Software Engineering & Design

CST IIG / CS Diploma
Alan Blackwell

OUTLINE OF COURSE

- Introduction – the “Software Crisis”
- Software Construction
- Object-oriented Design
- Interaction Design
- Design Challenges
- Project Management

Books

- ***Code Complete***: A practical handbook of software construction
  - Steve McConnell, Microsoft Press 1993
- ***UML Distilled*** (2nd edition)
  - Martin Fowler, Addison-Wesley 2000
- ***Interaction Design***: Beyond human-computer interaction
  - Jenny Preece, Helen Sharp & Yvonne Rogers, Wiley 2002
- ***Software Engineering*** (European edition)
  - Roger Pressman, McGraw-Hill 2001
- Further:
  - *Programming as if People Mattered*, Nate Borenstein
  - *The Mythical Man Month*, Fred Brooks
  - *Computer-Related Risks*, Peter Neumann
  - *The Sciences of the Artificial*, Herb Simon
  - *Educating the Reflective Practitioner*, Donald Schon
  - *London Ambulance Service* & *CAPSA* reports, Finkelstein

Supervisions

- The course is practical, not theoretical
- Designed specifically to feed in to your projects (and your future life …)
- No point in having supervisions to discuss the material until you have tried it in practice, so:
  - Part IIG students (group project)
    - Supervisions should help you apply project management, object-oriented design methods etc. in your group
  - Diploma students (project and dissertation)
    - Supervisions should address interface design issues, coding, testing and evaluation techniques.

Introduction

The Software Crisis

London Ambulance Service

- The project: automate inefficient manual operation
  - 999 calls written on forms
  - map reference looked up
  - conveyor belt to central point
  - controller removes duplicates, passes to NE/NW/S district
  - division controller identifies vehicle and puts note in its ‘activation box’
  - form passed to radio dispatcher
- Takes about 3 minutes, and 200 staff (of 2,700 total).
  - some errors (esp. deduplication),
  - some queues (esp. radio),
  - call-backs are laborious to deal with

LAS: Project Background

- Attempt to automate in 1980's failed
  - the system failed a load test
- Industrial relations poor
  - pressure to cut costs
- Decided to go for fully automated system:
  - controller answering 999 call has on-screen map
  - send "email" directly to ambulance
- Consultancy study to assess feasibility:
  - estimated cost £1.5m, duration 19 months …
  - ***provided*** a packaged solution could be found
  - ***excluding*** automatic vehicle location system

LAS: Award of Tender

- Idea of a £1.5m system stuck, ***but***
  - automatic vehicle location system added
  - proviso of packaged solution forgotten
  - new IS director hired
  - tender put out 7 February 1991
  - completion deadline January 1992
- 35 firms looked at tender
  - 19 submitted proposals, most said:
    - timescale unrealistic
    - only partial automation possible by January 1992
- Tender awarded to consortium:
  - Systems Options Ltd, Apricot and Datatrak
  - bid of £937,463 … £700K cheaper than next bidder

LAS: Design Phase

- Design work 'done' July
- main contract August
- mobile data subcontract September
- in December told only partial implementation possible in January –
  - front end for call taking
  - gazetteer + docket printing
- by June 91, a progress meeting had minuted:
  - 6 month timescale for 18 month project
  - methodology unclear, no formal meeting program
  - LAS had no full time user on project
- Systems Options Ltd relied on 'cozy assurances' from subcontractors

LAS: Implementation

- Problems apparent with 'phase 1' system
  - client & server lockup
- 'Phase 2' introduced radio messaging, further problems
  - blackspots, channel overload at shift change,
  - inability to cope with 'established working practices' such as taking the 'wrong' ambulance
- System never stable in 1992
- Management pressure for full system to go live
  - including automatic allocation
  - CE said: 'no evidence to suggest that the full system software, when commissioned, will not prove reliable'

LAS: Live Operation

- Independent review had noted need for:
  - volume testing
  - written implementation strategy
  - change control
  - training
  - … it was ignored.
- 26 October
  - control room reconfigured to use terminals not paper
  - resource allocators separated from radio operators and exception rectifiers
  - No backup system.
  - No network managers.

LAS: 26 & 27 October - Disaster

- Vicious cycle of failures
  - system progressively lost track of vehicles
  - exception messages built up, scrolled off screen, were lost
  - incidents held as allocators searched for vehicles
  - callbacks from patients increased workload
  - data delays - voice congestion - crew frustration - pressing wrong buttons and taking wrong vehicles
  - many vehicles sent, or none
  - slowdown and congestion proceeded to collapse
- Switch back to semi-manual operation on 27 Oct
- Irretrievable crash 02:00 4 Nov due to memory leak:
  - ‘unlikely that it would have been detected through conventional programmer or user testing’
- Real reason for failure: poor management throughout

The Software Crisis

- Emerged during 1960's
  - large and powerful mainframes (e.g. IBM 360) made far larger and more complex systems possible
  - why did software projects suffer failures & cost overruns so much more than large civil, structural, aerospace engineering projects?
- Term 'software engineering' coined 1968
  - hope that engineering habits could get things under control
  - e.g. project planning, documentation, testing
- These techniques certainly help – we'll discuss
- But first:
  - how does software differ from machinery?
  - what unique problems and opportunities does it bring?

Why is software different (and fun)?

- The joy of making things useful to others
- The fascination of building puzzles from interlocking "moving" parts
- The pleasure of a non-repeating task
  - continuous learning
- The delight of a tractable medium
  - "pure thought stuff"

What makes software hard?

- The need to achieve perfection
- Need to satisfy user objectives, conform with systems, standards, interfaces outside control
- Larger systems qualitatively more complex (unlike ships or bridges) because parts interact in many more than 3 dimensions.
- Tractability of software leads users to demand 'flexibility' and frequent changes
- Structure of software can be hard to visualise/model
- Much hard slog of debugging and testing accumulates at project end, when:
  - excitement is gone
  - budget is overspent
  - deadline (or competition) looming

The 'Software Crisis'

- The reality of software development has lagged behind the apparent promise of the hardware
- Most large projects fail - either abandoned, or do not deliver anticipated benefits
  - LSE Taurus £ 400 m
  - Denver Airport $ 200 m
  - CONFIRM $ 160 m
- Some software failures cost lives or cause large material losses
  - Therac 25
  - Ariane
  - Pentium
  - NY Bank - and Y2K in general
- Some combine project failure with loss of life, e.g. London Ambulance Service

Special emphases of this course

- Requirements:
  - User centred interaction design, not older requirements capture methods (Pressman describes both)
- Analysis and design:
  - Object-oriented design and UML, not older structured analysis (Pressman describes both)
- Construction:
  - Emphasise coding, not metrics
- Project management & quality assurance:
  - Pressman best on these (and also best overview, though weak on UML and interaction design)

Part I
Software Construction

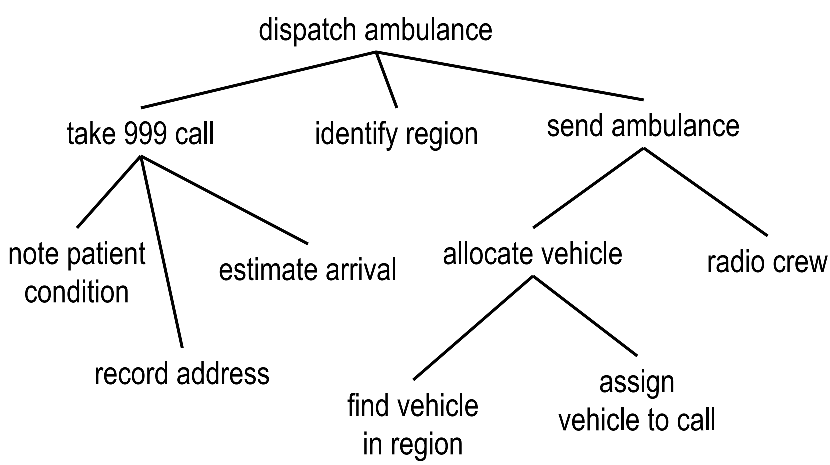
2 lectures

Software Construction

- Decomposition and Modularity
- Coding style
- Naming
- Configuration
- Testing
- Efficiency

Decomposition and Modularity

- top-down decomposition: stepwise refinement



Top-down versus Bottom-up

- This course is structured in a bottom-up way.
- Why?
  - Start with what you understand
  - Build complex structures from well-understood parts
  - Deal with concrete cases in order to understand abstractions
- The same advantages can apply to software as to teaching.
  - Real software construction combines top-down and bottom up.

Bottom-up design issues

- Some important programming skills apply to every language:
  - Naming variables and functions
  - Organising control structures
  - Laying out lines of code
  - Using comments
  - Achieving type-safety
  - Designing modules

Modularity - routines

- Is this routine required?
- Define what it will do
  - What information will it hide?
  - Inputs
  - Outputs (including side effects)
  - How will it handle errors
- Give it a good name
- How will you test it?
- Think about efficiency and algorithms
- Write as comments, then fill in actual code

Modularity beyond the routine

- Separate source files in C
  - Inputs, outputs, types and interface functions defined by declarations in “header files”.
  - Private variables declared in the source file
- Classes in Java
  - Inputs and outputs can be controlled by visibility specifiers and access functions
  - Aim for all data to be private, and as few public functions as possible
- Classes in C++
  - Somewhat like C, somewhat like Java
- Modules in ML

Using comments

- Comments help the person reading your code understand what you intended it to do.
  - The purpose of a class or routine
    - And also its limitations
  - Warning the reader of surprises
  - Defining data units and allowable ranges
- The person reading the comments may be you … in a year (or a few weeks) time.
- In larger group projects
  - Authorship (and copyright?)
  - Change history, especially in shared code

Coding style: layout

- Objectives:
  - Accurately express logical structure of the code
  - Consistently express the logical structure
  - Improve readability
- Good visual layout shows program structure
  - Mostly based on white space and alignment
  - The compiler ignores white space
  - Alignment is the single most obvious feature to human readers.
- Code layout is most like the art of typography

Expressing global structure

```
Function_name (parameter1, parameter2)

// Function which doesn't do anything, beyond showing the fact
// that different parts of the function can be distinguished.

    type1: local_data_A, local_data_B
    type2: local_data_C

    // Initialisation section
    local_data_A := parameter1 + parameter2;
    local_data_B := parameter1 - parameter2;
    local_data_C := 1;

    // Processing
    while (local_data_C < 40) {
        if ( (local_data_B ^ 2) > local_data_A ) then {
            local_data_B := local_data_B - 1;
        } else {
            local_data_B := local_data_B + 1;
        } // end if
        local_data_C := local_data_C + 1;
    } // end while

} // end function
```

Expressing local control structure

```
while (local_data_C < 40) {
    form_initial_estimate(local_data_C);
    record_marker(local_data_B - 1);
    refine_estimate(local_data_A);
    local_data_C := local_data_C + 1;
} // end while
```

```
if ( (local_data_B ^ 2) > local_data_A ) then {
    // drop estimate
    local_data_B := local_data_B - 1;
} else {
    // raise estimate
    local_data_B := local_data_B + 1;
} // end if
```

Expressing structure within a line

- Whitespacealwayshelpshumanreaders
  - `newtotal=oldtotal+increment/missamount-1;`
  - `newtotal = oldtotal + increment / missamount - 1;`
- The compiler doesn't care – take care!
  - `x = 1  *  y+2  *  z;`
- Be conservative when nesting parentheses
  - `while ( (! error) && readInput() )`
- Continuation lines – exploit alignment

```
if ( ( aLongVariableName & anotherLongOne ) |
     ( someOtherCondition() ) )
   {
    …
   }
```

Naming variables: Form

- Priority: full and accurate (*not* just short)
  - Abbreviate for pronunciation (remove vowels)
    - e.g. CmptrScnce (leave first and last letters)
- Parts of names reflect conventional functions
  - Role in program (e.g. "count")
  - Type of operations (e.g. "window" or "pointer")
  - Hungarian naming (not really recommended):
    - e.g. pscrMenu, ichMin
- Even individual variable names can exploit typographic structure for clarity
  - **`xPageStartPosition`**
  - **`x_page_start_position`**

Naming variables: Content

- Data names describe domain, not computer
  - Describe what, not just how
  - **`CustomerName`** better than **`PrimaryIndex`**
- Booleans should have obvious truth values
  - **`ErrorFound`** better than **`Status`**
- Indicate which variables are related
  - **`CustName`**, **`CustAddress`**, **`CustPhone`**
- Identify globals, types & constants (in C)
  - e.g. g_wholeApplet, T_mousePos
- Even temporary variables have meaning
  - **`Index`**, not **`Foo`**

Naming variables: Role

- Decide which is which, don't mix them up!
  - Fixed value (maybe not declared constant)
    - Always the same (e.g. screensize)
  - Stepper
    - Predictable succession of values (e.g. month)
  - Follower
    - Gets its value from another variable (e.g. previous)
  - Most-recent holder
    - Latest in a series (e.g. finish)
  - Most-wanted holder
    - Best value so far (e.g. largest)
  - Gatherer
    - Accumulating effect of individual values (e.g. total)
  - One-way flag
    - Never goes back to its initial value (e.g. error)
  - Temporary

Achieving type-safety

- Refine types to reflect meaning, not just to satisfy the compiler.
- Valid (to compiler), but incorrect, code:
  - `float totalHeight, myHeight, yourHeight;`
  - `float totalWeight, myWeight, yourWeight;`
  - `totalHeight = myHeight + yourHeight + myWeight;`
- Type-safe version:
  - `type t_height, t_weight: float;`
  - `t_height totalHeight, myHeight, yourHeight;`
  - `t_weight totalWeight, myWeight, yourWeight;`
  - `totalHeight = myHeight + yourHeight + myWeight;`

Compile error!

Defensive programming

- Assertions and correctness proofs would be useful tools, but are seldom available.
- Defensive programming includes additional code to help ensure local correctness
  - Treat function interfaces as a contract
- Each function / routine
  - Checks that input parameters meet assumptions
  - Checks output values are valid
- System-wide considerations
  - How to report / record detected bugs
  - *Perhaps* include off-switch for efficiency

Efficiency

- The worst mistakes come from using the wrong algorithm
  - 48 hours -> 2 minutes
- Hardware now fast enough to run most code fast enough (assuming sensible algorithms)
  - Optimisation is a waste of your time
- Optimisation is required
  - For extreme applications
  - When pushing hardware envelope
- Cost-effective techniques
  - Check out compiler optimisation flags
  - Profile and hand-optimise bottlenecks

Formal methods

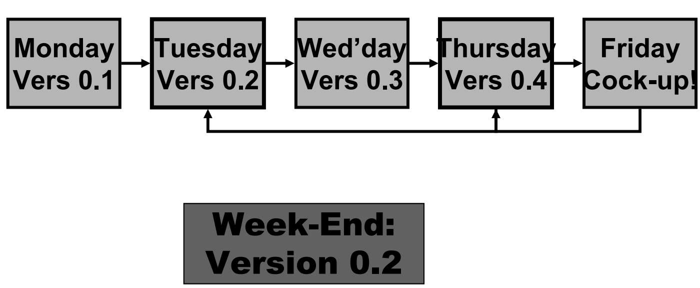
- Pioneers (e.g. Turing) talked of proving programs using mathematics
  - program verification started with Floyd (67)
  - followed up by Hoare (71) and others
- Now a wide range of techniques and tools for both software and hardware, ranging from the general to highly specialised.
  - Z, based on set theory, for specifications
  - LOTOS for checking communication protocols
  - HOL for hardware
- Not infallible – but many bugs are found
  - force us to be explicit and check designs in great detail
  - but proofs have mistakes too
- Considerable debate on value for money

Configuration Management

- Version control
- Change control
- Variants
- Releases

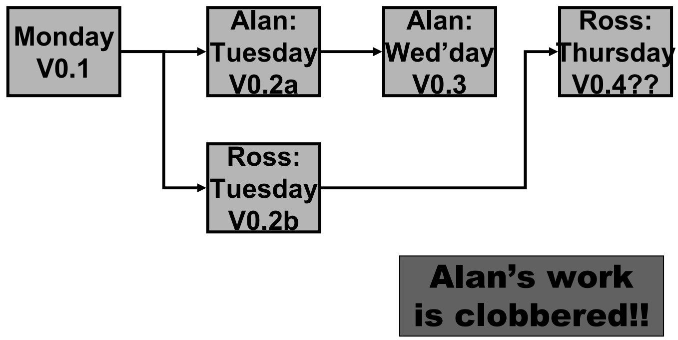
Version control

- Record regular “snapshot” backups
  - often appropriate to do so daily
- Provides ability to “roll back” from errors
- Useful even for programmers working alone



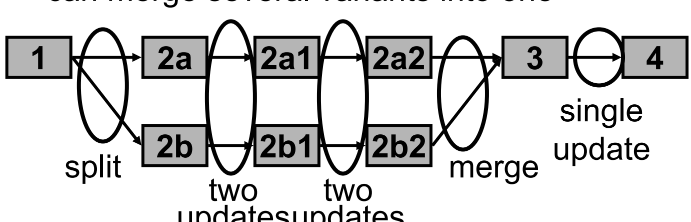
Change control

- Essential in programming teams
- Avoid the “clobbering” problem
  - Older tools (RCS, SCCS) rely on locking
  - More recent (CVS) automate merging



Variants

- Branching results in a tree of different versions or “variants”
- Maintaining multiple branches is costly
  - Merge branches as often as possible
  - Minimise number of components that vary in each branch (ideally only one configuration file)
  - If necessary, conditional compile/link/execution can merge several variants into one



Builds and Releases

- Record actual configuration of components that were in a product *release*, or even an overnight *build* integrating work of a team.
  - Allows problems to be investigated with the same source code that was delivered or tested
- Allow start of development on next release while also supporting current release
  - Universal requirement of commercial software development (at least after release 1.0!)
  - Bug fixes made to 1.0.1 are also expected to be there in 2.0, which requires regular merging
- Note: My version of Internet Explorer is 5.00.2920.000

Testing

- Testing is neglected in academic studies
  - but great industrial interest - maybe half the cost
- It takes place at a number of levels - cost per bug removed rises dramatically at later stages:
  - validation of the initial design
  - module test after coding
  - system test after integration
  - beta test 1 field trial
  - subsequent litigation
  - ...
- Common failing is to test late, because early testing wasn't designed for.
  - This is expensive. We must design for testability

Testing strategies

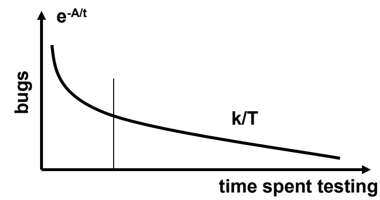
- Test case design: most errors in least time
- White box testing
  - Test each independent path at least once
  - Prepare test cases that force paths
- Control structure testing
  - Test conditions, data flow and loops
- Black box testing
  - Based on functional requirements
  - Boundary value analysis
- Stress testing: at what point will it fail?
  - (vs. performance testing – will it do the job)?

Regression testing

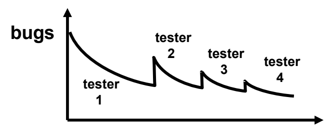
- Checking that new version of software gives same answers as old version
  - Probably single biggest advance in tools for software engineering of packaged software
- Use a large database of test cases, including all bugs ever found. Specific advantages:
  - customers are much more upset by failure of a familiar feature than of a new one
  - otherwise each bug fix will have a ~ 20% probability of reintroducing a problem into set of already tested behaviours
  - reliability of software is relative to a set of inputs. Best test the inputs that users actually generate!
- Test automation tools reduce mundane repetition

When to stop testing

- Reliability growth model helps assess
  - mean time to failure
  - number of bugs remaining
  - economics of further testing, ...
- Software failure rate
  - drops exponentially at first
  - then decreases as K/T



- Changing testers brings new bugs to light



- to get a mttf of 10^9 hours, need 10^9 hours testing

Tools

- We use tools when some parameter of a task exceeds our native ability
  - heavy object: raise with lever
  - tough object: cut with axe
- Software engineering tools deal with *complexity*. There are two kinds of complexity:
  - ***Incidental*** complexity dominated programming in the early days. eg. writing machine code is tedious and error prone. Solution: high level language
  - ***Intrinsic*** complexity of applications is the main problem nowadays. eg. complex system with large team. “Solution”: waterfall/spiral model to structure development, project management tools, etc.
- We can aim to *eliminate* incidental complexity but must *manage* intrinsic complexity

Part II
Object-oriented Design

2 lectures

Object-oriented design

- Design as modelling;
- The Unified Modelling Language;
- Use case analysis;
- Class modelling;
- Object interaction;
- State and activity descriptions.

Why structured design?

- “Organic” hacking doesn’t work when:
  - Many programmers on project.
  - Too large to hold in your head.
  - Need for accurate estimates.
  - Several companies involved.
- So design techniques must provide:
  - language for communication
  - decomposition and simplification
  - predictable relationship to implementation language
  - basis for contractual agreements

Why object-oriented?

- Partly fashion …
  - 1980s: structured design for structured languages
  - 1990s: OO design for OO languages
- … but basic principles still good:
  - Good designers used OO (and structured) techniques before methods became widespread.
  - OO (and structured) techniques are applicable to projects using older languages.
- Current best practice in techniques and tools.

Elements of OO design

- The word “design” can mean a *product* or a *process*.
- The Product: a collection of ***models***
  - like architects’ models, sketches, plans, details
  - models simplify the real world
  - models allow emphasis of specific aspects
- Diagrams (compare fashion sketches, engineering drawings)
  - a cultural tradition in software
    - easy to draw for personal/communicative sketching
    - can be made tidy with tools (templates, CASE tools)

The OO design process

- A process is some set of *phases* defined by project procedures
  - much more on project procedures later in the course
- Iteration between and within phases, e.g.:
  - requirement ⇔ system analysis
  - module design ⇔ architectural design
  - design ⇔ coding ⇔ test
  - … more on this later in course
- Process depends on context and policy
  - OO techniques must be flexible

Standardisation

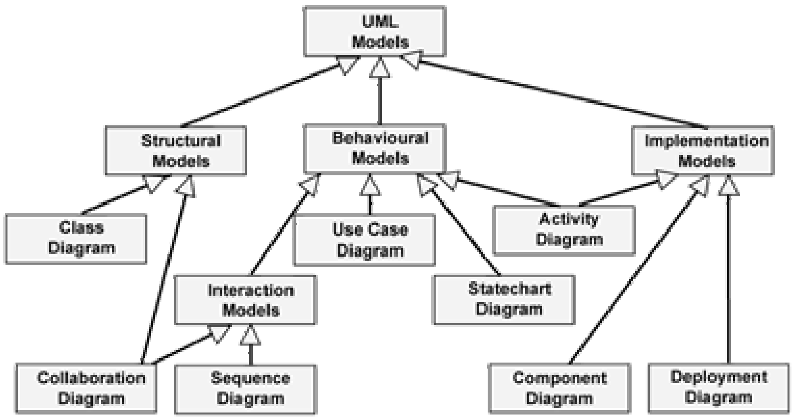
- Early 90s: the methodology wars
  - structured methods didn’t support inheritance etc.
  - many (>50) hacked upgrades and new methods
- The battlefield:
  - books, CASE tools, training
- Consolidation:
  - Natural leaders emerged, mostly by merit
  - Rational Software hired the leaders for UML
  - Object Management Group blessed the result

UNIFIED MODELING LANGUAGE

Tools

- Diagrams: most obvious benefit from CASE tools
  - drawing packages or specialist diagram tools will do
- Repositories: understand diagram content
  - maintain name/type database, diagram consistency
- Code generation: at the least, saves typing
  - dumping class signatures in Java/C++ syntax is easy
  - anything more is hard (and perhaps pointless)
- Alternative languages: UML still useful
  - inheritance, instantiation can be implemented in C etc.
  - OO design can be exploited in later development work

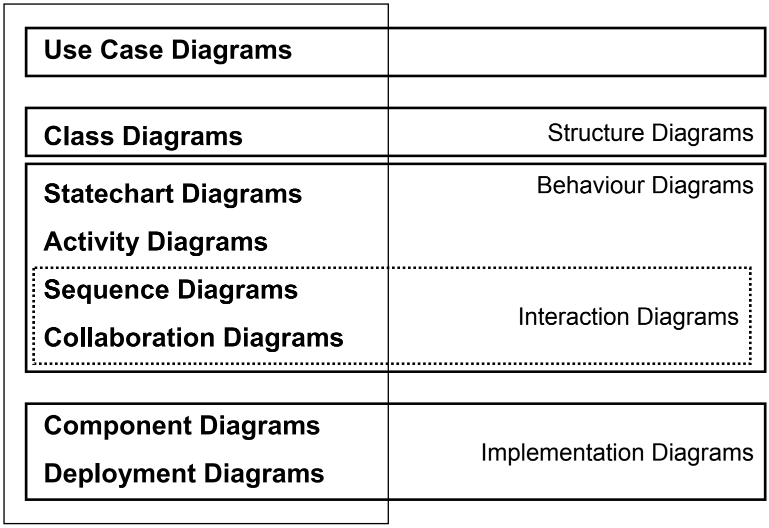
UML: Unified Modeling Language



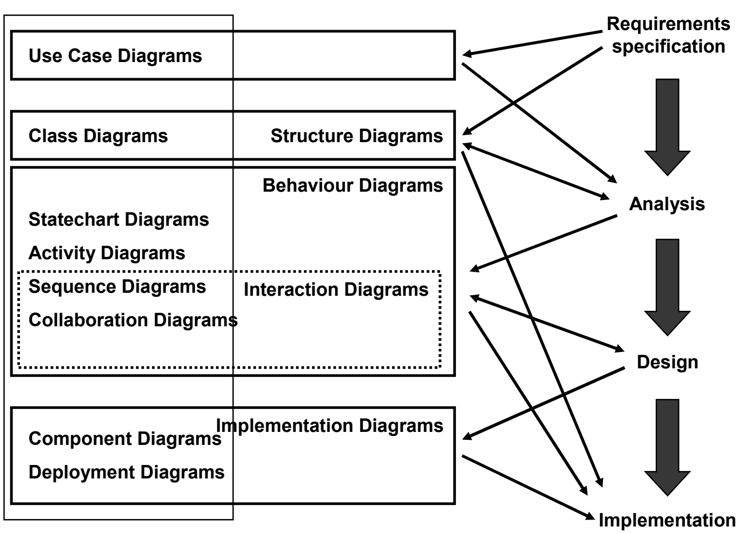
UML diagrams - overview

- **Use Case** diagrams - interactions with / interfaces to the system.
- **Class** diagrams - type structure of the system.
- **Collaboration** diagrams - interaction between instances
- **Sequence** diagrams - temporal structure of interaction
- **Activity** diagrams - ordering of operations
- **Statechart** diagrams - behaviour of individual objects
- **Component** and **Deployment** diagrams - system organisation

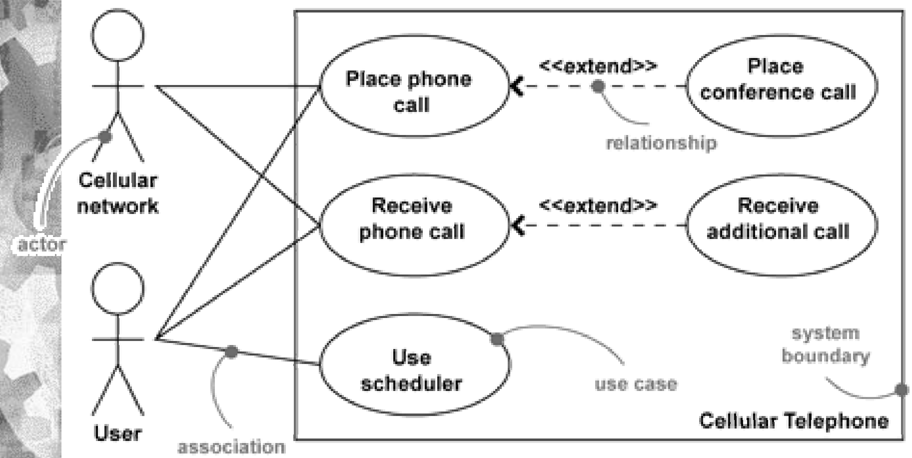
Design role of UML diagrams



UML diagrams in process context

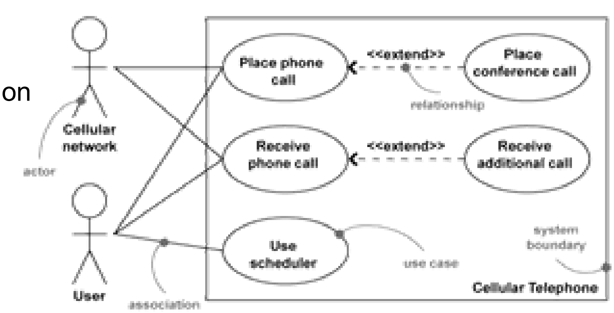


UML Use Case diagram

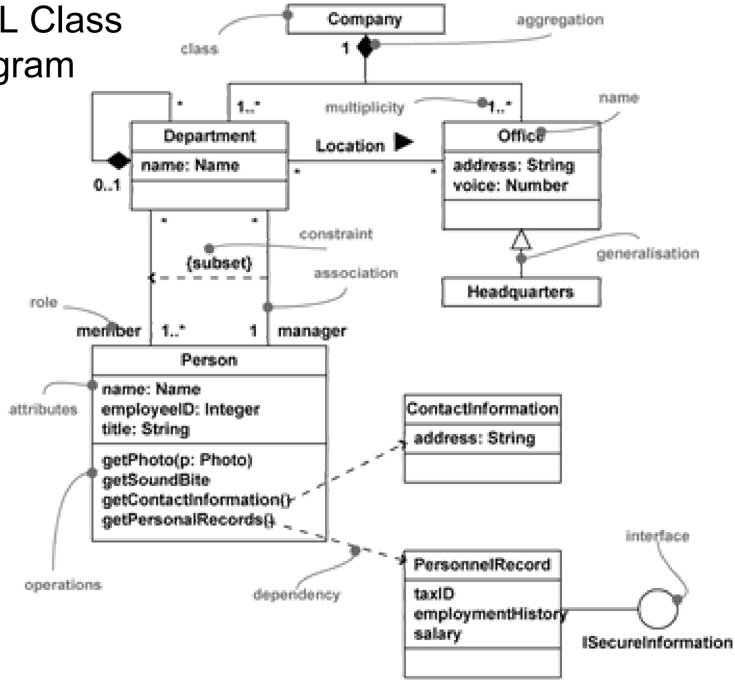


UML Use Case diagram

- Actors
  - play system *role*
  - may not be people
- Use case
  - like a scenario
- Relationships
  - include
  - extend
  - generalisation

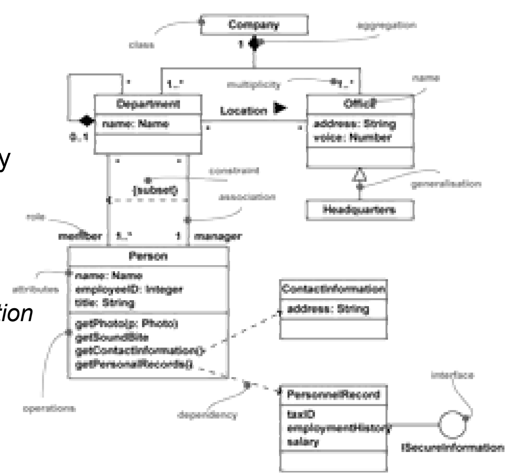


UML Class diagram

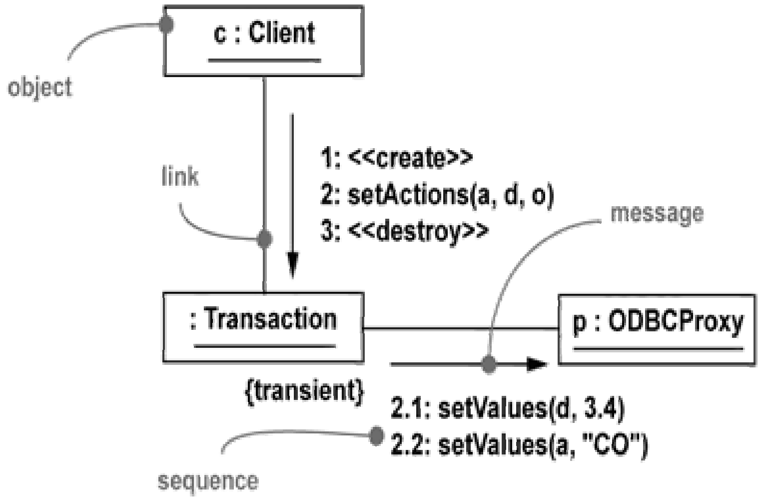


UML Class diagram

- Attributes
  - *type* and *visibility*
- Operations
  - signature and visibility
- Relationships
  - *association*
    - with *multiplicity*
    - potentially *aggregation*
  - *generalisation*

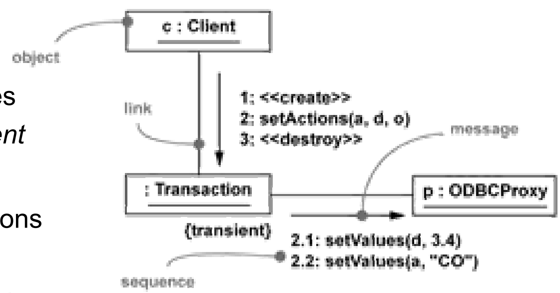


UML Collaboration diagram

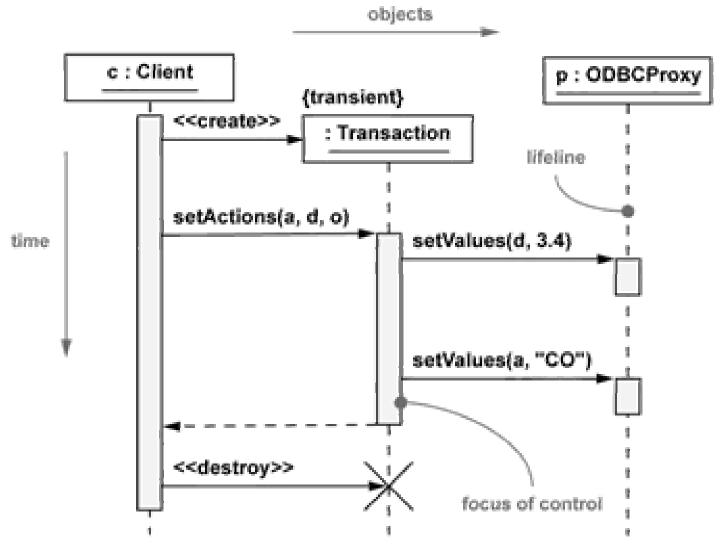


UML Collaboration diagram

- Objects
  - class instances
  - can be *transient*
- Links
  - from associations
- Messages
  - travel along links
  - numbered to show *sequence*

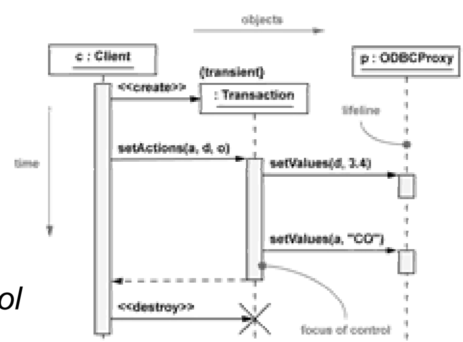


UML Sequence diagram

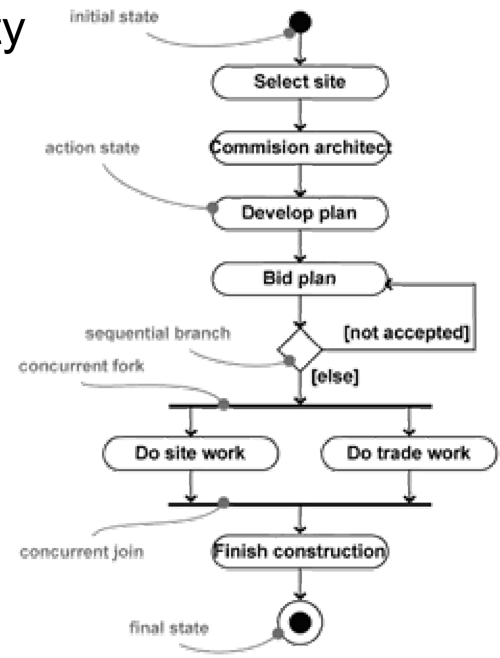


UML Sequence diagram

- Interaction again
  - same content as collaboration
  - emphasises time dimension
- Object *lifeline*
  - objects across page
  - time down page
- Shows *focus of control*

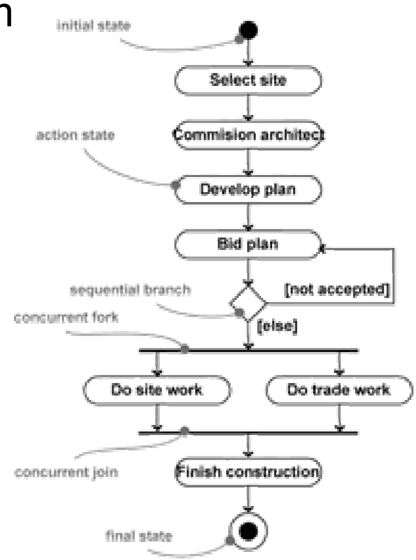


UML Activity diagram

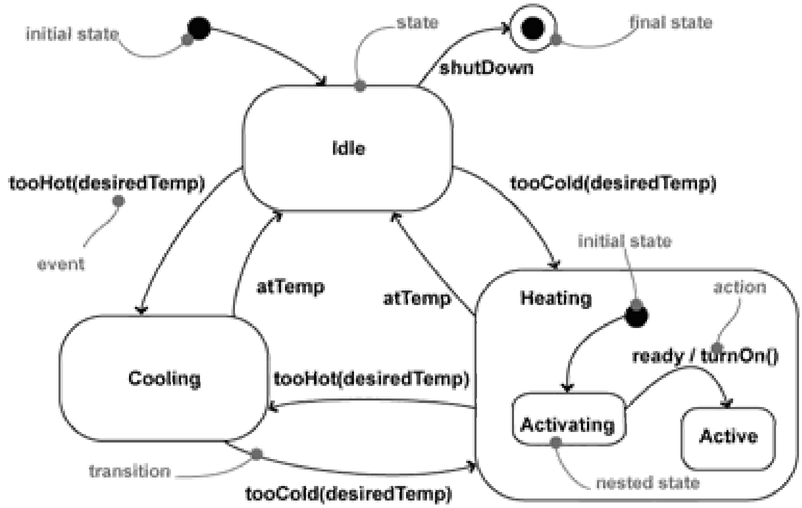


UML Activity diagram

- Like flow charts
  - Activity as action states
- Flow of control
  - transitions
  - branch points
  - concurrency (fork & join)
- Illustrate flow of control
  - high level - e.g. workflow
  - low level - e.g. lines of code

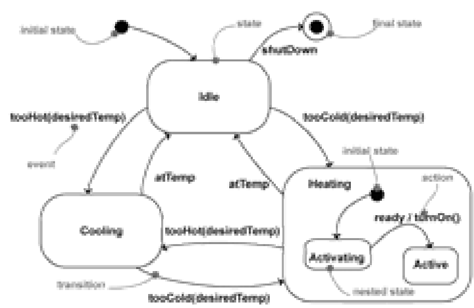


UML Statechart diagram

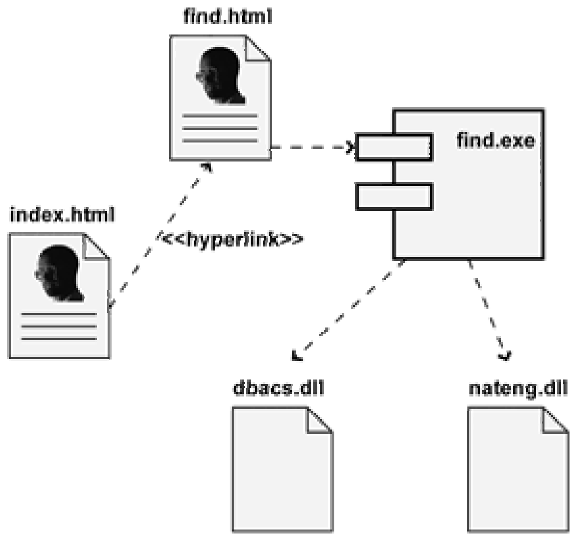


UML Statechart diagram

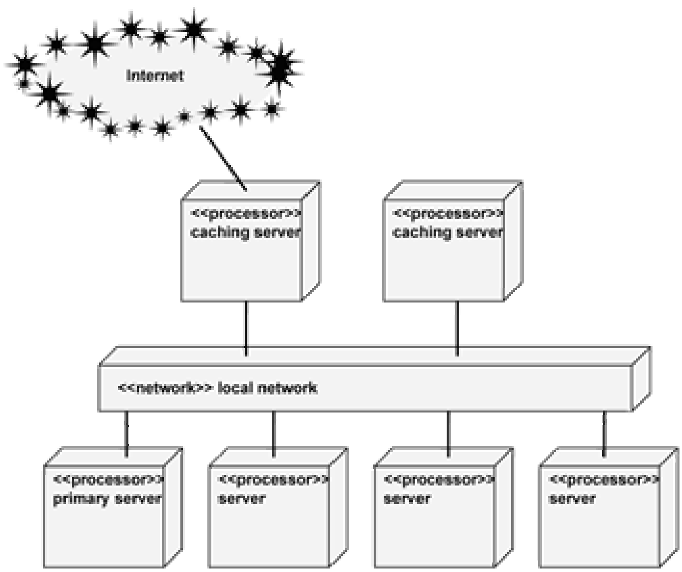
- Object lifecycle
  - data as state machine
- Harel statecharts
  - nested states
  - concurrent substates
- Explicit initial/final
  - valuable in C++
- Note inversion of activity diagram



UML Component diagram



UML Deployment diagram



Quality criterion: Cohesion

- Each component does “one thing” only
  - Functional cohesion – one operation only
  - Sequential – processing data in sequence
  - Communication via shared data
  - Things that must be done at the same time
- Bad cohesion
  - Sequence of operations with no necessary relation
  - Unrelated operations selected by control flags
  - No relation at all – purely coincidental

Quality criterion: Encapsulation

- Separating interface from implementation
- Design precautions:
  - Define visibility - keep implementation private
  - Avoid unnecessary associations
- Consequences:
  - Unexpected (forgotten) interaction and dependencies
- Implementation techniques:
  - Visibility declarations (C++/Java), module export

Quality criterion: Loose coupling

- Keeping parts of design independent
- Design precautions:
  - reduce relationships between diagram nodes
- Consequences:
  - achieve reusability, modifiability
- Implementation techniques:
  - may require several iterations of design for clear conceptual model

Quality criterion: Client-Server Contracts

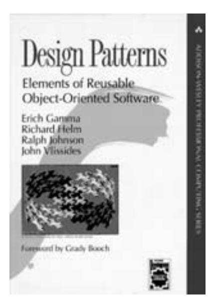
- Consider every object as a “server”
- Design precautions:
  - use Object Constraint Language (OCL) to express constraints on associations, events, messages
- Consequences:
  - reliability, improved partitioning, graceful degradation
- Implementation techniques:
  - support for pre- and post-conditions (e.g. Eiffel)

Quality criterion: Natural data model

- Creating a conceptually clear class structure
- Design precautions:
  - experiment with alternative association, aggregation, generalisation, before committing to code
- Consequences:
  - achieve good mapping to problem domain (hard to retro-fit generalisations).
- Implementation techniques:
  - relies on inheritance

Design Exemplars

- Complete designs come from project experience
- More general solutions: *Design Patterns*
  - The programmer's bag of tricks
  - e.g. Smalltalk Model-View-Controller
  - Collection of patterns: Gamma et. al.
    - (originally architect Christopher Alexander's "pattern language")



UML Extension & Formalisation

- UML provides extension mechanisms
  - Business modeling extension
  - Real-time extension
  - Executable UML
- UML includes a formal specification language
  - Object Constraint Language (OCL)
    - not covered in this course

Minimal UML design

- Some programmers don't like to design first.
  - (Not me)! But what if no management support, no tools?
- Quick and dirty OO design:
  - Write use case "stories", note commonality
  - Keep a piece of paper for each class
    - write attributes, operations, relationships
    - lay out on table, and "talk through" scenarios
  - Consider object lifecycle: state change, persistence
  - When your desk gets too small, buy a proper tool

Further resources: Web

- http://www.omg.org/uml/
- http://www.rational.com/uml/
- http://www.cetus-links.org/oo_uml.html

OO design: Summary

- Large, complex projects need a structured design process.
- Design (the process) involves creating models (the product).
- UML is an established common language for OO design.
- Design projects can be structured around UML models.
- A common design language provides a basis for assessing quality and standardised solutions.

Part III
Interaction Design

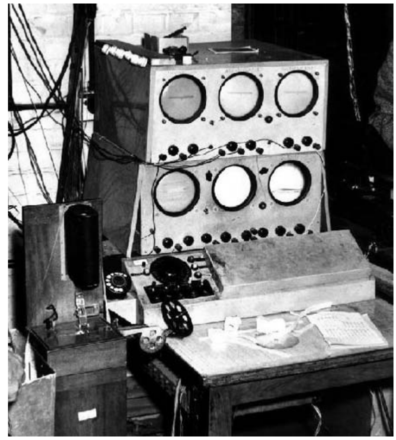
3 lectures

Interaction Design

- Interaction styles
  - The historic effects of interface hardware
  - Evaluation based on interface analysis
- Models of user cognition
  - Understanding the needs of users
  - Evaluation based on cognitive analysis
- Contextual requirements gathering
  - Understanding the situation of use
  - Evaluation of prototypes in context

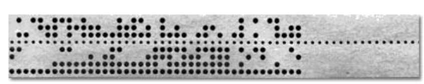
Control panels

- Early computers were like scientific instruments
- For specialists only
- Unit of interaction: the configuration of the machine



Mathematical languages

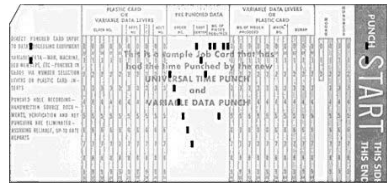
- Write down an equation on paper
- Punch it into a tape in some code
- Feed the tape into the machine
- Unit of interaction: whole programs

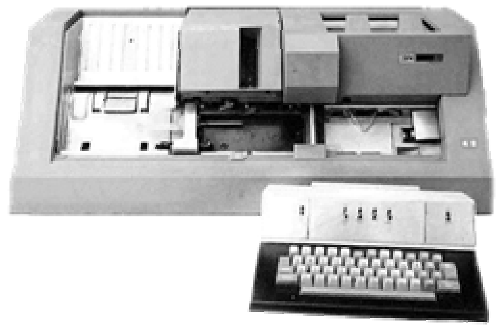


```
   DIMENSION A(11)
   READ A
2  DO 3,8,11 J=1,11
3  I=11-J
   Y=SQRT(ABS(A(I+1)))+5*A(I+1)**3
   IF (400>=Y) 8,4
4  PRINT I,999.
   GOTO 2
8  PRINT I,Y
11 STOP
```

Data files

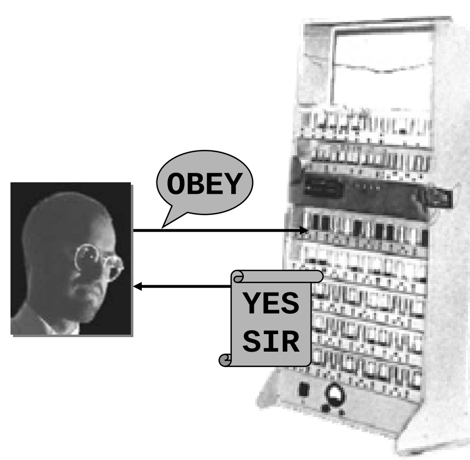
- Punch data records (or program lines) onto cards
- Feed stacks of cards into machine
- Unit of interaction: collection of prerecorded items
  - Can be rearranged





Command lines

- Teletype: like faxing individual requests to the computer
- Unit of interaction: command & response, creating a *dialogue*
  - UNIX started here
  - Disadvantage: users must remember possible commands



WYSIWYG

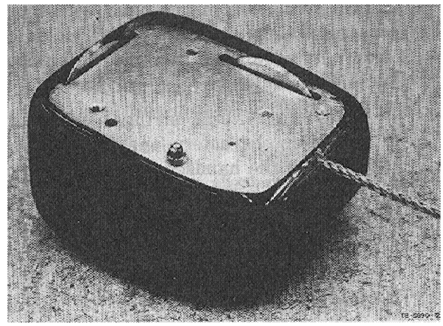
- Originally “Glass teletypes”
  - Look, no paper!
- Units of interaction:
- The *full-screen* editor
  - User can see the product being worked on
  - “What You See Is What You Get”
- All possible commands can be listed in a *menu*

Graphical displays

- Separate menu (text) and product (graphic)
- Unit of interaction: depends on *mode*
- Can commands and product be combined?
  - *Modeless* interaction

Pointing devices

- Allow seamless movement between menus and products on the same screen.
- Unit of interaction: the cursor position



Bitmapped displays

- Units of interaction: icons and windows
  - Windows: multiple contexts shown by frames.
  - Icons: pictures representing abstract entities.

WIMP: **w**indow / **i**con / **m**enu / **p**ointer

- Unit of interaction is not textual
  - (note no keyboard in this ad).
- Object of interest is the unit of interaction

Direct manipulation

- Described by Shneiderman:
  - objects of interest continuously *visible*
  - operations by physical *actions*, not commands
  - actions *rapid*, *incremental*, *reversible*
  - *effect* of actions immediately visible
  - basic commands for novices, more for experts

Heuristic evaluation

- Usability evaluation technique based on general interaction principles
- Comparing system design to set of usability heuristics.
  - *systematic* search for usability problems
  - *team* of evaluators, working independently
  - each evaluator assesses all of interface

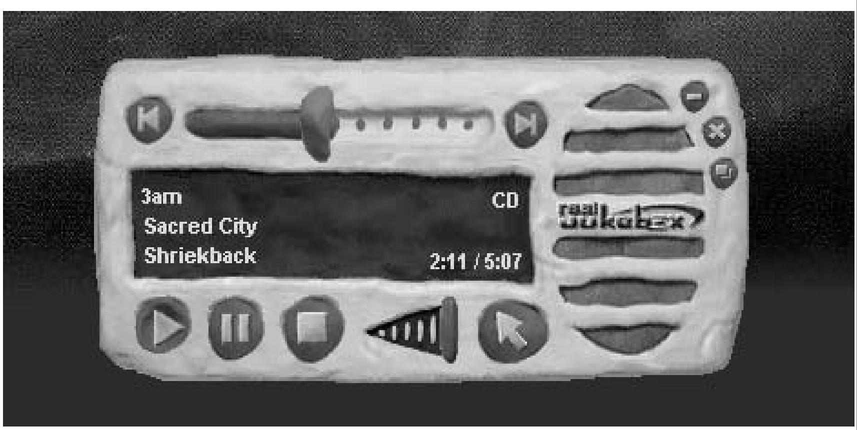
Sample heuristics

- Visibility of system status
  - keep users informed: appropriate feedback in reasonable time
- Match between system and the real world
  - familiar language, not system-oriented terms: obey real-world conventions & natural order
- User control and freedom
  - clearly marked “emergency exit”, undo & redo

Sample heuristics

- Consistency and standards
  - platform conventions, not new names for same things
- Error prevention
  - prevent problem from occurring in the first place
- Recognition rather than recall
  - visible actions & options, don't rely on user memory

Evaluation example



Partial evaluation example

- Visibility of system status
  - Current track, time, all visible
- Match between system and the real world
  - Like a tape recorder
- Help user recognise & recover from errors
  - Not many errors possible
  - But can’t get back to where you were after an accidental track change
- Consistency and standards
  - Access to Windows menu is unexpected

Interaction styles summary

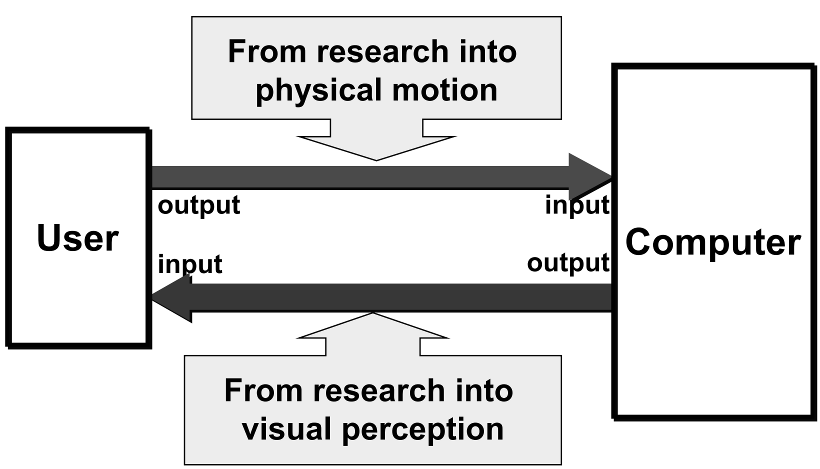
- History of interaction styles has emphasised changing units of interaction.
- Heuristic evaluation is at present the most popular usability evaluation method:
  - simple and cheap to conduct
  - easily justifiable on commonsense grounds
- Disadvantages
  - doesn’t address deeper system design problems
  - These require not just a surface description, but a model of the user.

User-centred design

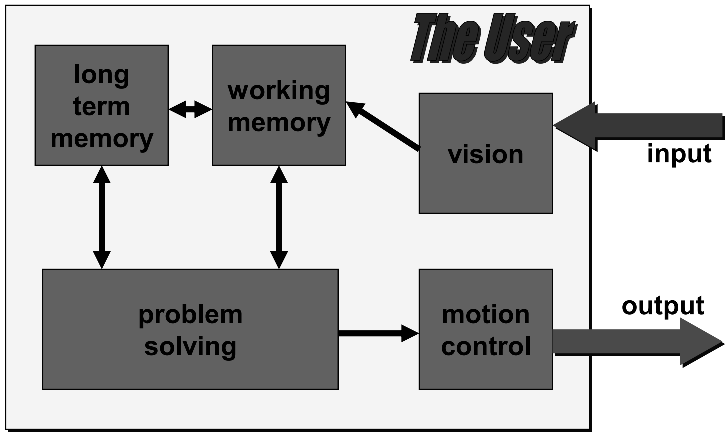
- Early focus on users and tasks
  - Cognitive models of user needs
  - Ethnographic observation of task context
- Empirical measurement
  - Experimental studies
    - Hypothesis testing methods
    - Think-aloud protocols
  - Surveys and questionnaires
    - Structured access to introspective data
- Iterative design
  - Prototyping
  - Contextual design

Models of user cognition

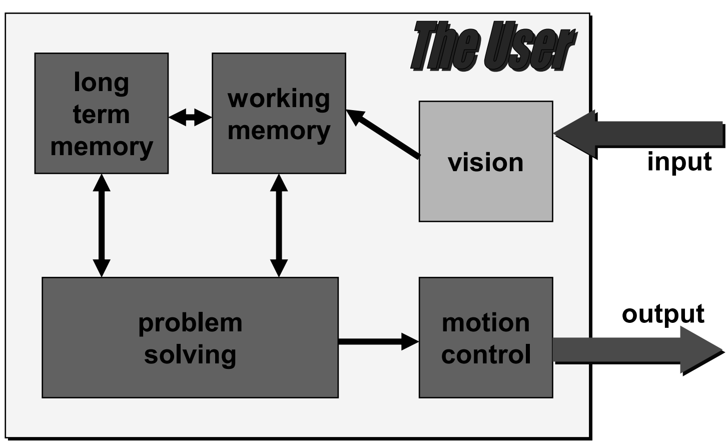
- Engineering view of the user (black box):



Top-down decomposition of user

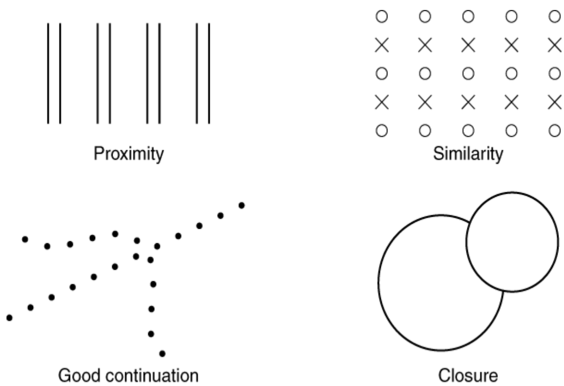


Vision



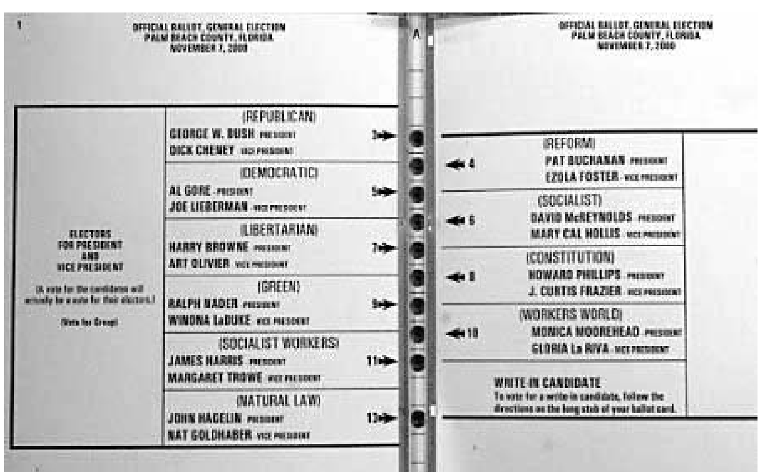
Gestalt laws of perception

- Principles of 2D display perception from 1920s



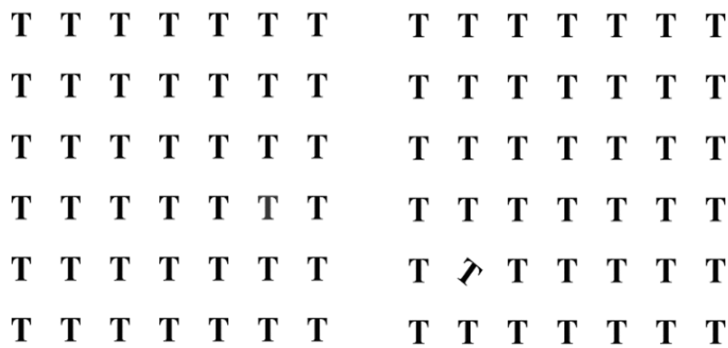
Gestalt laws in user interface

- Palm Beach County, Florida - U.S. Presidential Election 2000



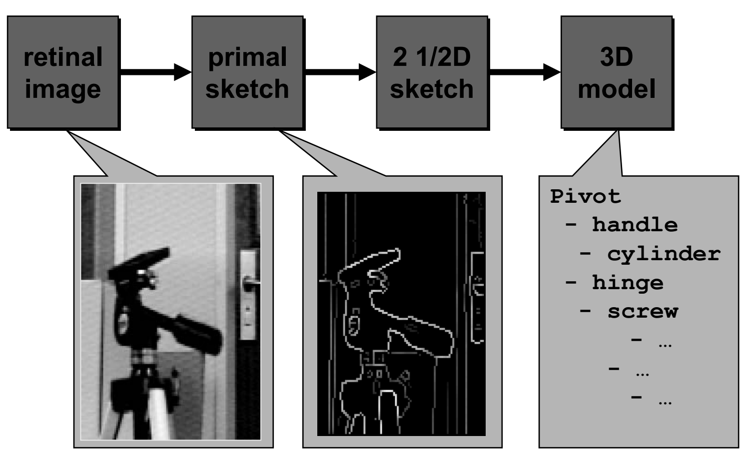
Visual search

- Finding letter in a list: $T_{LIST} \sim kN$
  `mvunmgsuignshetovazcvteown`
- Finding different {colour,orientation}: $T_{POPOUT} \sim k$



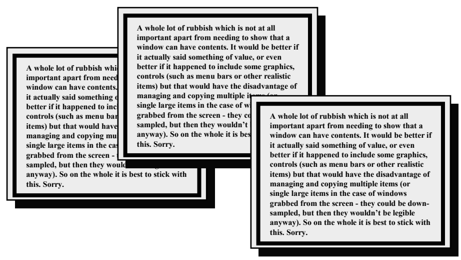
Visual input decomposed

- Marr's black box theory of vision

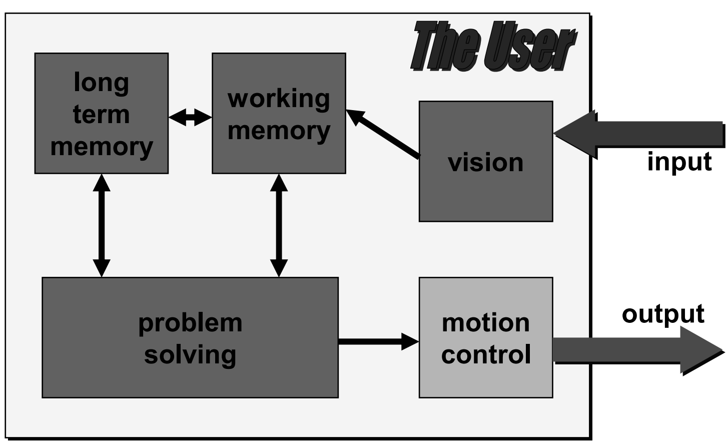


Marr's theory of vision

- The black boxes in Marr's model are almost an inverse of the modelling and rendering processes in 3D computer graphics
  - Is this suspicious?
- Whatever the 3D situation, current displays are actually rendered in 2 1/2 dimensions:

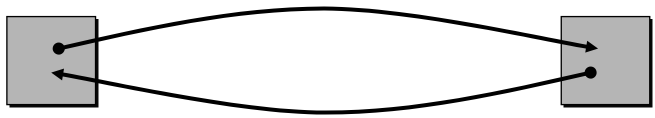


Motion control



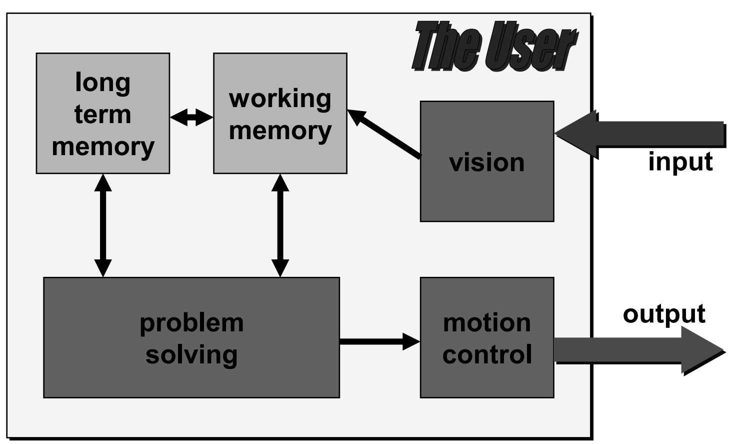
Motion control - Fitts' Law

- From experiments moving between two targets
  - Time to point at a target depends on:
    - target width
    - amplitude of movement



  - $T = K \log_2 (A / W + 1)$
- Useful prediction of time to move mouse across the screen to click on a button

Memory

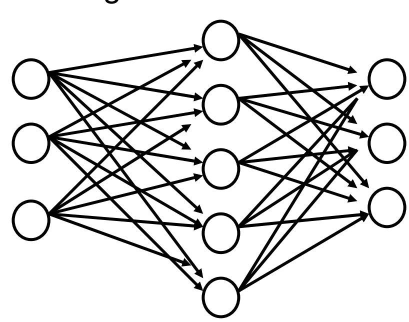


Working memory - Miller

- How many things can we remember at one time?
  - **`4786779`**
  - **`G522KUJ`**
  - In lists of letters, objects, digits: between 5 and 9
- ***Seven plus or minus two*** (G.A. Miller, 1956)
- Content is significant, e.g. remember 25 letters:
  - **`ksnehfifmwbtdoanebgocnesj`**
  - **`fruitapplegrapeguavalemon`**
- Working (short-term) memory can retain around 7+/-2 ***chunks*** of information.

Long term memory

- Learning involves re-coding from short-term to long-term memory.
- Distributed "connectionist" models of memory:
  - re-coding involves forming associations
  - (this model does not account for semantic structure).



Memory coding demonstration

- Get a pencil and paper

Word list memory

- **keyboard**
- **notebook**
- **speed**
- **banana**
- **absence**
- **withhold**
- **telephone**
- **category**
- **pencil**
- **rucksack**
- **concern**
- **camel**
- **classic**
- **right**
- **bicycle**
- **transfer**
- **operation**
- **armchair**

Write them down!

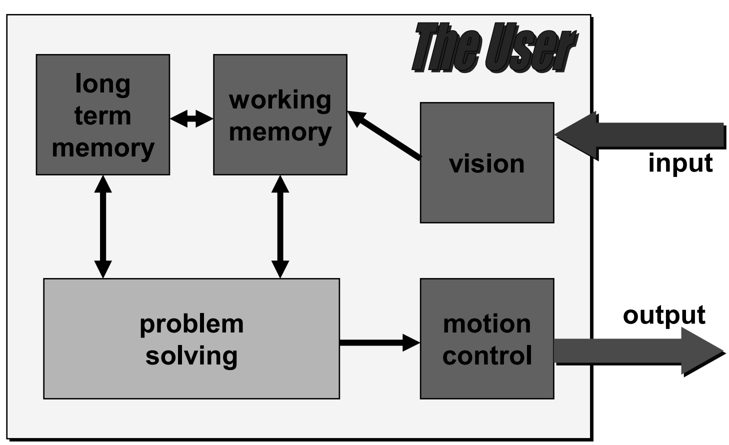
Word list memory

- keyboard
- camel
- rucksack
- bicycle
- armchair
- banana
- notebook
- telephone
- pencil
- concern
- speed
- absence
- withhold
- category
- classic
- right
- transfer
- operation

Working memory and mnemonics

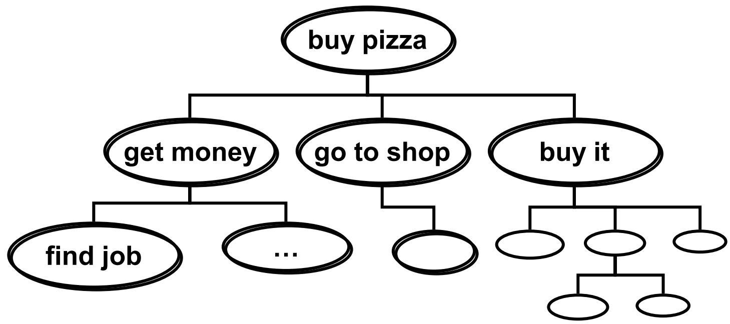
- Two kinds of working memory
  - Phonological loop (lists you can pronounce)
  - Visual-spatial sketchpad (mental images)
- Both modes can contribute to associations in long term memory.
  - Both at once produce stronger memory trace
- Words that can be visualised are easier to remember - ***dual coding*** (Paivio 1972)
- Basis of mnemonic techniques
  - explain benefit of pictorial mnemonics in UIs

Problem solving



Problem solving - GPS

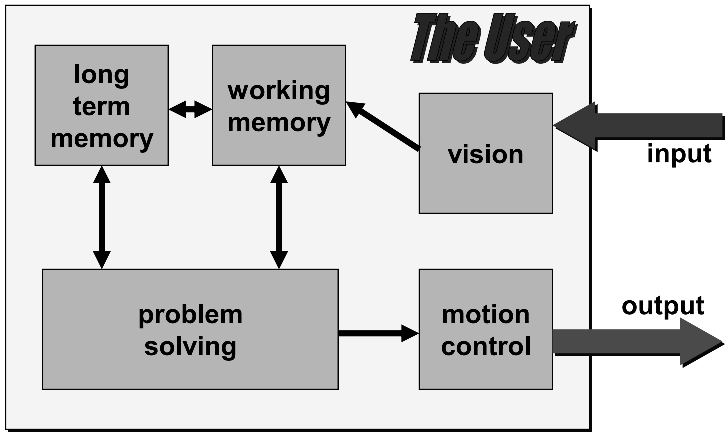
- Generalised Problem Solver (Ernst & Newell 1969)
  - Reduce difference between current and goal state
  - Decompose new goal into sub-goals (recursively)



Implications of GPS

- Computational model of problem-solving
- Recursive difference reduction results in a ***sub-goal hierarchy.***
- Leaves of the goal tree are physical operations.
- Main function of perceptual (visual) input is to identify required difference reductions.
- Working memory imposes limits on depth of goal tree (like a stack overflow).

Cognitive model of user needs



The Model Human Processor

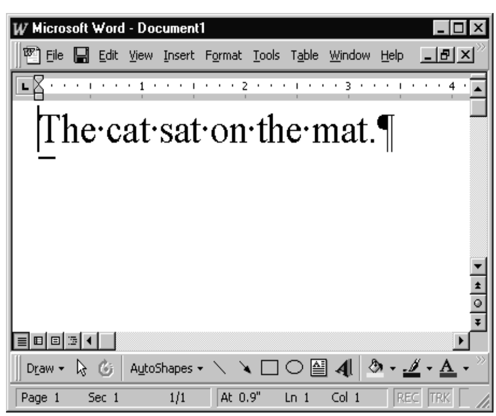
- Combine various psychological observations into a common quantitative framework.
- Decompose basic actions involved in user interaction tasks into:
  - perceptual events
  - motion events
  - cognitive events
- Keystroke level model: Evaluate interface options by estimating total time taken for all operations

Keystroke Level Model Units

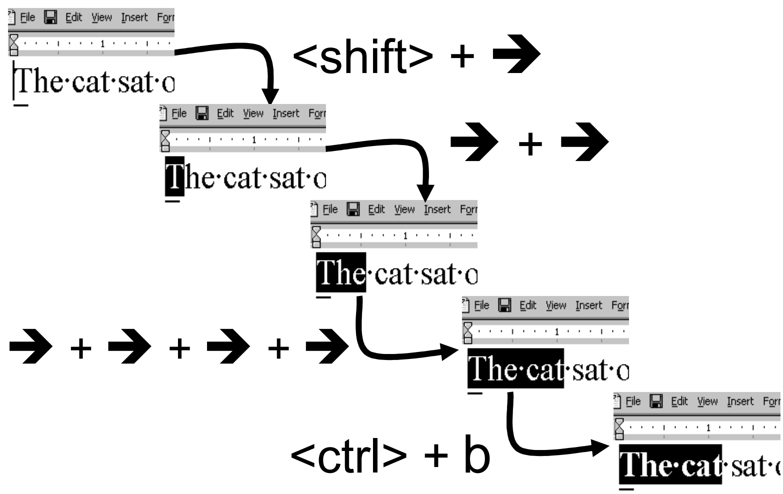
- **K**: press a key (constant given task and skill)
  - 0.12 seconds for good typist, 0.28 seconds for average, 0.75 seconds for difficult tasks.
- **H**: home hands on mouse or keyboard
  - 0.40 seconds.
- **P**: point with a mouse (using Fitts' law)
  - 0.8 to 1.5 seconds, 1.1 seconds on average.
- **D**: draw with mouse
  - forget this - original assumptions now outdated.
- **R**: system response time to an action.
- **M**: time the user takes to mentally prepare action.

KLM Example

- Problem: How long does it take to reformat a word in bold type within Microsoft Word, using either:
  a) Keys only
  b) Font dialog



Keys-only method



Keys-only method

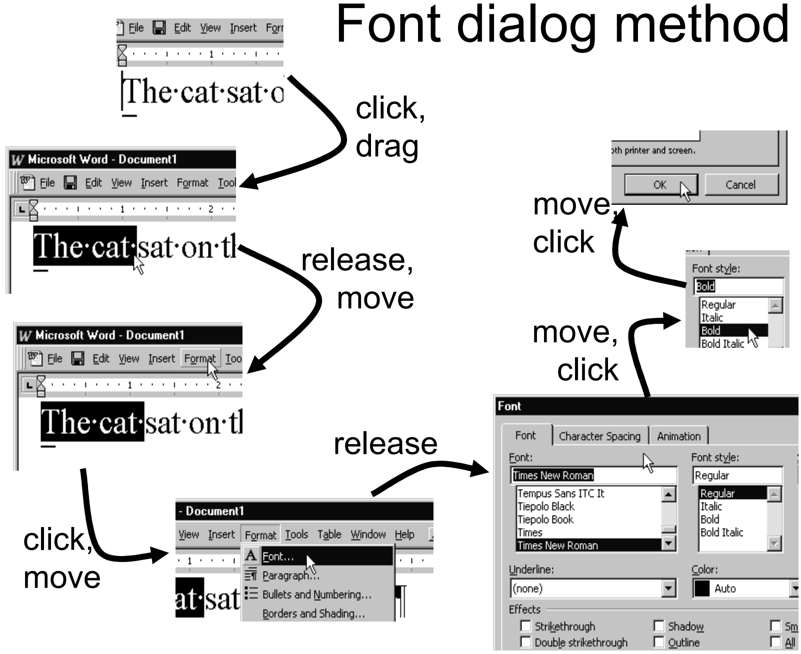
- Mental preparation: **M**
- Home on keyboard: **H**
- Mental preparation: **M**
- Hold down shift: **K**
- Press ➔ : **K**
- Press ➔ : **K**
- Press ➔ : **K**
- Press ➔ : **K**
- Press ➔: **K**
- Press ➔: **K**
- Press ➔ : **K**
- Release shift: **K**
- Mental preparation: **M**
- Hold down control: **K**
- Press b: **K**
- Release control: **K**

Keys-only method

- 1 occurrence of H — 0.40
- 3 occurrences of M — 1.35 * 3
- 12 occurrences of K — 0.28 * 12

7.81 seconds

Font dialog method



Fitts' law estimate

- Would normally calibrate experimentally
- Crude estimate based on screen distance, and KLM performance average:
  - $T = K \log_2 (A / W + 1) = 1.1s$ (on average)
  - Average distance: half window size ~ 220 pixels
  - Average button width: menu item radius ~ 32 pixels
  - $K = 1.1 / \log_2 (220 / 32 + 1)$
    $= 0.3695$

Motion time estimates

- Estimate K = 0.36 (from part II HCI course)
- From start of "The" to end of "cat" ($T = 0.36 \log_2 (A / W + 1)$):
  - distance 110 pixels, width 26 pixels, T = 0.88 s
- From end of "cat" to Format item on menu bar:
  - distance 97 pixels, width 25 pixels, T = 0.85 s
- Down to the Font item on the Format menu:
  - distance 23 pixels, width 26 pixels, T = 0.34 s
- To the "bold" entry in the font dialog:
  - distance 268 pixels, width 16 pixels, T = 1.53 s
- From "bold" to the OK button in the font dialog:
  - distance 305 pixels, width 20 pixels, T = 1.49 s

Font dialog method

- Mental preparation: **M**
- Reach for mouse: **H**
- Point to “The”: **P**
- Click: **K**
- Drag past “cat”: **P**
- Release: **K**
- Mental preparation: **M**
- Point to menu bar: **P**
- Click: **K**
- Drag to “Font”: **P**
- Release: **K**
- Mental preparation: **M**
- Move to “bold”: **P**
- Click: **K**
- Release: **K**
- Mental preparation: **M**
- Move to “OK”: **P**
- Click: **K**

Font dialog method

1 occurrence of H	0.40
4 occurrences of M	1.35 * 4
7 occurrences of K	0.28 * 7
6 mouse motions P	1.1 + 0.88 + 0.85 + 0.34 + 1.53 + 1.49
Total for dialog method:	**13.95 seconds** (+ 1× R)
Total for keyboard method:	vs. **7.81 seconds**

GOMS

- Extension of Keystroke Level Model:
  - GOMS = Goals Operators Methods Selection
- Includes model of problem solving based on General Problem Solver
  - User has some *goal* that can be decomposed.
  - *Operators* are those at the keystroke level.
  - Experienced users have a repertoire of *methods*.
  - Time is required to *select* a method for some goal.
- Model also accounts for memory and learning.

GOMS/KLM Assessment

- Can give actual quantitative performance estimates for a user interface design.
- Keystroke level model only describes *expert user* carrying out *familiar task*.
  - Only weak representation of perceptual, cognitive and motor subsystems
  - No consideration of the user’s knowledge.
- GOMS doesn’t account for main variations in performance
  - Errors
  - Strategy change

Beyond black box user models

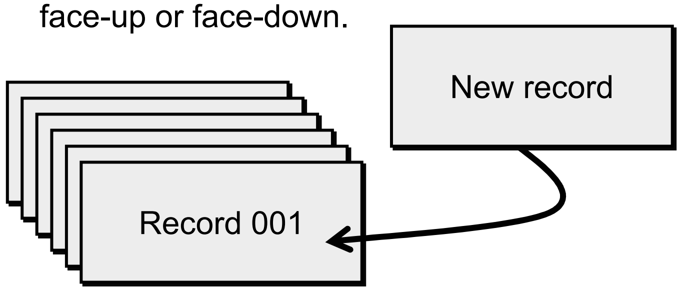
- Why do users do the things they do?
- *Mental models* of devices and programs
  - adequate for the task …
  - but may break down in unusual situations
- e.g. mental model of electricity as flowing water: (Gentner & Gentner 1983)
  - “taps” turn the flow on, “hoses” direct it to go where you need it.
  - it can leak out and make a mess.
- This is good enough! Detailed technical models don’t always help.

User’s model?

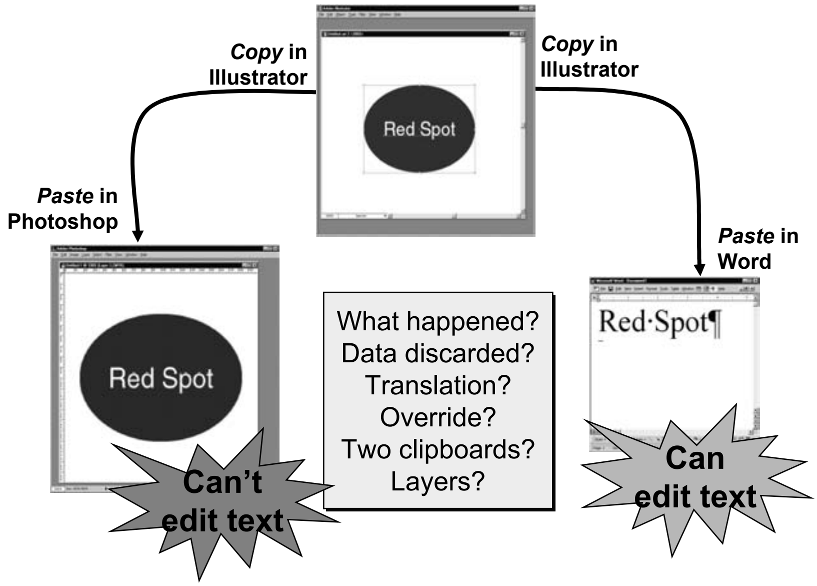
Designer's model

Mental simulation of a model

- Adding new record to a database
  - Will the new record appear as the first or last?
  - If mental model of database is as a stack of cards …
  - … the answer depends on whether the stack is face-up or face-down.

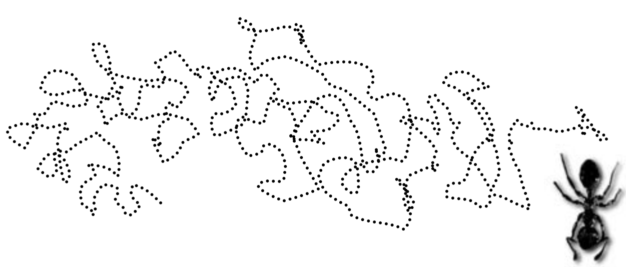


Problem solving with a model



Understanding user context

- Model-based planning theories neglect the problem of *situated action*
- Sometimes complex behaviour simply results from a complex environment – cognitive modelling is not enough to understand it.



Ethnographic field studies

- Emphasise the detail of user activity, not theories and rationalisation.
- Researchers work in the field
  - Observing context of user's work
  - Participating in subjects' activities.
- Main characteristics:
  - Observe subjects in a range of *contexts*.
  - Observe over a substantial *period of time*.
  - Full record of both *activities* and *artefacts*.
- Transcription from video/audio recordings

Structured ethnographic analysis

- Division of labour and its coordination
- Plans and procedures
  - When do they succeed and fail?
- Where paperwork meets computer work
- Local knowledge and everyday skills
- Spatial and temporal organisation
- Organisational memory
  - How do people learn to do their work?
  - Do formal methods match reality?

Interviews

- Ethnographic observation is usually supplemented by interview
- Often conducted in the place of work during *contextual enquiry*.
  - Encourages emphasis on user activity, rather than research concerns
- Can alternatively be theory-driven, with questions structured to:
  - collect data into common framework
  - ensure all important aspects covered

Empirical studies of usability

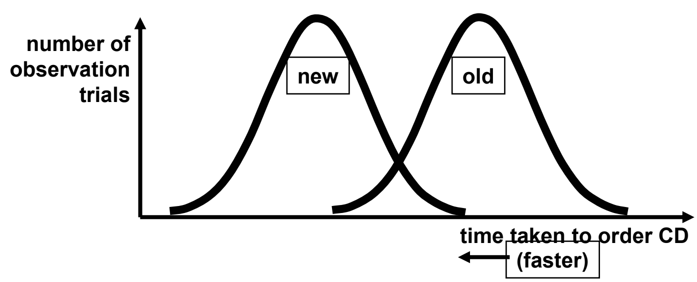
- Empirical measures tend to investigate specific questions
  - Ideally questions identified from contextual study
- Measure user characteristics
  - Estimate parameters of skilled performance
  - Identify common mental models
- Investigate potential designs
  - Compare benefits of alternatives
  - Assess performance against design goals

Controlled experiments

- Based on a number of observations:
  - How *long* did Fred take to order a CD from Amazon?
  - How many *errors* did he make?
- But every observation is different.
- So we compare averages:
  - over a number of trials
  - over a range of people (experimental subjects)
- Results usually have a normal distribution

Experimental treatments

- A *treatment* is some modification that we expect to have an effect on usability:
  - How long does Fred take to order a CD using this great new interface, compared to the crummy old one?
  - Expected answer: *usually* faster, but not *always*



Think-aloud studies

- Gain some understanding of mental models.
- Subject talks continuously while performing a defined experimental task.
- transcribed as a *verbal protocol* for detailed study of what user thinks is happening.
- Can be used to assess usability of prototypes during empirical evaluation, identifying *breakdowns* in usage or understanding.

Surveys and questionnaires

- Collect *subjective* evaluation from users
  - more like market research than like opinion polls
- *Closed* questions …
  - yes/no or *Likert* scale (opinion from 1 to 5)
  - useful for statistical comparison
- *Open* questions …
  - require *coding frame* to structure data
  - useful for exploratory investigation
- Questionnaires: valuable for online studies

Product field testing

- Brings advantages of task analysis to assessment & testing phases of product development.
- Case study: Intuit Inc.'s *Quicken* product
  - originally based on interviews and observation
  - *follow-me-home* programme after product release:
    - random selection of shrink-wrap buyers;
    - observation while reading manuals, installing, using.
  - Quicken success was attributed to the programme:
    - survived predatory competition, later valued at $15 billion.

Bad empirical techniques

- Purely *affective* reports: 20 subjects answered the question "Do you like this nice new user interface more than that ugly old one?"
- No observation at all: "It was decided that more colours should be used in order to increase usability."
- Introspective reports made by a single subject (often the programmer or project manager): "I find it far more intuitive to do it this way, and the users will too."

Iterative design

- Cycle of construction and evaluation
- User interface designs are seldom right the first time, so improve chances of meeting user's needs by repeating cycle of:
  - building a prototype
  - trying it out with users.
- Accurate simulation of interface helps develop and assess mental models
- Either illustrative mock-ups of interface, or interactive *rapid prototypes* as basis for discussion.

Prototyping product concepts

- Emphasise appearance of the interface, create some behaviour with scripting functions:
  - Visio – diagrams plus behaviour
  - Macromedia Director – movie sequence
  - JavaScript – web pages
  - Visual Basic (or VBA in PowerPoint, Excel …)
- Cheap prototypes are good prototypes
  - More creative solutions are often discovered by building more prototypes.
  - Glossy prototypes can be mistaken for the real thing – either criticised more, or deployed!

Prototypes without programming

- Low-fidelity prototypes (or mockups)
  - Paper-and-glue simulation of interface
  - User indicates action by pointing at buttons on the paper "screen"
  - Experimenter changes display accordingly
- "*Wizard of Oz*" simulation method
  - Computer user interface is apparently operational
  - Actual system responses are produced by an experimenter in another room.
  - Can cheaply assess effects of "intelligent" interfaces

Participatory design

- Users become partners in the design team
  - Originated in Scandinavian printing industry
  - Recent research even includes children
- PICTIVE method
  - Users generate scenarios of use in advance
  - Low fidelity prototyping tools (simple office supplies) are provided for collaborative session
  - The session is videotaped for data analysis
- CARD method
  - Cards with screen-dumps on them
  - Cards are arranged and rearranged on the table to explore workflow options

New trends in user interfaces

- Information appliances / Ubiquitous computing
- Adding computing and communication functions to common devices:
  - toasters, desks, radios, refrigerators
- Integrating devices into the environment:
  - by extending existing devices
    - PDAs, cellphones, smart cards
  - through new device categories
    - intelligent walls/paper, active badges, keyrings, jewellery
- Emphasise functionality with minimal "interface"

Part IV
Design Challenges

2 lectures

Design Challenges

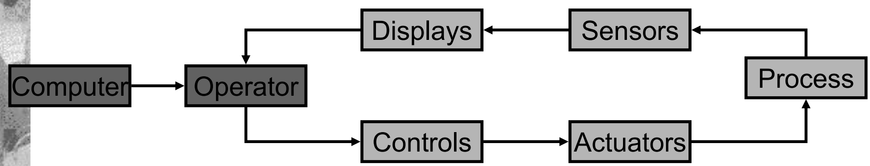
- Human errors and critical systems
- Hazards
- Risk
- Reliability
- Management failure (CAPSA case study).

'Human Error' probabilities

- Extraordinary errors 10^{-5}
  - difficult to conceive how they would occur
  - stress free environment, powerful success cues
- Errors in common simple tasks 10^{-4}
  - regularly performed, minimum stress involved
- Press wrong button, read wrong display 10^{-3}
  - complex tasks, little time, some cues necessary
- Dependence on situation and memory 10^{-2}
  - unfamiliar task with little feedback and some distraction:
- Highly complex task 10^{-1}
  - considerable stress, little time to perform
- Unfamiliar and complex operations O(10^**0**)
  - involving creative thinking, time short, stress high

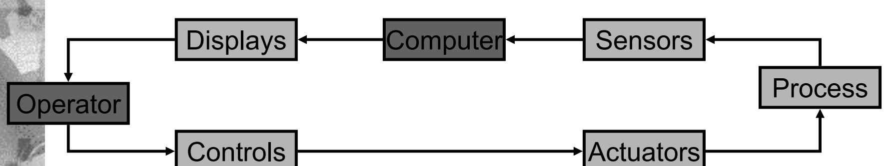
"Skill is more reliable than knowledge"

Modes of Automation



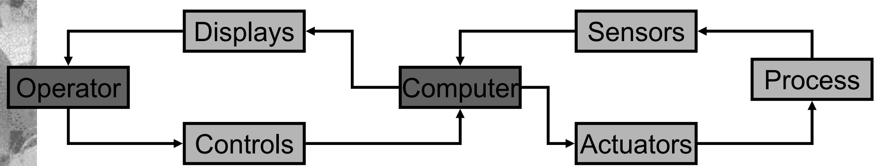
(a) Computer provides information and advice to operator (perhaps using a mechanical or electrical display, perhaps by reading sensors directly)

Modes of Automation



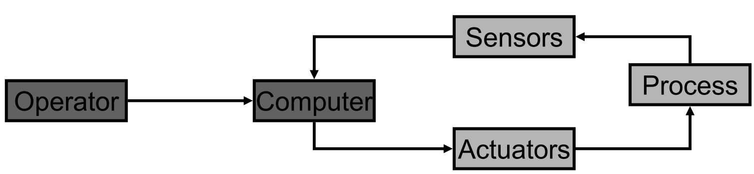
(b) Computer reads and interprets sensor data for operator

Modes of Automation



(c) Computer interprets and displays data for operator and issues commands; operator makes varying levels of decisions

Modes of Automation



(d) Computer assumes complete control of process with operator providing advice or high-level direction

Critical software

- Many systems have the property that a certain class of failures is to be avoided if at all possible
  - safety critical systems
    - failure could cause death, injury or property damage
  - security critical systems
    - failure could result in leakage of classified data, confidential business data, personal information
  - business critical systems
    - failure could affect essential operations
- Critical computer systems have a lot in common with critical mechanical or electrical systems
  - bridges, flight controls, brakes, locks, ...
- Start out by studying how systems fail

Definitions

- **Error**:
  - design flaw or deviation from intended state
- **Failure**:
  - non-performance of the system within some subset of the specified environmental conditions
- **Fault**:
  - Computer science: error → fault → failure
    - but note electrical engineering terminology: (error →) failure → fault
- **Reliability**:
  - probability of failure within a set period of time
  - Sometimes expressed as ‘mean time to (or between) failures’ - mttf (or mtbf)

More definitions

- **Accident**
  - undesired, unplanned event that results in a specified kind (and level) of loss
- **Hazard**
  - set of conditions of a system, which together with conditions in the environment, will lead to an accident
  - thus, failure + hazard → accident
- **Risk**: hazard level, combined with:
  - **Danger:** probability that hazard → accident
  - **Latency:** hazard exposure or duration
- **Safety**: freedom from accidents

System Safety Process

- Obtain support of top management, involve users, and develop a system safety program plan:
  - identify hazards and assess risks
  - decide strategy for each hazard (avoidance, constraint,....)
  - trace hazards to hardware/software interface: which will manage what?
  - trace constraints to code, and identify critical components and variables to developers
  - develop safety-related test plans, descriptions, procedures, code, data, test rigs ...
  - perform special analyses such as iteration of human-computer interface prototype and test
  - develop documentation system to support certification, training ,..
- Safety needs to be designed in from the start. It cannot be retrofitted

Real-time systems

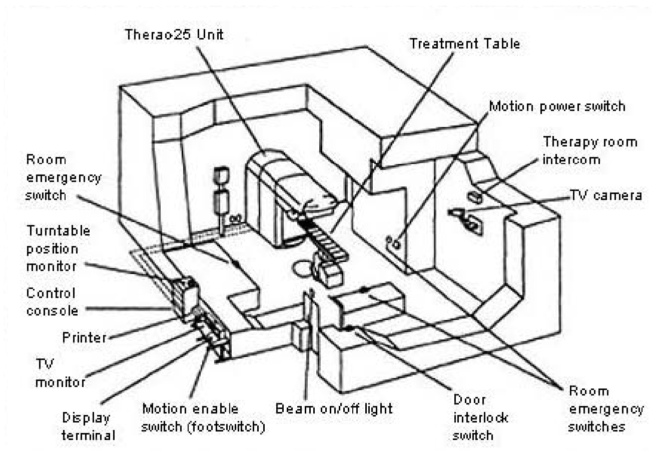
- Many safety critical systems are also *real time*
  - typically used in monitoring or control
- These have particular problems
  - Extensive application knowledge often needed for design
  - Critical timing makes verification techniques inadequate
  - Exception handling particularly problematic.
- eg Ariane 5 (4 June 1996):
  - Ariane 5 accelerated faster than Ariane 4
  - alignment code had an ‘operand error’ on float-to-integer conversion
  - core dumped, core file interpreted as flight data
  - full nozzle deflection → 20 degrees angle of attack → booster separation → self destruct

Hazard Analysis

- Often several hazard categories e.g. Motor Industry Software Reliability Association uses:
  - **Uncontrollable**: failure outcomes not controllable by humans and likely to be extremely severe
  - **Difficult to control**: effects might possibly be controlled, but still likely to lead to very severe outcomes
  - **Debilitating**: effects usually controllable, reduction in safety margin, outcome at worst severe
  - **Distracting**: operational limitations, but a normal human response limits outcome to minor
  - **Nuisance**: affects customer satisfaction, but not normally safety
- Different hazard categories require different failure rates and different levels of investment in varying software engineering techniques

THERAC-25

- 25 MEV ‘therapeutic accelerator’ for radiotherapy cancer treatment



THERAC-25 operation

- Two modes of operation:
  - 25 MEV focused electron beam on a target that generates X-rays for treating deep tumours
  - 0.25 MEV spread electron beam for direct treatment of surface tumours
- Patient in shielded room, operator console outside
  - operator confirms dosage settings from console
- Turntable between patient and beam contains:
  - scan magnet for steering low power beam
  - X-ray target to be placed at focus of high power beam
  - plunger to stop turntable in one or other position
  - microswitches on the rim to detect turntable position

THERAC hazard

- Focused beam for X-ray therapy
  - 100x the beam current of electron therapy
  - highly dangerous to living tissue
- Previous models (Therac 6 and 20)
  - fuses and mechanical interlocks prevented high intensity beam selection unless X-ray target in place
- Therac 25 safety mechanisms replaced by software.
  - fault tree analysis arbitrarily assigned probability 10^{-11} to fault 'computer selects wrong energy'.
- But from 1985-87, at least six accidents
  - patients directly irradiated with the high energy beam
  - three died as consequence
- Major factors: poor human computer interface, poorly written, unstructured code.

The THERAC accidents

- Marietta, Georgia, June 1985:
  - Woman's shoulder burnt. Sued & settled out of court. Not reported to FDA, or explained
- Ontario, July 1985:
  - Woman's hip burnt. Died of cancer. 1-bit switch error possible cause, but couldn't reproduce the fault.
- Yakima, Washington, December 85:
  - Woman's hip burnt. Survived. 'Could not be a malfunction'
- Tyler, Texas, March 86:
  - Man burned in neck and died. AECL denied knowledge of any hazard
- Tyler, Texas, April 86:
  - 2nd man burnt on face and died. Hospital physicist recreated fault: if parameters edited too quickly, interlock overwritten
- Yakima, Washington, January 87:
  - Man burned in chest and died. Due to different bug thought now to have also caused the Ontario accident

THERAC lessons learned

- AECL ignored safety aspects of software
  - assumed when doing risk analysis (and investigating Ontario) that hardware must be at fault
- Confused reliability with safety
  - software worked & accidents rare …
  - … so assumed it was ok
- Lack of defensive design
  - machine couldn't verify that it was working correctly
- Failure to tackle root causes
  - Ontario accident not properly explained at the time (nor was first Yakima incident ever!)

More THERAC lessons

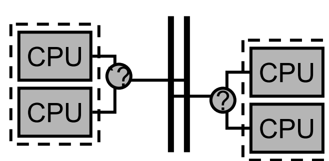
- Complacency
  - medical accelerators previously had good safety record
- Unrealistic risk assessments
  - "think of a number and double it"
- Inadequate reporting, follow-up and government oversight.
- Inadequate software engineering
  - specification an afterthought
  - complicated design
  - dangerous coding practices
  - little testing
  - careless human interface
  - careless documentation design

Failure modes & effects analysis

- FMEA is heart of NASA safety methodology
  - software not included in NASA FMEA
  - but other organisations use FMEA for software
- Look at each component's functional modes and list the potential failures in each mode.
  - Describe worst-case effect on the system
    - 1 = loss of life
    - 2 = loss of mission
    - 3 = other
  - Secondary mechanisms deal with interactions
- Alternative: Fault Tree Analysis
  - work back systematically from each identified hazard
  - identify where redundancy is, which events are critical

Redundancy

- Some systems, like Stratus & Tandem, have highly redundant hardware for 'non-stop processing'



- But then software is where things break
- 'Hot spare' inertial navigation on Ariane 5 failed first!
- Idea: multi-version programming
  - But: significantly correlated errors, and failure to understand requirements comes to dominate (Knight, Leveson 86/90)
- Also, many problems with redundancy management. For example, 737 crashes Panama/Kegworth

Example - Kegworth Crash

- British Midland 737-400 flight 8 January 1989
  - left Heathrow for Belfast with 8 crew + 118 passengers
  - climbing at 28,300’, fan blade fractured in #1 (left) engine. Vibration, shuddering, smoke, fire
  - Crew mistakenly shut down #2 engine, cut throttle to #1 to descend to East Midlands Airport.
  - Vibration reduced, until throttle reopened on final approach
  - Crashed by M1 at Kegworth. 39 died in crash and 8 later in hospital; 74 of 79 survivors seriously injured.
- Initial assessment
  - engine vibration sensors cross-wired by accident
- Mature assessment
  - crew failed to read information from new digital instruments
- Recommendations:
  - human factors evaluations of flight systems, clear ‘attention getting facility’, video cameras on aircraft exterior

Myths of software safety

- *Computers are cheaper than analogue or electromechanical devices*
  - shuttle software costs $100,000,000 p.a. to maintain
- *Software is easy to change*
  - but hard (and expensive) to change safely
- *Computers are more reliable*
  - shuttle had 16 potentially fatal bugs since 1980 – half of them had actually flown
- *Increasing software reliability increases safety*
  - perfectly functioning software still causes accidents

More myths

- *Testing or formal verification can remove all errors*
  - exhaustive testing is usually impossible
  - proofs can have errors too
- *Software reuse increases safety*
  - using the same software in a new environment is likely to uncover more errors
- *Automation can reduce risk*
  - potential not always realised, humans still need to intervene

CAPSA project

- Now Cambridge University Financial System
- Previous systems:
  - In-house COBOL system 1966-1993
    - Didn't support commitment accounting
  - Reimplemented using Oracle + COTS 1993
    - No change to procedures, data, operations
- First attempt to support new accounts:
  - Client-server "local" MS Access system
  - To be "synchronised" with central accounts
  - Loss of confidence after critical review
- May 1998: consultant recommends restart with "industry standard" accounting system

CAPSA project

- Detailed requirements gathering exercise
  - Input to supplier choice between Oracle vs. SAP
- Bids & decision both based on optimism
  - ‘vapourware’ features in future versions
  - unrecognised inadequacy of research module
  - no user trials conducted, despite promise
- Danger signals
  - High ‘rate of burn’ of consultancy fees
  - Faulty accounting procedures discovered
  - New management, features & schedule slashed
  - Bugs ignored, testing deferred, system went live
- “Big Bang” summer 2000: CU seizes up

CAPSA mistakes

- No phased or incremental delivery
- No managed resource control
- No analysis of risks
- No library of documentation
- No requirements traceability
- No policing of supplier quality
- No testing programme
- No configuration control

CAPSA lessons

- Classical system failure (Finkelstein)
  - More costly than anticipated
    - £10M or more, with hidden costs
  - Substantial disruption to organisation
  - Placed staff under undue pressure
  - Placed organisation under risk of failing to meet financial and legal obligations
- Danger signs in process profile
  - Long hours, high staff turnover etc
- Systems fail systemically
  - not just software, but interaction with organisational processes

Problems of large systems

- Study of 17 large & demanding systems
  - (Curtis, Krasner, Iscoe, 1988)
  - 97 interviews investigated organisational factors in project failure
- Main findings - large projects fail because
  - (1) thin spread of application domain knowledge
  - (2) fluctuating and conflicting requirements
  - (3) breakdown of communication and coordination
- These were often linked, with typical progression to disaster (1) → (2) → (3)

More large system problems

- Thin spread of application domain knowledge
  - who understands all aspects of running a telephone service/bank branch network/hospital?
  - many aspects are jealously guarded secrets
  - sometimes there is structured knowledge (eg pilots)
  - otherwise, with luck, you may find a genuine 'guru‘
  - So expect specification mistakes
- Even without mistakes, specification may change:
  - new competitors, new standards, new equipment, fashion
  - change in client: takeover, recession, refocus, …
  - new customers, e.g. overseas, with different requirements
- Success and failure both bring their own changes!

More large system problems

- How to cope with communications overhead?
  - Traditionally via hierarchy
    - information flows via managers, they get overloaded
  - Usual result - proliferation of committees
    - politicking, responsibility avoidance, blame shifting
  - Fights between 'line' and 'staff' departments
    - Management attempts to gain control may result in constriction of some interfaces, e.g. to customer
  - Managers often loath to believe bad news
    - much less pass it on
  - Informal networks vital, but disrupted by 'reorganisation‘

- We trained hard, but it seemed that every time we were beginning to form up into teams, we would be reorganised. I was to learn later in life that we tend to meet any new situation by reorganising, and a wonderful method it can be for creating the illusion of progress while producing confusion, inefficiency and demoralisation.
  - Caius Petronius (AD 66):

Part V
Project Management

2 lectures

Project Management

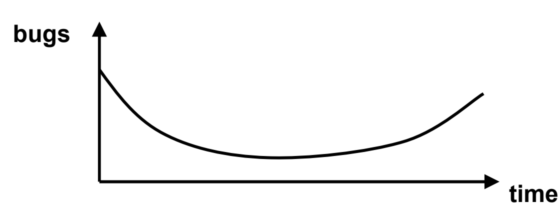
- Lifecycle costs and Brooks' Law
- The classic "waterfall model"
- Evolutionary and incremental models
  - Spiral model
  - Rapid Application Development
  - Rational Unified Process
- Novel structures
  - Chief programmer
  - Egoless programming
  - eXtreme Programming
- Changing (maturing) organisations

The software life cycle

- Cost of owning a system not just development but whole cost over life cycle:
  - Development, Testing, Operations, Replacement
- In ‘bespoke’ software days
  - 90% of IT department programming effort was maintenance of old systems
- Most research on software costs and methods focuses on this business model.
- Different business models apply
  - to safety critical and related software
  - to package software
  - but many lessons apply to them all

Common difficulties

- Code doesn't ‘wear out’ the way that gears in machinery do, but:
  - platform and application requirements change over time,
  - code becomes more complex,
  - it becomes less well documented,
  - it becomes harder to maintain,
  - it becomes more buggy.
- Code failure rates resemble those of machinery
  - (but for different reasons!)



More common difficulties

- When software developed (or redeveloped)
  - unrealistic price/performance expectations
  - as hardware gets cheaper, software seems dear
- Two main causes of project failure
  - incomplete/changing/misunderstood requirements
  - insufficient time
- These and other factors lead to the ‘tar pit’
  - any individual problem can be solved
  - but number and complexity get out of control

Life cycle costs

- Development costs (Boehm, 75)

	Reqmts/Spec	Implement	Test
Cm’d & Control	48%	20%	34%
Space	34%	20%	46%
O/S	33%	17%	50%
Scientific	44%	26%	30%
Business	44%	28%	28%

- Maintenance costs: typically ten times as much again

Reducing life cycle costs

- By the late 60's the industry was realising:

- Well built software cost less to maintain
- Effort spent getting the specification right more than pays for itself by:
  - reducing the time spent implementing and testing
  - reducing the cost of subsequent maintenance.

What does code cost?

- Even if you know how much was spent on a project,
  - how do you measure what has been produced?
  - Does software cost per mile / per gallon / per pound?
- Common measure is KLOC (thousand lines of code)
- First IBM measures (60's):
  - 1.5 KLOC / man year (operating system)
  - 5 KLOC / man year (compiler)
  - 10 KLOC / man year (app)
- AT&T measures:
  - 0.6 KLOC / man year (compiler)
  - 2.2 KLOC / man year (switch)

Metrics & estimation

- More sophisticated measures:
  - Halstead (entropy of operators, operands)
  - McCabe (graph complexity of control structures)
  - For estimation: "Function Point Analysis"
- Lessons learned from applying empirical measures:
  - main productivity gains come from using appropriate high level language
    - each KLOC does more
  - wide variation between individuals
    - more than 10 times

Brooks' Law

- Brooks' *The Mythical Man-Month* attacked idea that "men" and months interchangeable, because:
  - more people → more communications complexity
  - adding people → productivity drop as they are trained
- e.g consider project estimated at 3 men x 4 months
  - but 1 month design phase actually takes 2 months!
  - so 2 months left to do work estimated at 9 man-months
  - add 6 men, but training takes 1 month
  - so all 9 man-months work must be done in the last month.
- 3 months work for 3 can't be done in 1 month by 9 (complexity, interdependencies, testing, ...)
- Hence Brooks' Law:

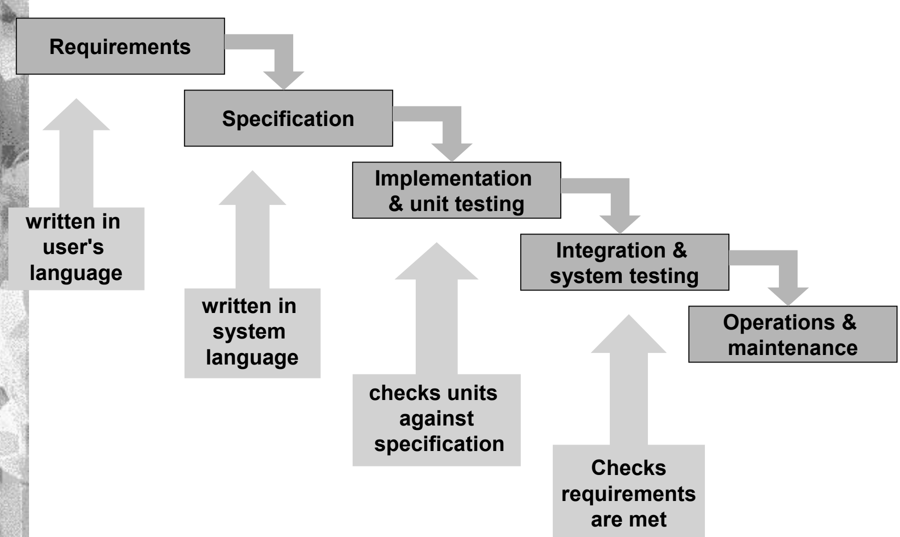
"Adding manpower to a late software project makes it later"

Boehm's empirical study

- Brooks' Law (described 1975) led to empirical studies
- Boehm *Software Engineering Economics,* 1981:
  - cost-optimum schedule time to first shipment, T
    - = 2.5 x cube root of total number of man months
  - with more time, cost rises slowly
    - 'people with more time take more time'
  - with less time, the cost rises sharply
  - Hardly any projects succeed in < 0.75T, regardless of number of people employed!
- Other studies show if more people are to be added, should be added early rather than late
- Some projects have more and more resources thrown at them yet are never finished at all, others are years late.

The Waterfall Model

- (Royce, 1970; now US DoD standard)



The Waterfall Model

Requirements are
developed by at le
two groups of peo
who speak differ
languages and w
come from differ
disciplines.

Specification, Design
and *Implementation*
are done b
of single-d
profession
usually
communicat
anoth

Installation is usually
done by people who
don't really
understand th
or the problen
solution

After a start-up period,
Op
alw
who
th
pr
(an

Maintenance is
usually performed by
inexperienced people
who have forgotten
much of what they
once knew about the
problem or the
solution.

(this information repeated on next slide)

Requirements are developed by at least two groups of people who speak different languages and who come from different disciplines.	*Specification, Design* and *Implementation* are done by a group of single-discipline professionals who usually can communicate with one another.	***Installation*** is usually done by people who don't really understand the issues or the problem or the solution.
After a start-up period, ***Operation*** is almost always left to people who don't understand the issues, ethics, problem or solution (and often little else).	***Maintenance*** is usually performed by inexperienced people who have forgotten much of what they once knew about the problem or the solution.	New York security consultant Robert Courtney examined 1000s of security breaches - 68% due to careless or incompetent operations.

Feedback in the waterfall model

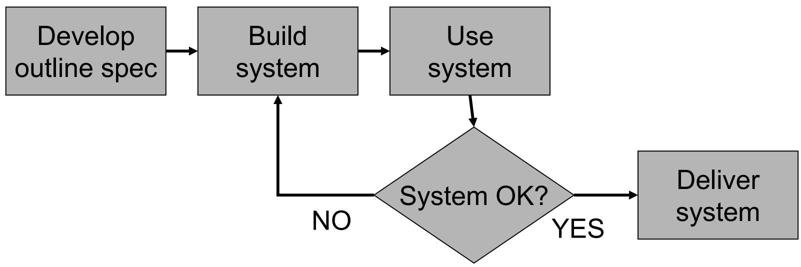
- *Validation* operations provide feedback
  - from Specification to Requirements
  - from Implementation/unit testing to Specification
- *Verification* operations provide feedback
  - from Integration/ system testing to Implementation/unit testing
  - from operations/maintenance back to Integration/system testing
- What's the difference?
  - Validation: `are we building the right system?'
  - Verification: `are we building it right?'
- What about validation from operations back to requirements?
  - this would change the model (and erode much of its value)

Advantages of waterfall model

- Project manager's task easier with clear milestones
- Can charge for requirement changes
  - each stage can even be a separate contract
- System goals, architecture & interfaces clarified together
  - conducive to good design practices
- Compatible with many tools and design methods
- *Where applicable*, waterfall is an ideal approach
  - critical factor: whether requirements can be defined in detail, in advance of any development or prototyping work.
  - sometimes they can (e.g. a compiler);
  - often they can't (e.g. user-centred design)

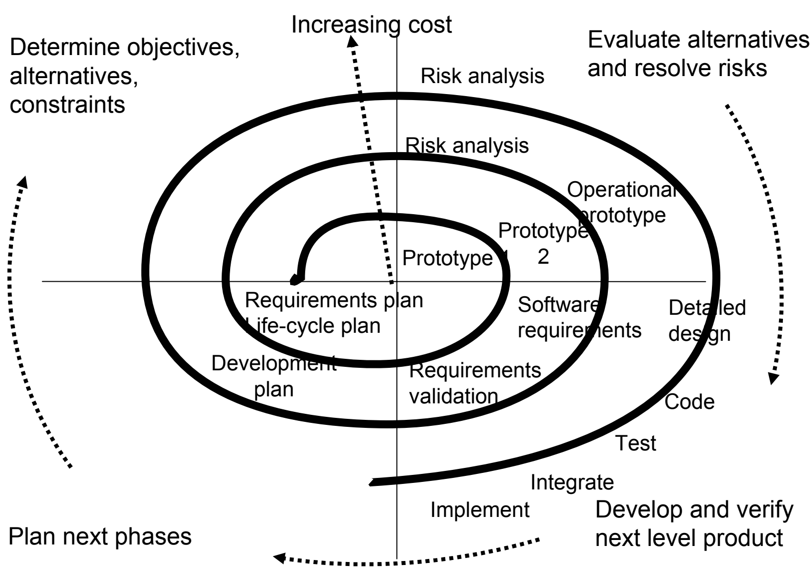
Iterative development

- Some systems need iteration to clarify requirements
- Others make operations fail-safe as possible
- Naive approach:



- This algorithm needn't terminate (satisfactorily)
- Can we combine management benefits of waterfall, with flexibility of iterative development?

Spiral model (Boehm, 88)



Features of spiral model

- Driven by risk management
- Fixed number of iterations, each of form:
  - identify alternatives, then
  - assess and choose, then
  - build and evaluate
- Allows for (some amount of) iterative prototyping in early stages

Rapid Application Development

- Mainly focused on user-centred design
- Includes “Joint Application Development”
  - Intensively collaborative requirements gathering exercise, with all stakeholders involved
- Implementation is iterative (<6 month cycles)
- Lifecycle phases
  - Project initiation
  - JAD workshop
  - Iterative design and build
  - Evaluate final system
  - Implementation review

Rational Unified Process

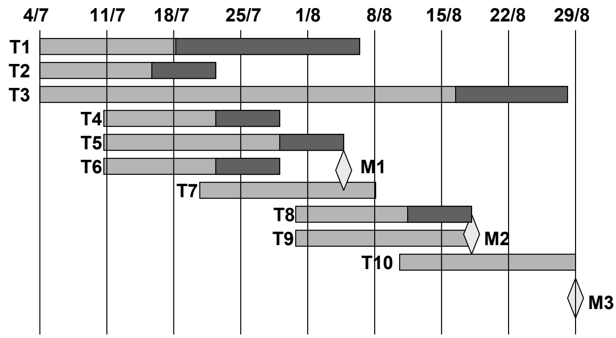
- Proposed by UML authors
- Phases (any of which may iterate)
  - Inception – capture business rationale and scope
  - Elaboration – domain model, architectural design, risk analysis, implementation planning
  - Construction – incremental implementation of use cases, iterative code change, refactoring
  - Transition – final touches, including optimisation
- Any may vary in degree of ceremony (documentation, contracts, sign-off etc.)

Universal project management

- Manager deals with human consequences of intrinsic complexity by:
  - Planning: estimation, identifying risk
  - Monitoring: progress & tolerance for “slip”
  - Controlling: effort distribution & scheduling
  - Motivating: may be based on technical respect from staff, but managerial competence essential
- Management tools:
  - PERT (program evaluation and review technique)
  - CPM (critical path method)
  - Software implementing these (e.g. MS Project)

Activity Charts

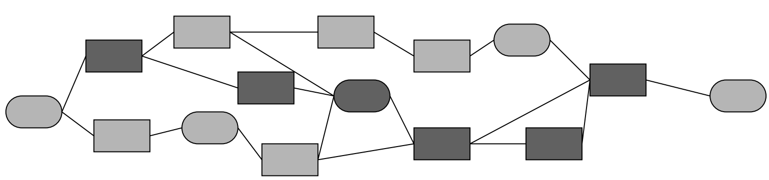
- Show a project's tasks and milestones (with allowable variation)



- Problem: relatively hard to visualise interdependencies and knock-on effects of any milestone being late.

Critical Path Analysis

- Drawing activity chart as graph with dependencies makes critical path easier to find and monitor



- PERT charts include bad / expected / good durations
- warn of trouble in time to take actions
- mechanical approach not enough
  - overestimates of duration come down steadily
  - underestimates usually covered up until near deadline!
- management heuristic
  - the project manager is never on the critical path

Documentation

- Projects have various management documents:
  - contracts - budgets - activity charts & graphs - staff schedules
- Plus various engineering documents:
  - requirements - hazard analysis - specification - test plan - code
- How do we keep all these in step?
  - Computer science tells us it's hard to keep independent files in synch
- Possible solutions
  - high tech: CASE tool
  - bureaucratic: plans and controls dept
  - convention: self documenting code

Alternative philosophies

- Some programmers are very much more productive than others - by a factor of ten or more
- ‘Chief programmer teams’, developed at IBM (1970-72) seek to capitalise on this
  - team with one chief programmer + apprentice/assistant,
  - plus toolsmith, librarian,admin assistant, etc
  - get the maximum productivity from the available talent
- Can be very effective during the implementation stage of a project
  - However, each team can only do so much
  - Complementary to (rather than opposed to) waterfall/spiral and other project management methodologies

More alternative philosophies

- 'Egoless programming'
  - code owned by team, not by individual (Weinberg, 1971).
  - in direct opposition to the 'chief programmer' idea.
- 'Xtreme Programming' (XP)
  - small groups work together for fast development cycle iteration, early exposure to users. (Beck 199x)
- 'Literate programming'
  - code as a work of art, designed not just for machine but for human readers / maintainers (Knuth et al)
- Objections:
  - can lead to wrong design decisions becoming entrenched, defended, propagated more passionately
  - 'creeping elegance' may be symptom of project out of control
- There is no silver bullet!

ISO 9000 quality standards

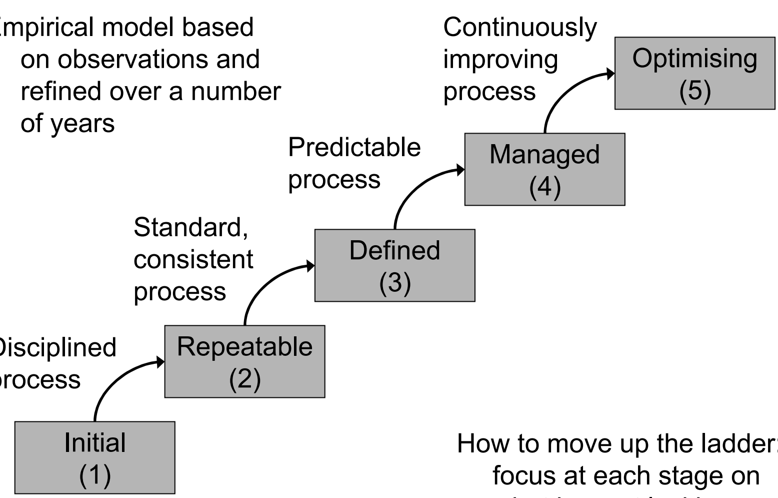
- Not software specific
- Requires a quality manual: documented quality system
- Design control: requirements documentation and functional traceability
- Inspection (review): plans and status
- Test status: what tests will be done, which have been conducted, which successful
- Handling: library, backup and configuration

Capability Maturity Model

- Emphasises shift from 'product' to 'process'
- A good team isn't permanent
  - need repeatable, manageable performance
  - not outcome dependent on individual genius or heroics
- Capability Maturity Model (CMM)
  - ‘market leading’ approach to this problem
  - developed at CMU with DoD funding
  - identifies five levels of increasing maturity in a software team or organisation
  - provides a guide to moving up from one level to the next

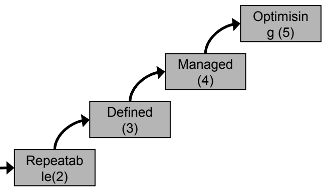
Levels of CMM

Empirical model based on observations and refined over a number of years



How to move up the ladder: focus at each stage on what is most lacking

Levels of CMM

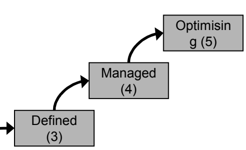


Initial (1)

Projects are chaotic

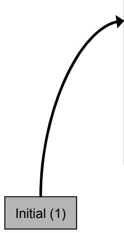
Success depends on luck and heroism

Levels of CMM



Repeatable(2)

Software configuration management
Software quality assurance
Software subcontract management
Software project tracking and oversight
Software project planning
Requirements management



Levels of CMM

Optimising (5)

Managed (4)

Defined (3)

Peer reviews
Intergroup coordination
Software product engineering
Integrated software management
Training programme
Organisation process definition
Organisation process focus

Repeatable(2)

Initial (1)

Levels of CMM

Optimising (5)

Managed (4)

Software quality management

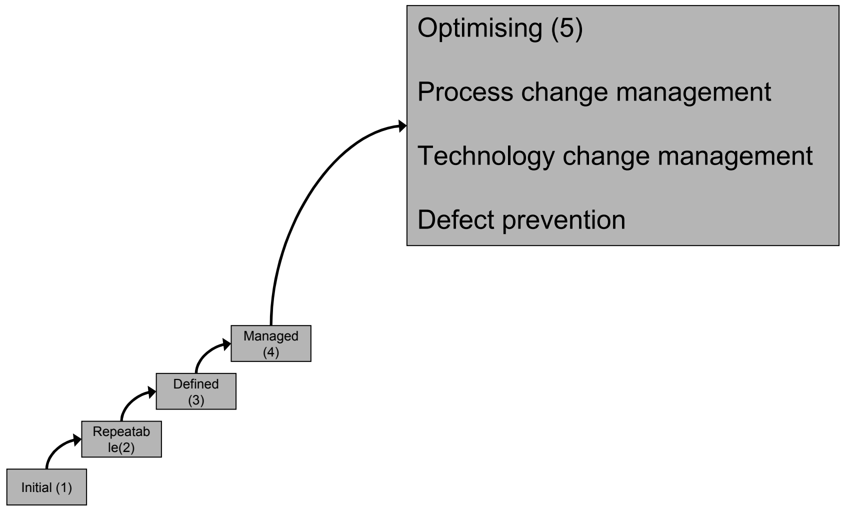
Quantitative process management

Defined (3)

Repeatable(2)

Initial (1)

Levels of CMM



CONCLUSIONS

- Software engineering and design are hard
  - Completely generic tools meet very specific tasks
  - Must engage with human needs in social context
  - Fundamentally about managing complexity
- Craft skills of software construction
  - Decomposition and modular construction
  - Modelling tools that enable analysis and design
- User centred design: knowledge & attitude
  - Broad understanding of human and social sciences
  - Protect user needs in corporate/technical environment
- Systematic management
  - Awareness of lifecycle model and suitable tools
  - Measuring and reflecting on process improvement