CHAPTER 1
INTRODUCTION

1.1 SOFTWARE FAILURE

Software is everywhere. It lets us get cash from an Automatic Teller Machine (ATM), make a phone call and drive our cars. An average company spends about 4 to 5 percent of its revenue on Information Technology (IT), whereas companies which are highly IT dependent, such as finance and telecommunications are spending more than 10 percent on it. In other words, IT sector has now one of the largest corporate expenses, outside employee costs. A lot of that money goes into hardware and software upgrades, software license fees, but a big chunk is for new software projects meant to create a better future for an organization and its customers. Out of a total of all IT projects that are initiated, 5 to 15 percent are abandoned before or shortly after delivery, as being hopelessly inadequate. Many others arrive late, are over budget and require massive reworking. Few IT projects, in other words, truly succeed.

1.1.1. Factors for Software Failure

Software project survival is not a miracle or a mere accident. Carefully followed methods and processes can ensure that projects do succeed. Software projects are inherently complex, risky and require careful planning. Proper planning ensures that a project doesn't move away from its targeted goals, while at the same time, customers get a clear definition of the project, know the project status and have a ready access to project deliverables at any point of time. Software failure, for the most part is predictable and avoidable. When a project fails, it jeopardizes an organization's prospects, put in danger national security and stunts the economic growth and quality of life. It is hard to say how many software projects fail worldwide, and how much money is wasted as a result of it. The most common factors cited for project failures are [1 - 9]:

- Unrealistic or unarticulated project goals
- Inaccurate estimates of needed resources
• Badly defined system requirements
• Poor reporting of the project's status
• Unmanaged risks
• Poor communication among customers, developers and users
• Use of immature technology
• Inability to handle the project's complexity
• Sloppy development practices
• Poor project management
• Stakeholder politics
• Commercial pressures
• Inadequate Testing and Quality standards

Out of these several reasons of Software failure, we will focus on three factors: **Requirement Analysis, Complexity and Testing** in the subsequent sections.

### 1.2. SOFTWARE REQUIREMENTS ANALYSIS

Understanding the problem domain and its requirements is key to a successful project. Institute of Electrical and Electronics Engineers (IEEE) defines Requirements as (1) a condition of capability needed by a user to solve a problem or achieve an objective; (2) a condition or a capability that must be met or possessed by a system, to satisfy a contract, standard, specification or other formally imposed document. Somerville and Sawyer in [32] define requirements as a specification of what should be implemented. They are descriptions of how the system should behave, or of a system property, attribute or constraint on the development process of the system.

Requirement engineering plays a vital role in understanding requirements. Requirement engineering process encompasses systematic and repetitive techniques in discovering, documenting, modeling and maintaining a set of requirements for a system as a whole, or specifically for software components of a system. These techniques are systematically followed to derive, validate and maintain systems requirement document. This process is guided by requirements method.
1.2.1 Findings on failures due to Poor Requirement Analysis

There are many studies recently done to quantify cost and causes of software failures [1-9]. Statistics presented in articles [1-4, 6, 7] have shown that, **60% – 80% of project failures can be directly attributed to poor requirements gathering, analysis and management.** In article [6], it is cited that **68% of IT projects fail primarily due to poor requirements.** In [9], it is projected that companies pay a premium of as much as 60% on time and budget, when they use poor requirements practices in their projects. Over 41% of the IT development budget for software, staff and external professional services is consumed by poor requirements at an average, where the company is using average analysts. Sloppy development practices are also a rich source of failure and they can cause errors at any stage of an IT project. Moreover, the costs of errors that are introduced during requirements phase and fixed later in the Software Development Life Cycle (SDLC) increase exponentially [10-12].

1.2.2. Conceptual Modeling for Requirement Analysis

Software process models produce conceptual system models after systematic analysis and documentation of requirements. These conceptual models are an important bridge between analysis and design process. They help in understanding and analyzing requirements to avoid software failures. A conceptual model also called semantic model represents implementation independent ‘concepts’ (entities) and relationships between them. Conceptual model is built (a) to express the meaning of terms and concepts used by domain experts to discuss a problem (b) to clarify the meaning of various usually ambiguous terms that could lead to software failures.

Some of the popular conceptual modeling techniques are Data flow model (DFD) [33], variants of DFD [34-36], Relational model [37, 38], Entity–Relationship (ER) model [39], (Extended Entity–Relationship) EER model, E²R diagram, Higher-Order Entity Relationship Model (HERM), Semantic Database Model i.e. SDM [40] , SOM (Semantic Object Model (SOM) [41], Object Role Modeling (ORM) [42], Conceptual Schema Language (CSL) [43], DATAID-1 data schema integration [44], REMORA methodology
Till early 90’s, Function-Oriented (F-O) methods [33, 34, 51- 58] and Data-Oriented (D-O) methods [39, 59-61] were used to draw in most of the effort from both the communities of researchers and practitioners. One of the major drawbacks of both these categories of software development methods is the paradigm mismatch between analysis and design. An important step towards reducing this mismatch was the invention of Object-Oriented (OO) software development methods. The most important early OO analysis and design methods are Booch method [46, 47], OOSE or Jacobson method [48], OMT or Rumbaugh method [49], Coad and Yourdon method [62] and Wirfs-Brock method [63].

OO Conceptual Modeling provides a number of benefits like modularity, abstraction, information hiding and reusability that are not present in traditional requirement approaches. With respect to software development process also, OO conceptual modeling is better than the traditional approach. In traditional approaches, different conceptual models are used for requirements analysis, design and implementation. For example, Data-flow diagrams are used for analysis, Structure charts for design and constructs of procedural language are used for implementation. In OO conceptual modeling, the same model is used consistently from analysis to implementation. In the next section, we will discuss Object-Oriented Analysis (OOA) and some of its techniques and tools.

1.2.3. Object-Oriented Requirement Analysis
Currently, OO Conceptual modeling techniques are the most popular software development methodology. In OO Conceptual Modeling, a system is modeled as a group of interacting objects. Each object represents some entity of interest in a system being modeled and is characterized by its class, its state (data elements) and its behavior.

In 1997, Grady Booch, Jim Rumbaugh and Ivar Jacobson unified the Booch, OMT and OOSE method to develop UML which became an industry standard, created under the auspices of the Object Management Group (OMG) [50]. UML is a graphical language for
visualizing, specifying and constructing artifacts of a software-intensive system [64 - 67] and has become a de facto standard for OO conceptual modeling. UML plays a very important role in OOA by describing what a system is functionally required to do, in the form of a conceptual model. UML 2.0 provides a set of diagrams for OO Conceptual Modeling. UML and Unified Process use Use Cases as a starting point in OO conceptual modeling.

Use case diagram is a description of a system’s behavior from a user’s perspective. Use case diagrams included in UML have been considered effective in modeling functional requirements of a system, in general [68 - 74] and software component, in particular.

1.2.4. Techniques and Tools used for OOA
A vital decision during OO Conceptual Modeling is to find objects and classes during OOA and then build a conceptual model for a problem domain. In this section, we describe various techniques that have been used in the past to extract components from requirements for building class and object models. Techniques proposed have either used Natural Language Processing (NLP) or employed Use cases to identify classes. Some of these approaches have been automated by building prototype tools.

1.2.4.1 Techniques for OOA
Russell J.Abbott presented a technique to derive data types from common nouns, variables from direct references, operators from verbs and attributes and control structures from their English equivalents [75]. Grady Booch popularized the concept of Russell and used singular nouns and nouns of direct reference to identify objects and plural and common nouns to identify classes [46]. However, this is an indirect approach to find objects or classes and all nouns are not always classes or objects. Some of them refer to entire assembly, subassembly, attribute or a service. Several parsers were built using this approach to extract nouns and noun phrases from large-scale texts [76 - 83].

In [84], authors have used computerized classification systems and thesauri for the purpose of OOA of a valid Slovenian earthquake code. This approach is also not very easy and
straightforward and is burdened with checking candidate classes with a thesaurus and is very much dependent on the structure and completeness of a thesaurus. It is always difficult to find one-to-one mapping between a thesaurus term and a class. In 1991-1992, pioneers like Codd and Yourdan, Shaler and Mellor and Ross identified certain categories like persons, role and organizations etc, which define application domain entities. These categories help experienced analysts to identify classes or objects. This approach only finds tangible objects but fails to identify abstract classes [85 - 87]. Ed Seidewitz and Mike Stark developed a technique in [88] to identify objects, classes and services from terminators, data stores and data flows of DFD. This approach also suffers from several problems like use of data abstraction instead of object abstraction, scattered pieces of objects across several DFD’s and fragmented objects and classes [88]. In [89], authors have presented an integration of several approaches under one head called taxonomic class modeling (TCM) methodology. This methodology is neither tested nor validated by a controlled experiment. It does not also provide any automation support. In [90 - 95], several approaches are presented, based on NLP of textual requirements to extract components of OOA model.

Work done in [96, 97] has presented a Use case-driven development process and its validation. However, it is reported in empirical findings that this technique leads to problems, such as developers missing requirements and mistaking requirements for design. Work in [98] identifies classes from goals of each Use case without descriptions, instead of scenarios. In [99], a set of artifacts and methodologies are used, to automate the transition from requirements to detail design. In [100], a process is proposed for generating formal OO specifications in Object constraint Language (OCL) and Class diagrams from a Use case model of a system through a clearly defined sequence of model transformations. Work in [101] presents a methodology and a CASE tool named Use-Case driven Development Assistant (UCDA) to automate natural language requirements analysis and class model generation based on Rational Unified Process (RUP). UCDA can assist a developer to generate Use-case diagrams, Use-case specifications, Robustness diagrams, Collaboration diagrams and Class diagrams in International Business Machine (IBM) Rational Rose.
1.2.4.2 Tools for OOA
Several authors have used techniques, described in the previous section, to develop automation support for analysts. This section describes some of those tools briefly. MOSYS (A Methodology for Automatic Object Identification from System Specification) tool proposed in [102], presents a new approach to the automatic identification of objects/classes from a system specification for a Distributed Real-Time System (DRTS). RARE (Reference Architecture Representation Environment) tool in [103] is designed to guide analysts systematically through class identification by applying heuristics associated with quality attributes and evaluating the resulting architecture based on relevant static metrics. CM-Builder (Class Model Builder) tool in [104] is an NL-Based CASE tool that uses robust NLP techniques to analyze software requirements texts written in English, and constructs either automatically or interactively with an analyst, an initial UML Class Model. Another tool called Linguistic assistant for Domain Analysis (LIDA) in [105] provides linguistic assistance in the model development process in identifying objects, their attributes and methods. OOExpert is another tool described in [106] that provides a methodology for object identification and refinement from software requirements based on object-based formal specification (OBFS). AURA (Automated User Requirements Acquisition) is a tool described in [107] that uses an end user’s domain knowledge in analysis of requirements. AURA uses a question-and-answer model to guide end users in describing their problem. GOOAL (A Graphic Object Oriented Analysis Laboratory) tool described in [108, 109], produces both static and dynamic object models from natural language description of problems by capturing cognitive schemes that analysts use to build their models of the world, through use of a precise methodology. Work done in [110 - 113] presents automated approaches to extract elements of OO system (namely classes, attributes, methods and relationships between the classes, sequence of actions, the use-cases and actors) using NLP techniques.

1.2.5. Limitations of existing Object-Oriented Requirement Analysis Techniques
The techniques described in the previous section use long descriptive requirements expressed in natural language, as a starting point. Apart from limitation of each of the
approaches discussed earlier, a natural language description of requirements often has the problem of incompleteness, inaccuracy and ambiguity. On the other hand, Use Case approaches, although, quite popular are at a deviation from basic OO concepts of visualizing systems in terms of objects for object modeling [69, 114, 115].

Moreover, recently several arguments against Use cases have been cited in literature. In [116], author has cited that UML Use case and Class diagrams have no formal way to capture (a) requirements as atomic units; (b) business context and (c) business axioms arising from business context. In [117], author has cited that Use cases do not alone solve the problem. It is a scenario that specifies concrete sequences of actions for requirements. An overall advantage can be achieved by integration of scenario-based approaches with functional requirements. Even work in [118] has emphasized that Use case based requirements engineering approach can be enhanced by integrating Use case with Scenarios.

In [119], it is cited that Use case approach is ill suited for projects involving data warehouses, batch processing, embedded control software, computationally intensive applications and real time systems. Use cases are also not suitable for specifying requirements of time triggered functions and for systems that involve complex business rules to make decisions. Susan Lilly has presented top 10 Use case pitfalls and problems, based on observation from a number of real projects and also presented their cure [120]. Stephen Ferg in [121] has pointed out that Use case approach focuses only on what happens at a system boundary. The author states that sometimes information needed to design a system remains hidden in Use cases. In [122], it is described that Use case has a problem of invisible scope creep, due to the fact that addition of a single phrase in a Use case description adds many more functional requirements, which need to be implemented without increasing the number of Use cases.

Use Cases have document centric, time consuming and declarative nature. They have problem of invisible scope creep and have inability to differentiate dynamic and static elements of specifications [123 - 127]. Although, improvisation of Use case based
requirements analysis approach has been done [118, 128 - 133] by either automating Use Case based model generation, improving Use Case Templates or enhancing Use Case based analysis with scenarios but these solutions have not proposed any other alternative, their starting point still being Use Cases.

In [128], it has been claimed that event modeling infuses rigor and discipline in Use case modeling by helping analysts in identifying what constitutes a Use case. According to [128, 133], event modeling helps in determining Use cases. It helps analysts create a reasonable first cut list of Use cases – based on how a business is supposed to behave in response to real world events. The vagueness that can accompany Use case modeling can be addressed by employing discipline of event modeling to create Use cases. Thus, one can say that Event modeling complements Use case modeling.

1.3 COMPLEXITY METRICS

Complexity is probably the most important attribute of a software because it influences a number of other attributes such as maintainability, understandability, modifiability, testability and cost [13-17]. Basili defines complexity as a measure of the resources expended by a system, while interacting with a piece of software to perform a given task [15, 201]. [202] distinguishes three kinds of complexities: computational, psychological and representational. The psychological complexity is the only complexity that is perceived by man. This kind of complexity comprehends structural complexity, programmer characteristics and problem complexity. Programmer characteristics are hard to measure objectively, while little work has been done to date, on measures of problem complexity. Structural complexity, instead, has been studied extensively because it is the only component of psychological complexity that can be assessed objectively.

Complexity is accessed by using metrics. The term metric is used in different ways to reflect various concepts, such as a) a standard or unit of measurement, b) synonym to measurement (of a characteristic of an object or activity), c) measured value, d) composite of two or more independent measures, or e) at times, a system of measurement. Metrics may be quantitative or qualitative in nature. Metrics are taken for a variety of reasons; to
measure the quality of a product, to assess the productivity of people involved in the building of the product, to assess the benefits of a new software tool, to form a baseline so that various estimations can be taken care of etc.

Software metrics help to measure some of the irritants encountered in the software development process like a) excessive costs, b) low productivity, c) inadequate quality, and d) lack of standards. Metrics can also be used for measurement that is either based on assessment or on prediction [199, 200].

Many metrics have been proposed for structural complexity and these measure a number of internal attributes of software. These metrics can be divided into intra-module metrics like size metrics, control flow complexity metrics, data structure metrics and cohesion metrics or Inter-module metrics like coupling metrics. Figure 1.1 represents graphically this classification of complexity as described in [203].

Different metrics are proposed for different phases of software development. For instance, function points are used in requirements phase to estimate the size of the resulting system [204, 205]. Similarly, the metrics for cohesion and coupling defined in [206, 207] are used for the design phase. More than 500 metrics have been proposed for the implementation phase. The suite of metrics for OO design [208], Lines of Code [209], Software Science [210] and Cyclomatic Number [211] are helpful instruments for managing software process effectively.
Function Points [204, 205] give a measure of the functionality of the system by identifying and counting the functions that the system has to perform. Work in [205] has shown that function points can be used effectively for estimating effort. [206, 207] proposed a methodology for defining and validating software metrics for high-level designs, either traditional or object oriented. [208] have defined a suite of metrics for object oriented design, which include metrics for cohesion, coupling, complexity, depth of inheritance, number of children and response set. The simplest software metric is the number of lines of code, which specifically includes all lines containing program headers, declarations, and executable and non-executable statements. Halstead measure uses a number of distinct and repeated operators and operands to determine a number of attributes of software, such as program length, volume, level, programming effort [198, 210]. This metric depends on a completed code and has little or no use as a predictive estimating model. McCabe developed a metric based on program control flow and validated experimentally that there is a degree of correlation between the Cyclomatic number and some quantities that surely influence control flow complexity, like reliability [211]. From this experimental work, McCabe derived the empirical rule that the Cyclomatic number of a module should not be
more than 10. Work in [212, 213] has shown that the cyclomatic number is strongly related to the Halstead metrics. Henry and Kafura in [214] identified a form of the fan in - fan out complexity for identification of faulty system components. McClure defined a structure metric called “Invocation complexity” [215], which involves all variables, which control (via conditional or iteration statements) the invocation of the component; and all components which can affect the value of these variables. Woodfield defined a metric based on the concept of “review complexity” [216]. Yau and Collofello defined “stability” measure in [217], which directs attention to the ripple effect, which occurs when a change to one component, necessitates changes in other components. Yin and Winchester have defined some architectural metrics based on analysis of a system’s design structure chart [218]. Douce’s et al in [219, 220] has developed “spatial complexity” metric inspired by cognitive psychology, to characterize the difficulty of understanding a program and thus, allows accurate estimation of the cost of a change. The spatial complexity metric is defined in terms of code spatial complexity and data spatial complexity of software.

1.3.1 Findings on failures due to Complexity

In [8], it is reported that a project's sheer size is also a fountainhead of failure. Studies indicate that large-scale projects, fail three to five times more often, than small ones. The larger a project, the more complexity there is, in both its static elements (the discrete pieces of software, hardware, and so on) and its dynamic elements (the couplings and interactions among hardware, software, and users; connections to other systems; and so on). Greater complexity increases possibility of errors because no one really understands all the interacting parts of a whole or has an ability to test them. Roger S. Pressman pointed out in his book, Software Engineering [18], that "Even a small 100-line program with some nested paths and a single loop, executing less than twenty times, may require 10 to the power of 14 possible paths to be executed." To test all of those 100 trillion paths, he noted, assuming each could be evaluated in a millisecond, would take 3170 years. According to a report published in December 2009 [19], the primary cause of software project failures is complexity. Complexity can create delays, cost overruns and lead to inability in a system to meet business needs.
1.4 SOFTWARE TESTING

Testing is a process of detecting errors, injected into a software during any phase of its development [20, 21]. Testing has been found to consume at least half of the effort expended on total development [22-27]. Testing is a process of executing a program with the intent of finding an error, where the goal is to show the presence of errors and not their absence [28]. Testing activity is aimed at evaluating an attribute or capability of a program or a system. It is a measurement of software quality [29].

According to IEEE/ American National Standards Institute (ANSI), testing is the process of a) operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component [269], b) analyzing a software item to detect the difference between existing and required conditions (that is, bug) and to evaluate the features of the software items [270]. Test is a set of one or more test cases; whereas a test case is (1) a set of test inputs, execution conditions and expected results developed for a particular objective, and (2) the smallest entity that is always executed as a unit, from beginning to end [270]. The set of test cases is called a test suite. The more you test a software artifact, the more immune it becomes to your tests [271].

Several approaches proposed for test case generation can be found in literature and these have been classified into random, path-oriented, goal-oriented and intelligent approaches [272]. Even though varied test case generation approaches are available, Model-Based Testing (MBT) approach has attracted many researchers and research is still being carried out to optimize the generation of test cases with minimum human effort. The next section presents the survey of related work done in the area of MBT using UML models.
1.4.1 Model Based Testing

A wide range of models like UML, SDL, Z, state diagrams, data flow diagrams, control flow diagrams, etc have been used in MBT [ 244 - 268]. In [244], conformance test cases are generated for a communication protocol, modeled in an EFSM (Extended Finite State Machine) by a fault coverage analysis. In [245], authors have described the Finite state Machine (FSM) based test case generation, the fault models for test case generation, fault coverage and three methods for test case generation using FSMs.

Several approaches have used activity diagrams [246 - 253]. In [246], authors have generated test cases based on activity path coverage criterion to cover faults like synchronization faults, faults in a loop. In [247], test cases are generated by comparing program execution traces with the given activity diagram. Test cases in [248] have been generated based on an I/O explicit Activity Diagram (IOAD) model. In [249], a gray-box method is used to generate test cases. In [250], the authors have developed an automated tool called TSGAD (Test Scenarios Generator for Activity Diagrams), which uses adaptive agents to directly generate test scenarios. [251] introduced an approach that captures stores and outputs usage scenarios derived automatically from UML activity diagrams. In [252], the authors have used anti-ant-like agents to directly generate test threads from the UML artifacts. Work in [253] used specification coverage to generate properties as well as design models to enable directed test generation using model checking.

Approaches using Sequence diagrams are described in [254 - 260]. In [254], authors have created message dependence graphs (MDG) from UML sequence diagrams for test case generation. Test cases are generated in [255] by using both sequence diagrams and state diagrams. In [256], the combination of TTCN- 3, as test description language, and UML message sequence charts (MSC) are used to specify and automatically generate test cases for communication protocols and distributed systems. Authors in [257] have used OCL to generate test cases automatically from UML sequence diagrams. Authors in [258] have generated test cases by augmenting the sequence diagram graph (SDG) nodes with different information from use case templates, class diagrams and data dictionary to compose test vectors. In [259], mobile phone applications testing is done by translating UML sequence
diagrams into Labeled Transition Systems (LTSs). In the work [260], authors have generated test cases using a formal operational semantics for a sequence.

Some of the model based testing approaches have used State Diagrams and State Charts [261 - 263]. In [261, 262] test cases are generated based on control and data flow in UML state diagrams. Work in [263] generates test cases from a flattened hierarchy structure Testing Flow Graph (TFG).

A few of them have used more than one specific UML model, taking benefit from the features of each of the model involved [264 - 268]. In [264] activity diagrams are used to generate test scenarios and then class diagrams of each scenario are used to achieve maximum path coverage criteria. Work in [265] has used UML state machines, UML class diagrams and OCL expressions to generate test cases. In [266], both Use case models and State diagrams are used for generating system-level test cases. In [267] authors have transformed Use Cases to UML state model to generate test cases. Finally work in [268] has used an AI (artificial intelligence) planner to generate test cases from test objectives, derived from UML Class Diagrams.

1.4.2. Findings on failures due to Poor Testing

In article [5], according to Gartner Research, “The lack of testing and Quality Assurance (QA) standards, as well as a lack of consistency, often lead to software failure and business disruption, which can be costly.” Gartner also reports that “testing consumes on an average 25% to 50% of the application life cycle and is often viewed as adding no business value.” Results of a survey conducted in June 2010 [30] have also shown that majority of software bugs are attributed to poor testing procedures or infrastructure limitations, rather than design problems. In this survey, fifty-eight percent of respondents pointed to problems in testing process or infrastructure as the cause of their last major bug found in delivered or deployed software, not design defects. A report that has cited six common failures of IT projects [31] has shown that poor testing and absence of proper change management are two main reasons of software failure.
1.5. **TOWARDS AN EVENT-DRIVEN PARADIGM**

Einstein, in his Theory of Relativity, used the term, ‘event’ to signify the fundamental entity of observed physical reality -- a single point in the time-space continuum. Similar to usage of events in modeling physical reality, the basic notion of event is also widely used to model software. Like in OO systems, event objects generated by mouse clicks, button presses and menu selections are passed between Graphical User Interface (GUI) components and business objects, programmed to react to user actions. At the operating system level, creation and destruction events mark beginning and end of life for processes and threads. Similarly, at the network level, events mark sending and receiving of packets, creation of links and acceptance or denial of requests. From a systems perspective, events describe the reality of asynchronous distributed software systems. Thus, one can conclude that events are being used in different contexts and applications. Next, sub sections discuss the various existing applications and role of events.

1.5.1. **Role of Events in Requirements Analysis & Specification, Modeling and Testing**

Work in [134] has iterated the fact that event thinking is an equally important way of modeling system. This section describes some of the approaches that have used events for the purpose of requirements analysis and specification. One classical example is of Event partitioning approach described in [135], that decomposes a system into sub-systems using events that require a planned response. The planned response(s) of a system is captured in an event-response table. Information from Event-Response Table is used for structured analysis of a system to build DFD’s in [135]. Event partitioning approach has also been applied to Real-Time Systems Analysis to model ‘non-event’ events [57].

Work in [136] has claimed that from events and event sequences, OO architecture can be delivered for systems of all sizes. Panel views expressed in [136] are extended in [137] by creating a “Synthesis Method for Object-Oriented Systems Development and Analysis and Design for Client/Server Systems Development”, that blends the two approaches: Event and Object partitioning. Work in [138] has described that events change state of objects, so it is important to understand and model events. It has been found that event modeling can
also be used as the basis for identifying and defining requirements process and data patterns
to capture the processing policy in response to a specific business event [139]. In [140],
requirements specifications are expressed as system’s reaction to events and safety
properties using event calculus. A new event-centric approach has been described for e-
commerce application development to promote easier maintenance and implementation of
specifications [141]. Events are also used to model relevant facts and abstract behavior of
application [142]. Composite events are also constructed from simple events [143].

Events have also played a very important role in modeling, that includes conceptual,
domain, activity, business process and dynamic modeling. Business events documented as
Event table have been used in [144 -146] to model object interaction, develop a metric suite
for domain modeling and for analogical reuse of structural and behavioral aspects of event-
based object-oriented domain models. Events are modeled in terms of UML, as object-
oriented structures that have attributes and associations, operations, state charts and
messages [147]. Event-activity diagrams are used to model information structures and
related activities in information systems [148]. Work in [149] has claimed that modeling
events and entities in the same manner provides substantial benefits. UML language also
distinguishes four kinds of events: (1) call events, which are invocations of operations; (2)
signal events, which are similar to objects, but are limited and intended for an asynchronous
communication between objects; (3) change events, which are a satisfaction of Boolean
conditions; and (4) time events, which are a satisfaction of time expressions [150, 151]. In
[152, 153], Events are used for modeling static and dynamic aspects of Information and
Communication Systems.

The role of events in testing is found in the area of GUI Testing. In [154], an Event and
Message driven Petri network (EMDPN) model and an EMDPN-based interaction graph
are used for class hierarchy testing. Event InterActions Graph (EIAG) is introduced in
[155] to describe test-case generation for concurrent programs. A scalable and reusable
model (called the event-flow model) of GUIs based on their event interactions and
consolidation of various models, built earlier for various aspects of GUI testing, is
presented in [156]. Combined with customized event-space exploration strategies (ESEEs),
the proposed event-flow model is used to quickly generate a large number of GUI test cases, which are effective in detecting GUI faults. Event-based notions and tools are also used to generate and select test cases systematically and introduce a holistic view of fault modeling [157]. A new GUI automation test model is presented in [158] that are based on an event-flow graph modeling to generate test cases in the daily smoke test and to create test cases in a deep regression test. Work in [159] has used events and event sequences to define coverage criteria for GUIs to help determine whether a GUI has been adequately tested.

1.5.2. Events in Complex Event Processing Systems
Event Processing is an emerging area. It started over a decade ago with a collection of research projects in various places. It is now getting into the main stream of enterprise software with applications in variety of industries and products by major software vendors and many start-up companies around the world. The academic roots of event processing originate in multiple disciplines: artificial intelligence, databases, simulation, verification, sensor handling, distributed computing, programming languages, business process management and many more.

The EPTS (Event Processing Technical Society) [160] was launched in June 2008 as a consortium that includes all vendors, some of the relevant academic community, analysts, and customers. EPTS is intended to promote the understanding of the event processing discipline, incubate standards and foster collaboration between industry and academia. An ACM conference, "Distributed Event-Based Systems" (DEBS) recognized as the "flagship" conference of the community, covers all topics relevant to event-based computing.

Complex Event Processing (CEP) described in the book, "The Power of Events" [161] in 2002 is another important motivation. In this book, Dr Luckham has shown how CEP can be used to enhance systems that deal with events. Luckham first identified key challenges faced by today's enterprise information systems and demonstrated the "event-driven" nature of management in the electronic enterprise. He showed how one can use the power of events to automate management without compromising managers' control. Luckham
illuminated fundamental concepts related to events such as events, causality, event hierarchies, event patterns, and rules; he further showed how these concepts can be used to solve key enterprise management and many more.

CEP focuses on the correlation and composition of atomic events into complex (compound) events. CEP is a defined set of tools and techniques for analyzing and controlling the complex series of interrelated events that drive modern distributed information systems. This emerging technology helps Information System (IS) and IT professionals understand what is happening within the system, quickly identify and solve problems, and more effectively utilize events for enhanced operation, performance and security. CEP can be applied to a broad spectrum of information system challenges, including business process automation, schedule and control processes, network monitoring, performance prediction and intrusion detection.

There are a few approaches that have used events for representing system requirements, identifying classes and generating class diagrams and use case diagrams. Critical review of these approaches reveals that there are some drawbacks and a possible scope for improvement. This critical review is presented in the next section.

1.5.3 Techniques for OOA based on Events
Author in [162 - 164] has proposed a new alternative Event-Oriented modeling approach called Behavioral Pattern Analysis (BPA) for Safety Critical System, Railroad Crossing System, Production cell System and Multi-agent System. In this approach, events are considered as primary entities of a model. Author has pointed that a system model should be Event Driven rather than Use case Driven. After all, use comes after creation. BPA approach facilitates the 'Safety Analysis', where one identifies and documents the critical events during the requirements definition stage. Also, explicit representation of the causal and temporal relationships between events enables analyst to do cause effect analysis and reason about any possible failure of a system.
The behavioral pattern proposed in this work has identified missing requirements, addressed failure issues, carried out critical event analysis and established temporal relationship between events, but it has not focused on the issue of generating class diagram specification from requirements. It has rather proposed for improving class diagrams that are already generated from Use cases.

Another approach [165] has used Event partitioning classical work done by Stephen M. McMenamin & John F. Palmer [135], to decompose a system into sub-systems using events that require a planned response. The planned response(s) that the system may carry out when the events occur is captured in an event-response table. Information from Event-Response Table is used to build UML Class diagram and Use case diagram. Authors have modified the event table to include input message, output message, includes, extends, specializes, destination and source fields and also proposed a five step process to build a Use case and Class diagram. A new CASE tool called UML diagrams generator (UMLdg), to automate the proposed approach is also introduced. It builds an event table, builds an event based use-case diagram, i.e. produces the related Use-case(s), actor(s) and relationship(s) for each event, builds an event based Class diagram, and i.e. produces the related class(s) for each event. Furthermore, it generates an integrated Use-case diagram and Class diagram for the whole event table, respectively.

The modified event table presented in this approach lacks in showing trigger and causative relationship among events that can be very useful in building both static and dynamic models of a system. Moreover, the authors have not done any empirical validation of their approach. Neither have they formalized proper rules for generating class diagram specification.

Work done in [166] has focused on recognizing the importance of object identification in business modeling, using the OO development approach. He also recognized the limitation of the lack of consideration for business policies at the modeling stage. In this paper, author has proposed the extension of the Use case modeling approach to include business policies
modeling. Events in Use cases formed the basis for identifying and specifying classes and business rules. A process known as Event Scripting is used to document an event and from it, objects and their relationships are identified. Business rules identified with events are attached to objects as part of their definitions in class specifications. For each event identified in a Use Case, an Event Script is written. Components of event script like Source, Participant Sets, Pre-Event Conditions and changes help analysts to identify classes and objects, their attributes, operations and business rules. These identified components form a class specification.

This approach starts from the core step of identifying Use Cases, documenting each Use Case in Use Case Template, then extracting events from the scenarios of Use Case description, and finally documenting each event using the proposed event script. Then from event scripts, focus is to define business policies and attach them to class specification. This event-based approach, to the best of our knowledge has not received empirical validation of their process. Neither have they formalized any rules for generating class diagram specification nor do they have any automation support for the user.

Authors in [133] have also used event partitioning approach to identify Use cases from event table, which records event, trigger, source, Use case, response, and destination. This approach helps to identify potential Use Cases but has not focused on class diagram generation.

1.6 PROBLEM STATEMENT

From the foregoing discussion, it is amply clear that inadequate planning and specifications, ill defined requirements, poor process of requirement analysis and testing, lack of metrics and measures to compute project’s sheer size and complexity, all together lead to numerous change requests, delays, significant added costs and an increase in the possibility of errors. A good requirement analysis method, proper management of software complexity and proper testing techniques, play a vital role in avoiding software failures. From literature survey, following is concluded:
(i) Natural language based approaches for OOA of requirements have their own inherent limitations. Use case based approaches for OOA have also received several critical reviews. Existing applications and perspectives of events have focused more on the aspect of processing of events. Though few approaches have been proposed to use events for OOA but these approaches either focus on improving Class diagrams that have been already generated from Use cases or generate Class diagrams from event tables without using proper rules. None of the existing event-based OOA approaches, to the best of our knowledge have addressed the issue of automatic extraction of events from requirements or have done any empirical validation of their process for OOA.

(ii) None of the existing event-based OOA approaches, have used meta-modeling concepts to give justification of using events as a starting point in requirement analysis or have formalized any rules for generating class diagram specification and automating support for the user.

(iii) None of the existing software metrics, to the best of our knowledge have used events and their interdependency as the basis for measuring complexity of a system.

(iv) Existing Model-based testing approaches have extensively used UML models. Existing event-based test case generation approaches have considered GUI events only and have not dealt with other types of events like business events, state events or control events that happen in a system. Not much of an effort has been devoted to test the functional requirements of a system, more specifically for event-based systems.

(v) Existing event-based test case generation approaches have used events either from source code, Petri net model, or from GUI applications, so they require the applications or code to be made available, prior to generating test cases. These applications are run on existing GUI automation framework to reverse engineer event-flow of an application. Not much of an effort has been devoted to use events captured directly from the requirements specification.
Therefore, an objective of this thesis work is to propose:

An Event-based systematic approach to:
(A) Develop an improvised process for OOA of requirements.
(B) Generate standardized Class diagram specification based on XML Meta-data Interchange (XMI) standard.
(C) Develop software metrics to measure complexity of a system based on system events at the analysis phase of SDLC.
(D) Generate Test Scenarios for testing functional requirements of a system.

And hence the title:

“An Event-Based Framework for Object-Oriented Analysis, Computation of Metrics and Identification of Test Scenarios”

1.7 ISSUES AND CONTRIBUTIONS

In order to propose an Event-based framework that is able to offer requirements analysis, design, modeling, program construction, testing and support activities of a typical software engineering process, following issues are identified and contributions are made to address the issues:

Issue 1: to identify, describe and document events from textual requirements.

Issue 2: to extract information to build conceptual models (class diagram specification) from requirements specifications for modeling the requirements.

Issue 3: to determine the complexity of a system at an analysis level, so that necessary changes can be made to reduce cost of developing such a system or avoid software failures, prior to design and implementation.

Issue 4: to generate test scenarios and test cases for testing functional requirements of a system.

Issue 5: to model changes in requirements and perform re-testing of the system after incorporating those changes.
**Issue 6:** to provide automated support for all issues described above, in order to reduce time and effort of analysts.

In order to address **issues 1 and 2:**

- An Event-based framework is designed and an Event-Meta Model is proposed for OOA. The proposed Event-based approach is presented in Figure 1.2.
- A methodology is defined to derive Class diagram specification from requirements and its empirical validation is carried out through a controlled experiment and its replication.

In order to address **issue 3:**

- A metric is proposed to compute system complexity from requirement specification using event flows and event interdependencies. The metric has also been evaluated in terms of Weyuker’s properties, in order to guarantee that it qualifies as good and comprehensive.

In order to address **issues 4 and 5:**

- A methodology is proposed (a) to generate event sequence-based test scenarios from Event-flow model (b) to specify and model changes in software requirements, either in terms of addition and/or modification of events or event interdependencies (c) to apply a safe and efficient regression test selection technique on event-flow model, so as to select tests from test scenario that are deemed necessary to validate the modified software.

Finally in order to address **issue 6:**

- A prototype tool is developed to implement the entire event-based framework that provides automation support to extract events from requirements, document the extracted events, derive specification for an analysis-level class diagram, automatically compute complexity of the entire system at analysis level and generate test scenarios from events.
As shown in Figure 1.2, software requirements are taken as input from which events are extracted either manually or automatically. The extraction process gives a list of events. Each event from the list is formally documented using Event Template. Event Templates are further used by Class Diagram Generator and Model Generator. Class Diagram Generator generates the XMI based Class Diagram Specifications whereas Model Generator generates an Event-Flow Model. The Event-Flow Model is fed to Test Scenario Generator and Metric Calculator that automatically generates Test Scenarios and computes the complexity of the system at analysis stage.
1.8 ORGANIZATION OF THE THESIS

The Thesis is organized in seven chapters.

**Chapter 1** This chapter presents an introduction and a detailed Literature review done to identify shortcomings and opportunities for the research work in the area of requirements analysis, modeling and testing. The objective of this chapter is to provide contextual background, identify gaps leading to the problem and discuss the approach proposed to solve the problem.

**Chapter 2** presents the objective and principle behind an event based framework that forms the basis for defining the proposed Event Meta-Model. Structure of an Event Template for documenting events is also discussed. In the light of critical review of Use Case Approach, it also presents a detailed comparison of Event Template with Use Case Template.

**Chapter 3** presents a detailed Event-based methodology and mapping rules for generating standardized Class diagram specification in XMI format, from Event Templates. This chapter also discusses a controlled experiment done to measure user’s perception about perceived ease of use and usefulness of an event based approach in comparison with conventional Noun based and Use case based approaches for OOA.

**Chapter 4** presents model based approach to construct software metric for analysis model. It first presents a detailed process to construct Event-Flow model based on interdependencies among events and then the metric is derived from Event flow model. The metric is termed as ‘Event-Flow Complexity metric’. The proposed metric measures the complexity of system on the basis of event flows and their interdependencies. The metric has also been evaluated in terms of Weyuker’s properties in order to guarantee that it qualifies as good and comprehensive.

**Chapter 5** In this chapter, a model-based testing approach based on events, is proposed for testing functional requirement of a system. The approach presents a process to derive test
scenarios and test cases from Event-flow model. A regression testing approach is also presented with objectives (a) to specify and model changes in software requirements in terms of addition and/or modification of events and (b) to apply a safe and efficient regression test selection technique on event-flow model, so as to select tests from test scenario that are deemed necessary to validate the modified software.

**Chapter 6** describes the prototype tool that has implemented the entire event-based framework in five parts. First part deals with Event extraction. Second part deals with formalization of events. Third part deals with class diagram generation. Fourth part deals with computation of system complexity and the last part helps to automatically generate Test Scenarios and Test cases from a graph-based Event-Flow Model. The automation of the entire Event-based Framework reduces the time and complexity associated with each part.

**Chapter -7** summarizes the work done in the thesis, highlighting the key contributions of the research work. It highlights all the areas where the proposed work can be successfully used. It also suggests scope for future work in this research. In the end, a list of all publications referred is given.