Error Propagation Metrics from XMI

Cihan Varol

Problem Report submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Engineering

Lane Department of Computer Science & Electrical
Engineering

Morgantown, West Virginia University
2005

Keywords: Error Propagation, Software Engineering

ABSTRACT

Error Propagation Metrics from XMI

Cihan Varol

This work describes the production of an application Error Propagation *Metrics from XMI* which can extract process and display software design metrics from XMI files. The tool archives these design metrics in a standard XML format defined by a *metric* document type definition.

XMI is a flavour of XML allowing the description of UML models. As such, the XMI representation of a software design will include information from which a variety of software design metrics can be extracted. These metrics are potentially useful in improving the software design process, either throughout the early stages of design if a suitable XMI-enabled modelling tool is deployed, or to enable the comparison of completed software projects, by extracting design metrics from UML models reverse engineered from the implemented source code.

The tool is able to derive the error propagation of metrics from test XMI files created from UML sequence and state diagrams and from reverse engineered Java source code. However, variation was observed between the XMI representations generated by different software design tools, limiting the ability of the tool to process XMI from all sources. Furthermore, it was noted that subtle differences between UML design representations might have a marked effect on the quality of metrics derived.

In conclusion in order to validate the usefulness of these metrics that can be extracted from XMI files it would be useful to follow well-documented design projects throughout the total design and implementation process. Alternatively, the tool might be used to compare metrics from well-matched design implementations. In either case design metrics will only be of true value to software engineers if they can be associated empirically with a validated measure of system quality.

Acknowledgements

I would like to express my deepest gratitude and appreciation to my research community and the department for giving me the opportunity to conduct research under their supervision.

I am honored to dedicate this paper to all the members of my family, who have encouraged me, and supported me throughout my life. I want to specifically express my love and appreciation to my brother who always helped me to come out from tough times.

Table of Contents

1 INTRODUCTION

XMI provides a standard format for representing UML models of software design, potentially allowing software engineers to archive and exchange models in a tool independent fashion. If a current or future version of XMI becomes widely accepted, and supported by commercial modelling tools, it will be highly desirable to develop freely available tools which can use and manipulate these XMI files. The major practical motivation for the work described in this dissertation was to develop such a tool and demonstrate that it can process XMI files to access software design parameters and calculate Error Propagation Probabilities. Further to this aim, it was hoped to determine whether any such software metrics extracted from the XMI representation would have any value in analysing and improving the software design process.

The first major Section (2) of this report reviews the topic of software metrics, with particular emphasis on definitions of object-oriented metrics and how it is hoped that these metrics may be used to measure how well a system design meets the accepted object-oriented design paradigm and hence to improve design quality. This is followed by a Section (3) detailing the technical background for the work: the salient features of UML notation, XML and XMI. Error Propagation term, the design, implementation and testing of this tool *Metrics from XMI* is described in Sections 4 and 5, Final conclusions from the MSc project are presented in Section 6.

2 LITERATURE REVIEW: SOFTWARE METRICS

Software metrics measure attributes of a software system and may be used to quantitatively express elements of a system model or of program code. A 'metric' may be a direct measure of a particular attribute (for example *Lines of Code* or *Number of Classes*) or, potentially more usefully, an indirect measure of a higher level features of the system, such as *Quality* or *Complexity*. These indirect metrics often express relationships both between the directly observable metrics, and also with external attributes of the system, such as runtime failures or problems (Bennatan, 1995; Fenton and Pfleeger, 1997).

Historically software metrics have been used to assist in both estimating the costs, effort and timescale for the development and maintenance of a system, or alternatively to provide a measure of the quality of the whole system and its individual components. The most common fundamental use of software metrics is to measure or predict system size, which is considered to be the major driver for estimations of system cost or development effort. Size metrics are also used as simplistic measures of a software engineer's productivity and to measure progress of a developing system (reviewed by Hughes and Cotterell, 1999). Software metric is valuable only if it can be shown empirically to be associated reliably with important quantitative or qualitative attributes of the system (Fenton and Pfleeger, 1997).

2.1 Uses for Software Metrics

Software metrics can input into several areas of the software development life cycle (Hughes and Cotterell, 1999):

- Effort or cost estimation: time and resource allocation: project planning
- Improving the design process
- Measuring ongoing project development in terms of specific outputs
- Evaluating the quality of the product, in terms of functionality, faults and design
- Evolving and maintaining the product

These functions are implemented in the following processes

1. Project Management

   Relatively simple metrics such as *lines of code*, or *defect rates* are widely used in industry for managing software development projects. Predictive metrics are used to estimate the effort, timescale and resource requirements of projects; while assessment metrics track progression of a project, as a means of assessing productivity (Fenton and Pfleeger, 1997).

   Simple metrics are favoured for this because of their ease of collation, application and comprehension, although limits to their usefulness are well documented (Bennatan, 1995).

2. Quality Control and Assurance

   Appropriate software metrics can also be used to measure the quality of the software product throughout development and upon completion. These metrics may be simple rates of fault detection, or be more abstract measures of system function and complexity (Bennatan, 1995).

3. Design Process

   Perhaps the greatest unrealized potential of software metrics is in the evaluation and improvement of the design process (Reiβing, 2001).

   Particularly in an object-oriented design environment, the quality of the design is critical for the implementation, structure and quality of the final product. Mistakes and bad choices in the design stage can be difficult and expensive to correct later in development.

   Metrics which can be used to capture high level design concepts and measure their quality have the potential to assist in the design of the overall system, and in identifying potential problem areas during implementation.

2.2 Types of Software Metrics

2.2.1 Direct versus Indirect

Software metric may be a directly derived attribute of the system such as: (thousand) Lines of Code (KLOC), number of errors per KLOC, Direct Source Instructions (DSI) or other, low-level, code-based metrics. An indirect metric has value as a measure of a higher level, abstract property of the system such as quality or complexity (Bennatan, 1995).

Direct metrics can of course be used as indicators of higher level properties, for example the average number of methods per class can be indirectly interpreted as a measure of complexity or quality, when compared to a quality standard or model. Alternatively, more complex metrics may be derived or calculated, often from low-level, direct metrics, in order to capture measures of system complexity (for example, function point analysis, Section 2.3) (Fenton and Pfleeger, 1997).

2.2.2 High Level versus Low Level

Low level metrics are recorded from direct inspection of the code, a process lending itself to automation, but necessarily not available until code is being implemented.

High level metrics focus on the architecture of a design or program. They might be defined on the basis of a design model and available early in the development process, or they may be derived from underlying low level metrics, and dependent on detailed design knowledge and code. Tools also exist to assist in the calculation of certain high level metrics, for example the *Together* UML design tool can extract design metrics from system models throughout development (Together 2002).

2.2.3 Predictive versus *a Posteriori*

A posteriori metrics (Nesi and Querci, 1998) are calculated from completed software projects, where a full range of detailed parameters can be derived from the fully implemented code and design. These metrics are useful for examining the quality of the system, and relating its final properties to earlier, predictive models and estimations. They can be used for the testing and evaluation of the system, and contribute to ongoing evolution and maintenance.

Predictive metrics, derived in the early, pre-coding design phase, or during the course of implementation, can be used both in project planning as effort estimators, and as 'early quality indicators' (Basili *et al*., 1996; Chidamber and Kemerer, 1998). Early prediction is a useful goal, allowing identification of high risk components which will be 'expensive' to implement or error prone (Emam *et al.,* 2001). As an example, a metric indicator of poor design might be 'exceptional class complexity'.

2.2.4 Procedural versus Object-Oriented

During the previous 30 years, a range of software metrics have been evolved for the assessment of software programs developed in functional programming languages (reviewed for example in Hughes and Cotterell, 1999). The sequential nature of 'traditional' software development lifecycle models has meant that these metrics were considered usefully adequate, if not ideal, for project management applications, and have also been useful for some quality control functions.

However, the introduction of object-oriented design and programming has lead to marked changes in working practices. The old models of development lifecycles are less relevant as design and implementation stages overlap and cycle, and as the balance of developers' time have shifted from the implementation of code to the analysis of design. Novel aspects of the object-oriented paradigm such as encapsulation, inheritance, abstraction, coupling and cohesion cannot be captured by the standard existing metrics (Booch, 2000).

Whereas the main cost driver for non object-oriented systems is deemed to be system size, measured by simple low level metrics or higher level estimates of complexity (such as function point analysis) there is a belief that the further structural properties of object-oriented systems will incur additional cost factors; hence new metrics must be derived to represent these. Furthermore, controlling the design complexity of object-oriented systems is considered to be of central importance, and new metrics should be defined which assess the quality of the design (Chidamber and Kemerer, 1994; Marchesi, 1998; Booch, 2000).

2.3 'Traditional' Software Metrics

The simplest code-based, software metrics have been used since the 1960s for measuring productivity and for time, cost and effort estimations. These include variants on (thousands) of Lines of Code (KLOC), delivered source instructions (DSI) and rates of defects per KLOC as a measure of quality. These metrics can be directly measured or statistically estimated. Several models for estimation of system cost or effort prediction use these metrics as inputs including SLIM (Putnam, 1978) and Constructive Cost Models (COCOMO: Boehm, 1981). Even such simple metrics are difficult to count or estimate accurately, and indeed to use meaningfully, requiring expertise and historical datasets with which to calibrate the models.

Metrics which try and estimate complexity are potentially better estimators of effort. Function complexity can be estimated directly from code, for example McCabe's cyclomatic complexity **Mc** (McCabe, 1976) and Halstead's measure **Ha** (Halstead, 1977). If these estimators measure complexity in a programming language neutral form they can be more readily applied and compared across a wider range of design projects.

More abstract measures of function size attempt to provide more useful predictors of system size. Function Point Analysis (FPA: Albrecht and Gaffney, 1983; Symons, 1988) provides a complex method to measure size in terms of functional outputs. System functions are enumerated and weighted according to complexity, and then scaled relative to a system complexity factor. Whilst theoretical and practical criticisms of FPA have been made (it is difficult to calculate, the weighting is complex and somewhat arbitrary, and there is dispute over how to define relevant independent functions and indeed whether weightings are necessary) it can be implemented early in design, and is useful for predictive modelling. FPA has been widely used in certain sectors of the industry (Heiat and Heiat, 1997).

These established metrics are well understood by practitioners and researchers, and there is extensive empirical evidence to support their use in structural systems albeit with limited accuracy. However, it is necessary to calibrate the metrics with relevant historical data sets or to otherwise account for the specific system environment (Kemerer, 1987; Subramanian and Corbin, 2001).

2.4 Object-Oriented Software Metrics

2.4.1 The Object-Oriented Paradigm

Object-orientation has become the predominant model for the analysis, design and implementation of software projects and applications. Object -orientation seeks to model the 'real world', as collections of objects which have attributes (state) and operations (behaviour).

An object-oriented program is based on classes that describe collections of objects and define the 'type' of an object, the properties and behaviour of objects. Object-orientation seeks to provide

- **Modularity:** the program is assembled from components, which can allow re-usability and pluggability
- **Interfaces:** public interfaces describe how components can be used by their clients i.e. their publicly accessible attributes and operations
- **Abstraction:** publicly accessible interfaces of modules hide the complexities of implementation from their clients, allowing pluggability
- **Encapsulation:** modules hide their information from clients, preventing its misuse

- **Minimal Coupling:** the dependency between modules is minimized which allows modules to be maintained and modified independently
- **Optimized Cohesion:** well designed modules provide related functionality, realized by operations acting on shared attributes
- **Inheritance:** Allows components to be extended, so that hierarchies of increasingly specialized components can be created from ancestors (superclasses)
- **Polymorphism:** through inheritance and overriding, attributes and operations have context dependent meaning and behaviour. This allows for late binding. (References Stevens and Pooley, 2000; *etc*.)

Object-oriented development is claimed to provide competitive advantage (facilitating faster development and more flexible products) and may be required for increasingly complex applications (Rational, 2000).

2.4.2 Metrics to assess the object-orientation of software

In the past 10 years a number of groups have developed sets of metrics which seek to capture and quantify the novel structural aspects of object-oriented design and software projects, namely inheritance, abstraction and encapsulation. Metrics defined for object-oriented applications can broadly be divided into system/package level, class level and method level. Method level metrics correspond to the traditional functional metrics discussed above (LOC, Mc, Ha, *etc.*), and to some extent class level metrics may be considered as aggregations of these, with additional parameters reflecting class architecture. However, the higher level package and system metrics seek to represent the uniquely important features of object-oriented design, and as such might be important aids to improving the object-oriented design (Nesi and Querci, 1998).

The work of Chidamber and Kemerer (1991, 1994, 1998) has been seminal in defining, theoretically validating, and to some extent empirically verifying a set of six object-oriented metrics. Their metric set is listed in Table 1, summarizing how each metric is derived, and which object-oriented features they seek to represent. These metrics seek to quantify how well a system meets the object-oriented paradigm, in terms of optimizing inheritance, ensuring encapsulation, minimizing coupling and improving cohesion. The metrics can then be used to judge the quality of a system, and to identify potential error prone elements, such as overly complex classes. To this end the utility of the metrics has to a degree been empirically verified by several studies (Li and Henry, 1993; Basili *et al*., 1996; Chidamber and Kemerer, 1998; Briand *et al*., 2000).

It can be argued that the Chidamber and Kemerer metric set focuses on class level metrics and that several of them are highly dependent on low level (i.e. code) metrics for their derivation, and as such are not ideally suited to early stage design analysis. Furthermore, the metric set may not capture overlapping properties of the system nor are the metrics formally and unambiguously defined. Considering again Table 1 while DIT and NOC can easily be formalized, WMC is somewhat vague in its definition. Futhermore to determine CBO requires detailed design data and RFC and LCOM would require code level analyses (Reiβing, 2001). Never the less, the value of refined versions of these metrics has been demonstrated by studies in which they have accurately predicted poorly designed, error-prone classes (Briand *et al*., 2000). The Chidamber and Kemerer metrics therefore give useful measures of class complexity, however, some evidence suggests that class size is still the most influential (and possibly useful) metric, and current measures of coupling and cohesion fail to markedly improve the value of the metrics (Briand and Wüst, 2001).

Several alternative object-oriented metric sets have been proposed by other workers, which tend to share many of the properties of the Chidamber and Kemerer set, but may be focused at a higher or lower level of the design. Lorenz and Kidd (1994) have defined an extensive set of metrics, which are relatively low level and directly measurable, and hence may give a more limited architectural view (Harrison *et al*., 1997).

Specifically in response to some of the criticisms of the Chidamber and Kemerer metric set alternative, 'early definition' metrics have been proposed which should be obtainable from early and incomplete program designs (Abreu *et al*., 1995; Martin, 1995; Marchesi, 1998, Table 2 and Table 3; Reiβing, 2001, Table 4 Bansiya and Davis, 2002). Operationally these metrics start by defining which direct metrics or parameters are available at an early stage of development, in the absence of code (see for example Table 2) and use these direct attributes to define higher level measures of structural complexity (see Table 3). Marchesi (1998, Table 2 and Table 3) defines sets of measures that are available at the very earliest design stages, *i.e*. class design, whilst others use properties that will emerge as the design is developed (Reiβing, 2001;Table 4). Again these metrics seek to allow the properties of a system to be compared to a quality model, i.e. against heuristic rules which suggest that classes should not have public attributes, coupling should be reduced, inheritance hierarchies have an optimal size, *etc*. As such this early-definition, high-level metrics would possibly reflect the properties of inheritance, encapsulation and coupling. The same metrics sets might then be applicable throughout the development lifecycle and *a posteriori* to measure system and component quality.

Marinescu (2001) has identified an important *a posteriori* use for metrics in the re-engineering of object-oriented applications. He has described a simple set of metrics that can be derived from implemented software projects, which might be used to identify potentially poorly designed classes. He pinpoints these as 'outliers' that conform badly to the object-oriented paradigm (see Table). These metrics are derived by definition from the fully implemented source code, a process which can be automated with parsing tools. Indeed some of Marinescu's metrics are incorporated in the *Together* design package's metric module (Together 2002).
Our approach was a bit different from the perspective of others and our aim is to calculate error propagation values from UML diagrams (see Table 6). We interpret EP(A,B) as the probability that an error in A is propagated by B (as opposed to being masked by B) because the outcome of executing B will be affected by the error in A. By extension of this definition, we let EP(A,A) be equal to 1, which is the

probability that an error in A causes an error in A. Given architecture with N components, we let EP be an N×N matrix such that the entry at row A and column B be the error propagation probability from A to B.

Table 1: Summary of Chidamber and Kemerer's Six Object-Oriented Metrics

METRIC	**HIGH-LEVEL ATTRIBUTE**	**GENERAL SUMMARY**
Weighted Methods per Class (WMC)	size	Various weighting schemes can be used, reflecting traditional low-level metrics
Depth of Inheritance Tree (DIT)	inheritance	Maximum level of inheritance hierarchy for a class, from its root superclass. An indicator of re-use and complexity.
Number of Children (NOC)	inheritance	Number of subclasses per class, indicates extent of re-use.
Lack of Cohesion in Methods (LCOM)	cohesion	Somewhat arbitrary definition of cohesion calculated by determining how many methods in a class share attributes.
Coupling Between Objects (CBO)	coupling	The number of classes to which a class is coupled, by using their methods or attributes.
Response set for a Class (RFC)	coupling	The number of methods that can be invoked in response to a message to a class.

Table 2: List of Early-Definable System Parameters (Marchesi, 1998)

Number of Classes
Number of Packages
Number of Root Classes
Number of Responsibilities for a Class
Number of Abstract Responsibilities for a Class
Number of Concrete Responsibilities for a Class
Number of Subclasses of a Class
Number of Dependencies of a Class
Number of Dependencies between a Pair of Classes

Table 3: Marchesi's Proposed 'Early Definition' Object-Oriented Metrics

Class Metrics	
CL1	Weighted number of responsibilities for a class
CL2	Weighted number of dependencies for a class
CL3	Depth of inheritance tree
CL4	Number of immediate subclasses of a class
CL5	Number of distinct classes dependent on a class
Package Metrics	
PK1	Number of dependencies outwith a package
PK2	Number of dependencies within a package
PK3	Average of PK1
Global Complexity Metrics	
OA1	Number of classes
OA2	Number of inheritance hierarchies
OA3	Average weighted number of class responsibilities
OA4	Standard deviation of OA3
OA5	Average number of direct dependencies of a class
OA6	Standard deviation of OA5
OA7	Percentage of inherited responsibilities with respect to total number of responsibilities

Table 4: Reiβing's Proposed 'Early Definition' Object-Oriented Metrics

Class Metrics	
NAP	Number of public attributes in the public interface of a class
NAI	Number of public attributes in the inheritance interface of a class
NIA	Number of inherited associations of a class
NLA	Number of local (non-inherited) associations of a class
NAA	Number of all associations of a class
Package Metrics	
DNH	Depth in the nesting hierarchy
NCP	Number of total classes in a package
NPP	Number of nested packages in a package
System Metrics	
NIH	Number of inheritance hierarchies
aggregates	• total number of classes • mean number of methods per class • maximum depth of inheritance hierarchy • *etc*.

Table 5: Marinescu's *a Posteriori* Metrics for Identification of Badly Designed Classes

Data-Classes: define few methods other than accessor functions		
Weight of Class (WOC)	Ratio of non-accessor methods to total number of interface members	A low WOC value indicates low functionality
Number of Public Attributes (NOPA)	The number of non-inherited attributes belonging to the class interface	A high NOPA violates encapsulation and couple clients to the class
Number of Accessor Methods (NOAM)	The number of non-inherited methods declared in the class interface	High NOAM values may indicate that the functionality of the class is misplaced in other classes
God-Classes: over-centralize the functionality of the system		
Access of Foreign Data (AOFD)	The number of external classes from which a given class accesses attributes	High AOFD indicates tendency to Godliness
Weighted Method Count (WMC)	A measurement of the size and complexity of a given classes methods	A high WMC may indicate a major abstraction class or Godliness
Tight Class Cohesion (TCC)	A relative index of the number of connected methods accessing common instance variables	Low TCC ratios indicate non-communicative behaviour within a class

Table 6: Our Metric Identification

Number of States
Number of Messages
Number of Classes
Number of Packages
Number of Abstract Responsibilities for a Class
Number of Concrete Responsibilities for a Class
Number of Subclasses of a Class
Number of Dependencies of a Class
Number of Dependencies between a Pair of Classes

3 TECHNICAL BACKGROUND

3.1 The Unified Modelling Language (UML)

(These sections reference the UML standards versions 1.3 and 1.4 - available from OMG(2002), and Stevens and Pooley, 2000).

UML arose as a standard language for the specification of the artefacts of software systems from the convergence of three object-oriented analysis and design methodologies, initially defined by the Rational Software Development Company (Rational 2002). Its standards and developments are now controlled by the independent Object Management Group (OMG). Being both expressive and extensible UML is also suitable for business and non-software object-oriented modelling. While used by many software development tools, UML is not itself a methodology, and is implementation independent. The language supports higher level development concepts such as components, collaborations, frameworks and patterns. As such it can be used to document reusable artefacts (components and frameworks) as well as supporting system development.

The current version of UML is 1.4 (OMG 2001 UML1.4) Increasing numbers of software development tools are compliant with (some of) the 1.3 standards, though many still work from previous standards.

A *model* is a precise, abstract representation of the essential details of a design or system, from a given view. UML represents a model by any number of various graphical diagrams which provide multiple perspectives of the system under analysis or development. The underlying model integrates these views, which are represented to the modeller as a number of artefacts including:

• use case diagrams

• class diagrams

•behaviour diagrams (statechart, activity, interaction (sequence, collaboration))

• implementation diagrams (component, deployment).

Architecturally the actual model is described by a UML meta-model and UML metamodels are themselves loose instances of MOF (Meta-Object Facility) meta-metamodels, which provides an architecture neutral format for the inter exchange of model objects.

Three main types of modelling diagrams are supported:

- **use case model** (expressing system requirements from a users viewpoint)
- **static model** (describing the elements of a system and their relationships)
- **dynamic model** (describing the behaviour over time of a system).

For the purposes of this study we will restrict consideration to the UML class diagram, sequence diagrams and state machines, which captures many of the metrics of potential interest for the purposes described in Section 2.

2.4.3 UML Diagrams

Class diagrams document the static structure of a system: what classes (and packages) there are and how they are related, without specifying how they implement interactions to achieve behaviour. They can be created early in the design process, and refined throughout development, and they are readily obtainable from implemented application code. The classes provide all the behaviour required by the system.

Some of the class diagram features from which software metrics can be derived of are shown in Figure 3. These adapt examples in Stevens and Pooley (2000) or the OMG UML 1.3 and 1.4 specifications (OMG 2002).

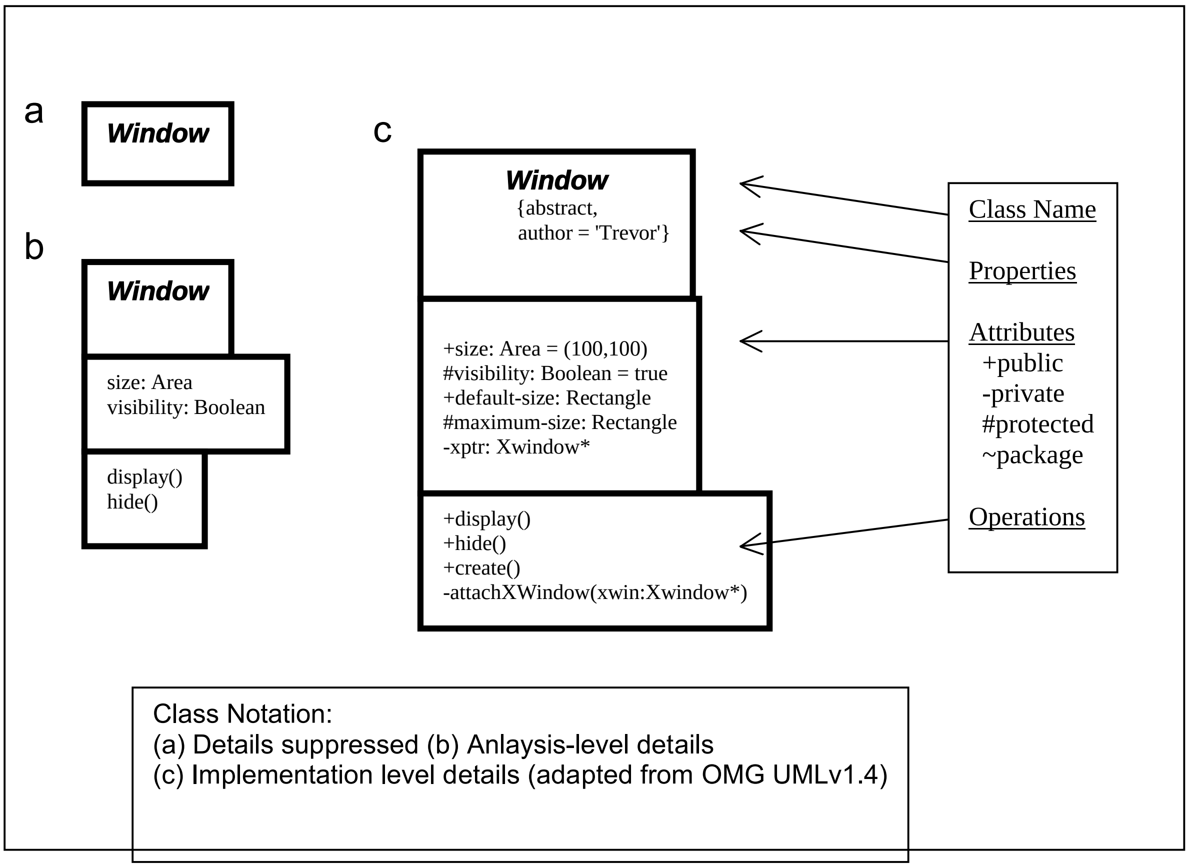


Figure 1: Class Notation

The attributes are the data contained in an object of a class, while operations define how objects interact open receipt of a message. The operation signature, with selector, return type and formal parameters, can be given (for example getLength*(b:Record): int*).

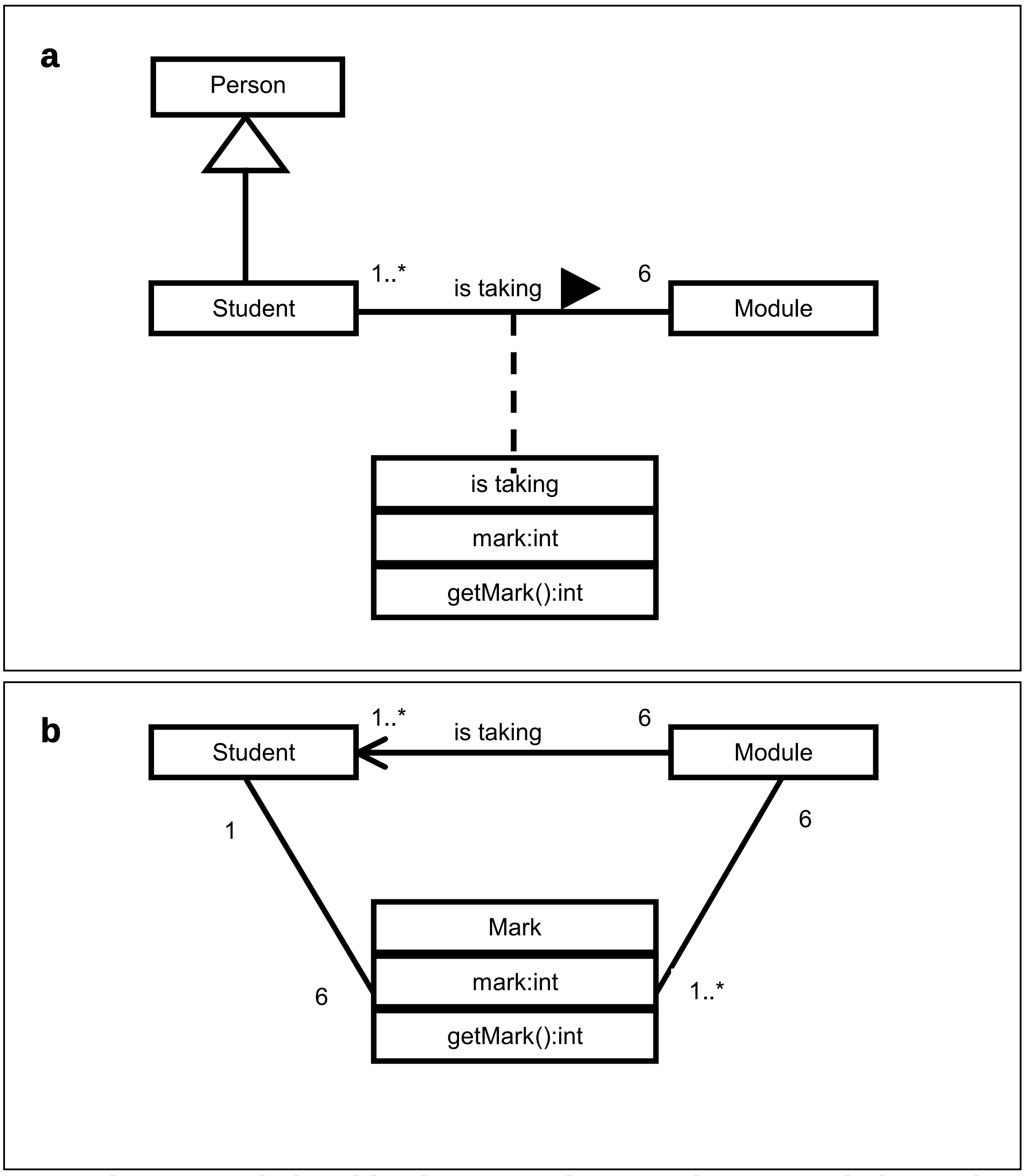


Figure 2: Relationships between Classes: Class Association and Generalization

In Figure 2a generalization (inheritance) is represented by the open arrow joining a subclass to the superclass it inherits from. The subclass should match the interface of the superclass, so that messages given to the superclass can also be given to the subclass (and a subclass can be used in place of a superclass (polymorphism)).

The navigability of an association shows the direction in which messages can be passed, where only one class knows of the other (as an instance variable for example). However, introducing navigability increases coupling between classes.

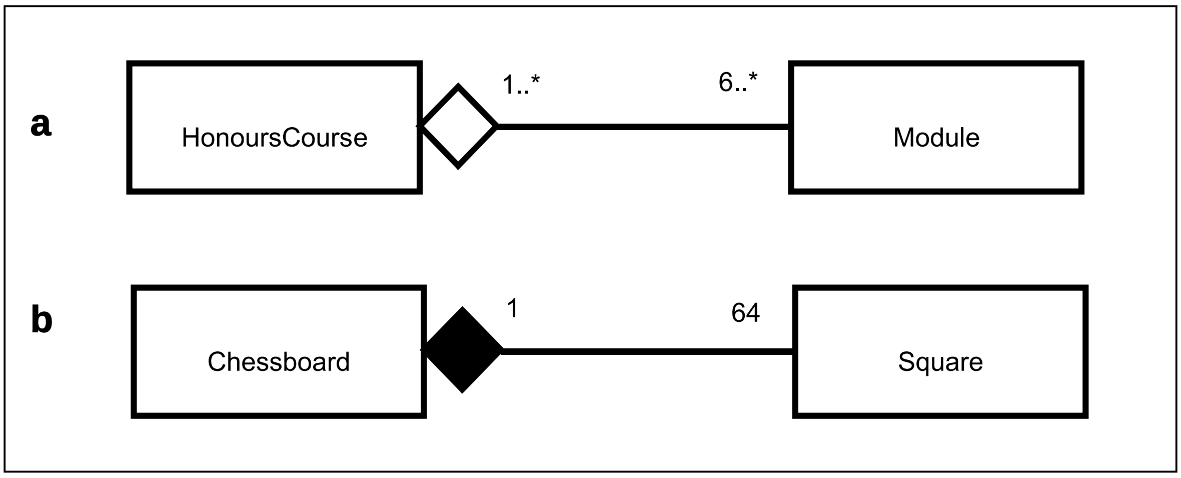


Figure 3: Relationships between Classes: Aggregation and Composition

Aggregations (Figure 3a) and compositions (or composite aggregations) (Figure 3b) are specialized (optional) forms of association where one class is part of an object of another class. Composition defines a much stronger ownership than aggregation, and for example deletion of the owning class deletes the associated classes.

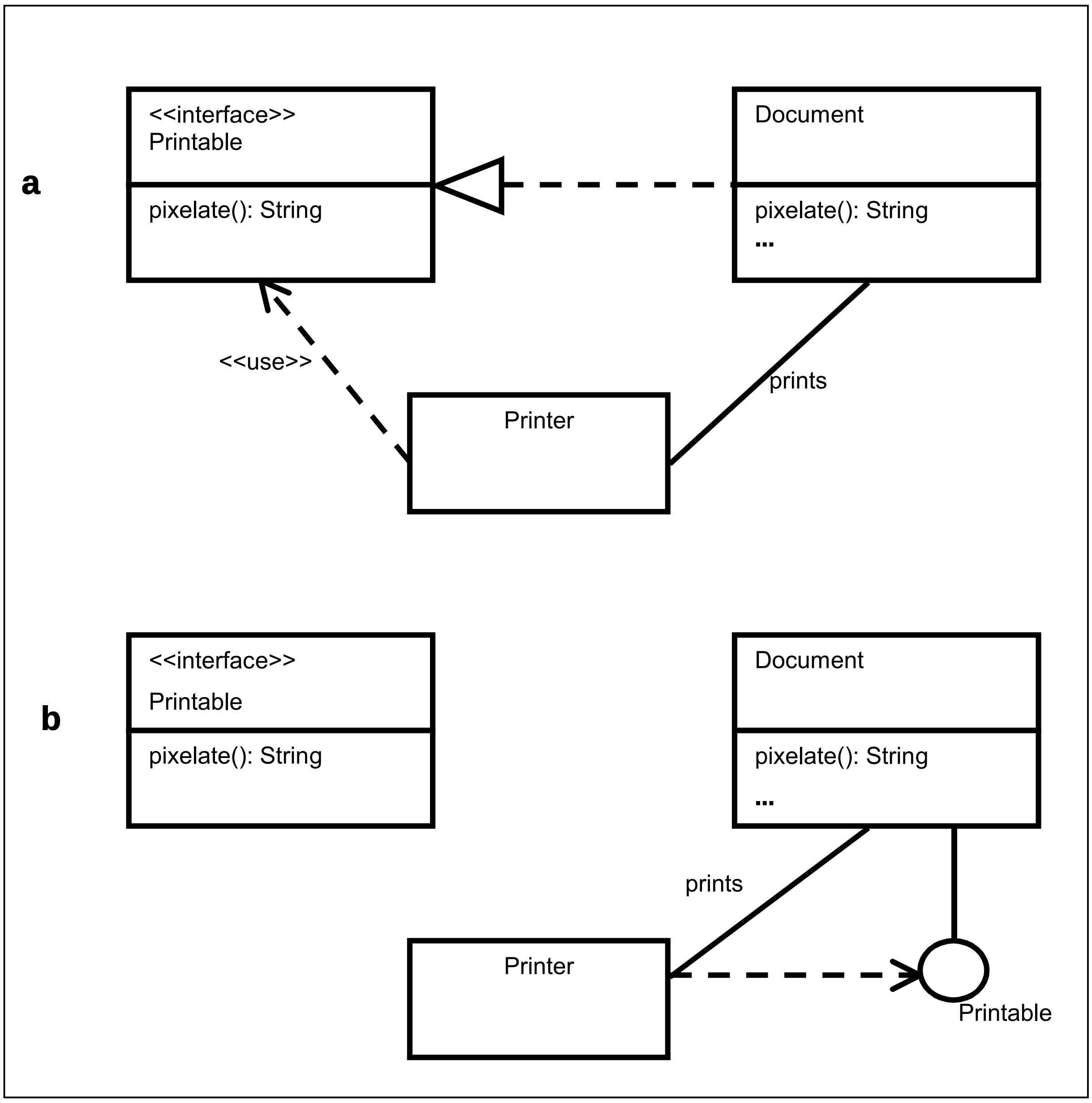


Figure 4: Relationships between Classes: Dependency and Interfaces

UML defines a number of 'stereotypes' and allows additional ones to be defined within a model - to provide extensibility. An <<interface>> stereotype defines a list of operations that any class matching (or realising/ implementing) the interface must provide. Classes may match more than one interface. Dependencies are necessarily reflected in close coupling between the dependent classes, and should be represented as explicitly as possible (for example generalisation is a form of dependency).

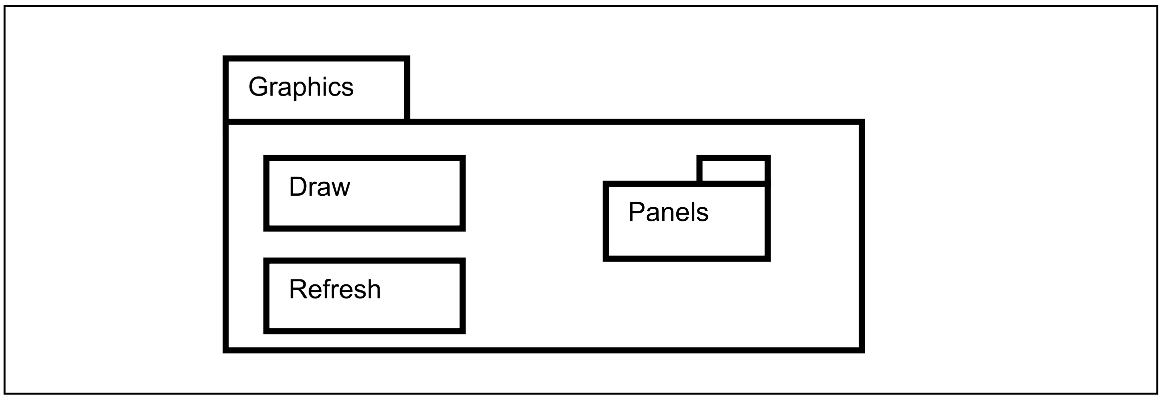


Figure 5: Packages

Packages are collections of any of the model elements composing the UML model, for example classes and the relationships between them. Package icons, illustrated in , can be used in several types of UML diagrams. Packages might define a design component, or may be used to divide a project up into workloads for the design team.

Collaboration diagrams allow the designer to specify the sequence of messages sent between objects in collaboration. The style of the diagram emphasizes the relationships between the objects as opposed to the sequence of the messages. In this column we will be discussing UML Sequence diagrams. Sequence diagrams contain the same information as Collaboration diagrams, but emphasize the sequence of the messages instead of the relationships between the objects.

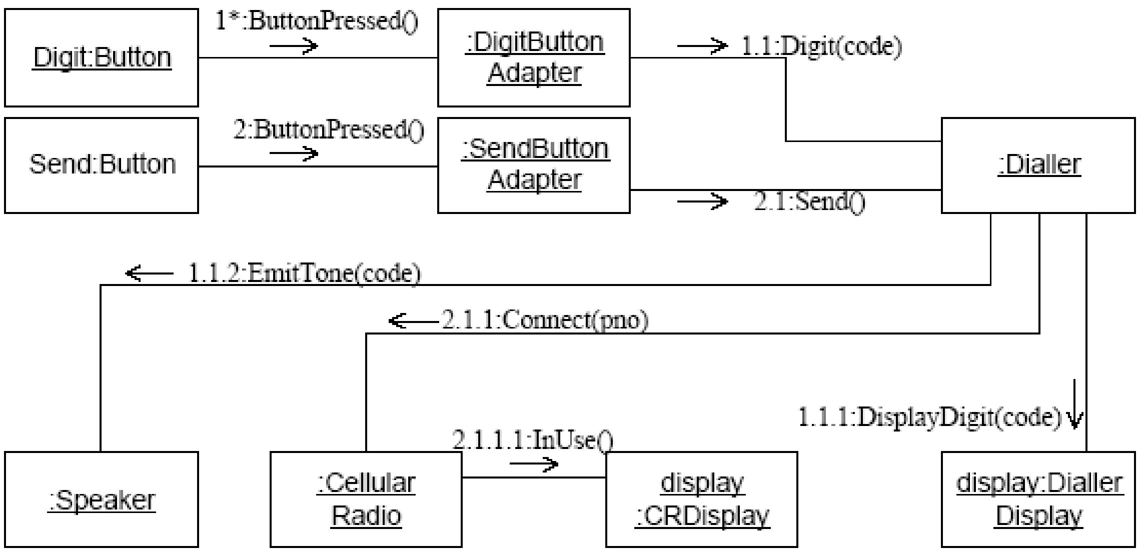


Figure 6: Collaboration Diagrams

State diagrams are used to describe the behavior of a system. State diagrams describe all of the possible states of an object as events occur. Each diagram usually represents objects of a single class and track the different states of its objects through the system.

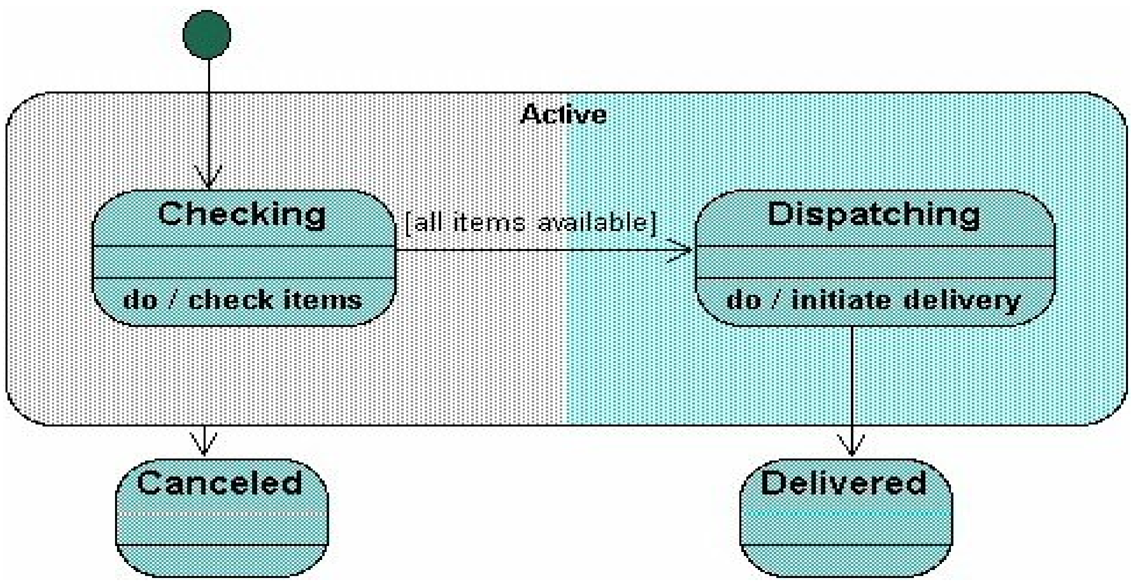


Figure 7: State Diagram

3.2 XML Representations of UML Diagrams

As described above many of the structural elements of a software application can be represented at the design stage, or upon implementation, as a UML diagrams. Class, collaboration and state diagrams presenting one of a number of views of a development model can be created using commercial or open source software development tools (for example *Rational Rose* (Rational 2002), *Together* (Together 2002) or *argoUML* (*Aro*UML 2002)). The requirement for this model information to be stored and transferable between different modelling tools, and other applications and repositories, was one of the main drivers behind the development of the XMI standard (XML Metadata Interchange). As XML (Extensible Markup Language) is not object-oriented XMI provides a standard method for mapping object models to XML to facilitate data exchange (Grose *et al.,* 2002).

2.4.4 Extensible Markup Language (XML)

It is necessary to be familiar with the concepts of XML in order to use XMI, as XMI implements XML DTDs (Document Type Definitions) and XML Documents. XML is an open standard, currently version 1.0, second edition, maintained by the World Wide Web Consortium, (WC3 2002 XML). (Additional references for this section: Carlson, 2001; Goldfarb and Prescod, 2001).

XML was defined as a lightweight, extensible meta-language for the representation of data and information about data (metadata) in the absence of details about its presentation. One of its primary aims is to facilitate the exchange of information in an application and architecture independent manner.

XML Parsers check that XML documents are 'well-formed', complying with a strictly defined and meaningful syntax, and create a data-structure (tree) of the entire document which can be accessed through the XML Document Object Model (DOM). The markup tags of XML define the meaning of the document structure, and can be extended through user defined tags. The allowable elements, tags, attributes and nestings can be described in internal or external SGML-style DTD or in more expressive, XML-compliant XML Schemas. Validating-Parsers can check the validity of an XML document to a given (linked) DTD or Schema.

XML documents are plain text documents containing nested tags describing element tags, attributes and data content. Stylesheets can be linked to XML documents in order to add presentational information. Cascading Style Sheets (CSS) can be used in a similar manner to their use with HTML, to enrich graphical display in a browser. Alternatively XSL stylesheets can be used to apply layout style in order to render an XML document for visual presentation.

3.2.1.1 Extensible Stylesheet Language (XSL)

W3C maintains the XSL standard, currently version 1.0 (W3C 2001 XSL). Conceptually XSL consists of two parts, a language for transforming XML documents, and an XML vocabulary for specifying how the (transformed) document is formatted. The first function is provided by XSLT (see Section 3.2.1.2) which transforms the source XML tree into another tree form, which may then be rendered for display by the formatting syntax of the second function (XSL-FO "flow-object"). This is shown in Figure 8 (from the W3C XSL specification, 2001).

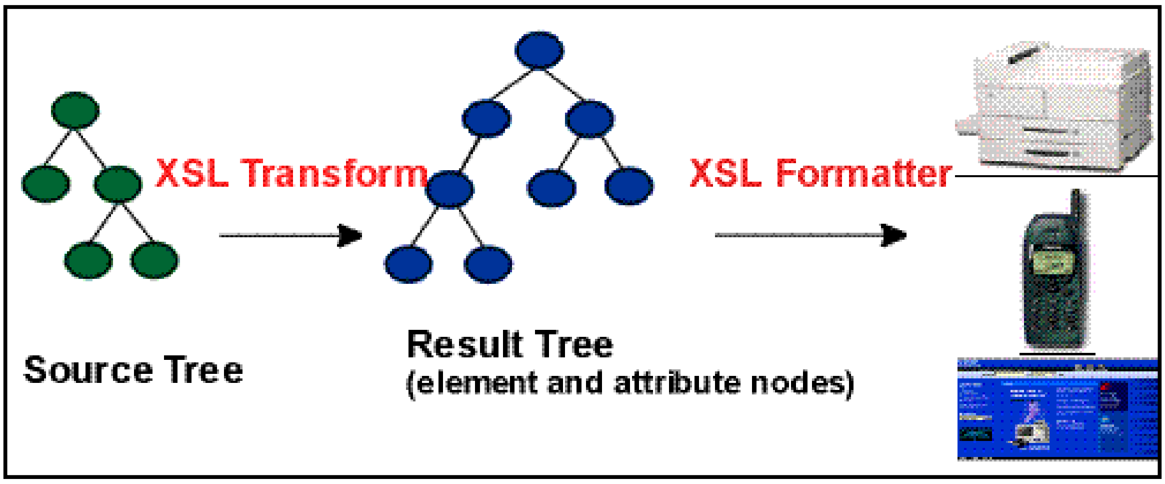


Figure 8: XSL Processing is Two-Stage

The XPath language provides the third necessary component: an addressing mechanism that allows specification of a path to any element of the source tree, allowing its manipulation.

Transformation converts from one XML vocabulary into another, or indeed into plain text, HTML style or other formats and markups. The formatter adds abstract formatting objects and attributes to the result tree produced by the transformation (for example paragraph styles, table style, font and colours) so that target applications (browsers, printers *etc*.) render the document as desired.

3.2.1.2 ***Extensible Stylesheet Language for Transformation (XSLT)***

XSLT provides a rich, non-procedural language for the transformation of an XML source document to one or more outputs, providing a lightweight alternative the creation of bespoke parsing applications to access and modify the XML document directly or through the DOM. (W3C XSLT version 1.0 specification, 1999). XSLT functionally resembles a scripting language, where the transformation process applies regular expressions to an input stream, transforming matched elements to an output stream, and has similar elements of control flow. Whilst XSLT is highly specialized for transforming XML trees, it is not very powerful at string or numeric manipulation.

XSLT can be used to transform an XML document from one schema (or DTD) to another, providing they have comparable semantics, and differ only in grammar. This can allow documents to be translated between standard and non-standard schemas.

An XSLT processor operates by applying order-independent template rules (specified in an XSLT document) to pattern-matched elements of the source document tree, returning the template results to the results tree, without altering the source document (see Figure 9). Each rule specifies a pattern for elements or attributes to match and a set of actions (template) of what to produce when a match occurs. Each rule adds a new node to the result tree, and can reorder and duplicate source elements, filter (delete) elements and attributes and add content to the document. The stylesheet language provides powerful techniques to access and rearrange all of the content (and tags) of an XML document by allowing conditional operations and specification of variables, parameters and indexable keys. Powerful pattern matching is performed using the XPath pattern matching language.

XSLT is highly suited to many XML processing applications, but cannot be used for continuous data streams, nor for heavyweight computational analysis or very large source documents (over a few megabytes) (Carlson, 2001).

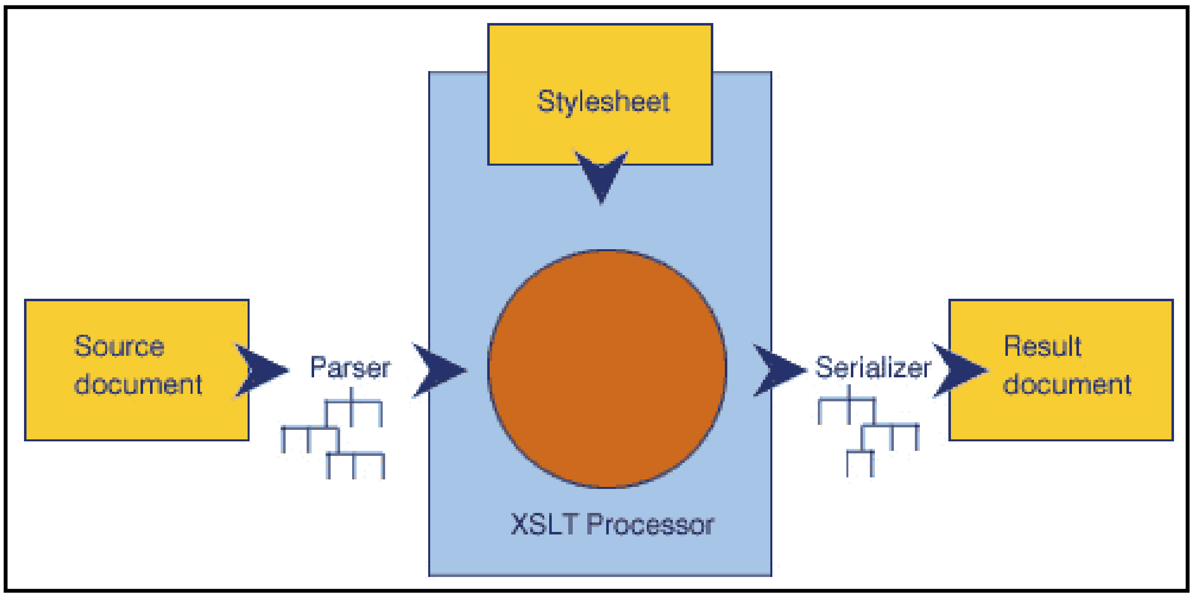


Figure 9: XSL Processing Converts the Source Node Tree into the Result Node Tree

3.2.1.3 XML Path Language (XPath)

XPath (W3C XPath version 1.0 specification, 1999) allows all the parts of an XML document to be addressed by providing a hierarchical datamodel of the document as a tree of element nodes. Under any given element node, there are further text nodes, attribute nodes, element nodes, comment nodes, or processing instruction nodes. Some node types have a name, and each node possesses an associated string value. Text nodes represent the textual data content of the document.

In general, an XPath expression specifies a *pattern* that selects a set of XML nodes. The location path resembles a filesystem naming hierarchy. An XPath expression may include a wide range of operators, functions and wildcards to allow accurate searching. Furthermore, XPath expressions can include predicates or selection-criteria, which allow nodes to be filtered or selected by name or value.

The path can be searched along different axes: the default child axis, the attribute axis, the content axis, the descendant axis, *etc*. Different axes allow a different set of defined node tests to be applied, for example in any context:

node()	returns all the child nodes
*	returns nodes of the current principal type
@*	matches any attribute node

or working within a content axis:

text()	returns any text node
comment()	returns any comment node.

XPath defines many functions. These can be used to return a set of nodes, or a string, number or boolean value. The functions include node-set functions, string functions, Boolean functions, positional functions, numeric functions and namespace functions. The operation of many of these functions depends on the context from which they are invoked.

XPath is used by XSLT and the XPointer language (which allows XPath expressions to be appended to URIs to point to XML data distributed over the Internet). It is also implemented in some applications of the DOM.

3.2.1.4 Programming Interfaces with XML

DOM

The Document Object Model (DOM) is the W3C platform- and language-neutral standardized API for managing and manipulating XML (and HTML) documents by building an object representation of the data (W3C (2002) DOM). The DOM allows programs and scripts to dynamically access and updates the complete content, structure and style of XML documents. DOM applications are well suited for interactive applications because the entire object model is present in memory, where it can be accessed and manipulated by the user. However, creation and retention of the DOM tree can be resource heavy for large XML documents, especially in a distributed environment.

The 'objects' held in the W3C standard DOM are in fact low level data-structures, not rich objects. An alternative, fully object-oriented API, JDOM, has been developed which represents XML documents in Java using an XML parser to build the document (jdom.org 2002). The alternative JDOM data representation seeks to provide a simpler programming environment, while integrating fully with the DOM and SAX standards.

SAX

The Simple API for XML (SAX) was developed informally as an API to work with XML parsers. It does not create a document tree, but handles an XML document as a series of events, streaming through the XML syntax, parsing and processing events in turn and returning only the desired output. This provides for lightweight processing, but does not allow for random access or return through the document. As such it is suited to server-side and high performance applications which do not require an in-memory representation of the data (for example, data-filtering, Web servers producing output to HTTP clients or data repositories).

Document parsing via the DOM or SAX provide alternatives to XSLT transformations for extracting and filtering information from an XML document. It is possible to combine SAX and DOM within a single system. Many parsers can produce both SAX and DOM output and a SAX stream can be used as input to a DOM builder, or a DOM's content can be used to generate SAX events (Akif *et al*., 2001).

2.4.5 XML Metadata Interchange (XMI)

XMI integrates three key industry standards:

1. XML - eXtensible Markup Language, a W3C standard

2. UML - Unified Modelling Language, an OMG modelling standard

3. MOF - Meta Object Facility, an OMG metamodelling and metadata repository standard'

In essence any metamodel that conforms to the MOF meta-metamodel can be represented as an XML document through XMI. XMI is therefore applicable to a wide variety of objects: for analysis, software, components and databases. XMI solves the problem of tool interoperability by providing a flexible and easily parsed information interchange format. The XMI stream contains both the definitions of the information being transferred and the information itself.

A UML model is an instance of a UML metamodel, which is in turn an instance of the MOF model, and XMI allows for such a compliant model to be treated as the metamodel and represented by an XML DTD and document, produced according to XMI. In more simple terms, XMI provides a vocabulary specified by an UML.DTD for the description of the components (model elements, attributes, associations, *etc*.) of a UML model.

The XMI format was designed to be produced automatically and consistently from a UML model using an XMI processor. The documents produced are designed for machine-readable XML data interchange, not compact, human readable documents (Carlson, 2001). A more complete description of the format XMI is given in Section 2.4.7.

2.4.6 XMI and Software Development Tools

Software development tools can save the details of UML diagrams as an XMI document.

A range of commercial object-oriented software development suites are capable of importing and exporting XMI representations of UML models. These include the proprietary *Rational Rose* (Rational 2002), *Together ControlCenter* (Together 2002), *Objecteering* (Softeam 2002), *Ideogramic UML* (Ideogramic 2002) and *Posiedon* (Gentleware 2002). *Poseidon* is based upon the opensource CASE tool *argoUML* (ArgoUML 2002) which uniquely was developed from inception to use XMI to store the UML model, not merely to facilitate data interchange via importing and exporting.

Several of the UML design suites have tools for measuring and analysing procedural and object-oriented software metrics. For example *Together* ControlCenter reports on 47 different metrics (including object-oriented metrics) whilst Objecteering derives an exhaustive set of 80 low and high level metrics based on the work of Lorenz and Kidd (1994), which aim to check and maintain model quality.

2.4.7 XMI Representation of UML Diagrams

Current XMI versions 1.0 to 1.2 do not support schemas, and use DTDs to specify the metamodel structures. DTD are derived automatically from the MOF metamodels as described in the XMI standard: the UML1.1 DTD is described in the XMI standards (up to 1.2) but a current OMG UML1.3 DTD is available and can be used by several of the CASE tools including *argoUML*. There is no requirement for XMI to implement XML validation, so XMI documents are not required to specify their DTD, and indeed might not necessarily validate against a specified DTD.

The XMI representation of UML class diagrams is best illustrated by example. Figure 10 shows a simple class diagram created with *argoUML*, consisting of a single **Class** 'ClosedFigure' **realizing** (implementing) the **Interface** 'Figure'.

XMI assigns each model element a unique xmi.id, and nests the elements within the root element Model 'Graphics', which is assigned xmi.id = xmi.1. This also defines a namespace for each element in the model. These unique IDs allow elements to reference associated elements, as xmi.idref values and also provide an access method to the data structure when processing with XSLT.

XMI (version 1.0) provides two further attributes which can act as identifiers for model elements xmi.label, for string descriptors, and xmi.uuid for a (globally) universally unique identifier. These attributes are used differently by the various model creation tools. *Rational Rose* optionally allows uuids to be generated, while *argoUML* uses them in a non-standard - but very useful - manner to identify elements that have been defined by the user, *i.e.* are not part of standard Java language packages *etc.* Therefore, in Figure 11 only the model namespace (Graphics), the two user-defined classes (Figure and ClosedFigure) and the dependency abstraction (xmi.3) are assigned an xmi.uuid. In contrast saving a similar class diagram produced with *Rational Rose* would have created xmi.uuids for many other model elements representing operations, arguments and attributes in addition to any Java language elements, whilst *Together* does not generate xmi.uuids in its implementation of XMI.

'Figure' and 'ClosedFigure' are nested within 'Graphics', and their operations and attributes are similarly nested within. The xmi.id of each element is illustrated on Figure 10. The dependency relationship between the Class and Interface is represented by an Abstraction element (xmi.3), and this is extended by the realization stereotype detailed by element xmi.31. The dependency client ('ClosedFigure') and dependency supplier ('Figure') are recorded by referenced xmi.idref values in the Abstraction element. Each participating element also records the relationship.

Every element has a number of associated properties whose values are recorded within the element. For example, details of the signature of the 'display()' operation are recorded as **.**visibility = "public" and **.**isAbstract = "true". Parameters of an operation, including return type, are represented by elements nested within the operation.

Datatypes are also represented by elements. *argoUML* defines simple (atomic) and common Class datatypes through java.lang library classes, which are recorded as part of the model. (For example, type float is recorded as element xmi.13, which is nested within java and lang elements). Additional user defined datatypes can also be created, for example the datatype 'Color' (xmi.27).

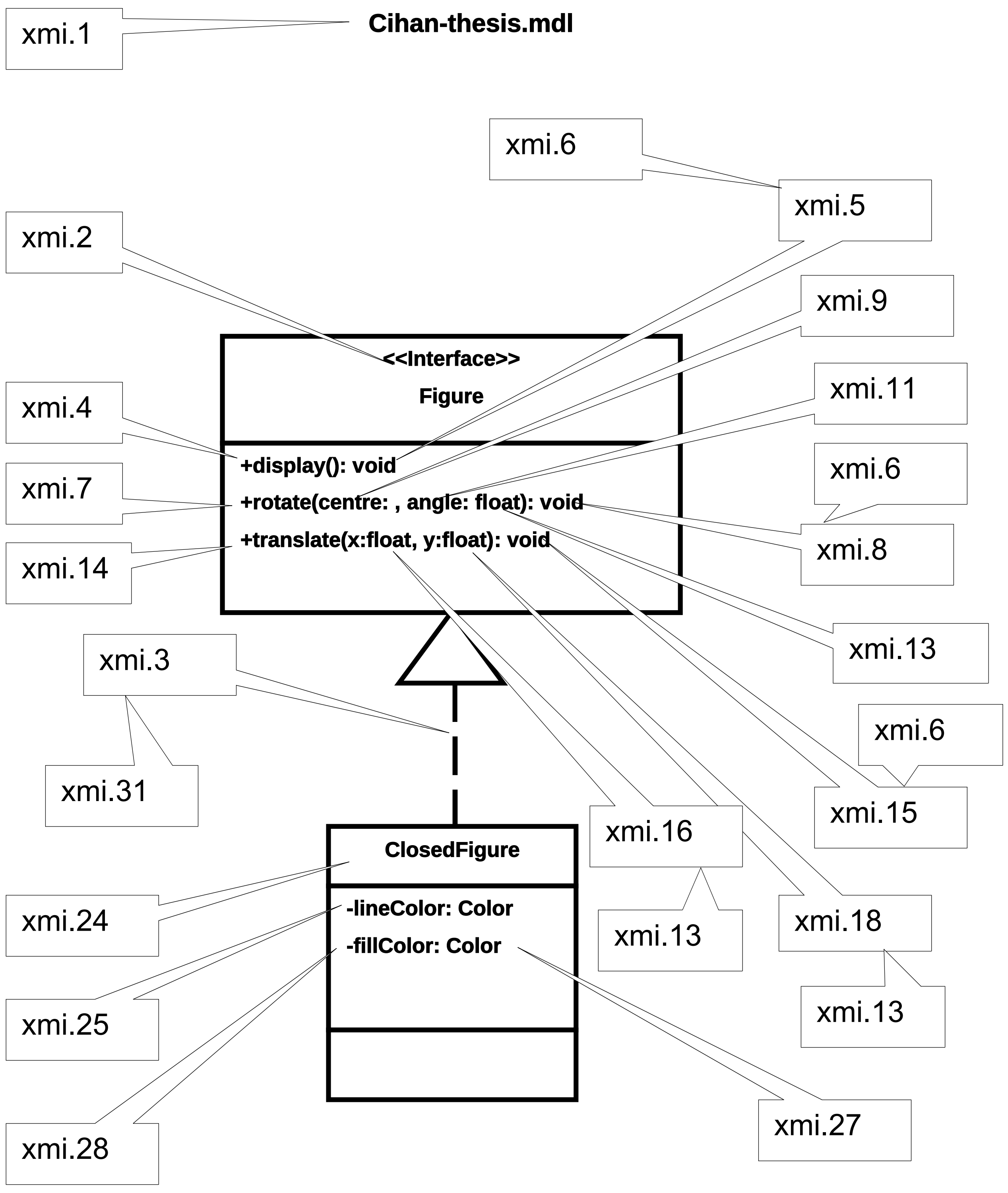


Figure 10: Annotated XMI representation of a simple class diagram

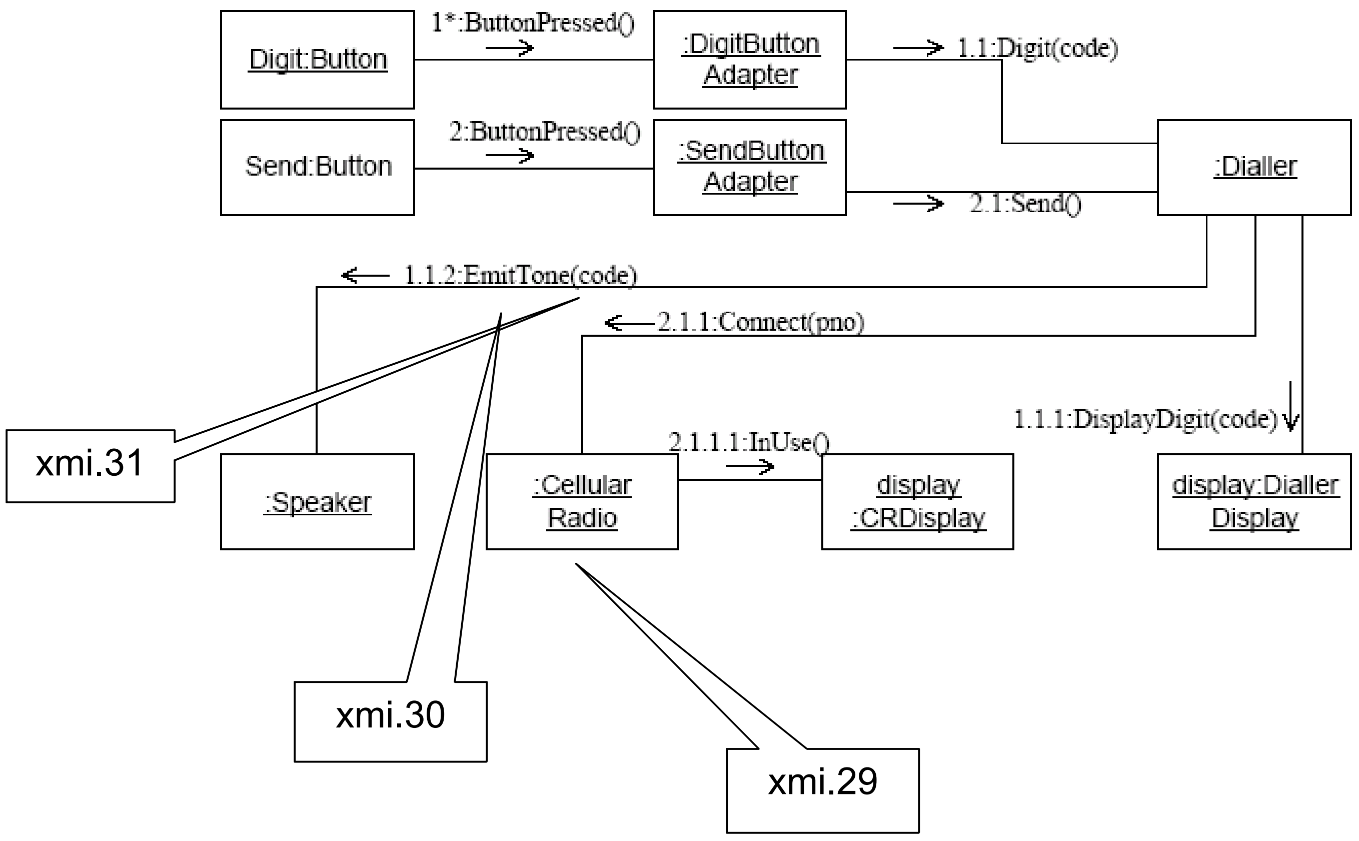


Figure 11: Annotated XMI representation of a simple collaboration diagram

A simple collaboration diagram is shown in Figure 11. Arrows shows the messages that are passing through classes or interfaces. Xmi.29 is assigned to show classes or interfaces. Xmi.30 is assigned to show the message name and xmi.31 is assigned to show attributes of messages.

A simple state diagram is shown in Figure 12. Below is an example of a state diagram might look like for an Order object. When the object enters the Checking state it performs the activity "check items." After the activity is completed the object transitions to the next state based on the conditions [all items available] or [an item is not available]. If an item is not available the order is canceled. If all items are available then the order is dispatched. When the object transitions to the Dispatching state the activity "initiate delivery" is performed. After this activity is complete the object transitions again to the Delivered state.

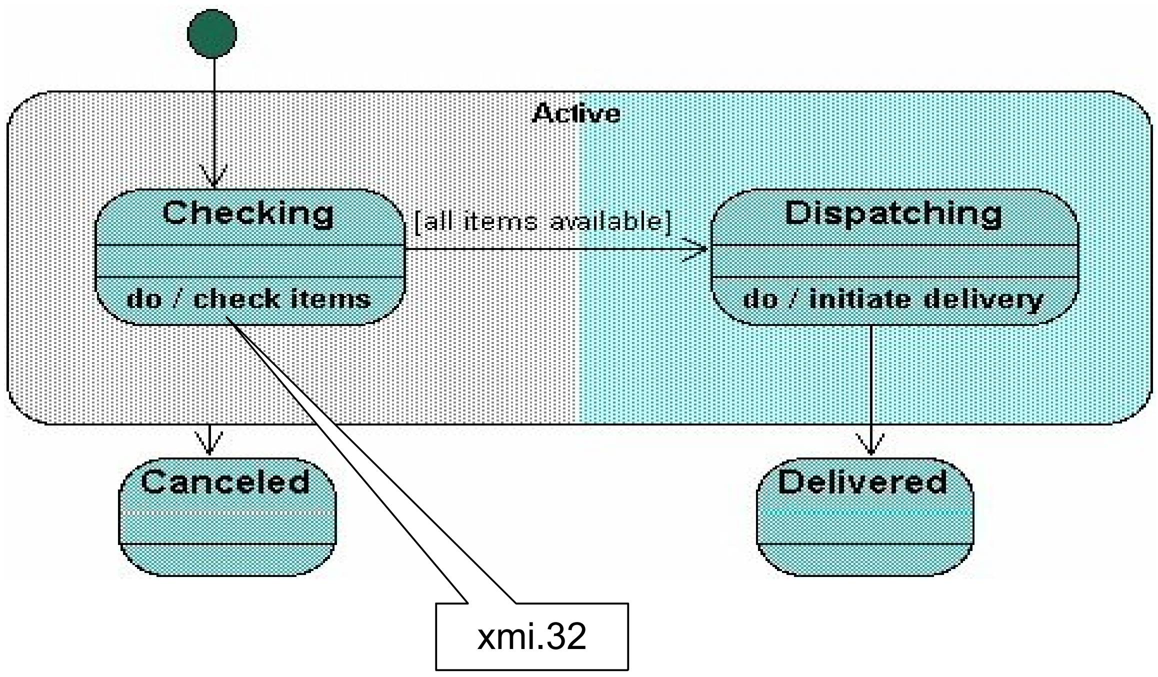


Figure 12: Annotated XMI representation of a simple state diagram

4 ERROR PROPAGATION PROBABILITIES

The study of software architectures is emerging as an important discipline in software engineering, because not only software architectures emphasize large scale composition of software products but also they support many emerging paradigms of software development, such as product line engineering, components-based software engineering, COTS-based development, as well as *software evolution.* In this part, the attribute of *Error Propagation Probability* will be discussed, i.e. the probability that an error that arises at run time in one component will propagate to other components. This effort is part of a larger project which investigates a wide range of attributes, including *Change Propagation Probabilities, Requirements Propagation Probabilities,* etc (Ammar et al. 2001). The focus on the architectural level (rather than design or code level) has a profound impact on our work, affecting both its goals and its means, as we discuss in the sequel; first, we introduce our view of software architectures, for the purposes of this study.

4.1 A Working Model of Software Architectures

According to Bass et al, *"The Software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them".* It is common to distinguish between five broad classes of architectures, called *architectural styles,* where each style is defined/ characterized by: component types; communication patterns/ protocols between the components; semantic constraints; and a vocabulary of connectors. The five architectural styles are:

- *Independent Components.* The architecture is an aggregate of independent processes/ objects that communicate through data or control messages.
- *Virtual Machines.* In this style, architecture is an aggregate of virtual machines arranged in layers, where each layer invokes the layer below it and provides the vocabulary to define the layer above it.
- *Dataflow Architectures.* The architecture is an aggregate of processing nodes whose activation is driven by the flow of data streams.
- *Data Centred Architectures.* The architecture is an aggregate of interacting components that communicate through a shared data repository.
- *Call and Return Architectures.* In this style, an architecture is an aggregate of components that are defined in programming terms (procedures, functions routines) and whose interactions are restricted to programming language supported interactions (call and return, parameter passing, etc).

Perhaps with some loss of generality, we focus our attention in this study on the first architectural style, i.e. independent components.

4.2 Architectural Goals

The focus of the study on software architectures has a direct impact on what attributes we may wish to define, characterize and quantify. Traditional software metrics that characterize source code or depend on the executable/ operational nature of source code for their definition (e.g. reliability, dependability) are not too much meaningful at the architectural level. Architectural quality attributes can be divided into two distinct classes:

- Attributes that view the software architecture as an intrinsic product, and characterize it as such.
- Attributes that view the software architecture as a blueprint for operational software systems, and characterize it by the properties of these systems.

Main focus of our attention on the latter class, so that when we say that architecture has some attribute, we actually mean that operational software systems that are derived from this architecture have this attribute.

As a matter of separation of concerns, and in order to facilitate our discussions, we define a three-tier hierarchy of attributes:

- *Qualitative Attributes,* which represent relevant features of an architecture that we want to define and characterize.
- *Quantitative Functions,* which represent formally defined functions that may be related to the qualitative attributes or may represent some aspect of a qualitative attribute.
- *Computable Metrics,* which represent quantitative functions that we can compute by analyzing the architecture.

4.3 Architectural Means

Not only does the focus on architectures affect our goals, it also affects the means we have at our disposal to achieve these goals. Within the architectural style that we have selected we cannot rely on the availability of source code-like structural or semantic information. We resolve to consider that the only information we can count on, across the various representations of software architectures, is information on the data flow and control flow within components and between components. In the absence of functional/ operational information, we rely on probabilistic arguments to quantify the information flow throughout the architecture. An intuitive approach is to model information flow by means of random variables and to quantify it by means of entropy functions; in the course of our study we will also use other functions when the need arises.

The focus on architectures limits not only the amount of information that we have access to, it also restricts the type of modeling we can make. In order to define a fault model for a system, we need two types of information regarding the system:

- *Structural information,* whose level of detail is commensurate with the precision with which we want to identify faults.
- *Operational information,* which catalogs the set of abnormal behaviors that we want to consider for each identifiable unit (re: level of structural detail).

In the absence of detailed operational information, we cannot define a credible fault model on software architecture; hence we shift our attention away from faults and focus it on errors instead. Furthermore, in keeping with our architectural model, we let our identifiable units be components and connectors; we model an error in a component by an alteration of its state, and we model an error in a connector by an alteration of the message that it carries.

4.4 Error Propagation Probabilities

In this section, we first introduce and discuss the feature of *error propagation* in architecture. Then we review some derivatives of this feature.

4.4.1 Error Propagation: Definition

We consider two components, say A and B, of an architecture, and we let X be the connector that carries information from A to B; for the purposes of our current discussion, the specific form of connector X is not important, we will merely model it as a set (of values that A may transmit to B). Also, the specific form of components A and B is not important for the purposes of our discussion; we will merely model them as functions that map an internal state and an input stimulus into a new state and an output.

Definition 1. The *Error Propagation Probability* from component A to component B is denoted by EP(A,B) and defined by:

$$EP(A,B) = \mathbf{P}\,([B](x) \neq [B](x') \mid x \neq x') \qquad (1)$$

where [B] denotes the function of component B, and x is an element of the connector X from A to B. We interpret [B] to capture all the effects of executing component B, including the effect on the state of B as well as the effect on any outputs produced by B.

We interpret EP(A,B) as the probability that an error in A is propagated by B (as opposed to being masked by B) because the outcome of executing B will be affected by the error in A. By extension of this definition, we let EP(A,A) be equal to 1, which is the probability that an error in A causes an error in A. Given architecture with N components, we let EP be an $N \times N$ matrix such that the entry at row A and column B be the error propagation probability from A to B (Nassar, 2002).

Note that nothing in our definition above indicates that x' is an erroneous message; all the definition says is that x' is different from x --- as far as this definition is concerned, both could be correct. While this may seem to be an anomaly, all it means is that we are measuring error propagation probabilities by a wider property, which is the probability that different arguments are mapped by function [B] to different images (a measure of injectivity of [B]).

4.4.2 Error Propagation Derivatives

In this section we derive three measures of interest from the error propagation probability we defined.

4.4.2.1 __Unconditional Error Propagation__

Note that the definition of the error propagation given above uses the concept of *conditional* probability, i.e. we calculate the probability that an error propagates from *A* to *B under the condition that A actually transmits a message to B*. It is often useful, however, to use the *unconditional error propagation* which we will denote simply as E(*A*,*B*), and define as the probability that an error propagates from *A* to *B* not conditioned upon the event that *A* sends a message to *B*. Function *E(A,B)* is clearly dependent on *EP(A,B),* but it further integrates the probability that *A* does send a message to *B*. In order to bridge the gap between the original *(conditional) error propagation* and the newly introduced *unconditional error propagation*, let us consider the *transmission probability matrix* ***T*** where the entry ***T***(*A*, *B*) reflects the probability with which the connector gets activated during a typical/ canonical execution. ***T*** is the *NxN* matrix whose entry ***T***(*A*, *B*) is the probability that the component *A* sends a message to component *B* given that the *A* is expected to transmit a message to *some* component. Note that:

- It is reasonable to assume that ***T***(*A*, *A*) = 0 for all components *A*,

The matrix ***T*** is used to distinguish between a connector that is invoked intensively in each execution and one that is invoked only occasionally, under exceptional circumstances. The matrix ***T*** reflects the variance in frequency of activations of different connectors during a typical execution.

By virtue of simple probabilistic identities, we find that the *unconditional error propagation* is obtained as the product of the conditional error propagation probability with the probability that the connector over which the error propagates is activated, i.e.

$$\mathrm{E}(A,B) = \mathrm{EP}(A,B) \times \mathbf{T}(A, B) \qquad (2)$$

The concept of *unconditional error propagation* is useful when we discuss *cumulative error propagation probabilities*, which we do in the next subsection.

4.4.2.1 Cumulative Error Propagation

So far we have focused our attention on single step error propagation from some component *A* to some component *B,* we want to consider, now, the probability that an error in some component *A* propagates to some component *B* in an arbitrary number of transmissions (steps) starting in *A* and ending in *B.* We call this the *cumulative error propagation probability* from *A* to *B.* We submit two premises pertaining to the analysis of cumulative error propagation:

- Cumulative error propagation probabilities must be derived, not from matrix EP but rather from matrix E. Indeed, the probability that an error propagates along some path depends first and foremost on the probability that the path is actually taken, combined with the probability that the error is propagated through each arc of the path.
- Second, the matrix of cumulative error propagation probabilities cannot be derived as the traditional transitive closure of matrix E, because while matrix T is stochastic, matrix E is not. Hence we need to find a specific formula for this case, which we do in the sequel.

Where E_s is the *s-step* error propagation matrix, i.e. $E_s(A, B)$ is the probability that an error in A propagates to B via *exactly s* connectors. The s-step error propagation matrix E_s is given by:

$$E^*(A, B) \leq \sum_{s \geq 0} \overline{E}_s(A, B) \qquad (3)$$

4.5 Estimating Error Propagation

We have found that analytically, the error propagation probability can be expressed in terms of the probabilities of the individual A-to-B messages and states, via the following formula:

$$EP(A \to B) = \frac{1 - \sum_{x \in S_B} P_B(x) \sum_{y \in S_B} P_{A \to B}[F_x^{-1}(y)]^2}{1 - \sum_{v \in V_{A \to B}} P_{A \to B}[v]^2} \qquad (4)$$

where $F_x^{-1}(y) = \{ v \in V_{A \to B} \mid F_x(v) = y \}$, and we assume a probability distribution *PB* on the set of states *SB* of component *B*, and a probability distribution *PA→B* on the set of messages *VA→B* passed from *A* to *B*.

The term $\sum_{v \in V_{A \to B}} P_{A \to B}[v]^2$ in the denominator of (4) is an exponent of the 2nd order Renyi entropy, which according to the recent studies is closely related to the classical Shannon entropy. If we assume that the states of *B*, as well the messages passing through the connector from *A* to *B* are equi-probable, then the formula for error propagation is simplified into

$$EP(A \to B) = \frac{1 - \frac{1}{|S_B||V_{A\to B}|^2} \sum_{x \in S_B} \sum_{y \in S_B} |F_x^{-1}(y)|^2}{1 - \frac{1}{|V_{A\to B}|}} \qquad (5)$$

Since the software practitioner cannot always extract from the available artefact the detailed information on the transition table *F* for the architectural components, it would be helpful to be able to estimate the right-hand side of (6) without using any knowledge of function *F*. The following inequality gives precisely such an estimate (upper bound)

$$EP(A \to B) \leq \frac{1 - \frac{1}{|S_B|}}{1 - \frac{1}{|V_{A\to B}|}}. \qquad (6)$$

4.5.1 Examples

For the general metrics,

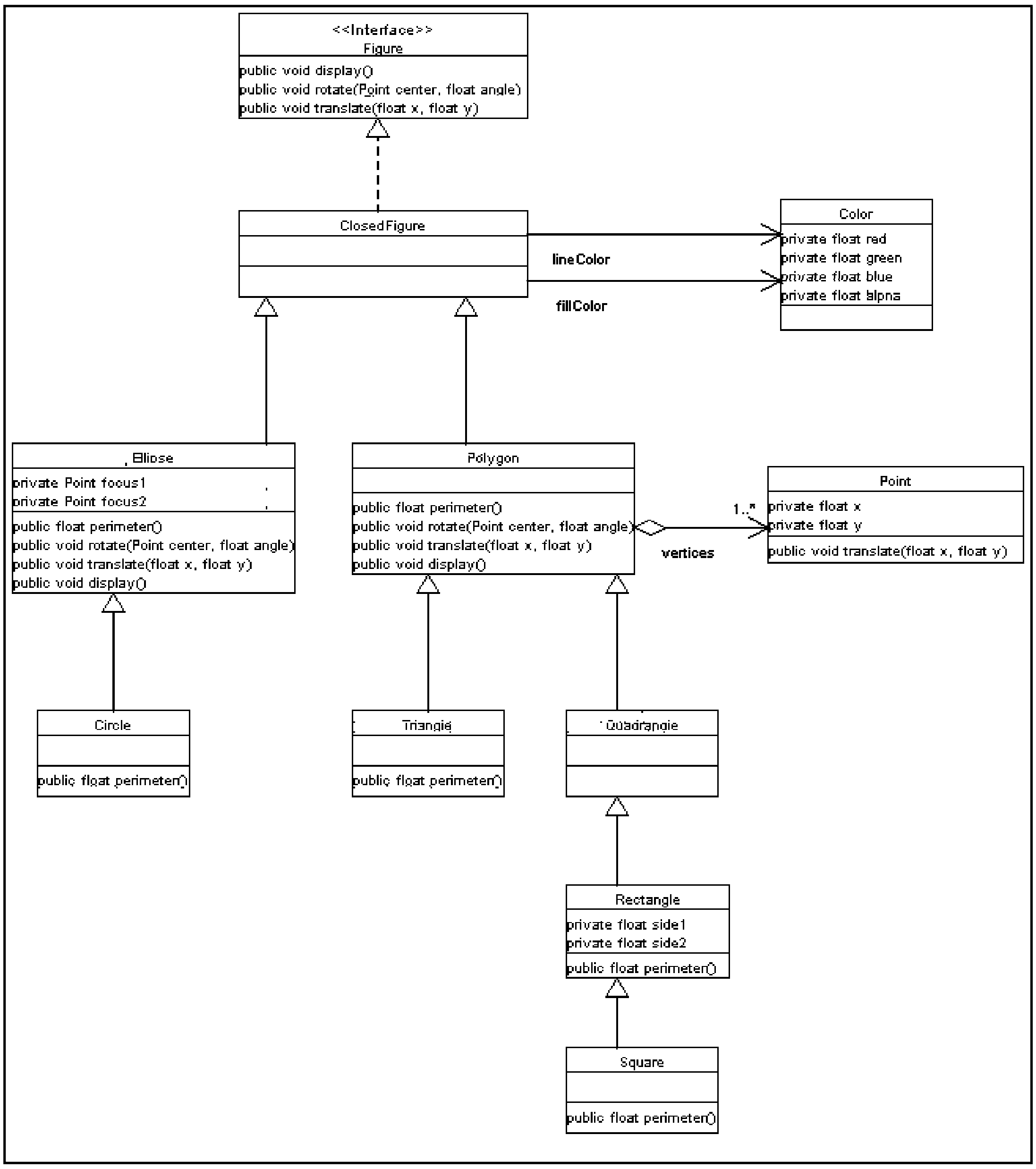


Diagram 1: The *Object by Design* Graphics Editor Model

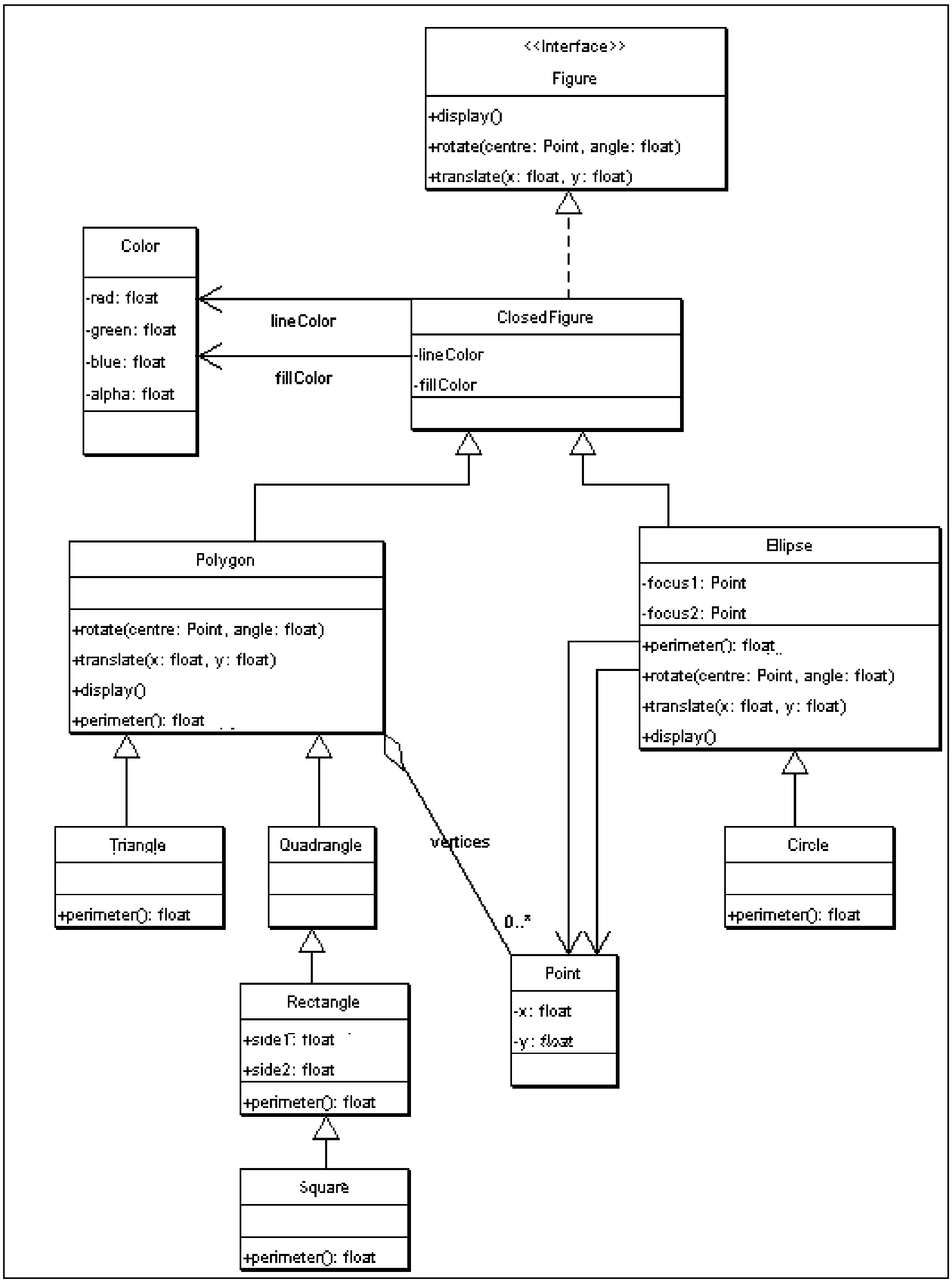


Diagram 2: The *Objects by Design* Graphics Editor Model Redrawn with *argoUML*

Table 7: OBD Class Metrics

Name	ID	nII	nC	DIT	nAt	+nAt	%+At	nM	+nM	%+M	nAs
Circle	xmi.28	0	0	2	0	0	-	1	1	100	0
ClosedFigure	xmi.36	1	2	0	0	0	-	0	0	-	2
Color	xmi.31	0	0	0	4	0	0	0	0	-	2
Ellipse	xmi.11	0	1	1	2	0	0	4	4	100	0
Point	xmi.2	0	0	0	2	0	0	1	1	100	1
Polygon	xmi.59	0	2	1	0	0	-	4	4	100	1
Quadrangle	xmi.82	0	1	2	0	0	-	0	0	-	0
Rectangle	xmi.84	0	1	3	2	0	0	1	1	100	0
Square	xmi.90	0	0	4	0	0	-	1	1	100	0
Triangle	xmi.74	0	0	2	0	0	-	1	1	100	0

Table 8: Class Metrics

Name	ID	nII	nC	DIT	nAt	+nAt	%+At	nM	+nM	%+M	nAs
Circle	xmi.88	0	0	2	0	0	-	1	1	100	0
ClosedFigure	xmi.21	1	2	0	2	0	0	0	0	-	2
Color	xmi.105	0	0	0	4	0	0	0	0	-	2
Ellipse	xmi.59	0	1	1	2	0	0	4	4	100	2
Point	xmi.10	0	0	0	2	0	0	0	0	-	3
Polygon	xmi.29	0	2	1	0	0	-	4	4	100	1
Quadrangle	xmi.86	0	1	2	0	0	-	0	0	-	0
Rectangle	xmi.92	0	1	3	2	2	100	1	1	100	0
Square	xmi.101	0	0	4	0	0	-	1	1	100	0
Triangle	xmi.82	0	0	2	0	0	-	1	1	100	0

Second example we use to illustrate our work is a large command and control system that is used in a life-critical, mission-critical application. This system was modeled using the Rational Rose Realtime CASE tool. It is a Computer Software Configuration Item (CSCI) that provides the following functions:

• Facilitating Communication, Control, Cautions and Warnings including subsystem Configuration Management, C&DH (Communication and Data Handling) Communications Control, Processing, Memory Transfer, C&DH Failure Detection, Isolation, and Recovery and Time Management,

• Controlling a Secondary Electrical Power System, and

• Environmental Control, which provides Temperature and Humidity Control.

We concentrate on the Thermal Control part of the system, which is a rather complex system with operations setting controller, fault recovery procedures, and pump control functionalities. System is responsible for providing overall management of pumps as well as performing the necessary monitoring and response to sensors data. Also, it is responsible for performing automated start-up.

During each execution cycle, a check is performed for incoming commands. Received commands are validated in the same execution cycle. Mode change commands, which will reconfigure the Internal Thermal System, are also accepted from other components of Thermal System to compensate for system component failures or coolant leaks. A failure recovery system detects failure conditions and performs recovery operations in response to the detected failures. Failure conditions include combinations of Pump failures and Shutoff Valve failures.

The system has a hierarchical architecture. Using these artifacts, one can identify the components and the connectors that describe the components-based system architecture and label the EP matrix rows and columns with the components names.

Figures show a sample message protocol between a pair of components in our system. This artefact provides us with the message set VA>B and VB>A that is going between the two components A and B. Similarly, using the Rose-RT tool we can get the whole sets of messages that are going on between each pair of components in the system.

The state chart shown in Figure is a sample of state chart of a component in the system. This provides us with the state set SB for this sample component. Using the Rose-RT tool, we can easily identify the triggering messages from one state to another. In a similar way, one can get all the state sets for all the components.

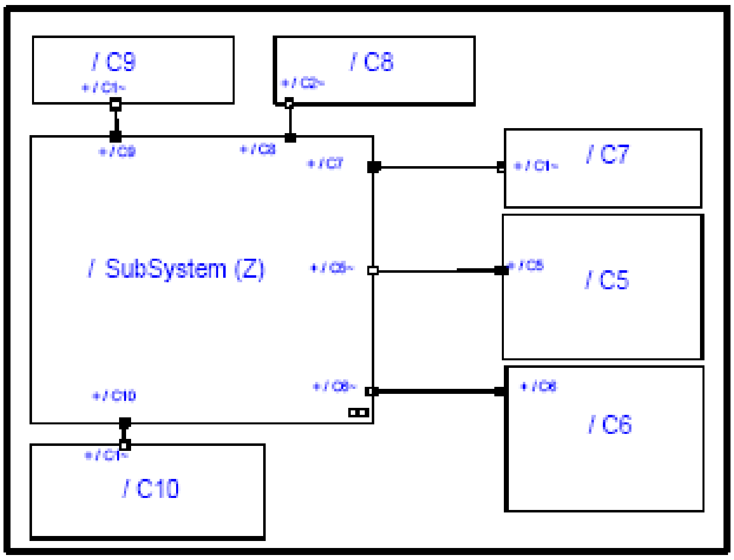


Figure 13: General view of the System Command and Control System

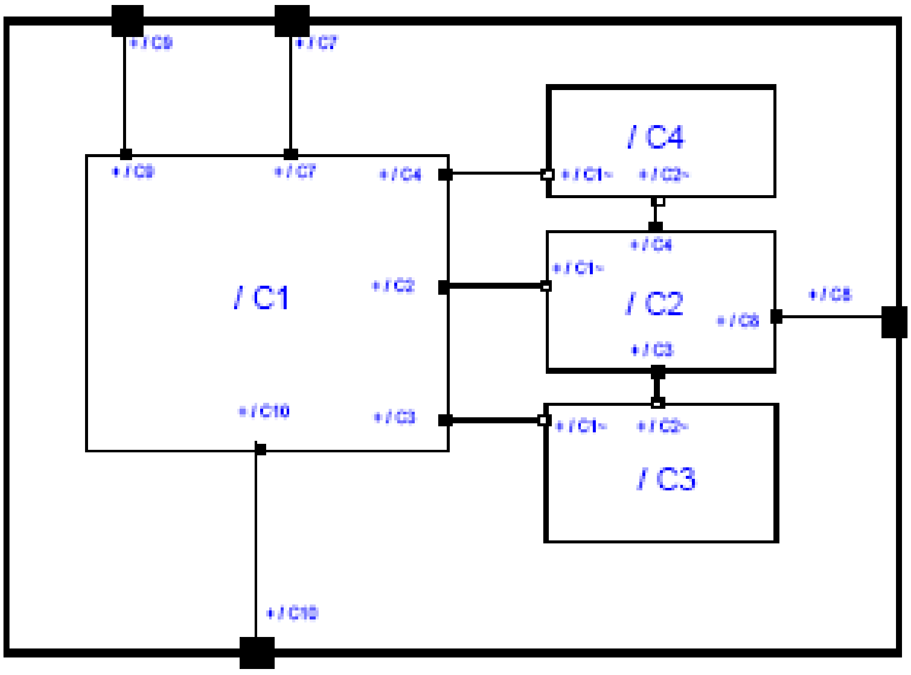
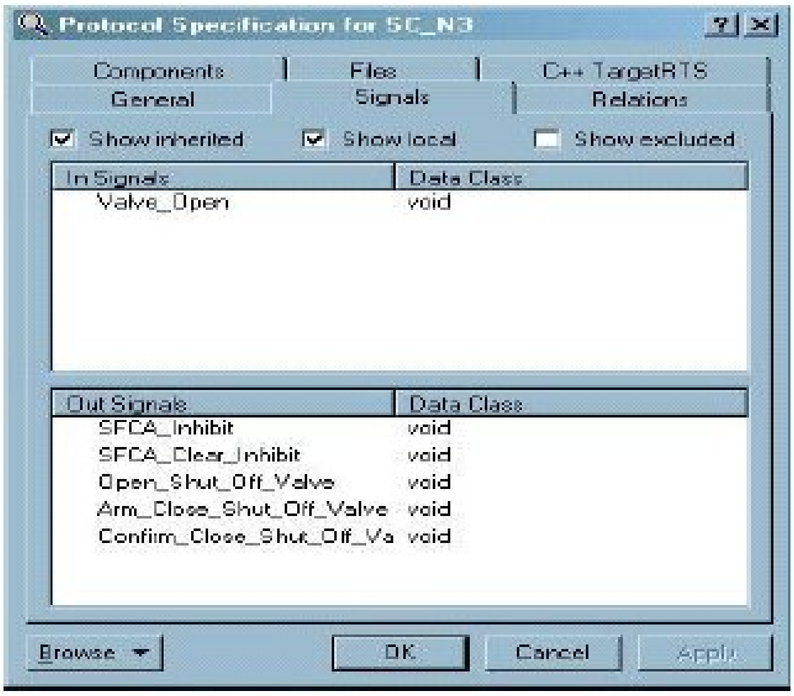


Figure 14: Subsystem Z: Command and Control System



☑ Show inherited ☑ Show local ☐ Show excluded

In Signals	Data Class
Valve_Open	void

Out Signals	Data Class
SFCA_Inhibit	void
SFCA_Clear_Inhibit	void
Open_Shut_Off_Valve	void
Arm_Close_Shut_Off_Valve	void
Confirm_Close_Shut_Off_Va	void

Figure 15: Protocol Specification for Component 5

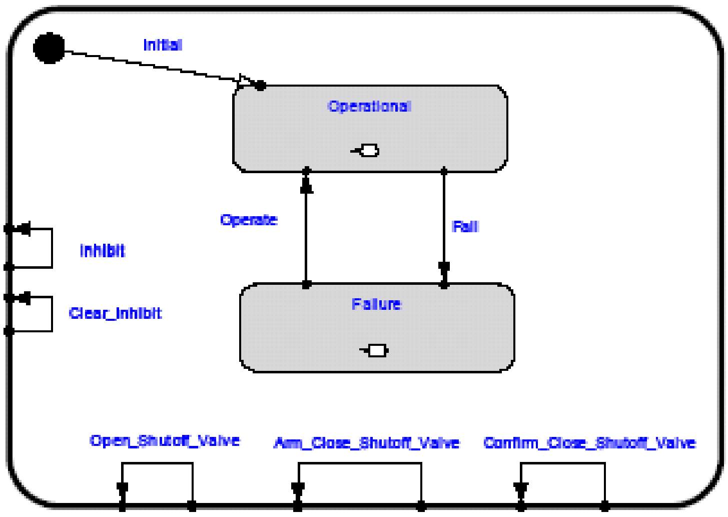


Figure 16: State Diagram of Component 5

Considering the CSCI system discussed above, we get the set of states SB and messages VA>B from the artifacts of the system specification. We obtain the matrix EP of (conditional) error propagation probabilities of this system, using the approximation. We assume equi-probability of states and messages.

As an example, we will demonstrate how to compute EP(1,5). Component 5 has SB = 2 from Figure 4.3, and VA>B =5 from Figure 4.4. So using the approximation, we get EP(1,5) =(1-0.5)/(1-0.2) = 0.625. Thus, the 1-to-5 error propagation cannot exceed 0.625.

For this particular case study, we have derived the connector activation matrix ***T*** as a stochastic matrix of probabilities that contains for each entry (A,B), the probability that connector (A,B) is activated, given that component A is broadcasting a message**.** Using this connector activation matrix, we derive the *unconditional error propagation matrix* EA***,*** also referred to as the 1-step error propagation matrix of the system. We get the matrix ***T*** through a simulation of the system representing the operational profile of the execution. Continuing our example, we got T(1,5) =0.023.

So, the probability that connector (1,5) is activated, given that component 1 is broadcasting a message is 0.023. Then, the *unconditional error propagation* EA (1,5) = 0.625 *0.023= 0

Error propagation matrix for this case study is:

Table 9: Conditional Error Propagation Matrix - Analytical Results

		B									
		C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
A	C1	1.0000	0.1061	0.4210	0.3368	0.4472	0.4623				
	C2	0.2001	1.0000						0.5238		
	C3	0.0105	0.4722	1.0000							
	C4	0.0190	0.2332		1.0000						
	C5		0.2765			1.0000					
	C6		0.1265				1.0000				
	C7	0.3761						1.0000			
	C8								1.0000		
	C9									1.0000	
	C10	0.0014									1.0000

For this particular case study, we have derived the connector activation matrix ***T*** as a stochastic matrix of probabilities that contains for each entry (A,B), the probability that connector (A,B) is activated, given that component A is broadcasting a message**.** Using this connector activation matrix, we derive the *unconditional error propagation matrix* EA***,*** also referred to as the 1-step error propagation matrix of the system; this is given in Table 10. We get the matrix ***T*** through a simulation of the system representing the operational profile of the execution.

Table 10: Unconditional Error Propagation Matrix - Analytical Results

		B									
		C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
A	C1		0.0012	0.0132	0.0102	0.0146	0.0145				
	C2	0.1104							0.1264		
	C3	0.0060	0.2024								
	C4	0.0107	0.1026								
	C5		0.1005								
	C6		0.0506								
	C7	0.3761									
	C8										
	C9										
	C10	0.0014									

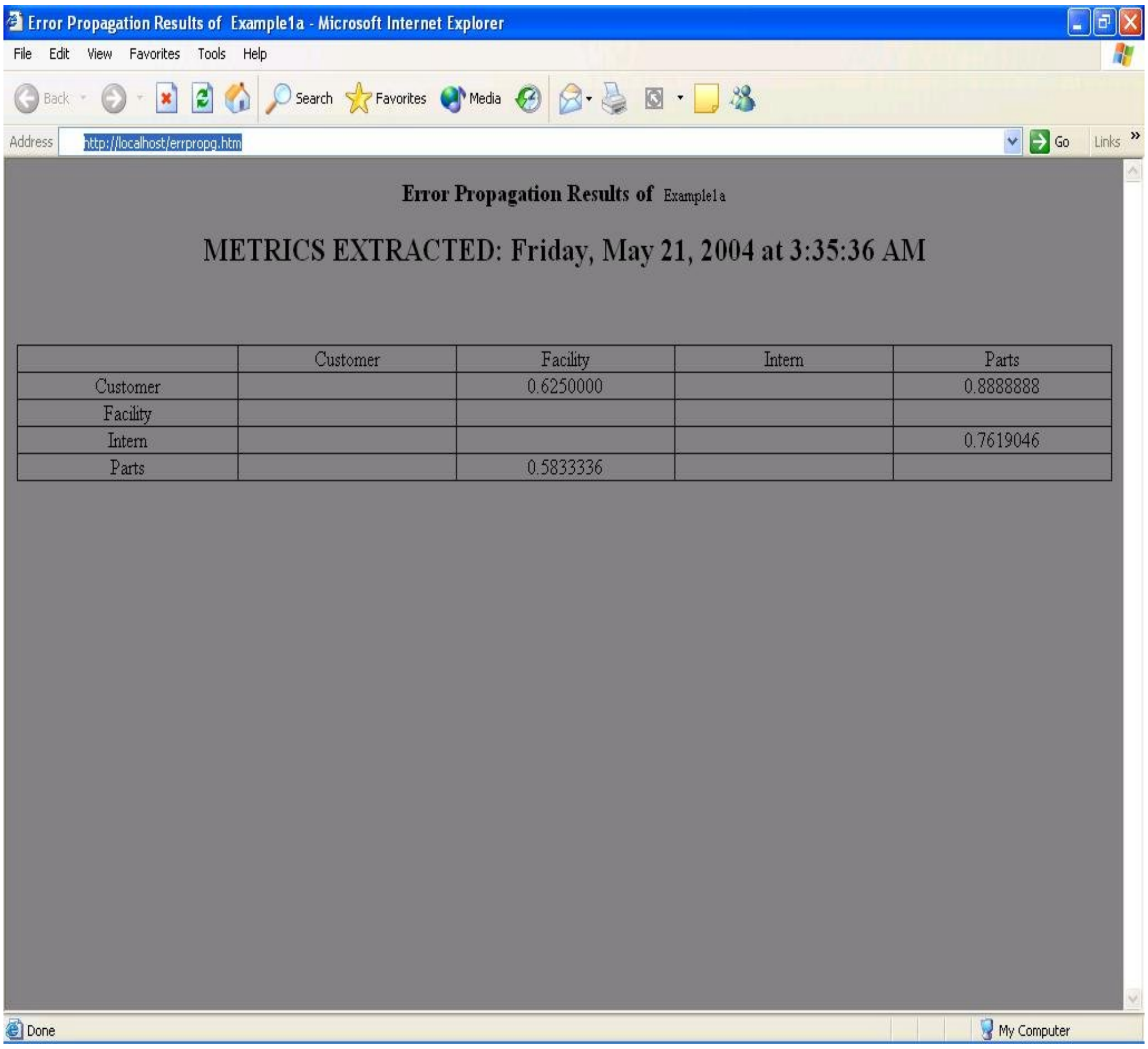
Case Study 2:

The case study has 4 components; two of them have state machines. Component Facility has 2 states and component Parts has 3 states.

To >> Component Facility: Customer has 5 messages passing and the Parts has 7 messages passing from them to the Facility.

To >> Component Parts: Customer has 4 messages passing and the Intern has 8 messages passing from them to the Parts.

Result is given in the below.



Error Propagation Results of Example1a

METRICS EXTRACTED: Friday, May 21, 2004 at 3:35:36 AM

	Customer	Facility	Intern	Parts
Customer		0.6250000		0.8888888
Facility				
Intern				0.7619046
Parts		0.5833336		

Figure 17: Case study output

Case Study 3:

In this example, a small software model, which enables to order commercial items from a web site, was developed. As for the calculation of Error Propagation Metrics only the collaboration diagram and state diagrams will be considered. In this example, total number of 5 classes and 25 messages are present with 4 methods (Figure 18, Table 11). Although there is a message from 'Stock Item' to 'Reorder

Item' due to miss-information in UML Specification, the message will be neglected (Table 11).

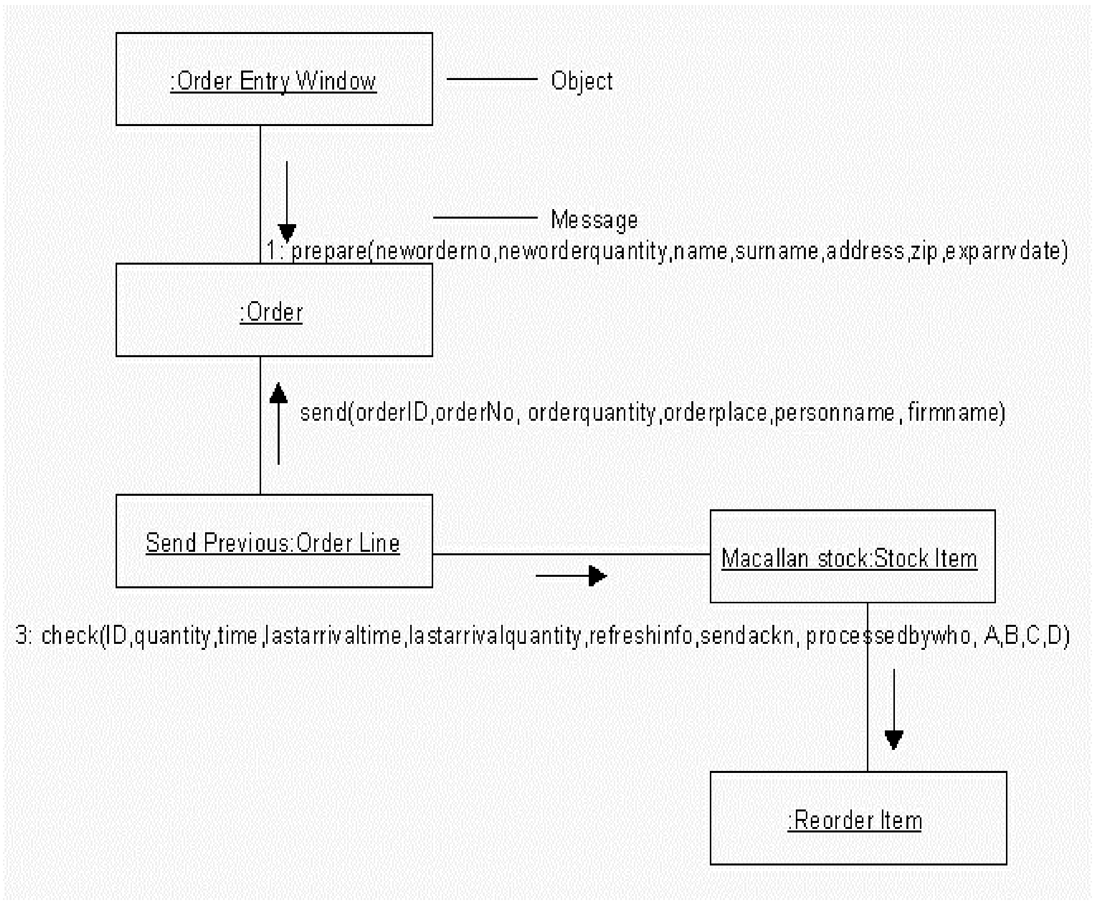

:Order Entry Window
Object
Message
1: prepare(neworderno,neworderquantity,name,surname,address,zip,exparrvdate)
:Order
send(orderID,orderNo, orderquantity,orderplace,personname, firmname)
Send Previous:Order Line
Macallan stock:Stock Item
3: check(ID,quantity,time,lastarrivaltime,lastarrivalquantity,refreshinfo,sendackn, processedbywho, A,B,C,D)
:Reorder Item

Figure 18: Collaboration Diagram of the Case Study 3

Table 11: Total number of Messages between components

Messages: TO (>) From(v)	Order	Stock Item	Reorder Item	State Diagram
Order Entry Window	7	0	0	No
Order Line	6	12	0	No
Stock Item	0	0	(?)	Yes
Order	0	0	0	Yes

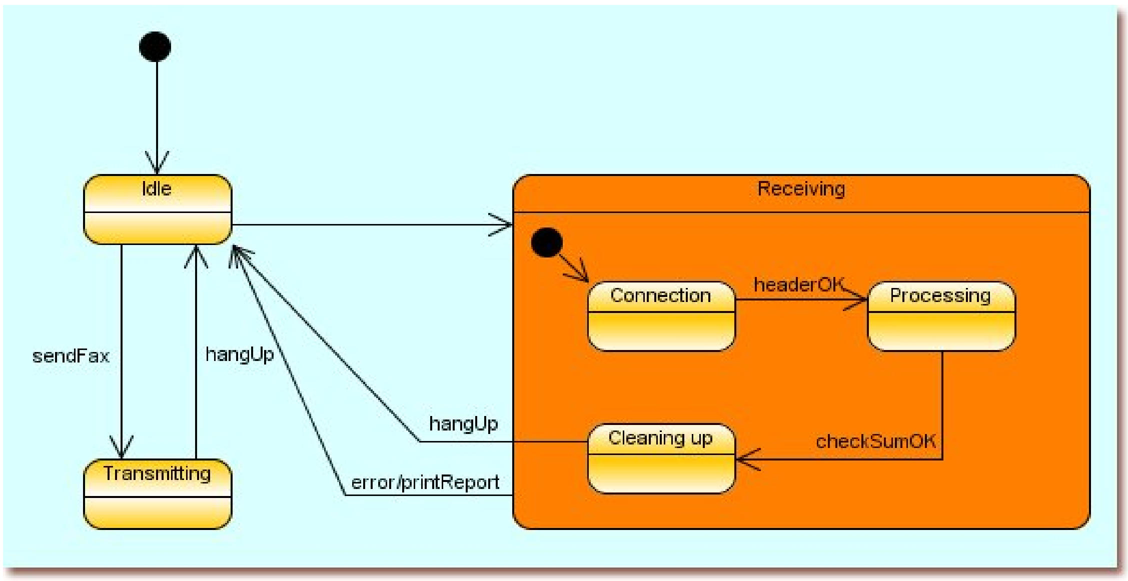


Figure 19: State Diagram of the Order Component

The system has two state diagrams. First state diagram was built to show the behaviour of the Order component (Figure 19). Second state diagram is part of the 'Stock Item' class to reflect the processes that is going inside of the particular component (Figure 20).

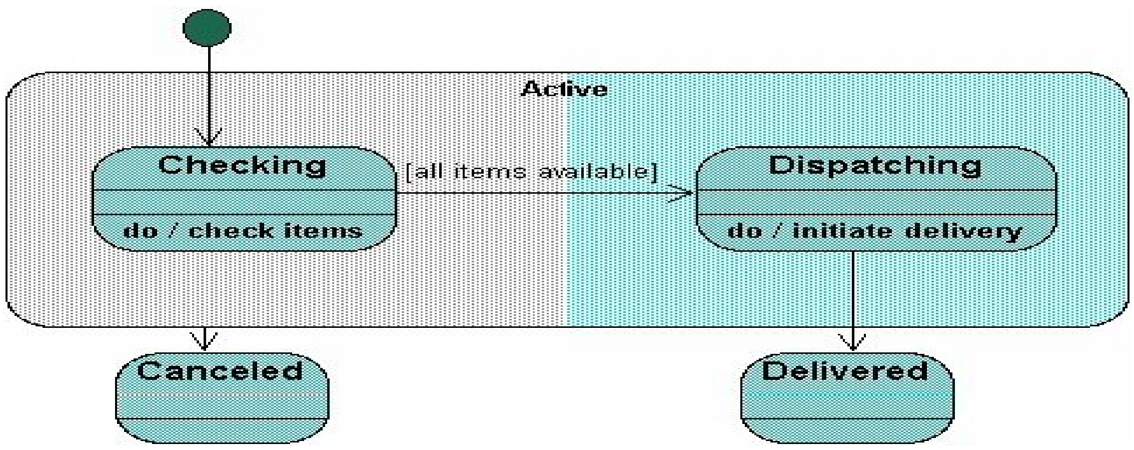
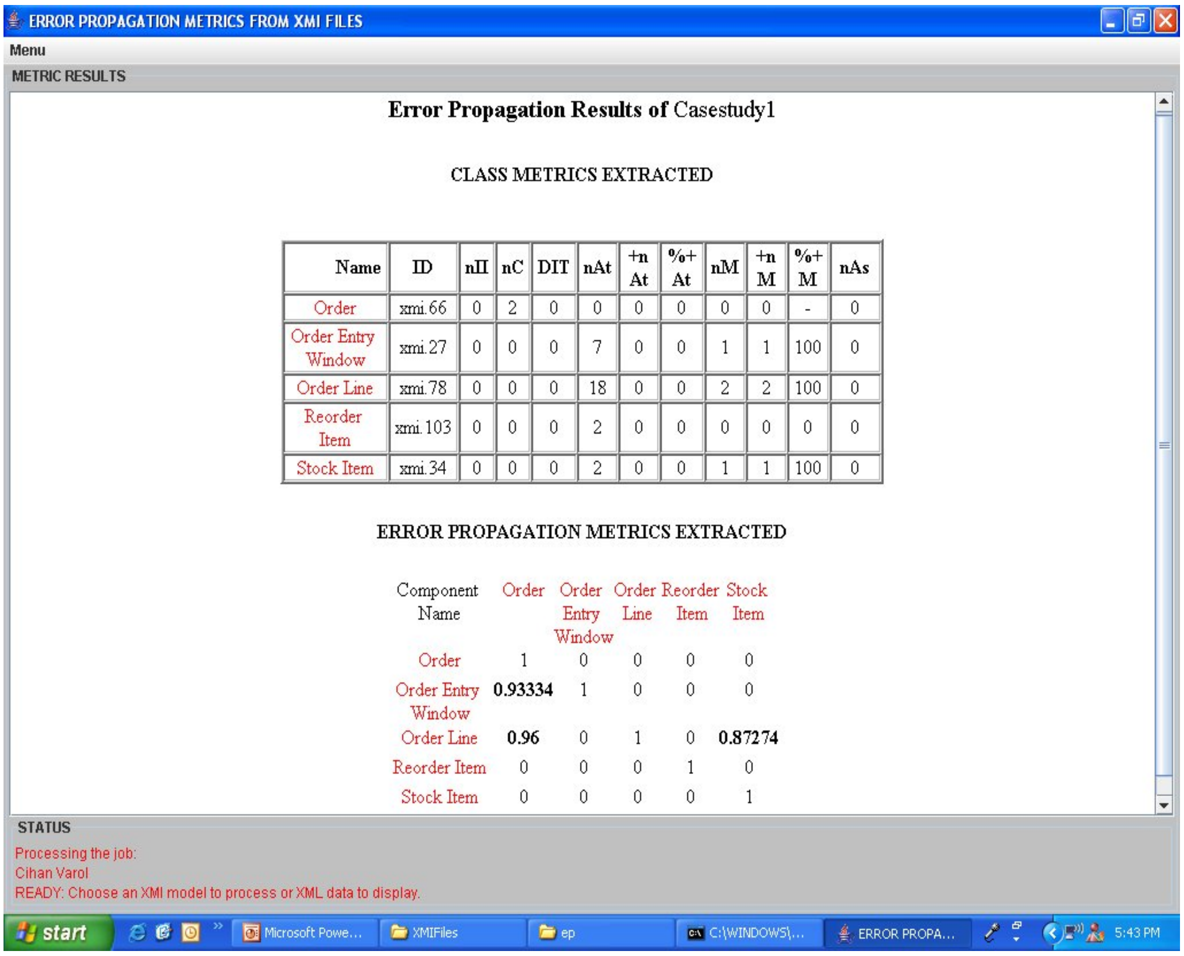


Figure 20: State Diagram of the Stock Item

After exporting this diagrams to XMI format. The relevant information was obtained with XSLT to provide the Error Propagation Metrics. Both the empirical results of the system can be seen from Table 12 and the tool output from Figure 22.

Table 12: Error Propagation Probabilities Between Components

C. Name	Order	Order Entry Window	Order Line	Reorder Item	Stock Item
Order	1	0	0	0	0
Order Entry Window	**0.93334**	1	0	0	0
Order Line	**0.96**	0	1	0	**0.87274**
Reorder Item	0	0	0	1	0
Stock Item	0	0	0	0	1



Error Propagation Results of Casestudy1

CLASS METRICS EXTRACTED

Name	ID	nII	nC	DIT	nAt	+n At	%+ At	nM	+n M	%+ M	nAs
Order	xmi.66	0	2	0	0	0	0	0	0	-	0
Order Entry Window	xmi.27	0	0	0	7	0	0	1	1	100	0
Order Line	xmi.78	0	0	0	18	0	0	2	2	100	0
Reorder Item	xmi.103	0	0	0	2	0	0	0	0	0	0
Stock Item	xmi.34	0	0	0	2	0	0	1	1	100	0

ERROR PROPAGATION METRICS EXTRACTED

Component Name	Order	Order Entry Window	Order Line	Reorder Item	Stock Item
Order	1	0	0	0	0
Order Entry Window	0.93334	1	0	0	0
Order Line	0.96	0	1	0	0.87274
Reorder Item	0	0	0	1	0
Stock Item	0	0	0	0	1

STATUS

Processing the job:
Cihan Varol
READY: Choose an XMI model to process or XML data to display.

Figure 21: Case study 3- Screenshot of the tool

5 APPLICATION DESIGN AND IMPLEMENTATION

The architecture of the system is based on XML and its family (XMI, XSLT) (Diagram 3). Although it may be possible to integrate database to the system, using database will increase the processing time of a particular case study. Moreover, it won't solve the lack of capacity of XMI documents. Therefore, in this system the information is stored in XML files.

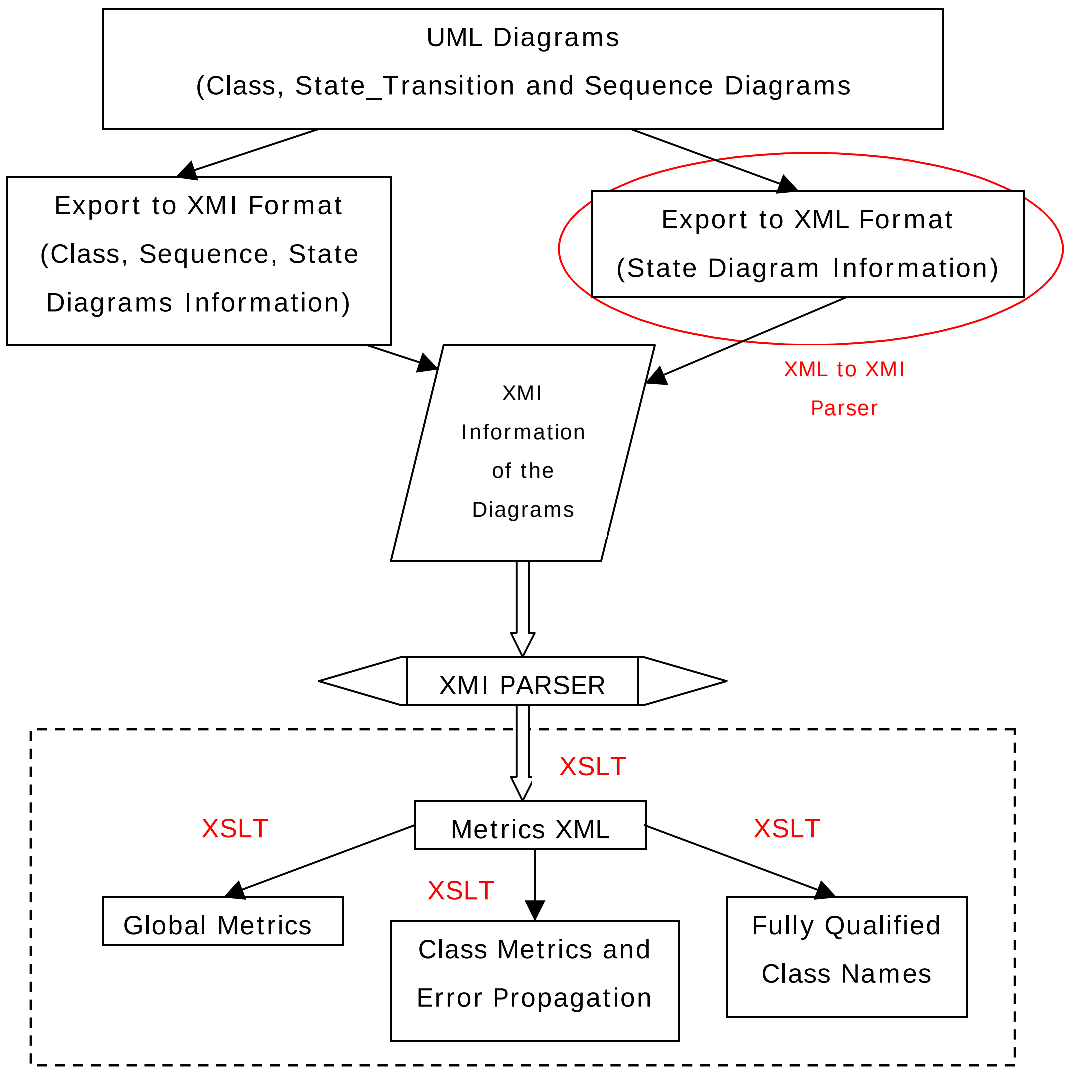


Diagram 3: System Architecture

A Java application was created to load, process and display the data extracted from XMI format file input. The underlying data processing for the application is performed by XSLT stylesheet transformations of XML files from one format to another. Use of XSLT technology allows for rapid extensions or alterations to the processing events, by purely textual editing of the stylesheets.

Each UML tool its own specification. Therefore provided XMI documents have their own structure, which means currently there is no common XMI specification for the UML diagrams. The designed system is built for work on Visual UML and Rational Rose RT programs. All the information for calculating the error propagation from Visual UML is gained from the XMI document and saved in an XML file with using XSLT stylesheet. Finally that information is presented in the tool. For Rational Rose RT, it is impossible to calculate the number of states per individual class. The only state diagram information can be get from Rational Rose RT's XMI document is the total number of states in the whole system. To overcome this problem, Class diagram and Collaboration diagrams information are calculated from the XMI document that is supported by Rational Rose RT, and the number of states in one component is calculated by the XML document provided by Rational Rose RT. Then, the information acquired from both XMI and XML documents are represented as one common XMI document to provide the Error Propagation Metrics.

Several sources of variability mean that: a robust process cannot be fully automated; the various tools support and create output using various versions of the XMI and UML standards; there are subjective differences in the level of detail and style of UML models produced using software design tools; and these tools do not reliably implement the same XMI standards to produce identical XMI files.

5.1 XML Transformations

The underlying data processing for the application is performed by XSLT stylesheet transformations of XML files from one format to another.

These transformations are

- XMI input to metrics XML
- metrics XML to summary HTML

  Ranked by:

  - number of methods (nM)
  - number of attributes (nAt)
  - number of states (nS)
  - number of messages passing each other (nMS)
  - number of associations (nAs)
  - number of children (nC)
  - depth in inheritance tree (DIT)
  - outlier status

In order to design the stylesheets to accomplish these transformations it is first necessary to consider which metrics are stored within the XMI format.

5.1.1 Metrics available from XMI files

From the preceding discussions it is possible to list the metrics which should theoretically be extractable from a standard XMI representation of a projects UML diagrams.

For Individual Classes:

nM	Number of Methods (locally defined or redefined)
+nM	Number of Public Methods
nS	Number of states
nMS	Number of messages passing each other (nMS)
nAt	Number of Attributes
+nAt	Number of Public Attributes
nC	Number of Children (direct subclasses)
DIT	Depth in Inheritance Tree
nAs	Number of Associations (non-inheritance dependencies)
nII	Number of Implemented Interfaces

Dissection of the types of associations would be complex, but potentially possible from sufficiently detailed models, as would further information on inheritance encapsulation.

5.1.2 Metrics available from XMI files using XSLT

An XSLT stylesheet, primaryProcess.xsl has been designed that is capable of enumerating basic class and global metrics from an XMI input file. This XSLT sheet uses a large number of template rules to extract the class metrics shown in Table 11, each value being saved as a new XML element in the result tree. Some problems and limitations to the technology that came to light are discussed in the Evaluation Section 2. Due to space limitations no attempt is made to detail the entire XSLT template rules used to extract these metrics, but a number of examples are shown in the following section.

Table 13: List of First Pass Metrics

GlobalMetrics	**ClassMetrics**
DateLastModified	(for each Class)
TimeLastModified	Name
FileName	implementedInterfaces
Title	numberChildren
DateProcessed	depthOfInheritance
TimeProcessed	numberAttributes
NumberOfClasses	publicAttributes
NumberOfUserClasses	percentPublicAttributes
NumberOfPackages	numberOperations
NumberOfUserPackages	publicOperations
NumberOfInterfaces	percentPublicOperations
NumberOfInheritanceTrees	numberAssociations
NumberOfOrphanClasses	ID
	fullName
InterfaceMetrics	numberofStates in a component
(for each Interface)	Numberof Messages passing each other
IFName	Operations
Implementations	

5.1.3 XSLT Template Rules for Metric Extraction

Some representative template rules are presented here to demonstrate the principles of the XSLT process (also refer to Sections 2-3).

5.1.3.1 Simple Value Copying

The template simply returns the value of element specified by the XPath expression, which is written to the result tree in the position from which the template is called:

```
<xsl:template name="Title">
  <Title>
   <xsl:value-of select=
            "//Model_Management.Model[@xmi.id]/
              Foundation.Core.ModelElement.name"/>
  </Title>
</xsl:template>
```

this is called by:

```
<xsl:call-template name="Title" />
```

Causing a complete <Title> element to be written.

5.1.3.2 Simple Counting Functions

Standard XSLT functions can count the number of occurrences of nodes matching an XPath pattern. In this case the xmi.id of a class is passed to the template, to allow counting of all the associations referenced in this class. The counting is simplified by assigning the node set matching the pattern to a variable.

```
xsl:template name="associations">
  <xsl:param name="source"/>
    <xsl:variable name="association_ends" select=
                  "//Foundation.Core.AssociationEnd[Foundation.Core.
                        AssociationEnd.type/*/@xmi.idref=$source]"/>
      <xsl:value-of select="count($association_ends)" />
</xsl:template>
```

In this case the template is called from within the definition of the new element <numberAssociations>, within the nested elements <ClassMetrics><Class>:

```
<xsl:element name="numberAssociations">
      <xsl:call-template name="associations">
            <xsl:with-param name="source" select="@xmi.id"/>
      </xsl:call-template>
</xsl:element>
```

5.1.3.3 Standard Java Extensions

Used here to record the current date and time of metric extraction, using standard Java package functions to obtain and format Date instances. The stylesheet must define namespaces for these functions:

```
      <xsl:stylesheet.................
                  xmlns:Date="xalan://java.util.Date"
                  xmlns:Format="xalan://java.text.DateFormat"
                                                     .....>
```

The template rule calls these functions to write a Date instance to the today variable, and a formatting object to the dateFormatter variable and then returns the

```
xsl:template name="date">
        <xsl:variable name="today" select="Date:new()"/>
        <xsl:variable name="dateFormatter"
                   select="Format:getDateInstance(FULL)"/>
        <xsl:value-of select="Format:format($dateFormatter,$today)"/>
        <xsl:fallback>
                  <xsl:text> Java Extension for Date is not
                  available</xsl:text>
        </xsl:fallback>
```

desired value by calling the format() method on these two arguments:

Thus when the template is called within the definition of the new element <DateProcessed> the current date is written to the result tree:

```
<xsl:element name="DateProcessed">
            <xsl:call-template name="date" />
      </xsl:element>
```

5.1.3.4 User Defined Extensions

User defined extensions can be called in a very similar fashion to standard extensions, but for this application it was more efficient to use the Xalan-specific extension mechanism has been used which bundles several methods to be called as an *lxslt component*:

```
       <xsl:stylesheet.................
                  xmlns:lxslt="http://xml.apache.org/xslt"
                  xmlns:readData="metrics2.ReadFile"
                  extension-element-prefixes="readData"
                                                 ...........>
```

These methods return the desired file name, creation date and time details from a temporary file written within the Java application before it invokes the XSLT processing.

```
      <lxslt:component  prefix="readData"
                        elements="init"
                        functions="getAgeDate getAgeTime getFileName">
            <lxslt:script lang="javaclass"
                  src="xalan://metrics2.ReadFile"/></lxslt:component>
```

By calling the init() method of the ReadFile class an instance of the class is created which has read the desired time, date and filename details from the temporary file, so that the values can then be returned simply by calling the appropriate method:

```
      <readData:init />
        <xsl:element name= "DateLastModified">
            <xsl:value-of select="readData:getAgeDate()"/>
        </xsl:element>
```

The information provided by these functions will be particularly important for keeping track of archived metrics from different versions of a given XMI project.

5.1.3.5 Recursive Templates

Recursion is used heavily in XSLT processing, as template rules are repeated for each node matching an XPath pattern (for example each class is processed in turn when matching

<xsl:for-each select="//Foundation.Core.Class[@xmi.id]">

As XSLT is a purely declarative language, a variable can only be assigned once, and not have its value modified. Limitations that this imposes can often be overcome by assigning the value of a recursive loop to a variable, so that the outer variable is only assigned after the inner recursion has terminated. This has been used to count how deep a class is in its inheritance tree (shown here), and also to concatenate the names of a class's ancestors in order to generate a fully qualified class name. In the example below the template is called recursively, passing-in the xmi.id of the current class to step up the inheritance tree, incrementing the exported value by one at each level:

```
<xsl:template name="inheritance">
  <xsl:param name="refid"/>
  <xsl:variable name="generalizations"
      select="..//Foundation.Core.Class[@xmi.id=$refid]/
                  Foundation.Core.GeneralizableElement.generalization/
                  Foundation.Core.Generalization"/>
  <xsl:variable name="gen_ref"
      select="..//Foundation.Core.Class[@xmi.id=$refid]/
                  Foundation.Core.GeneralizableElement.generalization/*/
                  @xmi.idref"/>
  <xsl:variable name="parent_id"
      select="//Foundation.Core.Generalization[@xmi.id=$gen_ref]/
                  Foundation.Core.Generalization.parent/*/@xmi.idref"/>
  <xsl:choose>
      <xsl:when test="count($generalizations) > 0 and
                  boolean(//Foundation.Core.Class[@xmi.id=$parent_id])">
         <xsl:variable name="counter">
            <xsl:call-template name="inheritance">
                  <xsl:with-param name="refid" select="$parent_id"/>
            </xsl:call-template>
        </xsl:variable>
        <xsl:value-of select="1 + $counter" />
      </xsl:when>
      <xsl:otherwise >0</xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

5.1.4 Derived Metrics by Chained XSLTs

In order to produce the error propagation metrics it is necessary to count, combine and obtain ratios of the various metrics obtained above. Whilst it should be possible to define complex template rules to derive these metrics within the single initial stylesheet, by using the XML output of the first XSLT as input for a second transformation simpler template rules can be used. In addition this second stylesheet, secondaryProcess.xsl, can copy the entire primary output into the secondary output to combine the original and derived metrics in a single XML document.

The template rules for the chained processing event are much simpler, as they define simple recombinations of the elements created in the first transformation, and consequently have simpler XPath expressions. The only point of importance to note is that empty, nil and null values have to be allowed for in the calculations, and return ' 0 ' or ' - ' (undefined) if appropriate.

For example the following variable assignment to record the maximum number of methods (operations) per interface returns 0 if there are no Interface operations defined in the project:

```
      <xsl:variable name ="numberofStates">
            <xsl:if test="sum(//State_Machine/State)=0">0</xsl:if>
            <xsl:for-each select="//State_Machine/State">
            <xsl:sort data-type="number"/>
                        <xsl:if test="position()=last()">
                              <xsl:value-of select = "."/>
                        </xsl:if>
            </xsl:for-each>
      </xsl:variable>
```

Table 12: List of Derived Metrics

Name of Component	A	B	C	D
A	1	Error Prop	Error Prop	Error Prop
B	Error Prop	1	Error Prop	Error Prop
C	Error Prop	Error Prop	1	Error Prop
D	Error Prop	Error Prop	Error Prop	1

5.1.5 A Document Type Definition (DTD) for metrics XML

The allowed structure of an XML document can be defined in a declared Document Type Definition (DTD). Parsers may then validate a given XML document against its declared DTD. DTDs define the allowable elements, attributes, entities and notations for a document. They therefore define the tag and data structure followed by XML documents conforming to the DTD.

While not essential for creating and parsing 'well-formed' XML documents, defining a DTD provides a useful reference structure for the data. The simplest DTDs merely list the allowable elements, and their allowable contents (further elements or 'parsed character data' (PCDATA)).

Defining attributes for elements allows more information to be stored, often metadata or data of secondary importance. Attributes can be more restrictively defined than elements, and may be of one of a limited number of given types, and can be given default and alternative values. However datatypes are not well supported in DTDs, and require the more expressive XML Schema to describe document structure.

It is desirable to create a simple XML DTD to define the metrics XML document produced from XMI by the chained stylesheet transformations, primaryProcess.xsl followed by secondaryProcess.xsl. This will allow a parser to check the validity of the resultant XML, and provide a reference document for the metrics contained in the file.

As this XML format is a data repository, acting as an intermediary for further processing it was desirable to keep the structure as simple as possible, to facilitate downstream processing. For this reason all the extracted metrics are stored as the value of individual elements, and no attributes used. The file *metric.dtd* therefore is merely a list of the allowable element tags, which contain other elements or PCDATA. The nested structure of a *metric.dtd* conformant document is shown.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE Metrics SYSTEM "metric.dtd">
     - <Metrics>
       - <FirstPassMetrics>
         + <GlobalMetrics> contains elements holding Global Metrics
           - <InterfaceMetrics>
             + <Interface> contain elements holding Interface Metrics
           </InterfaceMetrics>
           -  <ClassandErrorPropMetrics>
             + <Class> contain elements holding Class Metrics
             + <Class> contain elements holding Error Propagation
      Metrics
             </ClassMetrics>
         </FirstPassMetrics>
      + <DerivedMetrics> contain elements holding Derived Metrics
   </Metrics>
```

`metric.dtd` fully describes the allowed elements, tags and structures for the XML structure shown above. Once written into the XML DOCTYPE definition this DTD must be present for the XML parser to create a DOM tree from the XML document. However, the parser will only check the document structure against the DTD if it has DTD validation enabled. (By default the XML processing performed by the *Metrics from XMI* application is non-validating, although it was felt useful to have a validation option provided so that any non-functional XMI or XML documents could be investigated.)

metric.dtd

```
<!ELEMENT Metrics (FirstPassMetrics, DerivedMetrics)>

<!ELEMENT FirstPassMetrics (GlobalMetrics,
InterfaceMetrics, ClassMetrics)>

<!ELEMENT GlobalMetrics (DateLastModified,
TimeLastModified, FileName, Title, XMI.exporter?, DateProcessed, TimeProcessed,
NumberOfClasses, NumberOfUserClasses, NumberOfPackages, NumberOfUserPackages,
NumberOfInterfaces, NumberOfInheritanceTrees, NumberOfOrphanClasses)>

<!ELEMENT DateLastModified (#PCDATA)>
<!ELEMENT TimeLastModified (#PCDATA)>
<!ELEMENT FileName (#PCDATA)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT XMI.exporter (#PCDATA)>
<!ELEMENT DateProcessed (#PCDATA)>
<!ELEMENT TimeProcessed (#PCDATA)>
<!ELEMENT NumberOfClasses (#PCDATA)>
<!ELEMENT NumberOfUserClasses (#PCDATA)>
<!ELEMENT NumberOfPackages (#PCDATA)>
<!ELEMENT NumberOfUserPackages (#PCDATA)>
<!ELEMENT NumberOfInterfaces (#PCDATA)>
<!ELEMENT NumberOfInheritanceTrees (#PCDATA)>
<!ELEMENT NumberOfOrphanClasses (#PCDATA)>

<!ELEMENT InterfaceMetrics (Interface*)>
<!ELEMENT Interface (IFName, Implementations,
Operations, ID)>
<!ELEMENT IFName (#PCDATA)>
<!ELEMENT Implementations (#PCDATA)>
<!ELEMENT Operations (#PCDATA)>
<!ELEMENT ClassMetrics (Class*)>
<!ELEMENT Class (Name, implementedInterfaces,
numberChildren, depthOfInheritance, numberAttributes, publicAttributes, percentPublicAttributes,
numberOperations, publicOperations, percentPublicOperations, numberAssociations, ID, fullName)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT implementedInterfaces (#PCDATA)>
<!ELEMENT numberChildren (#PCDATA)>
<!ELEMENT depthOfInheritance (#PCDATA)>
<!ELEMENT numberAttributes (#PCDATA)>
<!ELEMENT publicAttributes (#PCDATA)>
<!ELEMENT percentPublicAttributes (#PCDATA)>
<!ELEMENT numberOperations (#PCDATA)>
<!ELEMENT publicOperations (#PCDATA)>
<!ELEMENT percentPublicOperations (#PCDATA)>
<!ELEMENT numberAssociations (#PCDATA)>
<!ELEMENT  ID (#PCDATA)>
<!ELEMENT fullName (#PCDATA)><!ELEMENT DerivedMetrics (Total_Interfaces, Numberof
Numberofmeesages, number of states,)>

<!ELEMENT  Number_of_States in a component  (#PCDATA)>
<!ELEMENT  Number of messages between components (#PCDATA)>
```

5.1.6 HTML from XML

While the metrics XML data format is ideal for storing metrics derived from XMI representations of UML models, it does not allow easy inspection of the data. XSLT readily allows XML source documents to be converted to HTML or plain text format. It is therefore possible to present different views of the metrics XML data via separate stylesheet transformations.
Seven different views have initially been developed for display by the 'Metrics for XMI' application.

The **Global Metrics Summary** view is created by summary_xml2html.xsl, which tabulates all of the global, class and interface metrics. Header information and up to four tables are created by mixing XSLT template calls within HTML tags. The header details name, file and last modification details for the XMI project. The first table ('GLOBAL METRICS') merely lists all of the global metrics, returned as the value of the metrics XML elements. No further calculations are necessary for this presentation, although a number of tests are performed to identify and skip null fields. The second table ('CLASS METRICS and ERROR PROPAGATION') lists all the classes alphabetically, with their metrics and error propagation values.

In addition each stylesheet also reproduces the 'FULLY QUALIFIED CLASS NAMES' table described above. Tables of interface metrics are only presented when sorted by number of methods and number of children (implementations).

6 CONCLUSION

In this thesis, first we look to the issue of general metrics and how will be extracted from UML class diagram then we have derived an analytical approach to estimate the probability of error propagation between components in software architecture. Further, we have illustrated our proposed formula by means of a fault injection experiment, applied on a large command and control system, and found a fairly meaningful correlation between our analytical estimates and our experimental observations. Given that our analytical approach is based on architecture specifications, and uses exclusively information that is typically available at an architectural level, we submit that our result can be used to estimate the error propagation behaviour of architecture, at a time when relatively little is known about the actual execution of products that instantiate the architecture. In addition to providing the basic conditional probability of error propagation over a given connector (conditioned on the activation of the connector), we have also provided analytical formulas for unconditional error propagation (which incorporate the probability of connector activation). Then, we also considered automating our architectural analysis tool to support the automatic computation of error propagation probabilities.

7 REFERENCES

Abreu, B.F., Goulao, M. and Esteve, R. (1995) Toward the design quality evaluation of OO software systems. Fifth International Conference on Software Quality.

Akif, M., Brodheas, S., Cioroianu, A., Hart, J., Jung, E. and Writz, D., (2001) Java XML: Programmers's Reference. Wrox Press. ISBN 1-861005-20-2

Albrecht, A. J. and Gaffney, J. (1983) Software function, source lines of code, and development effort prediction: A software science validation. IEEE Transactions on Software Engineering 9, 639.

APACHE (2002) http://xml.apache.org/xalan-j/samples.html#appletxmltohtml

Ammar, H., Yacoub, S. M, Ibrahim, A., "A Fault Model for Fault Injection Analysis of Dynamic UML Specifications," *International Symposium on Software Reliability Engineering*, IEEE Computer Society, November 2001.

ArgoUML (2002) http://argouml.tigris.org/

Badros, G.J. (2000) JavaMl: a markup language for Java source code. Computer Networks 33, 159.

Bansiya, J. and Davis, C.G. (2002) A Hierarchical model for object-oriented design quality assessment. IEEE Transactions on Software Engineering 28, 4.

Basili, V.R., Briand, L.C. and Melo, W.L. (1996) A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering 22, 751.

Bennatan, E.M. (1995) Software Project Management (Second Edition).McGraw-Hill ISBN 007 707648 6

Boehm, B.W. (1981) Software Engineering Economics. Prentice Hall.

Booch, G (2000) Measures of Goodness. Whitepaper published 2000 http://www.rational.com/products/whitepapers/393.jsp

Briand, L.C., Wüst, J., Daly, J.W. and Porter, D.V. (2000) Exploring the relationships between design measures and software quality in object-oriented systems. Journal of Sytems and Software 51, 245.

Briand, L.C.and Wüst, J.(2001) Modeling development effort in object-oriented systems using design properties. IEEE Transactions on Software Engineering 27, 963.

Carlson, D. (2002) Modeling XML aplications with UML: practical e-business applications. Addison Wesley. ISBN 0201709155

Chidamber, S.R. and Kemerer, C.F. (1991) Towards a metric suite for object oriented design. Sigplan Notices 26, 197.

Chidamber, S.R. and Kemerer, C.F. (1994) . A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20, 476.

Chidamber, S.R., Darcy, D.P. and Kemerer, C.F. (1998) Managerial use of metrics for object-oriented software: and exploratory analysis. IEEE Transactions on Software Engineering 24, 629.

Data Access Technologies (2002) http://umltool.d-a-t.com

Deitel, H.M., Deitel, P.J. and Nieto, T.R. (2001) e-Business and e-Commerce: How to Program. Prentice Hall ISBN 0-13-028419-X

el Emam, K., Benlarbi, S., Goel, N. and Rai, S.N. (2001) The confounding effect of class size on the validity of object-oriented metrics. IEEE Transactions on Software Engineering 27, 630.

Fenton, N.E. and Neil, M. (1999) Software metrics: successes, failures and new directions. Journal of Sytems and Software 47, 149.

Fenton, N.E. and Pfleeger, S.L. (1997) Software Metrics: a rigorous and practical approach. PWS Publishing. ISBN 053495425-1

Fioravanti, F. and Nesi, P. (2000) A method and tool for assessing object-oriented projects and metrics management. Journal of Systems and Software 53, 111.

Gentleware (2002) http://www.Gentleware.com

Goldfarb, C.F and Prescod, P. (2001) The XML Handbook (Third Edition) Prentices Hall. ISBN 0-13-055068-X

Grose, T.J., Doney, G.C. and Brodsky, S.A. (2002) Mastering XMI: Java Programming with XMI, XML and UML. OMG Press. ISBN 0-471-38429-1

Halstead, M. (1977) Elements of software science. North-Holland

Harrison, R., Counsell, S. and Nithi, R. (1997) An overview of object-oriented design metrics. Proceedings of the Eighth IEEE International Workshop on Software Technology and Engineering Practice, pp 230 -235

Heiat, A. and Heiat, N. (1997) A model for estimatingefforts requires for developing small-scale business applications. Journal of Systems and Software 39, 7.

Hughes, B. and Cotterell, M. (1999) Software Project Management (Second Edition).McGraw-Hill ISBN 007 709505 7

IBM alphaworks (2002) http://www.alphaworks.ibm.com/tech/xmitoolkit

Ideogramic (2002) http://www.Ideogramic.com

javaboutique.com (2002) http://www.javaboutique.com

jEdit (2002) http://www.jedit.org/

Kay, M. (2001a) IBMDeveloperWoks:XML Zone

Kay, M., (2001b) XSLT: Programmer's Reference (Second Edition). Wrox Press. ISBN1-861005-06-7

Kemerer, C.F. (1987) Empirical validation of software cost estimation models. Communications of the ACM 5, 416.

Li, W. and Henry, S. (1993) Object oriented metrics that predict maintainability. Journal of Systems and Software 23, 111.

Lorenz, M. and Kidd, J. (1994) Object-oriented Software Metrics. Prentice-Hall Object Oriented Series.

McCabe, T. (1976) A software complexity measure. IEEE Transactions on Software Engineering 2, 308.

Marchesi, M. (1998) OOA metrics for the unified modelling language. Software Maintenance and Reengineering, 1998. Proceedings of the Second Euromicro Conference. pp 67-73.

Marinescu, R (2001) Detecting design flaws via metrics in object-oriented systems. Technology of Object-Oriented Languages and Systems, 2001. TOOLS 39. 39th International Conference and Exhibition. p173 -182

Martin, R. (1995) Designing Object-Oriented C++ Applications Using the Booch Method, Prentice Hall. ISBN 0132038374

MetaIntegration Technology Inc (2002) http://www.metaintegration.net/

Nassar, D. M., Rabie, W. A., Shereshevsky, M., Gradetsky, N., Ammar, H.H, Bo Yu, Bogazzi, S., and Mili, A. Estimating Error Propagation Probabilities in Software Architectures. Technical Report, College of Computer Science, New Jersey Institute of Technology 2002. http://www.ccs.njit.edu/swarch/ep.pdf

Nesi, P. and Querci, T. (1998) Effort estimation and prediction of object-oriented systems. Journal of Systems and Software 42, 89.

OBD (2002) www.objectsbydesign.com

OMG (2000) XMI1.1 http://www.omg.org/cgi-bin/doc?formal/00-06-01

OMG (2001) UML1.4 http://www.omg.org/technology/documents/formal/uml.htm

OMG (2002) http://www.omg.org

OMG (2002) XMI1.2 http://www.omg.org/technology/documents/formal/xmi.htm
Putnam, L.H. (1978) A general empirical solution to the macro software sizing and estimating problem. IEEE Transactions on Software Engineering 4, 345.

OPHELIA (2002) http://www.cee.hw.ac.uk/ophelia/index.html

Rational (2000) The Rational Approach. Whitepaper published 2000 http://www.rational.com/products/whitepapers/333.jsp

Rational (2002) http://www.rational.com

Reissing, R (2001), Towards a Model for Object-Oriented Design Measurement. 5th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2001) http://www.iro.umontreal.ca/~sahraouh/qaoose01/Reissing.pdf

Russell, C.R. (2002) RESCU Reverse engineering source code to UML. Honours project report, Department of Computing and Electical Engineering, Heriot-Watt University.

Softeam (2002) http://www.Softeam.fr

SourceForge.net (2002) http://sourceforge.net

Stevens P. and Pooley R.J. (2000) Using UML: software engineering with objects and components. Addison-Wesley. ISBN 0-201-64860-1

Subramanian, G. and Corbin, W. (2001) An empirical study of certain object-oriented software metrics. Journal of Systems and Software 59, 57.

Sun (2002) http://java.sun.com

Symons, C.R.(1988) Function point analysis: difficulties and improvements. IEEE Transactions on Software Engineering 14, 2.

Together (2002) http://www.togethersoft.com/

W3C (1999) XSLT http://www.w3.org/TR/xslt.html

W3C (1999) XPath http://www.w3.org/TR/xpath

W3C (2000) XML http://www.w3.org/TR/2000/REC-xml-20001006

W3C (2001) XSL http://www.w3.org/TR/xsl/

W3C (2002) DOM http://www.w3.org/DOM/

8 APPENDIX

List of Tables:

List of Figures:

List of Diagrams: