Department of Computing, Imperial College, London

Course-descriptions - pre 2001 entrants

Hardware

Lecturer(s): Derek Brough and Duncan Gillies

To provide Computing Science students with a sound introduction to the fundamental electrical principles and devices used in the design of digital computers, and to the way in which primitive control logic can be organised to construct a programmable machine.

Boolean algebra; combinatorial logic functions; principles of CMOS semiconductor devices; CMOS logic gates; tri-state gates; adders and subtractors; bistable storage device; S-R flip-flop; D-type flip-flop; latch versus edge triggering; J-K flip-flops; registers; shift registers; counters; finite state machine design; PLA; multipliers; static and dynamic RAM; ROM; CAM; magnetic storage.

Practical Laboratory work: Characteristics of discrete MOSFETs; static and dynamic behaviour of basic CMOS gate and flip-flops; use of VLSI design tools to build and test simple circuits.

Discrete Mathematics I

Lecturer(s): Philippa Gardner

To introduce concepts and proof techniques of basic set theory. To introduce certain classes of relations and graphs. To introduce algorithm analysis.

Basic set theory: sets, relations, functions, cardinality, inverse, composition, bijection.

Relations: equivalence relations, orderings.

Logic

Lecturer(s): Ian Hodkinson

AIMS
To introduce the language and semantics of propositional and first order predicate logic, and natural deduction. To apply concepts of first order logic to program specification.

LEARNING OUTCOMES
Knowledge and understanding:
Students should understand and recall the definitions of the logics and logical systems presented in the course. They should know the difference between syntax and semantics. They should understand the meanings of the basic logical symbols. They should be able to read and understand logical formulas. They should understand the mechanisms of proof systems presented in the course. They should understand terms and notation commonly used in elementary logic texts.

Skills and other attributes:
a) intellectual skills
Students should be able to parse logical formulas correctly, evaluate them in given situations, translate them into English, and write formulas expressing ideas given in English. They should be able to undertake simple proofs using proof systems presented in the course.
b) practical skills
Students should be able to understand and construct pre- and post-conditions for simple programs.
c) transferable/key skills
As above.

SYLLABUS
Propositional and predicate logic:
syntax, informal and formal semantics. Validity, satisfiability, semantic entailment, equivalence. Many-sorted logic. Reasoning methods. Applications in program specification.

Declarative Programming I

Lecturer(s): Tony Field

Academic aims:
This course aims to teach the fundamentals of programming and problem solving through very high-level programming languages (Haskell and Prolog). This enables students to solve small-scale problems succinctly and at at an abstract mathematical level without being bogged down with the syntax and semantics of a conventional procedural or object-oriented language. The course is taught using a problem-solving approach with students being encouraged to use the various language features to solve realistic problems and to explore some fundamental algorithms and data structures in computer science. The course contains a strong practical element with weekly assessed laboratory exercises and supplementary unassessed problems.

Learning Outcomes:
By the end of the course the student will have a working knowledge of: Mathematical variables and scope; Functions, assertions and Relations; Basic data types; Recursive functions; Algebraic data types; Polymorphism and overloading; List processing; Higher-order functions; Lazy and eager evaluation. The student will be able to use this knowledge to solve small-scale problems in both Haskell and Prolog using the most appropriate features of both languages.

Pre/Co-requisites: None

The course is a pre-requisite for all the subsequent programming/software engineering courses.

Programming I

Lecturer(s): Susan Eisenbach

Aim: To impart practical skills in the specification, design, coding and testing of small programs. To teach the Turing programming language. To introduce design and object oriented programming.

Learning Outcomes: Students will be able to write Object Oriented Turing programs to solve small problems using the appropriate contril and data structures.

Syllabus: Specification and design: English specifications, Haskell specifications of Java programs and stepwise refinement. Programming in Object Oriented Turing: control, data-structures, simple algorithm construction, object orientation, testing, debugging.

Discrete Mathematics II

Lecturer(s): Iain Phillips

Graphs: Basic definitions, isomorphism, Eulerian paths, graph colouring, Dijkstra's shortest path algorithm.

Algorithm analysis: time complexity, worst case and average case analysis, optimality, orders, decision trees, recurrence relations, examples from searching and sorting.

Reasoning about Programs

Lecturer(s): Krysia Broda

Aims:
(i) To gain familiarity with use of pre and post conditions and loop invariants for showing correctness.
(ii) To learn some standard algorithms and be able to reason about their correctness.
(iii) To understand mathematical induction and apply it to reasoning about Haskell programs.

Learning Outcomes On completion, a student will
- be able to use mathematical induction and structural induction to show that Haskell programs meet their specification;
- be able to reason with pre and post conditions and to use the method of loop invariants to show correctness of programs;
- be familiar with, and understand the development of, some common algorithms, including binary chop, partition and quicksort, Warshall's algorithm and variations, string searching (including Boyer Moore algorithm) and to reason about them.

Syllabus:
To introduce rigorous reasoning in the specification and design of programs.
Induction: mathematical induction, structural induction.
Formal program techniques: specification by pre- and post-conditions, derivation and verification of programs, invariants, proofs of program properties. Conversion of recursion to iteration.
Common algorithms as examples (e.g. binary chop, Warshall's algorithm , partition and quicksort, string searching)

Prerequisites: None, Corequisites: 140, 120. Required for: C240 Algorithms and Complexity, C220 Software Engineering I, C302 Software Engineering - Methods

Operating Systems

Lecturer(s): William Knottenbelt and Julie McCann

To answer and develop the ability to answer the following questions: What is the purpose of an operating system? How are operating systems constructed? Introduction: resource manager view, virtual machine view, types of operating system. Evolution of operating systems: user driven, operator driven, simple batch system, off-line batch system, directly-coupled off-line system, multiprogrammed spooling system, on-line timesharing system, multi-processor systems, multi-computer/distributed systems. Program construction utilities: assembler, archiver, link editor, relocating loader.

Concurrent processes: interleaving, non-determinism, process interaction - sharing, synchronisation, communication, locks, semaphores, monitors.

The system nucleus (kernel): context switching, first level interrupt handling, kernel implementation of processes, Kernel implementation of semaphores.

Scheduling: priority - pre-emption, run to completion, time sliced, multi-level queues.

Input/output: polled input/output, interrupt driven input/output, device driver structure.

Memory management: single contiguous store allocation and overlays, fixed partition store allocation, dynamic partition store allocation and fragmentation/compaction, virtual addressing, memory management policies

Declarative Programming II

Lecturer(s): Christopher Hogger

To develop an understanding of logic as a declarative computational formalism. To be able to use Prolog, a logic programming language, to formulate and solve the kind of problems commonly met in artificial intelligence applications.

Introduction to logic programming using Prolog: syntax of terms and rules, procedural interpretation, non-determinism and search, unification and answer extraction. Type checking, list processing and arithmetic. Generate-and-test algorithms, aggregation, tree-pruning. Introduction to meta-programming.

Programming II

Lecturer(s): Margeret Cunningham

Prerequisites: Programming I

Aims of the course:
To help students gain a good understanding of, and ability to use, abstract data types; familiarise students with common abstract data types and associated operations.

To teach various design and implementation solutions for abstract data types; illustrate the practical effects of the different implementation choices; and illustrate their practical use in developing object oriented programs for real application problems.

Learning Outcomes:
Knowledge and understanding upon successful completion of the module, a student will:
- understand basic principles, main features and operations of abstract data types, in particular of lists, stacks, queues, trees, heaps, hash tables and graphs.
- differentiate specifications of abstract data types from particular implementation techniques.
- learn about fundamental algorithms associated with the above data types, including tree traversal, treesort, heapsort and graph traversal algorithms.

Intellectual and practical skills upon successful completion of the module, a student will:
- be able to develop Object Oriented Turing implementations of abstract data types using different approaches, and evaluate their differences.
- be able to use abstract data types and related implementations in designing and implementing efficient solutions to straightforward application problems.

Outline syllabus:
Introducing Abstract Data Types, a brief introduction to data abstraction and definition of the concept of Abstract Data Types (ADT).

Lists Definition of lists and associated operations for manipulating individual elements or entire lists, such as addition and deletion of an element, search for an element, replacement of an element in a list, computing the length of a list, sorting a list and copying a list. Array and reference-based implementation techniques. Examples of list variations, including double linked lists, circular lists.

Stacks, queues and recursion. Definition of stacks and associated operations. Array and linked-list implementation of stacks. A brief introduction to iteration and recursion. Queue and related access operations. Array and linked-list implementation of queues. Priority queues and double-ended queues. Example applications of stacks and queues.

Trees Definition of binary trees and associated operations. Array and reference-based implementation of binary trees. Various examples of tree traversal algorithms. Binary search trees and related operations such as finding, inserting, removing an element. Illustration of the Treesort algorithm.

Heaps and hash tables Definition of heaps and basic operations. Priority queues with heaps. Illustration of the Heapsort algorithm. An introduction to hash tables, hash function, and illustration of some of the issues related with the choice of hash functions and the use of double hashing.

Graphs Introduction of undirected and directed graphs and implementation of graphs. Illustration of basic graph algorithms such as depth-first and breadth-first search.

Directions

Lecturer(s): Abbas Edalat

The aim of this course is to introduce new students to some of the state of the art ideas they will meet later on in their degree programme. It gives an introduction to specific topics in several areas of specialisation.
Through guided individual study and supporting lectures it encourages students to work on their own and use reference material to expand their knowledge in a few of the areas introduced by the lectures.

Architecture I

Lecturer(s): Naranker Dulay

Introduction: Relationship to other courses, levels of abstraction, instruction set level, hardware design level, role of the computer architect.

Data representation: binary numbers, arithmetic, octal, hex, base conversion, sign and magnitude, 1's complement and 2's complement, BCD overflow, characters, ASCII/Unicode.

Main Memory Organisation: byte and word addressing, byte ordering, alignment, banks and interleaving.

CPU organisation and operation: simple CPU (TOY1), instructions, fetch-execute cycle, simple assembly programming.

Pentium architecture: programming model, registers, memory models, addressing modes, arrays, records, instructions, expressions, loops, procedures.

Input and output: device types and characteristics, controllers, ports, programmed I/O, interrupts, DMA, Pentium interrupt model, simple device drivers.

Floating point: conversion, normalisation, arithmetic operations, representation errors, overflow/underflow

IEEE standard: format, arithmetic, NANs, Infinity and denormalised values.

Mathematical Methods and Graphics

Lecturer(s): Duncan Gillies

The students should become proficient in analytic techniques involving functions of a continuous variable and have a good grasp of linear algebra. They should understand the relationship between discrete processes and continuous ones, and be able to model either one with the other. They should understand the importance of efficiency and reliability for practical algorithms. They should be able to apply established mathematical techniques to set up and draw both polyhedral objects and smooth surfaces. They should gain a clear understanding of the graphical techniques used, for example, in modern flight simulators.

Foundations of mathematical analysis: complex numbers, recurrence relations, convergence of sequences and series. Comparison test/absolute convergence. Power series/radii of convergence. Taylor's theorem in one variable.

Linear algebra: matrices and vector notation; solution of linear equations by Gaussian elimination, LU decomposition. Finite precision arithmetic and effect on computations. Eigenvectors and eigenvalues.

Several variables: partial derivatives, stationary points, chain rule.

Mathematical methods: contraction mapping, polynomial approximation of functions. Numerical approximation of functions. Various applications for scientific computing.

Vector algebra: vector addition, multiplication, dot and cross products; three dimensional geometry; algorithms for representing and drawing planar polyhedra; texture mapping.

Interpolation algorithms: Cubic spline curves and surfaces. Bezier curves and surfaces; the floating horizon algorithm.

Algorithms, Complexity and Computability

Lecturer(s): Margaret Cunningham

At the end, students should be able to: recall, evaluate and assess methodology of the Turing machine formalisation; construct and use appropriate Turing machines, and structure complex Turing machines from identified components; understand equivalence of generalisations of Turing machine; explain algorithmic unsolvability of certain problems; recall specified algorithms and undertake rigorous correctness proofs and informal complexity analyses, and classify paradigm examples.

This course will formalise the fundamental idea of a computable problem, by using Turing machines, simple but powerful idealised computers. Examples will be given. The relation of Turing machines to variants such as multi-tape machines will be studied. The limits of computation will be examined (Church's thesis, unsolvable problems, reducibility theory).

The second main topic will be complexity theory, which measures the relative difficulty of various solvable problems in a formal way, by P-time reduction. Non-deterministic Turing machines will be introduced, and problems (TSP, HCP, etc.) solvable (non-deterministically) in polynomial time will be considered. We will examine the famous question P=NP of whether non-determinancy matters. NP-complete problems and Cook's theorem on satisfiability of propositional formulas will be discussed. Some useful graph algorithms will be examined.

Compilers

Lecturer(s): Naranker Dulay and Paul Kelly

To develop an understanding of how a compiler for a high-level programming language works, how programming language design is influenced by compiler structure, and how computer architecture is influenced by the needs of compiled programs. The course provides the specific technical skills needed for constructing parsers, interpreters and translators as well as introducing topics in code optimisation and semantic analysis.

Language processors, compilers, interpreters and their relatives. The structure of a compiler. Its context: editors, and loaders. Phases and passes.

Code generation for arithmetic expressions, using a stack and using registers. The importance of effective use of registers. Sethi-Ullman numbers and the graph colouring approach. Code generation for statements, control structures and logical expressions. Short-circuiting logical operators. Dynamic and static instruction counts.

Semantic analysis: variable and record declarations and the use of a symbol table. The scope of variable declarations in a block-structured language. Activation records. Lexical depth. Static and dynamic link chains. Nested procedure declarations and calls. Parameter passing, by value and by reference. Run-time support, dynamic storage and garbage collection.

Grammars and the parsing problem. Classes of grammars. Top down parsing, backtracking. Factoring and removal of left-recursion. Analysing the structure of a grammar; context clashes. Lexical specification and analysis. Regular expressions and their relationship to context-free grammars and finite-state automata.

Types and type checking. Dynamic, static and strong type checking. Overloading and polymorphism.

Optimisation: peephole, basic block, inter-block ('global') and interprocedural. Strength reduction. Side-effects and aliasing.

Concrete versus abstract syntax. Interpreters. Name scope: environments; static versus dynamic scoping. Semantic checking: type checking. Translation: decompilation; pretty-printing. Code generation: stack machines; assembly language; run-time stacks and block structure; static and dynamic chains; global variables; register machines. Optimisation: register allocation; constant folding; Boolean expressions; assignment; statements; loops; invariant code. Data structures: arrays; generators; element access; bounds checking; common subexpressions; base and index register use; dope vectors. Syntax analysis: BNF; conctext-free grammars; recognisers; parse trees. Top-down parsing; recursive descent; ambiguity; left recursion; empty productions; backtracking and one-symbol-lookahead grammars. Warshall's algorithm; context clashes; producing abstract syntax trees. Bottom-up parsing; operator precedence; operator grammars; constructing precedence matrices. Lexical analysis: finite-state automata.

Databases I

Lecturer(s): James Jacobson and Marek Sergot

To introduce database systems with particular reference to the relational model, including design, query languages and update transactions. To introduce entity-relationship modelling and translation to the relational model.

Introduction to databases: data modelling, database management, data dictionary, query formulation and evaluation.

Relational databases: design, functional dependencies, keys and normal forms.

Relational database languages: relational algebra, tuple calculus. Views, integrity and security. Introduction to deductive relational databases.

Transaction management and recovery: transaction atomicity, database log, deferred and immediate database modifications, checkpoints. Concurrency: including serialisability, locking, two phase locking protocol, deadlock. Entity-relationship models.

The course will be supported by laboratory sessions using the relational database system, INGRES, and the language SQL.

Operating Systems II

Lecturer(s): Daniel Ruckert and Steffen van Bakel

Revision: Real time programming and relation to OS. v Definition & characteristics of real-time systems. OS Functions: resource management, providing a virtual machine. Non-determinancy

Concurrent Programming: Concurrency in languages. Process Interaction Mechanisms. Semaphores, Message passing.

The System Kernel: Hierarchical O.S. Structure. Kernel Functions & kernel entry. Process representation & states. First Level Interrupt Handler. Dispatcher & scheduling. Implementation of semaphores, and message primitives.

Device Management & Input Output: I/O software structuring. Device independent I/O. Device and interrupt handlers. Spooler and Buffering.

Time handling and Scheduling: Time handling facilities - delay and real-time clock. Scheduling strategies, priority preemption, & multilevel queues. Deadline specification and scheduling issues

Filing System: Directory Structures, Sharing. Space Management. Implementation Structure. Disc Access Scheduling

Security: Access control & protection domains. Access control lists and capabilities

Distributed Operating Systems: Characteristics of distributed systems. Client server, pipeline and other interaction paradigms. Interaction primitives. Name servers and file servers.

Software Engineering - Design I

Lecturer(s): Emil Lupu and Alessandra Russo

Aims:
· To provide in-depth understanding and practice of object-oriented programming.
· To teach the foundations of software development processes.
· To teach basic concepts of (object-oriented) formal specifications and enable students to acquire skills necessary to develop logic-based specifications of software systems.
· To provide the skills and knowledge necessary for systems modelling, design and analysis.

Learning Outcomes:
Knowledge: Upon completion of the course students would have acquired an in-depth knowledge of:
· Object-oriented programming, classes, interfaces, use of inheritance and polymorphism and the role of software architectures.
· Foundations of software development processes and the role of design in the software development life-cycle.
· Systems modelling, analysis and design across both architectural and behavioural specifications.
· Modelling and development methodology.
· Principles and techniques for the engineering of large software projects.
· Fundamental principles of formal specifications, including state, operation and class schemas.

Skills: The course will develop a wide range of skills necessary for the design and development of software projects. In particular, students will develop the skills necessary for:
· Decomposing problems and designing software architectures.
· Producing static and behavioural models of software programs.
· Applying software design methodologies.
· Developing formal specifications from informal requirements of software systems.
· Implementing software models in a structured and efficient way.

Prerequisites All 1st year programming, and the 1st year logic course

Co-Requisites java Lab

Courses for which the course is a pre-requisite all subsequent software engineering courses

Syllabus :
The course stresses the pragmatic skills needed in object-oriented program development and software design. It teaches the basic principles and techniques for modelling, analysing and designing software projects.

Principles of good software design, software development life-cycle, the role of design and modelling in software development.

Object-oriented programming in Java: objects and classes, interfaces, inheritance, polymorphism, exceptions.

Software design in UML: objects and links, classes and associations, aggregations and dependencies, structural and behavioural modelling, state-charts.

Formal specifications: definition of state and operation schemas, operation exceptions, class schemas with related issues including visibility and initialisation; class derivations and associated properties.

Design methodology.

Statistics

Lecturer(s): David Hand

The aim of this course is to equip students to make basic statistical analyses of data, and to enable them to critically assess and interpret others' analyses. Detailed handouts were given during the course, as well as extensive problem sheets.

1. INTRODUCTION What statistics is about: discovering structure in data Why the popular misconception is wrong: modern statistics, greater statistics, statistics as the ultimate scientific instrument. Relationships between statistics and computer science through the overlapping areas of data mining, neural networks, machine learning, pattern recognition, etc. Examples of the sorts of projects my research students are working on. The space shuttle Challenger disaster as a motivating example. Examples of data from computer science (including probability, reliability, large databases, knowledge discovery, machine learning, etc.) Examples of questions which need statistical answers

2. SIMPLE GRAPHICAL DISPLAYS Bar charts, histograms, pie charts, scattergrams, time series plots, contour plots, etc. How to (not to) cheat with graphs and how to detect when others are cheating.

3. SIMPLE NUMERICAL SUMMARIES Frequency distributions, stem and leaf plots Measures of location: mean, median, quartiles, mode Measures of dispersion: range, variance (divide by n or n-1), standard deviation, interquartile range Five figure summary Box plots Skewness, tails of distributions

4. PROBABILITY Motivation: inference (drawing conclusions from samples in the context of natural variation in the population being studied), modelling (characterising the main features of the mechanism underlying the process generating the data). Reminder of basic set theory Definition of probability Interpretations: classical, relative frequency, and subjective Addition law of probability Independent events, multiplication law of probability Conditional probability Bayes theorem

5. DISCRETE RANDOM VARIABLES Discrete random variables Probability functions, cumulative probability distribution Mean (expected) value, variance, and skewness Mean and variance of sums of independent random variables Discrete uniform distribution Bernoulli distribution Binomial distribution Geometric distribution Poisson distribution

6. CONTINUOUS RANDOM VARIABLES Cumulative distribution functions Probability density functions Uniform distribution Exponential distribution Normal distribution The standard normal distribution Use of tables for the normal distribution Joint, marginal, and conditional distributions

7. CENTRAL LIMIT THEOREM

8. HYPOTHESIS TESTS Null hypothesis, alternative hypothesis, significance level, p-value, test statistic, rejection region (critical region). The two types of error: Type I: Rejecting a true null hypothesis Type II: Failing to reject a false null hypothesis Prob (Type I) is denoted ; Prob (Type II) is denoted The power of a test is . The steps in testing a hypothesis: 1: State null hypothesis, alternative hypothesis, significance level. 2: Select test statistic 3: Define rejection region 4: Do experiment 5: State conclusion 'Reject' or 'fail to reject' null hypothesis t-distribution, degrees of freedom. Tables of the t-distribution t-tests for comparing means with hypothesised values. One-sided and two-sided tests Two sample t-tests for comparing means of two populations Paired comparison (or matched pairs) tests for comparing means of two matched populations. Goodness-of-fit tests for categorical variables: observed values and expected values The chi-square test statistic as an overall measure of discrepancy between observed values and the values one would expect if the hypothesised distribution was correct: , to be compared with a chi-squared distribution with d degrees of freedom, where d = k-p-1, where k is the number of categories and p is the number of parameters being estimated for the hypothesised distribution. Tables of the chi-squared distribution. Goodness-of-fit tests for continuous variables by grouping into categories and applying methods for discrete distributions. Chi-squared test for independence of two categorical variables: compute the expected values by multiplying the marginal probabilities (and rescaling by sample size). Then use standard chi-square test statistic above. Here, if there are r rows and c columns

in the cross-classification, then the chi-squared distribution with (r-1)(c-1) degrees of freedom should be used.

9. ESTIMATION Population quantities: parameters Sample quantities: statistics Point estimates and interval estimates An estimator - a formula, recipe, or algorithm for calculating the estimate from a sample Different samples will yield different results - each of these is an estimate. Since estimates will vary from sample to sample, they will have a distribution - the sampling distribution. The bias of an estimator. Method of least squares Method of maximum likelihood The likelihood function Confidence intervals and their interpretation

10. EXPLANATORY RELATIONSHIPS Linear regression, and least squares estimation of the parameters Correlation coefficient

11. RELIABILITY THEORY Survivor (or reliability) function, failure rate function, hazard function. The special case of the exponential distribution, and its properties Series and parallel systems and their reliability. 12. COMPUTING Using Splus: a brief introduction, with hands on experience.

Networks and Communications

Lecturer: Peter McBrien

Syllabus:

Introduction and basic concepts:
Applications of computer communications and types of data, Channels, bit rate and throughput, Network topologies, LANs, MANs and WANs, broadcast and point-to-point, Synchronous and asynchronous communication. Connection oriented and connectionless communication

Theoretical capacity of channels:
Shannon's law and the Nyquist relationship, Multiplexing, Queuing theory

Computer communication system architectures:

The OSI Reference Model:
overview of the seven layer model.

The TCP/IP Model:
Comparison with the OSI Model, overview of TCP/IP protocols.

The physical layer:
Transmission media. Properties of signals and signal degradation. Digital and analogue transmission, conversions between formats.

Error detection and correction Forward error control:
Hamming code.

Reverse error control:
parity, block sum check, cyclic redundancy check.

Data compression

The data link layer
Error control: Idle RQ and continuous RQ. Flow control

The medium access control sub-layer:
Slotted transmission, carrier sensing, token passing, distributed queues, IEEE MAC model and addresses, Ethernet: 10Mbps to 1Gbps. Token Ring. DQDB

The network layer:
Switching:
packet switched and circuit switched. Routing and Internetworking. Repeaters, bridges and routers flood routing, adaptive routing: link distance and backwards learning, Internet protocols (IPv4, IPv6) and ARP. ATM, The public telephone network: POTS and ISDN

The transport layer:
UDP and TCP. Quality of service

The session layer

The presentation layer:
Transport syntax: ASN.1 and ISO8825, Security: confidential and authenticated data, public key and private key cryptography, digital signatures, PGP, key escrow, firewalls.

The application layer:
OSI model protocols, DNS and URLs and FTP, HTTP, SMTP

Future directions:

Multimedia and the RTP

Software Engineering - Design II

Lecturer(s): Jim Cunningham and David Hand and Ian Harries

Aim:
To teach the principles of Human Computer Interaction and the design of interactive systems, including desktop windowing systems and networked e-commerce.

Syllabus:
A brief history of Human Computer Interaction: reasons for user-centered design and scientific experiment. Usability Engineering and usability case studies.

Evaluation methods: User reports - interviews and questionaires, specialist walkthroughs; Observational methods - scoping, level, and stages. Elements of experimental design and statistical factor analysis. Analytic methods - from keystroke analysis to cognitive goals.

Physical Interactions: input, output & the environment; human physical, visual, auditory and cognitive factors; elements of workplace design.

Dialogue design: Styles, models & metaphors, elements of windowing models. Normans stages of action, task analysis, dialogue definition. Visualisation case studies and principles Guidelines and standards for interactive system design.

Interface architectures and design tools: Single, multi-tasking and networked window system architectures, interface programming tools. The Java toolkit. Internet protocols, HTML, XML. Internet programming: CGI, Perl, Javascript. Database access - server side and remote.

Artificial Intelligence I

Lecturer(s): Derek Brough

To provide an introduction to the concepts of Artificial Intelligence. To discuss various AI techniques and investigate their application to a range of topic areas. A range of examples will be discussed from the following areas.

Introduction to AI: The two streams of AI; Intelligence, some characteristics; Classical AI examples; Need for tests of success, the Turing test, 'intelligence tests', peer assessment; the potential and danger of the AI approach.

Problem Solving: Constraint Satisfaction problems, Generate and test methods with optimisations; State Space problems. Search methods: Local vs global information; Heuristic Search. Two person games: Search and evaluation; Adversary search; Minimax search; Alpha Beta Search.

Expertise: Characteristics of Expert Systems, separation of Knowledge and Inference, Uniform representation, Explanation; Knowledge refinement; Expertise and knowledge; Accountability and the human window, human vs machine expertise.

Robotics: Vision, scene recognition and the trihedral world, Waltz constraint satisfaction algorithm; Speech synthesis, allophones, dictionary, morphology and rule based letter to sound approaches, stress and intonation.

Planning: viewed as state space search, means-ends heuristics; representation; separability & interactions, Sussman's anomaly; example worlds, Blocks world, robot world & program synthesis; Problems and extensions, Non linear planning.

Learning: rote, trial and error, Pavlovian and history learning; Learning by exploration, hill climbing, Learning g ames programs, credit assignment and term selection problems; linear and non-linear evaluation functions; learning by search through operator and concept spaces.

AI and Natural Language: Problem of understanding; Schank's explanation game; Conceptual Dependency (CD) theory, primitive elements; Scripts, players, props, events, headers and exceptions; Goal and plan directed understanding, role of themes.

Concurrent Programming

Lecturer(s): Jeff Magee

To study the concepts, methods and algorithms appropriate for the construction of concurrent programs.
Students will perform a laboratory exercise involving concurrent programming using Java.
Introduction to concurrent programming: Key ideas, interleaved actions, synchronisation, critical sections, deadlock, starvation, fairness, safety and liveness.
Monitors: entry queues, condition variables, wait and signal, alternative signalling mechanisms, reasoning about monitors.
Message Passing: processes, synchronous and asynchronous communication, ports, send and receive, request-reply communication, non-deterministic choice, configuration programming.
Specification: introduction to formalisms for the specification and verification of concurrent system; labelled transition systems, process calculi CCS, CSP.

Software structure: structuring applications into modular, distributable software components; component types and instances; nesting and dynamic structures; component interfaces; connection patterns and naming.

Architecture II

Lecturer(s): Wayne Luk

To build on the foundation laid by Architecture I; to show the relationship between hardware and software; to focus on the concepts that provide the basis for current computers.
Introduction: overview; performance. Instructions: formats, representations, interface with software.
Arithmetic: number representation; hardware for arithmetic operations; Arithmetic Logic Unit.
Datapath and control unit: single-clock and multiple-clock implementations; microprogramming; exception handling.
Memory hierarchy: caches, virtual memory.
Pipelining: pipelined datapaths, data and branch hazards, exceptions.
Advanced topics: hardware compilation, parallel architectures, special-purpose processors.

Software Engineering - Methods

Lecturer: Michael Huth

This course features state-of-the-art methods in software engineering practices from a managerial, technical, and ethical perspective.

First we present prominent and well proven agile and iterative development techniques, focusing on management and some programming aspects: we discuss the methods Scrum, Extreme Programming, the Unified Process, and Evolutionary Project Management and explore the history and utility of these methods.

On the more technical side as explore what solutions are currently available that aid implementors in achieving quality assurance of their code. We mostly study an approach of annotating source code with integrity constraints (object invariance, pre- and post-conditions etc) and survey some of the freeware tool support that is available at present. If time permits, we will give a brief introduction to the CASE

tool Rational Rose.

On the purely professional side, we identify some professional issues that may not have managerial or technical solutions. Notably we ask which problems have an ethical dimension and try to identify what an „ethical dimension“ is. We offer some exploration tools that help in recognizing and assessing ethical dilemmas and we apply these tools to some case studies.

Expected learning outcomes:
- Students will understand the core values and practices of key agile and iterative development methods, be familiar with their history (invention and cornerstone projects), and appreciate the advantages and trade-offs of these methods.

- Students will be competent in judging which agile development methods can be mixed, and to what extent, on a successfully managed project.

- Students will know what technological support is currently available for support of software quality assurance in implementation work. Notably they will be able to download and use tools for annotating Java and C# programs with integrity constraints such as object invariance and pre- and post-conditions for methods.

- Students will be able to recognise ethical problems in their private and professional sphere; they will avoid common but flawed ethical reasoning and will realize that ethical problems are typically far from having a 'unique' or 'optimal' solution.

- Students will be able to actively engage in ethical reasoning through realistic case studies; in doing so, they will be able to use basic ethical theories as tools for exploring dilemmas.

Advanced Databases

Lecturers: Khalil Amiri, Peter McBrien

Aims To provide students with a detailed theoretical and practical knowledge of how database management systems (DBMS) are implemented, how efficient applications are designed and implemented to work on DBMS, and DBMS may be linked to form 'distributed' DBMS (DDB).

Learning outcomes by the end of the course, student will understand:
- how SQL programs are implemented as series of primitive operations

- how series of those primitive operations are executed as atomic units of work called transactions, and how those transactions may be executed concurrently

- how to write SQL programs (and design DBMS schemas) to make the programs run efficiently

- how DDB are implemented, and how applications can be designed for those DDB.

- how to integrate existing databases to form a DDB.

Syllabus:
Database Management System Architecture main components of a DBMS buffers, caches, and optimisation high level query languages and low level primitive operations

Concurrency Control and Recovery ACID properties of transactions recoverability serialisability Transaction histories as a method for analysing database execution Two-phase locking (2PL) ANSI SQL concurrency control levels

Query optimisation

Distributed Databases (DDB) A general distributed database architecture Fragmentation: horizontal, vertical, hybrid Replication Top-up design of DDB: the allocation problem Bottom-down design of DDB: the schema integration problem Tasks in schema integration and strategies to follow Reverse engineering relational database schemas Schema integration transformations Concurrency control in DDB Replication of locks Distributed deadlock detection Atomic commitment of transactions.

Building Advanced (Distributed) Database Applications Distributed query processing and optimisation Programmer's interface Data migration: data warehousing OLAP XML and Relational Databases.

Alternatives to the Relational Database Model, Object-oriented databases and Object-relational databases.

Distributed Systems

Lecturer: Morris Sloman

Aims:
The objective is to give students a clear overview of the problems and issues that must be dealt with in constructing secure and flexible distributed applications. The emphasis is on the conceptual basis for distributed and networked systems rather than a detailed study of particular systems and standards. Concepts will be illustrated with examples from practical systems.

Learning Outcomes:
A sound understanding of the principles and concepts involved in designing distributed systems and Internet applications.

Ability to implement a distributed application using Java RMI

Ability to understand and evaluate security solutions.

Syllabus:
Overview of Distributed System Architecture: motivation, system structures, architecture, ODP Reference model and distribution transparencies, design issues.

Interaction Primitives: message passing, remote procedure call, remote object invocation

Software Structures and components: composite components, Darwin architecture description language, first & third party binding.

Interaction Implementation: message passing, RPC, concurrency and threads, heterogeneity of systems and languages.

Security: threat analysis, security policies - military (Bell Lapadula) vs commercial models; access control concepts - identification, authentication, authorisation and delegation; authorisation policy: access matrix, access rules and domains, firewalls; access control lists, capabilities, secret and public key encryption, digital signatures, authentication, Kerberos; web security; security management.

Distributed Systems Management: SNMP and OSI Management Models, monitoring and event generation, domains & policy.

Simulation and Modelling

Lecturer: Tony Field

Objectives:
This course provides an introduction to system modelling using both computer simulation and mathematical techniques. A wide range of case studies are examined, both in the lectures and tutorial exercises, although the emphasis is on the analysis of computer and communication systems using a combination discrete-event simulation and queuing theory. The course is self-contained, both in terms of notes and supporting software.

Contents:
Introduction and basic simulation procedures.

Model classification (with worded examples for each): Monte Carlo simulation, discrete-event simulation, continuous system simulation, mixed continuous/discrete-event simulation.

Queuing networks: analytical and simulation modelling of queuing systems.

Input and output analysis: random numbers, generating and analysing random numbers, sample generation, trace- and execution-driven simulation, point and interval estimation. Process-oriented and parallel simulation.

Learning Outcomes:
Students will, by the end of the course, be able to analyse computer and communication systems through case studies, lectures and tutorial exercises. They will also be able to demonstrate an understanding of system modelling through the competent use of Computer Simulation methods and Mathematical Modelling techniques.

Type Systems for Programming Languages

Lecturer: Steffen van Bakel

Contents:
The course sets of with the presentation of the Lambda Calculus, on which most functional languages are based. We will introduce the Curry Type Assignment System for this calculus, for which we show that types are respected by reduction, and that there exists a notion of principality on types, i.e. all types for a program can be constructed from the 'principal' type.

We will then focus on how to extend the language and the system of types in order to deal with Polymorphism. To this end, Combinator Systems are introduced as a slight extension of LC. In order to study how to deal with Recursion, the language of CS is extended to allow for recursive definitions, and we will discuss ways of typing this extension.
We will then look at Milner's basic ML system, and show that, in this calculus, a solution is present for all the issues discussed separately before.
A disadvantage of ML is that it is, syntactically, very distant from actual programming languages, and it is not easy to see that all properties shown for ML translate to the 'real' languages. In particular, definitions of functions by 'cases' (Pattern Matching) are not present in ML. In order to come to a formal notion of types that relates more closely to actual programming languages, we introduce Term Rewriting Systems, which are much closer to actual programs, and show that we can obtain much of the desired properties.
To conclude, we will have a brief visit to Intersection Type Assignment, which allows to prove strong properties. The aim of this course is to lay out in detail the design of type assignment systems for programming languages and focus on the importance of a sound theoretical framework, in order to be able to reason about the properties of a typed program. Students will study and compare a variety of systems and languages.

Lambda Calculus: terms, abstraction, application, reduction, normal form, normalisation, head normal form, head normalisation. The Curry type assignment System.

Types, type assignment rules, type substitution, unification, subject reduction. The Principle type for Curry's System.

Recursion and Polymorphism: the need for a recursor, how to type recursion,
Milner's let, the basic ML system, the principal type property for the ML system.

Church's typed lambda calculus: type checking versus type inference.

The Polymorphic Lambda calculus. Strong normalisation for the polymorphic Lambda Calculus.

Intersection type assignment: types, type assignment rules, subject expansion, undecidability, filter semantics.

Patterns: term rewriting systems, weak reduction, normal forms, strong normalisation. Dealing with patterns, recursion, subject expansion and reduction.

A decidable restriction of the intersection system.

Organisations and Management Processes

Lecturer: John Woodley

To introduce the students to basic theories of organisational behaviour and management.

To introduce techniques for evaluating the profitability of individual projects and for the construction and analysis of the main financial statements.

This course will cover the concepts of management organisation and behaviour; capital budgeting; analysis and interpretation of accounts.

Introduction to management: the evolution of managerial theories; managerial riles and functions; the nature and type of managerial work.

Development of individuals and groups: issues of motivation, decision making and integration; dynamics of leadership; the learning organisation; human resources.

Network Security

Lecturers: Naranker Dulay, Emil Lupu

Aims:
To survey the principles and practice of network security. The emphasis of the course is on the underlying principles and techniques of network security with examples of how they are applied in practice.

Learning Outcomes:
At the end of the course, a student will have an understanding of: the themes and challenges of network security, the role of cryptography, the techniques for access control and intrusion detection, the current state of the art. The student will have developed a critical approach to the analysis of network security, and will be

able to bring this approach to bear on future decisions regarding network security. Practical skills will include the implementation of a security protocol. Pre-requisites: Distributed Systems

Outline Syllabus:
Introduction: assets, threats, countermeasures; network security models, security functions: confidentiality, authentication, integrity, nonrepudiation, access control, availability, passive and active attacks, end-to-end vs link-to-link encryption.

Classical Cryptography: key ideas, steganography, codes, one-time pad, substitution and transposition ciphers, cryptanalysis, cryptographic strength.

Symmetric-Key cryptography: Feistel cipher; DES: basics, rounds, e-box, s-box, p-box, key box; Modes of Operation: ECB, CBC, CFB, OFB; Double DES, Triple DES, IDEA, RC5, AES, problems with symmetric key cryptography.

Public-Key cryptography: requirements, confidentiality, authentication, modular arithmetic, Diffie-Hellman key exchange, RSA, attacks against RSA, hybrid cryptosystems, Elliptical Curve, Quantum Cryptography.

Digital Signatures: characteristics, MAC's, one-way hash functions, signing and verification, birthday attack, public-key certificates, disavowed signatures, arbitrated digital signatures, chaffing & winnowing.

Mutual Authentication: basics, replay attacks, man-in-the-middle, interlock protocol, Andrew Secure RPC, Needham Schroeder, Wide-Mouth Frog, Neuman-Stubblebine, Woo-Lam. BAN-Logic.

Key Management: distribution, KDC, announcements and directories, public key certificates, X509 certification authorities, PGP web of trust, control vectors, key generation and destruction, key backup.

Intruders and Programmed Threats: host access, password systems and attacks, one-time passwords, token cards, biometrics, trapdoors, programmed threats: trapdoors, logic bombs, trojan horses, viruses, worms, countermeasures, intrusion-direction.

Firewalls: internet security policies, firewall design goals, firewall controls, TCP/IP, packet filtering routers, application-level gateways, circuit-level gateways, firewall architectures, VPNs.

Web Security: WWW, web servers, CGI, active content, Java applets, Java security model: sandbox, class loaders, bytecode verification, security manager, Java attacks, bypassing Java, mobile code cryptography.

Software Engineering - Environments

Lecturers: Susan Eisenbach, Michael Huth

Aim:
To introduce state-of-the-art advanced tools and techniques for large-scale software systems development. To develop the critical skills to judge which technique would be most appropriate for solving large-scale software problems.

Outline Syllabus:
Software Process Development Techniques: We present current methods for developing systems such as Aspect Oriented Programming, Extreme Programming, use of Open Source Software, and Agile Programming methodologies. The emphasis is on developing the ability to choose the most suitable methodology for each task and to be able to evaluate new methodologies as they become popular.

Maintenance: Software maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment. We look at a collection of software maintenance problems associated with old and new code (Fortran and Java) and at reverse engineering techniques for code improvement.

Quality Assurance: What solutions are currently available that aid implementors in achieving quality assurance of their code? We mostly study an approach of annotating source code with integrity constraints (object invariance, pre- and post-conditions etc) and survey some of the freeware tool support that is available at present. If time permits, we will give a brief introduction to the CASE tool Rational Rose.

Learning outcomes:
1) Students will have an understanding of some current software process methodologies and be able to apply critical facilities to valuing methodologies as they become fashionable.

2) Students will understand the scope of the software maintenance problem and will be familiar with several tools for reverse engineering software.

3) Students will know what technological support is currently available for support of software quality assurance in implementation work. Notably they will be able to download and use tools for annotating Java and C# programs with integrity constraints such as object invariance and pre- and post-conditions for methods.

Advanced Issues in Object Oriented Programming

Lecturer: Sophia Drossopoulou

Objectives:
To discuss issues around the design and implementation of object oriented languages, the rationale and explore alternatives. To use formal calculi as an unambiguous notation, and as a way to establish soundness.

Contents:
Motivation, type system of some language(s) unsafe. Static vs dynamic types. Sound type systems.

L1: a minimal, class based, imperative, object oriented language with methods and fields. Operational Semantics, Type system, Agreement, Soundness.

L2: L1+inheritance. Operational Semantics, Type system, Agreement, Soundness.
The expressive power of L2: Numbers and Boolean.

C++ features and implementation. Pointer vs Value Types. Single Inheritance, Multiple Inheritance. Virtual Inheritance. Object layout, virtual tables, implementation of assignment and of method call.

The Java virtual machine. The bytecode verifier. Formalization

Java dynamic linking.

Ownership types.

The Abadi & Cardelli Object Calculus.

After the course, students should:
- be able to develop a formal description of a small extension of the languages described in the course

- understand the interplay between static and dynamic checks, and the various checks applied at the different phases of execution

- understand how efficiency of implementation issues, and understand how some oo features can be implemented efficiently eg object layout, virtual tables, (multiple/virtual) inheritance

- understand the difference-similarities of the object based and class based paradigm

Computer Vision

Lecturer: Guang-Zhong Yang

To introduce the concepts behind computer-based recognition and extraction of features from raster images. To illustrate some successful applications of vision systems and their limitations

Overview of early, intermediate and high level vision: first and second moments of illumination for recognition and classification of machine shop components in silhouette.

Segmentation: region splitting and merging; quadtree structures for segmentation; mean and variance pyramids; computing the first and second derivatives of images using the isotropic, Sobel and Laplacian operators; grouping edge points into straight lines by means of the Hough transform; limitations of the Hough transform; parameterisation of conic sections.

Perceptual grouping: failure of the Hough transform; perceptual criteria; improved Hough transform with perceptual features; grouping line segments into curves.

Overview of mammalian vision: experimental results of Hubel and Weisel; analogy to edge point detection and Hough transform; neural networks based on the mammalian vision system.

Relaxation labelling of images: detection of image features; simulated annealing.

Grouping of contours and straight lines into higher order features such as vertices and facets.

Depth measurement in images; triangulation; projected grid methods; shape from shading based on multi-source illumination.

Matching of images: the correspondence problem for stereo vision; two camera and multiple camera systems; shape from motion as a further stage of stereo vision; optical flow between adjacent video frames.

Expert system modelling in computer vision: model based vision using inference engines and rules.

Learning Outcomes:
By the end of this course Students will be able to explain the concepts behind computer based recognition and the extraction of features from raster images. Students will also be able to illustrate some successful applications of vision systems and will be able to identify the vision systems limitations.

Models of Concurrent Computation

Lecturer: Philippa Gardner

To provide an overview of various formalisms for the specification and verification of concurrent systems.

Overview: communication via message passing or shared variables, non-determinism, processes as 'black boxes', interleaving versus true concurrency.

CCS (Calculus of Concurrent Systems): operators, case studies, Concurrency Workbench (software tool), transition systems, equational reasoning, bisimulation, observational congruence, Hennessy-Milner logic, timed CCS.

Pi calculus and related calculi, e.g. Spi calculus for cryptographic protocols.

CSP (Communicating Sequential Processes): operators, case studies, failures model, relationship with CCS.

True concurrency: Event structures, Petri nets, case studies, relationship with CCS/CSP.

Management - Economics and Law

Lecturer: James Jacobson

To investigate the principles of macro-economic theory and their effectiveness in practice and the economic climate in which management decisions have to be taken. To impart a basic practical knowledge of law.

National income: definition, evaluation, equilibrium positions. Investment. The business cycle. Objectives of government policy. Monetary and fiscal policies. Foreign trade and its impact. The English legal system. The law of tort: duties, obligations, liabilities, negligence. The intellectual property law: the law of patents, trade and service marks, protecting and exploring rights. Business start up - issues to consider on starting a small hi-tech business.

Distributed Algorithms

Lecturers: Jeff Magee, Jeff Kramer

Contents:
- Models of distributed computing
- Synchrony, communication and failure concerns
- Synchronous message-passing distributed systems
- Algorithms in systems with no failures - Leader Election and Breadth-First Search algorithms
- The atomic commit problem
- Consensus problems - the Byzantine Generals Problem
- Asynchronous message-passing distributed systems
- Logical time and global system snapshots
- Impossibility of consensus
- Fault-tolerant broadcasts
- Partially synchronous message-passing distributed systems
- Failure detectors

The Labelled Transition System Analyser (LTSA) tool
http://www.doc.ic.ac.uk/~jnm/book/ltsa-v2/index.html is used throughout the course for modelling and demonstrating the execution of various algorithms.

Learning Outcomes:
By the end of this course students will be able to understand and explain the concepts behind distributed algorithms, including the assumptions made and the potential benefits

and shortcomings. Students will also be able to assess the applicability of distributed algorithms to a particular circumstance.