The University of Nottingham
School of Computer Science and Information Technology

A Framework for the Computer Based Assessment of Diagram Based Coursework

by Athanasios Tsintsifas, BSc, MA

Thesis submitted to the University of Nottingham for the degree of Doctor of Philosophy, March 2002

To my parents…

Abstract

This research investigates the feasibility and usefulness of designing an authoring environment for diagram-based Computer Based Assessment (CBA). CBA systems present students with an environment that permits them to solve exercises. Upon completion, solutions to the exercises are marked automatically, and the results, along with instructional feedback, are returned to the students. CBA exercises are authored by educators who gain practical and pedagogic benefits from automating the assessment of their students. Until now, facilities to develop CBA have only been implemented for exercises where the answers are text-based.

The deliverables described in this dissertation implement novel solutions to problems in three areas. The first area concerns the customisation of graphic editors to the learning material. The proposed solution is an object-oriented framework for visually specifying and authoring diagram editors. The second area presents techniques for describing the marking criteria of exercises. The proposed solution is a generic marking system designed to support the development of pluggable marking tools for the numerous and inherently different types of diagram domains. The third area is concerned with the integration of the two previous designs with the functions of a CBA system. For this reason provisions were made in the design and implementation of the CourseMaster CBA system. CourseMaster was implemented to improve and replace the Ceilidh CBA system. Ceilidh has been successfully used throughout academia for the assessment of coursework in computer programming.

Evaluation results on diagram-based domains, such as circuit design and software design, indicate that the automation of the assessment of diagrams can be as effective and useful as that of programs. CourseMaster has been available within academia since 1999, providing support for the marking of programming coursework. The diagram authoring extension, entitled DATsys, was integrated into CourseMaster in May 2000. Together they provide a novel and realistic foundation towards authoring, running, and administering diagram-based CBA.

Acknowledgements

It is hard in only a few paragraphs to acknowledge everybody that helped me to produce this work. I attempt my best and apologise dearly for those whom I omit.

Foremost, I would like to thank Eric Foxley, the father of the Ceilidh system and the founder of the Learning Technology Group. Eric has been my former supervisor in the initial years of this research. My sincere gratitude goes to my latter supervisor, Colin Higgins that took under his wings the LTR group from Eric's hands and proved to be a more than worthy successor. The rest of the members of the LTR group, have been very helpful in providing new ideas and support through the years. Cleveland Gibbon gave inspiration and advice when I needed it the most. Pavlos Symeonidis helped in both my work and life and I'm highly indebt for his friendship. Also, I thank Ivan Vaghi and Martin Maya for their input in improving the dissertation.

Of course, a big thanks goes to the students and staff at the University of Nottingham. They have been patient and supported fully the idea of automated assessment even from the first days of CourseMaster back in 1998. It hasn't always been easy, especially in the early days when debugging was a daily occupation.

The help of a number of researchers, which I have met in various conferences, has been essential. Among them, John Vlissides and Erich Gamma influenced my work the most. They taught me design patterns and gave me an understanding that I would not have grasped by only studying their design patterns book. I owe my fascination with computer science to Alan Kay; his talk entitled "The Computer Revolution Hasn't Happened Yet" made a deep impression on me and allowed me to realise that computer science is young; everyone can contribute a little bit.

A special thanks goes to Charmaine McCracken who for the last six years managed to put up with me without complaining for my late-night coding.

Last but not least, I am highly indebt to my beloved family, George, Anna and Lydia that supported me emotionally, spiritually and financially in the difficult days of my seven years residence in Nottingham.

Thank you all, Athanasios Tsintsifas

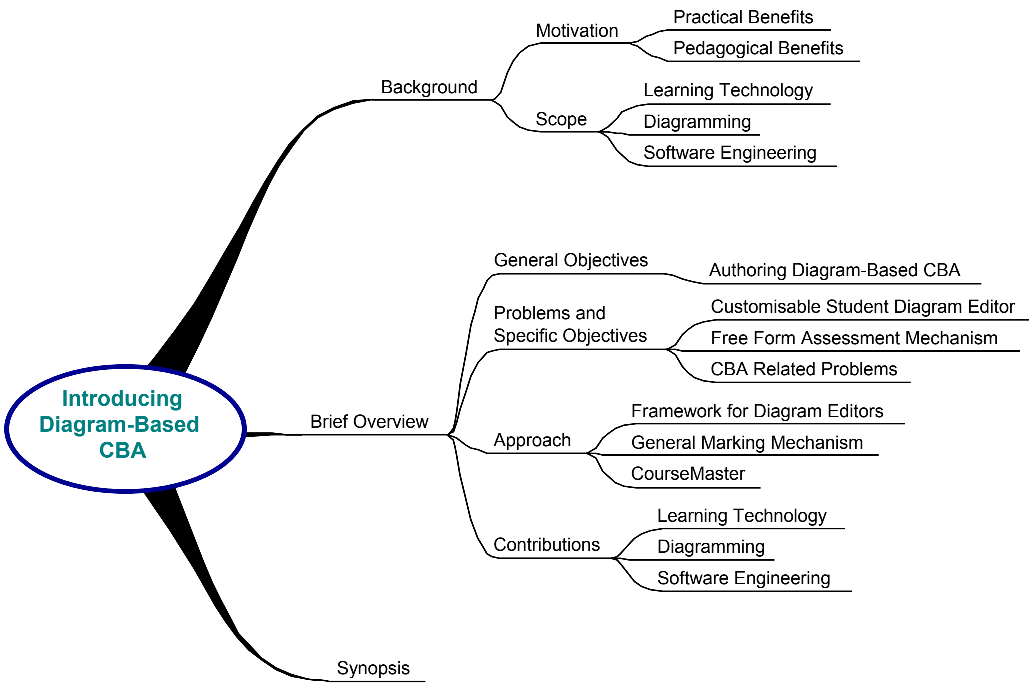
Table of Contents

List of Figures

List of Tables

Chapter 1,

Introducing Diagram-Based CBA



"A picture is worth 1000 words"

Ancient Chinese Proverb

Introduction

Providing higher degree education to an increased proportion of the population is a challenge for academic institutions. As more students enrol each year on courses, educators report the increasing difficulty in managing their class [SCB+96], [CS98], [PS98], [Gt00]. In large class sizes, the traditional means of lecturing, handing out notes and assessing, are limited. Lecturing can be assisted by advances in audio-visual technologies [SRJ97], [AHM99]. The handing out of lecture notes can be minimised, by publishing them as web pages and ensuring that students have access to the Internet.

Educators have limited choices when faced with the task of *assessing* large class sizes [Gg92]. The least desirable option is to reduce the amount of coursework, as good assessment practices ought to be both incremental and redeemable [BR96]. An alternative, given that resources are available, is to employ helpers to grade the submitted work. However, this incurs problems of marking inconsistency, untimely feedback of results and poor coordination between the helpers.

Computer Based Assessment (CBA) systems can alleviate educators from these problems by providing mechanisms to automate the assessment of student work. The Ceilidh system [BBF92] provided functions for the stages of development, deployment, marking and administration of CBA coursework. Focusing on the development aspect, Ceilidh offered generic tools for the authoring of automatically assessed exercises. This has been the most significant reason for its widespread use in the academic world [BBF+93]. Since its conception, Ceilidh has successfully been used as an infrastructure for research and experimentation in a wide range of subjects [FHT98]. It also had direct influence upon many of the CBA systems of the 1990's [JU97], [Dc99], [JL98], [KM00].

Both Ceilidh and related systems lack development tools for coursework that is diagram-based. These tools are needed to research the automation of assessment for standard and non-standard graphical representations such as, for example, flowcharts, circuits and object-oriented design diagrams. The work in this dissertation presents research, design, implementation and evaluation of techniques that facilitate the building of automatically assessed exercises that require solutions given in the form of diagrams.

This chapter introduces the concept of diagram-based CBA. It highlights the thesis' motivation and scope, and gives an overview of the project's core parts. The synopsis explains the structure of this dissertation and how it corresponds to chapters.

1.1 Background

1.1.1 Motivation

A key objective of this research has been to investigate and experiment with the development of diagram-based CBA. Such experimentation involves the authoring, running, marking and administration of diagram-based CBA. No earlier work describes how to combine diagrams with assessment. The reason is that only now, with a synergy of advances in computer hardware and software engineering can the complex task of creating an authoring environment for diagram-based CBA be approached.

In light of a rapid increase in distance learning, research into automatic assessment should not be seen as an option but as a necessity. As Tsichritzis predicts *"We are evolving toward a situation where some universities will go global and become famous while others will be restrained both financially and in scope"* [Td99]. An increasing number of published reports document the pedagogic and practical benefits of automating the assessment process. With appropriate use of CBA, efficiency, reliability and fairness of large scale assessment can improve.

The authoring of diagram-based CBA exercises can be interpreted as a natural step towards expanding what is already possible with Ceilidh. Ceilidh supports the authoring of CBA exercises that employ marking tools for programming, multiple-choice questions, numeric/string simple-answer questions, and free form technical essays. Facilities for the authoring of diagram-based CBA exercises complement the spectrum of possible types for automatic assessment.

1.1.2 Scope

The authoring of diagram-based CBA combines theory and techniques from learning technology, diagramming and software engineering. Figure 1.1 illustrates a high level view of the scope of this work.

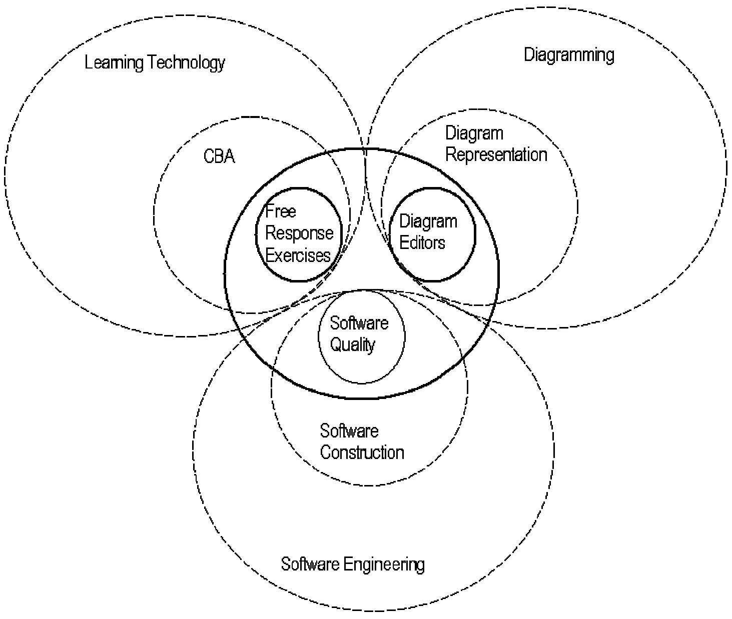


Figure 1.1: A high level view of the dissertation's scope

From a Learning Technology perspective, this research focuses on CBA. Specifically, it concentrates on the *full lifecycle* of *free response* CBA exercises. The full lifecycle incorporates the stages of authoring, running, marking, and administering CBA. Free response CBA exercises offer to the students the opportunity to solve coursework much more freely than selecting predefined options.

In the field of diagramming, two areas have been investigated. The first area is concerned with the structural representation of editors that support various types of diagrams. The second area focuses on the interaction for the manipulation and editing of diagrams. The detection of the commonality and variation of the structure and behaviour among various diagram editors together with requirements and constraints imposed by an orientation towards Learning Technology led to the design of the DATsys framework for authoring diagram editors, the system reported in this thesis.

Constructing software with quality is the focus in Software Engineering. Rangarajan et al, citing the ISO9126 standard for software quality, discuss six characteristics: functionality, reliability, usability, efficiency, maintainability and portability [RSH01]. This thesis supports the view that software quality depends also on the requirements set by the context within which the software is used. In addition to satisfying the six characteristics, high quality software anticipates future changes and thereby increases its longevity.

1.2 Brief Overview

1.2.1 General Objectives

The aim of this research is to investigate, propose, design and evaluate a set of techniques to significantly reduce the development effort required in building automatically assessed diagram-based coursework. The deliverables of this research have pragmatic use. They have been made available to those interested in constructing diagram-based CBA and those employing it for other diagram-based projects.

A number of practical questions set the rationale for this thesis:

- To what extent does diagram based automatic assessment constitute an effective way of marking student exercises?
- What are the advantages and disadvantages in comparison with traditional assessment of diagrams?

Another set of questions originates from the difficulties surrounding the problem of diagram editing:

- To what extent is it possible to generate domain and exercise dependent editors by means of configuring and drawing as opposed to the more difficult task of programming?

- How would a suitable architecture balance a high degree of generality without sacrificing functionality, and what are the limits of the architecture's applicability?

The next set of questions arises while considering a generic marking mechanism:

- Is it possible to classify and model diagram representations in a manner that suits the automation of their assessment in a learning and assessment context?
- What would be a suitable form of specification for marking rules that automatically assess student coursework?
- How can those marking rules return appropriate feedback to the students?
- What advantages would such a system have in contrast with other possible assessment mechanisms?

The final set of questions arises from the need to evaluate diagram-based CBA in a real environment:

- How does diagram-based CBA perform?
- Most importantly, what are the effects on the learning and assessment processes?

Providing authors with a suitable authoring environment for diagram-based CBA is a crucial objective and a necessary step in attempting to answer the above questions.

1.2.2 Problems and Specific Objectives

Three major problem areas need to be considered to answer the questions set in the general objectives.

The first problem area is concerned with the building of a graphical editor customised to support the exercise and its domain. Software engineering research has documented methods for the construction of generic diagramming editors over the past 10 years [Vj90], [Jr92], [Bj95], [BG97]. The aim for most of these methods has been to empower programmers with libraries powerful enough to considerably ease the development of domain specific graphical editors such as circuit, flowchart, and drawing editors. Yet, these libraries are complex and demand significant programming expertise. The effort involved in producing a single "per domain" or "per exercise" editor, is prohibiting their use in a CBA context where the time and resources spent in development have to be minimised. The objective of this first problem area is to design and implement a framework for generic diagram editors in which the creation of a new type of editor is a simple task that does not require programming.

The second problem area is concerned with the marking process. It extends to the description and execution of the marking criteria and the creation of appropriate feedback. The automatic assessment of diagrams belongs to the "free response" type of assessment. This is because the solution space of a diagram-based exercise has potentially infinite number of solutions with varying degrees of correctness. Foxley and Zin have described a generic technique to express marking schemes for assessing exercises in programming languages using "Oracles" [FZ93]. Oracles represent the marking criteria that are used to mark an exercise. They are described in a form that uses regular expressions and they express the text matching behaviour that can conveniently be used to assess an exercise solution. The objective of the second problem area is to investigate whether a similar technique can be used to describe marking schemes for diagram-based coursework.

The third problem is concerned with the provision of support for the full lifecycle of diagram-based CBA exercises. Initially this research started with the intention of extending the Ceilidh CBA system with a diagram-based type of exercise. However, for many reasons, including a need to increase scalability, performance, maintainability, extensibility and usability, Ceilidh had to be redesigned and re-implemented. The successor of Ceilidh is named CourseMaster [FHH+01]. While restructuring Ceilidh into CourseMaster, the objectives have been to integrate the diagram-based facility with the generic marking mechanism and to increase software quality.

1.2.3 Approach

A series of domain dependent graphical editors were developed to acquire a practical understanding of diagram editing. The experience gained from these prototypes indicated the commonality in the structure of diagram editors across multiple domains. A series of attempts were made to define a common base as an object-oriented framework from which a family of domain dependent diagram editors could be derived. The initial attempts had a high degree of structural complexity and redesign was necessary. In each subsequent version, the objective was to maintain the architectural decisions that made the framework easier to change and understand. Two such systems had to be evolved until the final version of DATsys was integrated with CourseMaster.

At first, the marking process of specific domains that use diagrams might seem to bear little resemblance across domains. Circuit diagrams are fundamentally different from software design diagrams and even more different from music transcriptions. However, the process of checking the syntactic correctness of the diagram solution and that of attributing marks can be abstracted and configured to reflect a great range of domains and exercise specific criteria. By encapsulating common marking checks into marking tools and defining the protocol of their interaction, it is possible to abstract the level of marking in such a way that new domains can be accommodated with very little effort.

Initially DATsys aimed at using the Ceilidh system for its functions of presenting information, administering and automatically assessing course modules. However, alterations essential to the marking mechanism forced the redesign of Ceilidh using object-orientation. The evolved system changed its name to CourseMaster and is described in [HHS+01], [FHH+01] and [Cm01].

1.2.4 Contributions

The primary contribution to this research is in the area of CBA. The combination of CourseMaster and DATsys considerably eases the building of automatically assessed diagram-based exercises. Prototypical coursework has been built for the areas of logic design and software design (flowcharts and object-oriented diagrams). Libraries of diagram elements have been developed for more than 50 diagram notations and are distributed together with the authoring system. The system offers facilities to educators to manage the full lifecycle of automatically assessed diagram-based coursework. CourseMaster and DATsys have replaced Ceilidh and are now in use in an increasing number of academic institutions.

The second contribution is in the field of diagramming. DATsys offers practical benefits for the diagramming community by providing a platform to ease the development of new graphical editors. It has found successful use in a number of projects, notably for its diagram editor generating facilities. These projects are described in chapter 8.

1.3 Synopsis of the Dissertation

This chapter presented the motivation, scope, and background for the research. It introduced the three main problem areas, set the general objectives, explained the approach and highlighted the contributions. The research combines theory and techniques from the fields of CBA, diagramming and software engineering. It aims for a configurable, usable and extensible design that facilitates the authoring of diagram-based CBA. Figure 1.2 illustrates a high level view of the chapters and their contents.

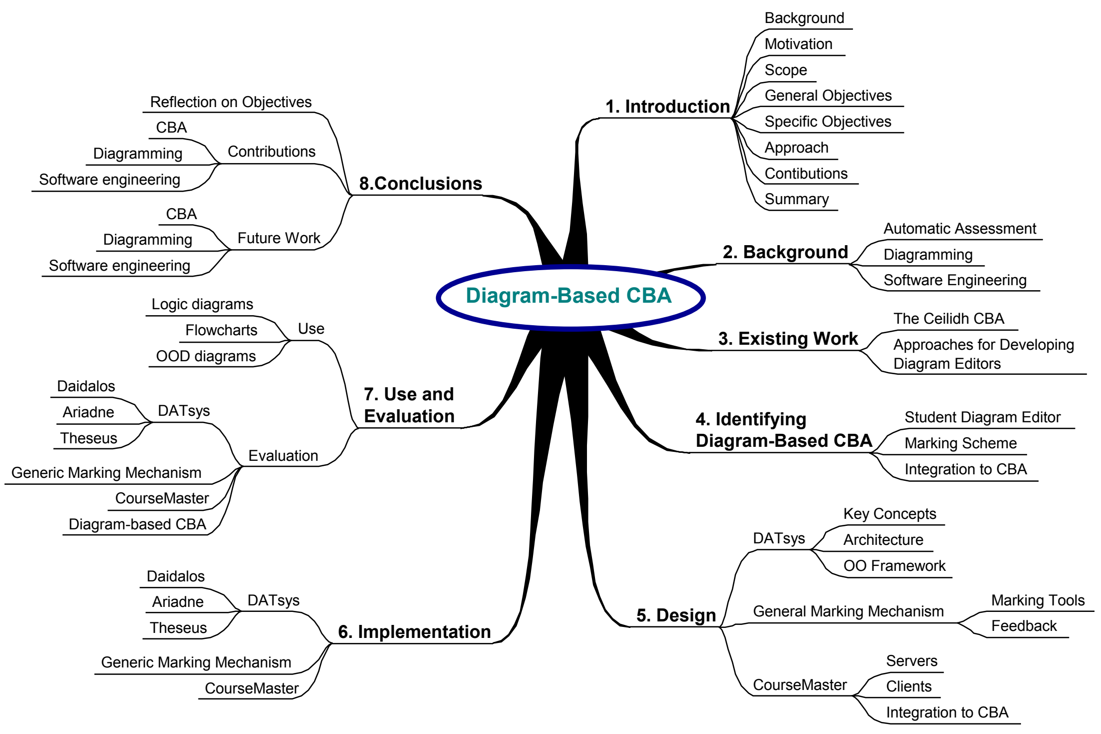


Figure 1.2: A mindmap diagram that represents a summary for this thesis

Chapter 2 introduces key concepts in the areas of CBA, diagramming and software engineering. The focus in CBA is on free form assessment for summative purposes. The focus in diagramming is on the relationship between learning and diagrams and on the use of diagrams in education. Finally, the focus on software engineering is on increasing software quality for CBA and diagram editors. This chapter highlights the main problems in each area, in order for the third chapter to present the most relevant approaches found in the literature.

Chapter 3 presents the work upon which this research is based and leads to the problem areas that are discussed in the following chapter. It documents the existing work in free response CBA and diagram editing. Firstly, it explains the techniques used in Ceilidh for free response CBA in a controlled environment and for formal marking. Secondly, it reviews and classifies systems for diagram-based editing.

Chapter 4 expounds the problem of authoring diagram-based CBA and presents the forces that need resolving. The three problem areas that have been identified are concerned with diagram editing, the marking process and the integration and support of the full lifecycle of diagram-based CBA. The problem of balancing the opposing forces of applicability and usability against simplicity and maintainability led to the early adoption of ideas belonging to object-oriented technology, application frameworks, and patterns. These are reviewed in the literature review of software engineering in chapter 2.

The main architecture and design decisions for creating DATsys, the generic marking system, and CourseMaster follow in chapter 5. The conceptual solution is shown to satisfy the objectives of each problem area. The problem of customising the diagram editor to the exercise's domain is approached by designing a framework and authoring system for developing diagram editors. The problem of developing and using assessment criteria for diagram based CBA is approached by designing a generic marking system that can be customised with little effort. Finally, the problem of integrating the two solutions into a usable CBA system, for summative assessment in a controlled environment, is approached by redesigning the existing Ceilidh system.

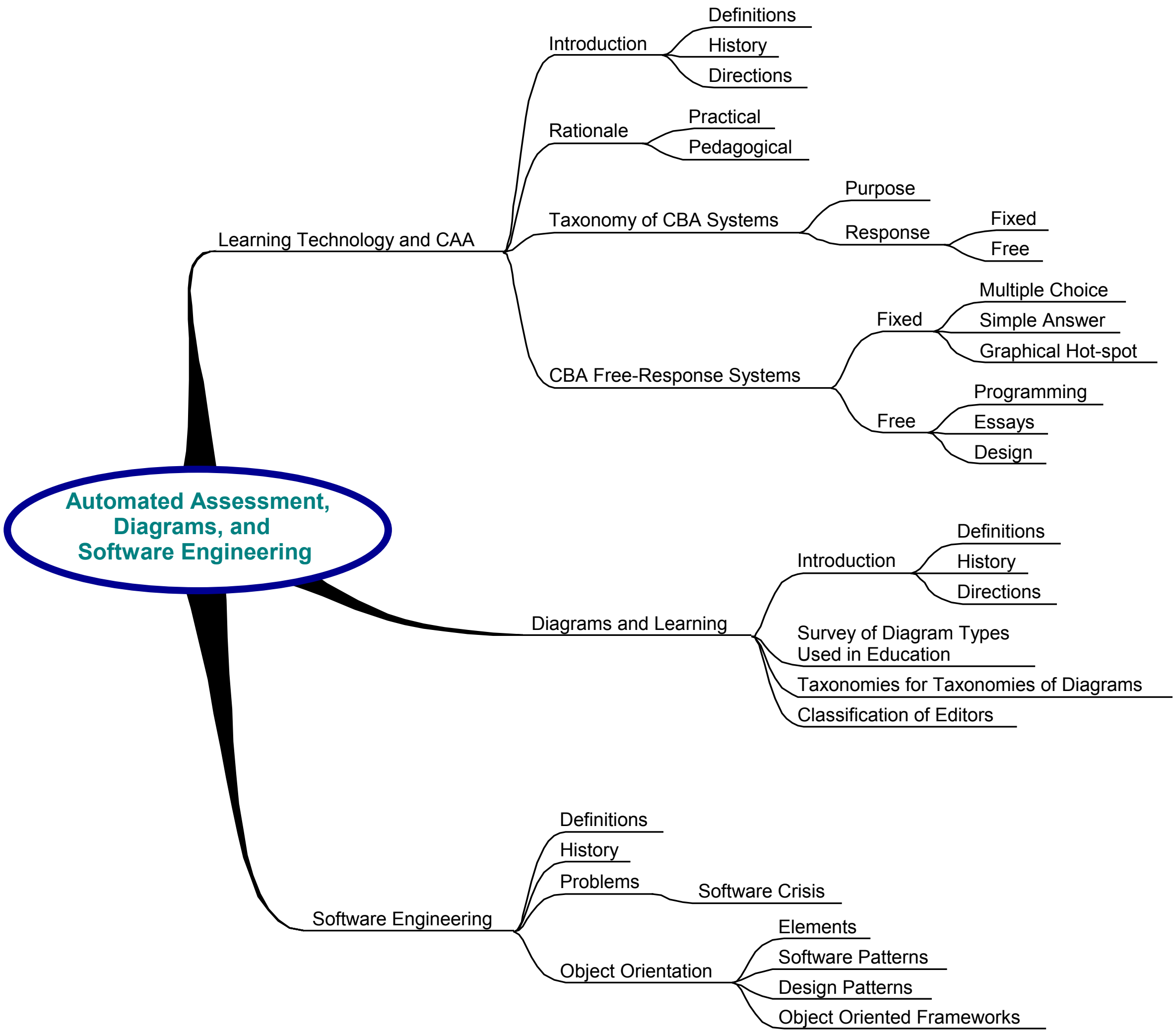
Chapter 6 reports on a prototype that implements the architectural designs described in chapter 5. It documents the parts, roles and interfaces of all the software deliverables. These include the developer's authoring environment (Daidalos), the teacher's authoring environment (Ariadne), the student's diagram-based exercise environment (Theseus), the object-oriented framework for diagram-based editing (DATsys), the generic marking system and CourseMaster.

Chapter 7 reports on the evaluation of the system. It demonstrates considerable success in each of the areas on which it has set objectives. The authoring environment facilitates the authoring of a variety of diagram-based CBA exercises in a range of domains. Automatically assessed exercises for three domains have been evaluated at the University of Nottingham. Evaluation results are also discussed for the design of DATsys, the generic marking system and for the CourseMaster CBA system. A number of evaluative perspectives substantiate the original speculation for the pedagogic and practical benefits of the deliverable software.

This thesis supports the argument that the development of an authoring environment for diagram based CBA is feasible and useful. Chapter 8 highlights these claims by reviewing the dissertation's key points and linking the results given in chapter 7 to the general objectives. It also discusses the contributions of this research to the fields of CBA and diagramming while indicating areas for future work.

Chapter 2,

Automated Assessment, Diagrams, and Software Engineering



"The map is not the territory",

Alfred Korzybsky, Science and Sanity, 1939

Introduction

This chapter presents the background of this research in the fields of CBA, diagramming and software engineering. It comprises of three sections, one for each field respectively.

The first section, starts with an overview of automatic assessment. It introduces key definitions, historical facts and briefly summarises the characteristics of assessment from a perspective of educational technology. Based on these, it examines the rationale in moving from traditional to automatic assessment. As the use of CBA systems in higher education is rapidly increasing, a range of systems supporting automatic assessment is emerging. Several publications include surveys of CBA systems in use [Bj93], [SM97], [CE98a]. The focus of this section is not to reiterate these summaries but to expose the background concepts around which diagram-based CBA can be established. Published work in diagram-based CBA is sparse. The only existing work in the field, elaborates on diagram-based assessment for software design.

The second section of this chapter introduces the concept of diagrams and examines their role in learning. It also reviews diagram-use and research directions among a number of disciplines. A categorisation of taxonomies of the study of diagrams is presented to point out the various research interests.

The third section investigates software engineering techniques for designing and implementing complex software. It introduces software engineering, and surveys the state of technology in object-orientation, software design and patterns. Although the focus concentrates on building software generally, this section leads to issues specific to the building of CBA software and diagram editors.

2.1 Learning Technology and Computer Based Assessment

Learning technology is a widely used term that refers to the application of technology to enhance the learning process. Computer Assisted Assessment (CAA) is a field of learning technology that studies the use of computers to deliver, analyse and mark student coursework. Research interests in the CAA field include the analysis of data collected via Optical Mark Recognition readers (OMR). A further specialisation of CAA is Computer Based Assessment (CBA). What differentiates CBA from CAA is that in CBA the coursework's solution is entered on-line through a computer terminal [CE98a].

The first subsection describes how the CBA community views the rationale for automating the assessment. It investigates what the education community believes constitutes the criteria for effective and quality-based assessment. These findings serve as a basis for debating the benefits, limitations and consequences of automating the assessment process. The second part surveys existing CBA systems, reporting on past and current research.

2.1.1 Automating the Assessment

2.1.1.1 Brief Historical Overview

The first attempts at using computers to automate the process of assessing student work were reported in the early 1960's [FW65]. Forsythe and Wirth presented a system for the automatic assessment of programming exercises based on Balgol, which was a dialect of Algol 58. The system allowed the students of a numerical analysis course at the University of Stanford to submit the solutions of their exercises on punched cards. The submitted work was executed, analysed and marked by a "grader" program after the coursework's deadline. Early automatic assessment systems were built almost exclusively towards computer science related subjects.

Systems supporting automatic assessment in other fields such as physics, mathematics and chemistry appeared soon after [TD76], [RH83], [Mr86]. Initial approaches dealt with the problem of marking student solutions by simplifying the types of question. Simple answer matching mechanisms were used to mark the solutions of simplified exercises.

CAA has been used in academia during the 1990's, to assess a wider variety of subject matters. In a study in the use of CAA in 1997, Stephens and Mascia presented the results of a survey carried out in institutions of higher education in the UK. From the 644 questionnaires that were sent, 445 were returned giving a response rate of 73%. According to the results, 280 respondents were using CAA, 20 were developing CAA, 10 were intending to use CAA and 132 were not involved in CAA. In a section that contemplated the future of CAA 73% of the respondents that already used CAA indicated that their use of CAA would most likely increase in the future.

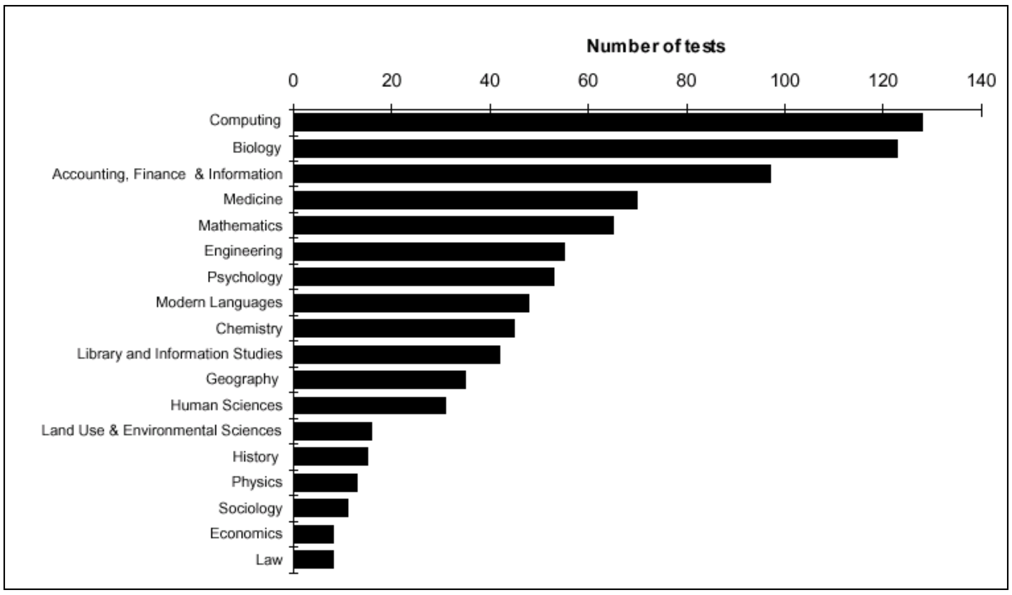


Figure 2.1: Number of CAA tests found in each subject category

Figure 2.1 depicts the number of CAA tests among subject areas [SM97]. In a subsequent survey that was carried out by McKenna and Bull, it has been reported that more than 80 universities employ some form of automated assessment [KB99b].

2.1.1.2 Motivation and Directions in CBA

Automatic assessment has the potential of becoming a subject in its own right. It concentrates the interest of educators from a wide range of disciplines who maintain the perspective of their discipline. This produces a variety of publications with interdisciplinary topics. Although themes vary, the commonly perceived benefits are *pedagogic* and *practical*. Bull summarises the opinion of the published community stating that "*computer assisted assessment can be used to enhance the student learning experience, expand assessment processes and potentially provide efficiency gains for academic and support staff*" [Bj99]. Summons et al express a pragmatic view on the need for automatic assessment: "*With large numbers of students with different learning styles, there is a need to develop instructional arrangements that maximise student learning, while trying to minimise the cost in terms of time, effort and money*" [SCB+96].

The marking process is a key concern for any CBA system because maintaining the balance between resource constraints and effective assessment in large classes can be exceedingly demanding. How much automation can be used and in which part(s) of the assessment is often the subject of debate. Canup and Shackelford support the view that although components of the marking process can be and should be automated, the actual marking has to be performed by humans [CS98]. Mason and Wait argue for hybrid approaches in which only limited parts of the marking process are automatic [MW99]. In an earlier attempt, Mason and Wait report on the experience in running final exams in an on-line fashion [MW98]. Arnow and Barshay support the idea of fully automating the assessment process and argue for the potential practical and pedagogic benefits [AB99].

2.1.1.3 A Pedagogic View of Assessment

Learning is measured through assessment. Research in educational psychology has documented various models for the study of learning. A comprehensive survey of the most important models for learning developed after the beginnings of the 19th century is given by Bransford, Brown and Cocking [BBC99]. The most cited model by the CAA published community is Bloom's Taxonomy [Bs56]. Bloom defined a classification of cognitive learning that contains ascending levels of abstraction that augment gradually from simple knowledge recall to the ability to evaluate.

Bloom's taxonomy has been used widely as a model to argue for the adoption of teaching strategies in both computer science and other sciences [BS00], [SM00], [Mp00]. The taxonomy lists knowledge, comprehension, application, analysis, synthesis, and evaluation as cognitive aspects of learning.

Cognitive aspect of learning	Ability
Knowledge	To remember
Comprehension	To understand
Application	To apply concepts to solve problems
Analysis	To break down to concepts
Synthesis	To combine concepts
Evaluation	To make judgements

Table 2.1: Cognitive aspects of learning according to Bloom

Knowledge is the foundation for all learning and the basis for all higher levels of thinking. The assessment of knowledge involves the recall of facts.

Comprehension represents the lowest level of understanding. It includes the ability to translate, interpret and extrapolate meaning while determining implications and consequences. The assessment of comprehension involves verifying assumptions and arguments and examining the process of inference.

Application is defined as the use of abstractions in order to solve a problem. The abstractions may be any information, ideas, methods, theories or skills. The assessment of application consists of evaluating the solution of a problem and analysing the process of deriving it.

Analysis emphasises the breakdown of a topic into its parts and the search for detecting relationships between the parts and the methods of their organisation. Assessing analysis involves evaluating the ability to use the parts of a topic coherently to arrive at conclusions.

Synthesis is defined as the process of combining parts or elements to form a whole. Assessing synthesis requires verifying the selection, composition of the parts, and all the steps in the synthesising process.

Evaluation is the highest level in Bloom's taxonomy. It involves reviewing evidence and making appropriate judgments. Assessing evaluation skills incorporates the detection of abilities to compare, argue and perceive fallacies in arguments.

The task of formulating an assessment strategy lies with the experience of the educator and/or that of the academic institutions. Although guidelines and helpful methodologies have been documented in literature, debates as to the efficacy of current assessment techniques to test the real abilities of learners often surface. This is especially true in fields that change rapidly where the need to reflect a change in the assessment process is even greater.

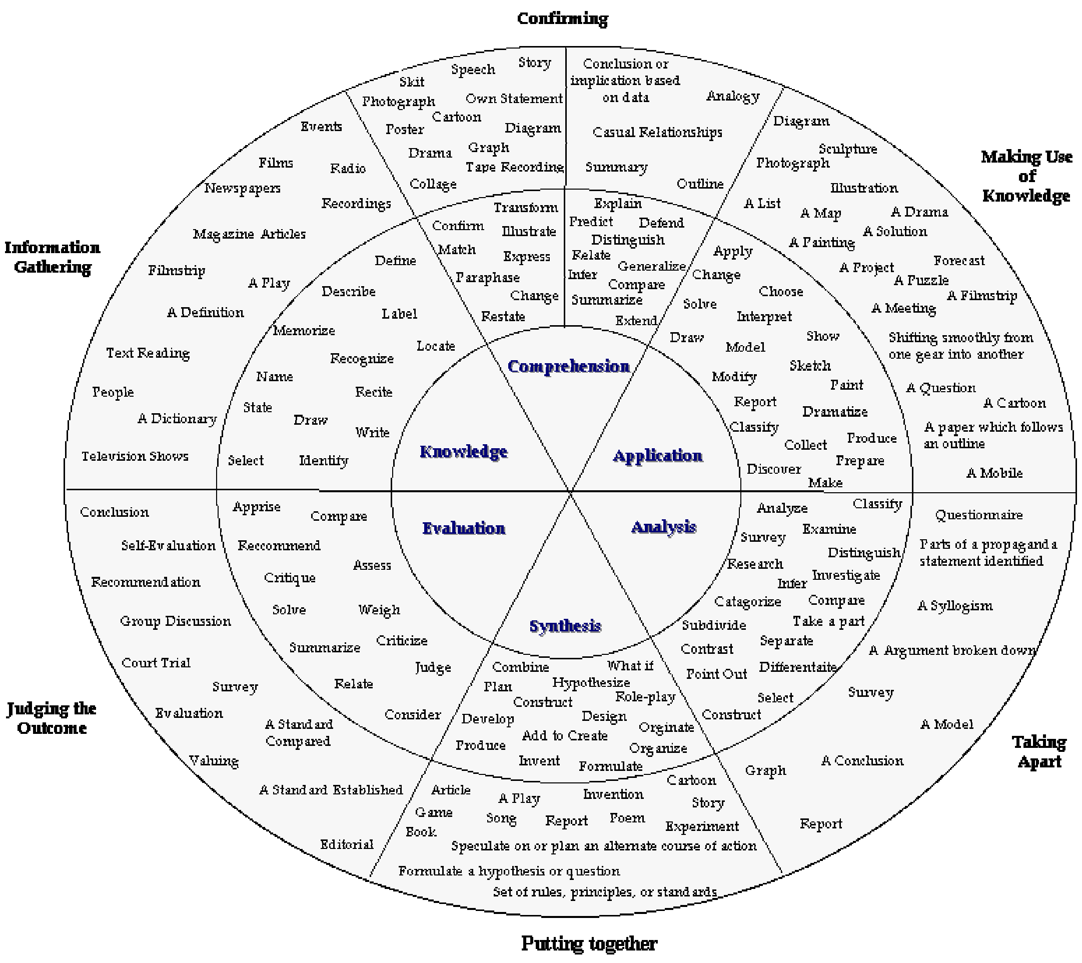


Figure 2.2: An adaptation of Bloom's wheel on the cognitive aspects of learning

Bloom's taxonomy is commonly used as a basis of designing assessment strategies. Bloom's wheel, depicted by Figure 2.2 is a tool that helps educators to identify what is important to assess in relation to learning.

2.1.1.4 Automatic Assessment: Advantages and Limitations

The automation of the assessment process for all but those tested at the knowledge level of Bloom's model is a complex task. The assessment of knowledge can be automated because it relies on a simple matching algorithm. For example, simple multiple-choice questions typically assess knowledge only. The assessment of synthesis, analysis and evaluation skills is much harder to implement and is largely domain dependent.

Presuming that assessment criteria can be modelled and automated, what effects would that have on the assessment process for formal marking in a controlled environment? Table 2.1 lists the qualities assessment should have according to Brown [BR96]. Although there may be some deviation between Brown's set of qualities and others in specific subject matters, it represents a common denominator.

Assessment must be:	In order to:
Valid	Accurately assess the delivered material
Reliable	Promote consistency between assessors
Fair	Offer fair opportunity for success
Equitable	Be indiscriminating between students
Formative	Give many opportunities to learn through feedback
Well timed	Provide learning stimulus and be fair
Incremental	Increase reliability and consistency over a period of time
Redeemable	Allow a series of opportunities
Demanding	Challenge students and ensure high standards
Efficient	Be manageable within the constraints of resources

Table 2.1: Summary of qualities of assessment according to Brown et al

The validity of assessment is proportional to the experience of the assessment designer. Reliability and fairness increase by automating the assessment process because the same marking mechanism is employed to mark each piece of work. There is no possibility of discrimination and students are well aware of the fact that everyone is treated equally by the system. Formativeness is related to the feedback given to students. The timely fashion which CBA software imposes, forces both students and educators to respect deadlines.

Automating assessment facilitates an incremental style of assessment. Benford et al suggest that weekly assessments of programming courses increase the consistency between the learning of the material and student results [BBF93]. Small and frequent chunks of coursework rather than a major piece of work deliver better results in subjects such as programming. By allowing students to submit their solutions more than once and by presenting them with immediate feedback upon submission, CBA coursework motivates students to work harder and get a better mark. During the lifetime of the course, educators can monitor the progress of the students and fine-tune the delivery of both the theoretical and the practical aspects of the taught material.

However, as Charman and Elmes underline, CBA can't be applied to assessment for all types of learning [CE98a]. There is a range of skills that CBA cannot address. Specifically, oral, presentation and interpersonal skills cannot be practically assessed yet using current technologies. Furthermore, the assessment of complex writing skills is still very limited, although continuous research is progressing towards better results. Finally, practical skills acquired in laboratory-based sessions demand much more complex CBA techniques.

A misunderstanding of the amount of effort that is needed to employ CBA appropriately is a considerable limiting factor to the use of CBA in higher education. Authoring CBA exercises is more time consuming than authoring distant learning material. As Doube reports [Dw00], the authoring of learning material should not be perceived as an additional responsibility for educators but as a separate task that does not interfere with their workload. Authoring of teaching and assessment material is more time-consuming than traditional on-campus teaching.

Neuman speculates that "*electronic education*" that includes CAA may be better for disciplined students who "expect more than just being entertained" and is more useful for teaching fundamentals [Np98]. This is because "*such courses are highly susceptible to cheating, which can be expected to occur rampantly whenever grades are the primary goal, used as a primary determinant for jobs and promotions*". Citing Brynjolfsson and Hitt, Neuman suggests that a new approach in education is required and might initiate a "*painful and time consuming period of reengineering, restructuring and organisational redesign*" [BH98].

2.1.1.5 Summary

The motivation in automating the assessment process in higher education is to alleviate the practical problems introduced by large classes and to harness potential pedagogic benefits. Both aims are worth pursuing and can be combined. Bloom's taxonomy on the cognitive aspects of learning is a useful model for understanding assessment in general. Not all of Bloom's cognitive levels can be easily assessed, if at all, using automatic assessment. However, advances in the automation of the assessment of essays and of programming exercises suggest that in some cases all Bloom's levels can be assessed automatically.

2.1.2 A Taxonomy for CBA

Culwin distinguishes between two types of automated assessment, *fixed response* and *free response* [Cf98]. Figure 2.3 depicts a taxonomy of CBA systems based on Culwin's view. In the literature, fixed response assessment is also referred to as *objective* and free response as *non-objective*. The distinction between the two is based on the nature of response to the CBA exercise. Fixed response systems require the learner to choose a solution from a list of available options. The assessment is based on a strictly discrete model of validating correct answers against a single solution. Charman and Elmes [CE98a] advocate that the fixed response type of assessment is sufficient to cover the main aspects of evaluating student learning. However, for disciplines emphasising design skills, it is much harder to devise effective automated assessment based on fixed response. This is because a single value answer is not adequate to assess the thought processes that are required to solve the design problem.

In contrast, free response systems can assess unanticipated solutions. Examples of such solutions are the implementation or design of computer programs, essays, diagrams, sketches and drawings. Typically, in free response assessment a qualitative strategy describes the criteria attributing to the evaluation of the exercise.

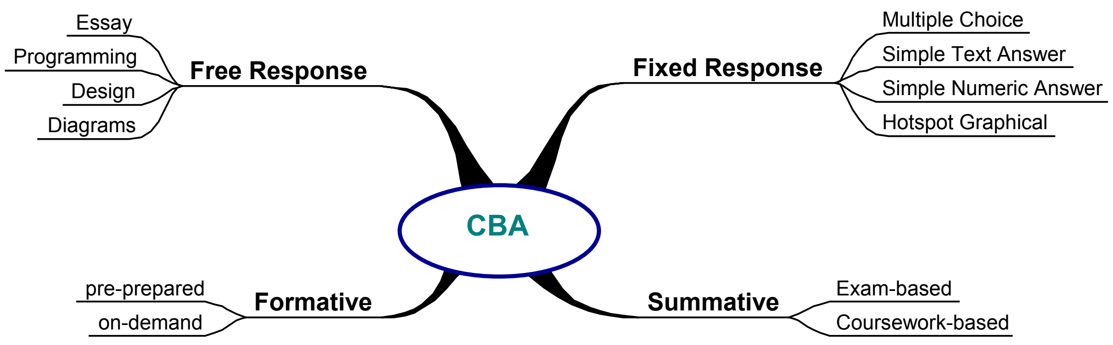


Figure 2.3: Diagram for categorising CBA systems

As Culwin explains, "*The majority of existing systems are fixed response and most of them are multiple choice*" [Cf98]. He continues stating that this is not surprising because "*the development of free response assessment is much harder or even impossible*".

The next division in CBA software is based on the purpose of its applicability. CBA can be used for *formative, summative* and *diagnostic* purposes. In formative assessment, the aim is to enhance learning by providing helpful feedback to the learner, whereas in summative assessment the aim is to discriminate between students' learning ability. Formative assessment is further categorised by the strategy of assigning coursework: either on-demand or pre-prepared.

Formative assessment enhances the learning process by providing immediate and periodic feedback. Student feedback is given to encourage self-assessment. It is monitored by educators to adjust the delivery of the material. The material for assessment can be pre-prepared or selected from an exercise database according to criteria based on the learner's profile, personal history or other relevant information. Formative assessment is not part of a curriculum. It therefore has limited use within an academic environment and typically is not completed by students. The purpose of diagnostic assessment is to identify whether a student has sufficient understanding of the prerequisite material needed to start a module.

Summative assessment concentrates on measuring and quantifying the learner's performance, attempting to formulate an accurate judgement about a student's achievement. It can be employed to offer quality assurance checks, both within an institution and externally, with evidence, results, and justifications.

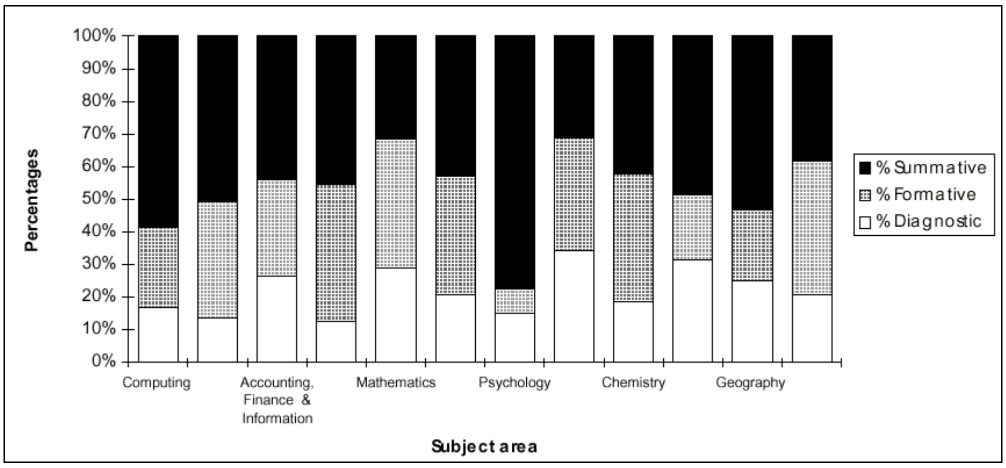


Figure 2.4: Use of different types of assessment in the most popular areas of CAA

In the study by Stephens and Mascia of the use of CAA in institutions of higher education in the UK, it has been found that in most fields CAA is used mostly for summative purposes. Figure 2.4 depicts the percentages of CAA use among fields.

2.1.3 Fixed Response Automatic Assessment

The majority of academic institutions that employ CBA technology, either for formative or summative assessment, use techniques based on fixed response assessment. CBA systems that support fixed response assessment use either:

- Multiple-Choice Questions (MCQs)
- Simple text or numeric value exercises
- Graphical hotspot exercises

The description of the marking process in fixed response CBA is simple. Only a minimum of information needs to be described. Typically, fixed response based CBA is employed by domain experts that do not have, nor wish to have, the knowledge required to describe complex marking schemes.

McKenna and Bull, in an investigation of the potential and the limitations of fixed response assessment, described techniques for improving assessment by integrating MCQs with other types of assessment [KB99]. According to their view, fixed response assessment is commonly assumed useful for assessing the first three or four levels of Bloom's taxonomy. However, they cite the view of other educationalists such as Simas and McBeath that suggest that by designing appropriate questions, all six levels can be sufficiently tested [SM92].

2.1.3.1 Multiple Choice Questions

Multiple-choice questions represent the simplest form of automated assessment. Their structure consists of a main body composed of question statements and a set of options. Typically, one of these options is correct and the others serve as a decoy with a varying degree of incorrectness. Variations of MCQs include those requiring a simple true/false answer, those requiring a priority order and those that need a multiple selection of options. Table 2.2 lists six of the commonly used types. MCQs have been successfully used to assess a wide range of learning in various fields [Bj93]. Literature that proposes methodologies in creating educationally appropriate MCQs abounds. The implementation of the marking process is algorithmically simple even in its most complex form, for example that of multi-response MCQs. Examples suggesting guidelines for designing the questions have been reported for many fields. There is some concern about "guessing" when MCQs are used for summative purposes. However, some techniques have been used to decrease this risk. Among these, subtracting marks for incorrect answers and increasing the amount of assessment appear to be efficacious deterrents.

Question Type	Task
True/False	Choose right or wrong
Item order	Match items in a list
Multiple Choice	Select correct answer from a list of alternatives
Multiple Completion	Select correct combination from a list of combinations
Assertion/Reason	Choose the correct reason for an assertion
Best Answer	Choose best answer from a list of correct answers

Table 2.2: Six examples of common types of multiple choices

Currently, a number of commercial and non-commercial systems offer authoring facilities for MCQ-based assessment. QuestionMark and QuizIt are between the most widely known systems. QuestionMark presents the educator with a graphical authoring environment that facilitates the development of several types of MCQ CBA: Matching, Matrix, True/False, Multiple Response, Pull-Down List and Ranking. QuestionMark has modules for administering coursework, exams and surveys and is multi-platform [Qm01]. QuizIt's approach to the authoring of material has been to create a language named QBL, on which to describe the CBA exercise. QBL has been designed as a subset of SGML [Gc92]. According to the QuizIt developers, Tinoco et al, the system contains an authoring module and a marking module. It can adapt to student abilities by branching, using conditional checks on their performance. After evaluating QuizIt, the authors reported an increase in efficiency and effectiveness of assessment [TBF97].

MCQs do not have to be purely textual. To make the assessment process more interesting to students, graphics and multimedia are often employed. WebMCQ is a system that uses graphics, and according to its authors, this makes the automatic assessment of multiple-choice exercises much more attractive to students [DG99].

2.1.3.2 Simple Text or Numeric Answer Questions

Exercises that require the completion of a sentence, of a fact or a statement in textual or numeric format represent the second category of fixed response assessment. The student response can be marked in a simple manner using an algorithm that compares the answer against the model solution. In a more advanced form, the comparison has to take into consideration all cases where there is a range of acceptable solutions. For example, in a question requiring text and asking for the name of the Greek philosopher who taught Plato, the answer could be either "Socrates" or "socrates". The use of regular expressions to realise such comparison is advantageous because the two above answers can be evaluated with the single regular expression `[S|s]ocrates`. Questions with numeric answers often need to take account of rounding and precision errors. Wrong answers might also have to be defined to check the degree of student's understanding for part of the concepts involved or for indicating the thought process that led to the answer.

Systems that facilitate the authoring and running of the simple text or numeric type of assessment include Ceilidh [BBF+93], TRIADS [Md99], and Examine [EX95].

2.1.3.3 Hotspot Graphical Questions

Hotspot graphical exercises require the student to select an area or item, drag and drop a component over an area or join/connect two or more items or areas. This type of assessment is advantageous in courses where the graphical element is very important. The marking scheme authored by the question developer must have a notion of the graphical elements. Most of the CBA systems that offer this type of exercise have been written in multimedia authoring packages such as Macromedia's Authorware [Mac95] and Assymetrix's Toolbox [Ass94]. An advantage of using multimedia is that young students, who have a limited attention span, are more attracted to the exercises. Students may experience exercises as games and it is a common argument that learning improves while in a playful state of mind.

TRIADS is a system that supplies functionality for the graphical hotspot type of exercise [Md99]. Most of the other systems are being developed in multimedia authoring packages.

2.1.4 Free Response Automatic Assessment

Free response automatic assessment is characterised by the free form response of students to the exercise and the criteria-based algorithm that marks the solution. Free response assessment is suitable when the higher levels of Bloom's taxonomy need to be assessed, specifically: the application of knowledge, analysis, synthesis and evaluation. CBA systems that support free response assessment are almost exclusively built for specific subjects such as:

- Programming
- Technical essay writing
- Diagrams and graphics

The ordering of the above categories represents the amount of work that has been invested in each category. Existing free response CBA systems are mostly used in the assessment of programming whereas the automation of assessment of essays, diagrams and graphics are still in their early stages.

2.1.4.1 Programming

The Ceilidh courseware system was one of the first CBA systems to provide functionality for the full lifecycle of programming courses. Although, as mentioned in the historical review of CAA in section 2.1.1, the first marking programs were implemented in 1965, Ceilidh was the first system to cater for the authoring of CBA coursework, the administration and management of modules and the presentation of information to the students. From its conception in 1988, Ceilidh had an important impact on the research and implementation of related CBA systems.

Demonstrating that the automatic assessment of computer programming exercises is feasible and effective [BBF+93], [BBF+95], the Ceilidh system's success can be largely attributed to the provision of an assessing mechanism for creating and running "oracles" [FZ93]. Oracles are checkers that run against a students' solution to find whether specific criteria are met. Oracles have been developed and used to aid the assessment of exercises in imperative, functional and object-oriented programming, music harmonisation and technical essay writing [FHT+99], [FSZ97], [MGC98],

[FL94]. Although generally oracles are restricted to the domain from which the exercise belongs, some are generic enough to be used across domains. An overview of the Ceilidh system and of the main oracles is presented in the next chapter.

The need for automating the assessment in programming has been perhaps greater than in any other field, primarily because programming courses have attracted an increasing number of students. Kay et al express the opinions of many educators in this field stating that: "*The time has come to devote effort and resources to developing robust, flexible, widely available tools for automatic program evaluation. We can no longer pretend to be able to assess student's work validly and reliably by manual means*" [KSI94].

Jackson and Usher describe the ASSYST system in [Jd00]. ASSYST is a hybrid CBA system in which some of the marking processes are automated and some are performed by human markers. Jackson and Usher claim that a hybrid approach has advantages over fully automated assessment. By allowing fine-grained selection between automatic and manual assessment a teacher can benefit from the marking consistency and speed while maintaining full control over student results. The assessment techniques used in ASSYSTS have been influenced by the Ceilidh system and this is acknowledged by Jackson and Usher [JU97].

Daly presented RoboProof, an on-line teaching web-based system [Dc99]. RoboProof has been used to teach the syntax and structure of programming languages. Daly after experimenting on a C++ course, reported results that demonstrate an improvement in learning and assessment. The exercise marker is used only for output checking, however, there is an option to customise the marking process. RoboProof is used in a less strict fashion than other programming based CBA systems. It allows any number of submissions and does not penalise for failures. Daly currently investigates making the marking system understand particular errors and to guide students to appropriate learning material. Again, the Ceilidh project has been a major influence and this is referenced by Daly's work.

Joy and Luck presented a CBA system entitled BOSS that includes facilities for both automatic and manual marking. In Boss a graphical user interface has been devised to ease the development of the marking scheme under the name "Electronic MarkSheets". Evaluating the Ceilidh system Joy and Luck state: "*One type of package in particular deserves some discussion because of their size and distinct approach. Systems such as Ceilidh* *are packages which provide a full programming environment, handling not only submission and testing of assignments, but also providing tutorial material and a user friendly interface to the machine*". Having evaluated Ceilidh, Joy and Luck concluded that a major problem in Ceilidh is not supporting customisation of its functionality [JL98].

Korhonen and Malmi have described TRAKLA, a system to aid the learning of data structures and algorithms. The system employs visualisation, animation and simulation for presenting concepts to the student. It also contains automatic assessment tools for formative evaluation. The architecture of TRAKLA's assessment component is similar to Ceilidh and this is acknowledged by Korhonen and Malmi [KM00].

2.1.4.2 Essay Exercises

The marking of essay exercises has attracted significant interest among CAA researchers for the last forty years. Evaluation results have been reported but only for prototypical laboratory testing. A limited number of CAA case-views refer to examples of automated essay marking in real educational environments. Current surveys reviewing the area include that of William's [Wr01], Christie [Cj99], and Whittington and Hunt [WH99].

In the earliest documented approach, the Project Essay Grade (PEG) system, a simple style analysis was used to identify the strengths and weaknesses of a solution. Page, making a distinction between marking for content and style, suggested that the surface features of an essay could be used to predict the mark that a human examiner would assign to the essay. Surface features included attributes such as the number of words, the average sentence length, the amount of punctuation and many other syntactical characteristics [Be94]. The PEG system primarily relied on linguistic features of the model solution to return a mark.

A second approach based on the Latent Semantic Analysis model (LSA) focuses on analysing the textual content to understand the deep structure of each statement. It ignores the linguistic and structural features of essays and instead attempts to find the significance of each individual word applying a matrix algebra technique known as Singular Value Decomposition (SVD). Foxley and Lou produced a prototypical essay-marking component for Ceilidh using a variant of LSA [FL94].

A third approach developed by Educational Testing Service (ETS) uses a hybrid model that combines analysis for linguistic features using Natural Language Processing techniques (NLP), together with statistical structure analysis. The criteria used in marking within ETS include syntactic variety, discourse analysis and topical content comparison [Edu02].

2.1.4.3 Diagrams and Graphics

Research that refers exclusively to the automation of assessment of student diagrams is very limited. However, in subjects such as software engineering the use of visual tools to promote learning has often been the subject of research.

Hirmanpour highlights the need for using diagrams in the teaching of software design [Hi88]. The tool used in Hirmanpour's work is a diagram editor that supports three types of diagrams: dataflow diagrams, entity relationship diagrams and structured charts.

Power gives an overview of a development environment called Designer that helps students understand concepts of structured program design [Pc99]. Designer allows the user to create and edit structure diagrams. On completion, Designer can translate the diagrams and generate programs. Composed by process boxes, decisions, loops, I/O boxes, and procedure calls, structure diagrams can be translated into control structures. The system can be used to run programs or perform code walkthroughs while allowing user interaction.

Hoggarth and Lockyer report on integrating Computer Aided Software Engineering (CASE) tools with Computer Aided Learning (CAL) [HL98]. Their work involved:

- Attaching the diagrammatic facilities of a CASE tool to a CAL product
- Embedding CAL into a CASE tool
- Implementing a diagram comparison system

The diagram comparison system does not compare visible similarities but the internal processing of two diagrams. It compares the internal processing, processing order and the connections between processes. Before submitting the solution diagram, students have to perform a symbol mapping between the components of their diagram and those of the model solution. Then the diagram comparison system finds the differences between the two diagrams giving guidance and feedback to the student. Hoggarth and Lockyer acknowledge the difficulty in building CAL systems that include diagram entry facilities stating that: "*By providing limited diagram presentation facilities within a CAL system the student can view and appreciate the design and development of system analysis and design diagrams. This is a difficulty for stand-alone CAL systems because they need to have all the information that is to be presented previously defined, as they cannot recognise the notation being used. This inflexibility requires a great deal of development work to be done to create beneficial courseware material*".

2.1.5 Summary

The first segment of this chapter introduced CAA and reviewed its relationship with educational technology. Automated assessment is not only a solution to problems imposed by increasing class numbers but also a pedagogically enhancing technology. CBA is CAA with the added characteristic that the coursework's solution is entered online. CBA can be used for formative, summative or diagnostic purposes. It can be divided into fixed and free response depending on the type of student response. Exercises based on fixed response are authored without too much effort but are limited in respect to the types of coursework that can be assessed. MCQs, simple text or numeric questions and hotspot graphical questions belong to the fixed response type of assessment. The assessment of programming, essays and diagrams belongs to the free response type of assessment. Although research has been reported for CBA for programming and essays, the subject of diagram-based CBA, in domains other than software engineering, has not yet been documented. The next segment introduces diagrams and examines their relationship to education and learning.

2.2 Diagrams and Learning

Combining CBA with exercises that have diagram-based solutions leads to the field of diagramming. This subsection presents definitions, historical background and highlights of the interests of the academic community in relation to diagrams. It also makes a summary of the most common diagrams and reports the view of the education community on the relationship between diagrams and learning.

2.2.1 Diagrams

James Maxwell's definition of a diagram is: *"a figure drawn in such a manner that the geometrical relations between the parts of the figure illustrate relations between other objects"* [eb11]. Maxwell's definition is general enough to subsume detailed definitions given by various fields. For example, the rigorous definition of a diagram in mathematics [Tk00] is radically different from the respective definition in architecture [EG97]. The word "*diagram*" in origin is ancient Greek and is a composite word from the prefix "dia" meaning "to", and "gram" meaning "line". Thus, the literal meaning is *to express using lines*.

Dodson, acknowledging that there is no crisp definition on what is a diagram suggests an informal generalisation: "*Something is a diagram if it has perceptually distinct localised parts, such as nodes and lines and uniform conventions of meaning apply to combinations of these elements*" [Dd99]. Dodson explains further that such elements can be:

- Nodes linked by lines
- Nodes in or overlapping other nodes
- Combinations of lines
- Labelling of elements by superimposed, adjacent or linked symbolic elements
- Items of the same colour, pattern, texture, and so on

According to Blackbell, diagrams are based in resemblance, metaphor and concrete descriptions [Ba98]. Resemblance is the most naive view of diagrams in that they simply look like the things they represent. Land maps, for example, belong to this category because they are made using elements that have a direct geometric association to the objects they describe. Metaphor is used to depict things that do not exist physically, yet they exist in the imaginary world. Abstract concepts and structure representations are examples of diagrams belonging to this category. Diagrams exist also as concrete descriptions of abstractions of physical entities and express helpful views on objects, relationships and processes of the world.

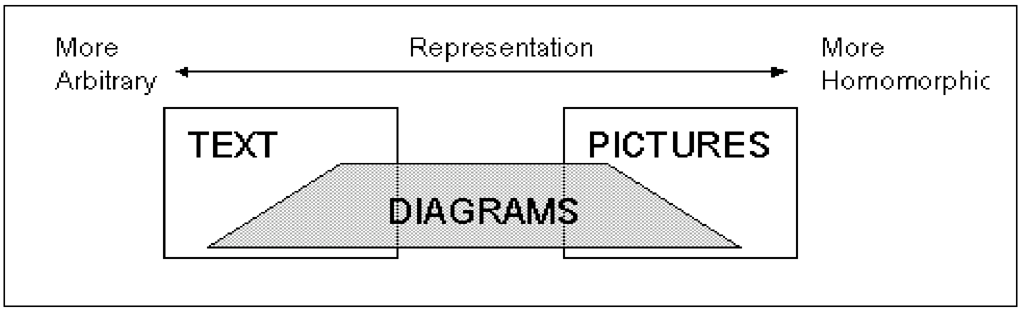


Figure 2.5: Diagrams are between the worlds of text and pictures.

Diagrams communicate information that spans between text and pictures. Figure 2.5 illustrates this concept in a diagram that uses a horizontal axis to represent the arbitrariness of diagrammatic representation. At the edges of the axis, a diagram can be described as text or picture. For example, a photograph can be considered as a diagram with a high degree of homomorphism between the representation and its subject. Text can also be considered as a diagram that contains symbols positioned and composed in a specific manner. To be widely comprehensible, most known types of diagrams obey certain rules of notation.

2.2.2 A Short History of Diagrams

Diagrams have been used throughout history to represent topological maps, geometrical concepts, religious and philosophical ideas, engineering plans and scientific abstractions amongst many other things.

The earliest recorded diagrammatic representations were land maps. Maps maintain a homomorphic relationship between the diagram and the physical space. Figure 2.6 shows a clay-tablet representing a map from the area of Ga-Sur in Mesopotamia, which is 200 miles north of the site of Babylon. The tablet is dated 2500 B.C. [Bl85].

Diagrams depicting abstractions have also been used from ancient times. Geometric diagrams have been found on Babylonian clay tablets that date from around 1700B.C. [Mn01]. Figure 2.7 depicts a tablet that shows a procedure for calculating the diagonal of a square with length of 30 units and a version of the visual proof for the Pythagorean theorem.

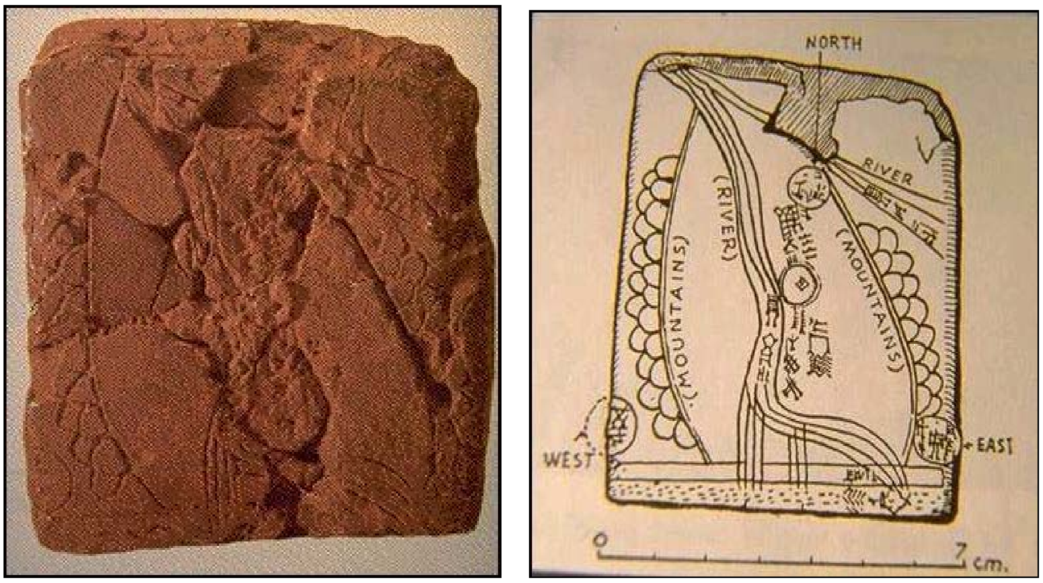


Figure 2.6: One of the earliest diagrams dating from 2500BC and its interpretation

Other occurrences of diagrams dating to ancient times have also been found in Egypt, China, India and Greece. In Greece, the work of Pythagoras, Euclid and Archimedes relied heavily on diagrams.

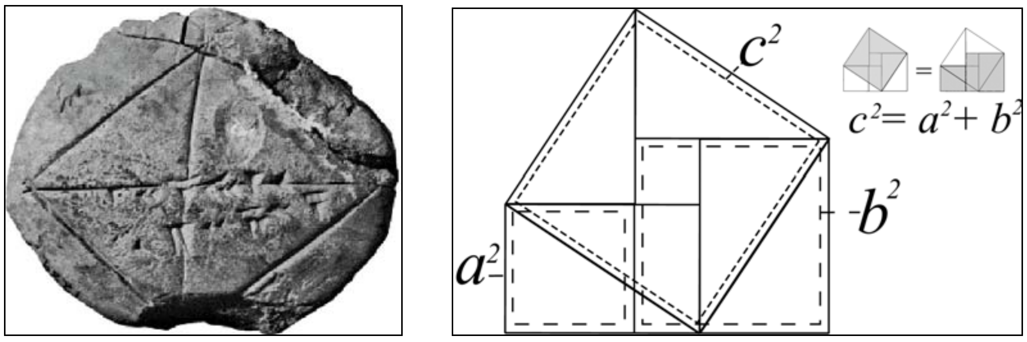


Figure 2.7: Babylonian tablet (1700B.C.) and the Pythagorean theorem

In the middle ages, diagrams have been used to represent philosophical concepts, religious ideas and lineage hierarchies. Figure 2.8 depicts two diagrams from the earliest books in print. The first diagram represents the four elements, and has been published in the "De responsione mundi et de astrorum ordinatione" [Hi72]. The second diagram entitled "tree of life" is a quite complex topological diagram that appeared in print in Kircher's work in 1652. It represented religious ideas based on the old testament [Ka52].

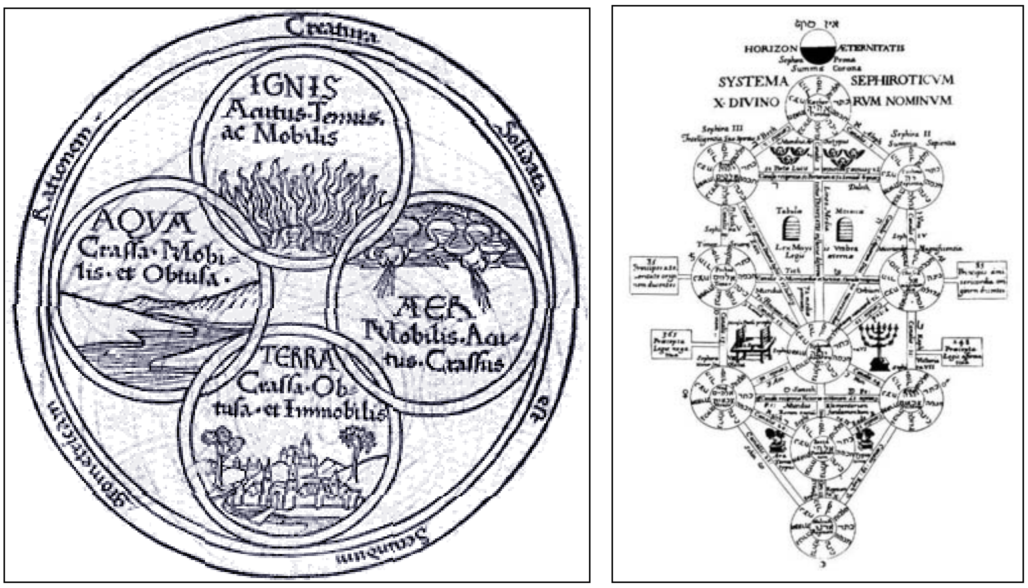


Figure 2.8: A diagram for the four elements (1482AD) and the tree of life (1652AD)

In the 17th century, the Cartesian coordinate system was conceptualised by Descartes. Descartes and Fermat became the fathers of analytic geometry that uses algebra to solve geometrical problems. This was a major step in a perception shift from geometry to arithmetic. Calculus profoundly aided scientific discoveries in physics and chemistry. This led to an increasing number of abstract diagrams recorded throughout the 18th century. Euler used circles to solve reasoning problems involving relationships between sets [El61]. Euler circles used enclosure, exclusion and partial overlap to represent the set related notions of containment, disjointness and intersection. Venn modified Euler circles by adding to the notation all the possible relationships between sets and by introducing shading for empty sets [Vj80]. Peirce proposed extensions to Venn diagrams and created existential graphs [Pc33].

In the 20th century, the number of types of diagrams has increased rapidly. With the advancement of computers, graphical user interfaces, CAD/CAM systems and diagram editors, diagrams are used in almost every known field. Research into diagrams as a mean of communication has also increased. Friendly and Denis documented an illustrated chronology of innovations in the history of thematic cartography, statistical graphics and data visualisation in [FD01]. The next section describes the interest of the disciplines that have strong links to research in diagramming.

2.2.3 Research and Use of Diagrams Across Disciplines

The topics of research presented to conferences for diagrams relate to numerous and broad areas of science, humanities and art. For example, in applied psychology, the interest in diagrams is in analysing how they relate to cognition. In cognitive science, studies are being made in models of diagrammatic reasoning and understanding. In education, diagrams are studied as tools that improve learning. History and philosophy of science examine past discoveries in relation to diagrams and how these influenced scientific thought.

Diagrams are widely used in computer science for software specification, for solving computational problems and for visualisation. The flowchart diagram was conceptualised by Von Neuman, who created a simple flowchart notation to visualise algorithms [GV47]. From the early days of computing, the flowchart became the de facto standard in illustrating algorithms [Rr63]. Among the hundreds of known diagram notations are data-flow diagrams (DFD's) [GS79], database schemas [BLN86], entity relationship diagrams (ERD's) [Cp76], [JN83], structure diagrams and process diagrams [MM85], Nassi-Schneiderman diagrams (NSD's) [SN73], pert charts [MP70], and object diagrams [RBP+91]. For solving theoretical problems well known diagrams include statecharts [Hd88], petri nets [Pc65], and state transition diagrams [BGK+96]. Diagrams to visualise data include the bar and pie chart, the scatter plot, and many of the other commonly used representations. Many types of diagrams exist to represent graphs and tree structures. Sixty graphical representations are illustrated in the work of Lohse et al [LBW+94].

In computer science, studies are made on software and algorithms for editing diagrams. A significant volume of work has been carried out in areas such as automatic layout and visualisation [HMM00], diagrammatic reasoning [Kz94] and software development for diagram editors.

Engineering disciplines use diagrams extensively. In electrical engineering, circuit diagrams for analogue and digital components are a major form of communicating design ideas. In mechanical engineering, the technical drawing describes in detail blueprints for manufacturing artefacts. The bond diagram in chemistry is commonly used to aid the understanding of the structure of matter.

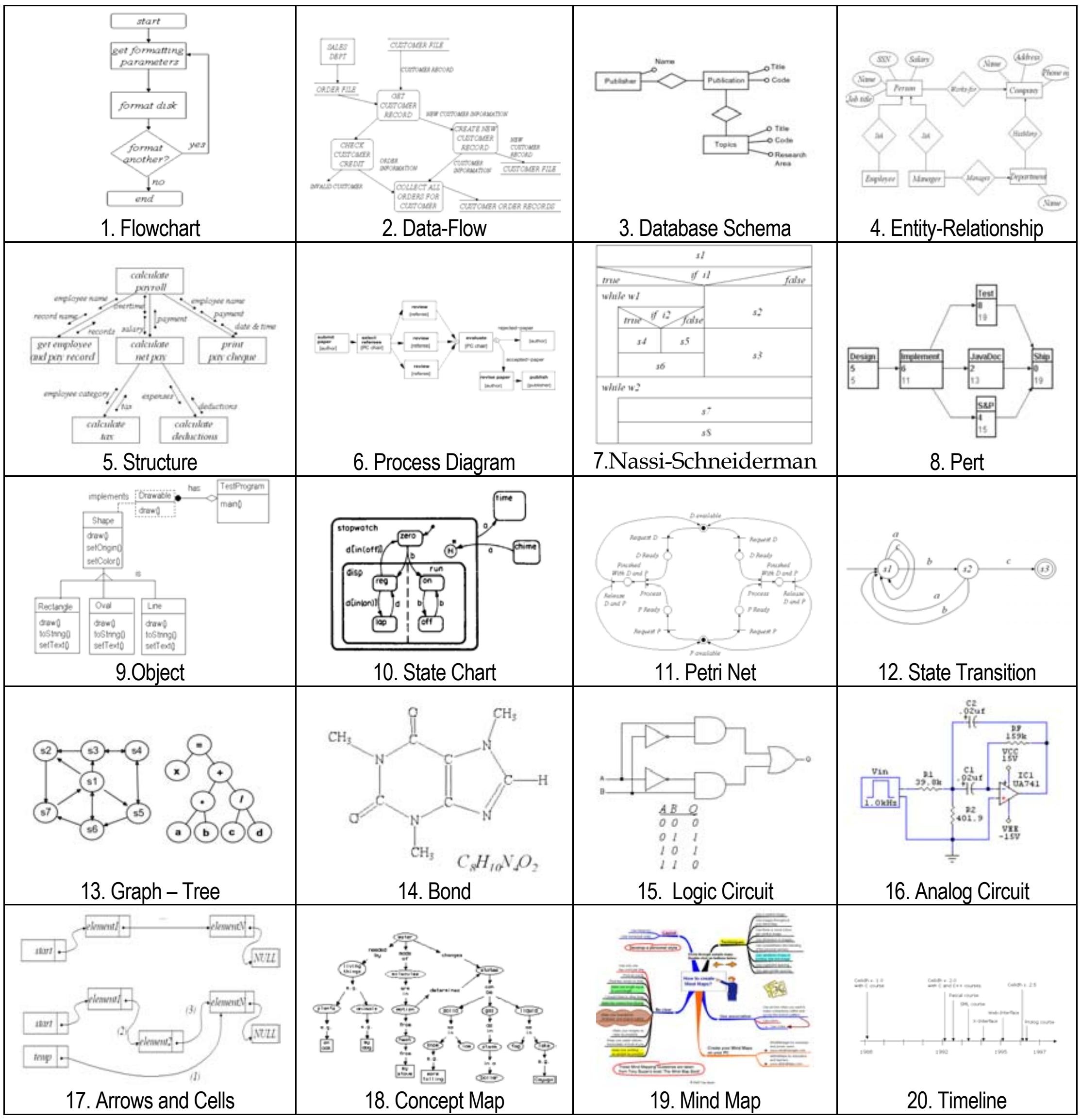


Figure 2.9: Examples of twenty types of commonly used diagrams

Hundreds of technical diagram notations exist for engineering. Descriptions for the most popular notations can be found in databases of standards such as ISO [ISO00], and ANSI [ANSI00].

Diagrams with less formal notations are commonly used to describe concepts and their relationships, hierarchies, time events etc. Concept maps [GS95], mindmaps [Tb93], and timeline diagrams are examples of types of such diagrams. Figure 2.9, illustrates examples of twenty of the most commonly used types of diagrams.

2.2.4 Diagram Taxonomies

The available studies that categorise diagrams are both domain dependent and diverse. Examples of such studies are given by Martin and McClure [MM85], and Wieringa [Wr98] on diagrams for software engineering, Burnett, Goldberg, and Lewis [BGL+95] and Burnett and McIntyre [BM95] on visual programming languages, and Richards [Rj84] on illustrational diagrams for graphic design. Further examples of diagram taxonomies related to education are given by Goldsmith [Ge84] and related to psychology by Narayanan [Nh98].

Blackwell and Engelhardt, after surveying a variety of perspectives and criteria in diagram taxonomies, have proposed a set of *taxonomic dimensions* that can be used as the basis for new taxonomies of diagrams and as a reference point for comparisons among the interests of the academic community in diagrams [BE98], [BE01]. Their taxonomy of taxonomies has six taxonomic dimensions: the representation, message, relation between representation and message, task and process, context and convention, and mental representation. Each one of the taxonomic dimensions represents a category of interests in research related to diagrams. Figure 2.10 depicts a mindmap of Blackwell's and Engelhardt's taxonomic dimensions for the study of diagrams.

The representation of diagrams is a subject that has been studied profoundly in graphic arts and design. From this perspective, a diagram is a composition of visual elements such as points, lines, text, and of various other shapes put together in an organised manner. The graphical domain defines the graphical vocabulary and the graphical structure defines the way relationships are illustrated.

The message that underlies a diagram has informational structure that can be described by its relational properties and is constrained by ontological categories such as space, time, and quantity.

The relation between representation and message concerns the association of the graphical structure to the message. This association can be seen as a correspondence of the elements of the message to those of the graphical structure. Diagrams in which the elements of the message correspond directly to the graphical structure of the diagram have a higher degree of pictorial correspondence.

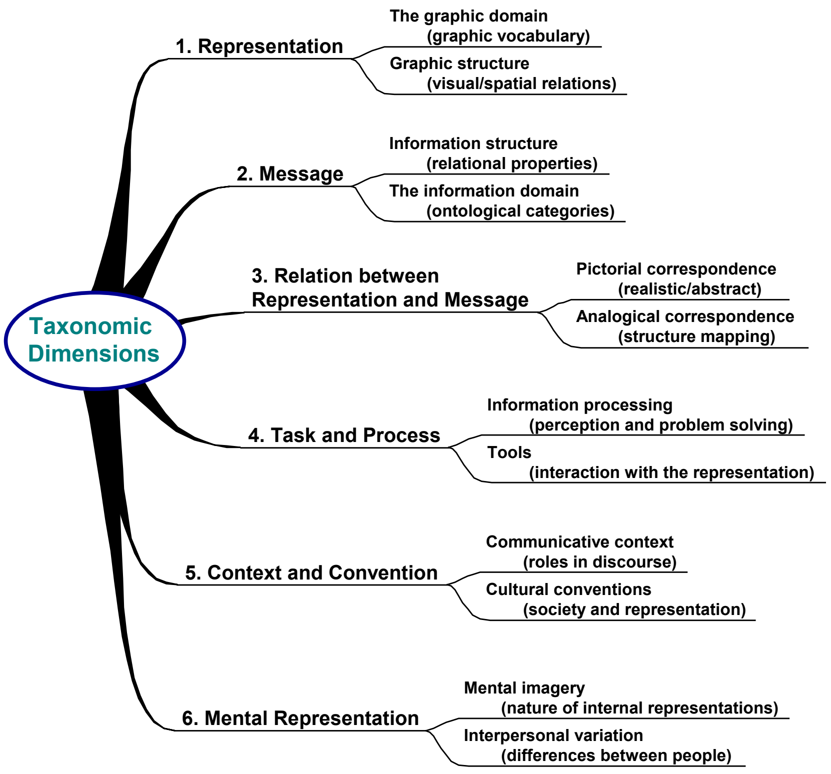

Taxonomic Dimensions
1. Representation
The graphic domain (graphic vocabulary)
Graphic structure (visual/spatial relations)
2. Message
Information structure (relational properties)
The information domain (ontological categories)
3. Relation between Representation and Message
Pictorial correspondence (realistic/abstract)
Analogical correspondence (structure mapping)
4. Task and Process
Information processing (perception and problem solving)
Tools (interaction with the representation)
5. Context and Convention
Communicative context (roles in discourse)
Cultural conventions (society and representation)
6. Mental Representation
Mental imagery (nature of internal representations)
Interpersonal variation (differences between people)

Figure 2.10: Taxonomic dimensions for taxonomies of diagrams

The tasks and processes involved in creating, interpreting and modifying diagrams are the concern of work that studies the use of diagrams. These processes can be internal cognitive processes or can be dependent on software tools.

The context and conventions underlying diagrams are the concern of work that analyses the cultural and communicative background in relations to the diagram and to its sociological effects. Finally, research into mental representation studies the nature of the differences between the diagram and its internal interpretation by the mind.

2.2.5 Using Diagrams for Education

Comparisons of verbal and visual tasks have shown that human capabilities differ in the way they are distributed throughout the brain [Tb93]. Different people choose different strategies to accomplish the same tasks. Differences exist also between the strategies that individuals choose in visual reasoning tasks. These strategies are significantly affected by education, expertise and culture.

Advocates of diagrams have suggested that the right hemisphere of the brain is "needlessly at rest and "under-utilised" when using text-only notation for descriptions. Some simple visual tasks require more time when carried out by the left hemisphere of the brain. There are tasks where verbal and visual information can be combined. Memory improves when an image is associated with a task. Larkin and Simon state that diagrams are "*computationally efficient, because search for information can be indexed by location; they group related information in the same area. They also allow relations to be expressed between elements without labelling the elements*" [LS87].

In the work of Brna, Cox and Good [BCG97] three main issues are identified when considering the use of diagrams in education:

- The balance between making things easy and helping students to learn
- The reuse of the learner's diagrammatic knowledge
- The burden of learning unfamiliar representational systems while learning conceptually new subject material

On the first issue, Brna et al, after presenting a survey of the work on the cognitive analysis of diagrams on learning conclude that it is unclear "*whether we should compensate for, or teach to cognitive style differences*". Citing Salomon, Brna et al explain that although ``*the high-verbal learner who is weak in visualization might be supplied with extensive diagrams and left to generate his own verbal representations*'', Cox, Stenning and Oberlander have shown that subjects classed as good diagrammatic reasoners perform better than poor diagrammatic reasoners. Brna et al suggest that the answer to two questions is fundamental in tackling the first issue: "*What tasks do diagrams make easier?*" and "*What are the benefits of making tasks easier?*"

On the second issue, Brna et al, explain that learning a diagrammatic method depends on many factors such as the task, the learner's experience, the characteristics of the diagrammatic method and the physical and social context. They state that "*Many of the problems in becoming an effective 'diagrammatic reasoner' can be viewed in terms of how to generalise from experience and transfer skills learned in one context to another*", and conclude with two questions that must be addressed: "*Can we provide a convincing account of how* *learners gain a high level of competence at operating with relatively unfamiliar external representation systems?"* and *"How important is the translation skill ?"*.

On the third issue, Brna et al explain that students can be excessively burdened as diagramming methods are typically taught together with the subject. Students can get confused by the two representations and *"commit to an interpretation of the representational system (or even some specific representation) which may be inconsistent or flawed in some way in relation to the intended meaning"*. An important question must be addressed for this issue: *"How can we partition the cognitive load in a sensible way?"*.

In specific subjects, the results for the use of diagrams on learning are extensive. Ackermann and Pope have presented findings from an experiment that aimed to determine the suitability of diagramming tools as a means to improve learning in software design. In their view, diagrammatic tools can be extremely useful. Their results indicate a definite improvement in learning. Students found the tools useful and favoured this approach [AP89]. Jeffrey reports on the experience while using petri-nets to teach concept in operating systems. Although it takes one or two lectures to teach petri nets, Jeffrey argues that the illustration of OS concepts using such diagrams promoted deeper understanding [Jj91]. Clarke proposes *Possible Model Diagrams* as an effective tool to teach logic to undergraduate students. In his view, diagrams are simpler to remember because of their visual appeal [Cm93].

2.2.6 Summary

Section 2.2 introduced diagrams, presented interesting historical facts and illustrated some of the most commonly used diagrams. Research on diagrams has considerably risen in the last half of the 20th century. The interest in diagrams varies to the degree that there is a proposition of creating a discipline for the study of diagrams. In order to classify the diversity of interests, Blackwell's taxonomy of taxonomies for research on diagrams has been overviewed. According to the education technology community, diagrams present unique learning advantages and offer many research opportunities.

2.3 Software Engineering

Systems for both CBA and diagram editing are complex software artefacts. The design and implementation of complex software is a concern that belongs to the field of software engineering. As a discipline, software engineering studies the process of managing the lifecycle of software. This lifecycle, according to Sommerville, includes the stages of software specification, development, management and evolution [Si96]. Schach compliments this view by highlighting that the aim of software engineering is the production of quality software, delivered on time, within budget, and satisfying users' needs [Ss93].

To simplify complexity and to increase software maintainability and reusability, object-orientation employs a range of ideas such as encapsulation, modularisation, abstraction and hierarchy. Object-orientation promotes the notion of maintaining software on a design level as a means to facilitate reuse, maintainability and change. Concise reviews of object-orientation are given by Korson and McGregor [KM90]. Comprehensive surveys examining directions in research on object-orientation are given by Guerraoui [Gr96]. Good traditional textbooks include Booch [Bg91], Coplien [Cj92], Rumbaugh [RBP+91], Meyer [Mb88] and Wirfs—Brock, Wilkerson and Wiener [WWW90].

The problem with object-orientation is that the design and implementation perspectives are inherently inconsistent. This causes the slow and iterative nature of the development process and the increased demands on testing and understanding. However, object-orientation does not produce reusable parts without careful engineering for reuse. The reuse community has suggested domain engineering as a technique to design reusable software [CE00].

The decade of the 90's saw the introduction of design concepts such as software patterns [GHJ+94], maintenance techniques such as refactoring [Op92], [Fm99], and theories on the building of object-oriented frameworks to derive families of related applications [Lt95], [JFS99]. The importance of a sound architecture, with low coupling and high coherence between its parts, has often been appreciated as a key factor in software quality [Ra96].

A simple definition of a software pattern is given by Schmidt: "*A pattern is a recurring solution to a standard problem*" [SJF96]. Early work in software patterns includes that of Gamma, Marty and Weinand [GMW88], Vlissides [Vj90], Coplien [Cj92], and Coad [Cp92]. At the end of 1994, the "Design Patterns" book by Gamma et al introduced patterns to the software community [GHJ+94]. Each design pattern captures the intent behind a reusable solution to a commonly recurring design problem by specifying the objects, classes, collaborations and interactions that are involved. Design patterns not only promote reuse of designs that have resisted the test of time but also propose a common vocabulary to communicate software at a much higher granularity than that of objects, classes, operations and attributes.

Object-oriented frameworks encapsulate design decisions for a particular domain and support reuse not only of implementation but also of design and analysis. Johnson gives two definitions: "*A Framework is a reusable, semi-complete application that can be specialised to produce custom applications*" and "*A framework is a set of classes that embodies an abstract design for solutions to a family of related problems*"[JFS99]. Frameworks are designed to capture the abstract and reusable parts of a domain so that new software can be developed for that domain simply by specialising the framework. Frameworks are represented by a set of interfaces, abstract classes and collaboration specifications of their instances. Applications can be developed using a framework by overriding, parameterising, configuring and modifying. Johnson states "*frameworks are an object oriented reuse technique*" [Jr97].

The benefits of using frameworks are improved modularity, reusability, extensibility, and control inversion. Frameworks are usually complex and hard to understand in terms of classes and objects. The important classes are the abstract classes that denote the responsibilities of the various class hierarchies. The development process of a framework is highly iterative and requires a deep understanding of both the application domain and of object-orientation. Software patterns are used explicitly in frameworks not only to reuse proven solutions, but also to document the design of the framework and help to communicate it to other developers. New applications produced using the framework often need to subclass classes of the framework and specialise behaviour.

2.4 Summary

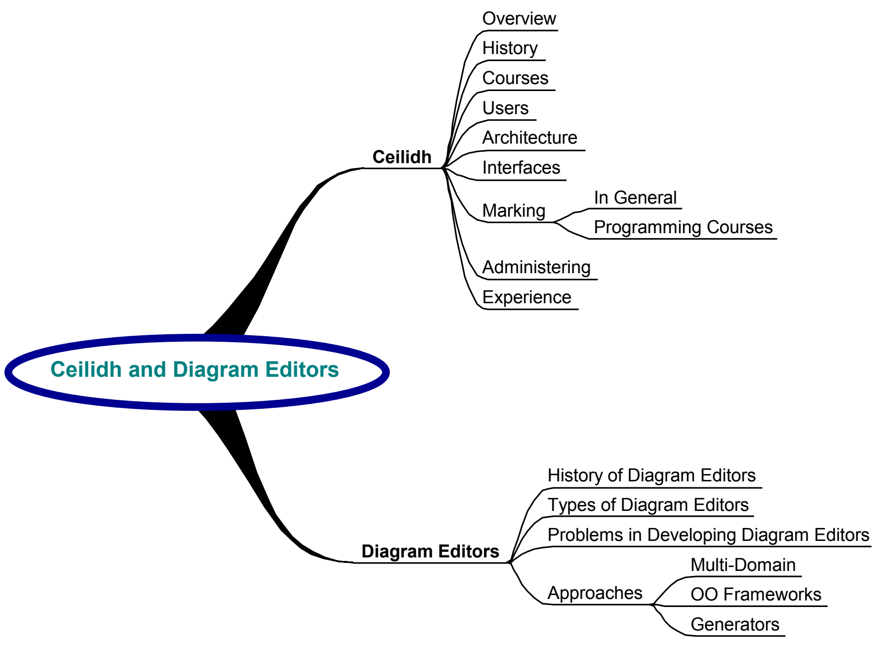
This chapter introduced the areas of CBA, diagramming and software engineering. CBA is a field of learning technology that is concerned with the online assessment of student work. The benefits of using CBA are both practical and pedagogic. CBA can be used for formative, summative or diagnostic purposes. The type of the student response to the system divides CBA systems into fixed and free response. Multiple-choice questionnaires, simple string/numeric answer exercises and hotspot graphical exercises are all types of exercises that belong to the fixed response type of CBA. Programs, essays, and diagrams belong to the free response type of CBA. Many systems exist for fixed response CBA whereas a few prototype systems have been reported for free response coursework.

To combine diagrams with assessment section 2.2 reviewed the area of diagramming. In order to classify the diversity of research interest in the diagramming literature, Blackbell's taxonomy of taxonomies has been discussed. Diagrams present unique learning advantages according to the educational technology community and should play a larger role in education.

Finally, the field of software engineering has been briefly reviewed, and software patterns and frameworks were discussed in the context of software development.

Chapter 3,

Existing work: Ceilidh and Diagram Editors



"Civilization advances by extending the number of important operations which we can perform without thinking about them"

Alfred North Whitehead, an Introduction to Mathematics

Introduction

The previous chapter introduced CBA, diagramming and software engineering as subjects relevant to the development of diagram based CBA. This chapter reviews existing work on the Ceilidh CBA system and on systems that address the editing of user-specifiable diagram notations.

Section 3.1 begins with a brief historical overview of the Ceilidh CBA system and an explanation of its notion of a course. A review follows, explaining how the supported functionality is distributed into five types of responsibilities that represent five user views. Ceilidh's initial objectives have been to create an extensible and portable CBA system that supports multiple courses, automatic marking with feedback, multiple interfaces and remote learning. These objectives have been supported by principal decisions throughout Ceilidh's design and are further discussed.

Section 3.2 introduces graphical diagram editors and reviews the software engineering difficulties concerning their development. The most notable approaches to developing diagram editors are surveyed and categorised into multi-domain diagram editors, diagram editor frameworks and diagram editor generators. The motives, aims and principles of each approach are examined.

3.1 The Ceilidh CBA System

The Ceilidh system was originally implemented to address the practical problems involved in the teaching of programming courses to large numbers of students. The problems, as discussed in chapter 1, occur in the presentation of material, in the administration of the course, and most importantly in the assessment of student work. Ceilidh supports features to address each of these three problem areas.

For the presentation of information to the students, Ceilidh implements a hierarchical course structure with courses, units and exercises in which lecture notes, tutorials and other course related information can be organised and published. Ceilidh keeps students informed of their marks and returns feedback for submitted coursework.

For the administration and management of courses, Ceilidh assists in a number of ways. Firstly, it defines a secure and on-line process for the creation and submission of student coursework. Secondly, it supports customised views for its users, distributing the various responsibilities amongst them. The view of the teacher supports functions for monitoring individual and overall student progress as well as individual and overall exercise progress. Thirdly, Ceilidh handles the archiving of submitted student work, and manages the intercommunication between users. In addition, Ceilidh supports features for plagiarism detection.

For the automatic assessment of student work, Ceilidh has put forward a generic technique that can be applied to the development of both fixed and free response types of assessment. By introducing the concept of *marking tools*, Ceilidh supports the development of a variety of CBA exercises. Marking tools embody programs that examine a specific quality in the submitted coursework and return marks and feedback to the invoker. Marking tools have been implemented in Ceilidh for assessing courses in programming languages, multiple choice questionnaires, question/answer exercises, single sentence /word answers and essays/reports.

Although the presentation of information to the students is an important aspect of the learning process, Ceilidh has concentrated primarily on the automatic marking of student work and secondly on the administration of course modules.

3.1.1 Ceilidh's Development History

Foxley conceptualised, designed and implemented the original Ceilidh in 1988 [BBF92]. Figure 3.1 presents a timeline diagram that highlights the most important developments in Ceilidh's history. Ceilidh was initially tested at the University of Nottingham for four years supporting the teaching of a course using the C programming language. In 1992, with funding from the Teaching and Learning Technology Program (TLTP), a second release became publicly available to all academic institutions. Ceilidh's second version added more features, a C++ course and increased security.

In 1995, a final release added a statistics package for the monitoring of various aspects of the course, an X-window front-end and a secure web interface for all user views, as well as several new courses.

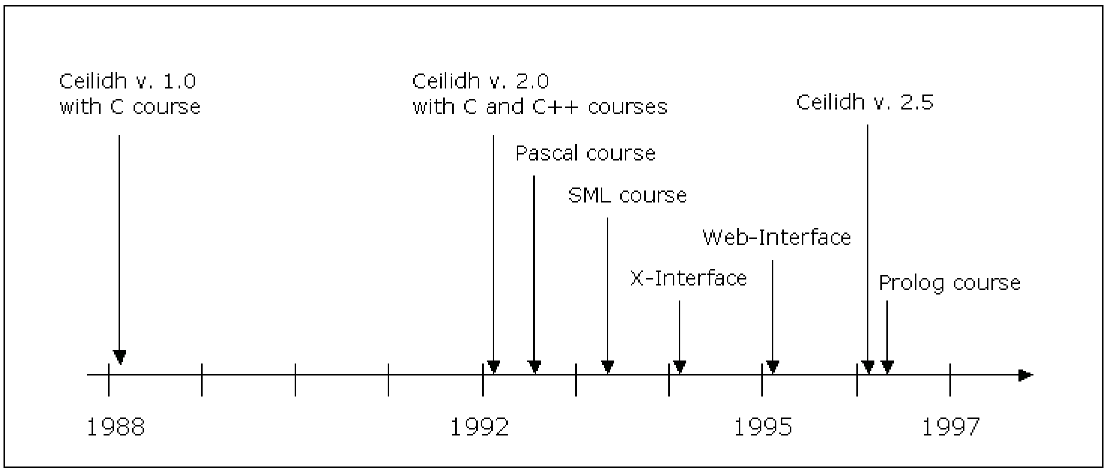


Figure 3.1: Ceilidh's development timeline

In 1996, according to a log analysis by Foxley, Ceilidh had been used by approximately 220 universities in more than 30 countries [FHG96]. Among the academic institutions that contributed new courses, the most notable are Heriot-Watt University with an SML course, Royal Holloway University with a Pascal course, Helsinski Technical University with a Fortran course, Ulster University with a Modula2 course and Liverpool University with an SQL course. Additional courses, have been implemented at the University of Nottingham for Prolog [MGC98], for the Z specification language [FSZ97] and for UNIX based software tools [FHG96]. Luton University, Sheffield Hallam Polytechnic, Manchester Metropolitan University, Cardiff University, Rotterdam University and Ngee Ann Polytechnic have been involved in various aspects of developing extensions for Ceilidh. Detailed historical information for the development of Ceilidh can be found in [BBF+93] and [FHT98].

3.1.2 Courses in Ceilidh

A Ceilidh course has a hierarchically organised file structure that contains units and exercises. Figure 3.2 illustrates a course structure. Ceilidh can host multiple courses simultaneously. Courses are authored by course developers who are familiar with Ceilidh's formats, tools and file structures. Every exercise in the hierarchical course structure has associated files for publishing information, such as lecture and tutorial notes, files used to configure the automatic assessment, such as marking properties and testing data, and files that are specific to the running course, such as student marks and solutions.

As a course progresses, exercises are assigned to students as formal coursework. Students have to submit solutions for their exercises within given deadlines after which no submission is permitted. Upon submission, marking is automatically carried out. This is accomplished with the use of marking tools, which represent different aspects of the exercise's solution quality such as, for example in the case of programming courses, metrics for the readability of source code. Instant feedback is returned by the system on the student's submission which reveals in detail the strengths and weaknesses of the solution.

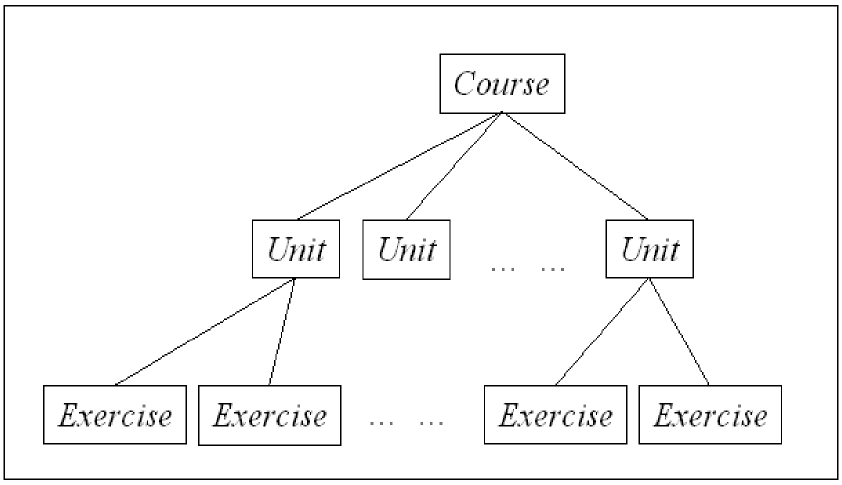


Figure 3.2: The structure of a course in Ceilidh consists of units and exercises

A new course in Ceilidh can belong to any of the following types of supported courses:

- programming languages
- multiple choice questionnaires
- question/answer exercises
- single sentence /word answers
- essay/report

Each type of course has its own marking tools and data formats. Comprehensive guides for developing courses include [BBF96] and [BBF+96]. New types of exercises can also be created, however not without some considerable programming involved.

The development of exercises on courses that have already been implemented needs significantly less effort.

3.1.3 Ceilidh's Functionality and Users

Ceilidh distinguishes between five roles for its users with respective responsibilities: students, tutors, teachers, developers and administrators. Each type of user has their own interface to the system for every course level. Every user interface has a system, a course, a unit and an exercise level that offer functionality specific to the responsibilities of each user as well as to the type of exercise.

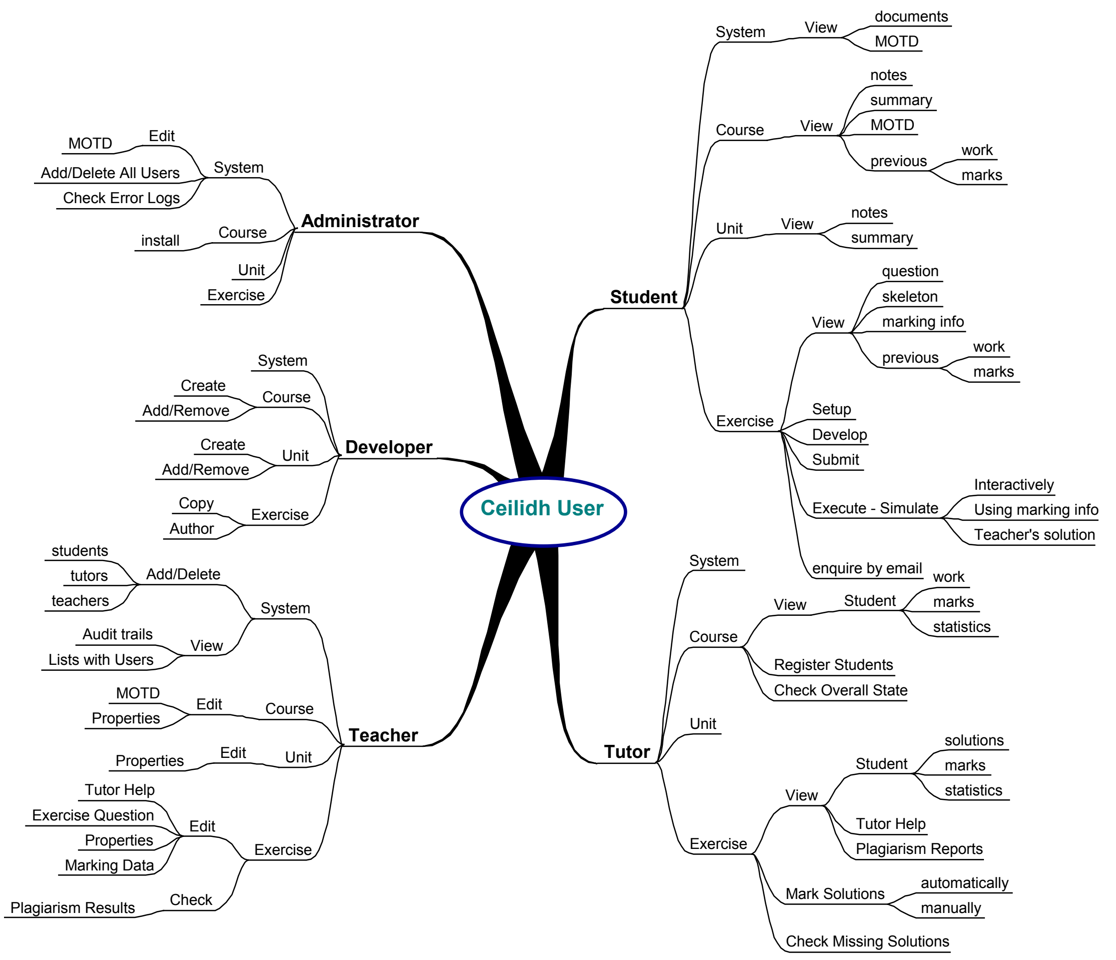


Figure 3.3: Ceilidh user-responsibilities for every course level

Without taking into consideration the specific tasks defined by the type of exercise, figure 3.3 illustrates the available functions for each type of Ceilidh user. Ceilidh's functionality on each user-view is additive: the view of higher-level users incorporates the functionality of the views of the lower level users. For example, a teacher can do everything a tutor and student can, and an administrator can do everything all other users can.

Students use Ceilidh's course levels to navigate through the course structure, to study the available reading material, and to get overall information about their previous work. At the exercise level, Ceilidh has options for setting up and developing an exercise, for executing and testing the solution, for submitting it and for checking previous exercise work and marks. Students can enquire about aspects of the course using email. As each course may be associated with specific tasks related to the development of respective coursework, the exercise level view is customised to the exercise type. Options to execute or test the solution can be configured. Tutors are assistants to the teacher responsible for the course. Groups of students are assigned to each tutor. The tutor's responsibility is to supervise, support and guide their assigned tutees, to administer student registration and to monitor progress. Tutors can instruct Ceilidh to re-mark a submitted solution and to return a full analysis of the results. They can also mark a solution manually and update Ceilidh to reflect the new mark.

Each course in Ceilidh needs to be associated with a teacher. The main responsibilities of a teacher include managing the registration of users on the course, providing and updating course material, editing exercise specific properties to suit assessment preferences, and in general monitoring and supervising all the aspects of the running course. Teachers obtain feedback for their teaching from tutors. They can also monitor exercise and student overall results in detail. Teachers are also responsible for ensuring the security of the marking by setting and checking audit trails and executing plagiarism detection tools.

Developers are users who build and extend Ceilidh courses. The developer's view has functions to create and edit course structures and to author all the appropriate files for an existing type of exercise such as programming, essay, question/answer, MCQs etc. Often the development of a new type of exercise can be based on configuring generic tools for existing courses that process text-solutions. In domains where Ceilidh's generic marking tools are unsuitable, new marking tools need to be developed.

Administrators in Ceilidh have the responsibility of installing Ceilidh, adding ready-made courses, administering the registration of all users and checking errors. They also supervise networking and execution performance, and fine-tune the system.

3.1.4 Ceilidh's Architecture and User-Interfaces

Ceilidh's design objectives have been to support multiple courses, automatic marking with feedback, multiple interfaces and remote learning, whilst maintaining extensibility and portability [BBF+94]. Ceilidh took an architectural approach based on three layers in order to separate the data from the logic and the user-interfaces. Figure 3.4 depicts the three layers as they relate to Ceilidh's users.

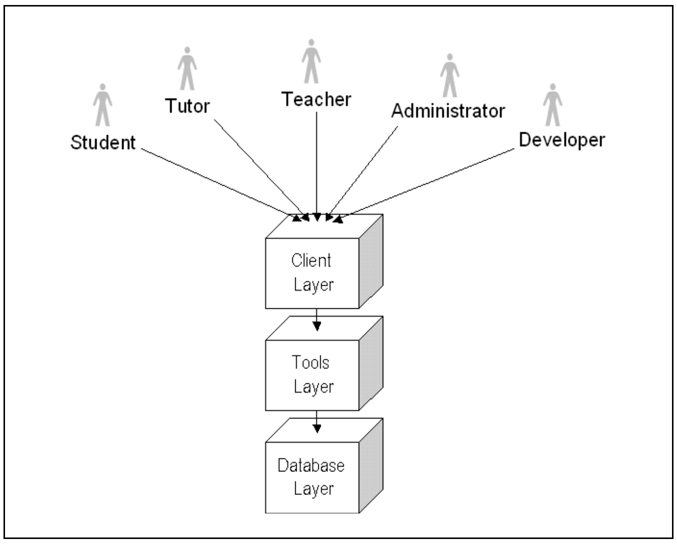


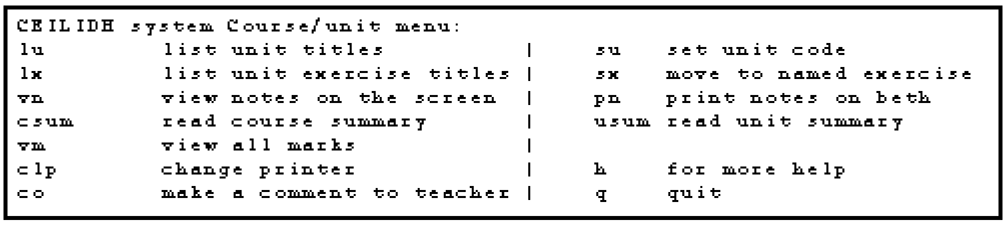
Figure 3.4: Ceilidh's three-layered architecture as it relates to its users

The database layer includes information stored for courses, such as authored material for notes and exercises, and data archived for the year such as user lists, submissions and marks, along with other transient properties and configurations. The structure of the information that constitutes the exercise material depends on the type of assessment. For example, programming exercises have different organisation than exercises based on MCQs. Courses can be developed for any exercise type supported by Ceilidh. The marking feedback is part of the properties of the exercise and is set by the exercise developer.

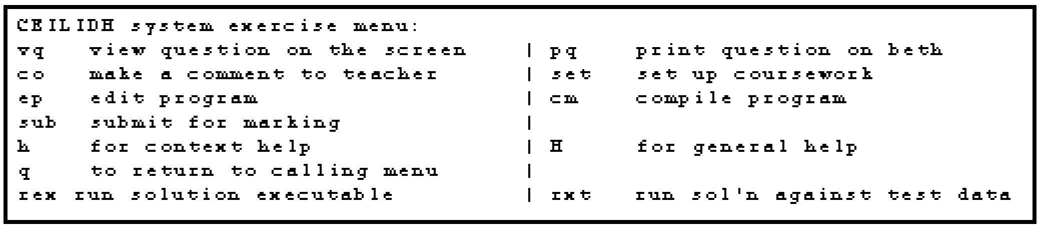
The tools layer consists of executable programs that are dependent on the formats of the exercise files within the database layer and inter-communicate through predefined protocols. The purpose of the tools layer is to increase extensibility. Many tools have been developed since Ceilidh's initial release providing functionality and metrics in new domains. Ceilidh's latest version includes 70 tools, 51 of which are Unix shell scripts and 19 are programs written in C [BBF+94].

The user-interface layer accesses the tools layer in order to make the functionality available. Separating the user-interface from the tools made easier the development of new views. Four types of interfaces have been developed for all users:

- **Dumb Terminal Menu Interface:** This is a simple text-based interface, in which users select commands from a text menu. It was built to support courses with a large number of novice students. At the time of developing the dumb terminal menu interface old hardware resources had to be used simply because there where not enough graphics-based terminals available. Figure 3.5 depicts the student's view for the course and exercise levels.



```
CEILIDH system Course/unit menu:
lu        list unit titles          |    su    set unit code
lx        list unit exercise titles |    sx    move to named exercise
vn        view notes on the screen  |    pn    print notes on beth
csum      read course summary       |    usum  read unit summary
vm        view all marks            |
clp       change printer            |    h     for more help
co        make a comment to teacher |    q     quit
```



```
CEILIDH system exercise menu:
vq   view question on the screen     | pq    print question on beth
co   make a comment to teacher       | set   set up coursework
ep   edit program                    | cm    compile program
sub  submit for marking              |
h    for context help                | H     for general help
q    to return to calling menu       |
rex run solution executable          | rxt   run sol'n against test data
```

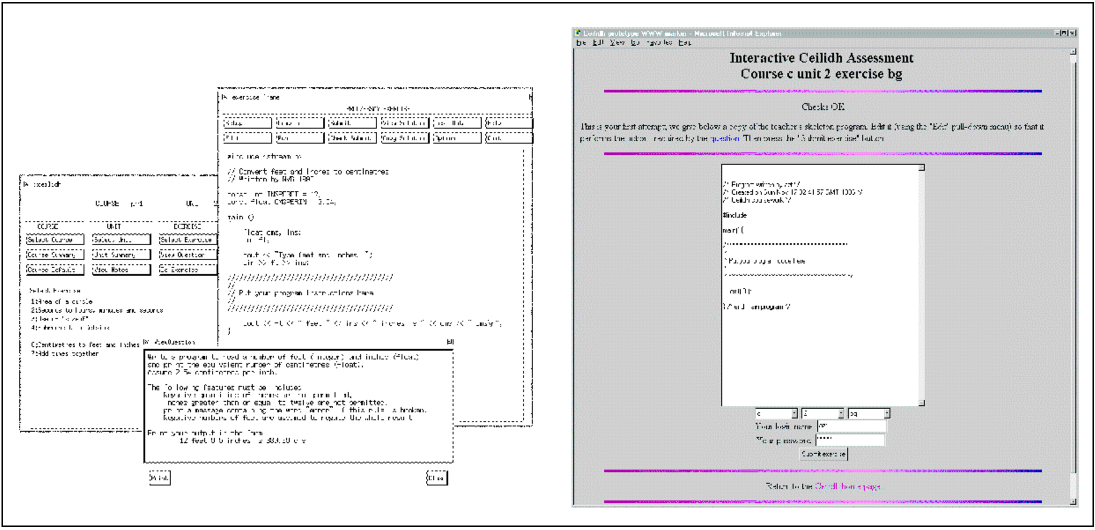
Figure 3.5: Student's view of Ceilidh's dumb terminal interface

- **Command Line Interface:** Every menu action from the Dumb Terminal Menu Interface is available as a shell command, which offers added flexibility, as it is possible to execute Unix commands or logout at any time without affecting Ceilidh's state. Experienced users prefer to issue commands instead of using the menu for increased speed.

- **X-Window Interface:** The X-Window Interface was developed using the OSF/Motif 1.2 toolkit [Osf91] and was the first user interface that used a graphical environment.

- **World Wide Web Interface:** This interactive HTML-based interface is generated automatically for efficiency, and is stored on a web server [FNT97].

Secure CGI scripts written in UNIX shell script are used for user identification and coursework submission [FHT+98].



```
#include <iostream.h>

// Convert feet and inches to centimetres
// Written by [illegible]

const int INSPERFT = 12;
const float CMSPERIN = 2.54;

main ()
{
    float cms, ins;
    int ft;

    cout << "Type feet and inches: ";
    cin >> ft >> ins;

//////////////////////////////////////////
//
// Put your program instructions here
//
//////////////////////////////////////////

    cout << ft << " feet " << ins << " inches = " << cms << " cms\n";
}
```

```
Write a program to read a number of feet (integer) and inches (float)
and print the equivalent number of centimetres (float).
Assume 2.54 centimetres per inch.

The following features must be included:
    [illegible]
    inches greater than or equal to twelve are not permitted;
    print a message containing the word "error" if this rule is broken.
    Negative numbers of feet are assumed to negate the whole result

Print your output in the form
        12 feet 0.5 inches is 393.07 cms
```

This is your first attempt, we give below a copy of the teacher's skeleton program. Edit it (using the "Edit" pull-down menu) so that it performs the action required by the question. Then press the "Submit exercise" button.

Return to the Ceilidh home page

Figure 3.6: Student's view of Ceilidh's X-window and Web interface

Figure 3.6 illustrates the exercise view of a programming course in C++ as it is seen by students on an X-window and web user interface.

3.1.5 Automatic Assessment in Ceilidh

Ceilidh implements a mechanism to perform automatic assessment in which the student solution is examined by marking tools. Marking tools are specific to the type of the course and can be further configured to the requirements of a specific exercise. Upon execution, marking tools evaluate specific aspects of the quality of a solution and return results to Ceilidh for further processing. After the execution of all the marking tools that participate in the marking of a student solution, Ceilidh calculates the overall mark and returns feedback to the student.

The scheme for invoking the marking tools that mark a student solution is described in a *marking action*. A marking action is a configuration file that dictates how each exercise is automatically assessed. It contains invocations of marking tools associated with weights that represent the highest mark that a marking tool can return. The sum of all the weights in a marking action represents the highest mark that Ceilidh can give for a single exercise, which typically has the value 100.

The concept of modelling the marking scheme, using a marking action, is fundamental in Ceilidh because it allows the configuration of the marking on a per exercise basis. For every exercise a marking action describes the highest level decisions of how the solution is going to be marked. For assessment based on fixed response, available marking tools carry out a simple matching search between the student solution and the model solution. For free response based assessment, the available marking tools require further configurations to the domain of the course and the specifics of the exercise. These configurations are specified using regular expressions in the form of *oracles*, which are text-checkers representing simple marking criteria.

From all types of exercises that Ceilidh supports, the one that has similarities with diagram-based assessment is the type for programming. Both programs and diagrams have explicit structure. The next section discusses how the programming type of exercises is designed to allow the assessment of new programming languages, and how the general marking tools facilitate the authoring of new exercises.

3.1.6 Assessing Programming Coursework

Ceilidh distinguishes between *dynamic* marking tools that test the runtime behaviour of programs and *static* marking tools that perform text based matching on the source code [ZF91]. Available dynamic marking tools include those that assess the execution correctness and those examining the efficiency of the submitted program with respect to the exercise's question. The available static marking tools are comprised of tools that assess the typographic and structure correctness, the algorithmic complexity and other measures related to the exercise features of the program.

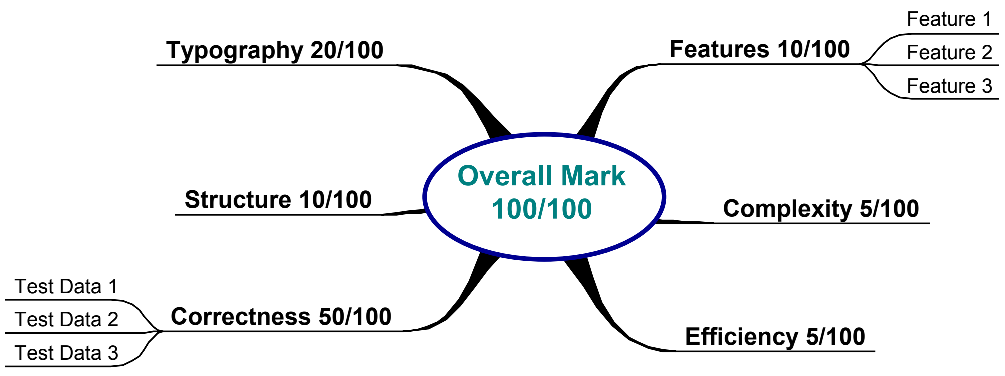


Figure 3.7: A typical marks distribution among various metrics for a C exercise

Figure 3.7 illustrates a typical distribution of marks described within a marking scheme for exercises based on the C programming language.

3.1.6.1 Dynamic Metric Tools

The dynamic metric marking tools execute the student's program against predetermined sets of test data and validate the output. As several forms of output can be considered valid according to the stated question, the responsibility of describing patterns of solutions lies with the developer of the exercise. In order to achieve this, Ceilidh developers have to use oracles according to their anticipation for possible weaknesses in student solutions.

Oracles check the output for the presence or absence of a certain token or combination of tokens. Each program would typically undergo several dynamic tests, with each test typically involving several regular expressions. Every oracle has an associated weight that might or might not be scaled, according to Ceilidh's mark scaling configuration.

Other dynamic metrics are based on common profiling techniques. During the execution of a program against sets of test data, programs can be profiled using tools similar to the UNIX *tcov* tool [DS98]. The *tcov* tool is a profiler that returns an execution count for every section of the program, a summary of the sections with the maximum counts and a summary of any non-executed sections.

3.1.6.2 Static Metric Tools

Static metric tools analyse the student's source code for typography, complexity, program structure, and features specific to the exercise.

Typography investigates the readability and maintainability of source code. Typical typographic metrics evaluate the presence and usage of comments, the preferred layout of braces, square and round parentheses, the length of identifiers, the ratio of white space to other characters and other known aspects of the typographic quality of programs. Table 3.1 lists some of the most notable typographic metrics.

Typographic Metric	Typical Value for C
Percentage of blank lines	15%-30% blank
Average characters per line	15-30 cpl
Percentage of average white space per line	10-30% blank
Average identifier length	5-15 characters each
Percentage of names with good length	70%-100% reasonable length
Percentage of comment lines	10%-60% lines to be comments
Percentage of chars in comments	10%-60% of source's length
Percentage of indentation	100% indentation
Percentage of indent errors consisting of braces	100% indentation
Percentage of indent errors consisting of brackets	100% indentation
Percentage of indent errors consisting of parenthesis	100% indentation
Percentage of uncommented closing brackets	100% comments

Table 3.1: Ceilidh's typographic metrics

Several marking tools have been implemented to examine the source code complexity using measures, like *software science* proposed by Halstead [Hm77] and *cyclomatic number* designed by McCabe [Mt76]. Other metrics include techniques proposed by Van Verth [Vp85], Henry and Kafura [HS84] and Oviedo [Oe80]. These metrics check the frequency of occurrence of various programming constructs and compare it with the model solution. Table 3.2 lists some of the most used in Ceilidh complexity metrics.

Program Complexity Metrics
Methods of types & data declarations
Reserved words, include files & literals
Assignment statements & complexity of expressions
Library functions & function calls
Operators, conditional statements (including their depth), loops (inc. depth)
Maximum depth of braces, square brackets and round brackets

Table 3.2: Ceilidh's program complexity metrics

Program structure metrics rely heavily on external tools such as the standard Unix C utility lint and variants for other languages, which comment on program source code. The student's mark is based on the occurrence or absence of possible problematic aspects of the source code. Table 3.3 illustrates the most notable program structure metrics.

Program Structure Metrics
Variable assigned / defined / declared but never used
Variable undeclared / re-declared previously as different kind of symbol
Variable used before set
Value computed / returned by a function never used
Value computed / returned by a function sometimes used
Statement not reached / with no effect
Assignment of integer from pointer lacks a cast
Float or double assigned to integer data type
Comparison is always 1 due to limited range of data type
Data definition lacks type or storage class

Table 3.3 Ceilidh's program structure metrics

Feature metrics are metrics that are specific to each exercise. They are used to ensure that the submitted solution uses the current week's newly introduced concepts. Table 3.4 list some examples of program features metrics.

Program Features Metrics
Numeric denotations which should have really been set as constants
Language specific features (e.g. use of switch statements, overloading, inline etc.)
Non-use of preferred denotations (e.g. "<=99" instead of "<100")
Definition of specific ASCII characters by integer detonations (e.g. 64 vs. 0100 (octal))

Table 3.4: Examples of Ceilidh's features metrics

Ceilidh developers use the existing marking tools to create new exercises in various programming courses. Among the available marking tools for programming languages, Ceilidh offers 120 single measurements and supports the creation of new ones. The design document for Ceilidh [BBF+94] describes all the tools and oracles.

The developer's guide for Ceilidh [BBF96] gives the details for developing exercises and configuring each of the available measurements.

3.1.7 Administration and Management in Ceilidh

Administration responsibilities in Ceilidh are distributed to tutors, teachers, developers and system administrators. Facilities exist to update course information, manage user registration, monitor progress, check plagiarism reports and manage audit trails. Tutors and teachers share the responsibilities of administering Ceilidh on a per semester and course basis.

Typically, the teacher sets a number of exercises to open on a weekly basis. Exercises might be in any of five states: open, late, closed, private or public. The first three states refer to the exercise's availability to submit, and the last two on the visibility of the exercise's solution. When an exercise is set as public, its solution is available to the students for viewing.

Monitoring facilities are available for many aspects of the running of a Ceilidh course. Tutors monitor their assigned tutees and teachers monitor the overall class. Monitoring facilities support queries to find specific and overall results in respect to students, exercises and courses. Results can be illustrating not only an overall mark but also the contribution of each marking tool and oracle. This helps teachers to acquire an understanding of the specific and general weaknesses of the delivery of the course and subsequently to introduce corrective measures. Ceilidh also supports queries to find missing students or submissions and to automatically send an email to the tutor responsible for the student.

Students may submit a solution to an exercise several times. The number of allowed submissions along with the enforced time between submissions is a property of the exercise that must be set by the teacher before opening the exercise. Typically, three submissions are allowed per exercise with five minutes enforced delay between submissions. More submissions together with instant feedback on the solution weaknesses result in higher than normal overall marks. As the distribution of marks has to agree with departmental policies, Ceilidh scales the results to a suitable distribution. The configuration of the mark scales is based at the course level and is set by the teacher at the end of the course.

Automatic plagiarism detection in Ceilidh is initiated either automatically at the closing of an exercise or manually by the teacher. The plagiarism detection tool compares every student solution with each of the other student solutions. The comparison uses a number of techniques to improve detection. Firstly, file attributes and contents are checked to find a percentage of similarity. Secondly, for exercises in programming languages, comments are removed, variables are substituted, and a consequent comparison takes place.

3.1.8 Experience Using Ceilidh

Foxley et al have described in [BBF+93], [FGZ+93], [BBG+93], [BBF+95], [FHT98] and [FHT+99] the experience accumulated over the years of using Ceilidh. Sources of feedback include results from questionnaires given every year to all the types of users, informal discussions with students and staff, studies of statistical data, reviews of archived work and student comments sent.

Using feedback from all the above sources, Foxley et al, support that Ceilidh offers both practical and pedagogic benefits. Firstly, Ceilidh saves time with large numbers of students, it is cost effective, saves on resources, and can be administered by teaching assistants freeing lecturers for teaching. Courses in Ceilidh can be reused and/or changed with relatively little effort. Notes and exercise properties can be customised to reflect the preferences of the teacher.

Secondly, Ceilidh offers pedagogic benefits. Marking with Ceilidh is consistent, equitable, incremental, and redeemable. Instant feedback is returned to the students and plagiarism is detected.

3.1.9 Summary

Ceilidh is a CBA system that supports the presentation of information to the students, the administration of the course, and the automatic assessment of student coursework. The structuring of a course into units and exercises is a fundamental concept in Ceilidh. Ceilidh's exercises support automatic assessment and can be configured for both free and fixed response type of assessment. Ceilidh differentiates among five types of users and provides views that are not only specialised to the user-type but also to the type of exercise.

Ceilidh's three layer architecture was designed aiming to support multiple courses, automatic marking with feedback, multiple interfaces, remote learning and increased extensibility and portability. Various user interfaces, tools and courses have been built and used throughout Ceilidh's development history.

The marking in Ceilidh is described in a marking action configuration file that contains the invocation of marking tools associated with marking weight. Each marking tool is further configured by the properties of the exercise. Oracles are used to describe such properties. Oracles search for expressions in text and use regular expressions. Six marking tools exist for the programming type of exercises and these are configured to reflect metrics for existing programming languages.

Ceilidh also contains functions for the administration of courses. It monitors student progress, manages the intercommunication among users, caters for the weekly maintenance and editing of the material and offers plagiarism detection.

In the 10 years of experience using Ceilidh, reports from several institutions demonstrate significant practical and pedagogic benefits. However, as the next chapter describes, Ceilidh presented scalability, performance and maintainability problems that made its redesign and re-implementation necessary.

3.2 Diagram Editors

A diagram editor is a computer program for creating and editing diagrams by *direct manipulation*. Shneiderman coined the term direct manipulation to refer to an interactive model of Human Computer Interaction (HCI) in which the user manipulates application objects and receives immediate visual feedback about the changes [Sb83]. The argument for introducing direct manipulation to an application is that the application engages the user to the task in a more intuitive and natural way. Usability can therefore improve for both novice and expert users. The concept of direct manipulation has been the subject of many studies in HCI. Meyer has documented extensive summaries of the state of the technology on the HCI aspects of direct manipulation in [Mb90], [MHC+96] and [MHP00].

Section 3.2 introduces the issues involved in the development of diagram editors. As highlighted previously in section 2.2, research into diagrams relates to many fields and exhibits a wide range of interests. This also applies to diagram-editors. For example, editors that are developed to aid the study of geometry [Ln97] have inherently different functions from editors used to specify software [MM85]. There is common agreement that in all but the most trivial case that the development of diagram-based graphical editors is a complex task [Vj90], [FWW00]. The cost of building a new diagram editor often exceeds its benefits.

A number of approaches have been pursued to address the development of graphical editors. The first subsection gives a brief historical overview of the graphical editors that first used direct manipulation. This is followed by a categorisation of graphical editors into bitmap, vector and object based editors. The key approaches in developing diagram editors are presented and categorised into multi-domain diagram editors, frameworks and diagram editor generators.

3.2.1 History of Diagram Editors

The history of diagram editors is very closely related to the history of the Graphical User Interface (GUI) [Mb98]. Surprisingly, however, the first diagram editor preceded GUIs. Sketchpad, developed by Ivan Sutherland in 1963, was the first graphical editor supporting the manipulation of objects using a light-pen. Sketchpad was a pioneering system in so far as it was the first to demonstrate a direct model of interaction for selecting, describing and moving objects, zooming, and making lines, corners and joints. Sketchpad also introduced the concept of *rubber-banding*, which is used to give visual, incremental and real time feedback to the user about the changes in effect of the running manipulation. In addition, Sketchpad included a technique to represent geometrical constraints between the elements of the diagram. Finally, it incorporated object-oriented elements such as inheritance and composition. Examples of diagrams supported in Sketchpad were electrical circuit and mechanical design diagrams [Si63].

Two years later, Raskin submitted his Ph.D. on human-computer interfaces in which he describes Quickdraw, the first graphical drawing program [Rj67]. Kay proposed the idea of overlapping windows in 1969 [Ka69], and a year later, headed research on GUIs at the Xerox PARC labs. In 1971, Xerox's Alto computer became the first commercially available computer to be operated by GUI, integrating a mouse with a bitmap display and "WYSIWYG" (what you see is what you get) programs for editing text and drawings. Later in 1977, Kay's concept of the dynabook [Ka77], which was the precursor of laptops, popularised the concept of direct manipulation. In 1984, a project began at Apple [Rl96] studying the design of various types of graphical application to discover a "correct" design. This led to MacApp, the main precursor of object-oriented graphical frameworks.

In the early 1980's, with the arrival of the personal computer and the standardisation of the mouse and bitmap screen, many more types of programs supporting direct manipulation started to appear. CASE tools, CAD systems, visual languages, music editors, graph editors and games are all graphical editors specific to a domain. Generic systems, metasystems, authoring environments and generators have been designed for many of those domains. MetaCASE, metaCAD, generators for visual languages, and authoring environments for multimedia, games and user interfaces are all examples of such undertakings.

In the early 90's, with the wider adoption of object-orientation, generic designs started to appear that represented common functionality across diagramming fields. Multi-domain editors such as Templa and Graphica, object-oriented frameworks such as Unidraw and HotDraw, and diagram editor generators such as DiaGen, aim to simplify the process of developing new diagram editors.

3.2.2 Bitmap, Vector and Object Based Graphical Editors

Graphical editors can be divided into bitmap, vector and object based editors. The point of distinction is based on the internal representation of the model of the drawing.

Bitmap based graphical editors maintain the representation of the drawing in a bit-based memory array. Typically, every position in the array represents the colour of a pixel on the drawing. Interaction using bitmap-based drawing tools result in changes in the memory array. The information that a bitmap drawing may acquire through user interaction cannot exceed the size of the predefined array. Bitmap editors are used mainly for making and editing drawings and images for graphic design and illustration purposes. The directness of the drawing's representation suits the development of a wide range of graphical tools that are based on image processing.

Vector based editors contain a set of primitive elements such as lines, curves and shapes, and a set of tools to manipulate the elements. The internal representation of the drawing is kept in a data-structure that maintains all the primitives and their characteristics. Vector based editors facilitate the interactive positioning of the primitive elements using coordinates in a given space. As such, they suit applications where topological accuracy is needed. Computer Aided Design (CAD) programs, Geographical Information Systems (GIS) and 2D and 3D modelling programs are examples of vector-based editors.

Object based editors represent sets of elements that are specific to a domain. The internal representation of a diagram in an object-based editor depends on the semantics of the modelled domain. Object based editors are suited to domains concerned with structures of interconnected elements. Typically, the representation of the diagram is used for other than illustration purposes. For example, often, circuit editors include tools for circuit simulation, flowchart editors include tools to generate and execute programs, and CASE tools include tools to generate source code. Case tools, visual languages, circuit editors, music transcription editors are examples of object based editors.

3.2.3 Problems in the Development of Diagram Editors

Diagram editors are hard and complex to develop for a number of reasons. Firstly, the editing task and the domain might be complex. For example, editors for domains such as 3D modelling have hundreds of different types of tools, each with its own interactive behaviour. Secondly, usability must be addressed and balanced against functionality. HCI must be considered in a user-centric and realistic manner. Thirdly, as diagram editors are in effect complex GUIs, they require iterative implementation and intensive testing. The absence of tools for automatic testing of GUIs makes testing more time consuming and error-prone. Fourthly, editors are reactive systems that need to be sensitive to user manipulations, and expose a high degree of liveness. Liveness is a term introduced by Tanimoto [Ts90], as Burnett and McIntyre cite, to describe " *the immediacy of semantic feedback that is automatically provided during the process of editing a program*" [BM95].

Diagram editors have also to be robust, to give helpful feedback, to recover gracefully from exceptional cases, and to be forgiving to user mistakes. A diagram editor should also be portable, scalable, expandable, and perform and adapt well to changes like any other good quality software. It should also be able to exchange data with other standard tools.

The limited amount of documentation on building diagram editors makes development even harder. There are no standards, no language support for diagram editors, and libraries that contain graphical primitives are typically very large, complex and not portable.

3.2.4 Approaches in Developing Diagram Editors

The current approaches for the development of new diagram editors can be divided into:

- Multi domain diagram editors
- Frameworks
- Diagram editor generators

Multi-domain diagram editors are graphical editors that have been implemented to address more than one domain. The objective of a multi-domain diagram editor is to address the editing of a family of diagram notations. Some designs specialise in areas such as software engineering or electrical engineering, others are general and can be configured to satisfy the editing requirements of a new diagram type.

Frameworks allow developers to develop new editors by reusing both design and implementation. The development of a new editor using frameworks in contrast to that using multi-domain diagram editors requires significantly more effort but grants more freedom in customisations.

Diagram editor generators generate the software implementation of a new diagram editor by parsing and compiling customised specification languages. Developers that generate a new diagram editor have to give a description in a specification language that defines the editor's semantics and behaviour. In this process, programming is not needed, however, users need to understand the generator and its description language.

Mugridge et al have reported on systems facilitating the development of customised diagram editors [MHG98].

3.2.2.1 Multi Domain Diagram Editors

The boundaries of multi-domain diagram editors are somewhat fuzzy. Single domain diagram editors can be used for more than one domain and visual programming environments can be used as multi-domain diagram editors. Vlissides [Vj90] suggests, three criteria for distinguishing between single domain editors, and multi-domain editors:

- Intended purpose of the editor
- Easiness of modifying the editor's semantics
- Actual purpose of the editor by its users

The first and third criteria are used to distinguish between multi-domain diagram editors and visual programming environments.

Sketchpad, which was the first diagram editor was also the first multi-domain diagram editor. Sutherland designed Sketchpad to be used as a multi-domain graphical editor. Sketchpad included interactive mechanisms for the specification of elements and their constraints. It was used for designing technical diagrams for mechanical and electrical engineering.

Another of the earliest systems, Thinglab, was designed by Borning [Ba79] who used ideas from Sketchpad. Thinglab took advantage of Smalltalk's interpreting nature to incorporate and execute source code at run-time. This mechanism was used by Thinglab for the description and execution of constraints. Constraints in Thinglab could do more than attaching objects together: the visual and non-visual attributes of the drawn elements could be related based on formulas. As formulas were described in Smalltalk, Thinglab could be used to describe complex simulation models.

Hekmatpour's Templa and Graphica system [Hs90], introduced the concept of a *diagram template* to customise a generic graphical editor. Implemented in C++ on a Macintosh computer, the system consists of two programs:

- Templa: an environment for the construction of diagram templates
- Graphica: a generic graphical editor that is specified by a diagram template

Figure 3.8 illustrates the view of both Templa and Graphica. Templa allows the definition of *notation families*, *places*, *links* and *relationships*. Notation families are hierarchies of diagram templates related with inheritance. Therefore, domains can be defined as sub-domains of more general domains. Places are the diagram's nodes, which can be composed by any graphical primitives. Links are drawing primitives such as lines, curves and arrows that are used to show a connection between two places. Relations are constraints that specify whether and how places can be connected. Templa and Graphica contains implementations for templates for entity-relationship diagrams, flow-charts, data-flow diagrams, trees, graphs and petri nets. Nickerson used Templa and Graphica to produce three types of visual languages [Nj94]. In his view: "*it is noteworthy that most diagrammatic representation can be reduced to minor variations on the drawing of graphs, and that the variations can be parameterised in terms of node types, links, and relationship rules.*"

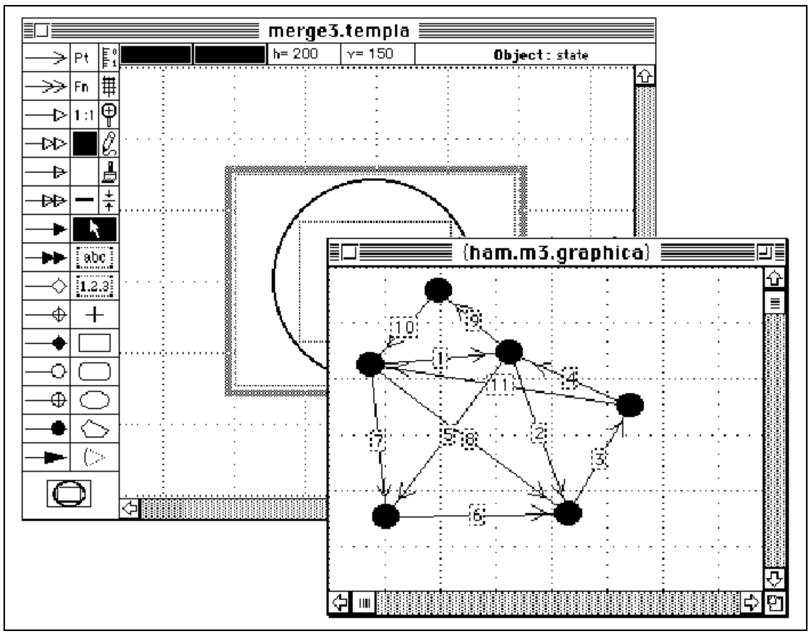


Figure 3.8: Templa and Graphica

A different approach is taken in Ferguson's MetaBuilder [FWW00]. MetaBuilder introduced graphical meta modelling to enable the rapid creation of diagram editors. The idea of graphical meta modelling is that specifications for a new editor are given in the form of a meta-diagram. A meta-diagram describes the elements, relationships and constraints of the modelled diagram. MetaBuilder's meta-notation employs the notion of object-oriented hierarchy. Elements in the meta-diagram can be related with inheritance and composition and therefore complex hierarchies of diagram elements can be specified. Elements can also acquire actions that define their graphical and computational behaviour. On completion of the meta-diagram, MetaBuilder generates a new diagram editor for the modelled diagram type. Although Ferguson argues that MetaBuilder in theory can produce editors for any type of diagram, actual implementations have been created only for diagrams related to software engineering.

After the 1990's, a number of commercial systems were released with the intention to improve the appearance of diagrams for publishing purposes. Multi-diagram editors such as MicroGraphics's ABC FlowCharter [Mic95], Autodesk's AutoSketch [Aut00], Microsoft's Visio [En01], PaceStar's EDGE [Pac01], Computer System Odessa's ConceptDraw [Cso01], and SmartDraw [Sma01] applied the concept of a user-definable diagram element to allow users to create customised notations. Typically, diagram elements have a customised graphical representation, connectivity, data semantics and visual and non-visual attributes.

3.2.2.2 Frameworks for Diagram Editors

Apple's MacApp, one of the first OO frameworks, was designed in 1985 to support the development of GUI based applications on Macintosh computers [App89]. Rosenstein cites that MacApp's designers studied the structure of many programs for the Apple computer while aiming to find a generic representation for applications [Rl96]. The initial version of MacApp was implemented in Object Pascal. It contained 180 classes distributed in several class hierarchies. Its architecture used a variant of the Model View Control (MVC) pattern to separate the model of the application from its appearance and control. MacApp introduced the concept of using objects to represent the actions performed by the user, an idea that was later identified as the command design pattern. MacApp also included a generic implementation for single level undo, an inheritance based event mechanism for delegating and handling events, and a collection of simple GUI components such as windows, scrollbars, buttons, menus and dialogs. MacApp's design was not intended to facilitate the development of diagram editors, however, it influenced greatly the future generation of frameworks. A comprehensive guide to MacApp's architecture is documented by Wilson, Rosenstein and Shafer [WRS90].

Gamma's ET++ object-oriented framework, started as an attempt to convert MacApp to the UNIX OS [GMW88]. ET++'s architecture integrated a more advanced set of GUI components such as trees and tables, as well as a collection of reusable data-structures. As UNIX did not have a standard windowing system, ET++ introduced a layer of classes that adapted the interface of ET++'s GUI primitives to those of the windowing system. ET++ included three main collections of classes: *Foundation*, *Application* and *User-Interface* classes. The Foundation classes included data-structures such as lists and dictionaries, a change propagation mechanism that later became the observer pattern, and classes for Input/Output. The application classes had similar structure to those of the MacApp framework and actions are implemented using the command pattern. Figure 3.9, illustrates a view of ET++'s architecture and examples of applications created using ET++.

ET++ consisted of 300 classes and made the development of new editors much easier. Gamma reported that programs such as drawing editors comparable to MacDraw, tree editors, spreadsheets and other graphical applications were easily developed as student projects. Ackermann used ET++ to produce a range of graphical editors such as music, 2D and 3D drawing editors [Ap93], and object-oriented design editors [Ap96]. A detailed overview of the architecture of ET++ is given in [WG95].

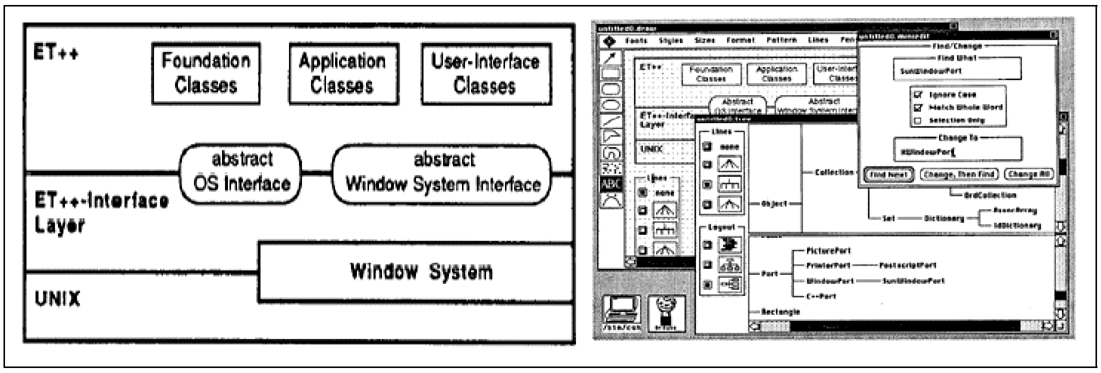


Figure 3.9: ET++'s architecture and an example of some graphical editors

Unidraw was one of the first object-oriented frameworks that aimed to solve the problem of developing domain specific graphical editors in a general way [VL89]. Completed in 1990 by Vlissides as part of his thesis in general object editing, Unidraw is based on the Interviews library for structured graphics [LVC89]. Similarly to Interviews, it was implemented in C++ under the X-Window system. Consisting of 180 classes, Unidraw's architecture addresses four basic types of abstraction: *components*, *tools*, *commands* and *external representations*. Components encapsulate the appearance, the data and the behaviour of the graphical elements of a drawing or a diagram. Components typically represent domain objects and can be connected using relationships that support dataflow. Tools allow the direct manipulation of components and other interactive operations on the drawing canvas. Commands are objects that encapsulate operations performed on components. External representations perform translations between the editor's internal format and other formats.

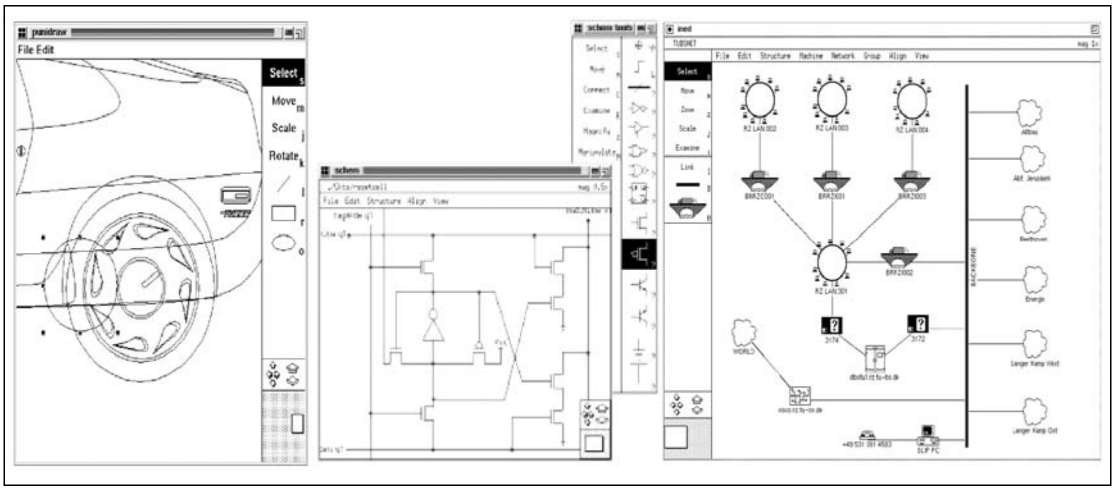


Figure 3.10: Editors created with Unidraw for drawing, circuit, and network design

To create an editor for a new diagram domain the developer has to develop specialised classes for each abstraction or simply reuse existing implementations. Vlissides demonstrated a considerable reduction in effort required to develop a new graphical editor. As an example of Unidraw's capabilities, three domain dependent graphical editors were developed: a drawing editor, a circuit editor and a GUI builder. Schonwalder and Langendorfer have used Unidraw to create INED, an independent graphical network editor [SL93]. Figure 3.10 presents a view of these editors. A detailed overview of Unidraw is given by Vlissides in [Vj90].

HotDraw was originally developed in 1987 by Beck and Cunningham [Tek87]. After Johnson's seminal paper on documenting frameworks using patterns [Jr92], Brant redesigned HotDraw as an example of a simple and educationally useful framework [Bj95]. HotDraw is implemented in Smalltalk-80. It employs the MVC pattern to separate the model of the editor from its graphical representation and control. It defines abstractions for *editors*, *drawings*, *drawing views*, *figures*, *tools*, *handles* and *constraints*. Editors are applications that hold a drawing and support application-based functions. Drawings are containers for figures drawn. A drawing has associated one or more drawing views that render the drawing to the screen. Drawings allow direct manipulation by delegating events to the current tool. Tools interpret the events and issue commands that when executed invoke operations on the figures. When a figure is selected, it exposes handles that can be directly moved to change a property of the selected figure. The interface of each abstraction is general enough to allow a diversity of extensions and customisations not only by programming but also by using special development tools.

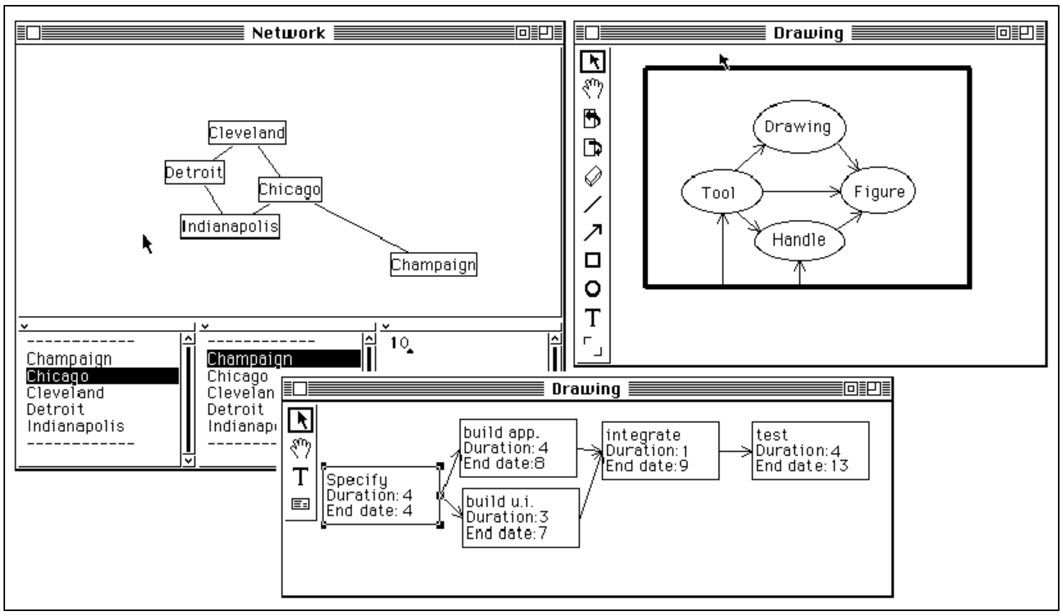


Figure 3.11: A network, drawing and pert editor based on HotDraw

Brant and Johnson reported on a development tool for HotDraw in which the user composes the various elements of a manipulation and its effects to produce a new tool [BJ94]. This could be a tool for creating a new figure, a special purpose select tool, a zoom tool or any other tool offering interaction on the drawing canvas. Examples of developed diagram editors in HotDraw include a PERT chart editor, a drawing editor, a network editor and a class inspection tool. Figure 3.11 illustrates the first three editors as reproduced from Johnson [Jr92].

A port of HotDraw to Java was implemented by Gamma in 1997 [BG97]. JHotDraw defines similar abstractions to HotDraw: *drawings*, *drawing views*, *tools*, *figures*, *handles* and *commands*. Each abstraction in JHotDraw is defined as an interface. All the interfaces together are contained in a package that is independent of other packages. This separates the design of the framework from specific implementations and hence demonstrates reuse of design. JHotDraw contains a range of concrete implementations for primitive and composite figures, for generic creation, connection and text tools. It also defines an abstract editor so that it uses implementations for running both as an application and as an applet. However, JHotDraw is missing features that are necessary for diagram editors, such as multilevel undo, support for advanced graphical features, zooming, and format translations. Examples of graphical editors developed with JHotDraw include editors for drawing, net diagrams, and pert diagrams. Figure 3.12 illustrates the view of these editors. Kaiser describes the process of developing a UML diagram editor in [Kw01].

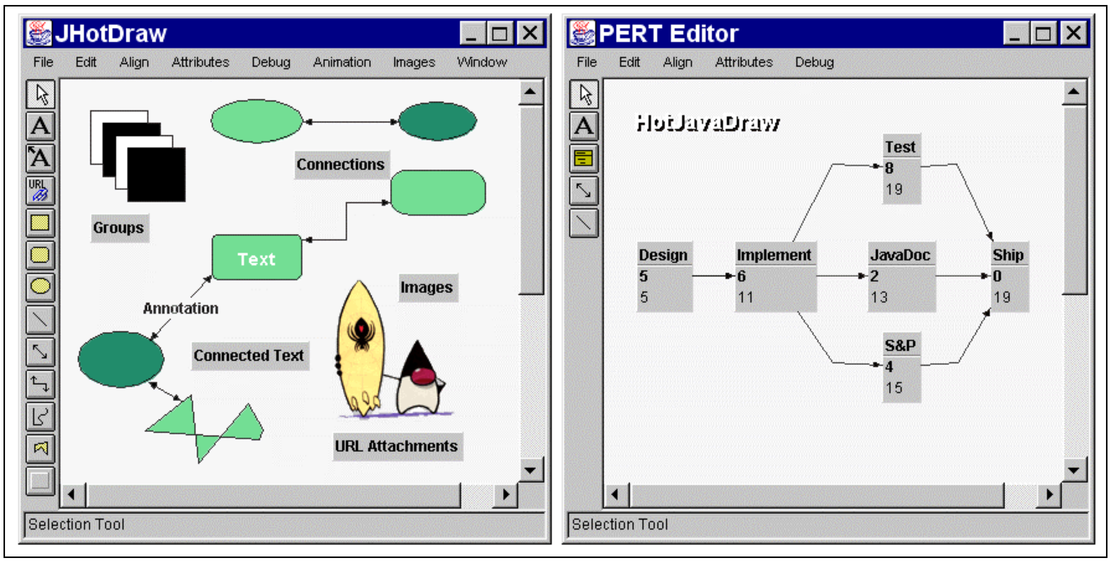


Figure 3.12: A drawing editor, and a pert diagram editor implemented with JHotDraw

3.2.2.3 Diagram Editor Generators

DiaGen as described by Minas is a diagram editor generator, developed in C++, for the X-Window system [Vg95]. DiaGen focuses on ensuring that the diagram constructed by the user will always have the correct structure. Consisting of a library of graphical components and of a diagram editor generator, DiaGen needs a specification of a hypergraph grammar to generate a new editor. A hypergraph according to Minas is *"a generalisation of a graph in which edges are hyper-edges; i.e. they can be connected to any fixed number of nodes"* [MV95]. The specification includes descriptions for the diagram's structure, the appearance of the elements, the layout mechanisms for the positioning of the elements, and the rules for behaviour specific to the editing of the diagram. It is expressed in:

- Context free hypergraph grammar that specifies the diagram syntax and the editing rules
- Dialog rules that define the interaction with the user
- Transformational rules that describe how to add and remove elements

DiaGen's original implementation demonstrates three diagram editors for Nassi-Shneiderman diagrams (NSD's), flowcharts and state machines. No references have been found reporting on the use of DiaGen from third parties.

Generators have also been used to create editors for custom visual languages. As diagrams can be considered visual languages, such generators can also be used to generate certain types of diagram editors. Haarslev and Wessel documented, with GenEd, an approach in which algebraic specifications are used to generate an editor for a visual language [HW96]. Implemented in CLOS, GenEd incorporates examples of editors for petri-nets, entity relationship diagrams, and geographical information systems.

3.2.4 Summary

Diagram editors use the direct manipulation paradigm to allow users to create and edit the diagram. As domain dependent graphical editors are complex to develop, various ideas address the problem of building new diagram editors. Existing approaches can be divided into multi-domain diagram editors, frameworks and generators. Multi domain diagram editors support a generic model of interaction and allow the customisation of diagram elements, connectivity semantics, and behaviour specific to the editing of the diagram. Frameworks for diagram editors encapsulate the design of an abstract diagram editor and allow a programmer to derive a new editor by providing specialisations and extensions. A lineage of frameworks starting with MacApp has demonstrated an evolution of an architecture based on design patterns. Diagram generators can create diagram editors automatically without requiring the use of programming at any stage of the generation. However, the process of deriving a new diagram editor necessitates writing the specification in a customised language that is specific to the generator.

3.3 Summary

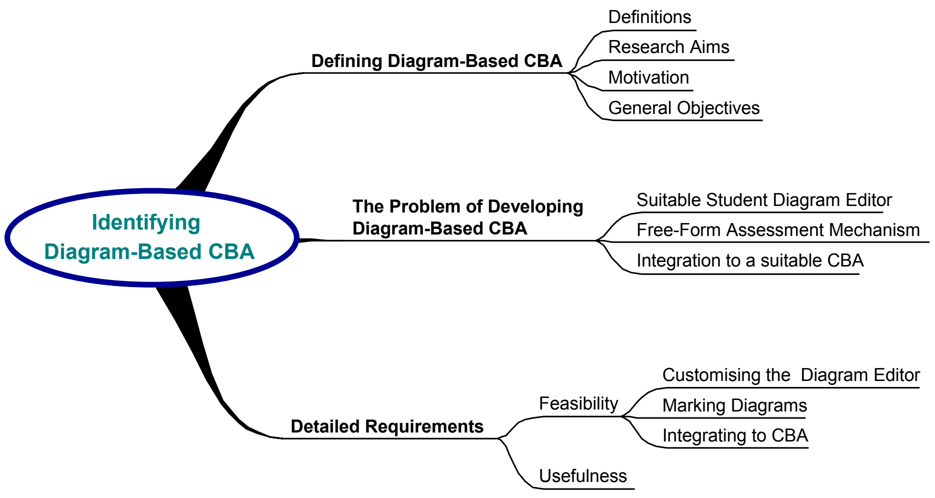
This chapter highlighted the key elements of the existing work upon which this research is based. Section 3.1 presented the Ceilidh CBA system, described the fundamental principles and illustrated its architecture. The concepts of marking action, marking tools and oracles have been introduced and explained. Based upon these concepts, Ceilidh built implementations to support the assessment of programming courses. Marking tools for the assessment of imperative programming languages are reviewed. Ceilidh's support for administration is examined. Evaluations of experience using Ceilidh demonstrate the gains in practical and pedagogic benefits.

Section 3.2 introduced graphical diagram editors and examined the problem of their development. It categorised approaches to solve the problem of creating a new diagram editor into multi domain editors, frameworks for graphical editors and diagram editor generators.

Based on Ceilidh and on the existing approaches for the development of diagram editors, chapter 4 asserts that a technique for general diagram-based automatic assessment is feasible by composing ideas from the marking of programming in Ceilidh and from diagram editor development approaches.

Chapter 4,

Identifying Diagram-Based CBA


Identifying Diagram-Based CBA
Defining Diagram-Based CBA
Definitions
Research Aims
Motivation
General Objectives
The Problem of Developing Diagram-Based CBA
Suitable Student Diagram Editor
Free-Form Assessment Mechanism
Integration to a suitable CBA
Detailed Requirements
Feasibility
Customising the Diagram Editor
Marking Diagrams
Integrating to CBA
Usefulness

"Having precise ideas often leads to a man doing nothing",

Paul Valery, 1871-1945

Introduction

The development of an authoring system for creating diagram based CBA is feasible and useful. This chapter defines diagram-based CBA, discusses the problems involved in developing it, and discuss why overcoming these problems is worthwhile. An authoring environment is proposed as a medium to ease the complexity of developing diagram-based CBA. Detailed objectives are set in relation to the fields of CBA, diagramming, and software engineering and to the phases of design, implementation and evaluation.

Section 4.1 offers a definition for diagram-based CBA and discusses the questions that this research aims to answer. Section 4.2 expounds on the problem of developing diagram-based CBA, and identifies three key problem areas that prior to this work have not yet been approached. Section 4.3 puts forward the idea of an authoring environment and proposes to integrate it to a CBA system that supplies the necessary features for assessment. The three problem areas are further analysed taking into consideration concepts presented in chapters 2 and 3.

The first problem involves the customisation of the student diagram editor to the requirements of the CBA exercise. The second problem concerns the design of a general and expressive mechanism to support the automated marking mechanism. The third problem involves the integration of the designs to a CBA that supports the full lifecycle of coursework for summative assessment in a controlled environment. The requirements for the feasibility and usefulness of the proposals to each of the problem areas are examined with respect to satisfying the general objectives.

The designs for the authoring system and the software components related to CBA are documented in the next chapter. Both take advantage of theory and practices in object-oriented frameworks and design patterns to answer to questions identified in section 4.2. The resultant architecture is founded upon software engineering research presented in sections 2.3 and 3.2.

4.1 Defining Diagram-Based CBA

Section 4.1, introduces the subject of diagram-based CBA by defining the key concepts and asking the key questions. Based on these questions, general aims are set and motivation presented.

4.1.1 Definitions

The distinction between CBA and CAA is that in CBA, the student solution is entered on-line, using a suitable environment.

Consequently, diagram based-CBA could be defined as:

> *"a type of CBA in which the solution to a required problem or the answer to a required question is drawn by the student using a suitable graphical environment and is assessed automatically according to appropriate marking guidelines"*

Diagram based-CBA can be used for formative, diagnostic or summative assessment. This research considers only summative assessment, for four reasons. Firstly, practical benefits are gained only through replacing part of the assessment process with CBA, and thus cutting the costs of manual labour. Secondly, in academic institutions, students are motivated to use CBA only when the attributed marks are part of the final course mark. Thirdly, challenges related to security, performance, and administration are present only in CBA for summative purposes. Finally, CBA built for summative purposes could accommodate formative and diagnostic assessment by simply not taking into account the marks.

A controlled environment refers to a setting where summative assessment can take place and where the security and integrity of the marking is a major concern. The stages of authoring, marking, and managing a CBA exercise are referred as the full lifecycle of a CBA exercise. The authoring stage conceptualises the exercise, develops all the required material and evaluates its validity with further testing. As soon as the required material has been completed, the exercise can be deployed and used by the students. The marking stage encompasses the details specific to the automatic assessment of the exercise and the creation of helpful feedback. The managing stage includes all the tasks involved in running and administering a CBA exercise, within an academic term. These tasks include monitoring student related information, changing course properties and customising features related to the execution and deployment of the CBA exercise.

4.1.2 Aims and Motivation

Work on diagram-based CBA is sparse. This section discusses some of the most initial and fundamental questions that originate to the fields of CBA, diagramming and software engineering.

As pointed out in section 2.1, educators increasingly acknowledge the practical and pedagogic benefits that are gained by using CBA. Can these benefits be gained using diagram-based CBA? An answer to this question serves as the principal aim of this work, an important contribution to the field of CBA, and a useful direction, as distance learning increasingly becomes reality. The practical benefits should be very similar to those gained by using CBA in general. Assessment can be performed on a large scale, in a cost effective way, and in accordance with Browns elements of quality for assessment. The pedagogic benefits should also be similar to those acquired by using free response assessment. In free response assessment there is more freedom in designing material that assesses student work in each of Bloom's six cognitive levels of learning.

Determining the kind of diagrams for which it is feasible and useful to have free response CBA and implementing a generic facility that supports them, results in the interest of this work in diagramming and software engineering.

The diagramming questions are mostly related to the structure of diagrams in relation to diagram editors and to CBA marking systems. What are the criteria that constitute a suitable type of diagram for automatic assessment and how can the marking criteria be modelled? The detection of both commonality and variation between diagram types can not only result in finding such criteria, but also present a generic solution to the problem of developing diagram-based CBA.

From a software engineering perspective, it is imperative to stipulate the qualities that the resultant software must have to comply with the functional requirements in a way that is satisfying to its users, allows future development and is amenable to change.

Change is an important property of the implementation that facilitates experimentation and development of diagram based CBA in a wide range of fields that use diagrams.

The approach to find the answers to the questions posed is pragmatic and practical. A key objective of the research is to investigate, propose, design and evaluate techniques to allow the development of automatically assessed diagram-based coursework that can be used in the context of academic courses. The main objective of the dissertation is to demonstrate that the proposal of designing an authoring environment that facilitates the full lifecycle of diagram-based CBA is both feasible and useful.

4.1.3 Summary

This section has defined diagram-based CBA and has posed the questions that need to be answered before establishing the practice of diagram-based CBA. Specifically, the questions address how CBA can benefit from diagram-based CBA.

4.2 The Problem of Developing Diagram-Based CBA

CBA systems, in a conceptual form, consist of a database, software components for authoring, marking, administering, presenting, and submitting coursework, and analogous UI interfaces. Figure 4.1 illustrates a view of the parts of a typical CBA system.

The database can be seen as the place where all the information is stored. Three basic types of data exist in CBA systems:

- Authored material such as coursework and marking related data
- Class related material such as student submissions, marks, and user information
- Properties for administration and marking

The software components are the modules of software implementation that support the requests from the user-interfaces to the database. Typically, these modules contain logic that describes the execution and constraints of the processes involved in CBA. The user-interfaces provide the means to use the system and support various options.

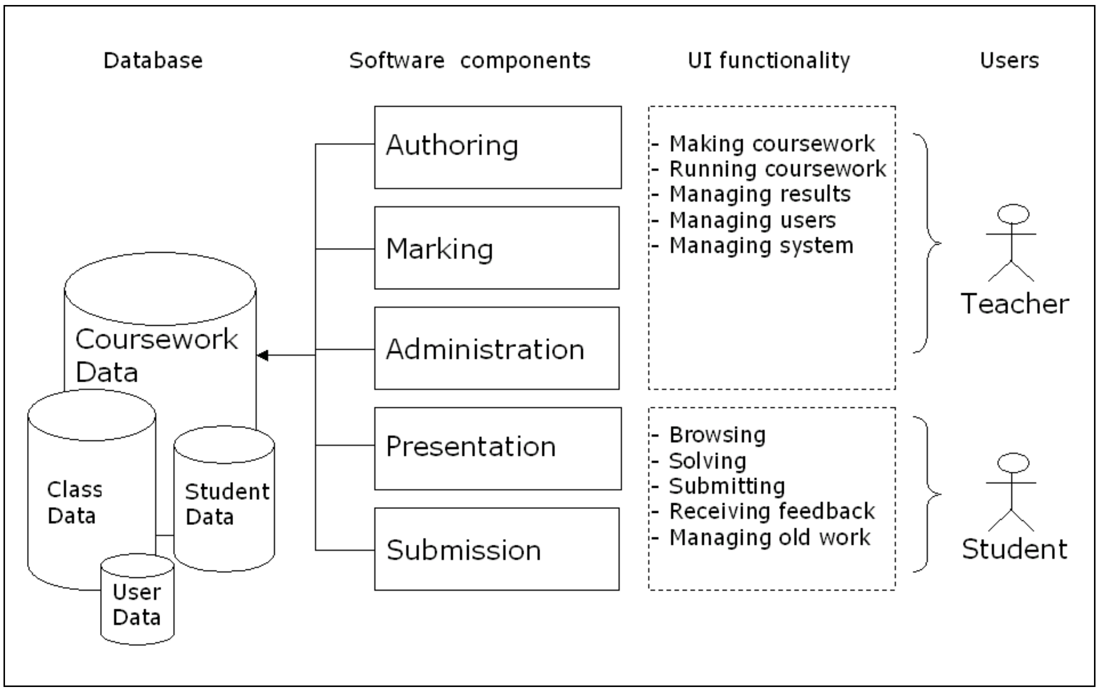


Figure 4.1: A high level view of the parts of typical CBA systems

The variation between types of assessment does not affect the database part, the software components for setting, submitting and administering coursework as well as many of the features of the UIs. All these parts can be implemented in a generic way, without having to know about the specifics of an assessment type.

In three areas, diagram-based coursework exhibits differences. These are on:

- The UI part that allows the entry and editing of the solution to the exercise by the student
- The marking mechanism that directs the marking of the student solution and returns appropriate feedback
- The process of authoring coursework and the refinement of the generic CBA model to support the full lifecycle of the CBA coursework for summative purposes in a controlled environment

The next three subsections analyse each problem area while referring to literature presented in the previous two chapters and discussing the options for exploiting research ideas and suggesting general objectives for each of the problem areas.

4.2.1. The Student Diagram Editor

In CBA, students are presented with a UI that allows the entering of the coursework's solution. This student environment varies according to the type of exercise. Figure 4.2 illustrates the types of student UI environments used by the CBA systems that have been reviewed in chapter 2.1. CBA systems for MCQs, such as WebMCQ [DG99] and QuestionMark [Qm01], Quizit [TBF97], and Ceilidh [FHG96], use web pages, GUI and text based applications respectively. Similar interfaces have been used in systems for the text and numeric type of CBA assessment, such as Ceilidh [BBF+93], TRIADS [Md99], and Examine [EX95]. Hotspot graphics exercises use multimedia authoring environments such as Authorware and HTML image-maps. CBA systems for essay marking such as Ceilidh and the ETS system [BKW+98] can use any standard word processor that can save documents in a predetermined format. In CBA systems for programming, such as Ceilidh, BOSS [JL98] and ASSYST [JU97], the environment is simply a text editor. Although an Integrated Developers Environment (IDE) could replace the text editor, this has not been chosen by any of the reviewed systems for two reasons. Firstly, learning to operate the IDE might present students with an extra level of cognitive complexity. Secondly, in large classes getting licences or even installing the IDE on all the computers for students is impractical. Specific diagram-based CBA exercises such as the ones described by Hoggarth and Lockyer in [HL98], use external graphical CASE tools.

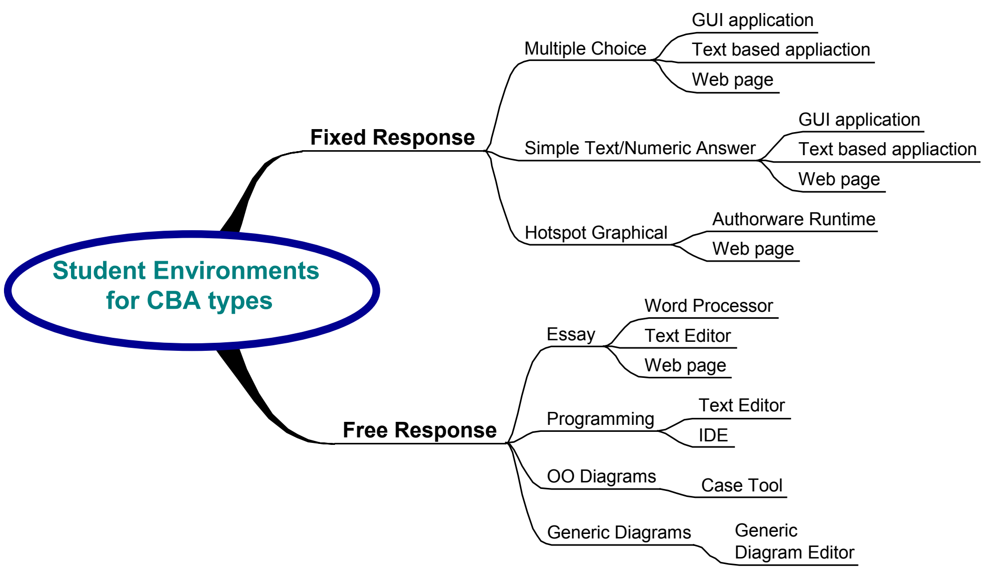


Figure 4.2: Types of student environments for CBA types of assessment

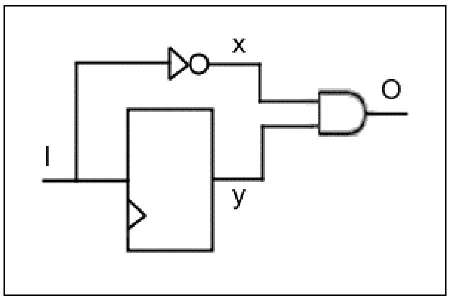
What kind of student editor would be suitable for diagram-based CBA? Two main choices exist:

- A standard UI that allows the editing of any solution
- A customised UI that has been tailored to the domain of the exercise

Selecting a standard UI for the student environment has the advantage of simplifying the authoring of a single exercise. The standard UI can be invoked on every exercise. A standard UI could be represented by using a:

- Text editor
- Specialised diagram editor
- Multiple purpose diagram editor

The choice of using a text editor means that the diagrams have to be described by the students in a text-based language. A pedagogic disadvantage of taking this approach is introducing unfamiliar and unwanted language that is far removed from the actual exercise solution – the diagram. As a simple example of this problem, Figure 4.3 reproduces from Fisler [Fk96] a simple pulser circuit diagram with one of its possible textual descriptions. The example demonstrates clearly the educational value of using a graphical view to depict a concept. The circuit's diagrammatic representation is easier to draw, understand and recognize. The textual description is harder to understand and implies creating a space-based mental picture, locating the components and their relationships.



$SP(I, O) \equiv \exists x \exists y : delay(I, y) \wedge not(I, x) \wedge and(y, x, O)$

$delay(I, O) \equiv \forall t : O(t + 1) = I(t)$

$not(I, O) \equiv \forall t : O(t) = 1 - I(t)$

$and(I1, I2, O) \equiv \forall t : O(t) = I1(t) \times I2(t)$

Figure 4.3: Both representations describe a configuration of the same single pulser

The decision of using a specialised domain graphical editor or a multiple-purpose diagram editor leads to important disadvantages. Firstly, as most specialised editors have been developed for meeting publishing and printing needs, they do not support the exporting of the structure of the diagram in a common way. Even if there is a way to export an understandable and full representation of the diagram in text, dependencies on external formats and tools defined by third parties can cause the breaking of the authored exercises. Subsequent releases of the external software can become increasingly unsynchronised with the CBA software. Secondly, external editors contain peripheral functionality that introduces learning hurdles for the students. The time learning the CBA tool has to be minimised so that students can focus upon the problem. Thirdly, very limited customisation restrains both the range and the types of potential exercises. Only diagram types that are explicitly supported can be assessed, experimentation is hard and depends on undocumented features and formats. The range of exercises is dependent on the range of diagrams that the tool supports. Finally, diagram editors do not offer any extension mechanisms for allowing the interaction and integration with a courseware system because they have not been designed to do so.

These disadvantages are minimised by choosing a solution that sees the customisation of the diagram editor to the exercise's domain. If the student diagram editor is a part of the CBA system, then the potential for satisfying the exercise requirements is much greater.

The problem of creating a diagram editor for a specific notation has been approached with multi-domain graphical editors, object-oriented frameworks and diagram editor generators. Could any of these solutions be reused within a CBA context? The criteria for reuse dictate simplicity for the creation of a new diagram editor and an ability to integrate with CBA in a way that can be used realistically in academia. Commercial systems such as Flowcharter and Visio, old systems such as Sketchpad and Thinglab, and frameworks in languages and operating systems that are obsolete cannot be directly used. These systems rather than being reused offer concrete ideas concerning aspects of the process of customising a diagram editor which are described in the following chapter.

Hekmatpour's Templa and Graphica system [Hs90], is the only multi-domain graphical editor that could be reused in an assessment context. In Templa and Graphica the configuration of a new diagram editor is given graphically, by selecting and defining the elements of a notation. However, because Templa and Graphica is an old system, many of the features of current GUI's are missing. Additionally, it was developed for Apple computers, thereby considerably limiting the number of installation bases.

Existing frameworks could be reused but only indirectly. Object-oriented frameworks for diagram editors such as Unidraw, Hotdraw and JHotDraw, ease the development of new diagram editors by providing abstractions common to all diagram editors. These frameworks are intended to be useful to programmers, who once they understand the framework's abstractions can develop new instances of the framework by specifying the variability between the modelled domain and the framework. However, such frameworks are complex and would be clearly overwhelming to educators interested in making diagram-based CBA. The complexity of such architectures is a consequence of supplying features that are not necessary or even useful in the context of assessment. Such features include automatic layout, routing and correction, beautification, constraints matching and externalisation of the graphical structure to file formats.

Diagram generators such as Mina's DiaGen could not be reused for similar reasons. Viehstaedt reports 1900 lines of code are needed to create a new editor for NSD's [Vg95]. Problems with diagram editor generators are that they still need to expose a great deal of the mechanics of the editor to the developer. The mapping between the language of the diagram editor description and the final diagram editor cannot be conveyed to non-programming users in a simple and rapid manner.

The aim in this area is to offer the ability to customise the diagram editor to the needs of the exercise by educators from non-programming backgrounds.

4.2.2. The Marking of Student Diagrams

This problem area involves the mechanism for marking the student diagram and returning feedback and results to students. What type of graphically solved exercises can be automatically assessed while ensuring consistent and helpful feedback to the student? How can the marking criteria for those domains be modelled?

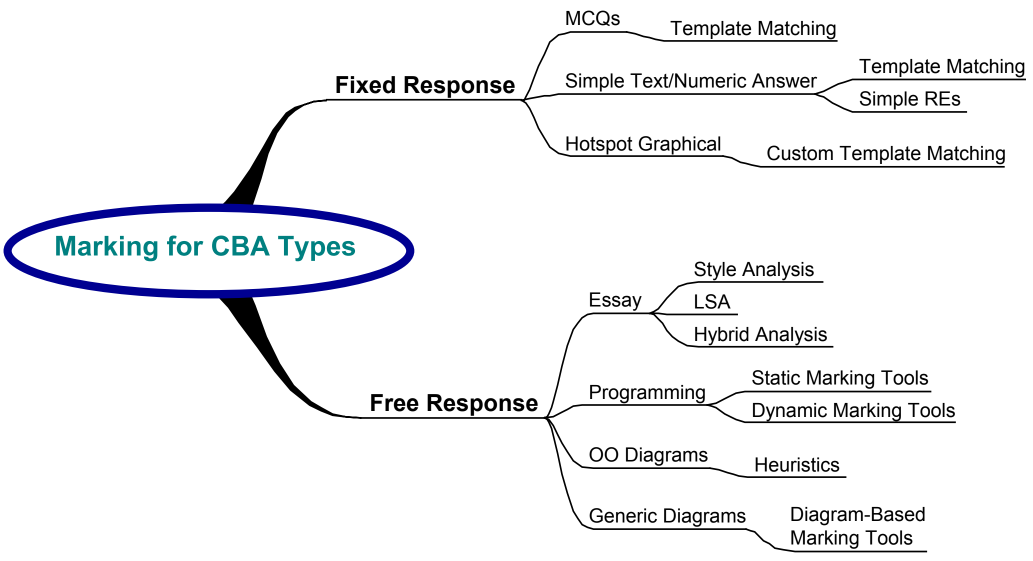


Figure 4.4: Types of marking mechanisms for CBA types of assessment

Mechanisms for marking fixed response assessment present no difficulties in their design and implementation. Simple matching algorithms can be used that compare the student solution to the exercise solution. Systems such as Quizit and MarkIt, use the notion of a solution template that contains answers to the quizzes. Systems for simple numeric/text answer use similar ways to describe simple variations in the answer. In Ceilidh, regular expressions are used to describe more than one correct answer. In graphics hotspot CBA and in systems such as TRIADS, the marking mechanism is based on template solutions that are customised. Simple comparisons are used in all cases for the matching.

Free response assessment requires more elaborate modelling for marking. This is because rather than comparing the student solution to the template solution, a free response assessment system has to evaluate various qualities that all together contribute to the final marking. Developing marking tools for evaluating criteria on per domain and per exercise bases is the approach taken by systems such as Ceilidh, BOSS, AssistIt, and RoboProof. Figure 4.4, illustrates the types of marking used within the CBA systems that have been reviewed in chapter 2.

A diagram-based exercise could potentially have an infinite number of correct solutions. As in all free response assessment, rather than comparing specific strings the marking mechanism has to know how to detect and evaluate specific aspects of the quality of the student diagram. Diagrams that have an explicit structure, such as circuits, flowcharts and graphs, could be marked in a similar way to the assessment of programming coursework. Dynamic and static metric tools could be written to assess the dynamic and static correctness of the flowchart, circuit or graph, using techniques used in programming. For diagrams that do not seem to have such explicit structure, techniques used in the assessment of essays or even new methods might be more appropriate. As Ceilidh's marking tools have been useful for CBA in many programming languages, diagram marking tools could have the same effect in diagram-based coursework.

The process of assessing a student diagram depends greatly on the diagram notation and on the underlying concepts that the notation represents. For example, what needs to be measured on a circuit design exercise is inherently different to a project management exercise that uses pert charts. The marking criteria for the circuit would involve simulation of the logic gates, whereas the ones for the pert chart exercise would involve a walkthrough of the graph and a check of the dependencies in terms of projects, constraints and resources.

Devising a prototype mechanism that allows experimentation and creation of novel automatically assessable and across domains diagram CBA is an important deliverable. By using this, metrics research for the evaluation of diagram-based coursework could be realistically tested in the context of the classroom.

4.2.3 Integrating Diagram-Based Assessment into CBA

Diagram-based CBA can be used in education if the full lifecycle of CBA exercises is supported. The full lifecycle of diagram-based CBA, similarly to that of other types of CBA, includes the authoring, marking, running and administration of the diagram-based exercise. Conceptually, many of the CBA parts between diagram-based CBA and other types of CBA are common and even reusable. The problem is to find what can be reused and what cannot, under the constraints imposed by the context of the student diagram editor and of the diagram marking.

Ceilidh has been the only system in literature to cater for:

- The full lifecycle of free response CBA
- The development of CBA in new domains

The important issue in this problem area is whether it is possible to integrate the designs for the problem areas of the diagram editor and the diagram marking within Ceilidh.

At a first glance, it might be reasonable to speculate that most of the parts of Ceilidh with the exception of the student environment and the marking mechanism can be reused. However, the integration of a diagram-based type of assessment into Ceilidh is not straightforward for three reasons:

- As the student diagram editor and the marking system have dependencies on other parts of the Ceilidh system, the changes on the student environment and the marking system propagates to Ceilidh's other subsystems.
- Highlighted performance, scalability and extensibility problems in Ceilidh could constraint the usefulness of diagram-based CBA.
- Architectural limitations in Ceilidh imposed by its three-layer model reduce its maintainability and extensibility to a degree that would significantly decrease the feasibility of the diagram-based coursework. These limitations could be rectified by using simple object-oriented reuse techniques.

An important reason to restructure Ceilidh is to provide support for more students and installation-bases and to satisfy a wider range of requests in terms of modifications and updates. The main concerns motivating this extensive change are to resolve Ceilidh's architectural limitations, to improve maintainability, allow functional extensions and diverse configurations and increase usability for all its users. An additional part of this problem area is also to determine whether limitations exist on the administration of a course in Ceilidh that could be improved for the assessment of diagram-based exercises.

The objective of this problem area is to integrate the functions of typical CBA with diagram-based assessment. Only with a pragmatic and complete system for authoring diagram-based CBA can the general questions posed by this work be answered. An integrated diagram-based CBA system would be useful practically and pedagogically, and would be fundamental to further experimentation and research.

4.2.4 Summary

This section raised three main problem areas. It presented the key issues in each area along with the motivation for solving them in a way that prior to this work has not yet been attempted. The general objective to investigate, propose, design and evaluate techniques to allow the building of automatically assessed diagram-based coursework has been broken down to general objectives for each one of the problem areas.

4.3 Detailed Requirements

To the problem of developing diagram-based CBA, this work proposes an authoring environment as a feasible and useful solution. The general form of solving the problem can be broken down to accomplishing the general aims of each of three problem areas. Those are:

- Addressing the authoring of the student diagram editor as part of authoring the CBA exercise
- Designing a generic marking mechanism that can be customised to mark a range of diagrams

- Integrating the two solutions in a CBA system that facilitates realistic experimentation and evaluation of diagram-based CBA for summative purposes

An authoring environment should allow users to experiment with the development of diagram-based CBA in a simple but also effective manner. It should support a wide range of customisations and extensions for both the student diagram editor and the marking scheme to suit potential types of diagram-based exercises. It has to be user-friendly for both authors and students, and to be designed according to methods that would most likely increase software quality and system longevity.

An authoring environment for diagram-based CBA that fits these requirements would not only be essential in answering the basic question about the potential practical and pedagogic benefits but also promote further experimentation for modelling assessment criteria in diagram-based domains.

The key issues in building such an environment are its feasibility and usefulness. Both are now examined in detail for each of the problem areas and specific requirements are set.

4.3.1 Feasibility

A plan is feasible if it can be implemented in a way that suits its requirements. The proof of feasibility for the authoring environment that solves the problem of developing diagram-based CBA is a fundamental deliverable of this work.

The idea of an authoring environment is novel, considering that the nearest attempt to a diagram-based type of CBA by Hoggarth and Lockyer in [HL98], has four major differences from the model proposed in this work. Firstly, Hoggarth and Lockyer 's system is implemented solely towards a concrete domain. Secondly, it does not explicitly promote a technique for the authoring of coursework. Thirdly, it does not allow customisations for marking and lastly it has not been built for summative assessment in a controlled environment.

Considering the question "*Can an authoring system be designed in a way that addresses the main requirements and makes diagram-based CBA useful*", the next section explains the requirements identified for each one of the problem areas, and discusses their relationship to the fields of CBA, diagramming and software engineering.

4.3.1.1 Requirements for Solving the Problem of Customising the Diagram Editor

The proposed authoring environment allows the customisation of the student diagram editor on a per exercise basis. A wide range of requirements exists for both the authoring environment and the student diagram editor. These requirements originate in the fields of CBA, diagramming and software engineering and refer to the phases of design, implementation and evaluation. Figure 4.5 illustrates a mindmap of the requirements.

As far as CBA is concerned, the customised diagram editor must be easy to use, functional, and should satisfy the editing requirements of a diagram-based exercise. Direct manipulation techniques must be employed as they allow the student to edit more naturally the diagram solution. The environment must provide facilities to customise the interaction for a particular notation or exercise. For example, in CBA for circuit design, the environment must support the creation, parameterisation and connection of components in a different way than in CBA for software design. This is because the two domains are inherently different. They describe different concepts and depict these with different visual symbols while using different composition rules. As requirements vary for different diagram notations, a common base has to be designed that takes into account the variation and allows suitable configuration.

The student diagram editor must be composed by standard GUI primitives that are common to most graphical applications and are most likely to be known to students, such as menus, toolbars, buttons, scrollbars, and so on. Various operations for the editing of the diagrams must be supported:

- Selection and translation (move, rotate, scale, stretch)
- Parametarisation (changing of attributes such as sizes, colours, labels, etc)
- Connection
- Canvas editing operations (such as zooming, scrolling, snapping to grids etc)

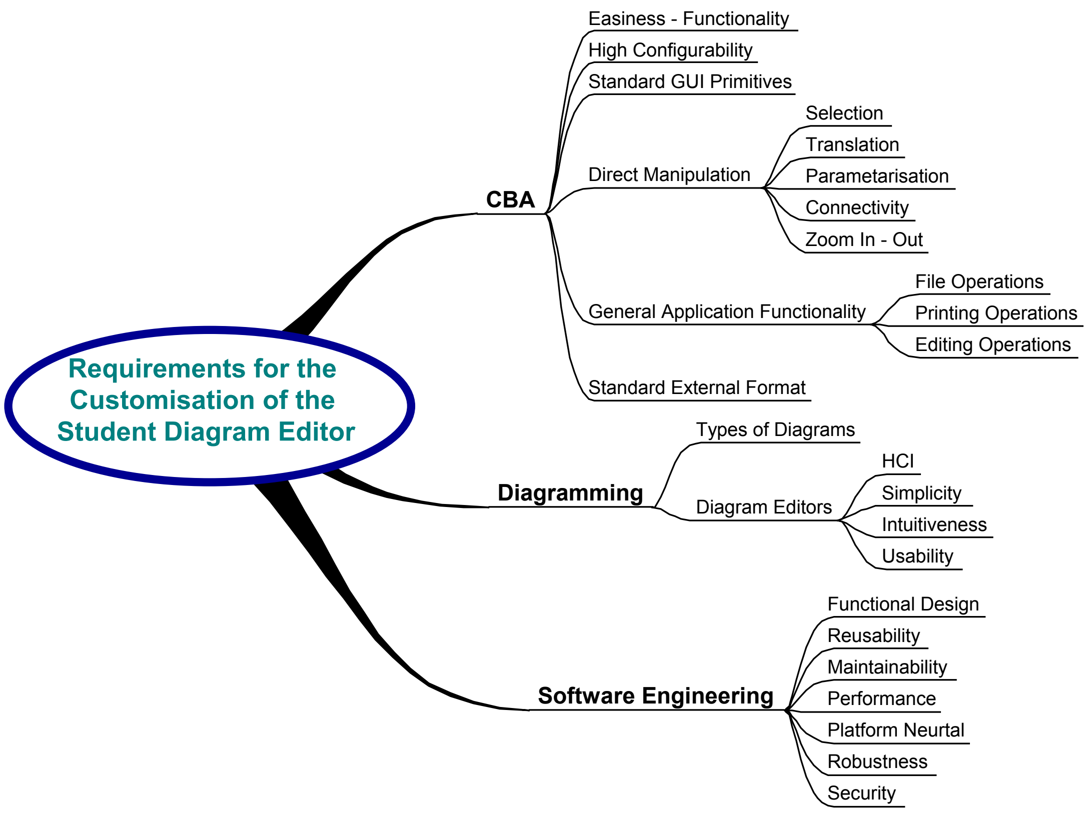


Figure 4.5: Requirements for the customisation of the diagram editor

General application functionality such as multi-level undo, file, editing and printing related operations must also be supported for usability reasons. Finally, the student diagram editor must be designed to store the diagram in a standard format that is comprehensible to the marking system, for all possible domains.

The authoring environment for customised diagram editors should comply with the requirements of the diagram editor. The authoring environment is used by authors to produce diagram editors, which are used by students to produce diagrams. It must provide features that permit the creation of student diagram editors for a number of the diagram notations surveyed in section 2.2.2. It should additionally define graphical tools for the specification of the variation amongst diagram editors and propose a technique for mapping the domain elements to diagram representations.

From a diagramming perspective, the requirements for both the student diagram editor and the authoring environment are to address useful to education diagram notations, whilst taking into consideration the standards set by the diagramming community. Standard practices in HCI have to be employed to design the interaction between the user and the diagram editor. Simplicity and intuitiveness of the diagram editing are very important to increase usability and improve the student's experience of assessment.

From a software design perspective, the authoring environment is a much more important and complex deliverable than the student diagram editor. Authored diagram editors can be seen as simply a "property" of a CBA exercise. The main requirement in software engineering for the authoring environment is to be designed in a way that makes the implementation of the functional specification feasible. Additionally, the design must result in an implementation that has reusable and maintainable parts that can sustain change. One of the most essential concerns in software over the last thirty years has been exactly this problem of unanticipated alteration. In an effort to handle change, there was a move towards abstraction, separation of design from its implementation and an increasing appreciation of the importance of software architectures. As software artefacts try to resist time decay, the realisation of the "*tyranny of a single implementation*" as Kay has accurately described, becomes progressively more obvious [Ka97]. Single implementations, even if designed to be flexible, fail to capture the essence of the problem domain in time. This is because time changes the system's original requirements in often unpredictable ways. As the structure that joins systems parts contains constraining relationships and dependencies, change tends to be hard and expensive.

Additional requirements in software engineering for both the authoring environment and the authored diagram editors are to perform well, to be platform neutral, robust, and to comply with security constraints enforced by the objective to execute diagram-based CBA in a controlled environment for formal marking. A complimentary requirement is to design the authoring environment so that it encourages further development related to other research interests. This is important because an authoring environment that requires very little authoring effort to make a new diagram editor is a deliverable much needed in many fields.

Complexity, which is not necessary for the objectives in CBA, should not be part of the system. Specifically, issues such as diagram editing constraints, efficiency in large scale diagrams, transformations to various file formats, automatic layout and routing which are often parts of diagram editors could all be avoided as they are not directly related to the CBA context. Instead, simplicity, extensibility, usability and standardisation on a generic external format are seen as much more important features.

In general, the idea of customising the diagram editor's behaviour to the properties of a domain and to those of an exercise presents three directions for evaluation:

- The extent of domain coverage for new diagram editors
- The easiness of the mapping between the representations of domain elements on the diagram and of suitable structures that can be marked
- The easiness of authoring a new type of diagram-based CBA exercise

The extent of domain coverage is the range of potential diagram editors. The authoring environment must be designed with the intention to balance domain coverage with simplicity of use. The mapping between the representations of diagrams and of a standard format is very important. It is necessary for the marking system to be able to understand fully the student diagram. The easiness of authoring a new type of a diagram-based exercise promotes experimentation and further evaluation.

4.3.1.2 Requirements for the Marking of the Diagrams

Following the proposal to solve the problem of automatically marking diagrams using similar techniques to the marking of programming, results in requirements that relate to the fields of CBA, diagramming and software engineering. Figure 4.6 illustrates a view of these requirements.

From a CBA perspective, a marking mechanism has to be designed that will be expressive and extensible for authors that want to create a new type of diagram-based exercise. Additionally, the marking system has to implement a mechanism to return detailed feedback. Consideration has to be taken of the generic ideas that made Ceilidh's marking mechanism useful amongst academic institutions. Specifically, the aim is to introduce more flexible and generic marking than Ceilidh did.

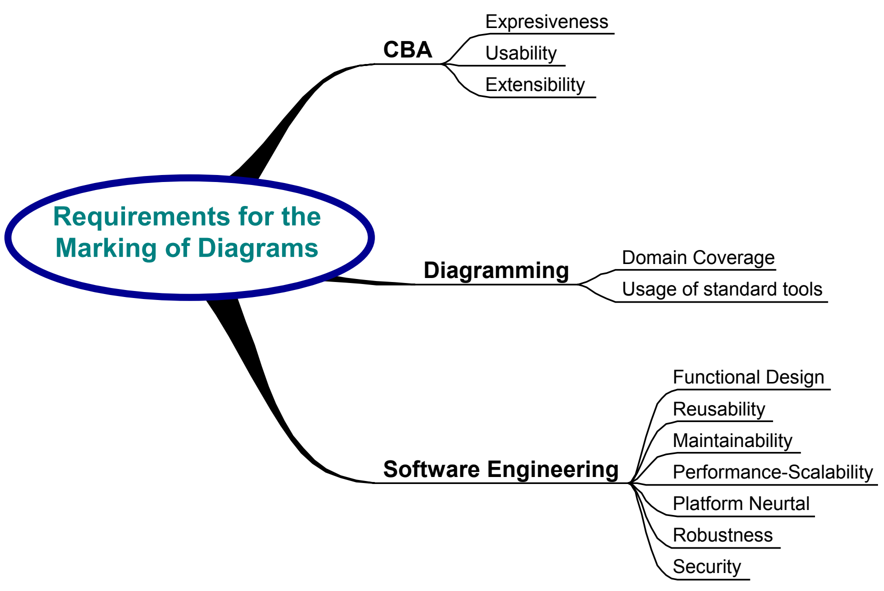


Figure 4.6: Requirements for a generic solution to the marking of diagrams

Concerning the field of diagramming, the marking mechanism must be configurable to mark a wide range of domains. To achieve this it must simplify the integration of external tools that evaluate some quality of the diagram. For example, if the type of diagram-exercise is analogue circuit design, the marking system must be able to communicate with analogue circuit simulators.

From a software engineering point of view, the requirements are similar to the ones set for the diagram editor. The marking system must be designed to support the intended functionality, be reusable and maintainable, perform well in large classes of students, and be platform neutral, robust and secure.

4.3.1.3 Requirements for Integrating Diagram-Based Assessment with CBA

Integration with a CBA system that supplies the common parts discussed in section 4.1 is necessary to allow the realistic evaluation of diagram-based assessment. Figure 4.7 depicts a view of the requirements within the fields of CBA, diagramming and software engineering.

From a CBA perspective, the full lifecycle of a CBA exercise includes the stages of authoring, running, marking, managing and evaluating the results. Each stage must be supported. Ceilidh would have been a good candidate system for the integration, because it supports the full lifecycle of free response CBA exercises. However architectural problems concerning its maintainability, performance, platform neutrality, scalability and expressiveness restrain this decision. These problems could be rectified with a redesign using standard object-oriented methods. Better maintainability, performance, platform neutrality, scalability and expressiveness together with a more robust and secure policy on the design of the assessment process are the objectives of the new design in software engineering.

As far as diagramming is concerned, the support for the full lifecycle of diagram based CBA exercises must also include a technique to categorise the various diagram editor configurations to a suitable organisation that will be useful to authors.

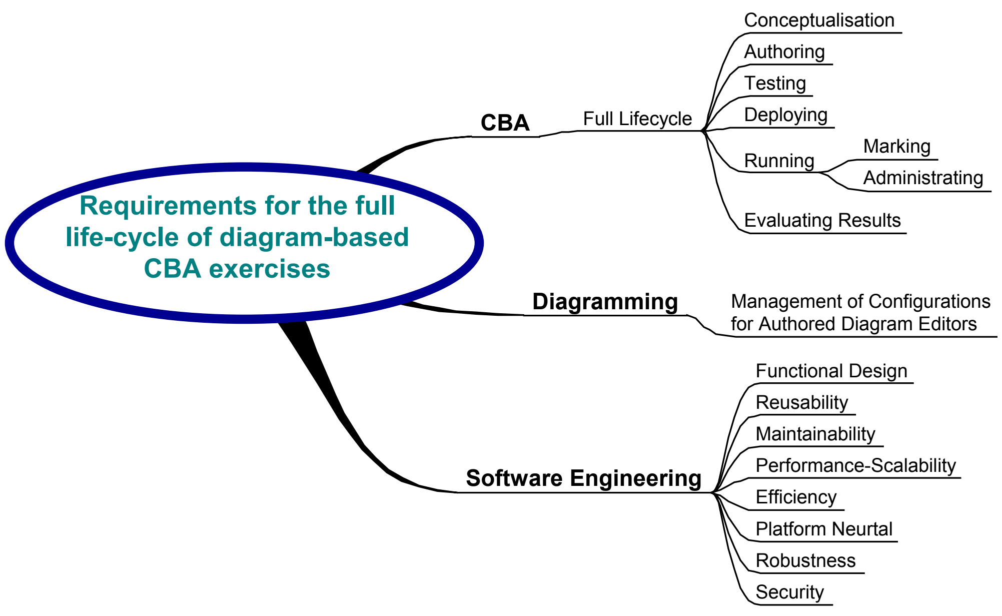


Figure 4.7: Requirements for the full lifecycle of CBA

4.3.2 Usefulness

Although this work is directly associated with the evaluation of diagram based CBA, it also presents deliverables that are useful to various parties within the three fields of CBA, diagramming and software engineering. Figure 4.8 depicts the levels of users.

For CBA, an authoring system for diagram-based assessment would benefit students, teachers, and researchers. Students and authors of CBA could cooperate to maximise both the practical and pedagogic benefits. Researchers in CBA could use the authoring environment to adapt its features to originally unpredicted diagram domains.

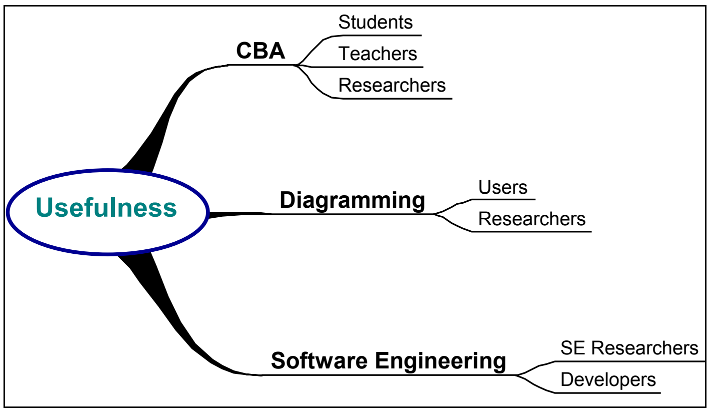


Figure 4.8: Potential for usefulness

In diagramming, such an authoring environment is very useful for traditional use (printing, typesetting, editing custom notations) as well as for research. Work related to the use of new diagram notations could employ the authoring environment to build editors for the new notations.

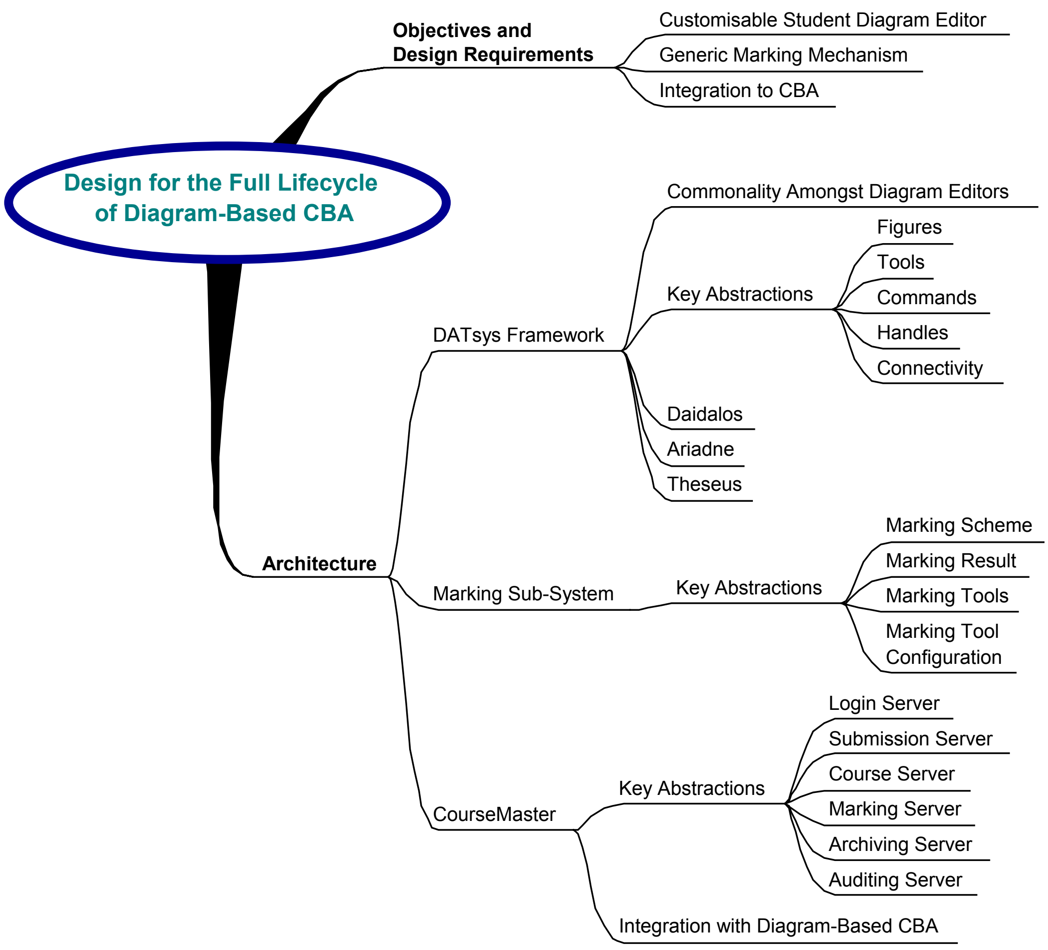
Finally, the deliverable is also useful to researchers in software engineering that perceive the architecture as a continuation of a lineage of architectures, from MacApp to JHotDraw and investigate the evolution of the design. A deliverable with high software quality is important for the continuation of the development and future work.

4.4 Summary

This chapter identified the subject of diagram based CBA and the problem areas involved in its development. Specifically, it has explained that the most important problem areas are imposed by the customisations needed for the student diagram editor and for the marking mechanism. For each problem area, a preliminary discussion that linked the issues involved with research in CBA, diagramming and software engineering argued for a proposal to design an authoring system. This proposal has been further investigated in relation to the requirements for each problem area, and in relation to its feasibility and usefulness. Detailed requirements have been set that are discussed further in the next chapter.

Chapter 5,

Design,



``It is easy to make things hard. It is hard to make things easy''

Myers Brad

Introduction

This chapter describes a solution to the problem of developing diagram-based CBA. It presents solutions to each of the three problems identified in chapter 4. The design meets the needs of the full lifecycle of diagram-based CBA.

The problem of customising the student's diagram editor to a specific exercise is approached by designing the DATsys object-oriented framework. DATsys encapsulates the design of a common base of diagram editors and defines explicit extension points for new diagram editors. The customisation of these extension points is made through graphical tools in a way that does not require programming.

The problem with marking any type of diagram can be approached by designing a generic marking mechanism that can be extended to model new marking criteria. The design offers abstractions for modelling the marking process that are independent from the type of assessment. Extensions for the generic mechanism can be developed and plugged in using Ceilidh's notion of a marking tool. In this chapter, examples of marking tools for circuit diagrams, flowcharts, and object-oriented design diagrams are discussed.

The problem of integrating the DATsys framework and the generic marking system into a CBA for summative assessment in an academic environment is approached with a redesign of the Ceilidh system using object-oriented methods. The redesign not only integrates DATsys and the marking system, but also makes considerable improvements on Ceilidh's maintainability, expressiveness, scalability, performance, security and robustness.

It is not the intention of this chapter to describe in detail every part of the design, only the most important and interesting parts are presented. The complete design documents for DATsys, the marking system and CourseMaster consists of approximately 1900 web pages and can be found in [Ta01], [TS00] and [TS99] respectively. Chapter 6 discusses implementation issues and shows how the choices made in the design support the requirements.

5.1 Design Objectives and Requirements

The main objective for the design of the proposed authoring environment is to allow research and evaluation of diagram-based CBA. Aiming for an infrastructure that can be used for research enforces design choices that tend towards sustaining and facilitating change. Aiming to evaluate diagram-based CBA presumes a functional design and implementation that can be tested in a realistic setting. It enforces design choices that consider reliability, usability, scalability, performance and portability.

The idea of making a framework for diagram editors part of the CBA, followed by the idea of creating an authoring environment to configure the frameworks' extension points, exhibit considerable benefits. The student diagram editor can match the editing requirements for the domain and the exercise. The authoring system can be used for experimentation in free response diagram-based CBA, an area of CBA that has not yet been approached.

The potential for variation within the set of possible authored diagram editors is the degree of the authoring environment's coverage. The design of the authoring environment must consider a balance between coverage and simplicity. Simplicity and usability are very important qualities that are fundamental towards the aim of making the authoring of diagram-based CBA exercises available to educators from non-programming backgrounds.

The objectives for the design of the marking subsystem are simplicity, generality, expressiveness and the potential for use in many domains. Based on the Ceilidh marking mechanism, the new design must also make improvements on usability, performance, robustness and security.

The objectives for the integration of the two solutions necessitated the redesign of the Ceilidh system into a new CBA system named CourseMaster. CourseMaster strives to maintain Ceilidh's functionality while providing integration with the authoring environment for diagram-based CBA and the generic marking mechanism. Additional requirements are to increase the scalability, performance, maintainability, extensibility, usability and platform independence of Ceilidh's distribution.

An infrastructure that enables the authoring, running, marking and administering of CBA exercises in diagram domains can be achieved by implementing the design and by distributing the various responsibilities to the users involved in the assessment process.

Consider a circuit diagram exercise:

- A course developer uses the authoring environment to build a simple circuit editor
- A teacher uses a different view of the authoring environment to build the CBA exercise in logic design by specialising the circuit editor for the exercise, describing the exercise, and configuring the marking system. It may be possible to reuse criteria defined in other types of exercises. If it is not, new criteria can be modelled and plugged-in.
- A student uses the produced circuit diagram editor via CourseMaster to solve the exercise. Upon submission, the marking system that has been configured by the teacher automatically marks the student solution and returns feedback.

5.1.1 The Student Diagram Editor

The student's editor must be usable and reflect the intentions of the exercise author. The author's environment must allow the creation of a range of diagram editors, as well as incorporating the common functions found in diagram editors. The authored editor must meet the requirements set by the CBA exercise that include the domain-dependent interactive behaviour of the diagram editor and its application-based functionality. This functionality is made available by the options available on menus, buttons, shortcuts, and other GUI primitives.

The authored editor must comply with the requirements of editing a number of diagram notations and must support a simple mechanism for interacting with the diagram. The manner in which elements are created, edited and connected in diagram editors is domain dependent. Therefore, the authoring environment must address the common and different techniques in:

- Creating diagram elements
- Selecting elements and interacting with the diagram
- Connecting diagram elements
- Editing the properties of both diagram elements and connections

Standard options such as unlimited undo-redo, constraints to the interaction, and editing actions such as cutting, copying and pasting should be available. In addition, mechanisms to customise the application-based options must be included. Examples of such options are the saving, printing, and loading of diagrams and the invocation of external tools and specialised functions.

Such an authoring environment can be build by designing and implementing an object-oriented framework that models the commonality and variation amongst diagram editors. For each new domain, the commonality can be reused, and the variation redefined. The variation can be modelled by designing appropriate parameterisation and extension points. The authoring environment can be both part of the framework and a tool that configures the framework's parameters and extension points.

According to Johnson's taxonomy of frameworks [JFS99], an important step in evaluating an object-oriented framework is to derive three application instances. Typically, the third application instance should be different enough from the other two to reveal the weaknesses of the framework.

The proposed approach sees the DATsys object-oriented framework being designed as a black box framework that contains visual tools to facilitate the composition of its available extension points. In this way, the task of building a new instance can become a very simple and understandable process. However, the freedom of making choices for building a new application instance may be constrained. This limitation can be rectified by designing a level of available extensions for programmers that evolve the framework in new directions. Furthermore, this extension-level could be made available to the user of DATsys at runtime, making the evolution of the framework a dynamic process.

5.1.2 The Generic Marking Mechanism

The generic marking mechanism must facilitate the marking of new types of exercises such as logic design, flowchart and object-oriented design. It must also increase the control over the assessment process. It should be possible to develop new types of criteria for a wide range of types of exercises, such as music composition, biology, chemistry, and so on. Additionally, flexible feedback and secure, robust and reliable running must be planned-for. To test the expressiveness of the marking system, at least three extensions have to be made in a range of diagram-based domains.

5.1.3 Integration with CBA

The integration of the authoring environment with the generic marking mechanism has to be seamless in a CBA system that provides the support for running CBA in a controlled environment for formal marking. As highlighted in the previous chapter, no system suited the criteria set, hence a new system had to be designed and implemented. CourseMaster is a redesign of Ceilidh that aims to integrate DATsys and the authoring system while improving Ceilidh's software qualities.

5.2 A High Level View of the Overall Plan

There are five types of users within Ceilidh: students, tutors, teachers, developers and administrators. The responsibilities of tutors and administrators are not affected by the type of CBA and therefore do not present any immediate interest concerning diagram based CBA. In contrast, students, teachers and developers have additional responsibilities. Students must draw the solution within an appropriate diagram editor, teachers have to author the exercise using authoring tools, and developers must create elements within new diagram domains.

Developers use the authoring system as a tool to create domain libraries for the teacher to use. Exercise developers use the authoring system to create concrete exercises. Students use diagram editors customised to the exercises. This customisation has been made by the exercise developer, while authoring the exercise. The marking of the diagram solution is immediate and can be customised in various levels of detail.

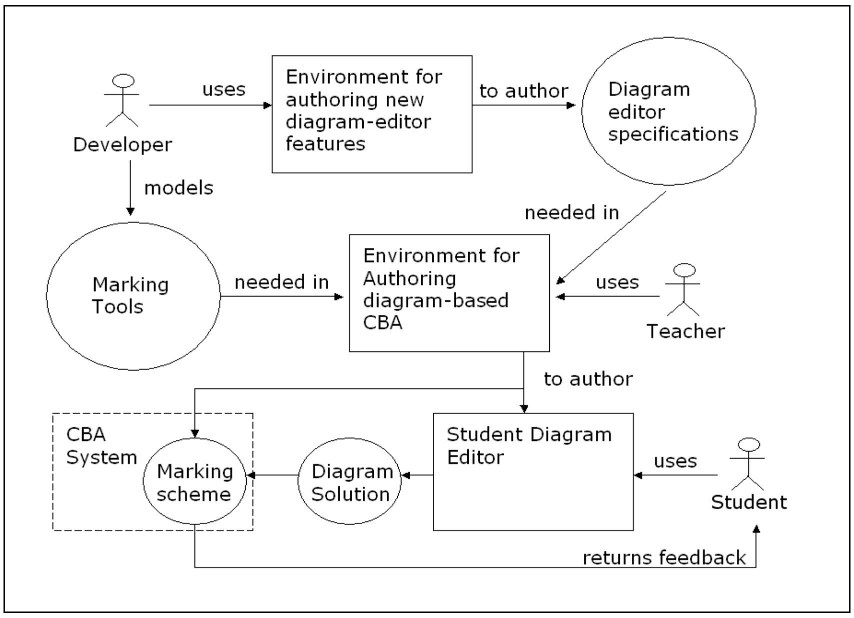


Figure 5.1: Overview of the conceptual plan for diagram-based CBA

Figure 5.1 illustrates the overall solution plan. The course developer prepares diagram editor specifications and marking tools for a specific domain. These are loaded into the teacher's environment and customised further to meet the requirements of the diagram-based CBA exercise. A newly built exercise contains the student diagram editor and marking customisations. The student uses the diagram editor to complete the exercise that is marked via a marking scheme. A marking scheme is a program that specifies the invocation and configuration of the marking tools. Finally, upon completion of the marking process, feedback is returned to the student.

5.3 The Design of the DATsys Framework

Detecting the commonality and variation between domain dependent diagram editors was key in the design of DATsys. The commonality is abstracted into a family of classes that are distributed into several modules. Frameworks define explicit extension points that allow developers to describe variation. DATsys defines extension points in such a way that changes can be made visually, using graphical tools.

The DATsys architecture has been influenced by frameworks such as Unidraw [Vj90], Hotdraw [Jr92], and JHotdraw [BG97]. In contrast to these frameworks, DATsys defines graphical tools that allow the configuration of extension points. Such tools are contained in two diagram editors, Daidalos and Ariadne. A third diagram editor, Theseus, represents the student editor.

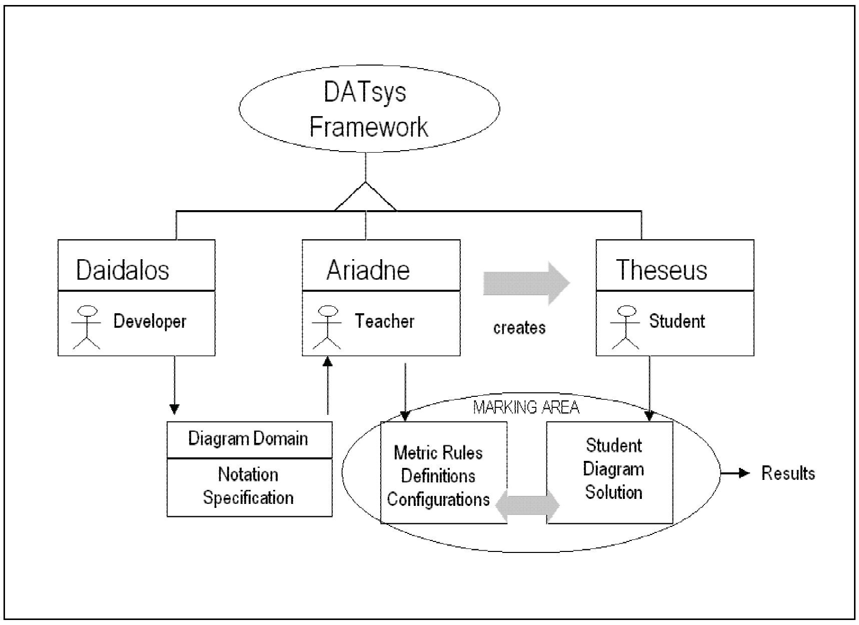


Figure 5.2: A view of how DATsys relates to the marking of diagrams

Figure 5.2 illustrates three diagram editors that are instances of the DATsys framework. Daidalos defines specifications for diagram notations as libraries. Ariadne uses these libraries to allow the authoring of diagram-based CBA exercises. The building of a diagram-based CBA exercise consists of describing how the student editor will function and how the student diagram will be marked. Both Daidalos and Ariadne are the front-end of the authoring system for diagram-based CBA, where Theseus is the customised diagram editor that is unique to the CBA exercise.

5.3.1 Commonality and Variation Amongst Diagram Editors

Diagram editors provide options to allow interaction between the user and the diagram. The interaction takes place directly on a canvas by using the mouse or other input devices. The available options are presented using simple GUI components such as menu items, buttons, toolbars, keyboard shortcuts, and so on.

Diagram notations typically define sets of elements that are considered as the tokens of the diagramming language. For example, circuit editors define gates, electrical components and wire. Flowchart editors define nodes for actions such as starting, ending, running, printing and conditioning. They also define relationships that graphically represent the direction of the flow using an arrowhead. Object-oriented tool editors use shapes and relationships to convey design notation.

The editing operations require a data structure to store the elements that are appropriate to the domain type being modelled. Depending on the functionality and architecture of the diagram editor, this structure may be a list, graph, tree or any other suitable data structure.

The interaction between the user and the diagram on the canvas is domain dependent. The editing actions for circuit diagrams are different from the ones needed for class diagrams. Diagram editors address the editing features of a diagram notation by using tools specific to the notation. Tools are selected by the user whilst editing and they interpret the user's actions on the canvas. Some editing modes may present a range of options to the user, therefore further points of interaction may be necessary. Handles ease the interaction with the diagram by allowing the changing of properties of the diagram directly.

Depending on the modelled domain, diagram editors have various commands associated with the diagram and external tools. Application components may be linked to execute any tasks. The tasks that are available to the user are presented using GUI components.

As figure 5.3 illustrates, the following concepts represent a common base amongst diagram editors:

1) **Figures**: These are drawing primitives that can be put together to create **diagram elements**. Drawing primitives include commonly used shapes such as lines, rectangles, ellipses, polygons, arcs, curves, bitmap images and text labels. Figures can be composed into groups of figures to any depth of compositional complexity.

2) **Diagram elements**: These are the tokens of a diagram notation. Diagram elements have three main parts:

    - A graphical view that consists of drawing primitives

- A data model that may represent any data

- Connectivity semantics towards the other diagram elements

The graphical view of a diagram element is composed of figures. Any part of the data model can be linked to the graphical view directly by annotating a value, or indirectly by relating the value to a graphical characteristic. The connectivity between diagram elements can be considered a diagram element itself, with its own graphical view, data model and connectivity.

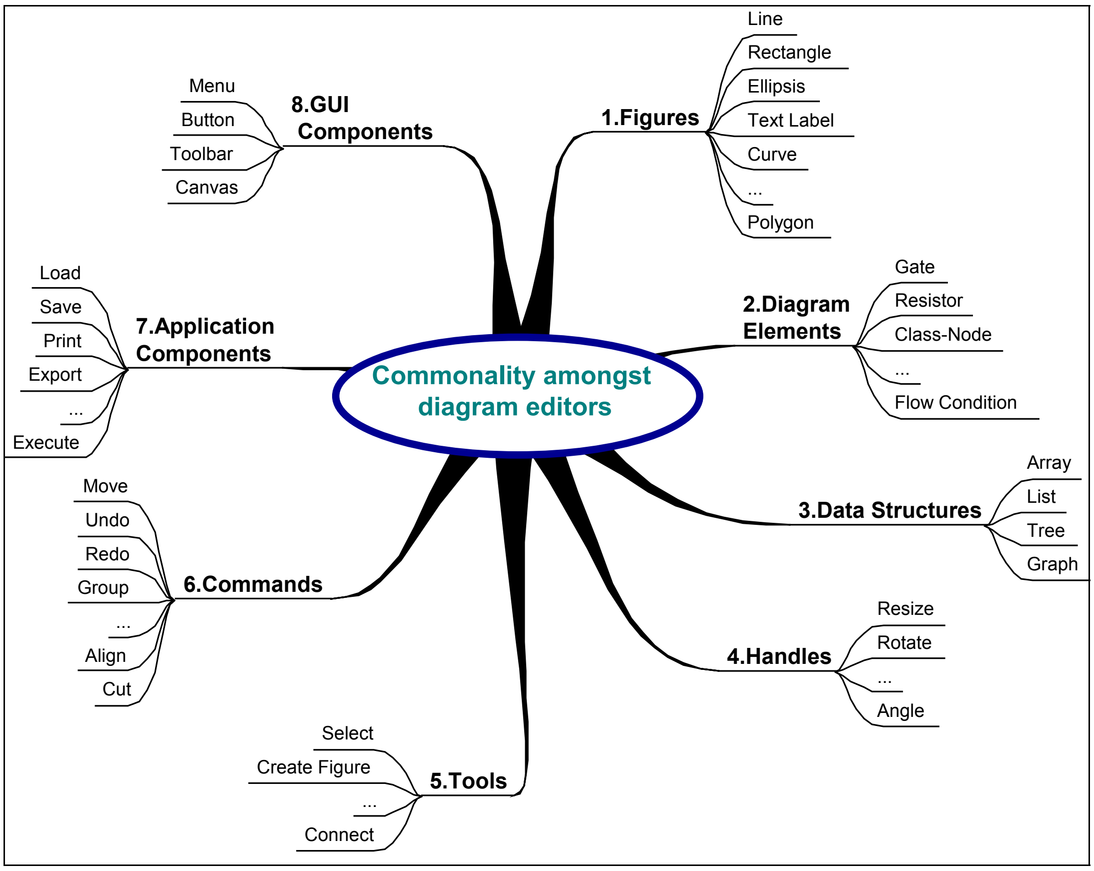


Figure 5.3: Examples of common concepts amongst diagram editors

3) **Diagram data structures:** These hold the diagram in memory with all of its diagram elements. Depending on the type of notation, editor, and performance requirements, this structure may be a list, tree, graph or any other suitable data-structure.

4) **Handles:** These allow the changing of some aspect of a figure by employing direct manipulation. Examples of handles, are the resize, rotate, and connect handles that appear in graphical applications.

5) **Tools:** These are used as ways of interpreting the interaction between the user and the diagram editor. Common tools in diagram editors implement functions such as the creation, selection and translation of diagram elements, the connection between elements, and the editing of data values. In most diagram editors, tools provide composition of interactions. For example, the select tool has options for single and multiple selection depending on the actions of the user. Often the select tool also implements the translation operation so that users do not have to change the selected tool to move the selected element.

6) **Commands:** These are operations on the diagram editor. Commands can be executed by both the user and the editor. Commands separate the GUI components from the operations. Commands describe the effects of the possible interactions as well as the effects of reversing them. The latter aids in supporting the "undo" operation, a necessary function in modern graphical editors.

7) **Application components:** These are components implementing responsibilities such as saving and loading, configuring, auditing, importing/exporting, and so on.

8) **GUI components:** These are graphical primitives used to build the user interface. Diagram editors employ interfaces composed of GUI components for selecting options and tools. A key GUI component is the drawing canvas, where all the interaction between the user and the diagram takes place. In some diagram editors, the canvas is decorated with grids, scrollbars, rulers, and allows zooming.

Examples of commonality in handles are the resize and movement handles, the select and create tools and the commands for the clipboard and for undoing and redoing previous actions. Common GUI components are the drawing canvas and the toolbar. Various other standard or custom GUI components are used depending on the features of the diagram editor. Common application-based features exist for storage management, printing, presenting help and logging exceptional cases.

Variation amongst diagram editors can be found on all the eight categories of concepts described. Figures and diagram elements vary according to the modelled domain. The type of data structure used internally depends largely on the format of the diagram elements and the needs of the various application components. For example, editors for VLSI that manage large diagrams use graph data-structures for optimising the speed of accessing the relationships between gates.

The operation of handles, tools and commands also vary, although a subset is present in most diagram editors. In addition, variation exists within the application and GUI components for the various application-specific features.

5.3.2 Key Abstractions

The main idea behind the design of DATsys is to address both the commonality and variation amongst diagram editors. Commonality can be modelled concretely and variation can be modelled through defining extension points and providing mechanisms for configuration.

Figures can be represented with a hierarchy of classes for all the primitive shapes. All figures support drawing, unlimited level grouping, connectivity, and application-based features such as saving, exporting, cloning and executing.

Diagram elements are an important extension point and are designed by the course developer. For every diagram element, the construction phase involves:

- Drawing the graphical view using primitive figures
- Specifying and associating a data model
- Describing connectivity properties

Diagram elements are compositions of figures with added responsibilities. The graphical view of a diagram element should be able to be drawn manually or via programming. In the latter case, a small program could parse any kind of data and create a graphical representation expressed in figures. The data model of a diagram element could have elements of any type that may or may not be visible in the diagram. The connectivity properties may include types, styles and constraints of connection between elements. All the characteristics of diagram elements must be named explicitly to ease the identification that is needed at the marking stage.

Another extension point is needed for the data structure that holds the diagram. The data structure can be encapsulated within a diagram object that does not allow direct access to the structure. By adapting the access interface of the diagram to a new data structure, the data structure can easily change according to the requirements of each editor.

Tools can be modelled in a hierarchy of standard and user-defined tools. After assembling the diagram elements of a domain, the course developer must create additional diagram editing tools. These are user-defines tools that supplement standard tools such as the select, connect and text.

Handles are modelled with a hierarchy of standard handles such as the resize, rotate, and radius handles. Handles can be made user-definable for diagram-elements, and can be linked to attributes of diagram-elements.

To increase the framework's longevity, GUI components could be modelled using adapter classes. Adapter classes allow the definition of a standard interface for use within the framework and for implementing parts that can be exchanged. In this case, the updated implementations can be part of new graphical APIs.

Finally, in order to vary the applications components, a mechanism for defining and linking diagram editor options is needed.

Figure 5.4 illustrates the modelled concepts. A `DiagramEditor` object encapsulates any diagram editor application. Diagram editor instances are configured with parameters for the available options, the use and attributes of GUI elements, and various application level parameters. Instances of `DiagramEditor` are associated with a `DiagramEditorView` that represents the editor's graphical view with all its GUI components. The creation of the GUI components is the responsibility of the `OptionMaker`.

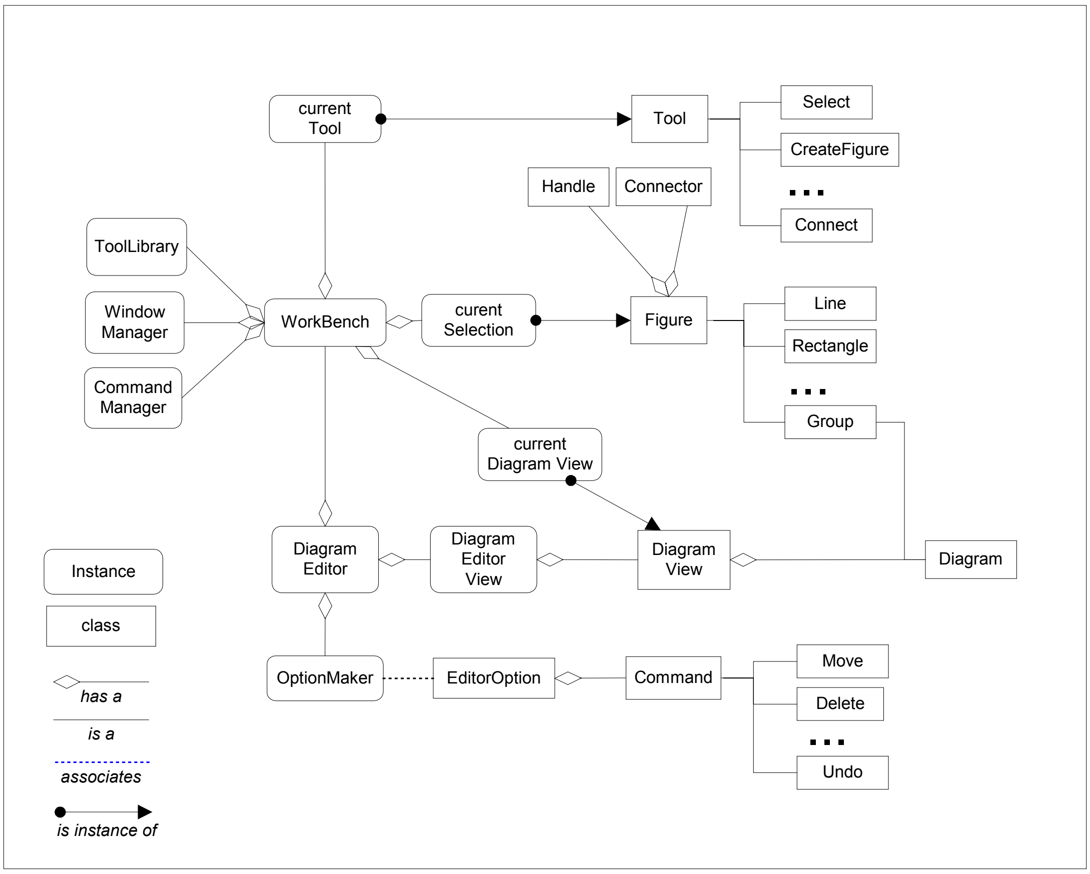


Figure 5.4: A high level view of the design of a diagram editor in DATsys

The `OptionMaker` reads the diagram editor's available options from a configuration file, associates commands with these options and represents these as GUI buttons, menu-items and shortcuts. Additionally, instances of `OptionMaker` maintain the automatic switching of the available options as the state of the editor changes. To understand the editor's state changes, `OptionMaker` listens to events spawned by the selection tool, the clipboard and other sources of change. For example, when the clipboard is empty, the paste option is switched off and when the current selection is empty, the cut and copy options are switched off.

A `DiagramEditor` is associated with a `WorkBench` object that operates as a façade for various operations. The `Workbench` maintains the selected figures on the canvas, the selected tool, the current diagram view, and a link to the tool library, window manager and command manager of the diagram editor. These managers are single instances that have global visibility from all the clients of the system.

Instances of `Diagram` represent the data-structure that stores the diagram elements of a diagram. Upon request, diagrams, much like figures and diagram elements, must be able to draw themselves. Diagrams are associated with a `DiagramView` that observes changes within figures through events, collects the damaged area that needs redrawing and decides when to issue a message to redraw the screen. In addition, `DiagramView` instances hold the diagram, manage the selection, draw the handles and dispatch the events to the current tool.

5.3.3 Figures

Figures represent the shapes that make up the graphical view of the diagram. Any simple or complex shape is an instance of a `Figure`. Primitives include straight and curved lines, shapes such as rectangles, ellipses, arcs and polygons, text labels, bitmap images and any other primitive shape that may be needed to draw the graphical representation of diagram elements.

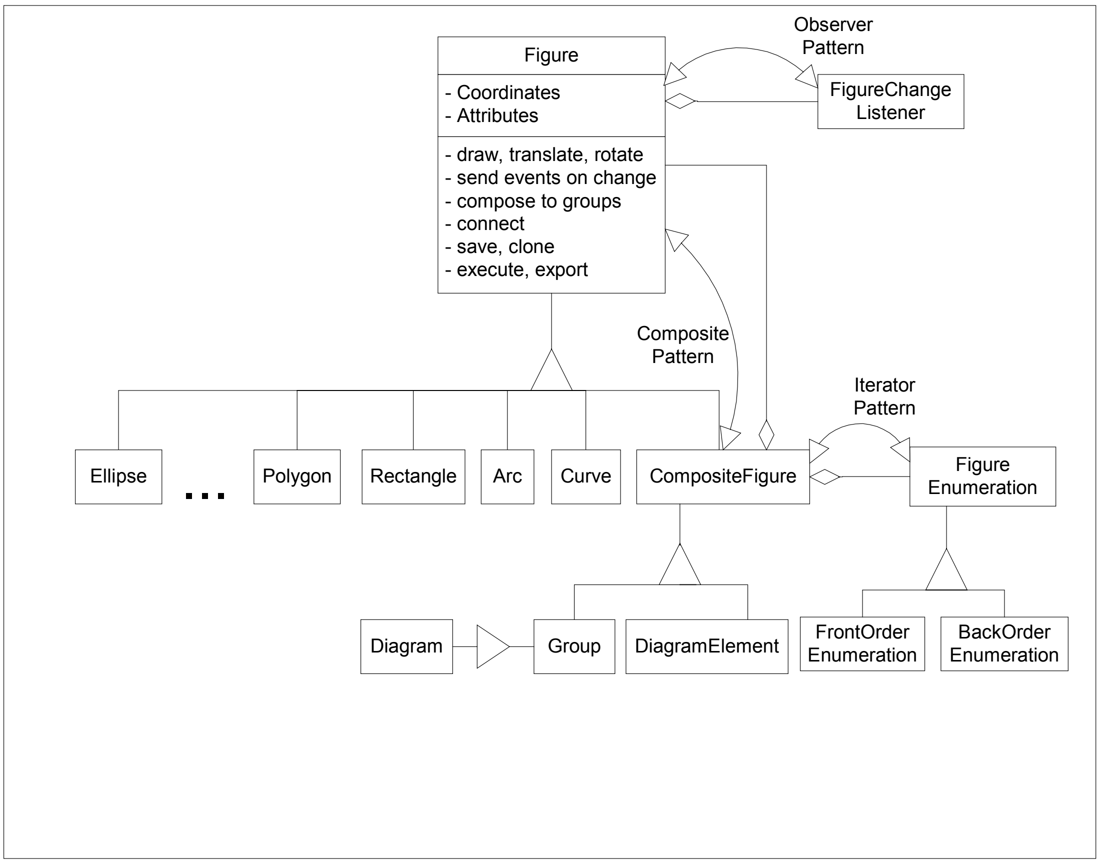


Figure 5.5: The design for the figure hierarchy

Figures maintain common attributes such as coordinates and colour. For specialised attributes, figures manage a hash-table that keeps name-value attributes. In this way, attributes can be associated with figures dynamically at run-time. Instances of `Figure` know how to draw themselves given a graphical context and how to transform their coordinates to move, scale and rotate.

Figure 5.5 illustrates the design of the Figure hierarchy. On attribute changes, figures issue and send events to registered `FigureChangeListeners`. The use of the observer pattern aids in decoupling the figures from the diagram and the selection editor. Decoupling figures from the diagram is important, as figures need to know the changes in connected figures to adjust accordingly. For example, while moving a connected figure the connection line has to follow. Decoupling figures from the selection editor of the authoring environment is also important, as the latter needs to know of changes on the selected figure to adjust its available options.

Figures can be composed to any depth of complexity into groups that are themselves figures. The composite pattern documents how to create this type of grouping. An important benefit for using the composite design pattern is that clients of the figure hierarchy do not need to differentiate between a primitive figure such as a rectangle and a complex group of figures such as a complete diagram. Specialisations of `CompositeFigure` are made by objects `DiagramElement`, `Group` and `Diagram`. Instances of `DiagramElement` are built graphically by domain developers. They consist of a composition of figures together with attributes representing a data model and instructions that represent connectivity semantics. `Group` objects encapsulate groups created for composition while editing. An instance of a `Diagram` is a specialisation of a group and inherits all of the group's children-management features. Its parent, the `CompositeFigure` implements the iterator design pattern, and returns `FigureEnumeration` instances that encapsulate the traversal of the composite. For a list structure, front to back and back to front enumerators are needed to allow depth ordering of the figures and precise hit detection. Figures contain handles that while dragged change attributes. Figures also contain connectors that encapsulate the way a figure can be connected to other figures. Finally, figures have application-based responsibilities and can either service or delegate messages for saving, cloning, exporting, executing, printing and translating to other formats.

5.3.4 Tools

Tools act as modes of interaction between the user and the diagram. The user selects a tool to interact with the diagram. Instances of `Tool` accept mouse and key events directly from the `DiagramView` and interpret them accordingly. Figure 5.6 illustrates a simplified view of the tool hierarchy.

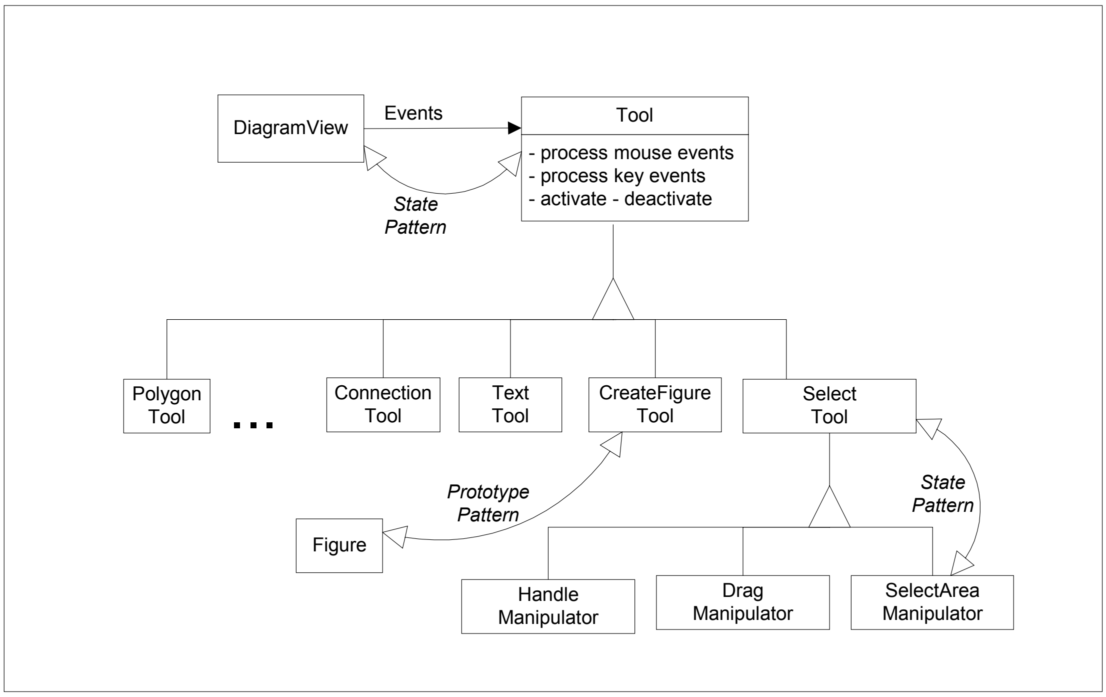


Figure 5.6: The tools hierarchy

Tools support operations such as creating, selecting, editing, and connecting the elements of a diagram. A parameterisable `CreateFigureTool` encapsulates the figure's creation behaviour for the click and the click and drag interaction. The latter is used in elements that need their dimension interactively defined at the moment of creation. The former is used in most diagrams in which diagram elements have either standard or automatically adjustable sizes. The `CreateFigureTool` employs the prototype pattern for making new instances. A request for returning a new instance sees the `CreateFigureTool` cloning its contained figure without having knowledge of its original type. The decoupling of the `CreateFigureTool` from the figure's type, facilitates the making of new creation tools at run-time. This is a fundamental extension point of the DATsys framework because additional diagram elements require new tools to create them. Figures that have non-standard interaction such as text labels, polygons, curves and arcs have their own creation tools.

The select tool exhibits context dependent behaviour. In accordance with the user's intention it can be used for single or multiple-selection, addition or removal of elements from the current selection, translation of coordinates, or interaction with handles. In each of these cases rubber banding gives visual feedback about the change. In addition, under the select tool, a user can move the selected figures and interact with their handles. Further specialisations of the select tool can be made so that the type of manipulation can be encapsulated in separate objects. The use of the state design pattern separates the state of the select tool from the select tool itself. Manipulators for handles, for dragging and for selecting an area are state objects that can be composed and used in other tools. Vlissides refers to a design of manipulators in Unidraw [VL89] in which the hierarchy of manipulators is separate from the hierarchy of tools. The decision of keeping the manipulator in the tools hierarchy may decrease the compositional orthogonality of the manipulators but increases simplicity.

The text tool allows the creation and editing of text labels within the diagram. A text label is a primitive figure that has a range of properties, such as size, font type, style, editability constraints, and so on. The text tool cooperates with external tool GUI components in order to accept configuration for these properties.

Finally, the connection tool creates connections amongst figures. It can be parameterised with any type of `ConnectionFigure`. A `ConnectionFigure` encapsulates the figure that appears to be connecting two diagramming elements and is typically represented by figures such as a simple, double or elbow lines with any decoration at its edges.

5.3.5 Commands

Commands encapsulate the possible actions of a user while editing the diagram. Command objects maintain the interaction so that its effects can be reversed upon request. Available commands include actions for grouping, aligning, transforming and editing attributes of the selected figures, transferring the selection, and interacting with the canvas. Figure 5.7 illustrates a small portion of the command hierarchy.

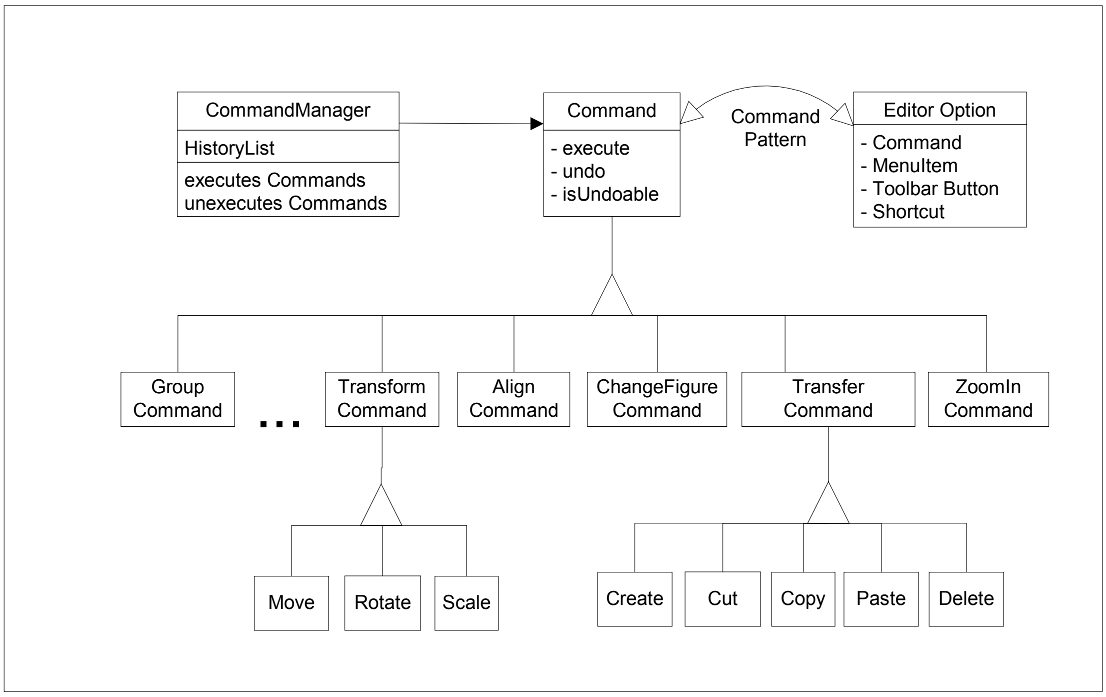


Figure 5.7: The commands hierarchy

Representing all the interactive actions with commands has many benefits. Firstly, the GUI can be decoupled from the classes of the editor. The various GUI components can execute commands using polymorphism, and therefore loose the dependence on their original type. The initial association of commands to GUI objects occurs at the initialisation phase through configuration. Secondly, a command hierarchy clearly partitions the functions of the editor and facilitates future extension. Thirdly, the addition of a `CommandManager` that handles command execution, reduces the amount of effort needed to implement a mechanism for undoing/redoing actions to any level of complexity. This mechanism has been described in literature as the command processor design pattern and is described by Buschmann at al. [BMF+96] and Sommerlad [Sp96].

The command pattern, when used in conjunction with other design patterns, lends itself to a set of extensions. For example, applying the composite pattern to the command hierarchy facilitates the creation of macro-commands that guarantee consistency. Macro-commands can be created by recording user actions and saving them within a composite command that can be later invoked.

In DATsys, a command is a property of an `EditorOption` that keeps together the GUI representations of the option, the state of the option, and the option's command. `EditorOption` objects are known to the listeners of the GUI components and upon activation delegate the command object for execution to the `CommandManager`. Consequently, the `CommandManager` executes the command and stores it on its stack of executed commands.

Additional commands can be created simply by adding new command classes that adhere to the command interface. Many reusable commands can be configured to reflect new behaviour. For example, commands for executing external tools, processing the diagram, connecting elements and many others can be parameterised to reflect the modelled diagram editor.

5.3.6 Handles

Handles allow the user to interact directly with figure properties such as dimensions, angles, sizes, and so on. Handles are associated with figures, position themselves to a location within figures, and encapsulate knowledge on how to be drawn. On a select operation, the `HandleManipulator`, if activated, delegates the events to the appropriate handle object.

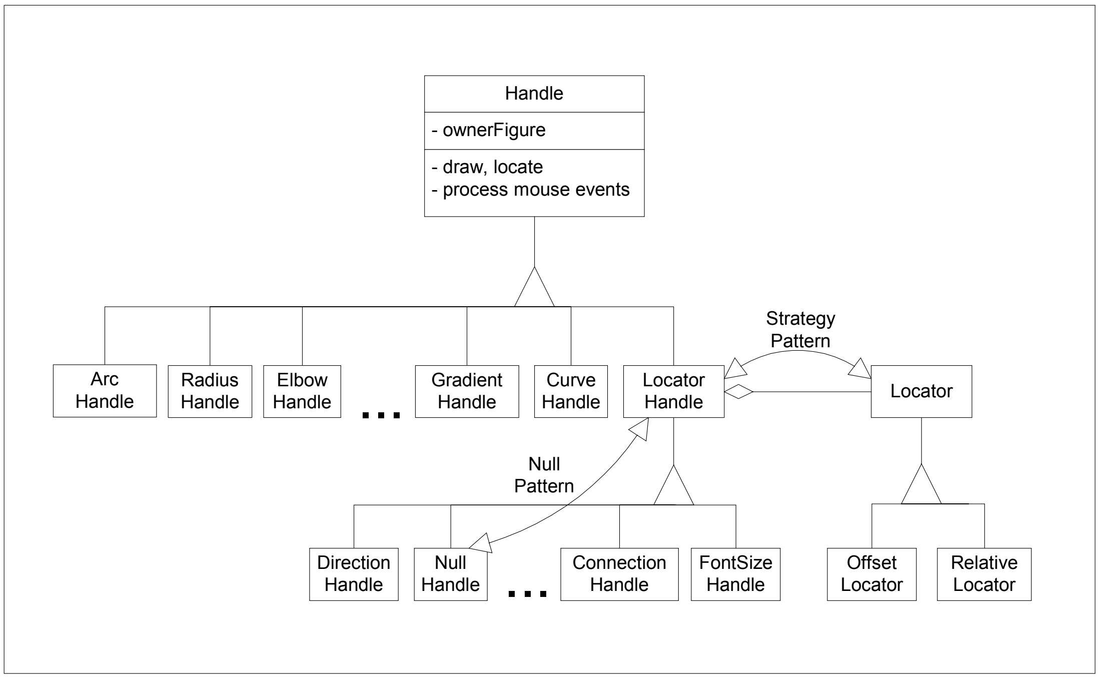


Figure 5.8: The handles hierarchy

A range of generic handles can be applied to all types of figures. Specifically, the handles for changing position and size, rotating and connecting can be applied to any figure. `DirectionHandles` for resizing are glued into the figure's edges and are available for every figure. These are created by using a `HandlesKit` that implements a simple factory pattern.

Special figures may need their own customised handlers. This is the case with figures such as arcs, curves, and polygons. Figure 5.8 illustrates a simplified view of the hierarchy for the handles.

Handles can find their position upon the figure that owns them by using an external hierarchy that implements the strategy design pattern. Locators encapsulate the strategy for locating the handle's coordinates on the figure. The design of handlers caters for two types of location strategies. The first strategy uses offset locators to calculate an offset from the origin coordinates of the figure. This is a useful strategy to define a handler's position as an absolute coordinate on top of its figure. For example, pin-type handlers on gate components use an `OffsetLocator`. The second strategy uses relative locators to find the location of a figure by calculating a coordinate relative to the existing ones. For example, size handlers use `RelativeLocator` in order to guarantee their placement at the corners and middles of the figure.

The `NullHandle` has been designed as an instance of the null design pattern. `NullHandle` handles receive all the events delegated to them but do not exhibit any behaviour. This eliminates the need for checking whether a handle is set to null.

`ConnectionHandle` handles can be parameterised with a type of connection so that users can connect figures during select mode. The `FontSizeHandle` handle, changes the size of a text figure. Other handles are unique to concrete figures.

5.3.7 Connectivity

Figures and diagram elements are connected using `Connector` objects. Connectors belong to figures, represent points of connectivity and can be linked using objects of the type `ConnectionFigure`. Figures can have any number of connectors associated with them, and these can have a variety of properties. Typical `ConnectionFigure` objects are constructed from lines that may use edge decorations such as arrows or diamonds.

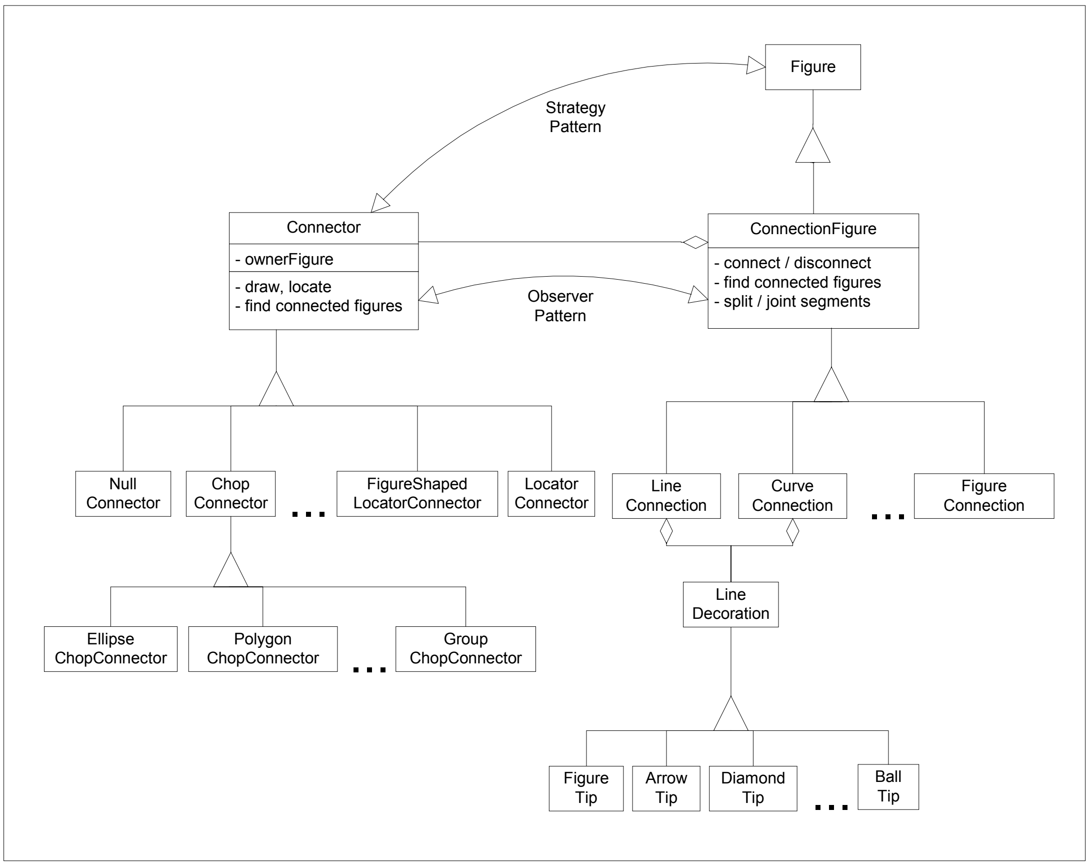


Figure 5.9: The connectors and connection figures hierarchies

Connectors implement the strategy design pattern towards the figure hierarchy. They encapsulate the strategy of how to connect with a `ConnectionFigure`, can draw themselves, and have the responsibility of returning the figures to which they are connected. Connectors also implement the observer pattern. Upon change, connectors notify their associated `ConnectionFigure` objects to automatically adjust.

Figure 5.9 illustrates the hierarchy of `Connector`, `ConnectionFigure` and `LineDecoration` objects. `ChopConnector` objects locate their coordinates by chopping the connection at the shape boundaries of the primitive figure to which they belong. `GroupChopConnector` objects do the same, but they belong to groups of figures. `LocatorConnector` objects represent connectors with relative coordinates to a figure. `FigureShapedLocatorConnector` objects obtain their graphical view and properties through a representative figure.

Users of Daidalos create `FigureShapedLocatorConnector` objects to represent customised types of connectors. `NullConnector` objects implement the null design pattern and support the services defined in `Connector` but are devoid of implementation.

`ConnectionFigure` objects are figures that provide services for connecting and disconnecting, defining, splitting and joining segments, and returning the figures that are connected to them. Typically, diagram notations use variations of lines to denote the relationship of connectivity. `LineConnection` and `CurveConnection` are types of `ConnectionFigure` that are implemented by poly-lines and poly-curve figures respectively. `ConnectionFigure` objects could also be customised with other types of figures such as rectangles and ellipses.

Both `LineConnection` and `CurveConnection` objects can be decorated at any of their ends with a `LineDecoration`. Typical decorations include arrows, circles, diamonds and boxes, of various styles and properties. `FigureTip` objects can be created by the user to represent a new type of relationship for a diagram notation.

5.3.8 Daidalos, the Environment for Authoring Diagram Notations

Daidalos is the authoring environment that allows the specification of diagram notations. As a `DiagramEditor` object, it follows the architecture described in section 5.3.2. Daidalos needs additional parts to create new figures, diagram elements, tools and commands. In contrast to other instances of `DiagramEditor`, Daidalos defines tools for making all these parts. In addition, Daidalos needs a selection editor that allows the editing of various properties, functions for the management of libraries of diagram notations and other options helpful for the authoring of new domains.

Daidalos could be considered as a meta-diagrammer, as it provides a graphical process for making parts for new diagram editors. The construction of these parts has to be interactive and simple for diagram-based CBA authors. Specifically, Daidalos must allow authors to define:

- Diagram elements with their graphical view, data model and connectivity
- Tools that describe the interaction with diagram elements

- Options and their links to commands

For the first task, Daidalos offers features for drawing graphical views to represent diagram elements. A range of primitive shapes and multi-level grouping features are available. Daidalos also supports the specification of a data model within a diagram element. The data model may have elements that can be visible or invisible, editable or non-editable. Connectivity properties can be specified visually. `Connector` objects can be defined as pins, perimeters of shapes, or custom made figures. `ConnectionHandle` objects can be associated to connectors that can be configured to create any type of `ConnectionFigure`. The `ConnectonFigure` most often used in diagram notations is a type of `Polyline` and this can be specified for its view, type, and decoration.

For the second task, Daidalos presents the user with facilities to manage libraries of tools. It defines the concept of a domain library and a tool library. Domain libraries are composed of tool libraries and tool libraries contain tools. Daidalos has functions to edit, load/save domain tool libraries, and create tools. Daidalos' functions for creating tools depend on the selection. By using the selection's contents, Daidalos chooses whether to make a tool for creating a figure, diagram element or connection. Any of the standard tools can be reused and become part of a new tool library.

For the third task, Daidalos allows creating a specification for defining the available options of a new diagram editor. Many editing options are unnecessary for specific diagram editors, especially when the diagram editor targets learners. For example, a zooming option for an editor of simple circuits is unnecessary, while for an editor that targets a complex flowchart exercise it is indispensable. Daidalos can accept new commands that can be linked to new options and used throughout the editors produced using DATsys. New commands can access the `WorkBench` object of an editor, link to the internal state of the editor, and provide any new feature.

5.3.9 Ariadne, the Environment for Authoring Exercises

Ariadne is the authoring environment for the specification of diagram-based CBA. Similarly to Daidalos, Ariadne is a subclass of `DiagramEditor` and follows the architecture described in section 5.3.2. Ariadne allows the specification of:

- The student diagram editor
- The exercise properties that are required by the CBA software
- The marking scheme that marks the diagram and returns feedback

The first task is supported by providing features for a further specialisation of the student diagram editor in respect of its tools, and its available options. Ariadne imports domain tool libraries specified in Daidalos and allows exercise authors to select the correct tools for a student diagram editor.

For the second task, Ariadne needs editing facilities for defining the properties of a CBA exercise. Properties include the exercise's question, weight, and various other types of information described in section 6.4.3.

The third task involves providing facilities to edit and test the marking scheme of an exercise. This can be done by incorporating a simple source-code editor into Ariadne and providing simple compilation and testing features. Finally, for the completion of the marking scheme, configurations for the marking tools that have been used should also be given.

5.3.10 Theseus, the Student Diagram Editor

Theseus is the diagram editor that is used by the students to solve the diagram-based CBA exercise. Theseus is represented with a subclass of `DiagramEditor`. All of Theseus' features are described as parts of its configuration. This configuration includes the exercise specific tool library and a specification for the available options. The tool library contains domain and exercise specific tools that use custom-made diagram elements. The diagram element's view, data model and connectivity have been described in the creation of the diagram element. The students can thus use the tools to solve the particular type of exercise by interacting with the tools, the canvas, the diagram elements and the various options.

Theseus should allow students to create a wide range of diagrams in many of the domains discussed in section 2.2.3. Theseus' GUI components can change by further parameterisation of Theseus with a new `OptionMaker`. In this way, Theseus can accept any type of component that is not available through configuration.

5.3.10 Summary

This section highlighted the main concepts in the design of DATsys and the three diagram editors that facilitate the authoring of diagram editors specialised to CBA exercises. The design of DATsys defines abstractions for making new diagram editors in a range of domains and explicit extension mechanisms for adding functionality.

Amongst the three diagram editors, Daidalos and Ariadne can be considered authoring environments that are used to create diagram-based CBA exercises. Daidalos provides functions for defining new notations and Ariadne provides functions for defining CBA exercises. Theseus is the student diagram editor that is uniquely customised, by the Ariadne user, to the requirements of the exercise.

5.4 The Design of the Generic Marking System

To facilitate experimentation with marking criteria in diagram-based exercises, an extensible marking system was designed that can be flexibly configured. Key concepts in the marking process of CBA coursework were modelled and explicit extension points were set for describing new metrics. In addition, the design included a technique to return immediate feedback and addresses concerns surrounding security, robustness and reliability.

Extensions for diagramming have been made for marking simple circuit diagrams, flowcharts and object-oriented designs. Extensions for programming have been made to assess Java and C++ programs. Guides on how to extend the marking system have been given in [HST02] and [Sp01].

5.4.1 Key Abstractions

The design of the marking mechanism is based on Ceilidh's approach of marking tools. The idea of marking tools has been a significant contributor to Ceilidh's success as a CBA system. Marking tools can be configured and used in new types of courses that were not anticipated during their development. Design improvements to Ceilidh's marking tools focused upon the expressiveness of the marking process, the configuration and composition of the various tests and the description of the student feedback.

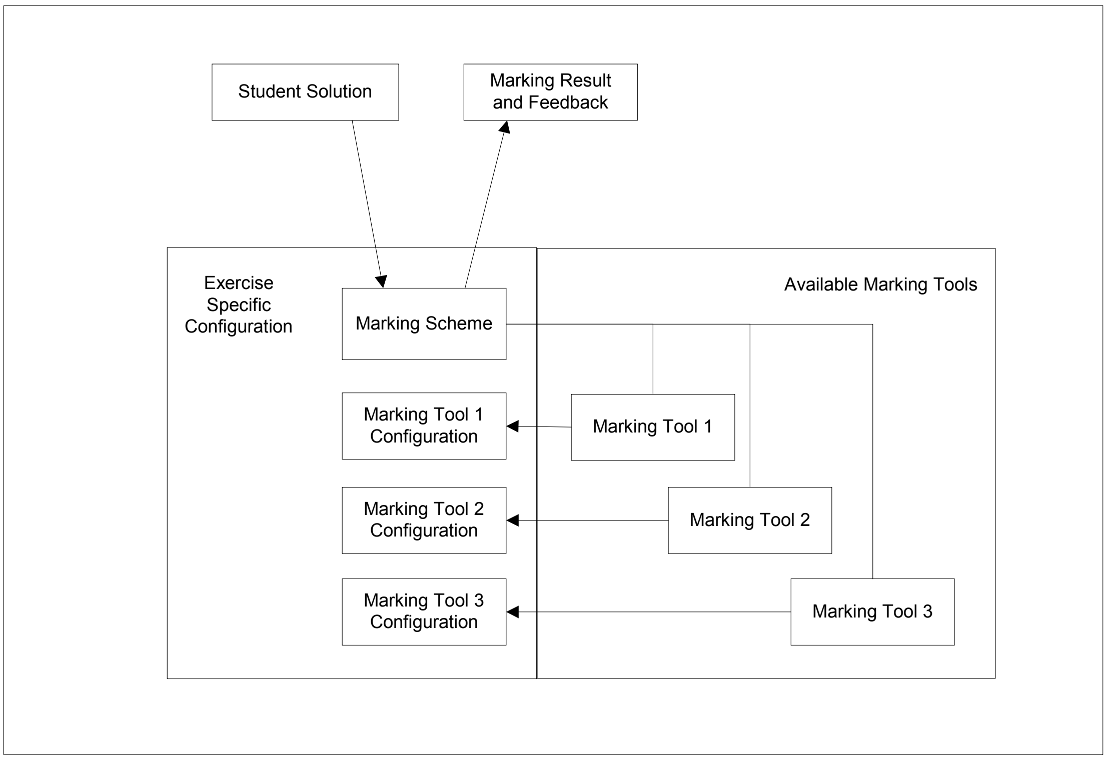

Student Solution
Marking Result and Feedback
Exercise Specific Configuration
Marking Scheme
Available Marking Tools
Marking Tool 1 Configuration
Marking Tool 1
Marking Tool 2 Configuration
Marking Tool 2
Marking Tool 3 Configuration
Marking Tool 3

Figure 5.10: A plan for a generic marking mechanism

The following concepts are identified for a general marking process:

- **Marking Scheme:** This describes the marking of any automatically assessable exercise. It is developed by exercise authors and is part of the exercise's configuration files. A marking scheme contains invocations to marking tools that have been configured for the specific exercise. It also relates weights to each marking test and builds the result mark is returned to the student.
- **Marking Result:** This is a tree structure that mirrors the execution of marking tests. Leaf nodes contain the mark and student feedback for the specific test. Group nodes contain the combined mark of their children and appropriate feedback.
- **Marking Tools:** These encapsulate types of marking criteria for specific domains or for generic use. Marking tools need exercise-based configuration to execute. Upon completion, marking tools return marking results.

- **Marking Tool Configurations:** These are configurations specific to marking tools and to exercises. Marking tools may use any form of configuration to specialise to the requirements of the exercise. CourseMaster uses regular expressions in the configuration of marking tools and associates weights and simple feedback with the result.

Figure 5.10 illustrates the main concepts for the generic marking mechanism. The student solution is sent to the marking scheme that invokes marking tools. The marking tools execute after being set to use exercise specific configurations. When the execution of all the marking tools is finished, marking results along with feedback are returned to the student. Any type of CBA marking can use this design so long as the appropriate marking tools and their configurations can be identified, designed and implemented.

5.4.2 Marking Scheme

The marking scheme is an evolved progression of Ceilidh's marking action. As explained in section 3.1.5, Ceilidh allows exercise authors to describe the marking process as a simple sequence of invocations defined in a "mark action", which is a configuration file. Each line of this configuration file has two elements:

- Name of the marking tool to be invoked
- Highest mark that the marking tool could contribute to the overall mark

The advantage of having a mark action as a property of an exercise is that new marking tools can easily be invoked for new exercises. The necessary configurations to the marking tools are placed into separate files that are stored together with other exercise related files. Although this method simplifies the authoring of exercises, it does not incorporate control structures for a finer grain customisation of the assessment for each exercise. Furthermore, Ceilidh's marking can only be extended through the implementation of new marking tools. For any small deviation from the functions of a marking tool, a course developer has to create new marking tools.

The restrictions of Ceilidh's marking action could be rectified if the marking action was a program, expressed in an imperative language. Customisations within programs can handle unanticipated extensions much better that within simple configuration files. Such programs could be made to invoke marking tools and to return results to the student. The clear limitation of this approach is a risk of decreasing readability. However, this risk can be partially rectified by restricting and simplifying the program, by using easy to understand aliases, or even by creating "wizard" tools to generate the program.

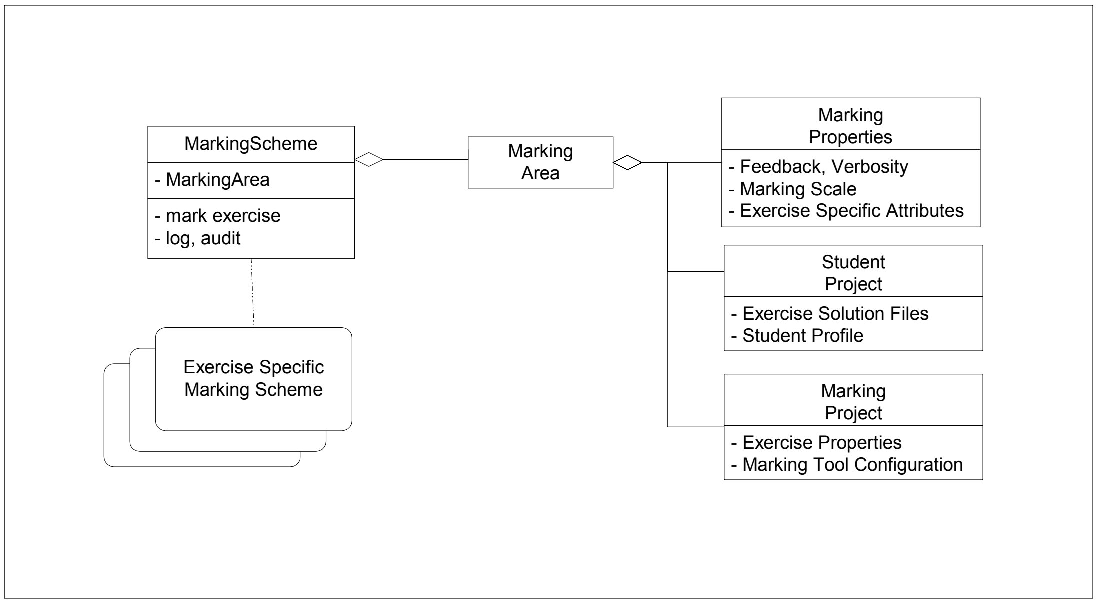


Figure 5.11: The marking scheme and its relationship to other data

Figure 5.11 illustrates the idea of designing the marking scheme as a program that is described within a class `MarkingScheme`. A description of the source for this class must be available for any assessable exercise. To mark an exercise, the appropriate marking scheme needs to be instantiated and executed. Instances of the `MarkingScheme` class need to obtain information related to the marking properties of the exercise, the student project, and all the exercise specific configurations of the marking tools. For simplifying the gathering of this information, `MarkingArea` is a facade object that contains the data a marking tool may request.

As the marking scheme of exercises is a program, control structures can be used together with other marking preferences. In addition, new marking tools, marking configurations, and external tools can be added with little effort.

5.4.2 Marking Result

After marking an exercise, marking results return to the student. Exercise marking results must be descriptive, comprehensive, and presented to the student unambiguously. Every marking tool creates and returns a group of marking results with as many results as the tests executed by the tool. As every marking test contributes to the creation of the overall marking results, an object `MarkingResult` can hold enough information to create both descriptive and comprehensive overall results.

Specifically, a `MarkingResult` can hold:

- An absolute percentage value that represents the attributed mark
- A weight value that represents the importance of the marking criteria that created the marking result
- A description for the type of marking
- Feedback information for specific aspects of the exercise's marking
- A style for rendering the marking result to the student

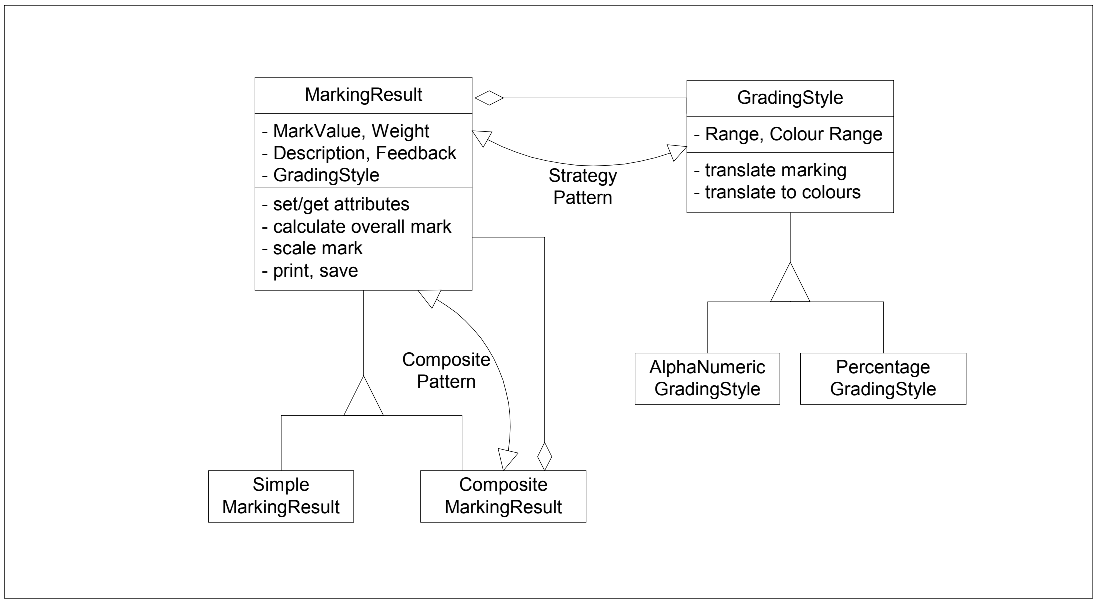


Figure 5.12: Marking results are associated with a style for rendering the marks to students

Figure 5.12 depicts, an abstract `MarkingResult` that can be either a `SimpleMarkingResult` or a `CompositeMarkingResult`. When marking tools execute marking criteria, they instantiate `SimpleMarkingResult` objects and aggregate these to `CompositeMarkingResults` objects. As marking finishes, all the `CompositeMarkingResult` objects returned by the top level marking tools, are composed into a root `CompositeMarkingResult` object that represents the overall mark.

Two design patterns are applied to the design of the marking results. Firstly, the composite design pattern between `MarkingResult` and `CompositeResult` allows the nesting of results into larger groups. Secondly, the strategy design pattern between the `MarkingResult` and the `GradingStyle`, objectifies the rendering of marking results into `GradingStyle` objects.

The overall mark can be calculated and scaled in a simple manner by adding a polymorphic method that traverses the tree structure using a depth-first algorithm. `MarkingResult` instances support the printing and saving into contexts that are supplied as parameters.

In Ceilidh feedback to the student is limited to a mark composed of the results obtained by the marking tools that participated in the marking process. No direct justification explains the loss of marks to the students. In addition, no explanation is given in order for the student to improve their mark. One benefit of using a tree structure for representing the results is that feedback with more details and precision can be created and presented to the students.

A `GradingStyle` encapsulates the rendering of the marks to the students. Grading styles exist for associating alphanumeric values to the marks, changing scale ranges, and linking presentational attributes such as colours and shapes. The exercise author can choose whether the student's mark is to be displayed in a numeric or in an alphabetic scale. The association between numeric values, letters, colours and shapes can be chosen through configuration.

5.4.3 Marking Tools

Marking tools encapsulate a specific test or series of tests that execute against the student solution. The abstraction that guides the design for marking tools is simple: marking tools execute and return an instance of a `MarkingResult`. Instances of `MarkingTool` objects can start marking when connected to a marking area. A marking area represents a conceptual area where the exercise marking takes place. It contains information necessary for the execution of marking tools for the specific exercise. Marking tools need to know all the exercise-based marking properties, the student coursework project and the marking project. The marking project contains all the exercise-specific configurations for the marking tools.

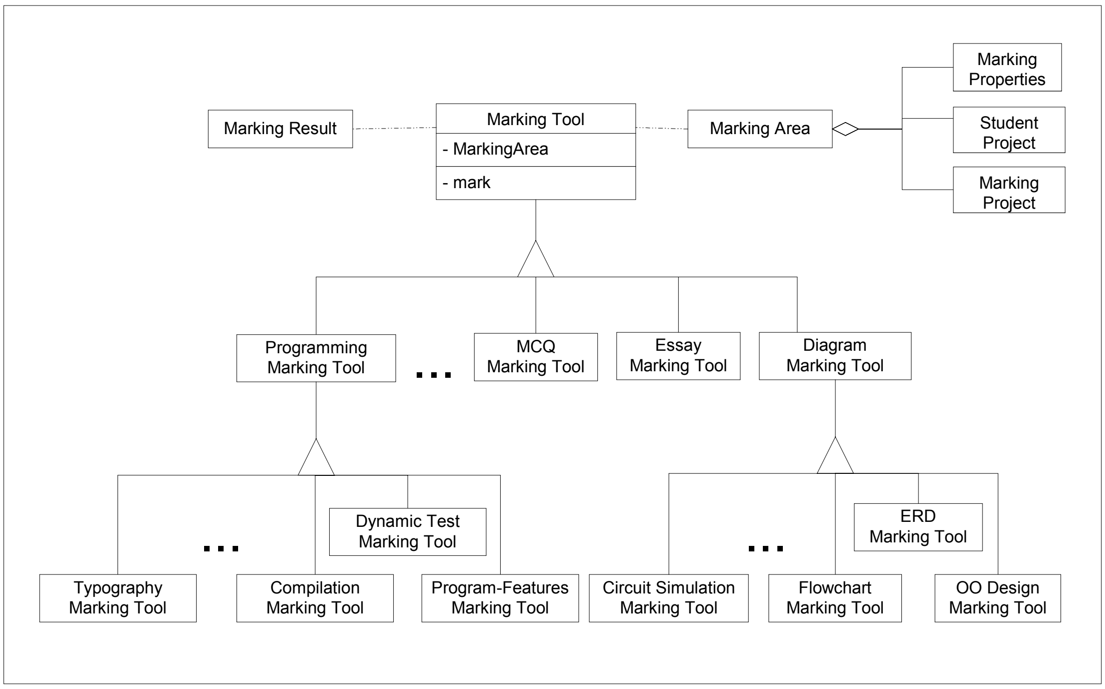


Figure 5.13: The hierarchy of marking tools for programming and diagram-based courses

Figure 5.13 depicts the hierarchy for `MarkingTool` objects and their association with the marking area. Marking tools can be developed for programming courses, MCQ's, essays and diagramming. Configurations for new marking tools must always be available within the `MarkingArea` at execution time. In addition, the `MarkingArea` holds the data that belongs to the student's submission.

Diagramming marking tools, after their conceptualisation, can be implemented with relative simplicity. The student diagram, being associated with the marking area, can be instantiated and is available as an object to the marking tools. It can be queried to provide information for its structure, its diagram elements and their relationships.

For example, a logic-simulator marking tool can use the names of the diagram elements to associate logic behaviour and simulate the circuit. An analogue circuit-simulator marking tool can export the diagram's structure in an appropriate format, and use an external tool to simulate and test the correctness of the diagram. A flowchart diagram marking tool can generate the source code of a program, execute it, and then use the dynamic marking tool, implemented for programming courses, to test the output in relation to the input. An object-oriented-design marking tool can run exercise specific metrics by configuring the feature marking tool of programming courses, or translate the diagram and use external metric tools.

5.4.4 Configuration of Marking Tools

Each marking tool is associated with exercise specific configuration that must be specified in an appropriate format. The tools discussed in section 5.4.3 could conveniently use Ceilidh's marking tool configuration mechanism that is based on oracles. Oracles are search criteria expressed using regular expressions. Oracles are described in detail in [FZ93].

The configuration of marking tools may contain the range of feedback appropriate to the exercise and student. For the former, feedback must be distributed to appropriate marking ranges. For the latter, a mechanism must be devised to use the student's profile and history to select the appropriate feedback.

5.4.5 Summary

This section highlighted the key concepts for the design of a generic marking system that can be used to mark student coursework. The design suggested having, for every exercise, a marking scheme expressed as a program in an imperative programming language that invokes and configures appropriate marking tools. It also defined a technique for the composition of results and the running of marking tools. A newly written marking tool has all the information it needs to execute and return a marking result. This information includes the various marking tool configuration files that have no restriction on their encoding format. The generic marking system can be seen as an open system that accepts plug-in marking tools and configurations to allow their execution. Marking tools can be designed and built for any type of assessment as long as the criteria for the assessment can be automated.

5.5 The Design of the CourseMaster CBA System

CourseMaster is a redesign of the Ceilidh system that aims to integrate the diagram editor authoring system and the generic marking system while improving Ceilidh's scalability, performance, maintainability, extensibility, usability and platform independence.

5.5.1 Key Abstractions

In an effort to reorganise Ceilidh's functionality in a more extensible way, the dependencies, commonalities and variations between Ceilidh's tools and data layer are identified. The commonalities are abstracted into class hierarchies. Explicit extension points are defined together with parameterisation for all the variation.

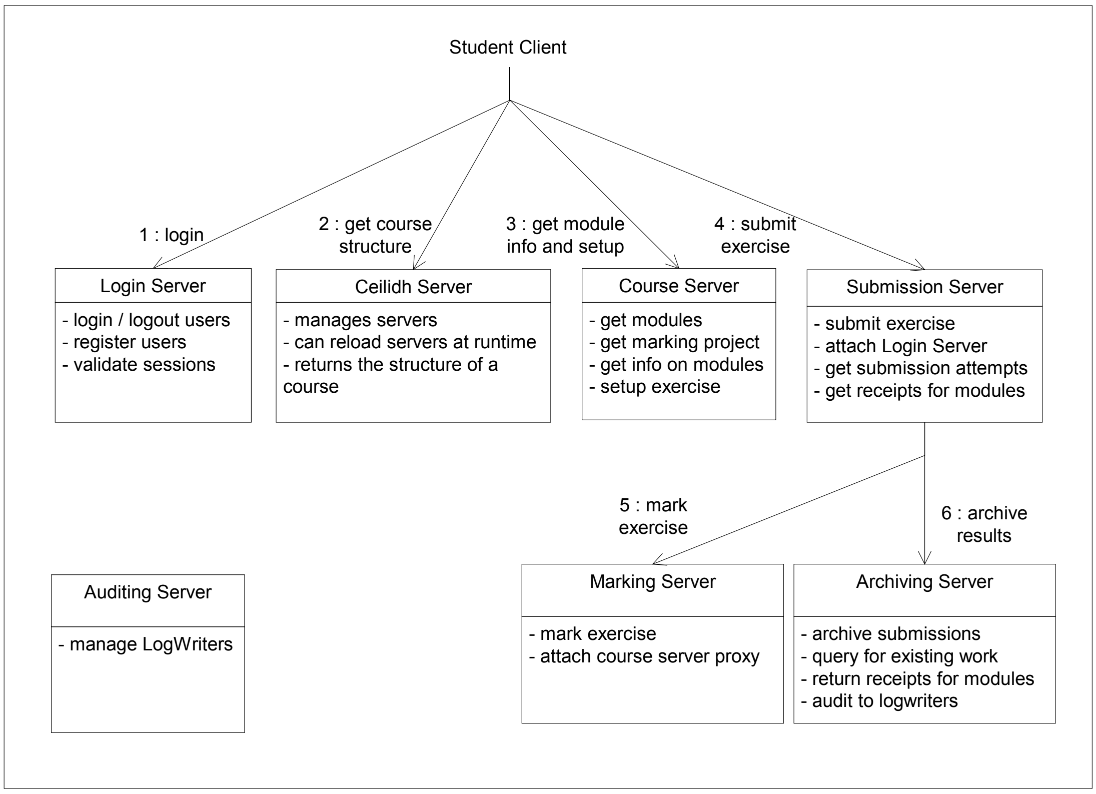


Figure 5.14: The organisation of CourseMaster servers

The dependencies between Ceilidh's parts can be decreased by separating the various responsibilities between seven logical parts. These parts can be designed as servers that operate as remote objects. Each server manages an associated file-store. Figure 5.14 illustrates a high level view of the relationship between the servers and shows the walkthrough of a student submission. Every server is decoupled from the others so that it can operate independently. For the internal communication between servers, proxy servers can delegate the messages, and common objects can be exchanged.

The login server is responsible for maintaining the user database and providing authorisation and registration. It also provides session validation services to the other CourseMaster servers. The course server manages course material and responds to related requests.

The submission server is a façade of the marking system and contains the logic to decide between accepting and rejecting a submission. The rules for this decision take into consideration exercise specific properties such as the state of the exercise and the number of available submissions, as well as student specific properties such as the number of submissions already spent, permissions for late submissions or additional submissions, and so on.

Upon validating the student submission, the submission server sends it to the marking server and expects a marking result. The marking server executes the generic marking mechanism described in section 5.4. Marking tools compose the exercise's overall marking result that is returned to the submission server and sent to the archiving server. The archiving server stores the student submission together with the marking result and issues an archiving receipt. The submission server uses this to issue a submission receipt that contains the marking result. At this stage the submission server associates the appropriate grading style with the marking result and returns it to the student.

The auditing server maintains auditing for a set of predefined functions of CourseMaster. All the other six servers communicate with the auditing server to log information to files or network sockets. The ceilidh server manages the interaction between servers by setting-up appropriate proxy servers. In addition, it builds and returns to students the module structure of a course. CourseMaster's servers inter-communicate using a set of objects that are exchanged between all parts. Table 5.1 lists the objects and describes their meaning.

A student makes a submission to the submission server, and expects a receipt issued from the archiving server and a marking result from the marking server. A course module is sent to the student after logging in, to allow browsing and selection of the appropriate unit and exercise.

A `project` is an object from a hierarchy of classes that represent the type of assessment. Projects represent any type of assessment and contain basic information about filename extensions, student exercise environments, and so on. Objects of the type `MarkingProject` contain all the configurations for the marking tools of an exercise. `GenericProject` objects can be customised at runtime to specialise even further the type of a project.

Common Objects	Description
`Submission`	Contains all the data of a student's coursework solution, includes student information and carries security related sessions keys.
`MarkingResult`	Represents a single or composite assessment result for a student's solution. It carries feedback for each of the assessment criteria.
`Receipt`	Confirms the completion of a submission. It is issued by the archiving and submission servers and is sent to the client.
`CourseModule`	Embodies a whole course, unit or exercise. Every level has its own structure and properties.
`Project`	Encapsulates the type of assessment. CourseMaster provides classes for projects in programming, diagramming, and essays.

Table 5.1: CourseMaster's basic objects for communication between the servers

5.5.2 Login Server

The login server manages the authorisation of users in CourseMaster. To increase security, the login server has an external and an internal interface. The external interface allows the authorisation and registration of users. The internal interface can be seen only by CourseMaster servers and supports authorisation for all user requests.

After validating a user, the login server asks a session manager to create a connection for the specific user. The session manager generates a session key, associates it with the user and clones an appropriate `Connection` object to send back. Clients can access

CourseMaster servers only through this connection object. The benefit of this design decision is that clients do not know explicitly the location of CourseMaster servers and therefore changes to the distribution of CourseMaster servers are transparent to the clients. This can be very useful in the case of adding load-balancing features to increase scalability and performance.

The session manager maintains all the created sessions and provides validation for sensitive tasks such as coursework submission and retrieval of previous marks and solutions. When a student exits, the session is destroyed. If a student logs in from another location, the session is replaced. Each session is date and time stamped, logged and audited.

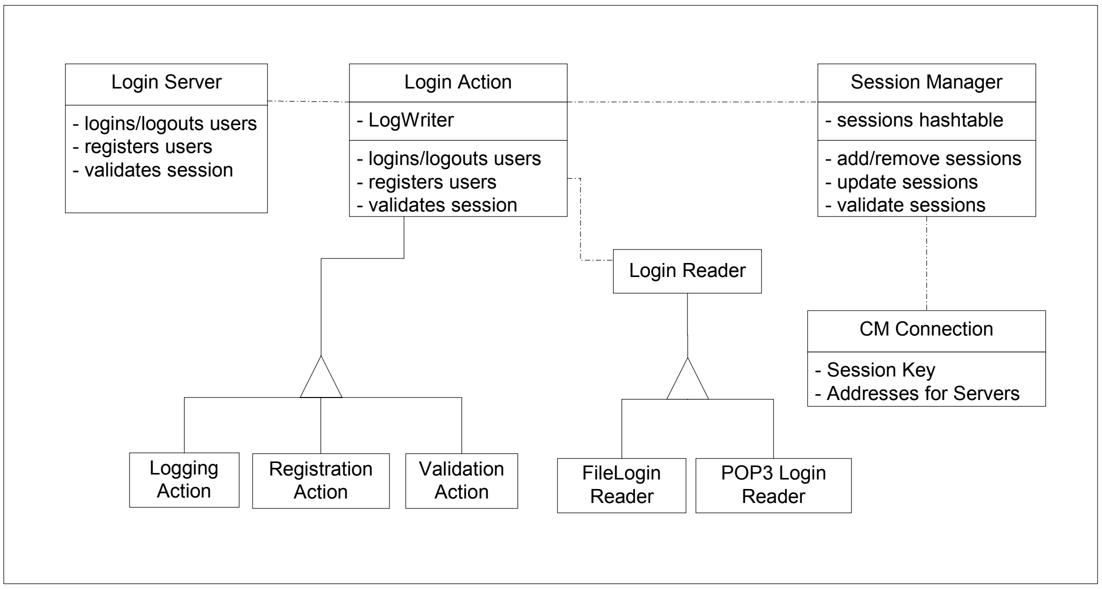


Figure 5.15: A high level view for the design of the login server

Figure 5.15 illustrates a view of the design of the login server. A `LoginServer`, depending on the nature of the request that it services, instantiates a logging, registration or validation action and executes it. A logging action uses a `LoginReader` instance to locate the password. The password may exist in a file, a POP3 server or any other server that offers validation services.

5.5.3 Course Server

The course server manages all the available modules and their data. On initialisation, the course server reads the file-store to build the structure for the modules. Modules are tree structures with four types of nodes. System modules are composed by course modules that contain unit modules. Unit modules contain exercise modules.

A `Module` object can build itself by delegating the building message to its children. Figure 5.16 illustrates the relationship between a course server and its modules. Every node has attributes. For example, a course module has a name, a title, notes and a summary. The properties of exercise objects are described in a property file that contains name–value pairs of variables. On the construction of a module, the exercise properties are read and are represented by objects of the type `ExerciseProperties`. Attributes within the `ExerciseProperties` contain parameters for various aspects of the exercise's configuration. For example, an exercise's skeleton name, the maximum allowed number of submissions, availability status (e.g. open/closed/late) and many other properties described in [TS99].

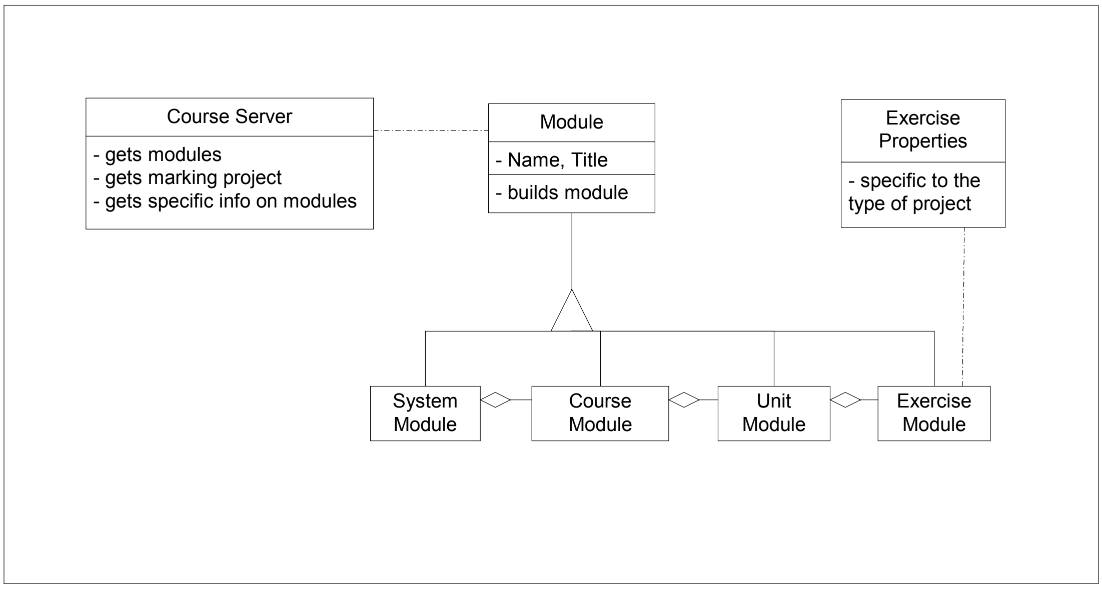


Figure 5.16: The course server is responsible for creating and managing course modules

Exercise properties in the Ceilidh system are inherited from the unit, course and system levels. Therefore, properties like the maximum allowed number of submissions can be specified for a whole unit and can be overridden for a specific exercise. Although it was anticipated that this mechanism would decrease the unnecessary repetition of properties, it was observed that it actually hinders readability and maintainability. CourseMaster omits the inheritance mechanism for the properties of exercises. Each exercise has a separate property file that contains all the required parameterisation. Further parameterisation can be made to address the configuration of other types of assessment such as diagramming and generic courses.

5.5.4 Submission Server

The submission server accepts student submissions and sends these to the marking server. Upon completion, the submission server contacts the archiving server and makes a request to archive the submission.

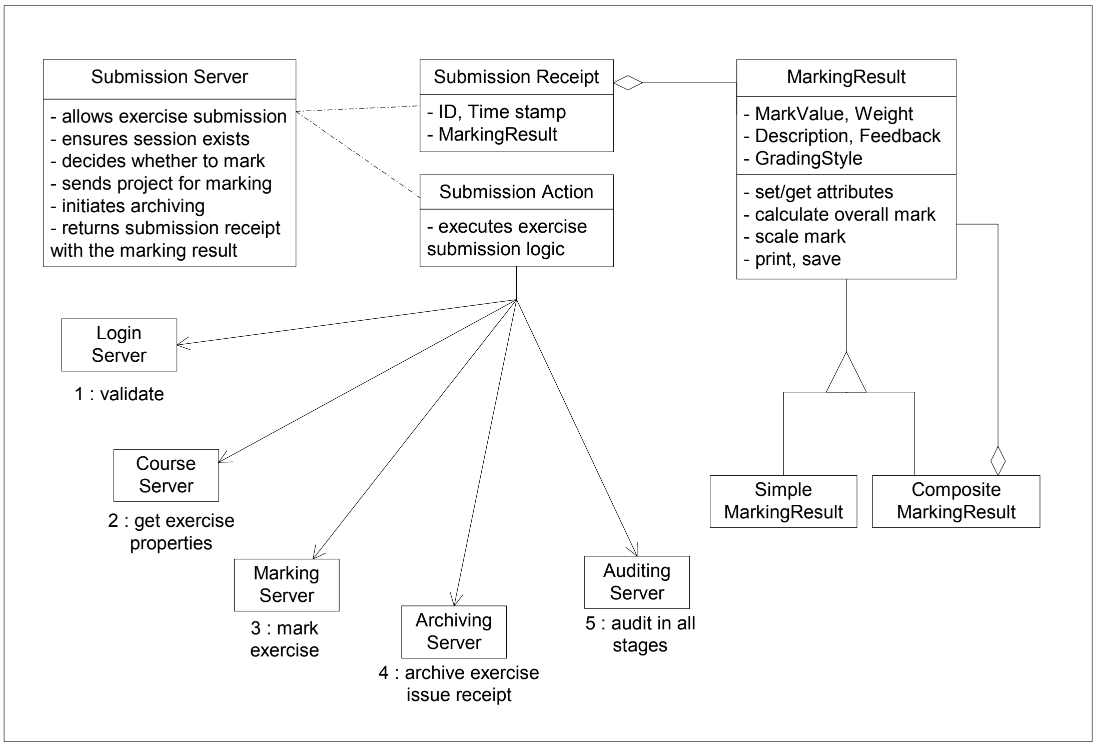


Figure 5.17: The order of messages for a single submission

The submission server decides whether to proceed with a submission after checking some rules: the exercise must be open for submission, the student must have at least one submission left, unless it is specified otherwise by a property of a late or extra submission. At the end of the submission process, the submission server issues a receipt that has a unique identity, a date and time stamp and the submission's marking result. The marking result is scaled according to the scales of the course and exercise. A `GradeStyle` object is associated with the marking results and customises their view as they appear to students.

Figure 5.17 illustrates the order of invoked services to complete a submission. It also shows a simplistic view for the design of the submission server as it relates to the other parts of CourseMaster. The execution of the logic that decides whether to permit the submission is within a `SubmissionAction` object. This gets instantiated by the submission server for every submission request.

A submission action has to first contact the login server to validate the session. Upon confirmation, it has to query the course server to receive the exercise's properties and marking properties. At the next stage, the `SubmissionAction` object has to send the student submission to the marking server, to receive the marking result and to send the submission to the archiving server for storage. Finally, it has to issue and return the submission receipt by configuring the marking result according to the properties of the exercise, and place it within a new submission receipt.

5.5.5 Archiving Server

The archiving server manages the archiving of student work and marking results. In addition, upon request, it returns stored coursework or marks for exercises, units, and courses. For security reasons, students shouldn't have access to the archiving server. The archiving server services are provided via delegation by the submission server.

Figure 5.18 depicts a high level view of the archiving server and the archiving action hierarchy. Upon request, the archiving server constructs an `ArchivingAction` object and requests it to execute. `ArchivingAction` objects model the actions of storage and retrieval of normal and late submissions.

Additional behaviour within the late submission action can define various strategies for dealing with late submissions.

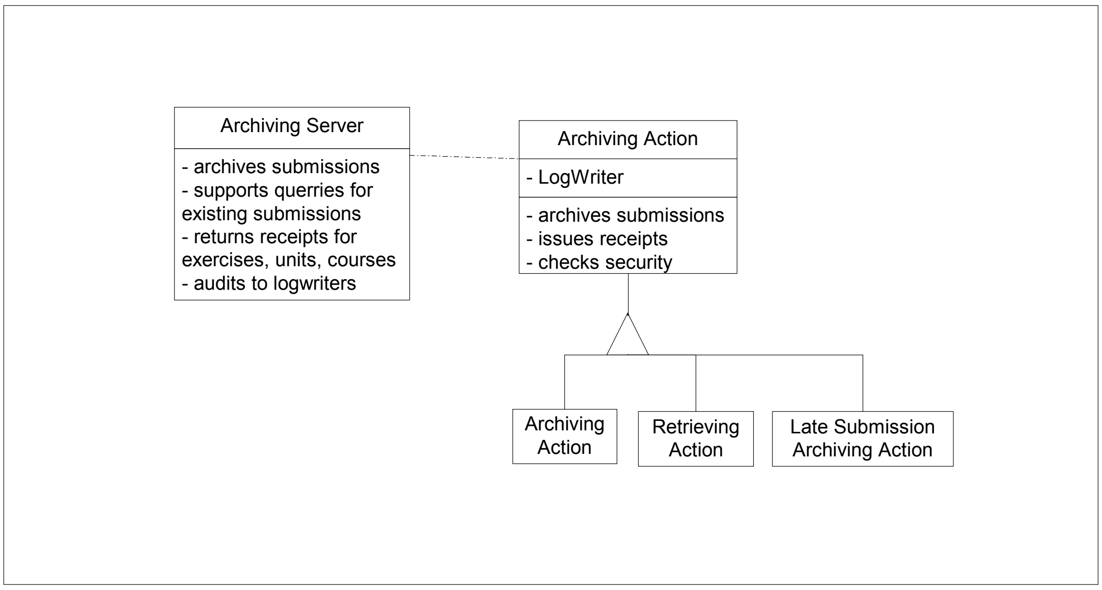


Figure 5.18: The archiving server and the archiving action hierarchy

5.5.6 Auditing Server

The auditing server is visible to all CourseMaster servers. Each server employs the auditing server accordingly to its auditing configuration. Four levels of log messages have been considered in CourseMaster:

- Level 0: Normal Auditing
- Level 1: Detailed Auditing
- Level 2: Troubleshooting
- Level 3: Debugging

In terms of auditing interest, logging of the appropriate level is outputted to `LogWriter` objects that have been registered with the auditing server. `LogWriter` objects can be grouped into `MultipleWriter` objects and broadcast messages. Every `LogWriter` instance has an associated `LogSink` instance that represents the sink for printing the output. Possible sinks include files, the screen and sockets.

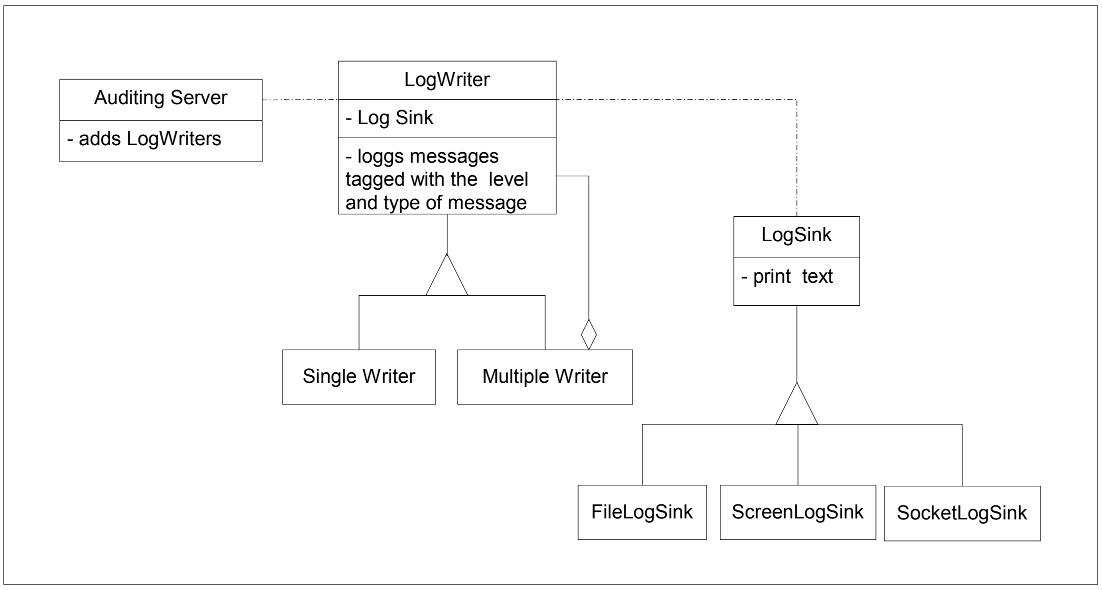


Figure 5.19: The auditing server and the `LogWriter` and `LogSink` hierarchies

Figure 5.19 illustrates a view of the design for the auditing server. CourseMaster servers register a `LogWriter` object with the auditing server and directly print log messages to their `LogWriter` object. According to the server's runtime configuration, `LogWriter` objects can be assigned to any instances of `LogSink` objects and filter an appropriate level of messages before printing. In large classes of students, both auditing and live monitoring of the execution of the servers are necessary features that not only improve security but also aid the administration tasks.

5.5.7 Integrating CourseMaster with DATsys and the Marking System

CourseMaster integrates a diagramming type of project that invokes Theseus. The diagramming type of exercise, in addition to the regular exercise files, includes a tool library and an application configuration file. Both configuration files are generated by Ariadne, are part of the coursework project that is sent to the student, and configure the student editor to the editing requirements of the exercise.

Within Theseus, when a student solution is to be saved, all the diagram elements with all of their details are saved. The diagram is restored to the marking area and can be processed in code defined in diagrammatic marking tools. A wide range of options allows traversing, translating, converting and understanding the diagram.

Identifying the elements of the diagram in most diagram notations is necessary and requires the naming of the elements within Daidalos. The diagram object is accessible to marking tools and can identify diagram elements with specific properties. All diagram elements can be queried concerning their connectivity and data model. Translating the structure of the diagram requires associating tags to the nodes and relationships of the diagram and invoking the exporting feature of the `Figure` class. Diagrams can return `FigureEnumeration` iterators that encapsulate the traversal of the diagram. Traversals can be described externally as well, because every `ConnectionFigure` object can return the diagram elements to which it is connected.

The generic marking system is managed by the marking server that, similarly to other CourseMaster servers, instantiates an action object. The marking action sets up the marking area and executes itself. Exercises that use new marking tools can plug into the system at run-time.

As a diagram-based CBA authoring environment, Ariadne is necessarily coupled with CourseMaster. Ariadne needs to have knowledge of the files that describe the exercise configuration of a diagramming project. Ariadne's options to build exercises are based on the format of these files.

Finally, Daidalos has been designed to operate completely independently from CourseMaster.

5.5.8 Summary

CourseMaster's design improves Ceilidh's design and integrates cleanly with the DATsys framework and the generic marking mechanism. Ceilidh's tools and database layers are broken into seven servers that can be distributed on a network. Each server manages its own file-store and is contactable in a secure way by the other servers. The seven servers together with the necessary abstractions that model standard CBA concepts support similar functionality to Ceilidh. However, CourseMaster's architecture has better foundations for flexibility, maintainability, scalability, performance, robustness and portability. The next chapter illustrates how the design decisions increased all these software related qualities. The integration of DATsys with CourseMaster required the addition of a new project type together with the appropriate marking tools and configurations.

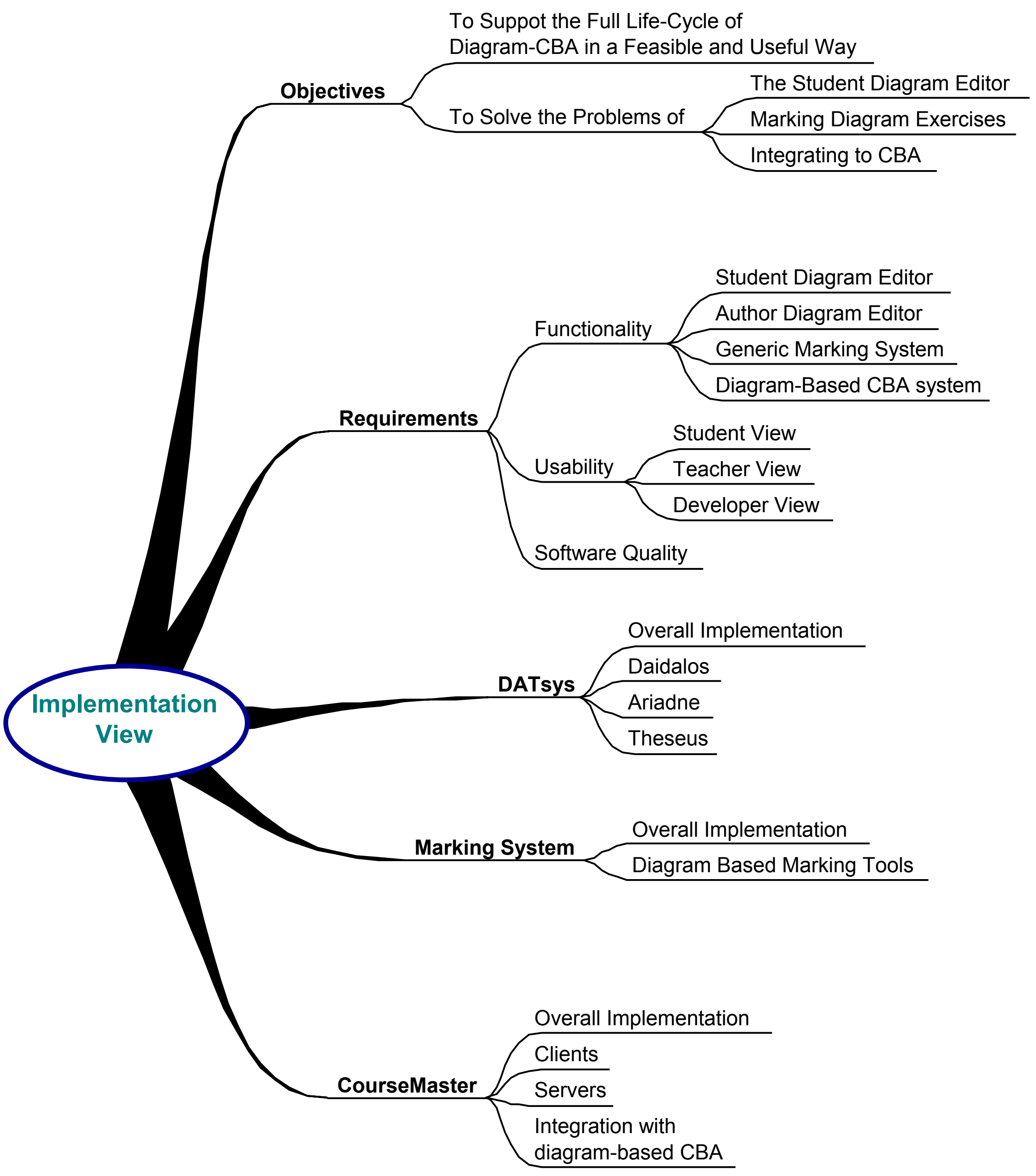
5.6 Summary

This chapter presented the design for an authoring environment and platform for the support of the full lifecycle of diagram based CBA. The authoring environment contains the DATsys object-oriented framework and three diagrammatic editors. Daidalos is a diagram editor that allows course developers to define abstractions for providing diagram notations. Ariadne uses these abstractions to allow exercise authors to create diagram-based CBA exercises. Theseus is the student diagram editor that is customised to the specifics of the exercise. Theseus options, editor tools and available diagram elements are configured on a per exercise basis by exercise developers.

The platform consists of the generic marking mechanism and the CourseMaster system. The generic marking system uses a marking scheme configuration per exercise to describe the invocation of marking tools. A programming language is used to express this description for increased extensibility and expressiveness. The generic marking system defines abstractions for domain dependent marking tools and their configuration and marking results. CourseMaster is designed with the intent to integrate DATsys and the marking system to a CBA, but also to improve on the existing Ceilidh's software qualities. The next chapter describes the implementation of the architecture and the designs described in this chapter.

Chapter 6,

Implementation



" I would rather write programs to help me write programs than write programs",

Dick Sites, 1974

Introduction

This chapter presents a reference implementation for this research. This consists of the DATsys object-oriented framework, the developer's authoring environment Daidalos, the teacher's authoring environment Ariadne, the student's diagram editor Theseus, the generic marking system and the CourseMaster CBA system.

Daidalos is a meta-diagramming editor for creating libraries of diagrammatic elements and templates for new domains. It contains functionality that allows the runtime parameterisation of predefined extension points. CBA authors use Daidalos to create customised student diagram editors that are unique to a CBA exercise. Ariadne provides functions for the construction of the automatically assessable exercises in diagram domains defined by Daidalos. Its users are exercise developers that author all the relevant information to create a diagram-based CBA exercise. The final diagramming editor, Theseus, is the student environment that is customised to both a specific domain and an exercise.

The objectives for the implementation of each part are set out in section 6.1. Section 6.2 describes the requirements for the implementation and the user views. It also discusses issues relating to software quality and general usefulness. Sections 6.3 to 6.5 present how DATsys, the generic marking system, and CourseMaster implement the design described in chapter 5.

6.1 Objectives

The main objective of this research is to investigate the feasibility and usefulness of designing an authoring environment for diagram-based CBA. Section 4.2.3 has explained that building an implementation which can be practically tested under real conditions is fundamental to evaluating the design's feasibility and usefulness. The implementation considers issues related to software quality, diagramming and education technology.

The implementation strives to meet the following goals:

- To implement the design for the customisation of the student diagram editor, to bind the editor with the generic marking system and to integrate these with an implementation of CourseMaster
- To provide features for supporting the full lifecycle of diagram-based CBA
- To address software quality issues
- To provide a realistic and extensible infrastructure in which diagram-based CBA can be researched and evaluated

6.2 Requirements

In light of these objectives, initial effort aimed at establishing the system and user requirements for each implementation. The requirements identify what the implementation should do (functional requirements) and how to do it (usability and software quality requirements).

The implementation requirements for each software part are:

- To support the functionality needed
- To address usability for all parties involved
- To satisfy software quality considerations

The next sections describe in details each set of requirements.

6.2.1 Functionality

A series of features must be implemented to support the full lifecycle of diagram-base CBA. In addition, for every stage of the lifecycle of a CBA exercise and for every participating type of user, the implementation must provide appropriate views and must link these to the available features.

The authoring stage encompasses the two main tasks that are necessary for building diagram based CBA. The first task is to create a specification for the domain of diagram notation to which the exercises belong. This specification must be described in a simple manner. It involves the design of each of the diagram-elements for the intended notation, with their views, connectivity semantics and data model. Once this specification is ready, the second task is to author the CBA exercise. This stage can be broken down into three distinct tasks. The first task involves selecting appropriate diagram-elements, tools, and application-based options for the student editor. The second task involves devising the marking strategy for the student diagram by creating a marking scheme that uses and configures appropriate marking tools. To complete the exercise, the third task involves adding all the necessary CBA-related information to the configuration of the exercise.

The completed CBA exercise can be deployed and set to accept submissions. The running stage presumes that a CBA system has been deployed and is operational. It involves the use of the CBA system by its users, the use of the student diagram editor to complete the exercise, the execution of the marking mechanism and the creation of appropriate marking results and feedback.

The administration stage encompasses all the tasks for course and exercise management. This entails monitoring student and overall course results, editing various course properties, registering and maintaining user lists and in general providing functions similar to those provided by the Ceilidh system.

6.2.2 Usability and Usefulness

The target audience for CourseMaster are non-programming users. Therefore, it was vital to address usability issues throughout CourseMaster's implementation phase. The design of the user views must adhere to well-defined HCI standards. The user views must be coherent, concrete and easy to navigate. When a design choice is presented that necessitates taking a choice between simplicity and functionality, simplicity should always be prioritised, and a respective trade-off has to be made.

The implementation of DATsys and its authoring environments must also take into consideration the fact that teachers, course developers and administrators do not necessarily have programming experience and do not have ample time to delve into the internals of the CBA system in order to create, amend, or extend CBA exercises.

DATsys novelty is that it allows such constructions and modifications to take place by *non-programmer* users through visual composition.

The implementation of the generic marking system takes into account the fact that extensions are not only permitted, but are to be expected at any time in the foreseeable future. The marking system has been designed in a highly reusable and extensible manner. Extensibility and reusability are indispensable qualities to an open and generic marking system which can support marking across a variety of domains and disciplines.

The implementation of CourseMaster, the software base under the marking system and DATsys, has to take into account the practices followed by the other two parts. As an open CBA platform, CourseMaster has to be reliable, maintainable, portable, extensible and secure.

6.2.3 Software Quality

The implementation must consider issues related to software quality. As discussed briefly in section 1.1.2, software quality depends on additional to usability aspects such as reliability, maintainability, portability and extensibility.

Reliability and robustness are very important to the generic marking system and to CourseMaster in general, especially if the software is to be used in a controlled environment for formal marking. Security is also paramount. Students that may already have programming experience must not have the chance to compromise the system's integrity.

Maintainability is a key quality that touches upon all parts of the system. Thorough documentation, code commenting and aggressive avoidance of states described in literature as anti-patterns [BMM+98] are considered important factors of maintainability. Future modifications and alterations must be predicted correctly in order to make maintenance easier. The object-oriented model aids in improving maintainability but only when correct choices have been made during the architectural and design phase. Maintainability aims to contain change, therefore making software more amenable to change in the future.

Portability is necessary from a practical point of view. Academic institutions run a variety of software in mixed-hardware environments. There is a need to support a number of operating systems and hardware configurations. Academic institutions cannot be expected to invest in a specific software/hardware platform in order to run a CBA system. Re-writing the CBA system for specific software/hardware platforms is not a viable option either, as this would require considerable more coding effort and comes with the burden of supporting more versions of the same software.

Finally, the objective to turn the implementation to an infrastructure for future research and evaluation requires a high degree of extensibility. Extensibility means that extensions can be made with little difficulty and added to the implementation. Modifications and amendments are to be expected in the lifecycle of software.

6.3 Implementation Overview

This section serves as an introduction on implementation related issues. Section 6.3.1 explains why Java was chosen as the implementation language while sections 6.3.2 and 6.3.3 describe a high level view of the parts and the relationships between CourseMaster, the generic marking system and DATsys.

6.3.1 Choosing Java as the Implementation Language

The Java 2 language was chosen as the platform for the implementation of the design for many reasons. Good guides for Java have been given for basic, intermediate and advanced level by Eckel [Eb98], Gosling, Joy and Steele [GJS97], and Venners [Vb98]. According to its specification [GJS97], Java is a simple, object-oriented, distributed, interpreted, robust, secure, architectural-neutral, portable, high performance, multithreaded, and dynamic language. The implementation can take advantage of all these qualities to produce a better deliverable.

Java is a suitable base to express the design ideas presented in chapter 5. Its support for networking and distribution simplifies the implementation of CourseMaster's networking needs. Its interpreted nature gives the freedom to dynamically load and execute parts of source code, a feature that is very useful in implementing the loading of the marking scheme. Its built-in support for robustness and security is essential for all the deliverables, especially for the generic marking system.

Portability is a crucial reason for selecting Java. Java is supported on many platforms and this alleviates portability concerns. CourseMaster's implementation did come across a number of aspects of Java that are not multi-platform. Operating system specific code had to be inserted to resolve this problem.

As Java is an interpreted language, its performance suffers in contrast to compiled programs. However, with the introduction of HotSpot compilation technologies [Gd98], Java is fast approaching the performance levels exhibited by compiled applications.

6.3.2 High Level View of the Implemented Parts

The implementation of the deliverable for this thesis consists of:

- Two base platforms: DATsys and CourseMaster
- Two authoring environments: Daidalos and Ariadne
- Various clients and seven servers within CourseMaster

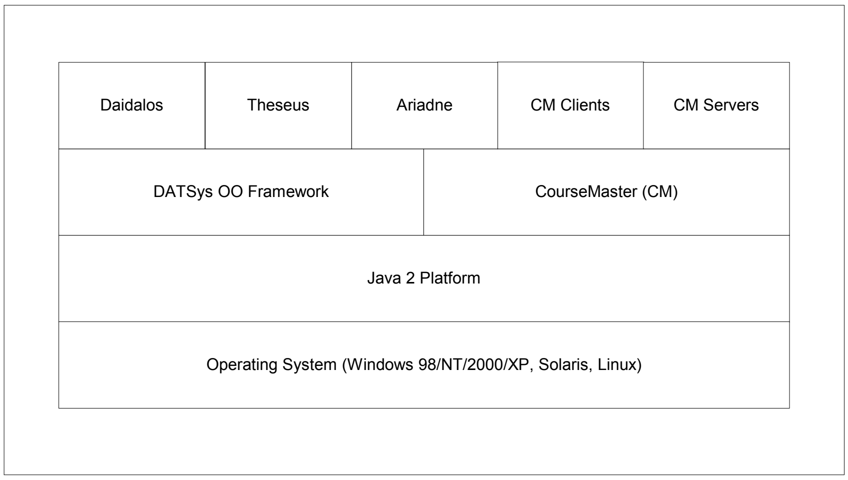


Figure 6.1: Software dependencies between parts of the implementation

Figure 6.1 illustrates a block-diagram of the relationship between these software parts. Both DATsys and CourseMaster are written exclusively in Java and have been tested under various platforms running Windows, Solaris and Linux. Daidalos and Theseus are applications based on DATsys. Ariadne is also based on DATsys, and additionally, requires parts of CourseMaster. The CourseMaster clients and servers are built on top of CourseMaster and are completely independent of DATsys.

6.3.3 High Level View of the Relationships between Parts

Figure 6.2 illustrates a simplified view of the data flow between the implemented parts. Diagram notation specifications are authored by the course developer and are expressed as domain tool libraries and application options. These are used in Ariadne by the course teacher who develops the diagram-based CBA exercise. Using Ariadne, the set of domain tool libraries and application options can be refined to address the intent of specific exercises. In addition, Ariadne produces exercise setting-up and marking files that are used by the generic marking mechanism that is contained within CourseMaster's marking server.

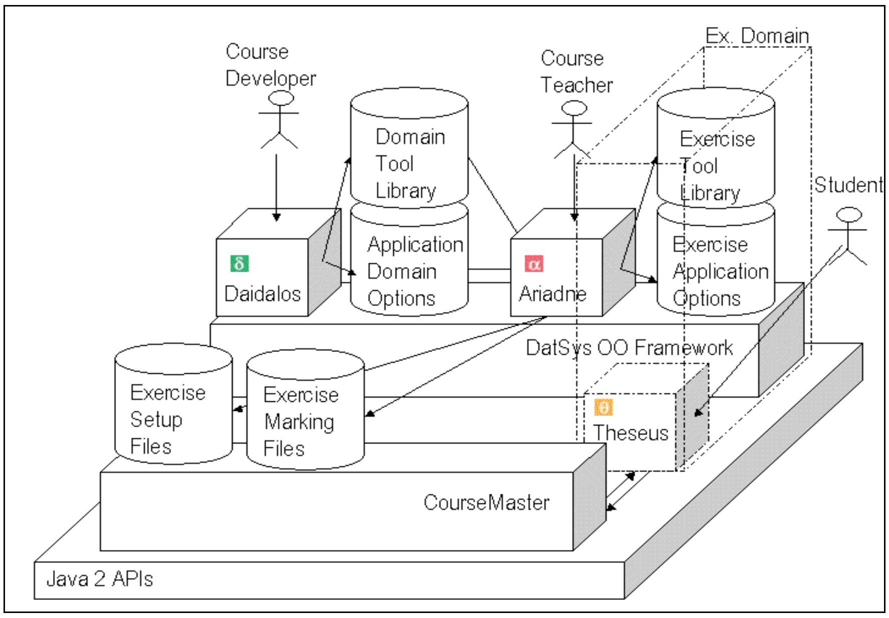


Figure 6.2: Relationship between DATsys parts

Theseus uses the tool library and the set of application options to allow the student to draw the exercise solution diagram. Upon submission, the marking server uses the marking files that have been created in Ariadne to mark and return the marking results with feedback to the student.

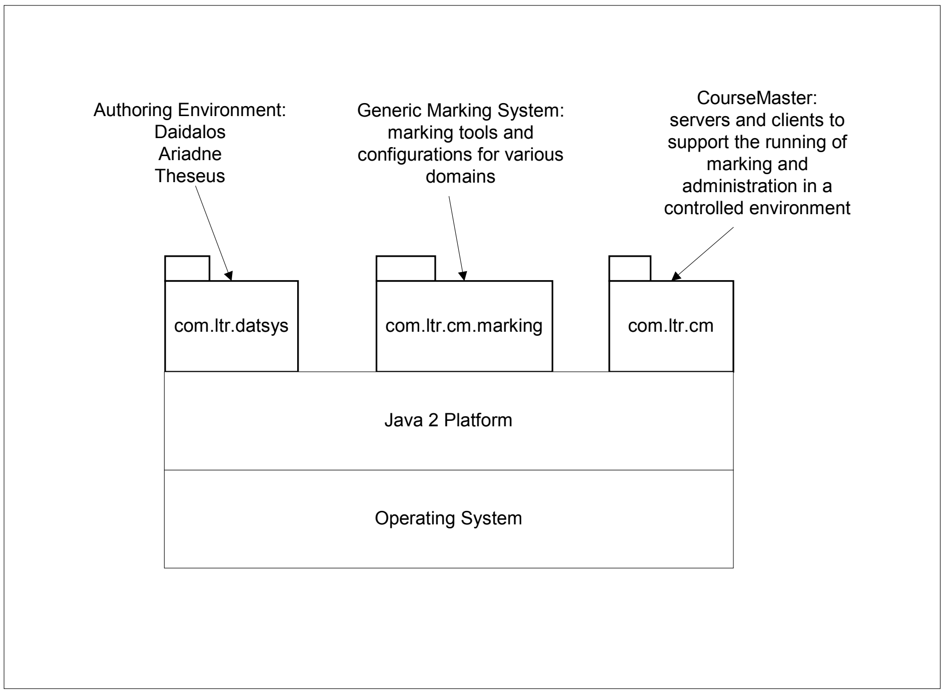


Figure 6.3: The three top-level packages that contain all the implementation

Figure 6.3 illustrates the distribution of the implementation in packages. The `com.ltr.datsys` package contains 12 packages for concepts within the DATsys framework and three packages for Daidalos, Ariadne and Theseus. The package `com.ltr.cm.marking` belongs to the CourseMaster package and contains two packages for marking commands and marking tools. The `com.ltr.cm` package contains 26 packages for the main functions, servers, clients and common objects. Packaging separates DATsys and CourseMaster so that each implementation can work independently.

6.4 The Implementation of DATsys

Each package in DATsys contains classes related to the concept that the package represents. DATsys contains 38 interfaces, 21 abstract classes and 271 concrete classes. Seven main hierarchies and several single concepts are distributed in these packages. In total, all the classes have approximately 2600 methods, and 1020 attributes. The complete source code consists of approximately 37,000 lines of code, making the ratio of code to method only 14.23 lines per method.

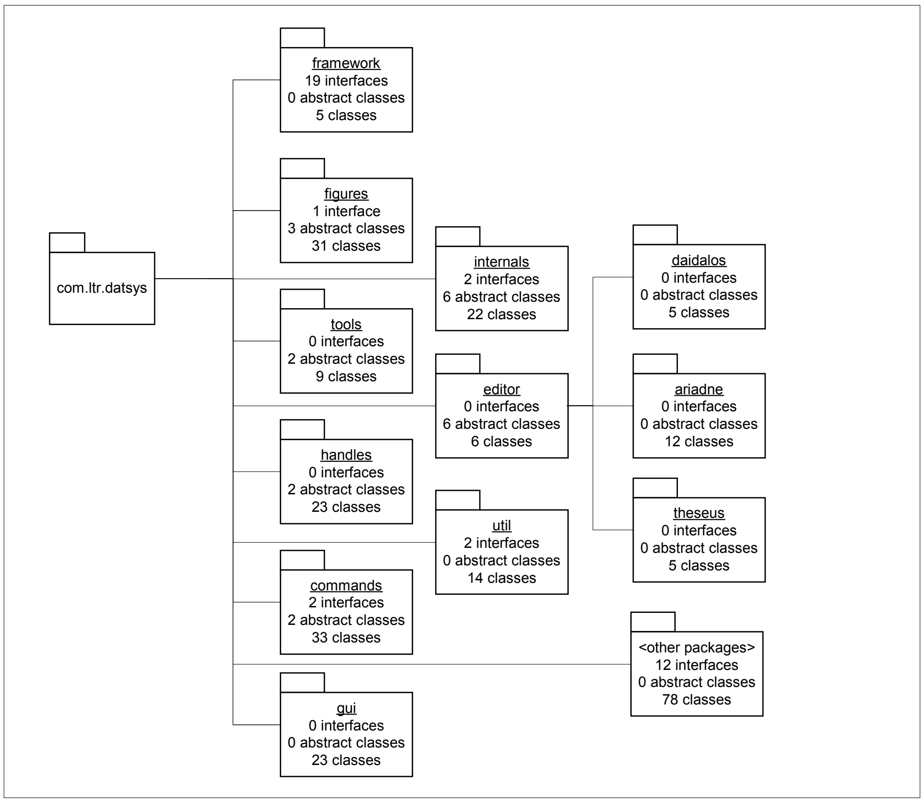


Figure 6.4: The distribution of classes to packages in DATsys

Figure 6.4 shows the main packages of DATsys. The `framework` package is the epicentre of DATsys as it contains the main interfaces that encapsulate the design of the framework.

The `figures` package contains all the classes of the figure hierarchy. In addition, it contains the classes for the connectivity and data model of diagram elements. The figure classes use the functionality supported by Java's 2D API [Sun01], [Hv99]. The 2D API contains numerous methods for drawing, colouring and painting of 2D objects such as lines, rectangles, ellipses, and so on.

The `tools` package contains the tools hierarchy as illustrated in section 5.3.4. Nine tools have been implemented for the most common functions of diagram editors.

The `handles` package contains all the implementation for the hierarchy of handles. 23 types of handles have been implemented. 16 are general handles and can be used in all the figures, while the other 7 are made for specialised figures.

The `commands` package contains classes that belong to the command hierarchy that is described in section 5.3.5. 33 concrete commands have been implemented, both internal and external. New commands can easily be created but require programming. Template commands exist for the invocation of external tools and for the construction of macro commands.

The `gui` package contains classes for specialised user interface components such as toolbars, tool library editors, selection editors, text editors, and so on. These are used within Daidalos, Ariadne and Theseus. The current implementation of the gui classes uses the JFC API [Sun98].

The `internals` package contains classes of implementations for the tool library, connectors, grids, canvases and the classes needed to add the feature of dynamic evolution.

The `editor` package contains classes for making editors. It encapsulates the abstractions needed for the editor's models and views, and associates editors to option makers. Three sub-packages of the editor package represent the two authoring environments and the students' diagram editor.

Finally, the `util` package contains helper classes and classes for general use. These are classes that load and cache bitmap files and sounds, perform geometrical calculations, load, save and convert diagram structures, read configurations to implement registries and convert between colour models.

6.4.1 Daidalos

The implementation for Daidalos adds 15 classes to the DATsys framework. Most of the additional classes define the implementations of its graphical user interface components.

Figure 6.5 illustrates Daidalos' input and output. When Daidalos starts, it loads a configuration file that describes properties for execution paths, properties for the graphical view, and other parameters. This configuration can be edited manually or automatically by changing values within Daidalos. One of the parameters represents the execution mode of Daidalos that can be set to "reset", "normal" and "evolve". Depending on Daidalos' execution mode, Daidalos presents different options to its users. While in "reset" mode, Daidalos puts default values in its existing configurations and negates the effects of any extensions made. In "normal" mode, Daidalos executes normally, and stores the changes made to make these available at the next execution. In "evolve" mode, Daidalos presents the user with features to develop predefined parts of itself. This experimental feature is described in more detail in section 8.4.3.

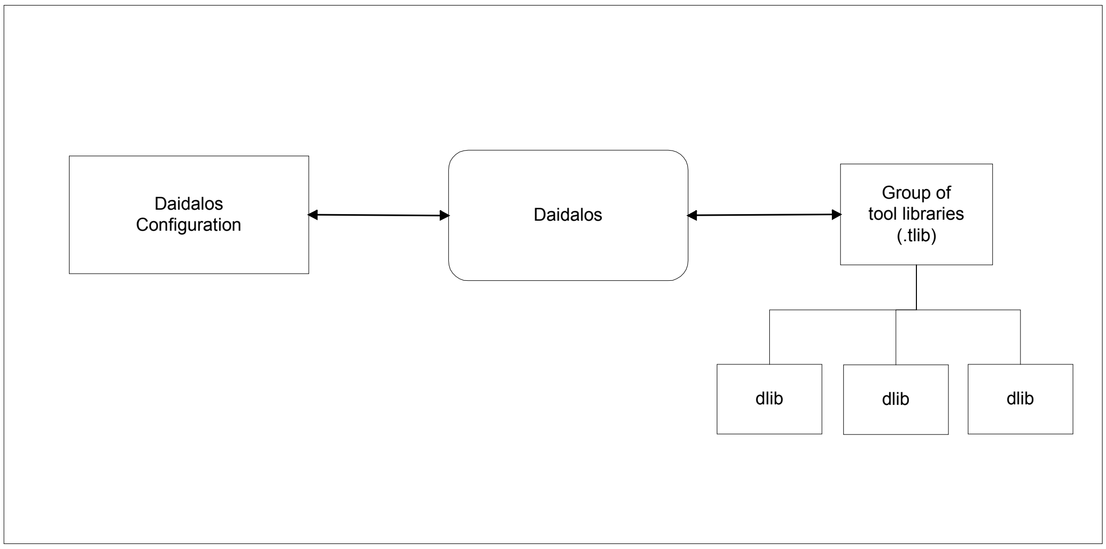


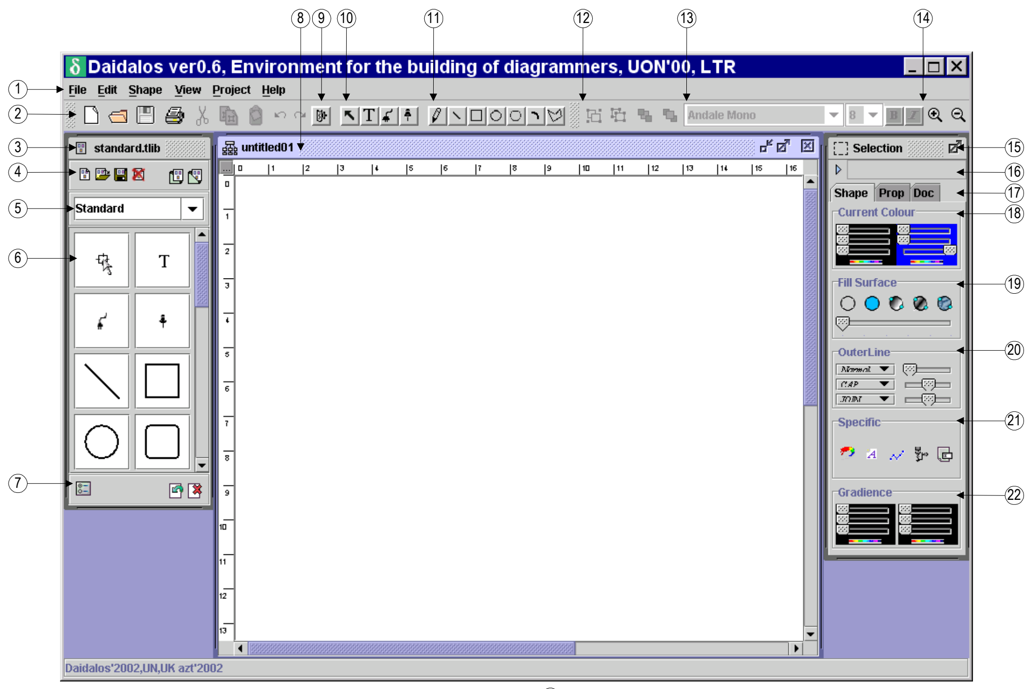
Figure 6.5: Daidalos and I/O

Users of Daidalos develop a tool library which is saved in a file that has a "`.dlib`" extension. Tool libraries are designed to contain tools that have been customised to suit specific graphical notations. As more than one notation often exists for a type of diagram, tool libraries can be grouped. A group is meant to represent a diagrammatic domain.

Daidalos' interface presents the user with three main windows for:

- Tool library management
- Interactive diagram element creation and editing (on the canvas)
- Selection editing

Figure 6.6 illustrates a view of Daidalos and describes all its associated options. The tool library window allows the organisation of tools into tool libraries and these into groups of tool libraries. Supported functions include loading and saving libraries, and adding and removing tools. The "Add Tool" button creates a new tool by using the figures that are currently selected. Depending on the selection, Daidalos interprets the type and configuration of the tool to be created. Before adding a new tool, the selection must contain a valid specification for a diagram element. The specification is visual, and consists of the graphical appearance of the diagram element, its data model and its configuration for connectivity to other elements.



① Menu Options
② Buttons: new, load, save, print, cut, copy, paste, undo, redo
③ Current group of tool libraries
④ Tool library buttons: new, load, save, delete
⑤ Choice for selecting a tool library that belongs to the group
⑥ Tool buttons for the current tool library
⑦ Buttons for tool properties, making, and removing a tool
⑧ Diagram canvas for creating diagram tools
⑨ Selection mode - sticky tool on/off
⑩ Prototype tools: select, text, connect, place connection pin
⑪ Scribble, line, rectangle, round rectangle, ellipsis, arc, polygon
⑫ Selection buttons: group, ungroup, send to front, send to back
⑬ Choice for type of font, size, bold, italic
⑭ Zoom-in, zoom-out
⑮ Selection editor
⑯ Name of the selected elements
⑰ Tab dialog for selection editor, properties, documentation
⑱ Foreground and bacground colour chooser
⑲ Shape surface: outline, colour, gradient, cyclic grafient, image
⑳ Shape outerline: width, dotted line parameters, joint connection
㉑ Colour palette, font, performance, execution, configuration
㉒ Foreground and bacground colour gradience chooser

Figure 6.6: Daidalos' map of features

All three aspects of a diagram element are described interactively. The graphical view is drawn using primitive figures. The data model is specified by adding typed data fields. The connectivity is specified by either choosing perimeter-based connections or pin-based connections. The appearance and type of connection lines can be further configured by selecting appropriate options.

The canvas for designing diagram elements allows the direct manipulation of figures. The prototype tools toolbar contains the primitive tools with which diagram elements are constructed. Primitive tools are used to specify diagram elements on the canvas. The selection editor window allows editing attributes for the elements that are currently selected. The available attributes include colours, transparency levels, gradient colour fill types, textures, and so on. The attributes mirror the graphical features found in Java's 2D API. The selection editor also provides an option for naming the selected element. This is imperative to ease the identification of elements during marking. Daidalos supports single and multiple selection, multi-level undo and grouping, zooming, Z-order layering, axis alignment, grids, and multiple fonts.

6.4.3 Ariadne

The implementation for Ariadne adds 7 classes to the DATsys framework. These added classes cater for gui components that allow editing CourseMaster exercise configuration files.

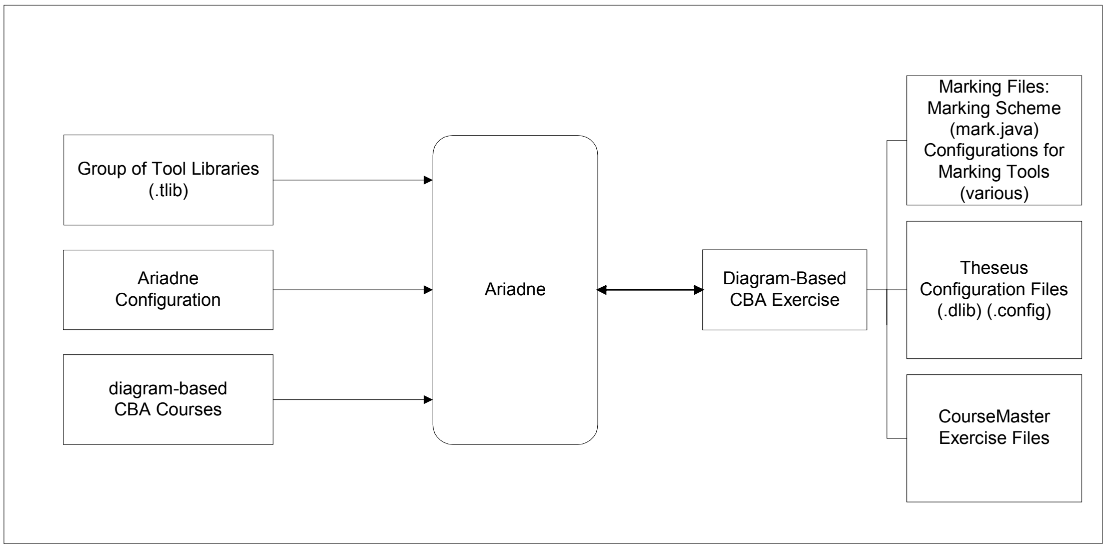

Group of Tool Libraries (.tlib)
Ariadne Configuration
diagram-based CBA Courses
Ariadne
Diagram-Based CBA Exercise
Marking Files: Marking Scheme (mark.java) Configurations for Marking Tools (various)
Theseus Configuration Files (.dlib) (.config)
CourseMaster Exercise Files

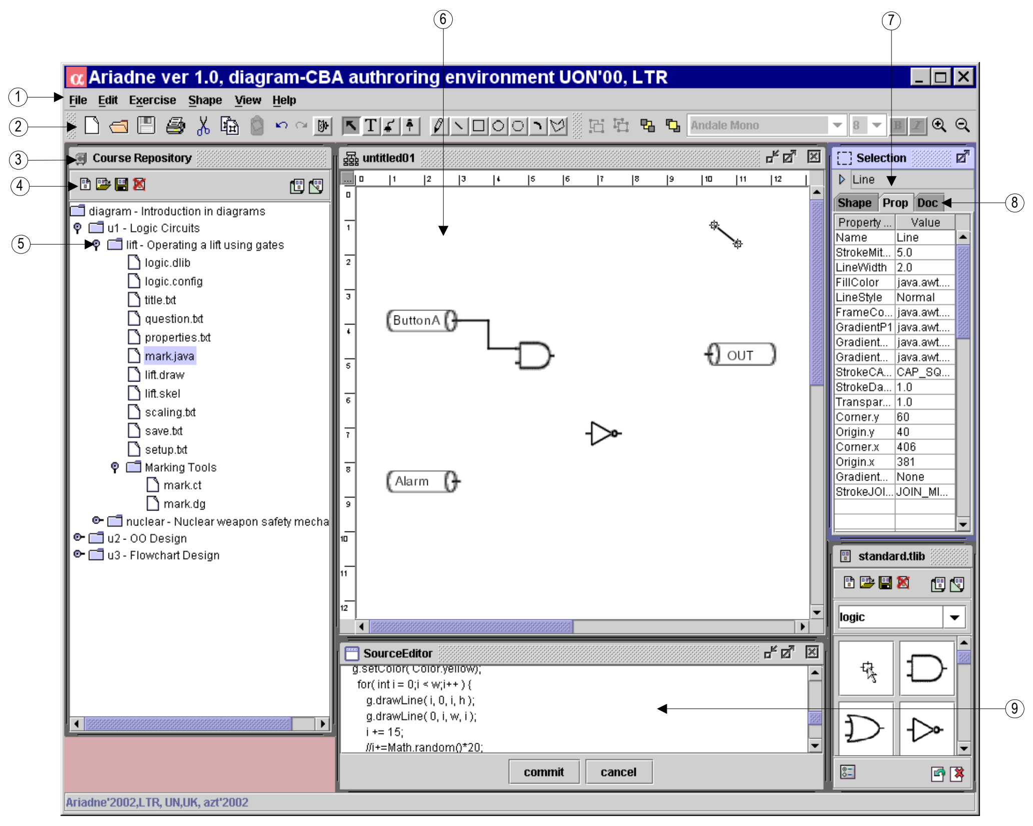
Figure 6.7 Ariadne and I/O

Figure 6.7 illustrates Ariadne's input and output. When Ariadne runs, it loads a configuration that contains properties similar to those in Daidalos. However, in contrast to Daidalos, Ariadne does not offer different types of execution modes.

Ariadne loads a predetermined or default group of tool libraries and the existing diagram-based CBA exercises. These exercises may have been already deployed in CourseMaster, in which case Ariadne loads them from the course area. For development and testing purposes, the exercises can also reside in a local directory.

Ariadne's users are teachers who aim to create and administer diagram-based CBA. To accomplish this, the output of Ariadne for a single exercise must consist of:

- An exercise specific tool library and application configuration file
- A marking scheme and configuration for the marking tools that are invoked
- Configuration for the CBA exercise



Property ...	Value
Name	Line
StrokeMit...	5.0
LineWidth	2.0
FillColor	java.awt....
LineStyle	Normal
FrameCo...	java.awt....
GradientP1	java.awt....
Gradient...	java.awt....
Gradient...	java.awt....
StrokeCA...	CAP_SQ...
StrokeDa...	1.0
Transpar...	1.0
Corner.y	60
Origin.y	40
Corner.x	406
Origin.x	381
Gradient...	None
StrokeJOI...	JOIN_MI...

```
g.setColor( Color.yellow);
for( int i = 0;i < w;i++ ) {
  g.drawLine( i, 0, i, h );
  g.drawLine( 0, i, w, i );
  i += 15;
  //i+=Math.random()*20;
```

1	Menu Options	6	Diagram canvas for exercise skeleton and soluiton files
2	Toolbar with same options as Daidalos	7	Selection property editor
3	Tree for course repository	8	Selection documentation editor
4	Exercise buttons: new, load, save, delete, load all, save all	9	Text editor for exercise text-files
5	A diagram-based CBA exercise		

Figure 6.8: Ariadne's map of features

Figure 6.8 illustrates a view of Ariadne and describes all its associated options. Ariadne contains most of the editing features of Daidalos. In addition, it contains a repository management window to manage the files that belong to a diagram-based CBA exercise.

For each of the configuration files, Ariadne opens an appropriate editor. For the description of the marking scheme, Ariadne offers the generation of the source code by using wizards, and a simple editor that provides compilation and testing features.

6.4.4 Theseus

Theseus is a generic template for diagram editors that accepts tool libraries and other configuration, and becomes a specialised diagram editor. The implementation for Theseus adds 3 classes to the DATsys framework. Those classes are useful in configuring Theseus to an external tool library and an "application option configuration" file.

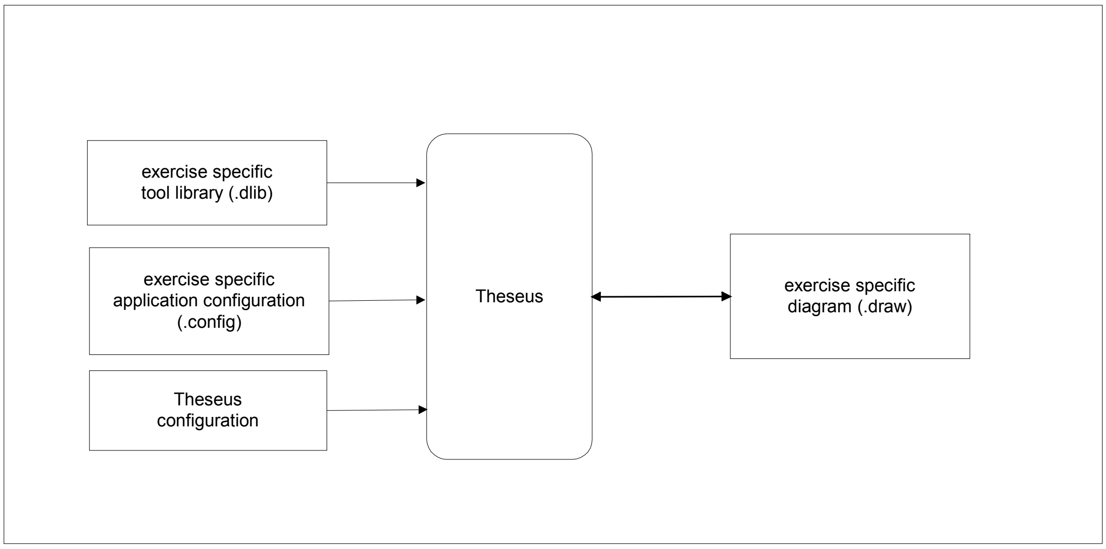
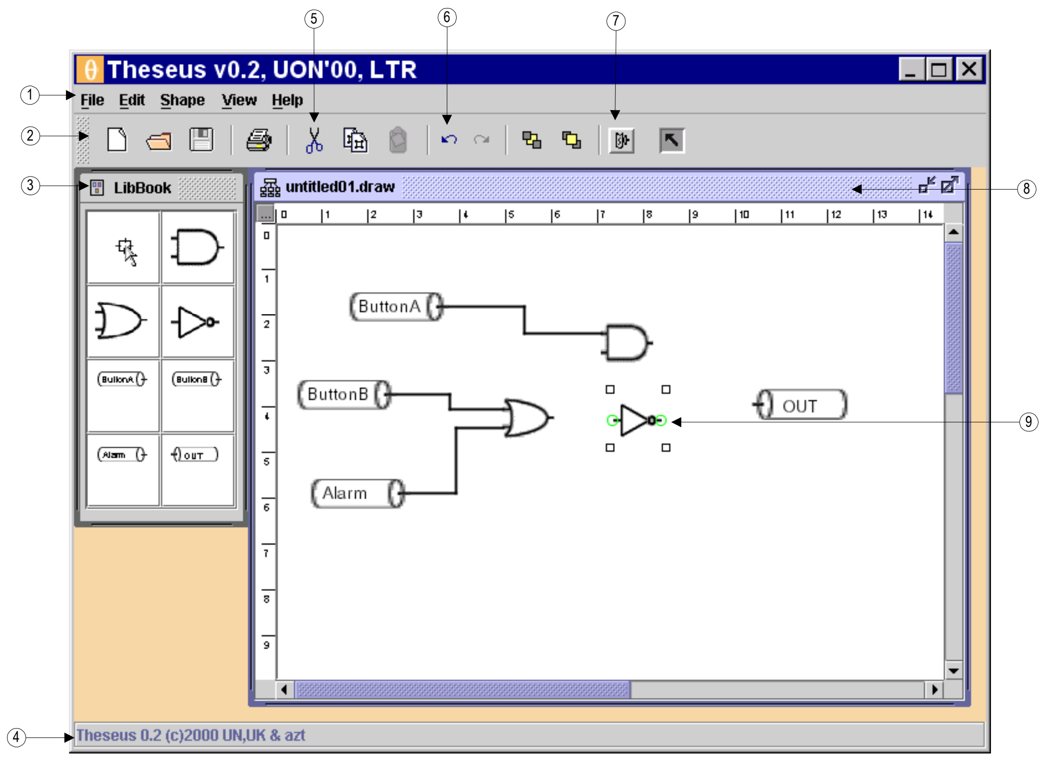

exercise specific tool library (.dlib)
exercise specific application configuration (.config)
Theseus configuration
Theseus
exercise specific diagram (.draw)

Figure 6.9: Theseus and I/O

Figure 6.9 illustrates Theseus' input and output. Upon execution, Theseus loads three configuration files. Firstly, Theseus needs the exercise specific tool library. This provides Theseus with the necessary tools that are placed on its toolbar. Secondly, Theseus needs an application option configuration file that describes the available options. Finally, Theseus needs a configuration for its general execution parameters, styles, paths, and so on.

Theseus' users are the students that have to draw the exercise solution. Theseus is customised to the requirements of the CBA exercise and it allows the creation of diagrams in a simple and effective manner. Theseus' output is a file with a "`.draw`" extension. The information within this file represents all the elements of the diagram and their attributes.

Upon completion, the students can press the "submit" button on their CourseMaster clients. The clients send a submission object to the submission CourseMaster server, which is delegated to the marking subsystem. This in turn executes the appropriate diagrammatic marking tool that examines the student's solution and returns marking results and feedback.



1. Menu Options
2. Diagram buttons: new, load, save, print
3. Available tools for the domain specific diagram
4. Statusbar for messages
5. Selection butons, cut, copy paste
6. Buttons: undo, redo, send to front, send to back
7. Buttons: selection sticky, select
8. Diagram canvas that allows the development of the exercise solution. The title takes the name of the exercise
9. Connectivity handle

Figure 6.10: Theseus' map of features

Figure 6.10 illustrates a view of Theseus for a sample exercise in logic design and describes all the associated options. Theseus contains only a subset of the editing features of Daidalos.

6.5 The Implementation of the Generic Marking System

The generic marking system consists of 3 packages that are part of the CourseMaster marking server. It contains 5 interfaces, 6 abstract classes and 36 classes.

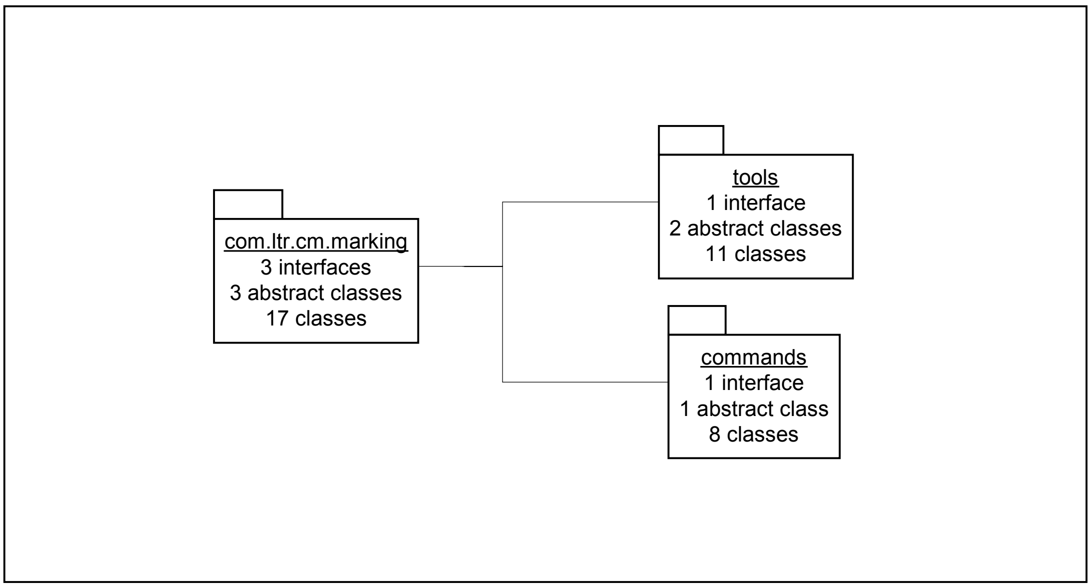


Figure 6.11: Package distribution of the generic marking mechanism

Figure 6.11 illustrates the main packages of the generic marking system.

The `marking` package contains the classes described in the design in section 5.4. It contains the abstractions for the marking scheme, the hierarchy of the marking result, the marking area and other helper classes.

The `commands` package contains a hierarchy of commands that adapt the interface of the marking tools to CourseMaster. The marking command hierarchy mirrors the hierarchy of the marking tools. The idea of adding a parallel hierarchy of adapting commands to the marking tools decouples the marking tools from CourseMaster, so that marking tools can be reusable.

The `tools` package contains the hierarchy of marking tools. Eight marking tools have been implemented in total for programming and diagramming courses. A number of marking tools are planned for implementation in the future. Section 7.2.5 discusses the design for more than 10 additional marking tools that will increase the number of supported diagram domains.

6.5.1 Marking Scheme

As explained in section 5.4.2, the marking scheme describes the marking invocations of a CBA exercise.

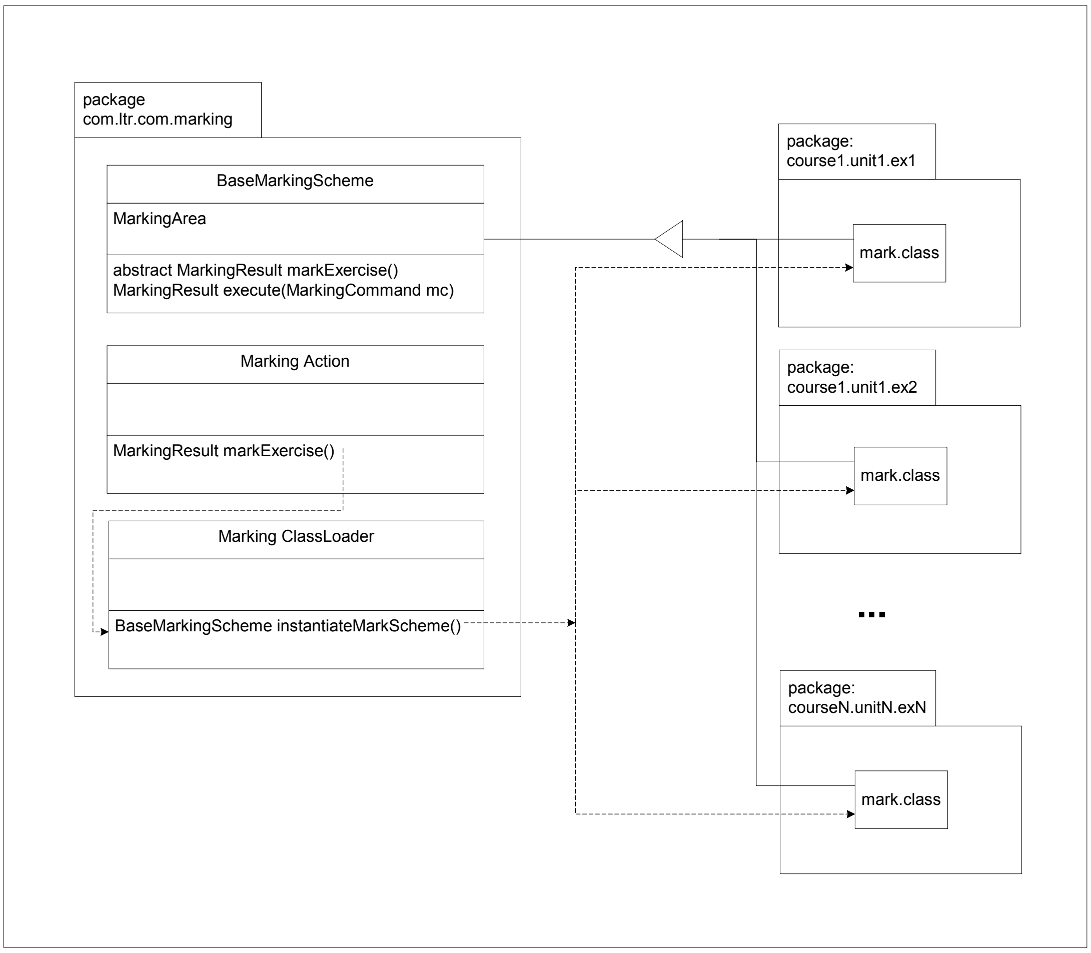
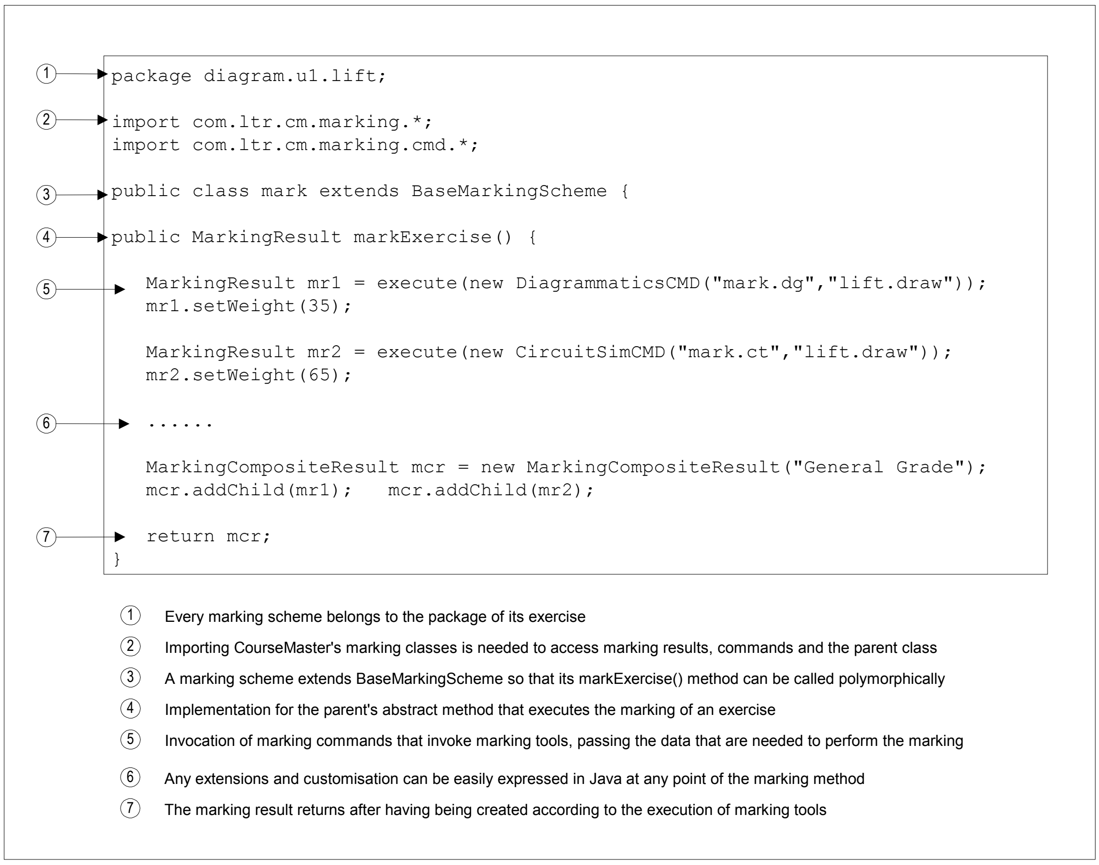


Figure 6.12: Every exercise maintains a marking scheme in a unique package

Marking schemes are implemented as Java program files. All automatically assessable exercises must contain a single source file named "`mark.java`". These files contain statements that instruct the marking subsystem on which actions to take in order to mark the students' solutions. As each exercise resides in its respective directory under CourseMaster's marking area, the individual marking scheme files must use Java's package statement to allow them to be separately loaded. Marking schemes are instantiated by a marking action and through a customised classloader. As Java's package statements have to be valid Java identifiers, packages (and thus, exercise directories) are not allowed to begin with numbers.

Figure 6.12 illustrates the implementation view of the marking scheme as it relates to its package. It also shows a simplified perspective of the instantiation of marking schemes using classloaders. Figure 6.13 illustrates an example of a marking scheme implemented for a diagram-based exercise. It shows in practice how the marking scheme relates to CourseMaster, how the invocation of marking commands commences and how marking results are created, composed and returned.



```
package diagram.u1.lift;

import com.ltr.cm.marking.*;
import com.ltr.cm.marking.cmd.*;

public class mark extends BaseMarkingScheme {

public MarkingResult markExercise() {

   MarkingResult mr1 = execute(new DiagrammaticsCMD("mark.dg","lift.draw"));
   mr1.setWeight(35);

   MarkingResult mr2 = execute(new CircuitSimCMD("mark.ct","lift.draw"));
   mr2.setWeight(65);

   ......

   MarkingCompositeResult mcr = new MarkingCompositeResult("General Grade");
   mcr.addChild(mr1);   mcr.addChild(mr2);

   return mcr;
}
```

1. Every marking scheme belongs to the package of its exercise
2. Importing CourseMaster's marking classes is needed to access marking results, commands and the parent class
3. A marking scheme extends BaseMarkingScheme so that its markExercise() method can be called polymorphically
4. Implementation for the parent's abstract method that executes the marking of an exercise
5. Invocation of marking commands that invoke marking tools, passing the data that are needed to perform the marking
6. Any extensions and customisation can be easily expressed in Java at any point of the marking method
7. The marking result returns after having being created according to the execution of marking tools

Figure 6.13: A simple example of an implementation of a marking scheme

A limitation that is initially not obvious is that corrections to an exercise's marking scheme can't be made at runtime. As Java normally uses a default classloader that caches the loaded classes, it is impossible to update class-definitions at runtime. This means that on any update of the marking scheme, the marking server needs to be restarted, so that the new class definition can be loaded. This problem is rectified with the use of custom classloaders. Custom classloaders are also used in Daidalos' support for runtime evolution for the same reason. Venners has given an in-depth review for the inner-workings of Java classloaders in [Vb98].

6.5.2 Diagram-Based Marking Tools

Diagram-based marking tools have access to a diagram object that represents a student's solution. In addition to general querying services, the diagram object also provides specific implementation features that are useful when translating the diagram into other structures.

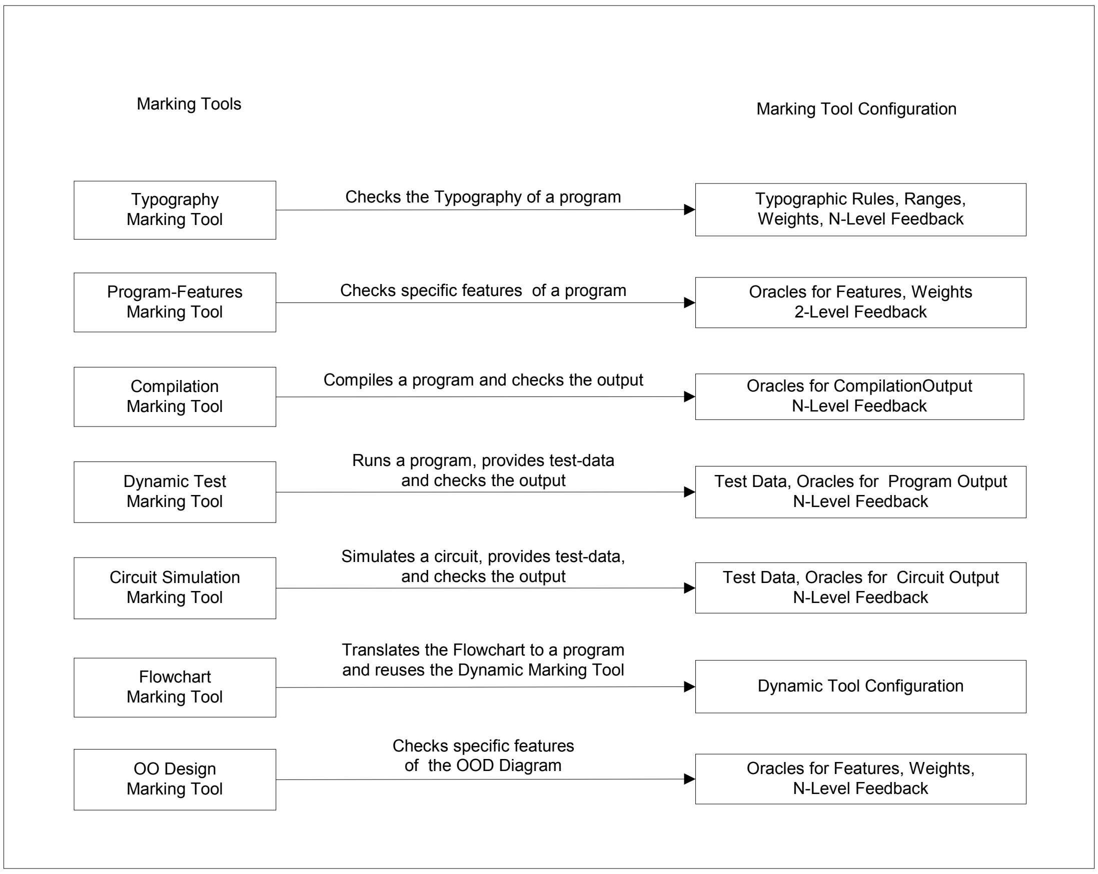

Marking Tools
Marking Tool Configuration
Typography Marking Tool
Checks the Typography of a program
Typographic Rules, Ranges, Weights, N-Level Feedback
Program-Features Marking Tool
Checks specific features of a program
Oracles for Features, Weights 2-Level Feedback
Compilation Marking Tool
Compiles a program and checks the output
Oracles for CompilationOutput N-Level Feedback
Dynamic Test Marking Tool
Runs a program, provides test-data and checks the output
Test Data, Oracles for Program Output N-Level Feedback
Circuit Simulation Marking Tool
Simulates a circuit, provides test-data, and checks the output
Test Data, Oracles for Circuit Output N-Level Feedback
Flowchart Marking Tool
Translates the Flowchart to a program and reuses the Dynamic Marking Tool
Dynamic Tool Configuration
OO Design Marking Tool
Checks specific features of the OOD Diagram
Oracles for Features, Weights, N-Level Feedback

Figure 6.14: Marking tools and their configuration

Figure 6.14 illustrates a range of programming and diagramming marking tools in relationship to their configuration. The typography, program-features, compilation and dynamic test tools all have the same configuration as in Ceilidh.

Three diagram-based marking tools have been implemented:

- `CircuitSimMarkingTool`: marks logic design based exercises
- `FlowchartMarkingTool`: marks flowchart based exercises
- `OOMarkingTool`: marks object-oriented design based exercises

The circuit-simulator marking tool needs as configuration:

- Test data that will be fed to the inputs of the circuit for the simulation,
- Oracles to describe the correct output values or ranges of values per test-data,
- Feedback messages for all testing cases.

The flowchart-diagram marking tool, after translating the diagram to a program, reuses the dynamic-test marking tool. Therefore, it needs the configuration of a dynamic- test tool.

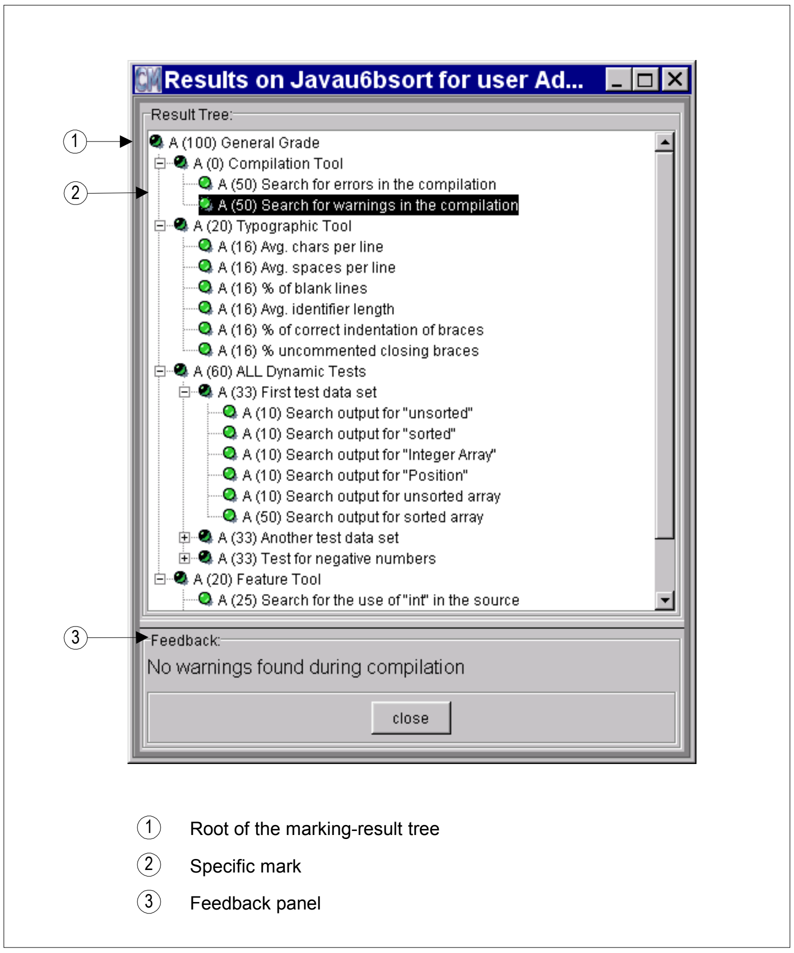
The object-oriented-design marking tool, investigates the student diagram for particular features such as the use of specific relationships between predefined classes and objects, the naming of classes and methods, and so on.

6.5.3 Marking Feedback

The configuration for the marking tools contains the marking feedback. All marking tools return a tree of marking results where each node contains appropriate feedback.

The marking results may need to be viewed using different marking styles. The rendering of the marks to the student GUI is the responsibility of the object `GradingStyle` that is associated with the marking result returned by the submission server in accordance with properties set within exercise properties. Students browse the resulting tree of marks and identify problematic areas within their solution while receiving comments on their submitted work.

Figure 6.15, illustrates the expandable tree of marking results and feedback as students receive it after a submission. The root node of the tree contains the overall mark of the student's submission. Each node in the marking tree represents the respective marking tool that has processed the student's submission. Each sub-node contains tool-specific information on the student's submission. Underneath the tree, the feedback panel informs the students on their work in detail.



```
Results on Javau6bsort for user Ad...
Result Tree:
A (100) General Grade
  A (0) Compilation Tool
    A (50) Search for errors in the compilation
    A (50) Search for warnings in the compilation
  A (20) Typographic Tool
    A (16) Avg. chars per line
    A (16) Avg. spaces per line
    A (16) % of blank lines
    A (16) Avg. identifier length
    A (16) % of correct indentation of braces
    A (16) % uncommented closing braces
  A (60) ALL Dynamic Tests
    A (33) First test data set
      A (10) Search output for "unsorted"
      A (10) Search output for "sorted"
      A (10) Search output for "Integer Array"
      A (10) Search output for "Position"
      A (10) Search output for unsorted array
      A (50) Search output for sorted array
    A (33) Another test data set
    A (33) Test for negative numbers
  A (20) Feature Tool
    A (25) Search for the use of "int" in the source
Feedback:
No warnings found during compilation
```

Figure 6.15: The tree components that students see for the marking results

The tree component follows a specific format. Each node in the tree contains three fields:

- A bitmap icon
- The grade/mark awarded by the specific test
- The name and description of the node

The grading style that is typically used to illustrate the results to students converts the marks to a customisable range of letters and bitmap icons. More grading styles are provided in CourseMaster for some of the marking policies that exist across academic institutions. The bitmap icon depends on the mark awarded by the specific test, and is customised per course by the course developer. CourseMaster uses a ball-shape bitmap, the colour of which represents the mark, making the result aesthetically pleasing to the student.

6.6 The Implementation of the CourseMaster CBA System

CourseMaster replaced Ceilidh's 70 tools with an object-oriented architecture that comprises of 29 interfaces, 28 abstract classes, and 279 classes contained in 23 packages. Six packages contain client-related classes, four contain common classes between clients and servers and the remaining packages contain server related classes. In total, all the classes have approximately 1300 attributes and 3350 methods. The complete source code consists approximately 44,000 lines of code, making the ratio of code to method 13.3 lines per method.

Figure 6.16 illustrates a view of all the packages within CourseMaster. The packages `archiving`, `login`, `audit`, `submit`, and `marking`, contain classes for respective servers. The Ceilidh server resides inside the server package. The course server is placed inside the modules package.

The `setup` package contains classes for the packaging of the exercise files that are sent to students upon exercise setup. Depending on the project type, the classes that gather the exercise files can be configured to use pre-processors. Pre-processors can make customisations to the exercise files before sending them to a student.

The `utils` package contains helper classes that deal with file manipulation, the parsing of configuration files, and so on.

The `debug` package contains classes that help with debugging the CourseMaster servers. A debug mode can be selected during the servers execution from the servers' options. It presents helpful on-screen information for administrators and developers on the internal and external state of the servers.

The `common` and `common.project` packages contain common classes that are needed both by the CourseMaster servers and the clients. Specifically, the `common.project` package contains a class hierarchy for types of student projects. A student project encapsulates a piece of coursework, contains the collection of student and exercise files and accepts configurations that are appropriate to the type of coursework.

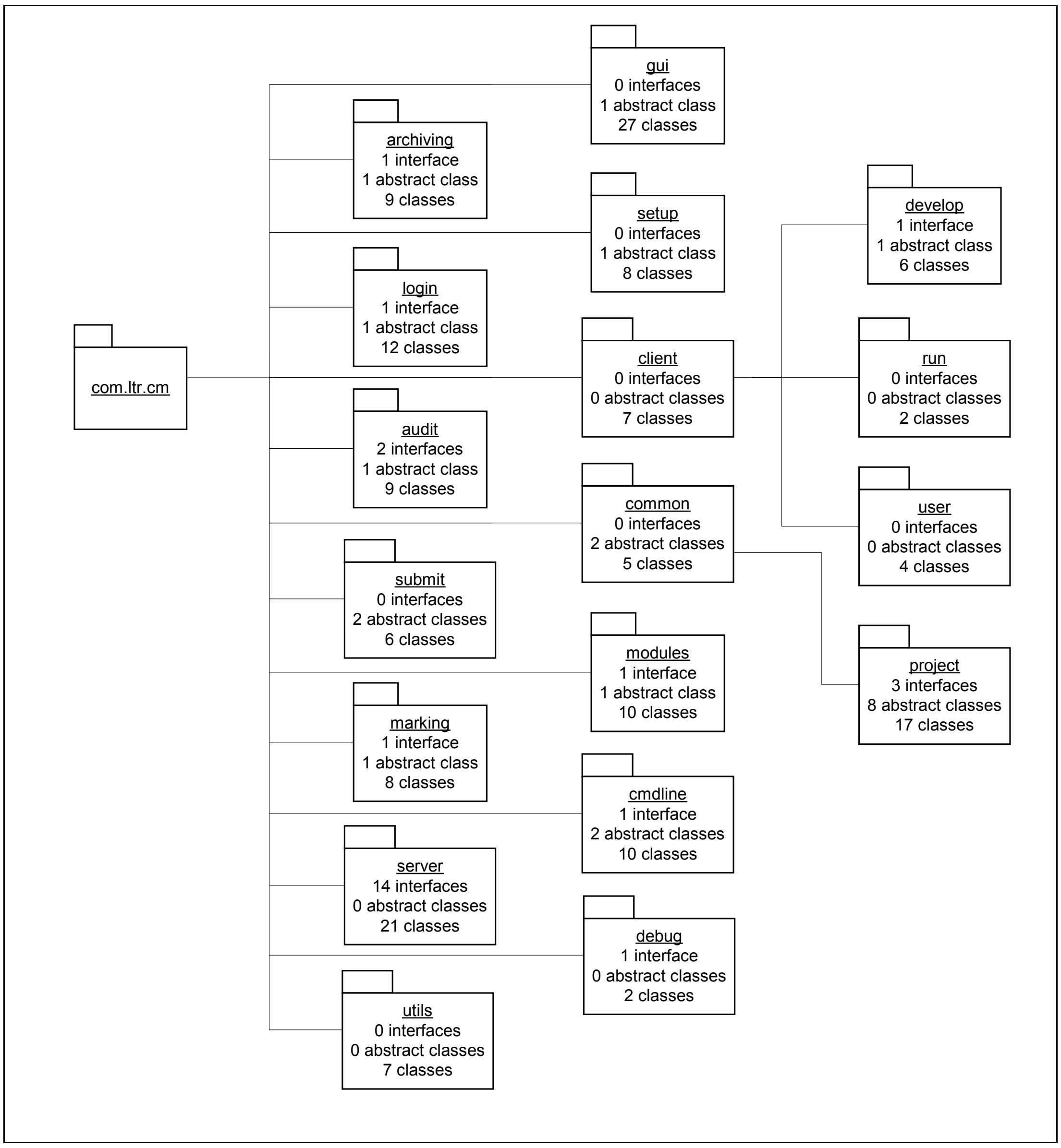


Figure 6.16: CourseMaster view of packages

The `client`, `client.develop`, `client.run` and `client.user` packages are used exclusively by the CourseMaster clients. They contain the necessary classes, abstractions, logic, and helper methods to provide the functionality of the CourseMaster clients. Specifically, the `client.develop` package deals with the configuration and use of various development environments. The `client.run` package allows for the running of student solutions (if they can be executed) by providing wrappers for all CourseMaster supported operating systems. The `client.user` package contains user-related classes that are providing the client-side of student authentication, password validation and client session information.

The `gui`, and `cmdline` packages are also used by the CourseMaster clients in their entirety. Both contain classes for providing an interface to the users of CourseMaster.

6.6.1 CourseMaster Servers

An in-depth design phase and a comprehensive architecture simplified the implementation of the CourseMaster servers. Because appropriate extension points were identified during the initial design phase, new functionality can be suitably inserted into the servers.

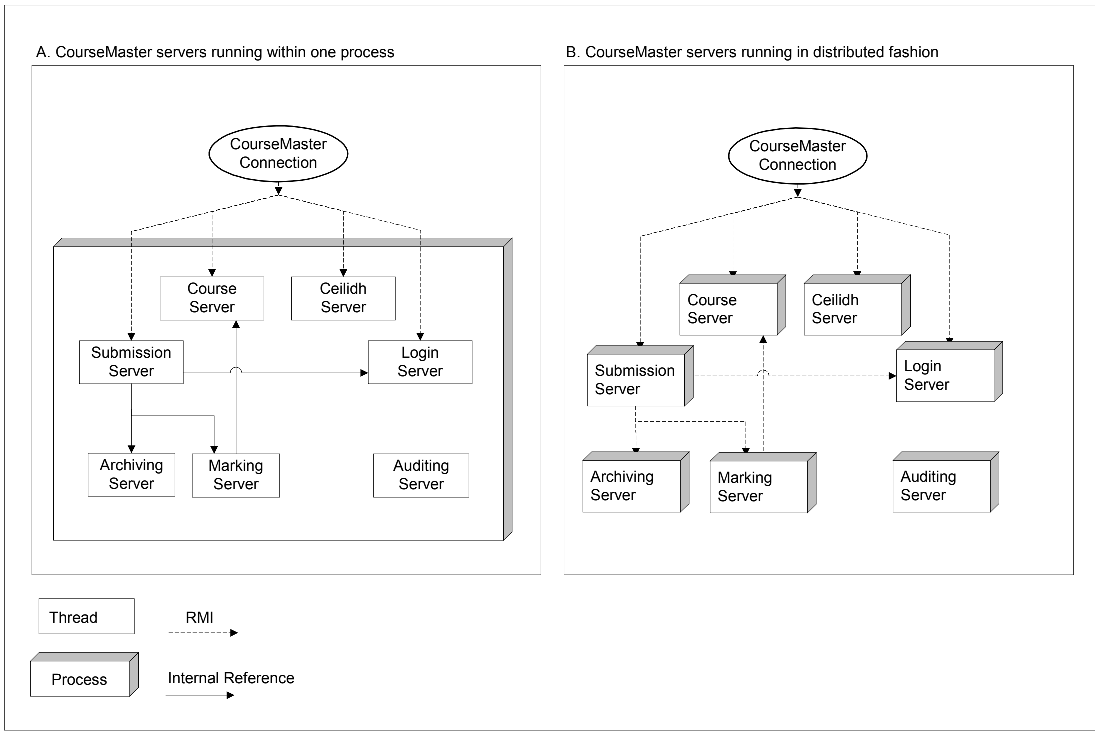
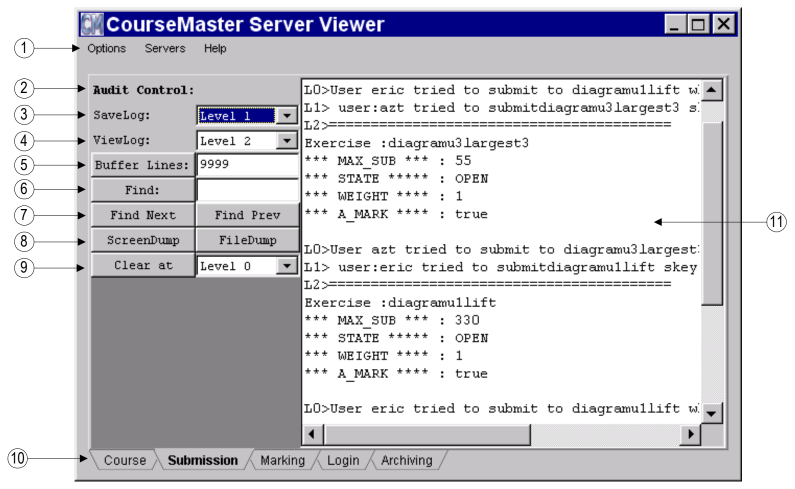


Figure 6.17: Two scenarios for the deployment of CourseMaster servers

The servers use RMI for the distribution. For this reason, all the common objects must support serialisation. It should be noted that RMI is currently not used for communication between servers, as the physical segmentation of servers has not yet been necessary. However, this can be achieved by providing a remote interface and appropriate implementation for communicating using RMI with other servers. Figure

6.17 illustrates two possible scenarios for the deployment of CourseMaster servers with either one central process or one process for each server.



```
Audit Control:
SaveLog:       Level 1
ViewLog:       Level 2
Buffer Lines:  9999
Find:
Clear at       Level 0
```

```
L0>User eric tried to submit to diagramu1lift w
L1> user:azt tried to submitdiagramu3largest3 s
L2>========================================
Exercise :diagramu3largest3
*** MAX_SUB *** : 55
*** STATE ***** : OPEN
*** WEIGHT **** : 1
*** A_MARK **** : true

L0>User azt tried to submit to diagramu3largest
L1> user:eric tried to submitdiagramu1lift skey
L2>========================================
Exercise :diagramu1lift
*** MAX_SUB *** : 330
*** STATE ***** : OPEN
*** WEIGHT **** : 1
*** A_MARK **** : true

L0>User eric tried to submit to diagramu1lift w
```

1	Menu options
2	Audit control buttons
3	Selection of level of logging to file - Four levels are available
4	Selection of level of logging to the screen buffer
5	Number of available lines for the screen buffer (10)
6	Textfield for finding a keyword
7	Buttons for finding next and previous occurences of a keyword
8	Buttons for printing and saving data to file
9	Options to filter the data of the screen buffer to a level of auditing detail
10	Tab dialog with options for each of the servers
11	The screen buffer rendered on a text component

Figure 6.18: A CourseMaster monitoring client for teachers

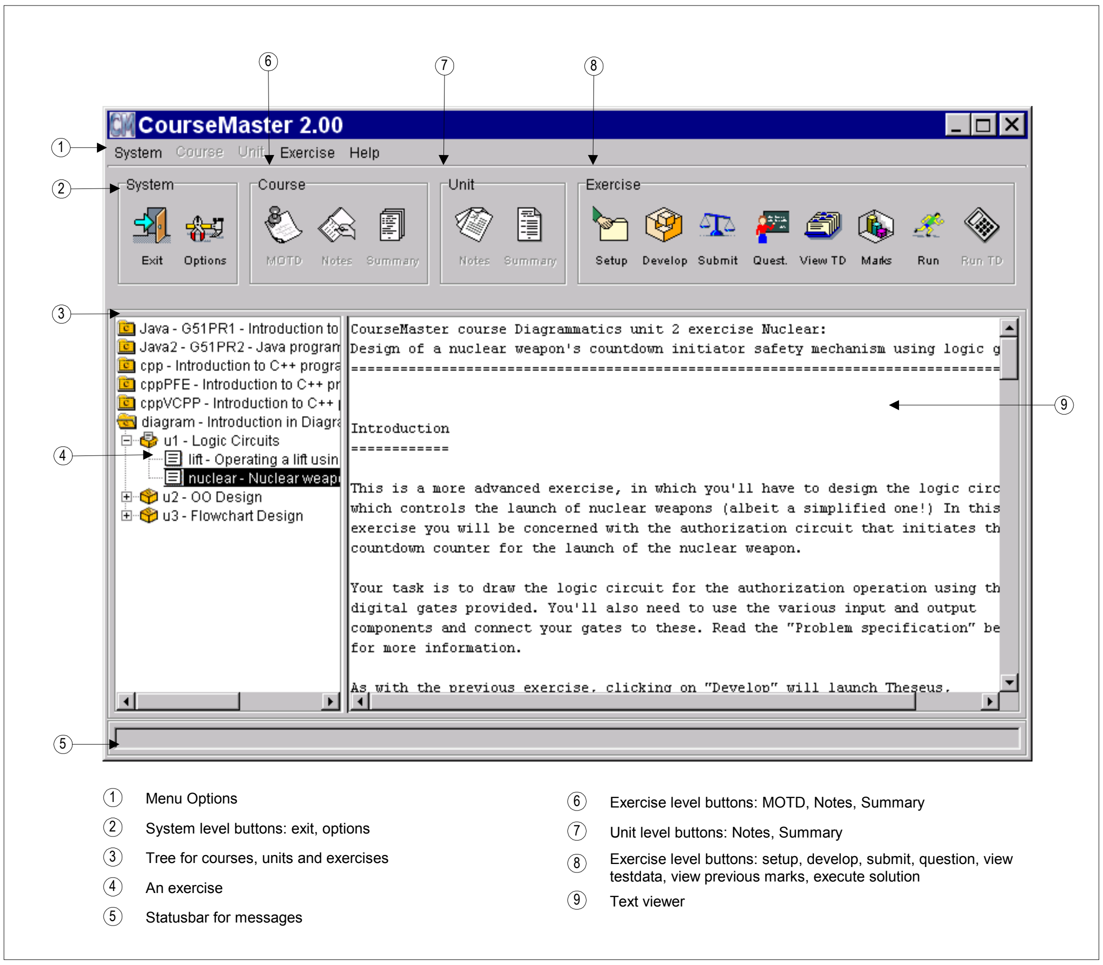
Figure 6.18 illustrates a monitoring client for administrators and teachers that displays CourseMaster's auditing information in real-time. In combination with CourseMaster options for statistics and debugging the monitoring client is invaluable in troubleshooting problems in complex networks.

The CourseMaster servers take advantage of Java's exception handling mechanism to improve their robustness. Typed exceptions represent all the possible exceptions and errors that may be encountered during CourseMaster's execution. When an error condition occurs, the error or exception propagates through the chain of invocation in order to reach the students' client. Exception masking is used throughout the propagation of exceptions between servers. For example, if an error occurs at the marking stage, a `MarkingException` is spawned by the marking server and is propagated to the submission server as a `SubmissionException`. The exception finally reaches the students' client in a GUI dialog box.

6.6.2 CourseMaster Clients

The view of the client is configuration dependent. Three different views have been written for the CourseMaster clients. CourseMaster's `gui` package contains both an AWT and a JFC based client. These are GUI clients and take advantage of Java's multiplatform capabilities. The JFC-based client can be configured to accept the look and feel of a native Java, Windows or Unix/Motif program.

The `cmdline` package contains a client that is based only on text and is geared towards graphical-less terminals such as DOS prompts and VT100 Unix terminals.


CourseMaster 2.00
System
Course
Unit
Exercise
Help
System
Exit
Options
Course
MOTD
Notes
Summary
Unit
Notes
Summary
Exercise
Setup
Develop
Submit
Quest.
View TD
Marks
Run
Run TD
Java - G51PR1 - Introduction to
Java2 - G51PR2 - Java program
cpp - Introduction to C++ progra
cppPFE - Introduction to C++ pr
cppVCPP - Introduction to C++
diagram - Introduction in Diagra
u1 - Logic Circuits
lift - Operating a lift usin
nuclear - Nuclear weap
u2 - OO Design
u3 - Flowchart Design
1
2
3
4
5
6
7
8
9
1 Menu Options
2 System level buttons: exit, options
3 Tree for courses, units and exercises
4 An exercise
5 Statusbar for messages
6 Exercise level buttons: MOTD, Notes, Summary
7 Unit level buttons: Notes, Summary
8 Exercise level buttons: setup, develop, submit, question, view testdata, view previous marks, execute solution
9 Text viewer

```
CourseMaster course Diagrammatics unit 2 exercise Nuclear:
Design of a nuclear weapon's countdown initiator safety mechanism using logic g
===============================================================================


Introduction
============

This is a more advanced exercise, in which you'll have to design the logic circ
which controls the launch of nuclear weapons (albeit a simplified one!) In this
exercise you will be concerned with the authorization circuit that initiates th
countdown counter for the launch of the nuclear weapon.

Your task is to draw the logic circuit for the authorization operation using th
digital gates provided. You'll also need to use the various input and output
components and connect your gates to these. Read the "Problem specification" be
for more information.

As with the previous exercise, clicking on "Develop" will launch Theseus.
```

Figure 6.19: A CourseMaster student client

Figure 6.19 illustrates a view of the CourseMaster client that is based on JFC. The options that are represented are context dependent. According to the state of the exercise, options progressively become available. For example, the option to submit coursework is available only if a solution has been developed, the option to develop a solution is available only after having set up the exercise files, and so on. The JFC client can be configured as to its graphical components, paths and colours. The window that presents the notes and exercise questions can render both text and simple HTML documents.

Currently, more clients are being developed, one of them being a web-based client that runs on Internet web browsers. This client uses a combination of static and dynamically generated HTML pages, Javascript and JSP in order to reach the same levels of functionality that the other three CourseMaster clients provide.

6.6.3 Integration with Diagram-Based CBA

The integration of DATsys with the generic marking system and CourseMaster was carried out easily as most implementation issues had been anticipated at design.

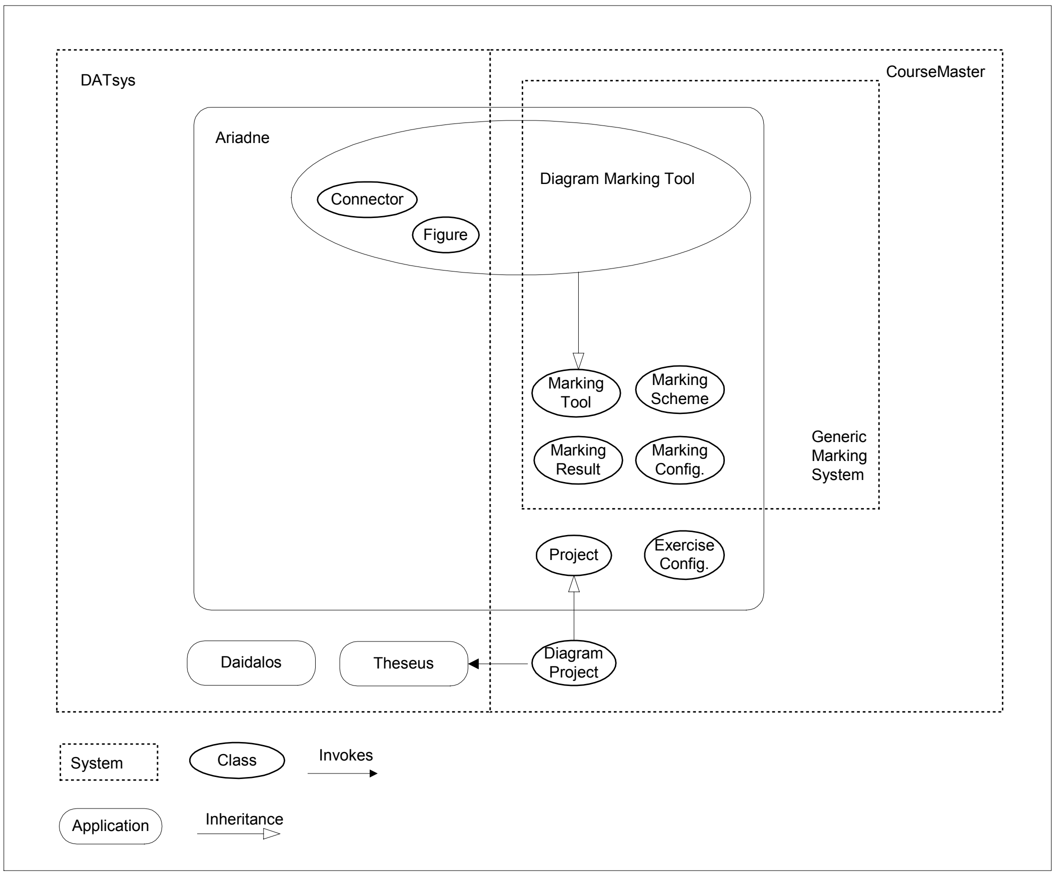


Figure 6.20: Dependencies between software components

Figure 6.20 illustrates the dependencies between the parts of DATsys, the generic marking mechanism and CourseMaster.

These dependencies reveal the points of integration. Daidalos is completely independent of the generic marking mechanism and CourseMaster. Theseus also is independent. Ariadne depends both upon parts of the generic marking mechanism and CourseMaster. These dependencies are necessary as Ariadne needs to allow the authoring of parts of the marking process and parts of CourseMaster's exercises.

Only diagrammatic marking tools from the generic marking system depend on DATsys classes. Specifically, diagrammatic marking tools need the `Figure` and `Connector` hierarchy to be able to traverse and query information about the diagram structure.

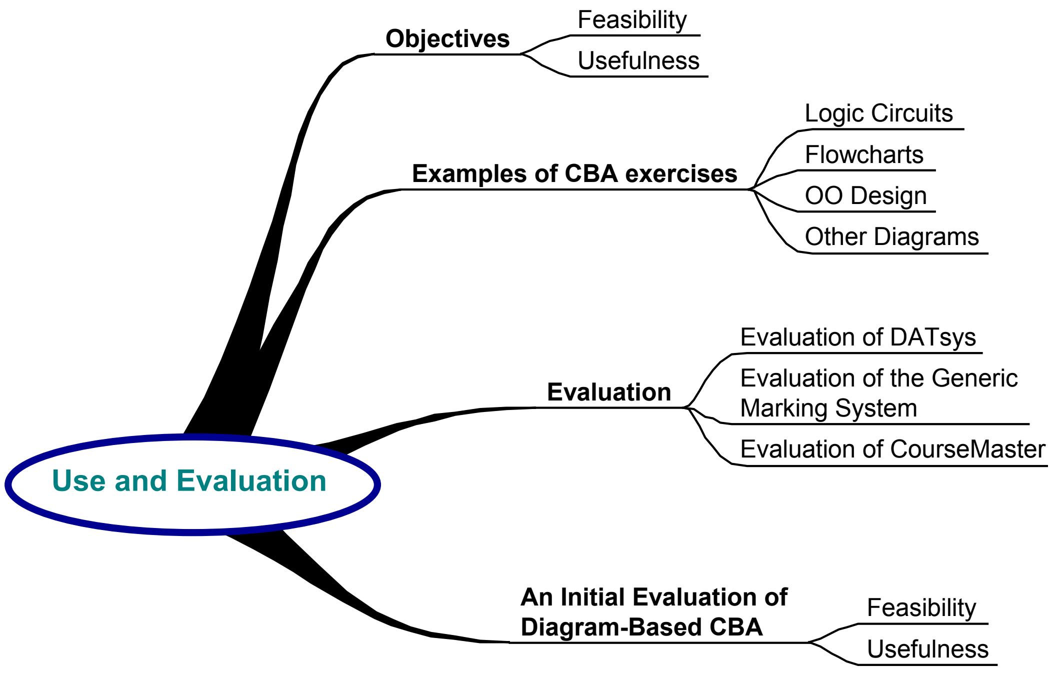
Only one point of CourseMaster knows about DATsys. CourseMaster diagram project objects need to be able to invoke Theseus when an action to develop an exercise has been requested by students.

6.7 Summary

This chapter presented an overview of the implementation of the system for the full lifecycle support of a diagram-based CBA. The system consists of the DATsys framework, the meta-diagramming editor Daidalos, the exercise authoring environment Ariadne, the student diagram environment Theseus, the generic marking mechanism and CourseMaster. The generic marking mechanism has been implemented as part of CourseMaster. The implementation used the Java programming language to adhere to the requirements and objectives that have been set in chapter 4.

The next chapter shows how the implementation is put into action for authoring, running, marking, administering and evaluating diagram-based CBA.

Chapter 7,
Use and Evaluation



"Not everything that can be counted counts,

and not everything that counts can be counted"

Albert Einstein

Introduction

This chapter argues that the development of an authoring environment for diagram-based CBA is feasible and useful. Following the implementation details given in chapter 6, this chapter illustrates the system's use, summarises the provisions that have been made for making customisations and extensions, evaluates the software from the perspectives of feasibility, usefulness, and software quality, and discusses initial results towards the evaluation of free response diagram-based CBA.

DATsys, the generic marking system and CourseMaster, can be used for a wide range of diagram-based exercises. All together they offer a simple and pragmatic technique to author, run, and manage diagram-based CBA. CourseMaster has been available within academia since 1999, providing support for the marking of programming coursework. The diagram authoring extension, entitled DATsys, was integrated into CourseMaster in May 2000.

Three examples of diagram-based exercises have been authored and tested in a controlled environment during the year 2000-2001 at the University of Nottingham. Section 7.2 describes the lifecycle of these exercises and discusses the advantages and limitations that have been detected. Sections 7.3 to 7.5 present an evaluation of DATsys, Daidalos, Ariadne, Theseus, the generic marking system and CourseMaster. Evaluation is considered from three perspectives, notably CBA, Diagramming and Software Engineering.

Section 7.6 gives some initial answers to the problem of developing diagram-based CBA, by presenting an evaluation of the usefulness of the authoring system to various parties and to assessment in general. It argues that the automation of the assessment of diagrams can be as effective as that of programs and discusses its practical and pedagogic benefits.

7.1 Objectives

This chapter addresses two main objectives:

- To evaluate the feasibility, usefulness and software quality of the authoring environment and platform for the support of the full lifecycle of diagram-based exercises

- To test diagram-based CBA in practice and draw initial conclusions about its benefits for assessment and learning in general

Examples of authored diagram-based CBA address the feasibility issue and are described in section 7.2. Usefulness and software quality aspects are addressed in sections 7.2 to 7.6.

The objective for DATsys and its three editors has been to offer for educators from non-programming backgrounds, customisation of the diagram editor to the requirements of the exercise. Many Theseus editors have been designed towards this goal. The aim for the generic marking system has been to cater for extensibility and to accept marking criteria for new diagram-based domains. Criteria have been implemented for circuits, flowcharts and object-oriented designs. The objective of CourseMaster has been to support the full lifecycle of diagram-based CBA. All the core functionality of Ceilidh has been implemented in CourseMaster, while software quality has been significantly improved.

7.2 Examples of Diagram-Based CBA exercises

The authoring of a diagram-based CBA exercise involves the following stages:

- Using Daidalos to build a tool library for creating and connecting diagram elements

- Using Ariadne to build a CBA exercise (by choosing a subset of Daidalos' tools for the student tool library, selecting application features, developing the marking scheme, and configuring the marking tools and the CBA exercise)

The full lifecycle of CBA exercises involves the following additional stages:

- Testing and deploying the exercise through CourseMaster

- Running the exercise and marking student solutions

- Administering the exercise and evaluating the results

In the academic year 2000-2001, exercises in three different domains were automatically assessed using DATsys and CourseMaster at the University of Nottingham. The exercises were part of a diagramming course that taught a range of diagram-based concepts. The authored exercises domains were:

- Logic design
- Flowchart design
- Object-oriented design

The exercises were authored using Daidalos and Ariadne and were deployed via CourseMaster servers. The class consisted of 167 first year undergraduate students majoring in computer science.

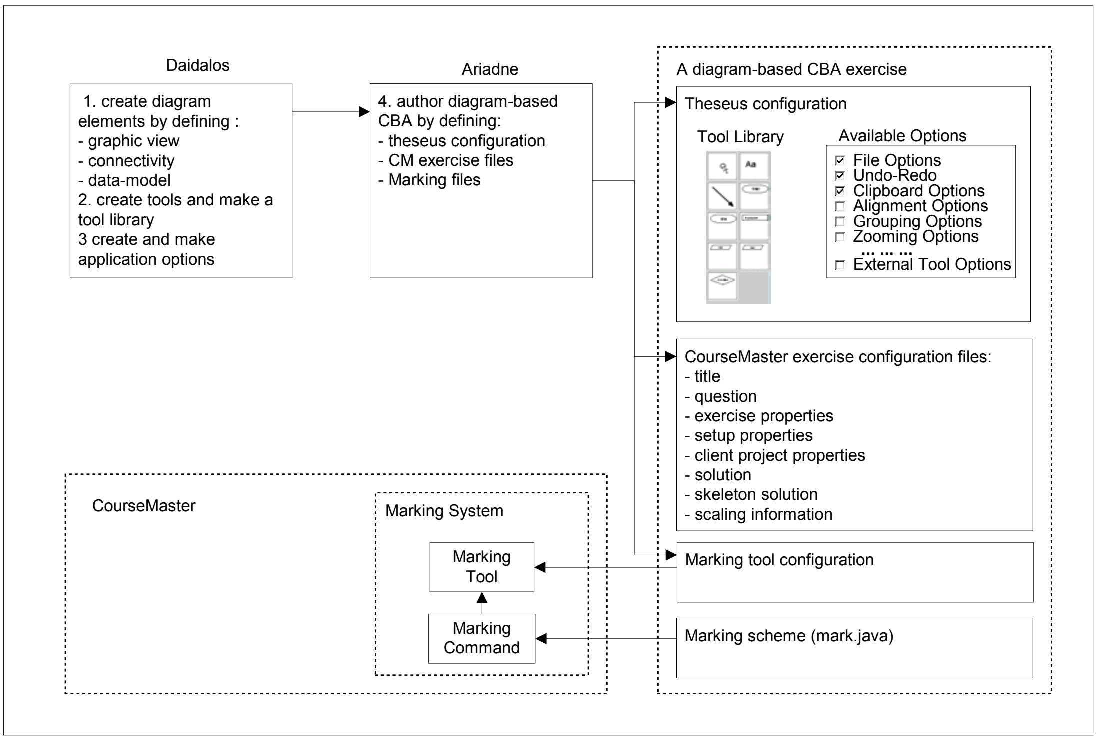


Figure 7.1: Steps for authoring diagram-based CBA

Diagram-based CBA exercises are authored with ease. The task is lengthy, but straightforward and the outcome benefits both students and educators alike. Figure 7.1 depicts the overall process. Daidalos is used to create tool libraries with tools for the creation and editing of diagram elements. The first step in the creation of a tool library is to draw the diagram elements. The connectivity properties for every element can then be entered. The next step is to create a data model for all the elements for which it is required. Once a diagram element is complete, it can be placed into the tool library.

Once a tool library has been created, Ariadne can be used to author CBA exercises in the domain for which the tools have been created. Application options for Theseus must be selected, and Ariadne is then used to develop the marking scheme and configure the marking tools and the properties of the specific CBA exercise. Ariadne invokes its own text editor in order for the user to enter the title and question of the exercise. The editing of the exercise's properties, the set up properties file and the CourseMaster clients' project properties come next. If automated marking is required, additional exercise related files must be created (such as the marks scaling configuration, and the configuration of the marking tools).

The exercise is then ready to be deployed and tested through CourseMaster. The exercise author should test the exercise by using CourseMaster and Theseus. After drawing the solution of the exercise, the exercise author must describe and test the marks and feedback for variations of the solution. As soon as the exercise is complete, it can be placed under the appropriate CourseMaster directory. The teacher can then decide when to make the exercise available to the students.

The next sections describe example coursework in the three aforementioned domains and highlight the variations between the numerous processes of supporting the coursework's full lifecycle.

7.2.2 Logic Design Coursework

7.2.2.1 Developing the Logic Design Exercises

Diagram-based CBA exercises in logic design were the first to be authored. Figure 7.2 depicts the entire process.

Daidalos is used to create the tool library that represents the logic gates; this task does not require much effort. The view of the various gates and other components can be made either by composing primitive shapes or by placing bitmap pictures of the various gates into a rectangle with a transparent perimeter. The connectivity properties for each gate can then be set by placing pin figures on the appropriate connection points. Pins need to be named and characterised as input or output. The connection figure that connects the two pins of two gates can be a simple three-segment poly-line (an elbow type of connection line). The interaction for connecting two gates requires clicking with the mouse a connection handle within a pin-figure, and dragging the connection line to another pin, thus creating a wire relationship. The stage of configuring a data model for every diagram element can be omitted, as gates do not have a data model. Once a gate's design is complete, the tool that creates tools is used to add a creation tool for the selected gate into the tool library. All gates can be constructed in a similar manner.

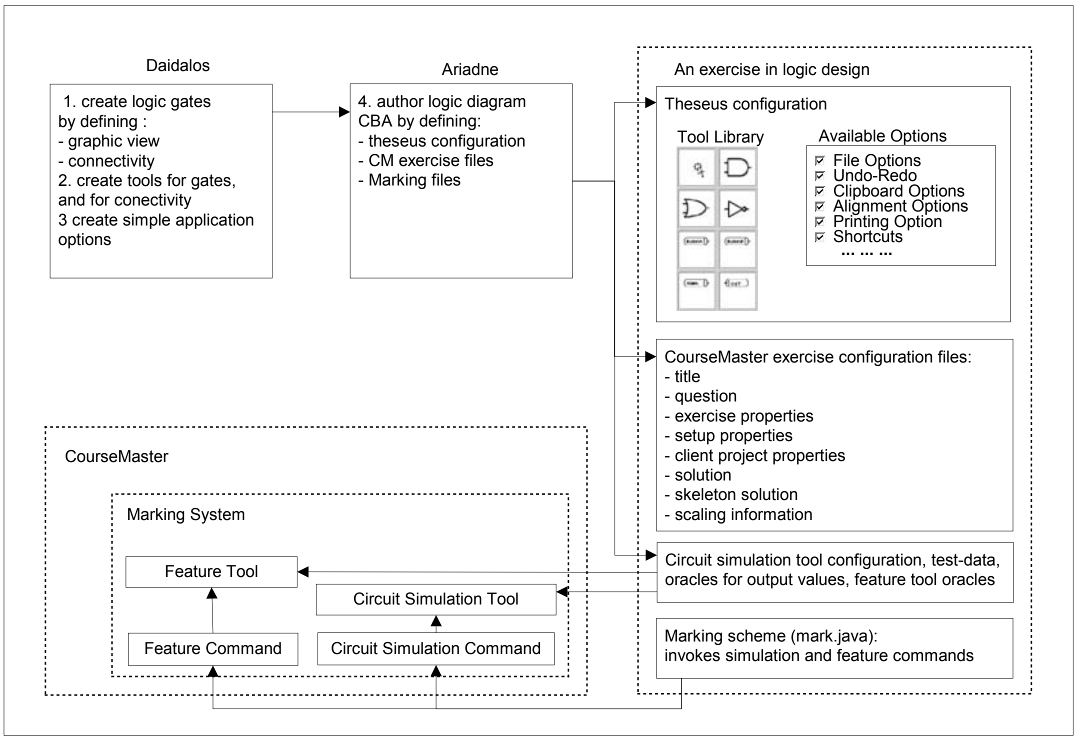

Daidalos
1. create logic gates by defining :
- graphic view
- connectivity
2. create tools for gates, and for conectivity
3 create simple application options
Ariadne
4. author logic diagram CBA by defining:
- theseus configuration
- CM exercise files
- Marking files
An exercise in logic design
Theseus configuration
Tool Library
Available Options
File Options
Undo-Redo
Clipboard Options
Alignment Options
Printing Option
Shortcuts
... ...
CourseMaster exercise configuration files:
- title
- question
- exercise properties
- setup properties
- client project properties
- solution
- skeleton solution
- scaling information
Circuit simulation tool configuration, test-data, oracles for output values, feature tool oracles
Marking scheme (mark.java):
invokes simulation and feature commands
CourseMaster
Marking System
Feature Tool
Circuit Simulation Tool
Feature Command
Circuit Simulation Command

Figure 7.2: Steps for authoring CBA exercises in logic design

Once a tool library for gates is complete, Ariadne can be used to select the application features that will configure Theseus. For example, as simple circuit design does not require zooming, so this feature is disabled. However, multiple undo and redo is useful, so this feature will be incorporated into Theseus. Ariadne is then used to develop the marking scheme and configure the marking tools and the CBA exercise. The CircuitSimMarkingTool is a marking tool for assessing simulated logic gates. This tool requires configuration, test-data and oracles for output values that the exercise author has to describe by choosing to edit the appropriate file from the exercise file repository.

Another marking tool that can be used in logic design CBA is the feature tool. This tool can test the students' ability to create a circuit with specific characteristics. For example, the feature tool can be set to check for the minimum or maximum number of gates that must be used. The feature tool requires another set of configurations and oracle files. The title, exercise question and the exercise's properties have to be given next. Optional exercise-related files include the marks scaling configuration, the setup properties file and the CourseMaster clients' project properties.

After the authoring process using Ariadne, the exercise can be deployed and tested through CourseMaster. Upon completion of the testing phase, the exercise can be made available to the students.

7.2.2.2 Use and Evaluation of the Logic Design Exercises

Two logic design exercises have been set as formal coursework at the University of Nottingham. The first exercise requires students to draw a simple circuit for an elevator control board. The second exercise requires the students to design a circuit for a switchboard that controls a nuclear facility.

A quick demonstration of Theseus was important to familiarise the students with the environment. Students were already familiar with CourseMaster, as they had already used it for assessment in a Java programming module during their previous semester.

The students that were taught the basics of logic diagrams had limited programming experience and had to think in terms of simple Boolean formulas. The solution of the first exercise required the use and connection of 3 gates (AND, OR and NOT), 3 input components (ON/OFF buttons), and 1 output component (an alarm). The second exercise's solution contained 9 gates, 3 inputs and 3 outputs. It is worth noting that the students were allowed to use more components to produce the same logic, and would have got full marks for the dynamic tests during the simulation of their circuit for doing so. However, during the features checking part of the marking process, the marking feedback would suggest to the students that their solution was not optimal. A small percentage of their marks would also have to be deducted for the feature test.

The marking server ran on a Pentium III 650 with 256MB RAM. The marking server was seen to mark up to 15 submissions concurrently. The administration of the exercise did not differ from previous CourseMaster programming assignments. The administrator was able to open and close the exercise on the appropriate dates, and give extensions and additional submissions to students that had the appropriate permission. These are standard features of the CourseMaster CBA system, available to the administrators for all the courses and exercises.

The two logic circuit exercises proved to be successful. Once students understood the question, they were able to identify the necessary components and draw the solution. The vast majority of the students came up with the correct results. Some students experiencing difficulties were helped by CourseMaster's feedback and as a result were able to draw the correct circuit. The two logic design exercises introduced the students to Theseus. Most of the students were very happy with the system. They particularly liked the uncluttered feel of Theseus and the speed with which the system responded during the drawing of their solution and the marking of their work. The students also mentioned that learning Theseus was easy and that the intuitiveness of the interaction with the diagram induced a playful state of mind.

7.2.3 Flowcharts

7.2.3.1 Developing the Flowchart Exercise

Diagram-based CBA for exercises that use flowchart diagrams was the second type of exercise that was created. Figure 7.3 depicts the authoring process.

The authoring of the tool library that represents the flowchart diagram symbols requires additional editing to that of logic gates. The view of the flowchart diagram elements can be made by composing shapes. The connectivity properties for each element can be set by selecting a perimeter-type of connectivity. The connection figure that connects two flowchart elements should be a simple line figure, decorated with an arrowhead to denote the direction of the flow. The data model for each element is a text label that holds the statement within the flowchart symbol. Once the diagram elements have been created, the tool library can be constructed.

For the next stage, Ariadne is used to configure Theseus and CourseMaster's marking system. Theseus must be configured to allow zooming as flowcharts can turn out to be quite large in size. Other options may be enabled such as grouping and alignment, font editing, and z-layer ordering. Appropriate marking tools can mark the student flowchart as effectively as in the case of logic design. The flowchart tool converts the diagram into BASIC code and reuses the dynamic marking tool to test the correctness of the flowchart's execution. The feature tool is also used to tests the students' ability to use the correct diagramming components. The feature tool also requires configuration and oracles.

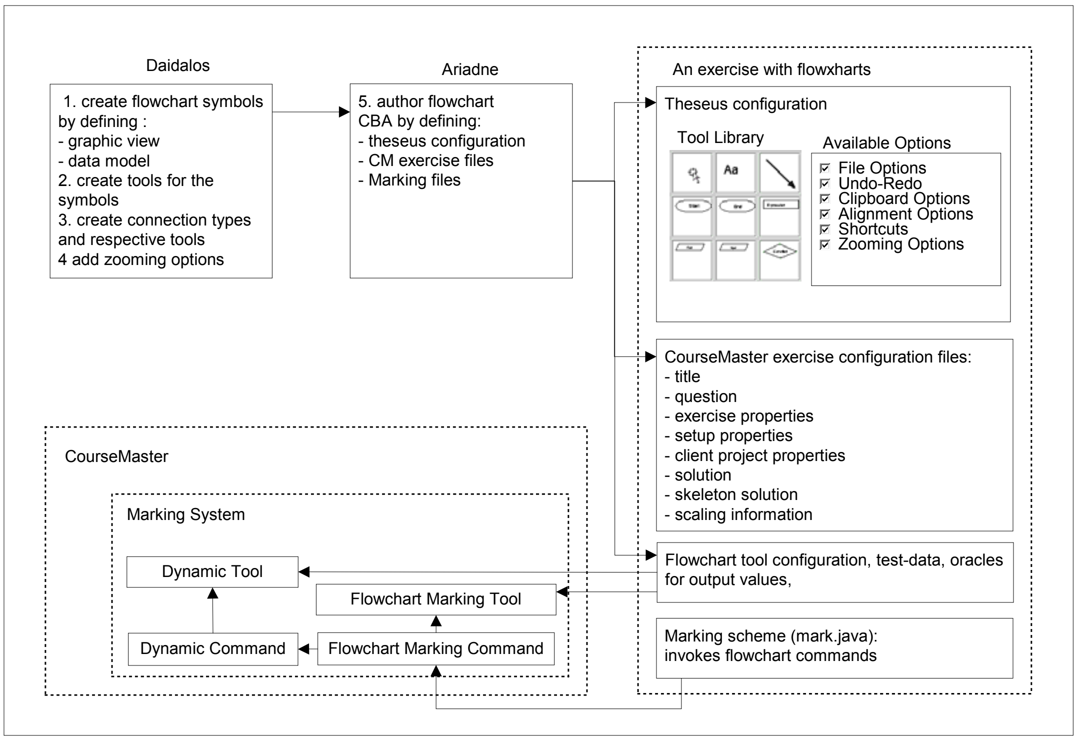


Figure 7.3: Steps for authoring CBA exercises in flowchart design

The remaining configuration needed for CourseMaster exercises is similar to the configuration in logic design exercises. The title, question, solution and exercise's properties along with exercise related files, such as the marks scaling configuration, the setup properties file and the CourseMaster clients' project properties must all be entered. The exercise is then deployed and tested through CourseMaster. As with the logic circuit exercise, the exercise developer should test the flowchart exercise by using CourseMaster and Theseus. After this stage, the exercise is ready to be run.

7.2.3.2 Use and Evaluation of the Flowchart Exercise

One exercise has been tested as formal coursework for flowcharts. The exercise requires students to draw a flowchart for comparing three numbers. Although simple, this example uses all the nodes of the flowchart diagram notation.

The students were taught the basics in flowchart diagrams. Being novice programmers, they had to design a simple algorithm. The solution of the first exercise required the students to:

- Draw a starting and ending flowchart node, three input statements, three conditional statements and three printing statements
- Enter statements within each flowchart symbol to define its meaning. For this reason the question description explained the simple syntax of these statements
- Connect the flowchart symbols using single arrowed lines

As with the circuit design exercise, the marking server run on the same server, a Pentium III 650 with 256MB of memory. Server statistics indicated a marking speed of up to 15 submissions at any time. The administration of this exercise did not differ from other CourseMaster exercises.

The flowchart exercise proved to be very popular. The majority of the students entered their solution directly into Theseus. Some less astute students got the order of input wrong and produced mixed results, but with the help of CourseMaster's feedback, they rectified their mistake to complete the exercise.

7.2.4 Object-Oriented Design

7.2.4.1 Developing the Object-Oriented Design Exercise

Diagram-based CBA for exercises in object-oriented design was the third type of exercise that was created. Figure 7.4 depicts the authoring process.

The authoring of the tool library that represents the symbols for object-oriented design diagrams requires more complex techniques than those of logic circuits or flowcharts. The view of the diagram elements for object-oriented design can be made either by composing primitive shapes or by writing skeleton Java code and using Daidalos' reverse engineering options. The latter is a feature of Daidalos that allows the user to select a directory that contains Java classes and/or source code. Daidalos then parses the classes in order to produce a UML diagram. The reverse engineering feature speeds up the creation of suitable diagram elements.

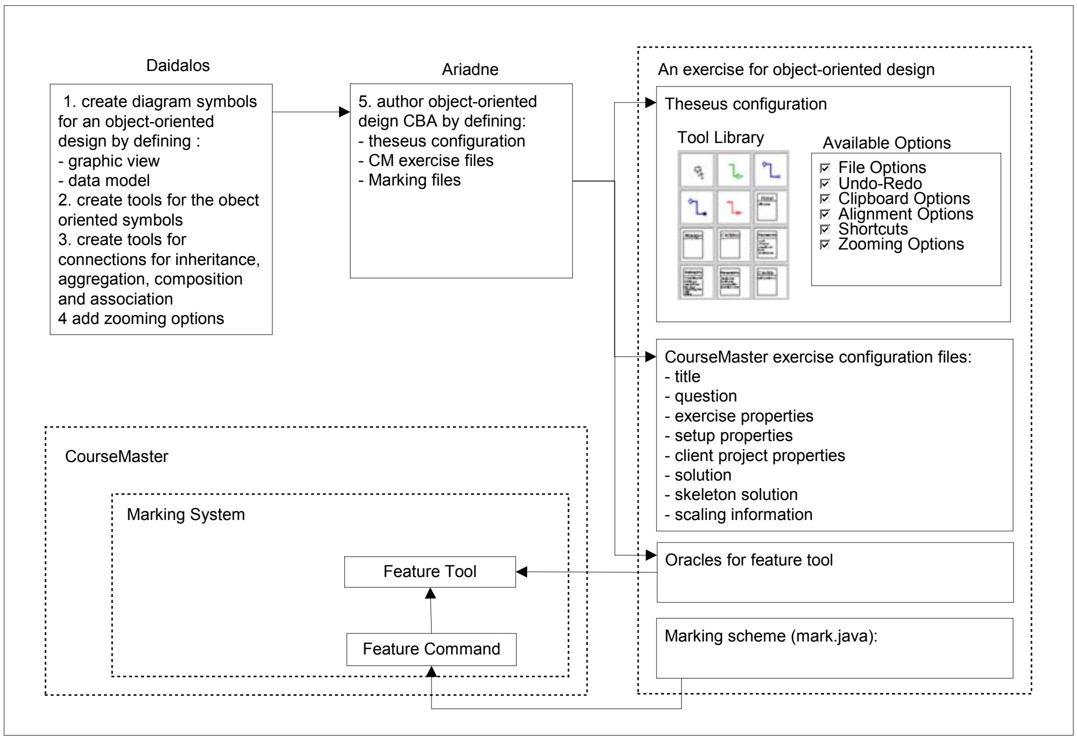


Figure 7.4: Steps for authoring CBA exercise in object-oriented design

As in flowcharts, the connectivity properties for each object-oriented node can be set by selecting a perimeter-type of connectivity. The connection figure that connects two object-oriented design elements can be a simple line figure or an elbow poly-line, decorated with a symbol to indicate the relationship type. Such symbols for typical object-oriented notations can be represented with figures such as circles, diamonds, and arrows. The data model for each element consists of a name and a list of attributes and methods. Any convention for describing the attributes and methods can be used.

Upon completion of the tool library, Ariadne is used to configure Theseus, CourseMaster's generic marking system, and its exercise properties. Theseus is configured to allow zooming because even simple object design diagrams can be quite large. Grouping and alignment options are also necessary and must be selected. Object-oriented design diagram exercises rely heavily on CourseMaster's feature tool. The feature tool requires configuration and output oracles. It is used to validate the relationships between diagram elements. The tool is also used to identify redundant classes and/or interfaces, and to distinguish between the cardinality of the diagramming components and their relationships. The remainder of the configuration is similar to the other two types of diagram-based CBA.

After the configuration stage, the exercise is ready to be deployed and tested. As with the other diagramming exercises, the exercise developer must test the exercise by using CourseMaster and Theseus. Subsequently, the exercise can be opened.

7.2.4.2 Use and Evaluation of the Object-Oriented Design Exercise

Formal coursework was set to test the diagram-based assessment of object-oriented designs. It required students to design a hotel management application according to a well-defined specification of requirements. This exercise is harder to solve than the ones described because of the expressiveness of the object-oriented diagram notation.

The students knew already about class diagrams and had to perform critical thinking on which object-oriented elements to use and how to connect them. The solution of the first exercise required the students to place 12 components and draw 17 relationships. Four additional components were available in the toolbar as decoys.

As with the rest of the diagramming exercises, the marking server ran on a Pentium III 650MHz server with 256MB RAM. The additional complexity of this exercise did not additionally tax the server. A marking speed of up to 15 concurrent submissions was again recorded. The administration of this exercise did not differ much from other CourseMaster exercises.

The object-oriented design exercise has been popular due to its clear and well-focused question. The complex nature of the solution drove a small number of students to draw the solution on paper first and then to enter it into Theseus. This caused some initial concern on the usability of Theseus. However, upon closer inspection, the root of the problem seemed to stem from the fact that these students simply preferred to design on paper. Overall, all diagramming exercises were met with considerable success.

7.2.5 Exercises in Other Diagram Notations

Many other types of exercises in various diagram-based domains can be created using the combination of DATsys and CourseMaster.

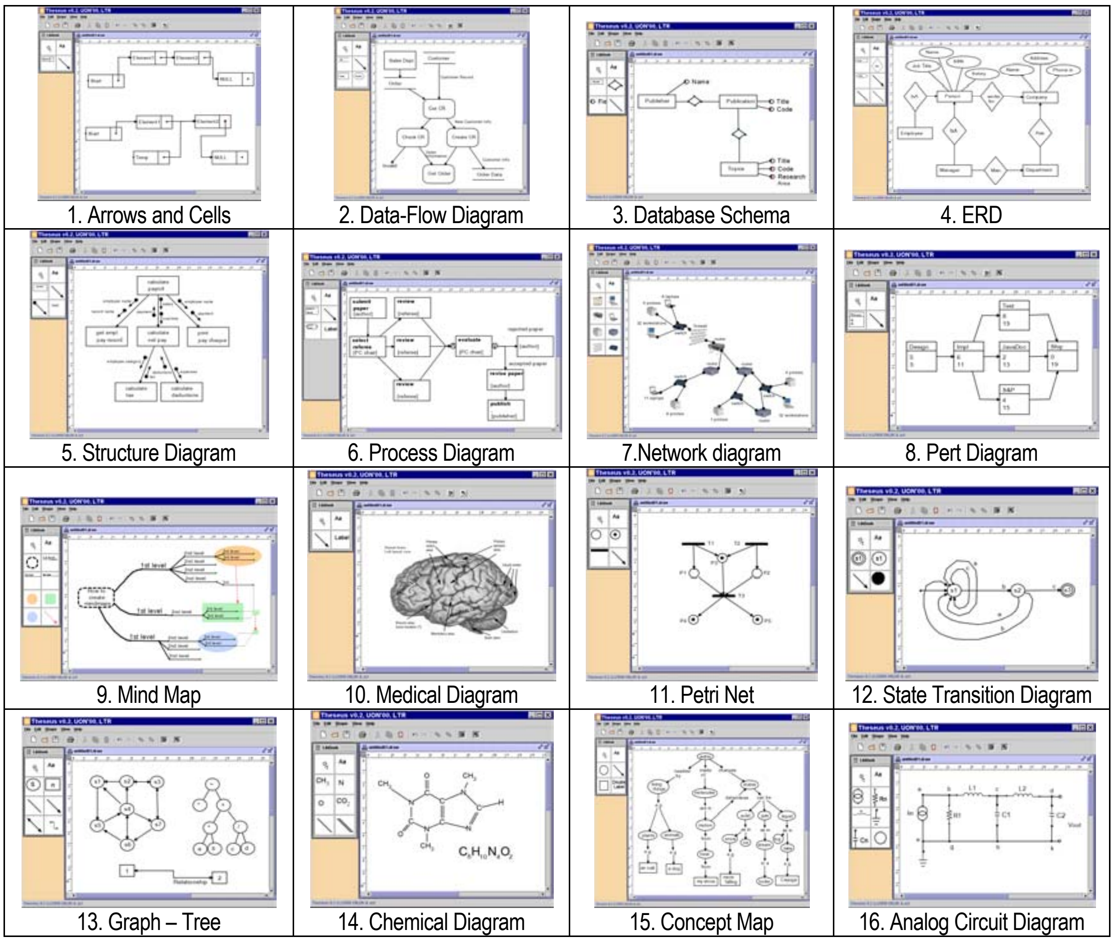


Figure 7.5: Various views of Theseus for sixteen notations

Figure 7.5 illustrates 16 types of possible coursework for which tool libraries have been implemented. The time spent authoring for these 16 examples was minimal. It took just three hours to make all the tool libraries using Daidalos. An additional three hours were needed to draw the diagrams using Theseus.

This experience answers the question that was introduced in section 1.2.1. “To what extent is it possible to generate domain and exercise dependent editors by means of configuring and drawing as opposed to the more difficult task of programming?”.

The simplicity of making the 16 additional types of diagram editors shows that a multitude of editors can be authored with minimal effort. This has been made possible by following the strategy of making the commonality across diagram editors part of the framework and the variation to be specifiable within an authoring environment. The limits of domain coverage using this technique are imposed by the way in which the variation is described. However, by fully applying Daidalos' prototypical dynamic software evolution technique, these limitations may be fully resolved as modifications unpredictable to the original design could be made to the structure and behaviour of the software, at runtime. Section 8.3.3 describes how this can be accomplished in the future. It encompasses some challenging and unresolved issues in the domain of software evolution and aspect-orientation.

Creating a tool library within Daidalos is sufficient only if the developer is satisfied with the similar-looking types of Theseus editors. A course developer with some programming experience could also extend DATsys with a new package of classes representing a completely new editor. In this case, the concepts modelled within DATsys could be directly reused.

Devising a marking strategy and appropriate marking tools is an essential step in authoring the CBA exercise. In some cases, common diagrammatic marking tools can be reused. In others, completely new tools and configurations are necessary.

Diagram-based marking tools could be developed to mark a selection of diagram notations that have been illustrated in figure 7.5:

- Data-flow diagrams could be marked in a similar manner to flowcharts. A data-flow marking tool could convert the diagram into an intermediate format and evaluate the result using oracles. The feature tool can be also used to examine concrete characteristics of the students' solutions.

- Database scheme diagrams could be marked by using a suitable tool that converts the diagram to a database table, runs SQL queries, and tests the output data using oracles. Depending on the result of the queries, marks can be awarded. The feature tool can also be used to address any shortcomings.

- Network diagrams might be converted into formats understood by various network simulator tools. Such tools can perform a variety of tasks including load balancing distribution examination, data throughput analysis and performance scaling investigation. The output of such a network simulator tool could be read back by the marking system and marks could be awarded. The Feature tool can further address any limitations of the simulator tool.

- Pert diagrams can be marked with the combined use of a pert simulation tool and a feature tool. The simulation tool would calculate and evaluate time dependencies and identify any discrepancies in the students' solutions.

- Medical diagrams (and any other type of picture-based diagrams) can be assessed by developing a marking tool of the graphical/hot spot category. The developer would configure the tool with the areas of interest, along with their name and coordinates. The tool could then examine the students' solutions and award marks accordingly.

- Analogue circuit diagrams could be marked with the use of an analogue simulator tool. A converter would need to convert the diagram to a net-list in a format that is understood by an external simulation tool such as Spice [Va94]. Spice could be used directly or through a Java wrapper such as JSpice [SHG+98]. The simulator can then simulate the circuit and its output can be read back by a feature marking tool and evaluated.

- Concept maps can be marked by latent semantic analysis tools similar to Lou's work on essay-based assessment [FL94]. A feature tool can then comment on the presence or absence of certain types of concepts.

The ideas for the above diagram-based marking tools are further discussed as part of future research.

7.2.6 Summary

This section has shown that the software does support the full lifecycle of diagram-based CBA. Exercises for three domains have been authored, deployed, marked, administered and evaluated with 167 students at the University of Nottingham. Many more example exercises can be created. Daidalos has libraries for more than 50 types of diagrams. With the addition of appropriate marking tools diagram-based CBA can be created for most types of diagram. Although developing new marking tools is a development task that needs planning, reuse of existing marking tools with minimal configuration modifications is possible.

7.3 Evaluation of DATsys

The main objective for designing and implementing DATsys has been to make the creation of a new type of editor a simple task that does not require programming. This objective has been more than successfully met as the creation of a new diagram editor literally takes minutes and does not involve any kind of programming or complex task.

The specific feasibility requirements that were introduced in section 4.3.1.1 gave three directions for the evaluation of DATsys. These were:

- The extent of domain coverage for new diagram editors
- The easiness of the mapping between the representations of domain elements on the diagram and of suitable structures that can be marked
- The easiness of authoring a new type of diagram-based CBA exercise

DATsys can be used to produce editors for a range of diagram notations that extends from simple structured graphics to graph types of diagram. A limitation of this research is that the architecture caters for connection-based diagrams. The mapping between the representations of domain elements and of structures that can be marked is anticipated only for diagrams that are similar to graphs. For other types of diagram notations that are inherently based upon the metric space, such as mechanical design and geometry, DATsys needs alterations.

The mapping between the representations of domain elements and of suitable structures that can be marked within Daidalos can be made on a one to one basis by uniquely naming every diagram element and tool. This mapping process is simplified by always using the same standard format for exporting a diagram, independently of the diagram type. The student solution is loaded by the marking server as a diagram object and can be processed in terms of its figures, their attributes and connectivity.

Additional marking tools can be provided with little effort. Such marking tools would either query for marking criteria or convert the diagram elements to other structures that can be marked by external tools. Developing marking tools is not difficult once CourseMaster's main marking classes are familiar. The most complex tool that has been developed as part of this research consists of only 320 lines of Java source code of intermediate difficulty. For some areas it will be harder to create the appropriate marking tools than others.

Authoring a new type of diagram-based exercise as described in section 7.2 is relatively simple. All the common objects are implemented and only definitions for the variations between courses needs to be given. In this sense, the software deliverables can be seen as an infrastructure for research and experimentation.

Theseus' potential is limited due to its lack of interoperability with other diagram editors. Ideally, DATsys would understand the format of standard diagram editors, render the diagram and let the user convert it to its native format . Another limitation is caused by the absence of functions to execute and simulate the diagram within the student's environment. This would have required a much more complex authoring process in which the execution behaviour for every diagram element and relationship would have to be defined within Daidalos.

7.4 Evaluation of the Generic Marking Mechanism

The generic marking system has been designed and implemented as an evolution of Ceilidh's marking mechanism. Section 4.3.1.2 identified the requirements for a new marking mechanism that achieves more flexible and generic marking than Ceilidh's. This section discusses the manner in which these design goals have been reached.

For diagram-based CBA, the extensibility of the marking mechanism has been demonstrated by configuring marking tools for three different domains. As section 7.2.5 has described, many more can be developed. Converting the diagram to some other structure takes some programming effort. However, by making the diagram available to the marking tools in the Extensible Markup Language (XML) [Hs01] the options for conversions become much wider.

Running the generic marking mechanism has been successful and CourseMaster's intended functionality has increased. The marking mechanism and CourseMaster preceded DATsys by almost two years. As part of CourseMaster, the generic marking mechanism has been more widely tested for assessing programming courses.

Some marking tools are generic and can be reused across domains. The feature tool, for example, is used in object-oriented design coursework and as a complementary tool to the simulation tool in logic design coursework. The feature tool is also widely used on CourseMaster's programming courses, Java, Java2 and C++. Once an exercise has been authored, the process of authoring similar exercises requires considerably less effort. The configuration files can just be copied and the required modifications can be applied. This significantly decreases the development time.

The decision to design CourseMaster using an object-oriented architecture has greatly increased its maintainability. The marking system benefits from CourseMaster's architecture and has evolved over the three years of its use to support more types of projects. Exercises have been customised to a much greater extent than in Ceilidh. Furthermore, Java's platform independence helps maintainability because there are very few platform specific segments of code to maintain.

The decision to implement the generic marking system in Java has increased its performance and scalability. More than 1000 student programs per week are automatically marked at the University of Nottingham alone. The generic marking system has run reliably for over three years. The fact that Java transparently supports Symmetric MultiProcessing (SMP) gives to the marking system a considerable advantage when run on a platform with multiple CPUs. Moreover, by allowing the allocation of servers to different machines, the marking subsystem can be placed on its own machine, thus increasing performance and scalability even further.

The generic marking system was developed in Java and was expected to exhibit the usual platform independent properties of the Java platform. However, this was not the case. Implementing a cross-platform marking system required some sections of platform dependent code to be written. Extensive testing under heavy loads was performed on Microsoft operating systems as well as Solaris and Linux.

Security has been an important design consideration. A major security concern arises when assessing student executables. However, the risk is reduced when assessing diagram-based coursework. Ceilidh's supports only the SUID and GUID security that is supported in Unix systems [GS96]. CourseMaster's marking system enhances security by running the student programs in a sandbox.

The flexibility of the generic marking system allows the exercise developer more freedom in expressing the specifics of the marking process for each exercise. CourseMaster's generic marking system is more expressive than Ceilidh's and supports extensive exercise customisation via the use of marking programs written in Java, additional exercise properties and improved feedback mechanisms. The generic marking mechanism has been found invaluable by both developers and teachers in improving Ceilidh's existing exercises and fine-tuning the students' learning experience.

The extra configuration that is needed to create an exercise within CourseMaster could mistakenly result in the assumption that the authoring process is harder under CourseMaster than under Ceilidh. Certainly, the process of authoring an exercise is more time consuming, but not particularly more complex. Furthermore, the additional effort spent during exercise authoring benefits the students. A simple exercise takes, on average, a few hours to create, a complex one takes up to a day. The time spent includes writing the exercise's question, its model solution, the related marking files and testing the exercise. The idea of describing the marking scheme as a program resulted in a flexible and extensible form of specification for marking criteria that assess student coursework. The feedback is implemented in custom ranges for each of the criteria described within the configurations of the marking tools.

The advantages of the designed generic marking mechanism over hotspot exercises are considerable. Diagram marking tools have access to the structure of the diagram. This is not possible on a graphical hot-spot CBA exercise. Ideally, in free response assessment, as the solution space of an exercise has potentially an infinite number of solutions, tests should attain "total coverage" of the solution. This needs careful planning and analysis. The use of oracles for the configuration of the marking tools has been appropriate to the three domains described in section 7.1. It will also be appropriate for the sixteen domains illustrated in figure 7.5.

The generic marking system has been a successful prototype mechanism that allows experimentation and creation of novel, automatically assessable, and across domains diagram-based CBA. Metrics research for the evaluation of diagrams can realistically be tested in large classes of students.

7.5 Evaluation of CourseMaster

The problem with supporting the full lifecycle of diagram-based CBA exercises is resolved by designing and implementing CourseMaster as an evolution of the Ceilidh system. Section 4.3.1.2 identified the requirements for CourseMaster:

- To improve Ceilidhs's performance, scalability, maintainability, platform neutrality, expressiveness, robustness, security and usability

- To seamlessly integrate with DATsys and the generic marking mechanism

CourseMaster approached Ceilidh's limitations by considering them as vital requirements during the design and implementation stage.

A current limitation of CourseMaster is that a piece of coursework cannot contain different exercise types. For example, CourseMaster cannot set coursework that consists of MCQs, programs and diagrams. This requirement was not planned for during the design phase. A number of enhancements must be made to the architecture of CourseMaster to allow incorporating such feature.

7.5.1 Timeline Highlights

CourseMaster was implemented in 1998 and tested for the first time as a replacement for Ceilidh in the academic year 1998-1999. The first courses that were authored covered two Java programming modules containing 35 exercises. CourseMaster was made available to other academic institutions in late 1999 and is currently in use in more than 15 academic institutions, where it supports the automation of coursework in classes that have as many as 1500 students. CourseMaster was employed at the

University of Nottingham supporting 150 students during 1998-1999, 270 students during 1999-2000, 291 students during 2000-2001 and 310 students during 2001-2002.

7.5.2. Available Courses

Two Java courses have initially been authored for CourseMaster. The University of Nottingham has been teaching these two programming courses to their first year undergraduate students since 1998. Furthermore, a C++ course has been recently developed. Ceilidh's extensive exercise base can be used as a source of exercise material. It is relatively simple to convert Ceilidh's C and C++ exercises to CourseMaster following the appropriate guidelines. The web administration tool assists developers and teachers with this task.

A "diagrammatics" module and a respective course for CourseMaster have been created as part of this work in April 2000. The module taught the basics in theory and practice of diagrammatic representation and explained some of the most common diagram notations used in computer science. The three types of diagram-based CBA presented throughout section 7.2 belonged to this module. All the exercises are distributed with CourseMaster.

7.5.3 Academic Institutions and CourseMaster

Many universities and academic institutions around the globe have acquired CourseMaster. Over 40 institutions have tried the trial version: The National University of Singapore, Ngee Ann Polytechnic, Johensuu University, and King's College have been some of the earliest to evaluate CourseMaster. More than 15 institutions are actively using CourseMaster's second version, released in July 2001.

The majority of test sites are old Ceilidh users that made the transition to CourseMaster. Feedback from these universities indicate that they find CourseMaster much more successful than Ceilidh in many respects. CourseMaster is easier to install, successful with scaling and helpful for the students. Anderson from Singapore University stated that "I … believe that overall the use of CourseMaster enhanced the course considerably" [Ah00], and Plasman from Glamorgan University stated that "We are very impressed by CourseMaster" [Pp00].

7.5.4 User Evaluation

Feedback received from students in the form of questionnaires and from informal discussions, indicates that students tend to like CourseMaster more as the course goes on. They appreciate the immediate feedback and feel that having three to five available submissions for an exercise helps in improving their skills. Less experienced students tend to like the idea of automatic assessment because they feel they can take their time in completing the coursework. Overall, student feedback has always been positive with minor exceptions.

Teachers are also very pleased with the system. They appreciate the fact that they no longer need to mark hundreds of exercise solutions. Because course administration and monitoring are very effective, even less time is spent on these activities.

Administrators are satisfied with the system. CourseMaster is much easier to set-up and run than Ceilidh. Furthermore, because CourseMaster comes with extensive Web administration tools, administration requires less effort.

7.5.5 Improvements over Ceilidh

CourseMaster made improvements over Ceilidh on:

- Functionality: by adding diagram-based CBA and further features
- Software quality: by noticeably improving maintainability, extensibility, performance, scalability, usability, and security

This section also discusses plagiarism detection, administration, user evaluation and future enhancements.

7.5.5.1 Maintainability

Maintainability is an important quality for software, often overlooked in favour of short-term objectives. This results in software being more expensive during the maintenance stage.

CourseMaster has a considerably higher degree of maintainability than Ceilidh for many reasons. Its object-oriented design exposes a structure that is amenable to change. CourseMaster can be updated through run-time and compile time configurations, sub-classing and class and method substitutions. The basic building blocks of maintainable code, encapsulation and modularisation, are inherent within the design which makes use of key object-oriented principles. In addition, Java's platform independence ensures that platform specific segments of code are kept to minimum. Finally, in contrast to Ceilidh, CourseMaster's source code demonstrates improved readability and comprehensive documentation.

CourseMaster supported many changes and extensions over the three years of its use. Two major and six minor versions were released in this period and more are scheduled for the future. Exercises in CourseMaster have been customised to a much greater extent than in Ceilidh.

7.5.5.2 Extensibility

Extensibility has been one of the most important objectives from the early stages of development. Facilitating experimentation and research for assessment has always been one of Ceilidh's most important strengths. Various types of courses have been developed over the years, the highest proportion of which were authored by Ceilidh's extended community.

CourseMaster increases Ceilidh's extensibility with three significant enhancements. Firstly, it employs the idea of describing the exercise's marking process in Java, which subsequently allows for a great degree of customisation. This is not only because Java's control structures can be used to fine tune the marking of an exercise but also because the marking scheme can access information directly from CourseMaster's internal state. The latter opens up many possibilities for future extensions. For example, marking tools could be written to take advantage of information about the student profile of the submitting student to personalise the exercises feedback.

Secondly, CourseMaster provides extensive facilities to inter-operate with external programs. It also features a generic type of course, which can be parameterised to create user-defined course types. Custom projects can be defined with little effort. In this case, configuration for invoking an external environment for the students must be given. Generic projects have been implemented for:

- Essay-coursework, that invokes MS-Word on the client [Mic01]

- Object-oriented design-coursework, that invokes Rational Rose on the client [Rat00]

- Generic diagram-based coursework, that invokes Visio on the client [En01]

Thirdly, CourseMaster incorporates DATsys, which allows the authoring of generic diagram-based exercises. DATsys presents many opportunities for extension as discussed in section 7.3.

Any platform that supports Java can be used to run and extend CourseMaster. Thorough testing under real conditions has been performed on Microsoft platforms, Solaris and Linux.

7.5.5.3 Performance and Scalability

The decision to re-implement Ceilidh in Java has been a vital element in increasing the system's performance and scalability. Using early versions of the Java language resulted in unsatisfactory runtime performance. However, these performance issues disappeared as improvements were made to both the Java platform and its supporting tools, namely compilers and its Hot-Spot technology [Gd98].

CourseMaster outperforms Ceilidh for two reasons: Firstly, CourseMaster has considerably reduced the number of spawned processes. Ceilidh's tools layer consists of externally invoked programs that require the operating system to spawn processes to interact with the user at all levels, including navigating through Ceilidh's menu structure. CourseMaster represents Ceilidh's tools as internal objects that run within a single process. User requests are handled on a per-thread basis. Consequently, the high cost associated with process-based context switching is avoided and CourseMaster performs much better than Ceilidh.

Secondly, by offloading the user tasks to the client, CourseMaster has succeeded in relieving the servers from the burden of executing student-spawned processes such as compilations, simulations and visualisations. A typical programming exercise in Ceilidh sees a student compiling their program many times before submitting. By moving those user tasks to the client, CourseMaster considerably decreases the amount of resources it requires to support students completing exercises.

CourseMaster's implementation has been further optimised using profiling and optimisation tools. Even better performance can be achieved by upgrading the hardware platform underneath the system, or by using two or more processors in SMP mode. CourseMaster can take advantage of servers with multiple CPUs thus significantly increasing its execution performance without the need of re-compilation or re-configuration.

The future performance of a system can be directly linked to its potential for large-scale use. CourseMaster scales better than Ceilidh because the system has been designed aiming execution in a distributed manner. By designing CourseMaster as a loosely coupled, location independent system of distributed objects, CourseMaster is better positioned to spread its processing load across a number of physical machines.

CourseMaster has been used to assess more than 1000 student exercise solutions per week at the University of Nottingham. Although the load has been considerably high before deadlines, CourseMaster has had relatively few problems and has run reliably for over three years. Reliability is also indicated by responses from other universities that have using CourseMaster. Received feedback shows that the system is very reliable at high loads. For example, the National University in Singapore (NUS) has been running CourseMaster for over a year marking more than 1500 assignments per week.

7.5.5.4 Usability

The choice of separating the system logic between clients and servers allows the development of many different types of clients.

A number of CourseMaster clients have been developed. Prior to the first release at the University of Nottingham, an initial text-based client was developed for testing purposes. A GUI client that used Java's AWT (Abstract Windowing Toolkit) was the first client that students used at Nottingham in 1998. In 1999, a new client was developed, using Java's JFC windowing toolkit. The new client required more processing power than the one based on AWT but it is more intuitive and it has a better look and feel. The options for the JFC client have been described in section 6.6.2.

In comparison to Ceilidh, CourseMaster presents a superior interface to the students. The tree component represents the available courses, units and exercises and allows the student to browse through the course's material. The options are context dependent and sensitive to the state of the exercise. For example, students that have not compiled their programs cannot select the option to submit. Options appear as menu items, toolbar buttons and shortcuts. Students can personalise their client view in various functional and presentational ways. The area where the notes are presented can render either text or HTML documents.

CourseMaster also improved Ceilidh's usability in respect to its deployment and configuration. CourseMaster's installation process is much easier than in Ceilidh. Ceilidh uses shell-scripts that require an in-depth knowledge of paths, environmental variables and external tools. In contrast, CourseMaster employs a graphical installation wizard that guides the user through the installation sequence. However, the configuration of CourseMaster requires some basic networking knowledge, such as configuring TCP/IP addresses and ports, and some minimum Java knowledge, such as setting the CLASSPATH variable correctly.

7.5.5.5 Security

Systems that support automatic assessment for summative purposes have additional security requirements.

One of the major security risks is posed when assessing programming executables. An astute student could devise malicious code in order to gain unauthorised access and cause damage. Both Ceilidh and CourseMaster feature a number of security mechanisms in order to ensure safe and trouble-free execution.

Ceilidh's security is based on the SUID and GUID security supported in Unix systems [GS96]. CourseMaster security has been a primary design concern. For programming courses, CourseMaster uses the SUID and GUID security mechanisms when running under Unix. Alternatively, the "runas" command can be used when running under Windows 2000. Students are allowed to use CourseMaster only if their username has been added to the login list. The administrator is responsible for maintaining user lists. CourseMaster provides two forms of authentication. The default way is to store the user's password in the login file. An alternative way requires the setting up of a POP3 server that will authenticate the users on behalf of CourseMaster.

The passwords that are transmitted between the clients and the servers may pass through potentially insecure networks. CourseMaster uses the DES password encryption algorithm to overcome this problem and to minimise risks of security breaches through network packet interception.

Encryption between clients and servers ensures that information stays hidden from unauthorised users. For each successful login, a unique session key is generated for authorisation purposes. Each key is assigned to a CourseMaster client and is validated on every transaction for the lifetime of that login. Session key identification improves security in ensuring that users are who they claim they are.

The actions of the various subsystems can be logged at four levels of detail. Submissions, marking, log-ins and user related information is archived by default. Additionally, CourseMaster can log its internal operations for debugging purposes. The level of detail for the auditing process can be configured at start-up or at runtime. The logging trails created by the auditing subsystem can be monitored either by examining the log files or online using CourseMaster's remote server console tool. CourseMaster's auditing facilities allow for simultaneous screen, file and network output of the system's trails.

All of these mechanisms work in conjunction with any other restrictions and privileges an administrator may assign to the students.

7.5.5.6 Plagiarism Detection

With the emergence of the Internet, academic institutions are increasingly concerned with the submission of plagiarised material. Operating system based security measures have to be taken into account in order to deter students from copying from each other. However, there are no means of guaranteeing that students will not share their work with others.

A solution to this problem was first introduced in Ceilidh in the form of a plagiarism detection tool. The tool compares all the student solutions with each other and reports evidence of plagiarism based on the similarities of student work.

When used for programming projects, the tool can detect comment and variable alterations as well as syntactical variations (e.g. the transformation of a "for" loop to a "while" loop, or the modification of a "switch" statement to multiple "if-else" statements). CourseMaster includes a re-implementation of this tool. The tool can be accessed through CourseMaster's web admin tools.

For diagram-based projects, a plagiarism tool can be written that detects copies of the same diagram file, by checking the coordinates of the diagram elements and relationships. However, much better detection of student plagiarism can be made by using audit trails within the student environment Theseus.

7.5.5.7 Administration

The adding and removing of users, courses, units and exercises, the opening and closing of exercises, and the monitoring of students are some of the most essential functions needed for the administration of a CBA system. Both Ceilidh and CourseMaster support these basic features for administering a course.

Ceilidh provides administrative tools in the form of shell scripts. CourseMaster has a web administration facility that performs similarly to Ceilidh's shell scripts. It uses a combination of dynamically generated HTML pages and CGI scripts. The available facilities include student statistics, changing of exercise properties and viewing missing/submitted student coursework. Monitoring of the system can be performed using either the web facilities or the remote server console client. The web facilities help the administrator to add and delete users, view error logs, edit course documents, create and install new courses units and exercises, gather exercise metrics and grant extensions to students.

One new feature of the web facilities can suggest to the teacher a selection of exercises to choose from based on a variety of selection criteria. Another new feature is a web-based wizard that allows the exercise developer to create new programming exercises for CourseMaster. The wizard guides the developer through the authoring process by presenting them with the sequence of steps that have to be completed. An additional mechanism allows the exercise developer to create a new exercise by modifying an existing one with similar features.

CourseMaster has a secure remote server console client. The client allows an administrator to connect to the CourseMaster servers and perform tasks such as monitoring and shutting down the system. The remote server console client can also be used to dynamically shutdown, unlink, and reload selective CourseMaster subsystems at runtime, should a reason to do so occur. This can be particularly useful if the CourseMaster developers add a feature or fix a bug in a specific subsystem. The CourseMaster servers don't have to stop in order for the new Java class definitions to be loaded.

System statistics and debugging facilities are available only on the administration console tool described in section 6.6.1. The system statistics display, amongst other information, the number of submissions processed, assessed and archived and the number of users currently logged in. The administration console tool also allows the reloading of the course directory structures, should a change occur. This is needed as CourseMaster caches the directory structures in memory to increase performance.

Ceilidh archives only the most recent submission of a student. On contrast, CourseMaster archives all students' submissions. CourseMaster makes it possible to revert to a previous submission if there is sufficient reason to do so, for example if requested by a tutor. All submissions are date and time stamped. Student receipt files are generated on every submission, and both binary and ASCII versions of those files are kept. The binary versions are used internally in CourseMaster, while the ASCII versions are used by the web tool. The choice of archiving all the information concerning student submissions has been found to be imperative in cases of disagreements and disputes with the students over their mark, time of submission, feedback given, and so on.

7.5.5.8 User Evaluation

Over the years of running both Ceilidh and CourseMaster at the University of Nottingham, questionnaires were given to the students. Discussions have also been held with students at the end of each course. The results analysed so far indicate that students find both systems helping them enhance their learning experience. CourseMaster questionnaire results reveal that all students would rather use CourseMaster than Ceilidh, mostly for its usability, user-friendliness and improved feedback.

The general impression given is that students find CourseMaster to be remarkably supportive. Some students expressed their concern at having trouble obtaining a full grade when trying to raise their total mark from mid nineties to high nineties. Conversely, other students reported that they use CourseMaster to get a good mark and then proceed to their next assignment. In general, students use CourseMaster in a helpful and appropriate manner.

Questionnaires have also been given to administrative and support staff. The staff firmly believes that CourseMaster is much easier to set-up and run than Ceilidh. They also report that the administrative workload has decreased, even if the number of students has more than quadrupled. The monitoring of student activities has become easier with CourseMaster's web facilities. Extensive use of links makes the administrative tasks quicker to perform.

7.6 Evaluation of Diagram-Based CBA with CourseMaster

Free response diagram-based CBA is a feasible and useful technique to automatically assess diagram-based exercises. The software deliverables described in this work demonstrate the feasibility of supporting the full lifecycle of diagram-based CBA. The evaluation illustrates the usefulness to those involved in assessment.

This work has taken a novel approach by integrating a custom built student diagram editor as part of the authoring of diagram-based CBA. This was important to align the student solution with the various marking tools.

Choosing to implement a generic marking mechanism that accepts pluggable marking tools has been very important in being able to create new diagram-based CBA without having to do major development.

The decision to redesign and reimplement Ceilidh as CourseMaster was fundamental to the success of integrating DATsys with the generic marking system.

The diagram domains presented in sections 7.2 and 7.2.5 have explicit structures that permit the simple specification of marking criteria. Both programming and diagram-based types of coursework are free response types that accept qualitative marking criteria. This work argues that as long as the diagram to be marked has an explicit structure, the effectiveness of its assessment can approach that of programming courses.

7.6.1 Practical Benefits

Diagram-based CBA exhibits the same practical benefits as other types of CBA.

Firstly, diagram-based CBA saves time. With a large number of students, the total time spent is much less than the time spent during manual marking. Secondly, diagram-based CBA saves on teaching resources and can reinvested back into the course. For example, lecturers have more time to research and concentrate on the taught material and the management of their class. Thirdly, diagram-based CBA scales well for large classroom sizes and can be used to assess exercises that are part of a distant learning environment.

7.6.2 Pedagogic Benefits

Diagram-based CBA using DATsys and CourseMaster exhibits considerable pedagogic benefits. It exhibits three important properties:

- Repeatable; when a student exercise is submitted to the marking system with the same inputs, it will always receive the same mark
- Consistent; the state of the marking system is the same both before and after marking a student's exercise
- Reliable; when the student exercise is submitted it is guaranteed that a mark will be produced for the student

Using CourseMaster and DATsys offers a fair opportunity for success. All students have the same options on their client and go through the same notes material. There is no discrimination between students for both the notes and the assessment. These issues need careful planning in the case of delivering personalised feedback and assessment.

CourseMaster treats all students anonymously. This is important in ensuring that exercise assessment cannot be influenced by the identity of the student submitting the exercise. This promotes fairness within CourseMaster because customisation of the exercise feedback is always based upon marking criteria.

CourseMaster supports the scheduled opening and closing of exercises. This can be done either manually or by setting timers. Experience in large classes of students has shown that there is always a need for submission extensions. For example, CourseMaster supports the late submission of exercises. In addition, the frequency by which exercises are open and closed lets the teachers dictate the pace of a course within CourseMaster.

Once students become familiar completing exercises within CourseMaster, academic institutions are no longer focusing upon course management and instead can be using that time to improve the course. For the student, using CourseMaster means more frequent and immediate feedback throughout the course.

CourseMaster makes assessment redeemable as it can be configured to allow many opportunities for submission to the students. Attempting to protect students from rushing into a second submission, CourseMaster also includes a parameter to set the time for the gap between student submissions.

By integrating diagram-based coursework with free response assessment, CourseMaster allows the description of marking criteria that have access to information that could be conducive to the assessment of all of Bloom's levels of learning. For courses that involve design, this is an important challenge.

Assessing knowledge and understanding of a design area could be done with appropriately designed MCQs. However, by allowing the student to design a diagram as a solution, marking criteria can infer cognitive processes that indicate learning on an analysis, synthesis and evaluation level.

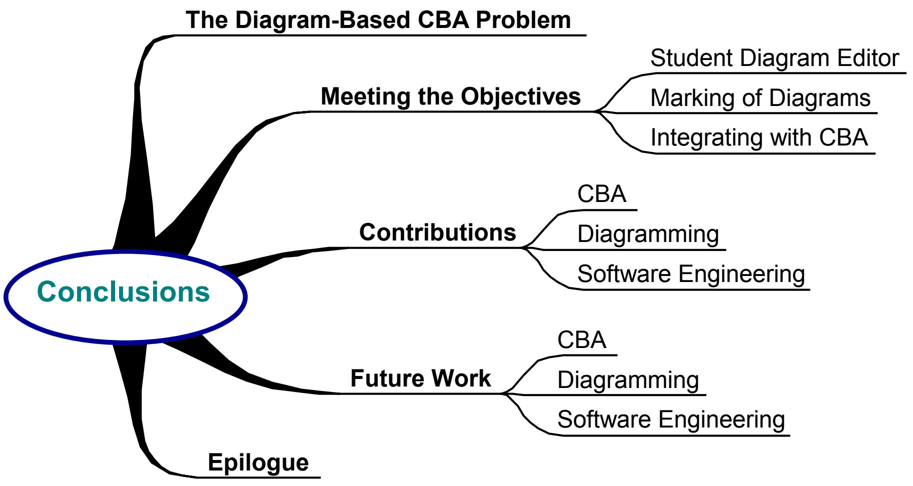
7.7 Summary

This chapter has described how the development of an authoring environment for diagram-based CBA is a feasible and useful idea. It has presented the key issues surrounding the use and evaluation of DATsys and CourseMaster, that together support the full lifecycle of free response diagram-based CBA.

Diagram-based exercises have been authored, tested and evaluated for logic design, flowcharts and object-oriented diagrams. Many more types of exercises can be authored in a wide range of diagram domains. The authoring process is relatively simple and rapid. Results from the exercises that have been selected for testing show both practical and pedagogical advantages.

The requirements for the design and implementation of DATsys have been met. The authoring of a student diagram editor became part of authoring the diagram-CBA exercise by using Daidalos and Ariadne. The generic marking system has met its requirements in replacing Ceilidh's marking mechanism and in making improvements in its expressiveness, platform independence, extensibility and usability. CourseMaster has also been successfully integrated with DATsys and the generic marking system, and replaced the Ceilidh system. CourseMaster improves Ceilidh's software qualities and adds a new type of course for diagrams.

Chapter 8,
Conclusions



"In theory, there is no difference between theory and practice.

But, in practice, there is"

Jan L.A. van de Snepscheut

Conclusions

Section 8.1 discusses the diagram-CBA problem and summarises how this work has met its general and specific objectives under the requirements that have been introduced in chapter 4. The contributions of this work to the fields of CBA and diagramming follow in section 8.2. Section 8.3 discusses topics for future work and indicates references for follow-up research. Finally, section 8.4 concludes this dissertation with an epilogue on diagrams, software, and automatic assessment.

8.1 Meeting the Objectives

This section revisits the objectives set out in the fourth chapter and demonstrates that the key objectives have been accomplished. These were to:

- Design a generic student diagram editor that can be easily specialised to the exercise
- Design a generic marking mechanism that can be customised to mark a range of diagrams
- Integrate the two designs in a CBA system that facilitates realistic experimentation and evaluation of diagram-based CBA for summative purposes

8.1.1 Customisable Student Diagram Editor

Incorporating the authoring of the student diagram editor as part of the authoring of the CBA exercise is a necessary step towards the integration between the diagram editor and the marking system. This decision greatly simplifies experimentation and research into new types of diagram-based CBA. It also gives the advantage of presenting the student with an environment built specifically for the exercise. Students are not required to learn a potentially complex environment to create their diagrams because Theseus can be authored to expose only the relevant features needed to complete a diagram-based solution.

The design and implementation of DATsys, Daidalos, Ariadne and Theseus has been successful. The potential domain coverage for new diagram editors is large. In the course of this research, Theseus has been used to edit more than fifty types of diagram notations. This research has demonstrated that the mapping between domain objects and diagram elements is both simple and effective. It takes minutes, rather than days, for a course developer to create diagram elements, connection types and supporting editor tools. Theseus can incorporate the range of standard application options typically found in modern diagram editors. All of Theseus' interactive features are founded on standard HCI techniques for direct manipulation. Its user interface utilises standard GUI components that are natural and intuitive for students.

DATsys has been designed with software quality as a prime concern. It has implemented the required functionality in a reusable and maintainable manner. This is reflected in its high performing and platform neutral implementation. DATsys provides a solid foundation for exploring possibilities for many types of assessment and for novel ideas in learning technology.

The authoring of a new type of diagram-based CBA is vastly simplified once appropriate marking tools have been created. Developing new marking tools requires some programming effort and knowledge of DATsys, the marking classes and CourseMaster. However, once a marking tool has been developed, it can be reused with other exercises of the same type. Diagram-based tools can be developed with minimal effort because they have direct access to a student's diagram object with all of its attributes and features.

8.1.2 The Generic Marking System and Marking of Diagrams

This research has argued that each domain has its own notions of quality. This fact was acknowledged from the project's initial stage and therefore this work opted for a generic solution to which changes and additions can be made with ease.

Thus, the main requirement for the marking mechanism, as discussed in section 4.2.2, has been to devise a prototypical mechanism for experimentation and the creation of novel automatically assessable and multi-domain diagram CBA. In this aspect, the design and implementation of the generic marking system has been successful.

Marking tools have been developed for three diagram-based domains and for programming in Java and C++.

The idea of expressing the marking scheme in a program worked very well. Although a marking scheme can be very descriptive, it does not require complex programming. The extensibility, expressiveness and feedback quality of the marking mechanism has increased considerably. In addition, by taking advantage of Java's dynamic linking and custom classloader features, it has been possible to edit and re-load marking schemes at run-time.

Other highlights of the marking system include the unlimited level of grouping of marking results, the use of styles for rendering the marks to students, the use of oracles to define marking tool configurations and the technique for setting feedback to user-defined ranges for the marking results.

From a software quality perspective, the generic marking system implements the necessary functionality in an effective way. Its marking tools are reusable and composable to facilitate building more complex marking tools. The design explicitly promotes maintainability by separating marking commands from marking tools. Marking tools are independent of the marking commands, while the latter play the role of adapting their interfaces to the former. Marking tools are available to the marking scheme only through marking commands. As section 7.5.5.3 has discussed, the implementation of the generic marking system is significantly more efficient than the marking mechanism used in Ceilidh. Performance has also been optimised with the help of profilers. Further performance enhancements can be achieved by spawning marking server processes to be hosted on multiple physical servers.

Security and robustness have been a fundamental requirement. Firstly, as the marking system is implemented by CourseMaster, it incorporates the security features discussed in section 7.4. Secondly, the marking server is completely hidden from the students. It can only be accessed via submission servers and only after the successful validation of the users' sessions.

A limitation of the generic marking system is that the process of creating new marking tools involves some programming. It might be possible to express marking tools using diagrams built within DATsys. In this way, the authoring of new types of exercises would be much easier. The potential of this idea should be investigated in the future.

8.1.3 Integration with CourseMaster

As discussed in section 4.1, the integration of DATsys and the generic marking system into a CBA system that supports the running, marking and management of CBA coursework has been fundamental to the pragmatic use of diagram-based assessment. CourseMaster's design aimed to facilitate this integration while mirroring Ceilidh's functional specification and improving its software quality.

The major improvements of CourseMaster over Ceilidh are:

- Object-oriented design and software patterns have been employed and have successfully aided in improving Ceilidh's maintainability.
- CourseMaster configurations are wider and more expressive.
- Performance has improved considerably.
- Scalability has increased tenfold. As a result, even when run as a single process, CourseMaster has been able to mark up to 1500 submissions on a weekly basis. With server clustering, scalability can increase even further.
- CourseMaster runs on many platforms. It has been tested in various configurations, types and versions of operating systems.
- Security and robustness have been addressed as part of the design, and improvements have been made to many parts of the assessment process.

CourseMaster has succeeded in replacing the Ceilidh system. Currently it is being used both at the University of Nottingham and in many of the universities that were using Ceilidh in the past. No testing sites have yet developed diagram-based CBA. However, from direct feedback, it is predicted that with more examples of marking tools and test exercises, diagram-based CBA will be a viable exercise type with UK universities that currently use CourseMaster.

8.2 Contributions

Although the main contributions of this work are in the area of CBA, considerable advances can be also demonstrated in the area of diagramming. Experience in these fields has been gained through the design, implementation and evaluation of DATsys, the generic marking system and CourseMaster. The following sections highlight the key contributions and a reflection on the lessons that were learnt as part of this research.

8.2.1 CBA

The first contribution to CBA is simply the development and running of a new type of CBA. Free response diagram-based CBA is a novel type of CBA that has not been attempted prior to this work. However, the most important contribution to CBA is that the software deliverables can be used as foundations for new research and experimentation with diagram-based CBA. In effect, CourseMaster can be used for the full lifecycle support of any type of fixed or free form CBA because it supports all the processes that are generally common in assessment.

Prototypical coursework in logic design, flowchart logic and object-oriented design indicates that the assessment of diagram-based CBA can be as effective as that in programming CBA. Tool libraries for more than fifty diagram notations have been authored and they are distributed together with CourseMaster and DATsys. Associated marking tools can be developed in the future.

CourseMaster contributes and impacts directly upon the CBA community. More than 15 academic institutions have successfully deployed CourseMaster and experiment with new programming exercises. Its use benefits students, teachers, coursework developers and researchers.

As classes of students get larger, so does the necessity to employ automatic assessment tools. DATsys and CourseMaster play an important role in this transition to automatic assessment, thereby underlying the key contribution to the field of CBA.

8.2.2 Diagramming

DATsys contributes to the area of diagramming by suggesting a process of authoring diagram editors that is simple and effective. As the design and implementation of diagram editors in general is a complex task, DATsys benefits researchers with little or no programming experience that want to simply, effectively, and rapidly create a diagram editor.

DATsys has being used in a number of projects for its diagram editing features. Ainsley used Daidalos to create a tool to reverse engineer object-oriented implementations [Ac00]. Students at the University of Nottingham have used Daidalos to create graph-tree diagrams and save these in various formats. Daidalos can also be used to build editors that support the editing of visual languages. Customised visual languages can be conveniently used to configure data structures or to describe complete programs.

DATsys is also useful for typesetting reasons. Certain Daidalos features such as the precise rotation of dotted lines, the mixing of transparency to bitmap textures, the configuration of connectivity and many others are not supported in existing diagram editors. Many of the diagrams of this thesis have been created in editors that have been authored within Daidalos.

As the base of computer users without programming expertise increases, so does the need to describe information in diagrammatic forms. DATsys approaches diagram editor creation in a novel way and allows the authoring of diagram editors by using interactive diagrammatic and graphical tools.

8.3 Future Work

Amongst the main contributions of this work has been the creation of a solid foundation for future extensions. The possible directions that can be researched to continue this work are numerous and span various subjects. This section highlights the most interesting ones from the perspectives of CBA, diagramming and software engineering.

8.3.1 CBA

Several final year and masters dissertations have been implemented for adding new features both in CourseMaster and DATsys [Hl98], [Hj98], [Ac00], [Mj00]. Interesting topics for student projects include the development of courses, new features for various user views, marking tools, and exercises. Currently, the support of dissertations related to psychology and education is being planned.

An interesting idea is to allow the mixing of assessment types to a composite CBA exercise. CourseMaster already supports the assessment of programming and diagram-based CBA. An exercise could use a combination of assessment techniques, by adding support for multiple-choice questions and essays. This idea necessitates revisiting the architecture of CourseMaster and making changes to all the objects that are using the project hierarchies.

The Learning Technology Research group at the University of Nottingham intends to make CourseMaster a foundation for a fully integrated teaching environment that will support:

- Intelligent monitoring of student progress and automatic student guidance using AI agents
- An intelligent tutoring system for automatic course delivery

For the first area, AI agents will be investigated as to whether they can be used to personalise student feedback and with the administrative tasks. Agents can take advantage of numerous types of data that have been collected and are stored within CourseMaster's archiving server.

For the second research area, links have been made with the REDEEM [AWW01] and Whurle [BMS+01] projects, and related technologies are under investigation. Currently a design phase is under progress that aims to integrate CourseMaster with UPortal [Up02].

Other active ideas for research include the marking of GUIs and networking programs.

8.3.2 Diagramming

The most prioritised extension for the diagramming part of this work is to incorporate conversion mechanisms between the diagram, the XML and the Scalable Vector Graphics (SVG) [Fj99] format types. This is necessary in order to allow the interfacing of DATsys to an increasing number of diagram editors that support the two formats.

Other extensions include the addition of more tool libraries, commands, primitive figures, handles and GUI components.

Although some simplified layout algorithms have been implemented, new layout managers for the diagram elements can be very useful for non-CBA related uses.

8.3.3 Software Engineering

Interesting future work on the implementation of the software deliverables is presented by further exploring of the idea of dynamic evolution in combination with aspect-orientation.

The idea of dynamic evolution involves making software aware of its source and of its compilation environment in order to provide features to users such as direct or indirect editing of the source code and re-linking of the new version at runtime. Taking advantage of this idea means that software effectively includes the development environment of itself, but in a restricted manner. This concept has been tested within Daidalos for classes representing commands, handles, connections and grid types of the diagram canvas. This environment is available when Daidalos is running in the "evolve" execution mode and allows direct editing of the source only on particular hotspots of the framework.

The benefits of employing this feature are substantial. Users can evolve the system at runtime in ways unanticipated prior to its design. The need to go through the steps of development to perform a modification disappears. These steps typically include loading the developing environment and the project for the software, locating the segment of code that needs updating, making the change, re-compiling the project, putting together a new version and ultimately re-installing and redeploying the software. Most importantly, with an appropriate mapping between the source code and diagrams, the evolution of the software can be given visually by users.

Facilities for user-oriented dynamic evolution of hotspots within DATsys can be usefully developed for more than 30 hotspots. This includes single classes representing new subtypes for all the major hierarchies. The set of hotspots should include classes for describing new primitive and composite figures, tools, handles, connectors, connection figures, GUI components and application options.

The issue of ensuring consistency between the old and newly evolved parts is an area that requires additional research. One way to achieve this would be to investigate the use of XML parsers and constraints [Cc01]. Subsequently, the software could also include management of the history of its evolution and its functions in order to export the changes and amend them into other installation bases. It is worth investigating if the process of installation of software can be discarded and replaced with software cloning that keeps inheritance relationships with its parent software.

The changes that can be made following this model can only be applied due to explicit hotspots of the architecture expressed as methods. It is interesting to explore the potential of allowing updates to the structure of the framework at runtime. In that case, DATsys could evolve further than the limits imposed by the constraints of the predetermined hotspots.

The second area for future work in DATsys, the generic marking system and in CourseMaster is the investigation of the benefits of applying aspect-orientation to the current architectures. Aspect-orientation helps improving the modularisation of a system by encapsulating cross cutting concerns such as exception handling, debugging, auditing, and so on. The implementation of such concerns is usually spread through the classes and objects of a system. This makes code harder to understand and less maintainable/evolvable. Aspects can introduce fields and methods to any class and they can be notified before and after the invocation of methods.

Aspects use designators to attach code. Designators are described using regular expressions. Therefore, obtaining all the methods of a set of classes and introducing code that prints their name as they are invoked, is a simple task. A simple tracer aspect has been added to DATsys for debugging purposes. Aspects can be developed and added selectively to the system at compile time. A theoretical view of aspect-oriented programming (AOP) has been given by Kiczales et al in [KLM+97], and a comprehensive practical guide is described in [KH01].

Encapsulating concerns that crosscut class hierarchies within DATsys and CourseMaster could increase both their modularity and maintainability. In addition, an interesting idea is to model and implement extension points into aspects and attempt to dynamically load these at runtime. A large amount of changes can be made to both the structure and behaviour of the running code by dynamically updating aspects at runtime. This may lead to considerable improvements in extensibility.

8.4 Epilogue

This research investigated the feasibility and usefulness of the idea of designing an authoring environment for developing diagram-based CBA. Free response diagram-based CBA has not been reported prior to this work. This can be largely attributed to the difficulties involved in the customisation of the student diagram editor to the specifics of the exercise, the inflexibility of existing marking mechanisms to accommodate new criteria and the lack of support for the full lifecycle of CBA exercises.

An innovative facility has being designed and implemented to allow experimentation, research and development of diagram-based CBA coursework for summative assessment in a controlled environment. Two complete systems, DATsys and CourseMaster, are the deliverables of this research that together make the support for the full lifecycle of diagram-based CBA coursework both viable and realistic. The running of free response diagram-based CBA is a solid advancement in the CBA field.

DATsys solves the problem of customising the diagram-editor to the specifics of the exercise. The generic marking system solves part of the problem of marking diagram-based coursework. It abstracts the common parts of the assessment mechanism and allows variation to be expressed using marking schemes and marking tools. The generic marking system is a part of CourseMaster. CourseMaster solves the problem of supporting the full lifecycle of CBA while integrating DATsys and the generic marking mechanism. CourseMaster has been successfully used for three years in academic institutions by educators for assessing large classes of students.

The problems of this research were solved by applying current software engineering principles. DATsys is an object-oriented framework designed to allow extensions through visual specification. The generic marking system is a set of cooperating classes that allows the description of marking schemes and pluggable marking tools. Three marking tools have been implemented in this work as examples. Many more can be implemented and appended to the marking system with relatively little effort.

Ceilidh and its successor have proven to be invaluable to the University of Nottingham and to other academic institutions worldwide. CourseMaster demonstrates considerable improvements over Ceilidh. Its architecture and implementation satisfy the objectives of maintaining Ceilidh's core functionality while increasing performance, scalability, maintainability, extensibility and usability. The modifications and amendments made on the assessment and administration processes have also been successful, as they have improved the expressiveness of the marking process and eased the management of courses, thus helping academic institutions in their transition to automatic assessment.

Bibliography

AB99 Arnow D., Barshay O., On-line programming examinations using Web to teach, Proceedings of the 4th annual SIGCSE/SIGCUE on Innovation and technology in computer science education, Krakow, Poland, June 27 - 30, 1999, Pages 21–24

Ac00 Ainsley C., Java Reverse Engineering Application, B.Sc. Dissertation, Computer Science Department, The University of Nottingham, 2000

Ac77 Alexander C., A Pattern Language, Oxford University Press, New York, 1977

Ac96 Alexander C., OOPSLA'96 keynote speech, Conference video, 1996, Available from www.murl.microsoft.com

ACM00 ACM Statement, A Summary of the ACM Position on Software Engineering as a Licensed Engineering Profession, Available from www.acm.org/serving/se_policy/, July 2000

Ad96 Anthony D., Patterns for Classroom Education, In Vlissides J., Kerth N., Coplien J., (editors), Pattern Languages of Program Design, Volume 2, Addison-Wesley, 1996, Reading, Massachusetts, US, Pages 391-406

Ah00 Anderson H., private email sent to LTR, march 2000

AHM99 Alexander W., Higgison C., Mogey N., Videoconferencing for Teaching and Learning: Case Studies, Institute for Computer Based Learning, Heriot-Watt University, Edinburgh, UK, 1999

AIS79 Alexander C., The Timeless Way of Building, Oxford University Press, New York, 1979

ANSI00 American National Standards Institute (ANSI), Available from www.ansi.org, 2000

AP89 Ackermann E., Pope W., Computer aided program design experiments: diagrammatic versus textual material, Proceedings of the 20th SIGCSE technical symposium on computer science education, Louisville, USA, February 23 - 24, 1989, Pages 117–121

Ap93 Ackermann P., Combining 2D User Interface Components and Interactive 3D Graphics in ET++, Conference Proceedings, Technology of Object-Oriented Languages and Systems (TOOLS 93), Santa Barbara, USA, August, 1993

Ap96 Ackermann P., Developing Object-Oriented Multimedia Software - Based on the MET++ Application Framework, dpunkt Verlag, Heidelberg, 1996

App89 Apple Computer, MacAppII Programmer's Guide, Apple Computer Inc., Cupertino, CA, 1989

Ass94 Assymetrix Inc, ToolBook User's Manual, Assymetrix Incorporated, 1994

Aut00 Autodesk Inc., AutoSketch version 7.0, 2000, Available from www.autodesk.com

AWW01 Ainsworth, S., Williams, B., Wood, D., Using the REDEEM ITS authoring environment in naval training, IEEE International Conference on Advanced Learning Technologies, 2001

Ba79 Borning, A., Thinglab — A Constraint-Oriented Simulation Laboratory, Ph.D. thesis, March, 1979

Ba98 Blackwell A., Metaphor in Diagrams, PhD thesis, Cambridge University, 1998

Bb86 Boehm B., A Spiral Model of Software Development and Enhancement, ACM SIGSOFT Software Engineering Notes, August, 1986

BBC99 Bransford J., Brown A., Cocking R., How people learn: Brain, Mind, Experience and School, Committee on Developments in the Science of Learning, National Research Council, US National Academy of Sciences, 1999

BBF+93 Benford S., Burke E., Foxley E., Gutteridge N. and Zin A. M., Experiences with the Ceilidh System, Proceedings of the 1st International Conference on Computer Based Learning in Science (CBLIS'93), Vienna, Austria, 1993

BBF+94 Benford S., Burke E., Foxley E., Gutteridge N. and Zin A. M., The Design Document for Ceilidh version 2, LTR Report, Computer Science Department, The University of Nottingham, 1994

BBF+95 Benford S., Burke E., Foxley E., Higgins C., The Ceilidh System for the Automatic Grading of Students on Programming Courses, ACM Press, Proceedings of the 33rd Annual ACM Southeast Conference, Clemson, South Carolina, March, 1995

BBF+96 Benford S., Burke E., Foxley E., Gutteridge N., Zin A. M., Course Developer's Guide to Ceilidh's Xwindow interface, LTR Report, Computer Science Department, The University of Nottingham, UK, 1996

BBF92 Benford S., Burke E., Foxley E., Courseware to support the teaching of Programming, Proceedings of the Conference on Developments in the Teaching of Computer Science, University of Kent, 6th-8th April 1992, pages 158-166

BBF96 Benford S., Burke E., Foxley E., Developer's Guide to Ceilidh, LTR Report, Computer Science Department, The University of Nottingham, UK, 1996

BBG+93 Benford S., Burke E., Gutteridge N., Foxley E., Zin A. M., Experience using the Ceilidh System, Proceedings of the All Ireland Conference on Delivering the Computer Curriculum, Dublin, September 1993

BC87 Beck K., Cunningham W., Using Pattern Languages for Object-Oriented Programs, OOPSLA-87 workshop on the Specification and Design for Object-Oriented Programming, Technical Report CR-87-43, September 17, 1987

BCG97 Brna, P., Cox, R., Good, J., Learning to think and communicate with diagrams, Proceedings of Thinking and Learning with diagrams II workshop, Portsmouth, 1997

BE01 Blackwell A., Engelhardt Y., A meta-taxonomy for diagram research, Diagrammatic Representation and Reasoning, Olivier P., Anderson M., Meyer B., (editors), Springer-Verlag, 2001

Be94 Batten E., New Computer Grading of Student Prose, Using Modern Concepts and Software, Journal of Experimental Education, 1994, Issue 62, Volume 2, Pages 127-142

BE98 Blackwell A., Engelhardt Y., A taxonomy of diagram taxonomies, In Proceedings of Thinking with Diagrams 98: Is there a Science of Diagrams? , 1998, Pages 60-70

Bg91 Booch G., Object Oriented Design with Applications, Benjamin Cummings, Redwood City, CA, 1991

BG97 Beck K., Gamma E., Advanced Design with Patterns in Java, OOPSLA'97, Tutorial No. 30

BGK+96 Broy M., Grosu R., Klein C., Rumpe, B., State Transition Diagrams, Technical Report TUM-I-9630, Technical University of Munchen, 1996

BGL+95 Burnett, M., Goldberg, A., Lewis, T., Visual Object Oriented Programming , Manning Publications Co., 1995, ISBN 0131723979

BH98 Brynjolfsson E., Hitt L., Beyond the productivity paradox, Communications of the ACM, Volume 41, Issue 8, Publisher ACM Press New York, NY, USA, 1998, Pages 49-55

Bj93 Bull J., Using Technology to Assess Student Learning, TLTP Project Alter, December 1993, ISBN 1 85889 091 8

BJ94 Brant J., Johnson R., Creating tools in HotDraw by composition, In TOOLS Europe 13, 1994

Bj95 Brant J., HotDraw, MSc thesis, University of Illinois, Urbana-Champaign, 1995

Bj99 Bull, J., Update on the National TLTP3 Project: The implementation and evaluation of computer-assisted assessment, Keynote, Proceedings of the 3rd Annual Computer Assessment Conference, Loughborough, UK, 16-17 June, 1999

BJR98 Booch G., Jacobson I., Rumbaugh J., The UML User Guide, Addison Wesley, 1998

BKW+98 Burstein J., Kukich K., Wolff S., Lu C., Chodorow M., Computer Analysis of Essays, NCME Symposium on Automated Scoring, April 1998

Bl85 Bagrow L., History of cartography, 2nd edition, Chicago, Precedent Publishers, 1985

BLN86 Batini C., Lenzerini M., Navathe B., A comparative analysis of methodologies for database schema integration, ACM Computing Surveys, Volume 18 (4), December 1986

BM95 Burnett M., McIntyre D., Visual Programming, Computer 28, Volume 3, 14-16 March, 1995, Available from citeseer.nj.nec.com/burnett99visual.html

BMF+96 Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M., Pattern-Oriented Software Architecture - A System of Patterns, Wiley and Sons, 1996

BMM+98 Brown W. H., Malveau R. C., McCormick III H. W., Mowbray T. J., Anti Patterns - Refactoring Software, Architectures and Projects in Crisis, Wiley, 1998, ISBN 0471197130

BMS+01 Brailsford T., Moore A., Stewart C., Zakaria M., Choo B., Davies P., Towards a framework for effective web-based distributed learning, 10th International World Wide Web Conference, Hong Kong, May 1-5, 2001

BR96 Brown S., Race P., 500 Tips on assessment, Cogan Page, London, UK, 1996

BS00 Buck D., Stucki D., Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development, Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education, TX USA, March 7 - 12, 2000, Pages 75 – 79

Bs56 Bloom, B., Taxonomy of Educational Objectives, New York, David McKay Company, 1956

Cc01	Cleaveland C., Program Generators with XML and Java, Prentice Hall PTR, 2001, ISBN 0-13-025878-4
CE00	Czarnecki K., Eisenecker U., Generative Programming, Methods, Tools, and Applications, Addison Wesley, 2000
CE98a	Charman D., Elmes A., Computer Based Assessment: A guide to good practice, Volume I, University of Plymouth, 1998
CE98b	Charman D., Elmes A., Computer Based Assessment: A guide to good practice, Volume II, University of Plymouth, 1998
Cf98	Culwin F., Web hosted assessment--possibilities and policy, Proceedings of the 6th annual Conference on the Teaching of Computing/3rd Annual ITiCSE Conference on Changing the Delivery of Computer Science Education, 1998, Pages 55–58
CHS+97	Codenie W., Hondt K., Steyaert P., Vercammen A., From custom applications to domain-specific frameworks, Communications of the ACM, Volume 40, 1997, Pages 70–77
Cj92	Coplien J., Advanced C++ Programming Styles and Idioms, Addison-Wesley, Reading, MA, 1992
Cj95	Coplien J., A Development Process Generative Pattern Language, In Coplien J., Schmidt D., (editors), Pattern Languages of Program Design, chapter 13, Addison-Wesley, Reading, MA, 1995, Pages 183-237
Cj99	Christie J., Automated Essay Marking for both Style and Content, Proceedings of the 3rd Annual Computer Assessment Conference, Loughborough, 16-17 June 1999
CM01	CourseMaster Design Document, LTR Group, Technical Documentation, Computer Science Department, University of Nottingham, June, 2001
Cm93	Clarke M., Possible models diagrams: a visual alternative to truth tables, Proceedings of the 24th SIGCSE technical symposium on computer science education, Indianapolis, USA, February 18 - 19, 1993, Pages 232–236
Cp76	Chen P., The Entity-Relationship Model: Toward a Unified View of Data, ACM Transactions on Database Systems, Volume 1, March, 1976, Pages 9-36
Cp92	Coad P., Object-Oriented Patterns, Communications of the ACM Volume 35, Issue 9, 1992, Pages 152-159
CS95	Coplien J., Schmidt D., (editors), Pattern Languages of Program Design, Volume 1, Addison-Wesley, Reading, 1995

CS98	Canup M., Shackelford R., Using software to solve problems in large computing courses, Proceedings of the 29th SIGCSE, Technical Symposium on Computer Science Education, Atlanta, USA, 1998, Pages 135–139
Cso01	Computer Systems Odessa Ltd., Concept Draw Users Manual, July 2001, Available from conceptdraw.com/en/resources/Help/CONTENTS.html
CW00	Coplien J., Wolf B., A Pattern Language for Writer's Workshops, In Foote B., Harrison N., Rohnert H., (editors), Pattern Languages of Program Design 4, Addison Wesley, Reading, MA, 2000, chapter 25, Pages 557-584
CY90	Coad P., Yourdon E., Object-Oriented Analysis, Yourdon Press/Prentice Hall, Englewood Cliffs, New Jersey, 1990
Dc99	Daly C., RoboProf and an introductory computer programming course, Proceedings of the 4th annual SIGCSE/SIGCUE on Innovation and Technology in Computer Science Education June 27 - 30, 1999, Krakow Poland, Pages 155 – 158
Dd99	Dodson D., Diagrammatic Interaction, Tutorial, Computer Science Department, City University, London, 12 February 1999
Df83	Day F., Computer Aided Software Engineering (CASE), Annual ACM IEEE Design Automation Conference archive Proceedings of the 20th Design Automation Conference, 1983, Pages: 129–136, ISBN 0-8186-0026-8
DG99	Dalziel, J. and Gazzard, S., Next generation computer assisted assessment software: the design and implementation of WebMCQ, Proceedings of the 3rd Annual Computer Assessment Conference, Loughborough, UK, 16-17 June, 1999
DMN+97	Demeyer S., Meijler Y., Nierstrasz O., Steyaert P., Design guidelines for tailorable frameworks, Communications of the ACM, Volume 40, 1997, Pages 60–64
Dp89	Denning P., A debate on teaching computing science, Communications of the ACM, Volume 32, 1989, Pages 1397–1414
DS98	Dowd K., Severance C., High Performance Computing, 2nd Edition, 2nd Edition, July, 1998, Pages 121-122, ISBN 1-56592-312-X
Dw00	Doube W., Distance teaching workloads, Proceedings of the 31st SIGCSE technical symposium on Computer Science Education, Austin, TX USA, March 7 - 12, 2000, Pages 347–351

Ea96	Endres A., History of Software Engineering, Seminar 9635, Schloß Dagstuhl, Conference title: A Synopsis of Software Engineering History - the Industrial Perspective, August 26-30, 1996
eb11	Encyclopaedia Britannica, 11th edition, University Press, Cambridge, 1911
Eb98	Eckel B., Thinking in Java, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998
Edu02	Educational Testing Service, Available from www.ets.org, 2002
EG97	Do E., Gross M., Thinking with Diagrams in Architectural Design, International Conference on Thinking with Diagrams, Portsmouth, January 9-10, 1997
El61	Euler L., Lettres a une princesse d'Allemagne, volume 2, Letters Number 102-108, 1761
En01	Eaton N., Microsoft Visio Version 2002 Inside Out, Microsoft Press, June, 2001, ISBN 0735612854
ES00	English J., Siviter P., Experience with an automatically assessed course, 5th annual SIGCSE /SIGCUE conference on Innovation and Technology in Computer Science Education, Helsinki Finland, July 11 - 13, 2000, Pages 168 – 171
EX95	Examine, Biology Software News, Issue 6, September, 1995, page 30
FD01	Friendly M., Denis D., Milestones in the History of Thematic Cartography, Statistical Graphics, and Data Visualisation, An illustrated chronology of innovations, Available from www.math.yorku.ca/SCS/Gallery/milestone/index.html, May 2001
FGZ+93	Foxley E., Gutteridge N., Zin A., Benford S., Burke E., Early experiences of computer aided assessment and administration when teaching computer programming, Association for Learning Technology Journal, Issue 2, 1993, Pages 55-70
FHG96	Foxley E., Higgins C., Gibbon C., The Ceilidh System : A General Overview, LTR Report, Computer Science Department, The University of Nottingham, UK, 1996
FHH+01	Foxley E., Higgins C., Hegazy T., Symeonidis P., Tsintsifas A., The CourseMaster CBA System: Improvements over Ceilidh, 5th Annual Computer Assisted Assessment Conference, Loughborough, UK, 2-4 July, 2001, Pages 189-201, ISBN 0-9539572-0-9
FHT+98	Foxley E., Higgins C., Tsintsifas A., Symeonidis P., Security Issues under Ceilidh's WWW Interface, Proceedings of ICCE'98, Volume 1, Beijing, China, October 14-17, 1998, Pages 235-240

FHT+99	Foxley E., Higgins C., Tsintsifas A, Symeonidis P. Ceilidh, a System for the Automatic Evaluation of Student Programming Work, Proceedings of the 4th International Conference on Computer Based Learning in Science (CBLIS99), University of Twente, Netherlands, July 2-6, 1999, Section I6
FHT98	Foxley E., Higgins C., Tsintsifas A., The Ceilidh System: A general overview, Proceedings of the 2nd Annual International Computer Assisted Assessment Conference, Loughborough, UK, 17-18 June, 1998
Fj99	Ferraiolo J., Scalable vector graphics (SVG) 1.0 specification, W3C Working Draft, December, 1999
Fk96	Fisler K., Exploiting the Potential of Diagrams in Guiding Hardware Reasoning, In G. Allwein and Barwise J., (editors), Logical Reasoning with Diagrams, Oxford University Press, 1996
FL94	Foxley E., Lou B., A Simple Text Automatic Marking System, Artificial Intelligence and Simulation of Behaviour 94 Conference for: Computational Linguistics for Speech and Handwriting Recognition, Workshop in Leeds University, UK, April 12th, 1994
Fm97	Fowler M., Analysis Patterns: Reusable Object Models, Addison Wesley, Reading, 1997
Fm99	Fowler M., Refactoring: Improving the Design of Existing Code, Addison-Wesley, Reading, 1999
FNT97	Foxley E., Nobar P., Tsintsifas A., Ceilidh and the World Wide Web, Proceedings of the 3rd International Conference on Computer Based Learning in Science, Leicester, UK, July 4-8, 1997, Section G6, ISBN 80-7040-217-2
Fr79	Floyd R., The paradigms of programming, Communications of the ACM, Volume 22, 1979, Pages 455–460
FS97	Fowler M., Scott K., UML Distilled: Applying the Standard Object Modelling Language, Addison-Wesley, New York, 1997, ISBN 0-201-32563-2
FSZ97	Foxley E., Salman O., Shukur Z., The automatic assessment of Z specifications, Working group reports and supplemental proceedings, Uppsala, Sweden, June 1 - 5, 1997
FW65	Forsythe G., Wirth N., Automatic grading of programs, Communications of the ACM, Issue 8, 1965, Pages 275-278
FWW00	Ferguson R., Hunter A., Hardy C., MetaBuilder: The diagrammer's diagrammer, Lecture Notes in Artificial Intelligence, Springer Verlag, Theory and Applications of Diagrams, First International Conference, Edinburgh, September 2000, Pages 407-421

FZ93 Foxley E., Mohd Zin A. M., The Oracle output recogniser, LTR Report, Computer Science Department, The University of Nottingham, 1993

Gc92 Goldfarb C., The SGML Handbook, Oxford University Press, 1992

Gd98 Griswold D., The Java HotSpot virtual machine architecture, March, 1998, Available from www.javasoft.com/products/hotspot/whitepaper.html

Ge84 Goldsmith, E., Research into illustration: An approach and a review, Cambridge University Press, 1984

Gg92 Gibbs G., Assessing More Students, Oxford Centre for Staff Development, Oxford Brookes University, 1992

GHJ+94 Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994

GJS97 Gosling J., Joy B., Steele G., The Java Language Specification, Addison Wesley, 1997

GMW88 Gamma E., Marty R., Weinand A., ET++ - an object oriented application framework in C++, Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'88), San Diego, CA USA, September 25 - 30, 1988, Pages 46-57

Gr96 Guerraoui R., Strategic Research Directions in Object-Oriented Programming, ACM Computing Surveys, Volume 28 Issue 4, December, 1996

Gr98 Gisselquist R., Engineering in software, Communications of the ACM, Volume 41, 1998, Pages 107–108

GS79 Gane C., Sarson T., Structured Systems Analysis, Prentice-Hall, 1979

GS95 Gaines B., Shaw M., Concept maps as hypermedia components, Knowledge Science Institute, University of Calgary, 1995

GS96 Garfinkel S., Spafford G., Practical Unix and Internet Security, O'Reilly and Associates Inc., 1996

Gt00 Greening T., Pedagogically sound responses to economic rationalism, Proceedings of the 31st SIGCSE technical symposium on Computer Science Education, Austin, TX USA, March 7-12, 2000, Pages 149–156

GV47 Goldstine H., Neuman J., Planning and Coding Problems for an Electronic Computing Instrument, Volumes I, II, III, Van Nostrand, Princeton, N.J., 1947 and 1948

Hd88 Harel D., On visual formalisms, Communications of the ACM, Volume 31,May 1988, Pages 514 – 530

HHS+01 Higgins C., Hegazy T., Symeonidis P., Tsintsifas A., The CourseMaster CBA System, submitted to the Journal of Education and Information Technologies (official Journal of the IFIP Technical Committee on Education) published by Chapman and Hall, ISBN 1360-2357

Hi72 Hispalensis I., De responsione mundi et de astrorum ordinatione, Ausburg, Gunther Zainer, December 7, 1472

Hi88 Hirmanpour I., A student system development diagrammer, Proceedings of the 19th SIGCSE technical symposium on Computer Science Education, 25 – 26 February, 1988, Atlanta, GA USA, Pages 104-108

Hj94 Holmevik J., Compiling SIMULA: A Historical Study of Technological Genesis, Annals of the History of Computing, Volume 16, Issue 4, 1994

Hj98 Hammond J., Chart-Wizard: A marks visualisation editor for CourseMaster, M.Sc. Thesis, Information Technology Institute, The University of Nottingham, UK, 1998

HL96 Holland I., Lieberherr K., Object-Oriented Design, ACM Computing Surveys, Volume 28, No 1, March 1996

HL98 Hoggarth G., Lockyer M., An automated student diagram assessment system, Proceedings of the 6th Annual Conference on the Teaching of Computing/3rd Annual Conference on Integrating Technology Into Computer Science Education on Changing the Delivery of Computer Science Education, Dublin, Ireland, 18 – 21 August, 1998, Pages 122 - 124

Hl98 Hern L., The Java Assessor, B.Sc. Dissertation, Computer Science Department, The University of Nottingham, UK, 1998

Hm77 Halstead M., Elements of Software Science, North Holland, 1977

HMM00 Herman I., Melancon G., Marshall M., Graph visualization and navigation in information visualization: A survey, IEEE Transactions on Visualization and Computer Graphics, Volume 6, Issue 1, 2000, Pages 24-43

Hs01 Holzner S., Inside XML, New Riders Publishing, 2001, ISBN 0-7357-1020-1

HS84 Henry S., Kafura D., The evaluation of software system's structure using quantitative, software metrics, Software - Practice and Experience, Volume 14, Issue 6, June, 1984, Pages 561-573

Hs90 Hekmatpour S., Templa and Graphica A Generic Graphical Editor for the MacIntosh, Prentice Hall, New York, 1990

HST02 Higgins C., Symeonidis P., Tsintsifas A., The Marking Sybsystem for CoureMaster, to be presented at the 7th Annual Conference on Integrating Technology Into Computer Science Education, University of Aarthus, Denmark, June 24-26, 2002

Hv99 Hardy V., Java 2D API Graphics, Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1999

HW96 Haarslev V., Wessel M. GenEd : an editor with generic semantics for formal reasoning about visual notations, IEEE Computer Society Press, In 1996 IEEE Symposium on Visual Languages, Boulder, Colorado, USA, 3-6 September, 1996, Pages 204—211, Available from citeseer.nj.nec.com/haarslev96gened.html

ISO00 International Standards Organisation (ISO), Available from www.iso.org

JCJ+92 Jacobson I., Christerson M., Johnsson P., Overgaard G., Object Oriented Software Engineering: A use case driven approach, Prentice Hall, Englewood Cliffs, New Jersey, 1992

Jd00 Jackson D., A semi-automated approach to online assessment, 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, Helsinki, Finland, 11 – 13 July, 2000, Pages 164-167

JFS99 Johnson R., Fayad M., Schmidt M., Building Application Frameworks, Wiley, 1999

Jj91 Jeffrey J., Using Petri nets to introduce operating system concepts, Papers of the 22nd SIGCSE Technical Symposium on Computer Science Education, San Antonio, TX USA, March 7 – 8, 1991, Pages 324–329

JL98 Joy M., Luck M., Effective electronic marking for on-line assessment, Proceedings of the 6th annual conference on the teaching of computing/3rd annual conference on integrating technology into computer science education on changing the delivery of computer science education, Dublin, Ireland, August 18 - 21, 1998, Pages 134–138

JN83 Jajodia S., Ng P., On the representation of relational structures by entity-relationship diagrams, Entity-Relationship Approach to Software Engineering, editors Davis C.G., Jajodia S., Ng P.A., Yeh R., North-Holland, 1983, 249-263

Jr92 Johnson R., Documenting Frameworks using Patterns, Proceedings of OOPSLA'92, ACM SIG-PLAN Notices, Volume 27, Number 10, Vancouver BC, Canada, October 1992, Pages 63-76

Jr97 Johnson R., Frameworks = (components + patterns), Communications of the ACM, Volume 40, 1997, Pages 39 – 42

JU97	Jackson D., Usher M., Grading student programs using ASSYST, Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education, San Jose, CA USA, February 27- March 1, 1997, Pages 335–339
Ka52	Kircher A., Oedipus Aegyptiacus, Rome, Italy, 1652
Ka69	Kay, A., The Reactive Engine, Ph.D. Thesis, Electrical Engineering and Computer Science Department, University of Utah, 1969
Ka77	Kay, A., Personal Dynamic Media, IEEE Computer Magazine, Volume 10, Issue 3, 1977, Pages 31-42
Ka97	Kay A., The Computer Revolution Hasn't Happened Yet, Key-talk in the International Conference for Object-Oriented Programming Languages and Applications (OOSPLA'97), Video session distributed by University Video Communications, Stanford, Available from www.uvc.com, 1997
KB99	McKenna C., Bull J., Designing effective objective test questions: and Introductory Workshop, Proceedings of the 3rd Annual Computer Assessment Conference, Loughborough UK, June 16-17, 1999
KB99b	McKenna C., Bull J., The CAA Centre National Survey of CAA, Association for Learning Technology Conference, Bristol, UK, September, 1999
KH01	Kiczales G., Hilsdale E., Aspect-Oriented Programming with AspectJ version 1.0rc2, Tutorial in the International Conference for Object-Oriented Programming Languages and Applications, (OOPSLA 2000), October 15, 2000
KLM+97	Kiczales G., Lamping J., Mendheker A., Maeda C., Lopes C., Loingtier J. M., Irwin J., Aspect-Oriented Programming, ECOOP'97, Springer-Verlag, Lecture Notes in Computer Science, 1997, Pages 220-242
KM00	Korhonen A., Malmi L., Algorithm simulation with automatic assessment, 5th annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science and Education, Helsinki Finland, July 11-13, 2000, Pages 160-163
KM90	Korson T., McGregor J., Understanding object-oriented: a unifying paradigm, Communications of the ACM, Volume 33, 1990, Pages 40-60
KSI+94	Kay D., Scott T., Isaacson P., Reek K., Automated grading assistance for student programs, Selected papers of the 25th annual SIGCSE Symposium on Computer Science Education, Phoenix, AR USA, March 10–12, 1994, Pages 381–382
Kt70	Kuhn T., The Structure of Scientific Revolutions, University of Chicago Press, Chicago, 1970

Kw01	Kaiser W., Become a programming Picasso with JHotDraw, JavaWorld, February 2001, Available from www.javaworld.com/javaworld/jw-02-2001/jw-0216-jhotdraw.html
Kz94	Kulpa Z., Diagrammatic Representation and Reasoning, Machine Graphics & Vision, Volume 3, 1994, Pages 77-103
LBW+94	Lohse G., Biolsi K., Walker N., Rueter H., A classification of visual representations, Communications of the ACM Volume 37, Issue 12, 1994, Pages 36-49
Ld94	Lea D., Christopher Alexander: An introduction for object-oriented designers, Software Engineering Notes, Volume 19, Issue 1, January 1994, Pages 39-46
LH92	Lim B., Hunter R., DBTool: a graphical database design tool for an introductory database course, Proceedings of the 23rd Technical Symposium on Computer Science Education, Kansas City, MO USA, March 5-6, 1992, Pages 24-27
Lk96	Lieberherr K., Adaptive Object-Oriented Software, PWS Publishing Co., 1996
Ln97	Labeke N., Prise en compte de l'usager enseignant dans la conception des EIAO: Illustration dans Calques 3D, Ph.D. Thesis, Henri Poincare University, Nancy, France, 1997
LB81	Lientz B., Burton E., Problems in application software maintenance, Communications of the ACM, Volume 24, 1981, Pages 763–769
LS87	Larkin J., Simon H., Why a diagram is (sometimes) worth ten thousand words, Cognitive Science, 1987, Pages 65-99
Lt95	Lewis T., Object Oriented Application Frameworks, Manning Publications Co., 1995
LVC89	Linton M., Vlissides J., Calder P., Composing User Interfaces with InterViews, IEEE Computer, February, 1989, Pages 8-24, Available from citeseer.nj.nec.com/linton89composing.html
Mac95	Macromedia, Macromedia Authorware Version 3: Using Authorware, Macromedia Inc., 1995
Mb88	Meyer B., Object-Oriented Software Construction, Prentice Hall, New York, 1988
Mb90	Myers B., A new model for handling input, ACM Transactions on Information Systems, Volume 8, Issue 3, 1990, Pages 289-320
Mb98	Myers B., A Brief History of Human Computer Interaction Technology, ACM interactions, Volume 5, Number 2, March, 1998, Pages 44-54

Md99	Mackenzie, D., Recent developments in the Tripartite Interactive Assessment Delivery System (TRIADS), Proceedings of the 3rd Annual Computer Assessment Conference, Loughborough, UK, June 16-17, 1999
MGC98	Mansouri F., Gibbon C., Higgins C., PRAM: Prolog Automatic Marker, Proceedings of the 6th Annual Conference on the Teaching of Computing/3rd Annual Conference on Integrating Technology Into Computer Science Education on Changing the Delivery of Computer Science Education, Ireland, 1998, Pages 166–170
MHC+96	Myers B., Hollan J., Cruz I., Bryson S., Bulterman D., Catarci T., Citrin W., Glinert E., Grudin J., Ioannidis Y., Strategic directions in human-computer interaction, ACM Computing Surveys, Volume 28, Issue 4, 1996, Pages 794-809
MHG98	Mugridge, W. B., Hosking, J. G. and Grundy, J. C., Vixels, CreateThroughs, DragThroughs and Attachment Regions in BuildByWire, in Computer Human Interaction (CHI'98), December, 1998
MHP00	Myers B., Hudson S., Pausch R., Past, present, and future of user interface software tools, ACM Transactions on Computer-Human Interaction, Volume 7, Issue 1, 2000, Pages 3-28
Mic01	Microsoft Corporation, Microsoft Word 2000, 2001, Available from www.microsoft.com
Mic94	Microsoft Corporation, Visual C++ and Microsoft Foundation Class Library Manuals, Microsoft Press, 1994
Mic95	MicroGraphics Inc., ABC FlowCharter User's Manual, MicroGraphics Inc., 1995
Mj00	Morley J., Animating the execution of programs, B.Sc. Dissertation, Computer Science Department, The University of Nottingham, 2000
MM85	Martin J., McClure C., Diagramming techniques for analysts and programmers, Prentice-Hall International, Englewood Cliffs, New Jersey, 1985
MMM95	Milli H., Milli F., Milli A., Reusing software: Issues and research directions, IEEE Transactions On Software Engineering, Volume 21, Issue 6, 1995, Pages 529-561
Mn01	Miller N., A Diagrammatic formal system for eucliden geometry, Ph.D. Thesis, Cornell University, 2001
MO95	Martin J., Odell J., Object-Oriented Methods: A Foundation, Prentice Hall, Englewood Cliffs, New Jersey, 1995

Mp00 Machanick P., Experience of Applying Bloom's Taxonomy in Three Courses, Technical Report, TR-Wits-CS-2000-7, May, 2000

MP70 Moder J., Phillips C., Project management with CPM and PERT, Van Nostrand Reinhold Company, 1970

Mr86 Myers R., Computerized Grading of Freshman Chemistry Laboratory Experiments, Journal of Chemical Education, Volume 63, 1986, Pages 507-509

Mt76 McCabe T., A Software Complexity Measure, IEEE Transaction in Software Engineering, December, 1976, Pages 308-320

MV95 Minas M., Viehstaedt G., DiaGen: A generator for diagram editors providing direct manipulation and execution of diagrams, Proceedings of the International Conference on Visual Languages (VL'95), Darmstadt, Germany, 1995, Pages 203-210

MW98 Mason D., Woit D., Integrating technology into computer science examinations, Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education, Atlanta, GA USA, February 26 - March 1, 1998, Pages 140-144

MW99 Mason D., Woit D., Providing mark-up and feedback to students with online marking, Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education, March 24–28, 1999, New Orleans, LA USA, Pages 3-6

Nh97 Narayanan H., Diagrammatic communication: A taxonomic overview, Perspectives on Cognitive Science, Volume 3, New Bulgarian University Press, 1997

Nj94 Nickerson J., Visual Programming, Ph.D. Dissertation, New York University, 1994

Nk00 Nørmark K., A suite of WWW-based tools for advanced course management, 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, Helsinki, Finland, July 11 – 13, 2000

Np98 Neumann P., Risks of e-education, Communications of the ACM, Volume 41, 1998, Page 136

NR68 Naur P., Randell B., NATO Conference on Software Engineering, Garmish, Germany, October 1968

Op92 Opdyke W., Refactoring Object Oriented Frameworks, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1992

Oe80 Oviedo E., Control flow, data flow and program complexity, Proceeding of IEEE COMPSAC, 1980, Pages 146-152

Omg96	Object Management Group (OMG), The Common Object Request Broker: Architecture and Specification (CORBA), Revision 2.0, OMG, 1995
Osf91	Open Software Foundation, OSF/Motif Programmer's Guide, version 1.1, 1991
p99	President's Information Technology Advisory Committee, Information Technology Report: Inversting in our Future, Report to the President, February 1999, Available from www.ccic.gov/ac/report/
Pac01	PaceStar Software, EDGE User's Manual, PaceStar Software, Phoenix, USA, 2001, Available from www.pacestar.com
Pc33	Peirce C., Collected Papers, Volume 4, Harvard University Press, 1933
Pc65	Petri C., Kommunikation mit Automaten, Ph.D. thesis, Translation by Greene C. F., Supplement to Technical Report RADC-TR-65-337, Volume 1, Rome Labs, Griffiss Air-Force Base, New York, USA, 1965
Pc99	Power C., Designer--a logic diagram design tool, Proceedings of the 4th Annual SIGCSE/SIGCUE on Innovation and Technology in Computer Science Education, Krakow, Poland, June 27 – 30, 1999, Page 211
Pp00	Plasman P., private email sent to LTR, University of Nottingham, april, 2000
PS98	Preston J.A., Shackelford R. A system for improving distance and large-scale classes, Proceedings of the 6th Annual Conference on the Teaching of Computing/3rd Annual Conference on Integrating Technology into Computer Science Education on Changing the Delivery of Computer Science Education, Dublin, Ireland, 1998, Pages 193–198
Qm01	QuestionMark Inc, Question Types Used to Computerised Assessments, QuestionMark Documentation, 2001, Available from www.questionmark.com/us/learningcafe/index.htm
Ra96	Riel A., Object Oriented Design Heuristics, Addison Wesley, 1996
Rat00	Rational Software Corporation, Rational Rose 2000, Available from www.rational.com
Rb96	Randell B., The 1968/69 NATO Software Engineering Reports, Dagstuhl Seminar 9635 organized by Aspray W., Keil-Slawik R. and Parnas D., August 26-30, 1996
RBP+91	Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W., Object-Oriented Modelling and Design, Englewood Cliffs, New Jersey: Prentice Hall, 1991, ISBN: 0-13-630054-5

RH83	R. M. Rottmann and H. T. Hudson, Computer Grading As an Instructional Tool, Journal of College Science Teaching, Volume 12, 1983, Pages 152-156
Rj67	Raskin J., A Hardware-Independent Computer Graphics System, Ph.D. thesis, Computer Science Department of Pennsylvania State University, 1967
Rj84	Richards, J., Diagrammatics: an investigation aimed at providing a theoretical framework for studying diagrams and for establishing a taxonomy of their fundamental modes of graphic organisation, Ph.D. thesis, Royal College of Art, UK, 1984
RJB99	Rumbaugh J., Jacobson I., Booch G., The Unified Modelling Language Reference Manual, Addison Wesley, Reading, 1999
Rl00	Rising L., Pattern Almanac 2000, Addison Wesley, ISBN 0-201-61567-3, 2000
Rl96	Rosenstein L., MacApp: First Commercially Successful Framework, in Object Oriented Application Frameworks, Lewis T. (editor), Manning Publications Co., 1995, Pages 112-114
RMB97	Riehle D., Martin R., Buschmann F., (editors), Pattern Languages of Program Design, Volume 3, Reading, Massachusetts, Addison-Wesley, 1997
Rr63	Rossheim R., Report on proposed American standard flowchart symbols for information processing, Communications of the ACM, Volume 6, Issue 10, 1963, Pages 599-604
RSH01	Rangarajan K., Swaminathan N., Hedge V., Jacob J., Product Quality Framework: A Vehicle for Focusing on Product Quality Goals, ACM SigSoft, Software Engineering Notes, Volume 26, Number 4, July, 2001, Pages 77-82
Sa98	Sinan A., UML in a Nutshell, O'Reilly, 1998
Sb83	Shneiderman, B., Direct Manipulation: A Step Beyond Programming Languages, IEEE Computer, Volume 16, Issue 8, 1983, Pages 57-69
SCB+96	Summons P., Coldwell J., Bruff C., Henskens F., Automated assessment and marking of spreadsheet concepts, Proceedings of the Second Australasian Conference on Computer Science Education, Sydney, Australia, 3 – 5 July, 1996, Pages 178–184
SHG+98	Souder D., Herrington M., Garg P., DeRyke D., JSPICE: A component-based distributed Java front-end for SPICE, Proceedings of the 1998 Workshop on Java for High-Performance Network Computing, 1998

Si63	Sutherland, I., Sketchpad: A Man-Machine Graphical Communication System, in AFIPS Spring Joint Computer Conference, Volume 23, 1963, Pages 329-346
Si96	Sommerville I., Software Engineering, 5th Edition, Addison-Wesley, 1996
SJF96	Schmidt D., Johnson R., Fayad M., Software Patterns, Communications of the ACM, Special Issue on Design Patterns, Volume 39, Number 10, October, 1996
SL93	Schonwalder J., Langendorfer H., Ined - an application independent network editor, In Proceedings of the International Conference On Tools and Techniques for System Administration, Networking, and Security, Arlington, Virginia, USA, April, 1993
SM00	Sanders I., Mueller C., A fundamentals-based curriculum for first year computer science, Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education, Austin, TX USA, March 7–12, 2000, Pages 227-231
SM88	Shlaer S., Mellor S., Object Oriented System Analysis: Modelling the World in Data, Prentice Hall, Englewood Cliffs, New Jersey, 1988
SM92	Simas R., McBeath R., Constructing multiple choice items, In Instructing and Evaluating in Higher Education, Educational Technology Publications, Englewood Cliffs, New Jersey, 1992
SM97	Stephens, D., Mascia, J., Results of a Survey into the Use of Computer Assisted Assessment in Institutions of Higher Education in the UK 1995, Loughborough Flexible Learning Initiative, ISBN 095332 1029, 1997, Page 24
Sma01	SmartDraw.com, SmartDraw User's Guide, Version 5, 2001, SmartDraw, San Diego, USA, Available from www.smartdraw.com
SMM+96	Sharp H., Manns, M., McLaughlin P., Prieto, M., Dodani M., Pedagogical Patterns - Successes in Teaching Object Technology, ACM SIGPLAN Notices, Volume 31, Issue 12, December, 1996, Pages 18-21, Available from www.cs.unca.edu/~manns/oopsla.html
SN73	Nassi I., Schneiderman B., Flowchart Techniques for Structured Programming, ACM SIGPLAN Notices Journal, Volume 8, 1973, Pages 12-26
Sp01	Symeonidis P., An in-depth review of CourseMaster Marking SubSystem, Technical Report, LTR Group, University of Nottingham, UK, 2001

Sp96	Sommerlad P., Command Processor, in Pattern Languages of Program Design, Volume 2, editors Vlissides J., Kerth N., Coplien J., Addison-Wesley, Reading, Massachusetts, 1996
SRJ97	Smith T., Ruocco A., Jansen B., Digital video in education, Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education, New Orleans, LA USA, 1997, Pages 122-126
Ss93	Schach D., Software Engineering, 2nd Edition, Richard D. Irwin, Inc. and Aksen Associates, 1993
Sun01	SUN Microsystems Inc., Java 2D API White Paper, 2001, Available from java.sun.com/products/java-media/2D/whitepaper.html
Sun96	Sun Microsystems Inc., RMI Documentation, 1996, Available from www.javasoft.com/apis
Sun98	Sun Microsystems Inc., Java Foundation Classes Specification, 1998, Available from java.sun.com/products/jfc
Sun99	Sun Microsystems Inc., Enterprise JavaBeans Specification 1.1, 1999, Available from java.sun.com/products/ejb
Ta01	Tsintsifas A., The DATsys Design Document, Technical Documentation, Computer Science Department, University of Nottingham, May, 2001, Available from www.cs.nott.ac.uk/~azt/daidalos/docs/design.html
Ta97	Tuovinen A., Towards a Framework for Developing Processors for Visual Languages, European Conference on Object Oriented Pprogramming (ECOOP'97), 1997
Tb93	Buzan T., The Mind Map Book: Radiant Thinking - the major evolution in human thought, BBC Publications, 1993
TBF97	Tinoco L., Barnette D., Fox E., Online evaluation in WWW-based courseware, Proceedings of the 28th SIGCSE technical symposium on computer science education, San Jose, CA USA, February 27 - March 1, 1997, Pages 194–198
TD76	Taylor J., Deever D., Constructed-response, computer-graded homework, American Journal of Physics 44, Pages 598-599
Td99	Tsichritzis D., Reengineering the university, Communications of the ACM, Volume 42, 1999, Pages 93-100
Tek87	Tektronix Computer Research Laboratory, Semantic Drawing with HotDraw, Technical Report CR-87-34, April, 1987
Tk00	Tourlas K., Diagrammatic Representations in Domain Specific Languages, Ph.D. thesis, Computer Science Department, University of Edinburgh, 2000

TS00	Tsintsifas A., Symeonidis P., The CM Marking System Design Document, Technical Documentation, Computer Science Department, University of Nottingham, January, 2000, Available from www.cs.nott.ac.uk/CM/docs/marking/index.html
Ts90	Tanimoto S., VIVA: a visual language for image processing, Journal of Visual Languages, Computing, Issue 2, June, 1990, Pages 127-139
TS99	Tsintsifas A., Symeonidis P., The CourseMaster Design Document, Technical Documentation, Computer Science Department, University of Nottingham, January 2000, Available from www.cs.nott.ac.uk/CM/docs/design.html
UML97	The UML Group, Unified Modelling Language, version 1.0, Rational Software Corporation, Santa Clara, CA-95051, USA, January 1997
UML99	Rational Corporation, UML documentation, version 1.3, Available from www.rational.com/uml/
Up02	Uportal Group, Java in Administration Special Interest Group, 2002, Available from www.ja-sig.org
Va94	Vladimirescu A., The SPICE Book, John Wiley, New York, 1994
Vb98	Venners B., Inside the Java Virtual Machine, McGraw-Hill, 1998
Vg95	G. Viehstaedt, A Generator for Diagram Editors, Ph.d. thesis, University of Erlangen-Nürnberg, Band 28, Number 2, 1995
Vj80	Venn J., On the diagrammatic and mechanical representation of propositions and reasonings, Philosophy Magazine, 1880, Volume 123
Vj90	Vlissides J. Generalized Graphical Object Editing, Ph.D. thesis, Stanford University, 1990
Vj98	Vlissides J., Pattern Hatching: Design Patterns Applied, Reading, Addison-Wesley, Massachusetts, 1998
VKC96	Vlissides J., Kerth N., Coplien J., (editors), Pattern Languages of Program Design, Volume 2, Addison-Wesley, Reading, Massachusetts, 1996
VL89	Vlissides J., Linton M., Unidraw: A Framework for Building Domain-Specific Graphical Editors, Technical Report CSL-TR-89-380, Stanford University, July 1989
Vp85	Verth, P., A System for Automatically Grading Program Quality, Technical Report, Buffalo, 1985

WG95	Weinand A., Gamma E., ET++, A Portable Homogeneous Class Library and Application Framework, In Lewis T., Object Oriented Application Frameworks, Manning Publications Co., 1995, Chapter 7, Page 154-194
WH99	Whittington D., Hunt H., Approaches to the computerized assessment of free text responses, Proceedings of the 3rd Annual Computer Assessment (CAA) Conference, Loughborough, UK, June 16-17, 1999
WJ90	Wirfs-Brock R., Johnson R., Surveying current research in object-oriented design, Communications of the ACM, Volume 33, 1990, Pages 104–124
Wr01	Williams R., Automated essay grading: An evaluation of four conceptual models, In Herrmann A. and Kulski M. (Editors), Expanding Horizons in Teaching and Learning, Proceedings of the 10th Annual Teaching Learning Forum, Perth, Curtin University of Technology, February 7-9, 2001
Wr98	Wieringa R., A survey of structured and object-oriented software specification methods and techniques, ACM Computing Surveys, Volume 30, Issue 4, 1998, Pages 459-527
WRS90	Wilson D., Rosenstein L., Shafer D., Programming with MacApp, Addison-Wesley, Reading, Massachusetts, 1990
Wt79	Winograd T., Beyond programming languages, Communications of the ACM, Volume 22, Number 7, July, 1979, Pages 391-401
WWW90	Wirfs-Brock R., Wilkerson B., Wiener L., Designing Object-Oriented Software, Prentice Hall, New Jersey, 1990, ISBN 0-13-629825-7
ZF91	Zin A. M., Foxley E., Automatic Program Quality Assessment System, Proceedings of the IFIP Conference on Software Quality, SP University, Vidyanagar, India, 1991