), the machine learning methods do not draw their power from a particular induction method, but instead from proper formulation of the problems and from crafting the representation to make learning tractable.

**1.4.3.2 Guideline for selecting learning methods**

The availabilities for data and domain theories (models) in software engineering tasks exist across a wide spectrum. Quantitatively, some tasks may be data-rich whereas others data-poor; qualitatively, available data may range from noisy, incomplete to accurate, adequate. Moreover, the availability of a domain theory for a given SE task may vary from correct and complete, to inaccurate or incomplete, or to nonexistent.

**ML houses two paradigms:** Inductive learning and analytical learning. Inductive learning formulates general hypotheses that fit observed training data. It is derived from statistical inference, requires no prior knowledge, and can fail if there exist scarce data, or incorrect inductive bias. Analytical learning, however, formulates general hypotheses that fit domain theory. It is derived from deductive inference, can learn from scarce data, but can be misled when given incorrect or insufficient domain theory (Mitchell 1997b).

Since the availability and utilization of data and domain theory play a vital role in these two paradigms, we can use data and domain theory as guiding factors in considering the adoption of learning methods.
When a given task is data-rich, methods of inductive learning can be looked at. If there exist a well-defined model for a task, then we can adopt analytical learning methods. Also, these two paradigms can be combined to form a hybrid inductive-analytical learning approach. We can utilize these hybrid methods in situations where both data and domain theory are less than desirable. However, methods of either paradigm will be good candidates if a task has both an adequate domain theory and plenty of data.

The use of such a dichotomy of data and domain theory marks the first step towards the learning method selection process. Additional properties should also be taken into consideration subsequently.

Thus, in this research, attempts are made to solve software engineering issues with machine learning. The research offers a road map on applying machine learning for work-force modularization, followed by cost prediction, requirement engineering and for the maintenance phase. Finally to identify the risk and correctness assessment, a procedure using supervised machine learning is proposed. The research also focuses on the code reusability.

1.5 MOTIVATION OF WORK

In the field of software engineering (Ebert et al 2008; Hillegersberg & Herrera 2007; Hussey & Hall 2008; Prikladnicki & Yamaguti 2004; Leake & Wilson 2001) common issues that affect software project schedules are cultural difference, high number of distributes sites, different knowledge expertise, and domains, many communication dependencies, time zero differences. Traditional methods required a lot of manpower and critical time. Further earlier approaches lack quantitative method that takes into account of variability and complex nature of software engineering.
It is an accepted fact that the future of software engineering lies in automation from the reviews. The automation can be achieved with generative programming, especially automatic code generation. Granted automation is at its infancy stage, to give it a definitive growth, an intensive study is made in automatic code generation in the first phase of the work and the use of machine learning algorithms in software engineering is discussed in the second phase of the work.

1.6 OBJECTIVES OF WORK

The study has the following specific research objectives

- To generate a code from a simple textual class specification containing the class behaviour specified by the Extended Finite State Machine (EFSM), state model. To produce fully executable and correct source code as specified by the state model irrespective of the system.

- To generate generic forms.

- To elaborate on an improved method for real time automatic generic code generation through generative programming by using various models.

- To facilitate real time code generation, a new search algorithm on relational dataset to be proposed.

- To develop a generative paradigm based system to obtain optimization in software production.

- To propose the use of Machine learning algorithms in order to facilitate the software engineering process.
To identify the risk and correctness assessment procedure using supervised machine-learning methods in comparison to traditional methods.

To propose an algorithm using Bi-Layered Graph Model for code reusability and content based filtering.

1.7 CHAPTERS IN THESIS

The thesis is organized in ten chapters with a concluding chapter.

**Chapter 1** is an introduction that gives a brief overview of automatic code generation generative programming and machine learning from the perspective of software engineering aspects. It is a literature review, where previous studies on use of Generative programming in source (automotive) coding generation are reviewed along with the use of machine learning in software engineering process. Further the review identifies limitations in the existing model and provides gap for the present study through which research objectives are developed and provides strong foundation for the future chapters. The motivation of this research work is also explained.

**Chapter 2** discusses the assumptions on which Automatic code generator, ACG was applied basically using Extended Finite State Machine and proposes the new approach and its steps involved. This section provides all assumptions using codes generated. It also focuses on automatic form generation.

**Chapter 3** presents the efficient retrieval of data through generative approach in web based programs, a new approach of improved efficiency of data access through Link Rank has been proposed, which makes data retrieval
very efficient, in turn making the web applications highly dynamic and real
time based.

**Chapter 4** presents the abstract model to reduce the software
complexity by automatically dealing with the most of the problems. It focuses
on a system for automatic code (real time) generation.

**Chapter 5** introduces Generative System, its inputs and outputs.
Further, a newly proposed algorithm on which system can work is discussed.
An algorithm searching an algorithm to produce the code is clearly explained.
Analysis of the algorithm is explained precisely. Finally, the chapter
concludes with the application of new algorithm in a scenario, particularly
search engine and real time code generation.

**Chapter 6** presents an idea of how automatic code generation can
be used efficiently in networking (error correction).

**Chapter 7** proposes how optimization can be achieved using
generative programming. Assumptions used are clearly explained and the
finding of reusability is projected.

**Chapter 8** discusses applications of artificial intelligence in
sequential steps of software engineering particularly, in planning and
modularization, requirement engineering, analysis of product viability, profits
estimation, strategic decision-making, and maintenance strategies. This
chapter applies the Machine learning algorithms to identify the steps based on
the history of similar modules and presents the findings in comparison with
the traditional method. In this chapter the datasets used are obtained from a
reputed company.
Chapter 9 presents the results of newly proposed technique, Neural networks with back propagation in order to identify the risk and assessment detection needs in real time. The proposed model is compared with the traditional method, logistic regression. The proof achieving high precision is clearly discussed in this chapter. The graphs of precision, recall, F-measure, and processing time are plotted and the results are discussed elaborately.

Chapter 10 explains an algorithm using Bi-Layered Graph Model to depict code reusability. In this chapter in addition to the collaborative filtering scheme, content based filtering scheme is been proposed to further refine the result of recommendation.