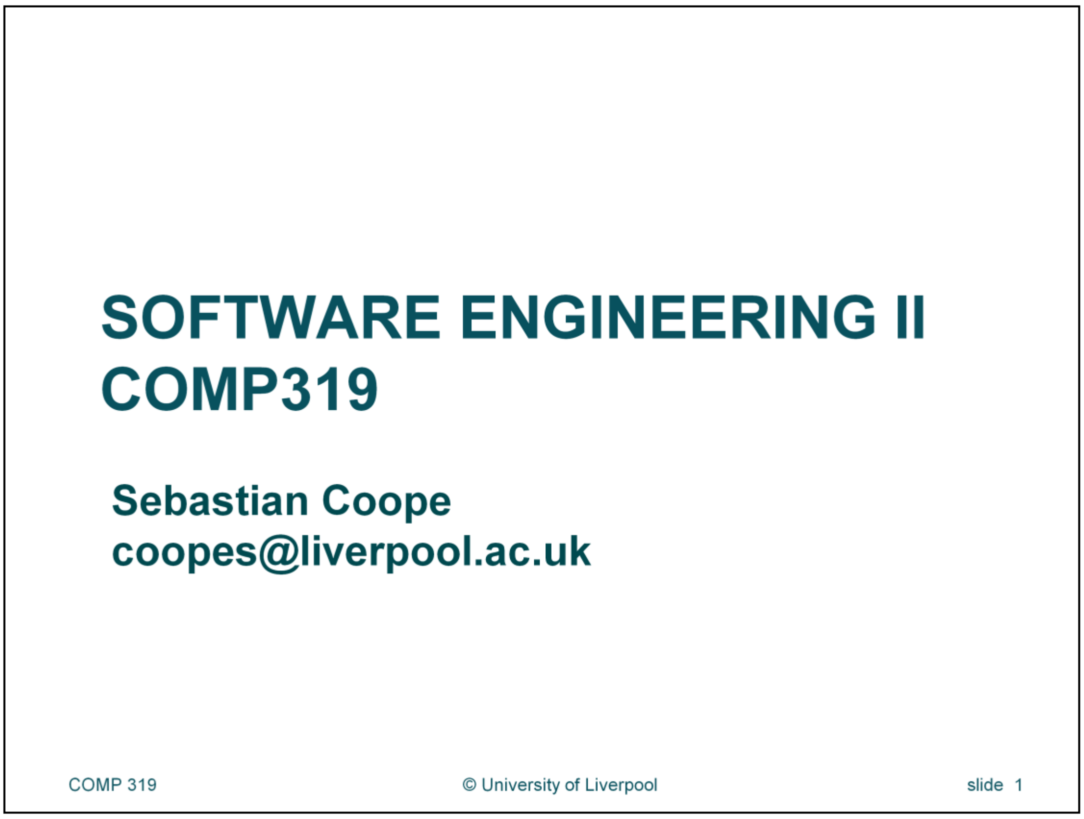
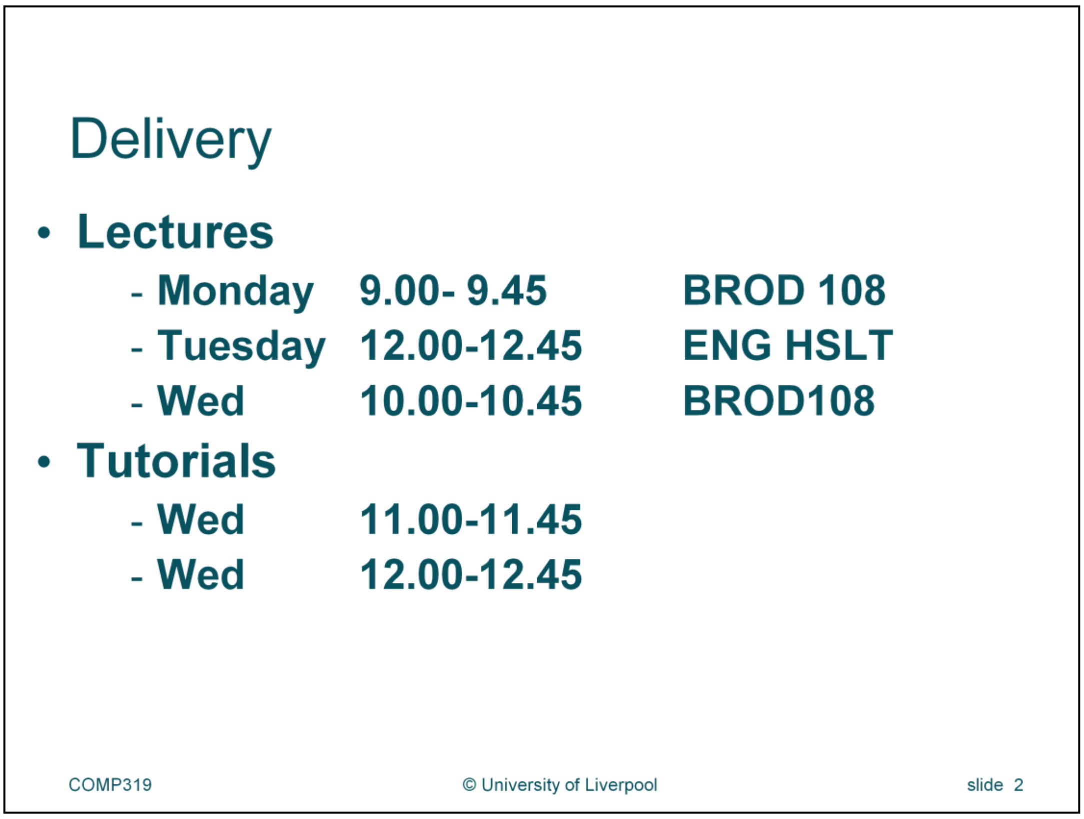


SOFTWARE ENGINEERING II COMP319

Sebastian Coope
coopes@liverpool.ac.uk



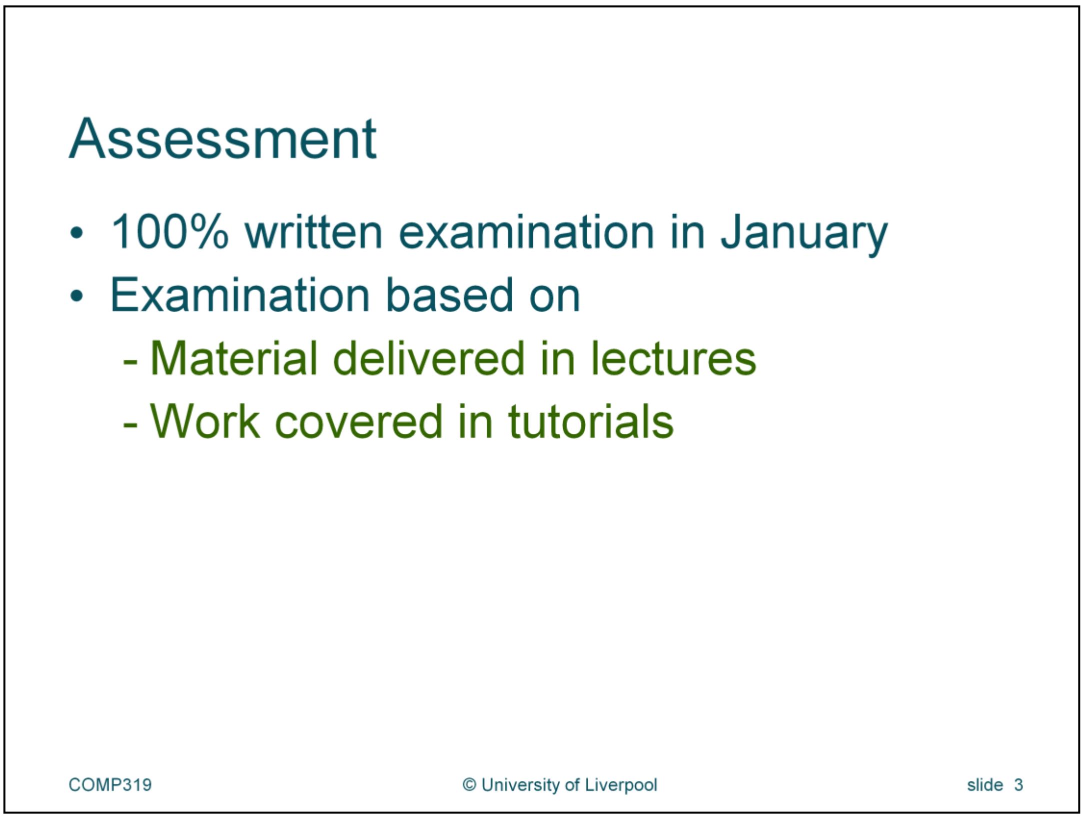
Delivery

- **Lectures**

- Monday	9.00- 9.45	BROD 108
- Tuesday	12.00-12.45	ENG HSLT
- Wed	10.00-10.45	BROD108

- **Tutorials**

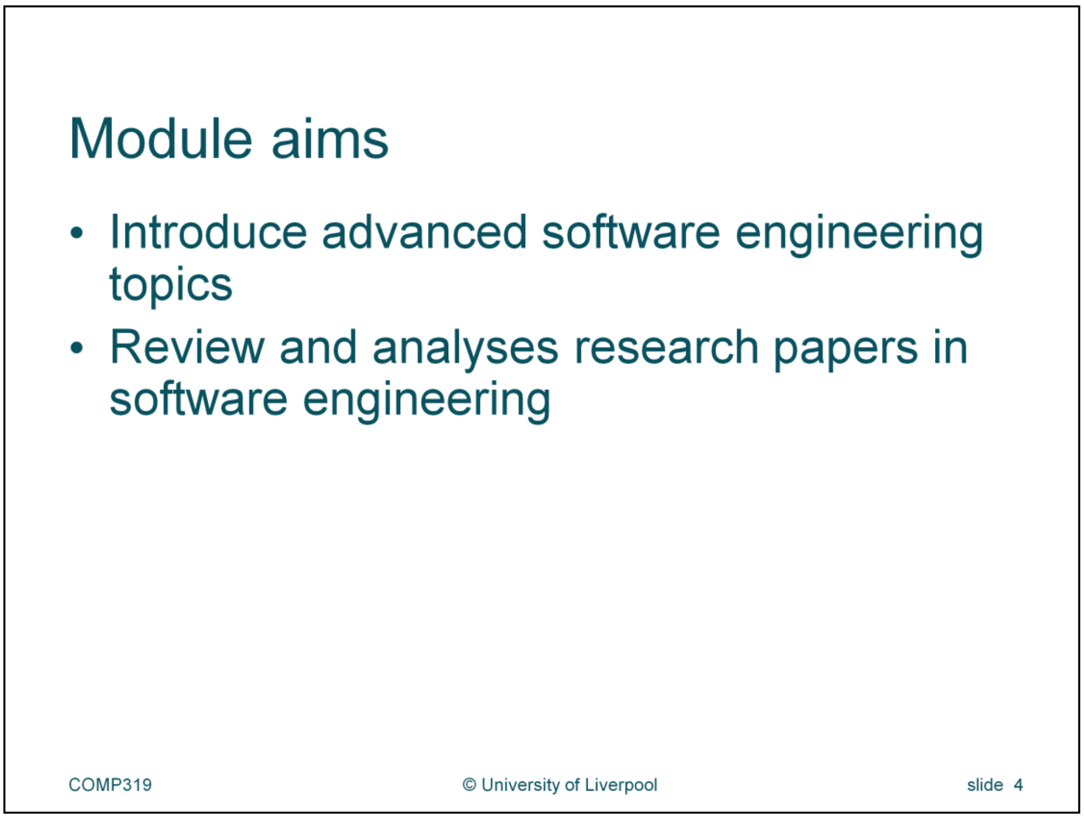
- Wed	11.00-11.45
- Wed	12.00-12.45



Assessment

- 100% written examination in January
- Examination based on
  - Material delivered in lectures
  - Work covered in tutorials

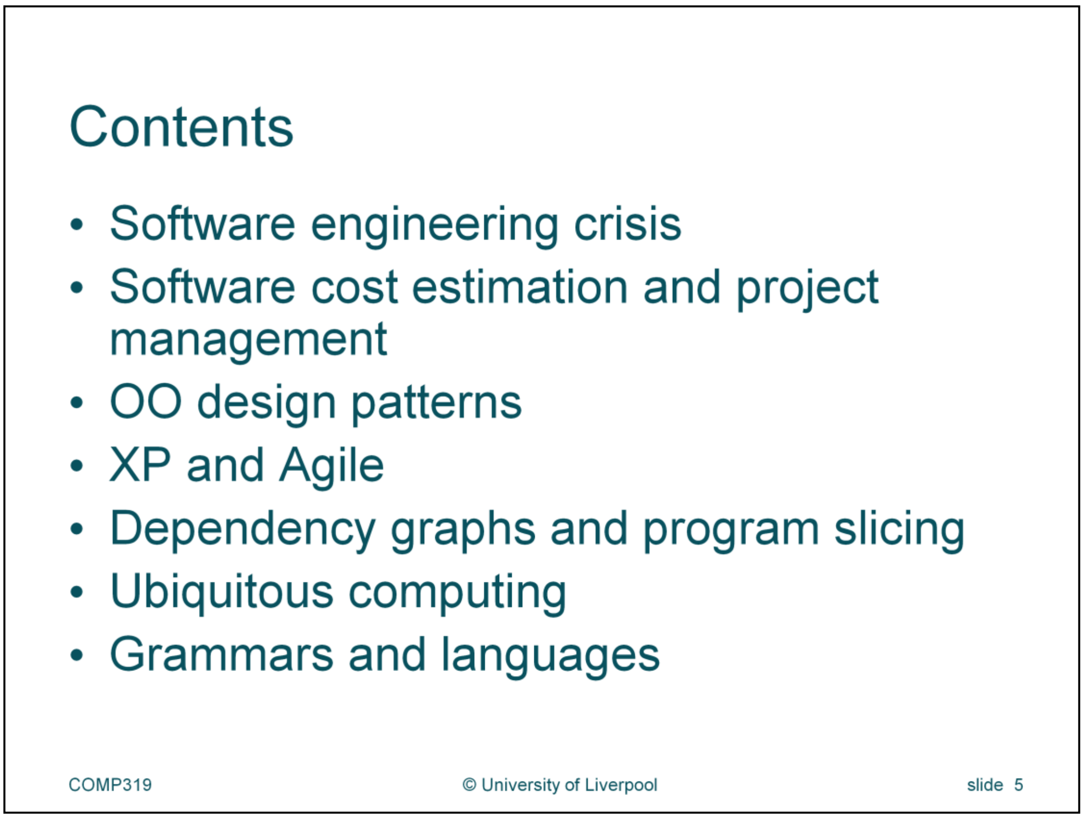
COMP319 © University of Liverpool slide 3



Module aims

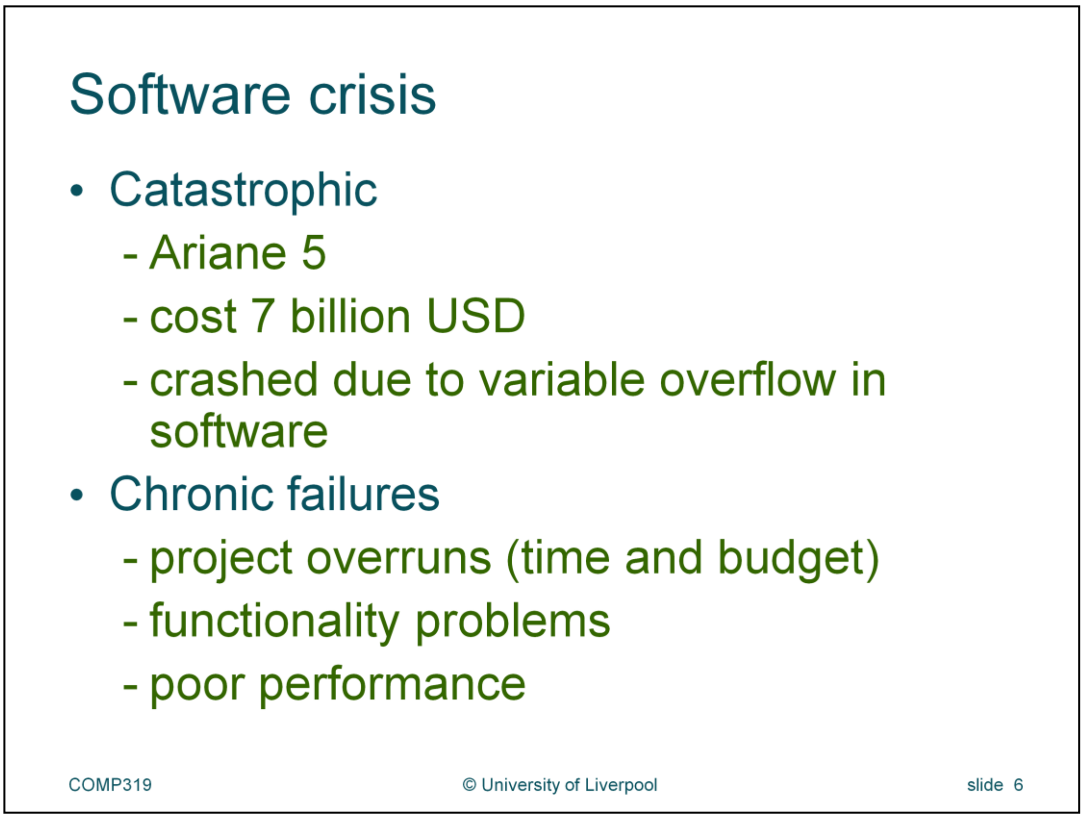
- Introduce advanced software engineering topics
- Review and analyses research papers in software engineering

COMP319 © University of Liverpool slide 4



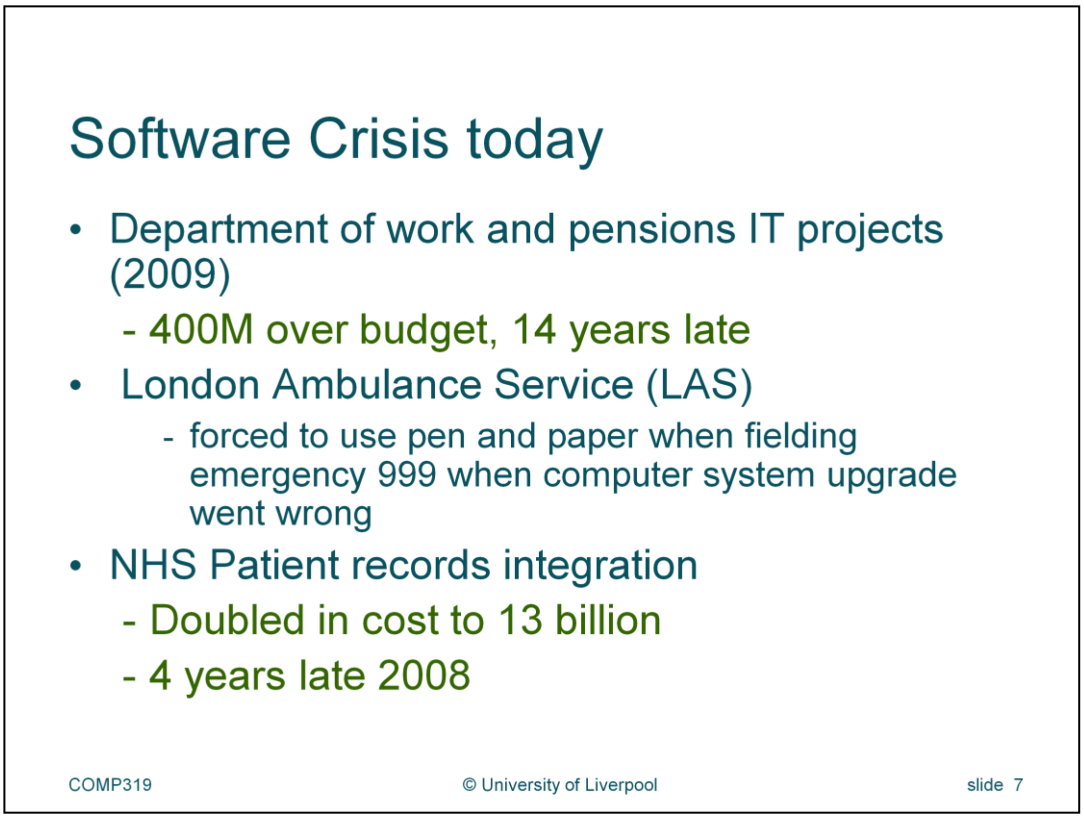
Contents

- Software engineering crisis
- Software cost estimation and project management
- OO design patterns
- XP and Agile
- Dependency graphs and program slicing
- Ubiquitous computing
- Grammars and languages



Software crisis

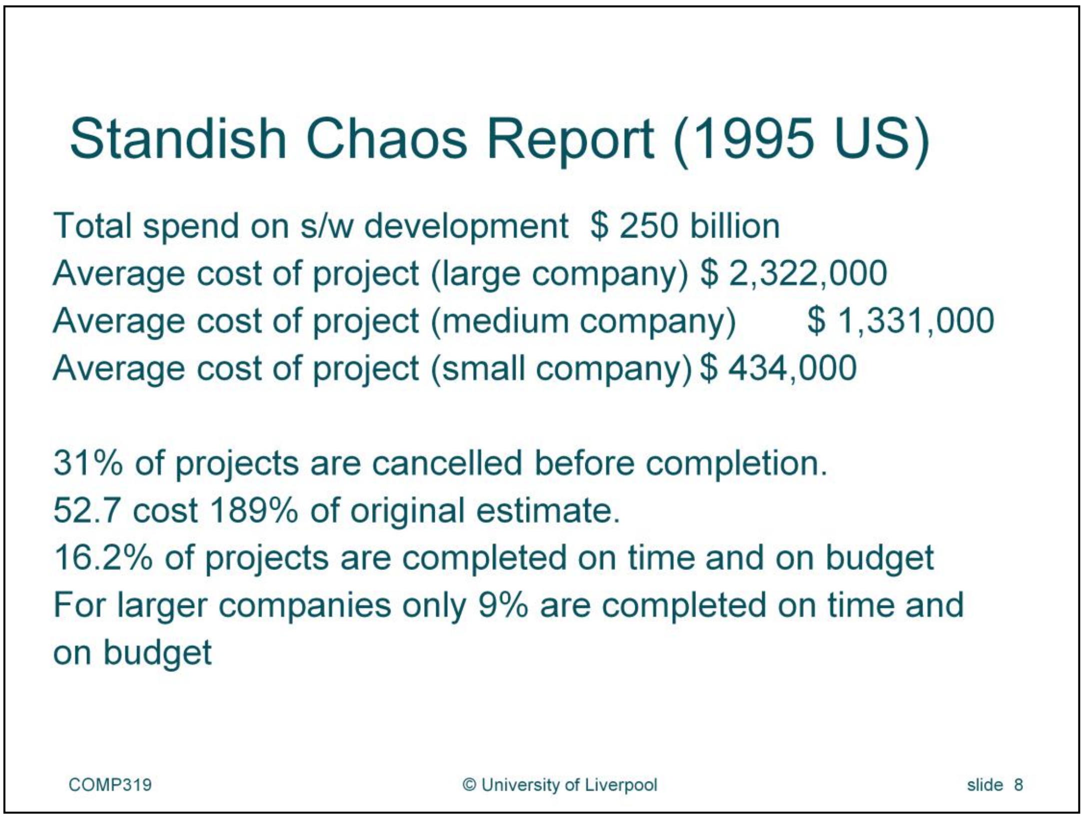
- Catastrophic
  - Ariane 5
  - cost 7 billion USD
  - crashed due to variable overflow in software
- Chronic failures
  - project overruns (time and budget)
  - functionality problems
  - poor performance



Software Crisis today

- Department of work and pensions IT projects (2009)
  - 400M over budget, 14 years late
- London Ambulance Service (LAS)
  - forced to use pen and paper when fielding emergency 999 when computer system upgrade went wrong
- NHS Patient records integration
  - Doubled in cost to 13 billion
  - 4 years late 2008

COMP319 © University of Liverpool slide 7

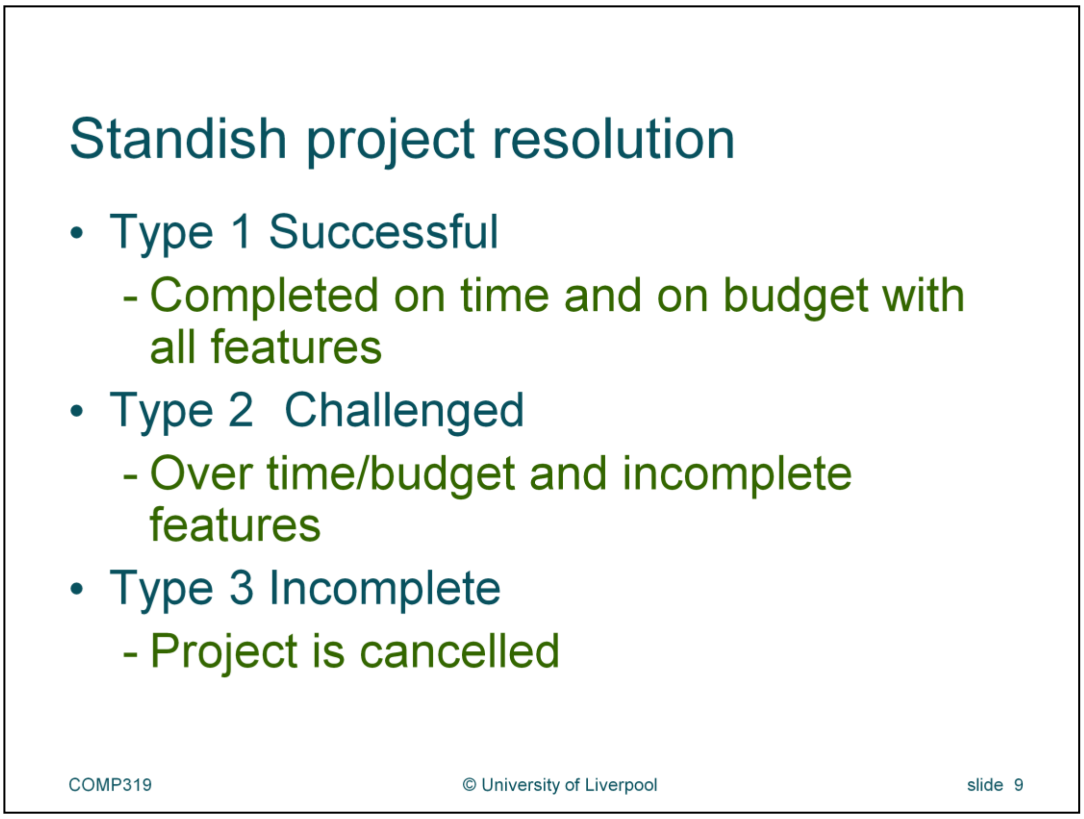


Standish Chaos Report (1995 US)

Total spend on s/w development $ 250 billion
Average cost of project (large company) $ 2,322,000
Average cost of project (medium company) $ 1,331,000
Average cost of project (small company) $ 434,000

31% of projects are cancelled before completion.
52.7 cost 189% of original estimate.
16.2% of projects are completed on time and on budget
For larger companies only 9% are completed on time and on budget

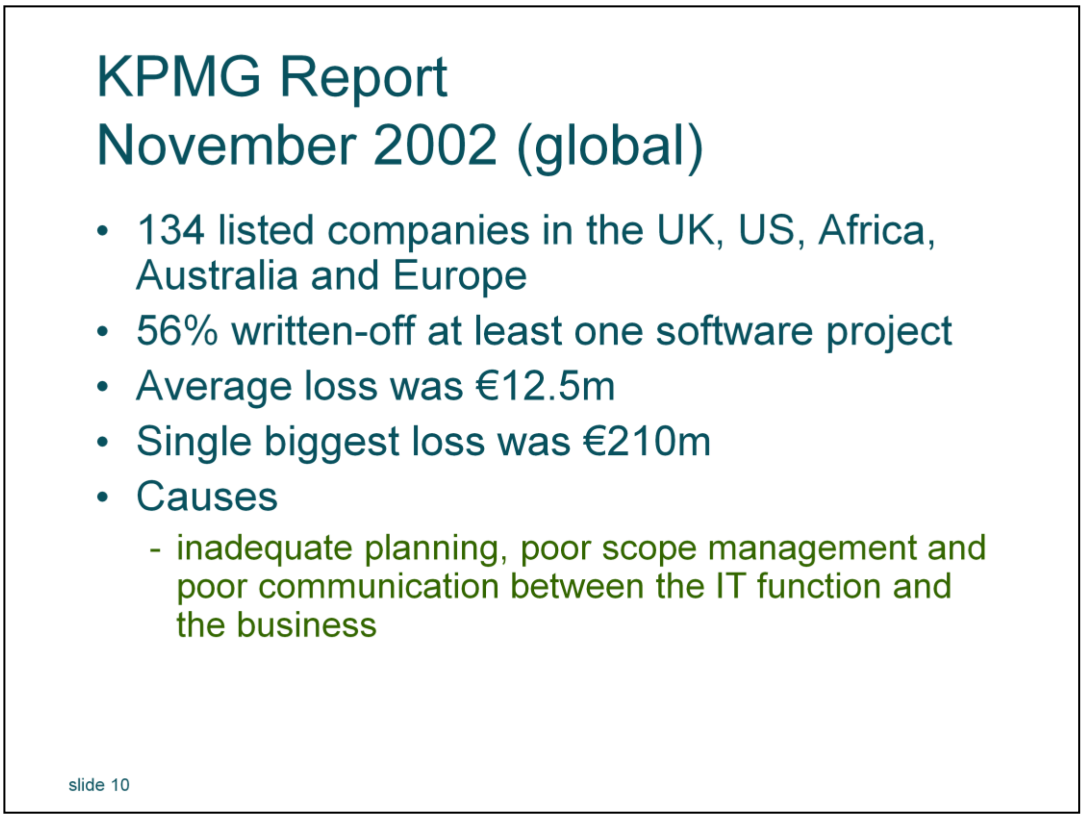
COMP319 © University of Liverpool slide 8



Standish project resolution

- Type 1 Successful
  - Completed on time and on budget with all features
- Type 2 Challenged
  - Over time/budget and incomplete features
- Type 3 Incomplete
  - Project is cancelled

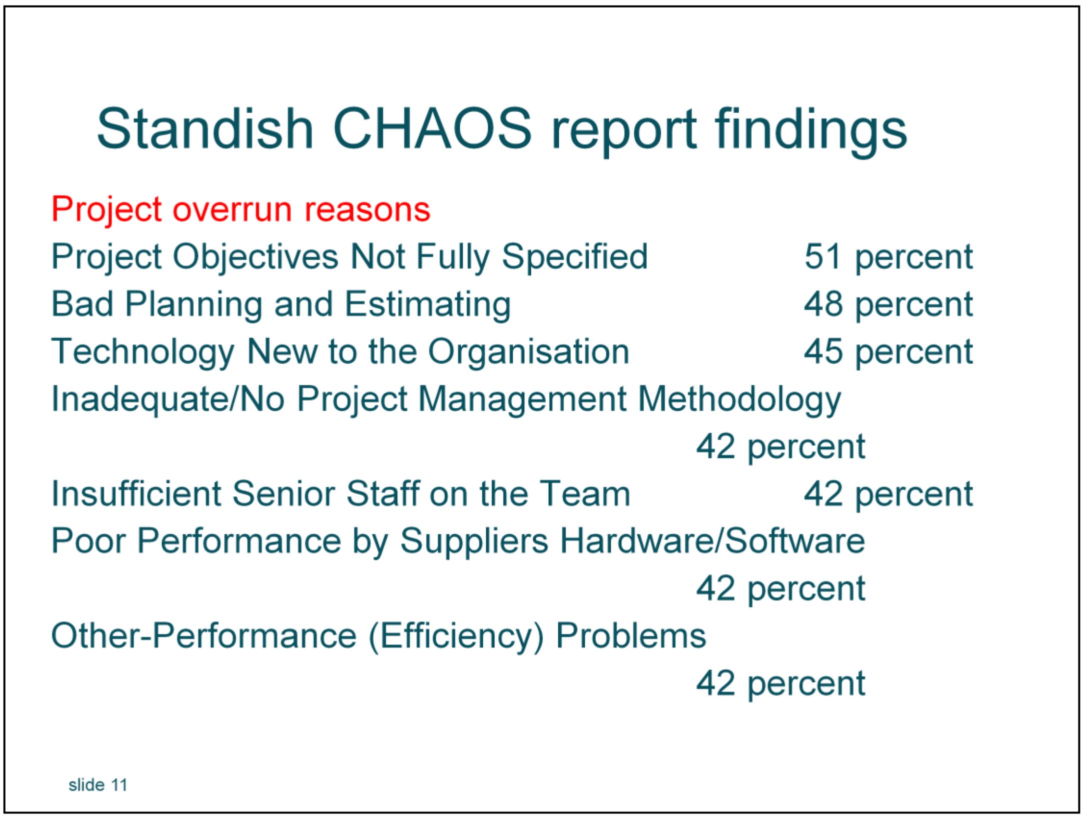
COMP319 © University of Liverpool slide 9



KPMG Report
November 2002 (global)

- 134 listed companies in the UK, US, Africa, Australia and Europe
- 56% written-off at least one software project
- Average loss was €12.5m
- Single biggest loss was €210m
- Causes
  - inadequate planning, poor scope management and poor communication between the IT function and the business

slide 10

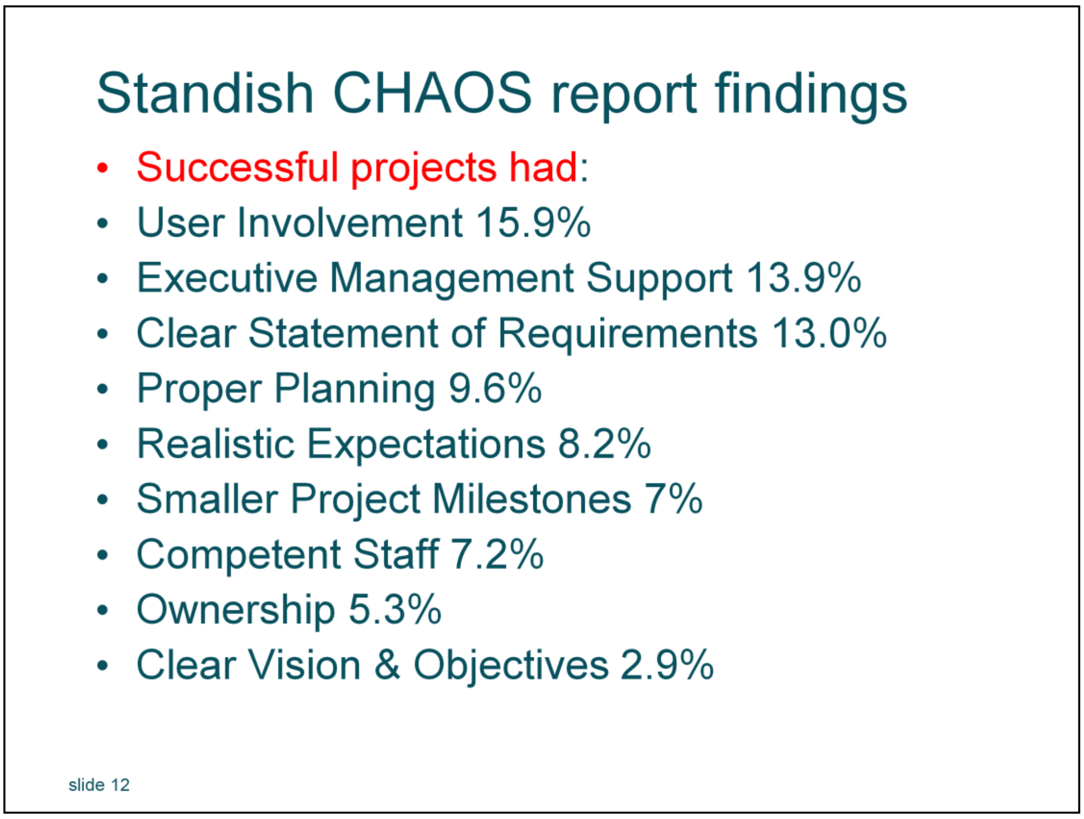


Standish CHAOS report findings

Project overrun reasons

Project Objectives Not Fully Specified	51 percent
Bad Planning and Estimating	48 percent
Technology New to the Organisation	45 percent
Inadequate/No Project Management Methodology	42 percent
Insufficient Senior Staff on the Team	42 percent
Poor Performance by Suppliers Hardware/Software	42 percent
Other-Performance (Efficiency) Problems	42 percent

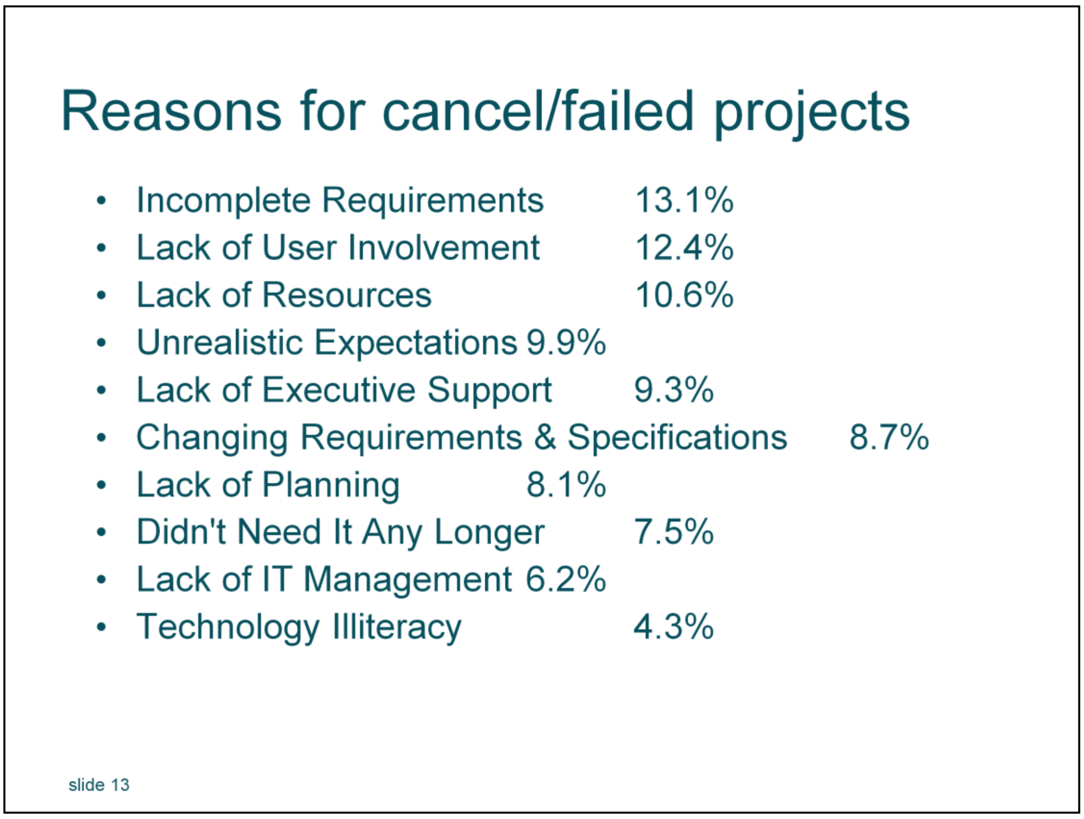
slide 11



Standish CHAOS report findings

- Successful projects had:
- User Involvement 15.9%
- Executive Management Support 13.9%
- Clear Statement of Requirements 13.0%
- Proper Planning 9.6%
- Realistic Expectations 8.2%
- Smaller Project Milestones 7%
- Competent Staff 7.2%
- Ownership 5.3%
- Clear Vision & Objectives 2.9%

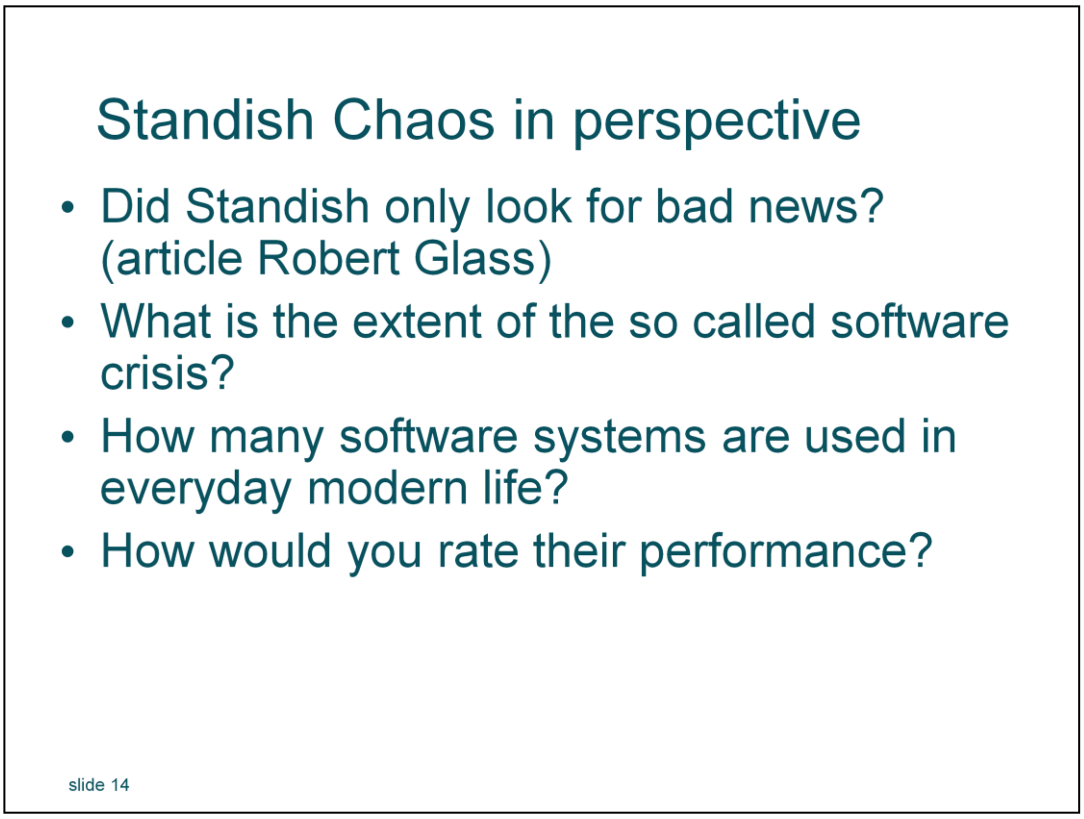
slide 12



Reasons for cancel/failed projects

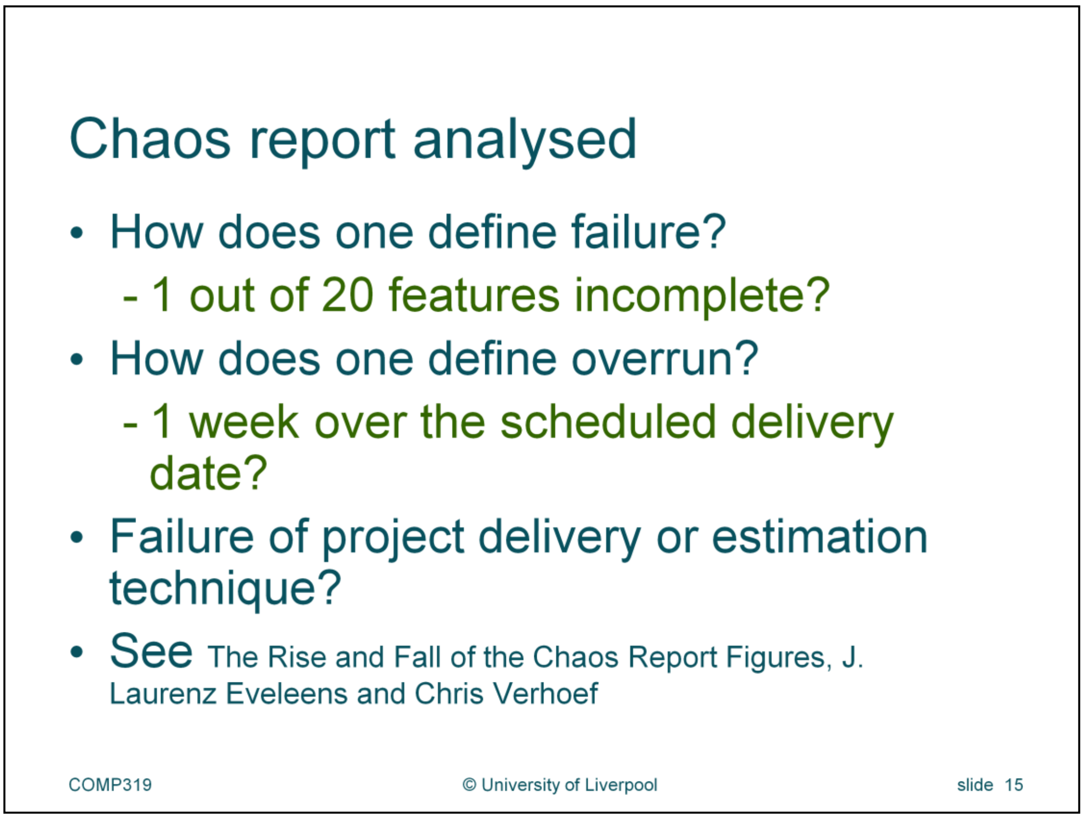
Incomplete Requirements	13.1%
Lack of User Involvement	12.4%
Lack of Resources	10.6%
Unrealistic Expectations	9.9%
Lack of Executive Support	9.3%
Changing Requirements & Specifications	8.7%
Lack of Planning	8.1%
Didn't Need It Any Longer	7.5%
Lack of IT Management	6.2%
Technology Illiteracy	4.3%

slide 13


Standish Chaos in perspective

- Did Standish only look for bad news? (article Robert Glass)
- What is the extent of the so called software crisis?
- How many software systems are used in everyday modern life?
- How would you rate their performance?

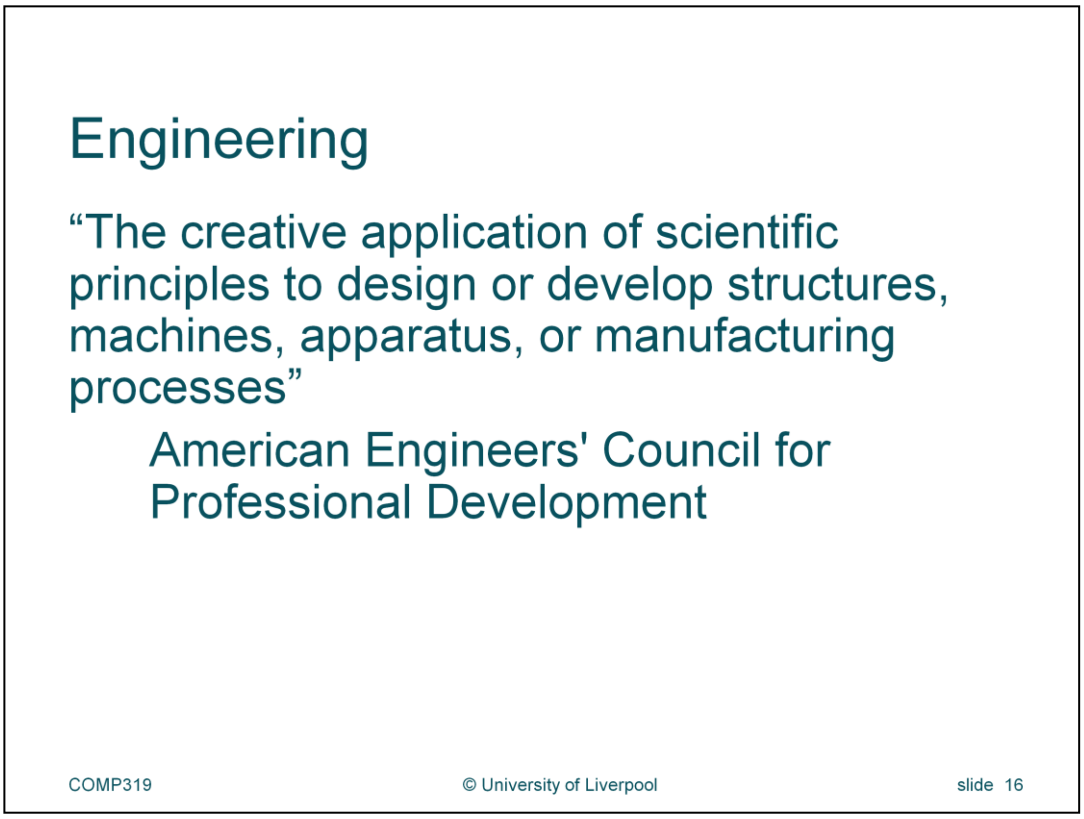
slide 14



Chaos report analysed

- How does one define failure?
  - 1 out of 20 features incomplete?
- How does one define overrun?
  - 1 week over the scheduled delivery date?
- Failure of project delivery or estimation technique?
- See The Rise and Fall of the Chaos Report Figures, J. Laurenz Eveleens and Chris Verhoef

COMP319 © University of Liverpool slide 15

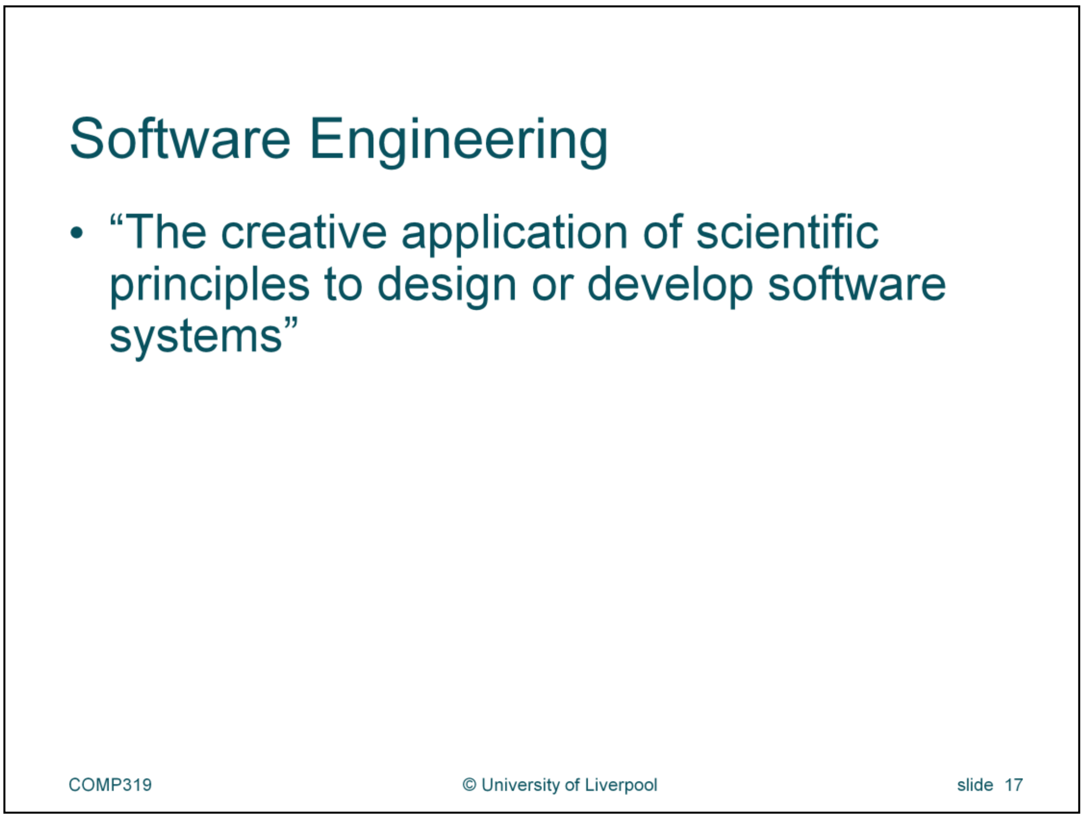


Engineering

"The creative application of scientific principles to design or develop structures, machines, apparatus, or manufacturing processes"

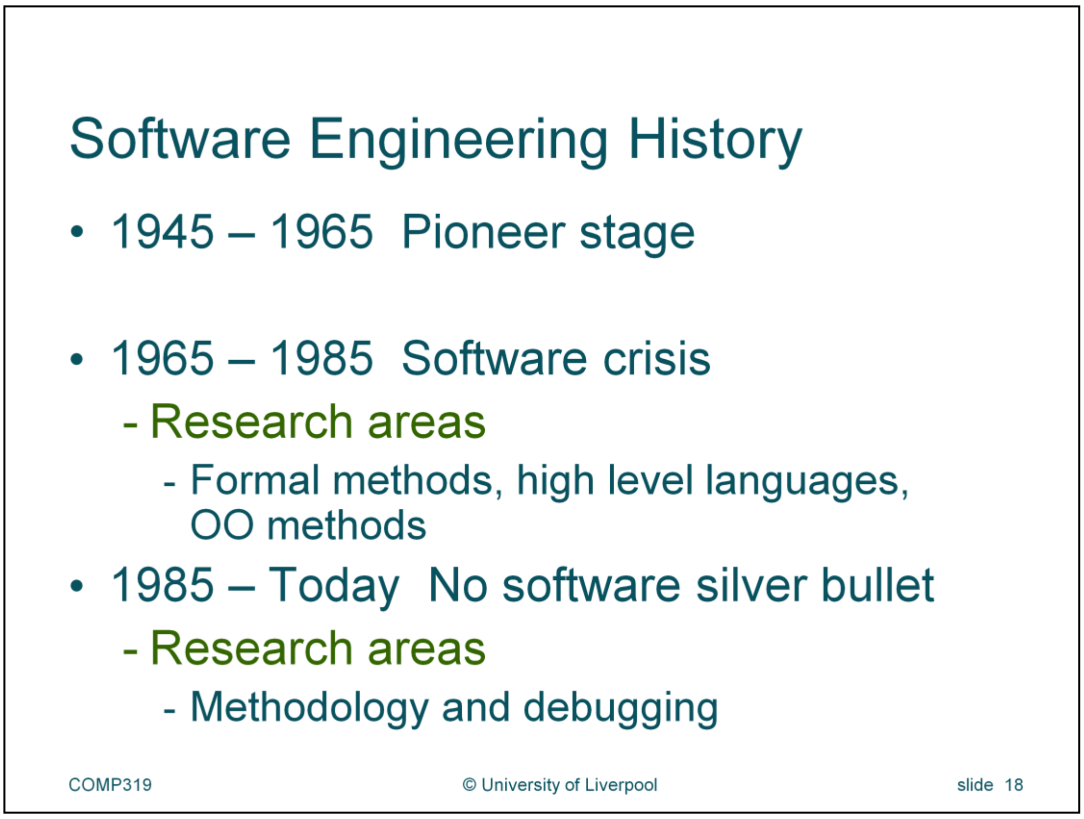
American Engineers' Council for Professional Development

COMP319 © University of Liverpool slide 16


Software Engineering

- “The creative application of scientific principles to design or develop software systems”

COMP319 © University of Liverpool slide 17


Software Engineering History

- 1945 – 1965 Pioneer stage
- 1965 – 1985 Software crisis
  - Research areas
    - Formal methods, high level languages, OO methods
- 1985 – Today No software silver bullet
  - Research areas
    - Methodology and debugging

COMP319 © University of Liverpool slide 18

The broad history of the subject falls into three 20 year periods.

The term “software” is credited to John W. Tukey a Princeton statistician (responsible among other things for several fast fourier analysis algorithms) who coined it in 1957 to describe the activity in a statistical analysis of the growth of various parts of the computer industry.

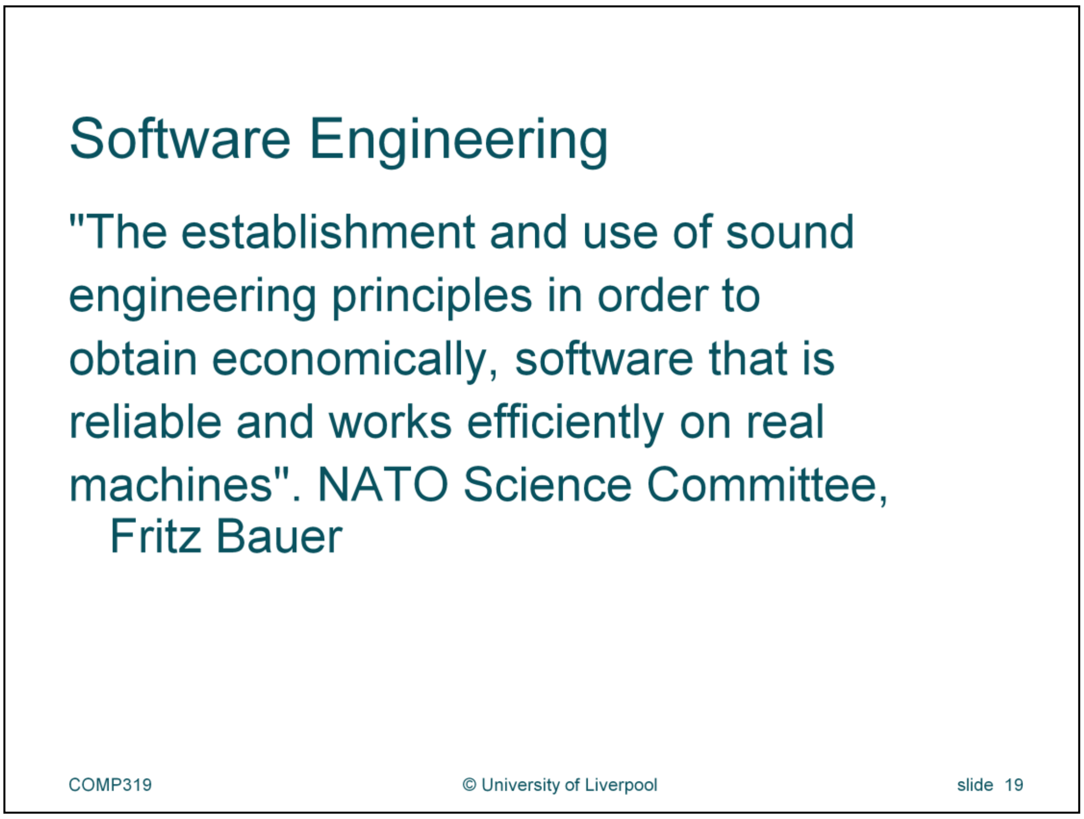
The pioneer stage sees the development of the ideas, the languages, and systems – essentially for *operating* computers. There were few applications, usually bespoke.

Towards the end of the pioneer stage it became apparent that software development was complex costly, and time consuming – and that hardware development was happening much faster than software development. This problem was termed the “software crisis” and described as a “monster” or “werewolf” that needed to be killed, and quickly.

Solutions to these problems (silver bullets) were sought with which to kill the monster and this was, and continues to be, one of the key preoccupations of research in the subject.

However, in 1985 a paper was published that suggested that there were no silver bullets; with that the subject moved into the current phase – the “no software bullets” stage ….

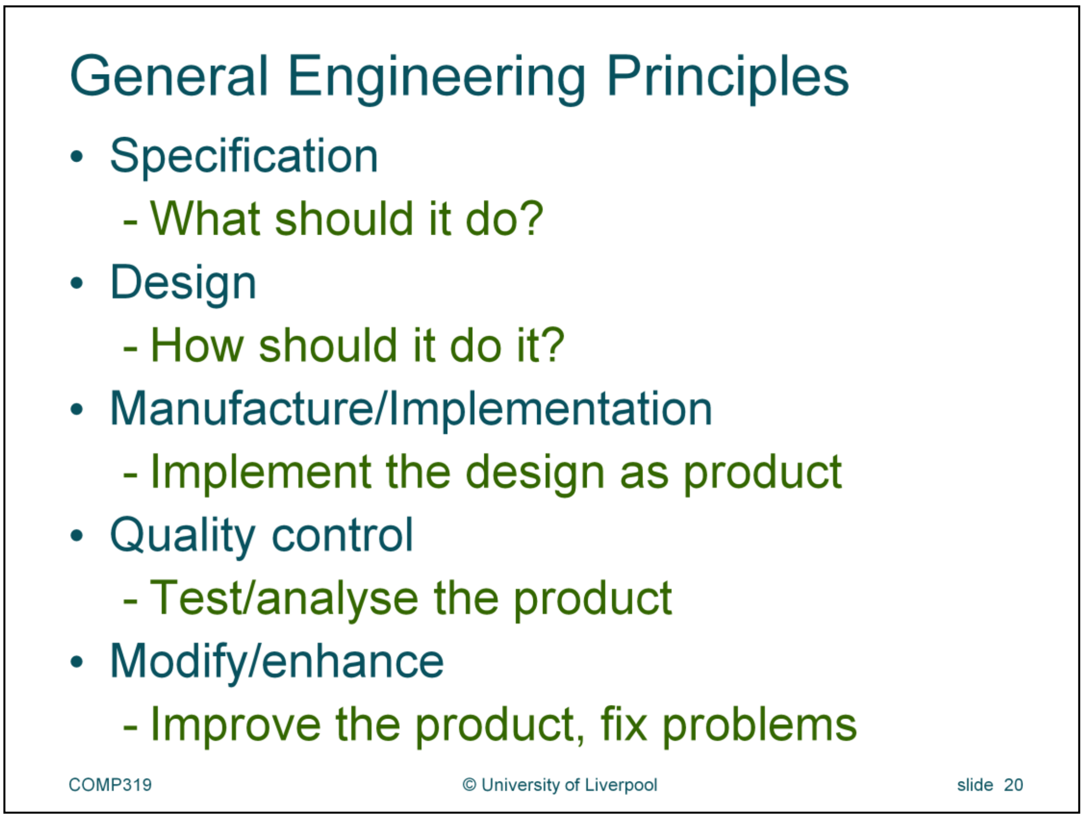
Which is where we are today.



Software Engineering

"The establishment and use of sound engineering principles in order to obtain economically, software that is reliable and works efficiently on real machines". NATO Science Committee, Fritz Bauer

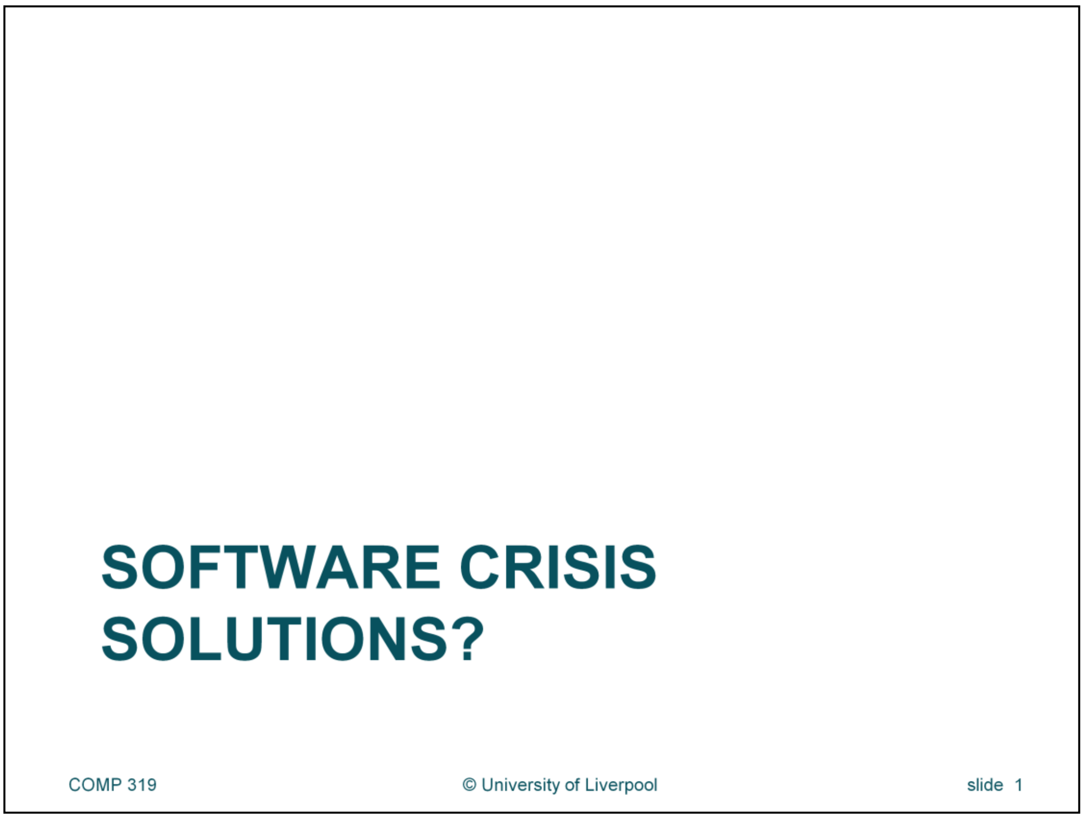
COMP319 © University of Liverpool slide 19



General Engineering Principles

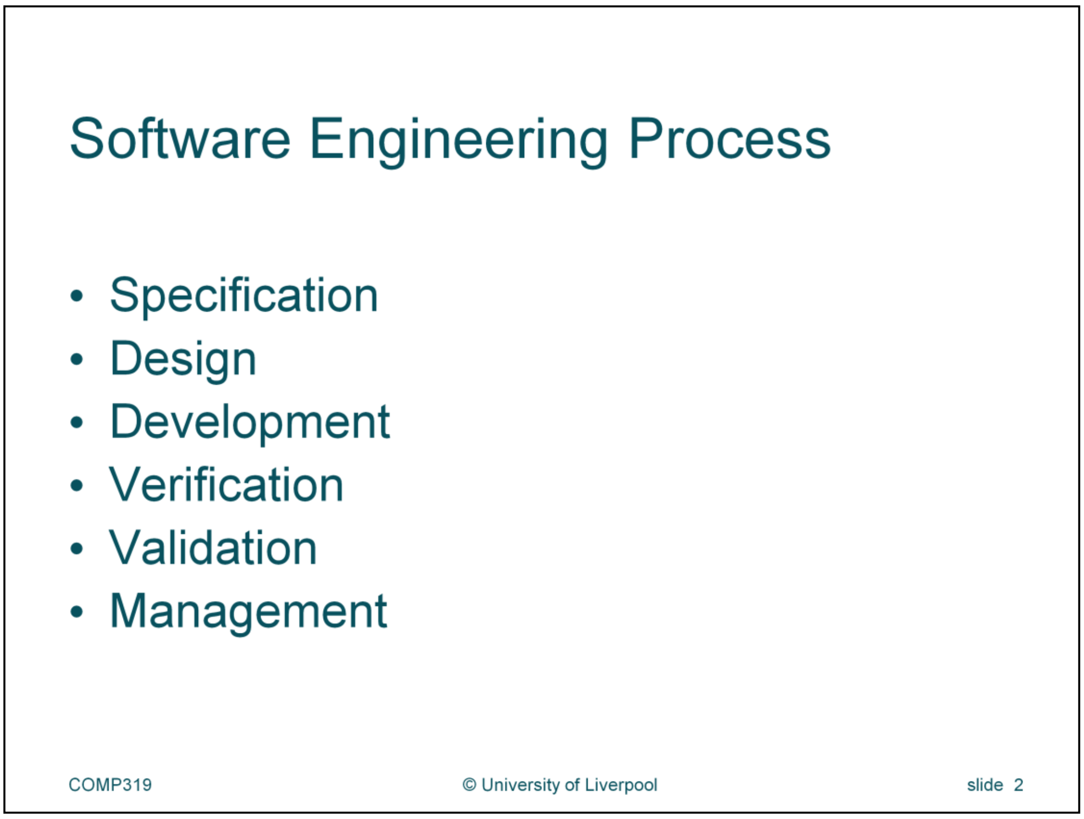
- Specification
  - What should it do?
- Design
  - How should it do it?
- Manufacture/Implementation
  - Implement the design as product
- Quality control
  - Test/analyse the product
- Modify/enhance
  - Improve the product, fix problems

COMP319 © University of Liverpool slide 20



SOFTWARE CRISIS SOLUTIONS?

COMP 319 © University of Liverpool slide 1



Software Engineering Process

- Specification
- Design
- Development
- Verification
- Validation
- Management

COMP319 © University of Liverpool slide 2

What is the essence of software engineering?

Sommerville ("Software Engineering", 7th ed., 2004, Pearson Education Ltd) summarises on page 2 what the essential software processes are. He notes in particular that product and people are involved and that the product is not just "ware", but must include people and organisation in the "socio-technical" system that is generated.

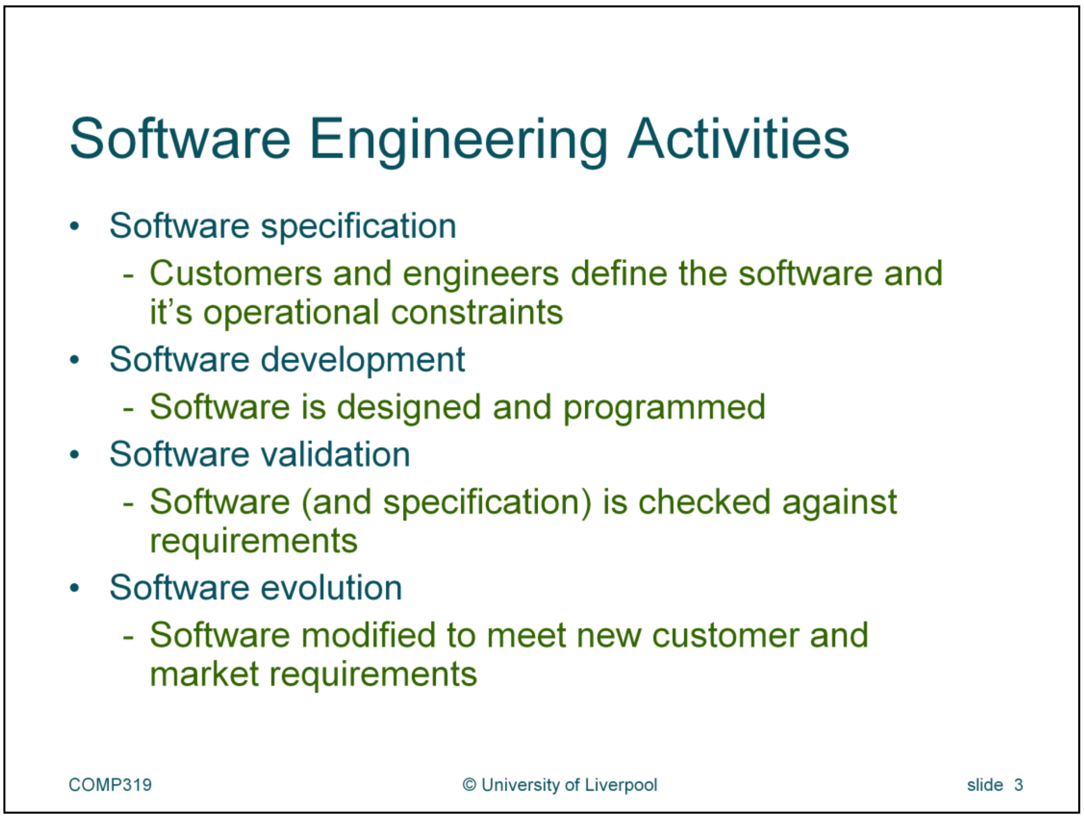
In this module we are not going to look at this in any details … you should have covered it in earlier modules. If you have not then at least read the book, or read it again.

In this course we will look at research into these processes as well as the process as a whole, "**the methods**" of software engineering.

These methods while in practice no more than "rules of thumb" do seem to form the "holy grail" of software engineering, although what they do is to wrap up the software processes above into one named (and sold) process.

Examples of such software engineering methods are: Structured Analysis (DeMarco, 1978); JSD (Jackson, 1983); OO methods (Booch, 1994; Rumbaugh et al, 1991).

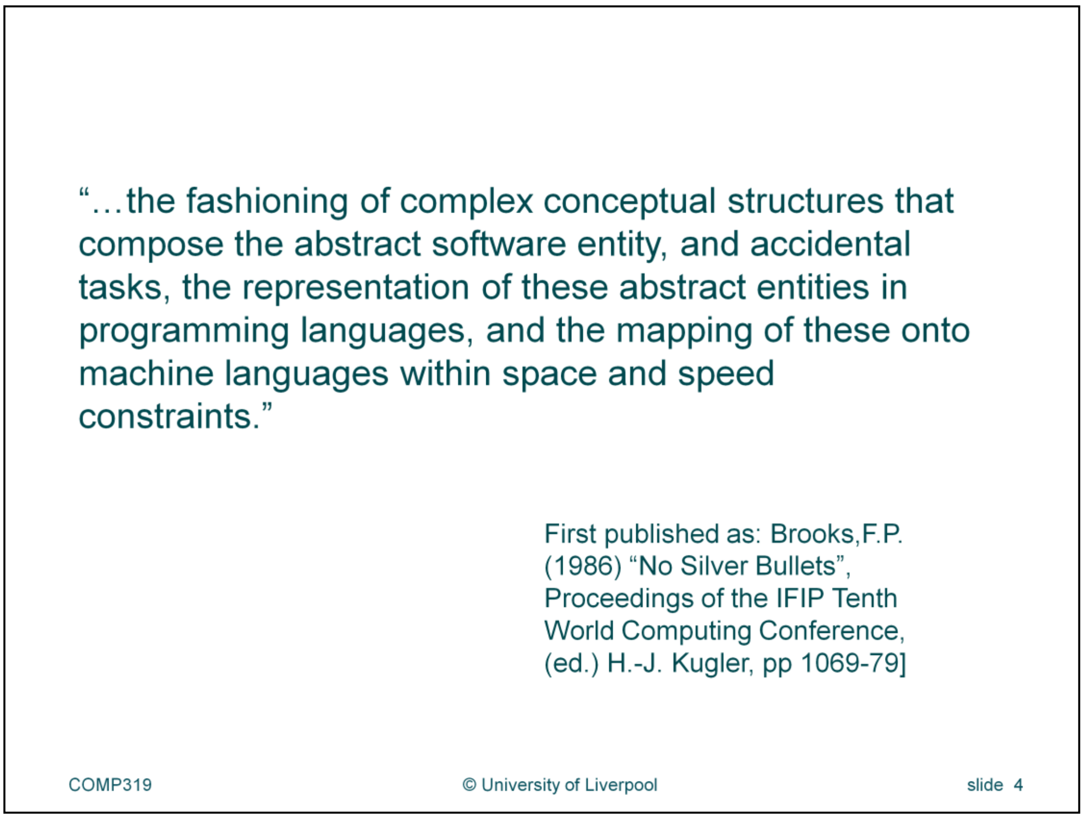
These have been integrated together in the Millennium development "UML" (Booch, 1999; Rumbaugh et al, 1999, etc) which we will look at more critically, later.



Software Engineering Activities

- Software specification
  - Customers and engineers define the software and it's operational constraints
- Software development
  - Software is designed and programmed
- Software validation
  - Software (and specification) is checked against requirements
- Software evolution
  - Software modified to meet new customer and market requirements

COMP319 © University of Liverpool slide 3



"…the fashioning of complex conceptual structures that compose the abstract software entity, and accidental tasks, the representation of these abstract entities in programming languages, and the mapping of these onto machine languages within space and speed constraints."

First published as: Brooks,F.P. (1986) "No Silver Bullets", Proceedings of the IFIP Tenth World Computing Conference, (ed.) H.-J. Kugler, pp 1069-79]

COMP319 © University of Liverpool slide 4

The complexity is of two kinds and arises because of the fundamental difficulty associated with " … the fashioning …

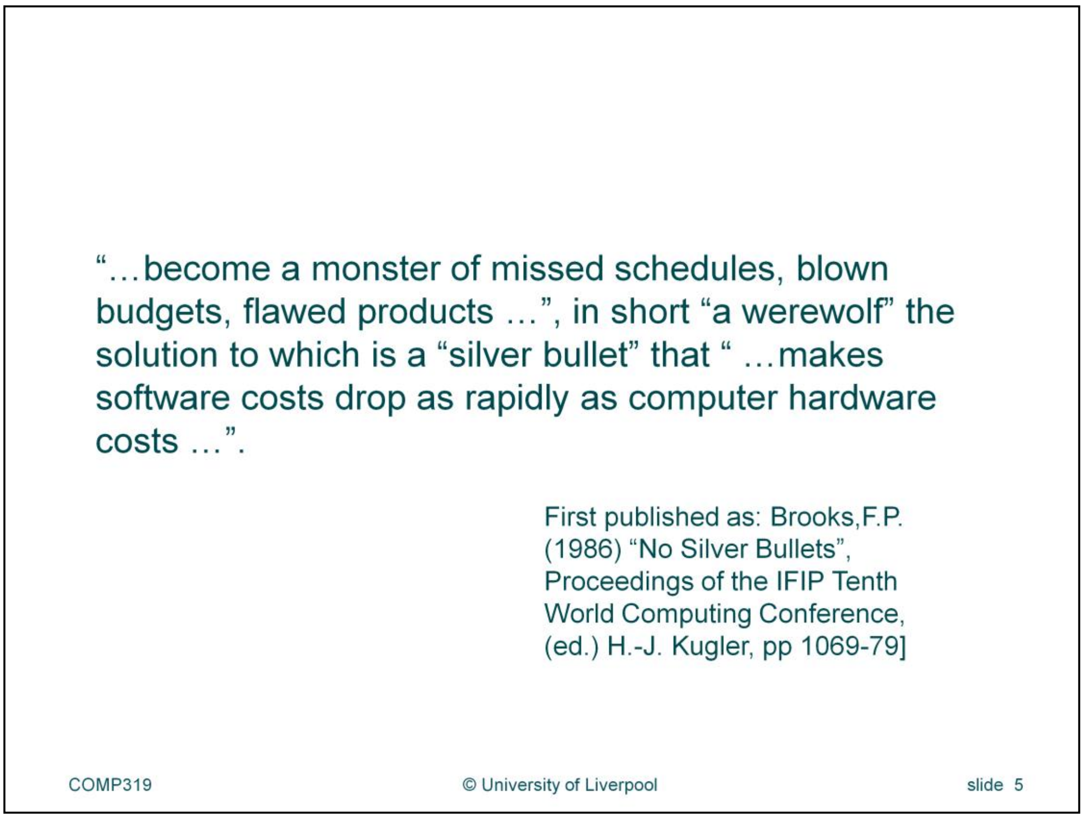
The conceptual structure is the first goal of software engineering.

It is important to be clear what is meant.

The conceptual structure captures the essence of the software and is associated with all software products.

When the conceptual structure is implemented accidental tasks arise that must be solved e.g. coding, bugs, data generation, etc. All these are additional complexity to the complexity of the conceptual structure. They are always going to be present and in this sense are unavoidable, and the aim will be to try and predict them and to minimise their impact.

Both activities are complex and because of that they are hard.


"…become a monster of missed schedules, blown budgets, flawed products …", in short "a werewolf" the solution to which is a "silver bullet" that " …makes software costs drop as rapidly as computer hardware costs …".

First published as: Brooks,F.P. (1986) "No Silver Bullets", Proceedings of the IFIP Tenth World Computing Conference, (ed.) H.-J. Kugler, pp 1069-79]

COMP319 © University of Liverpool slide 5

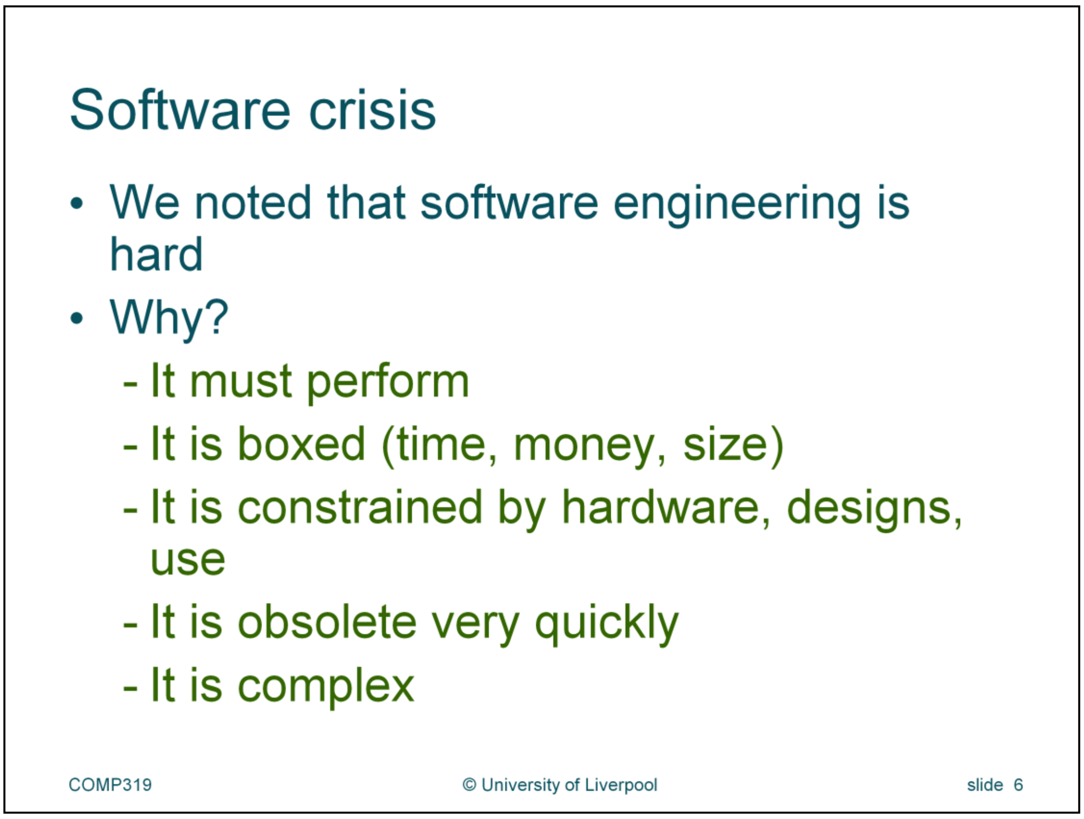
The crisis comes in getting to a product.

As Brooks put it getting the conceptual structure and then getting it as a product has "…become a monster …

The software engineering monster (colourfully here a werewolf) is thus software productivity.

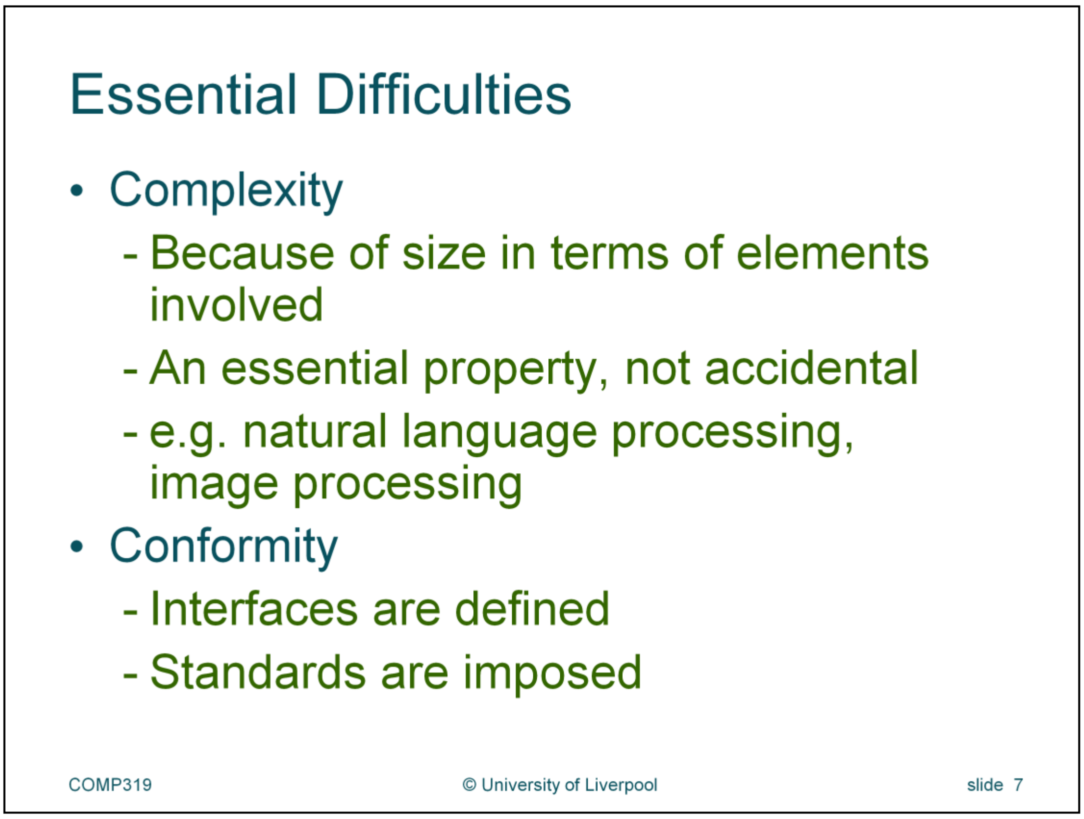
The werewolf exists because software engineering is complex and tackling the complexity is hard.

But what exactly makes it hard …


Software crisis

- We noted that software engineering is hard
- Why?
  - It must perform
  - It is boxed (time, money, size)
  - It is constrained by hardware, designs, use
  - It is obsolete very quickly
  - It is complex

COMP319 © University of Liverpool slide 6



Essential Difficulties

- Complexity
  - Because of size in terms of elements involved
  - An essential property, not accidental
  - e.g. natural language processing, image processing
- Conformity
  - Interfaces are defined
  - Standards are imposed

COMP319 © University of Liverpool slide 7

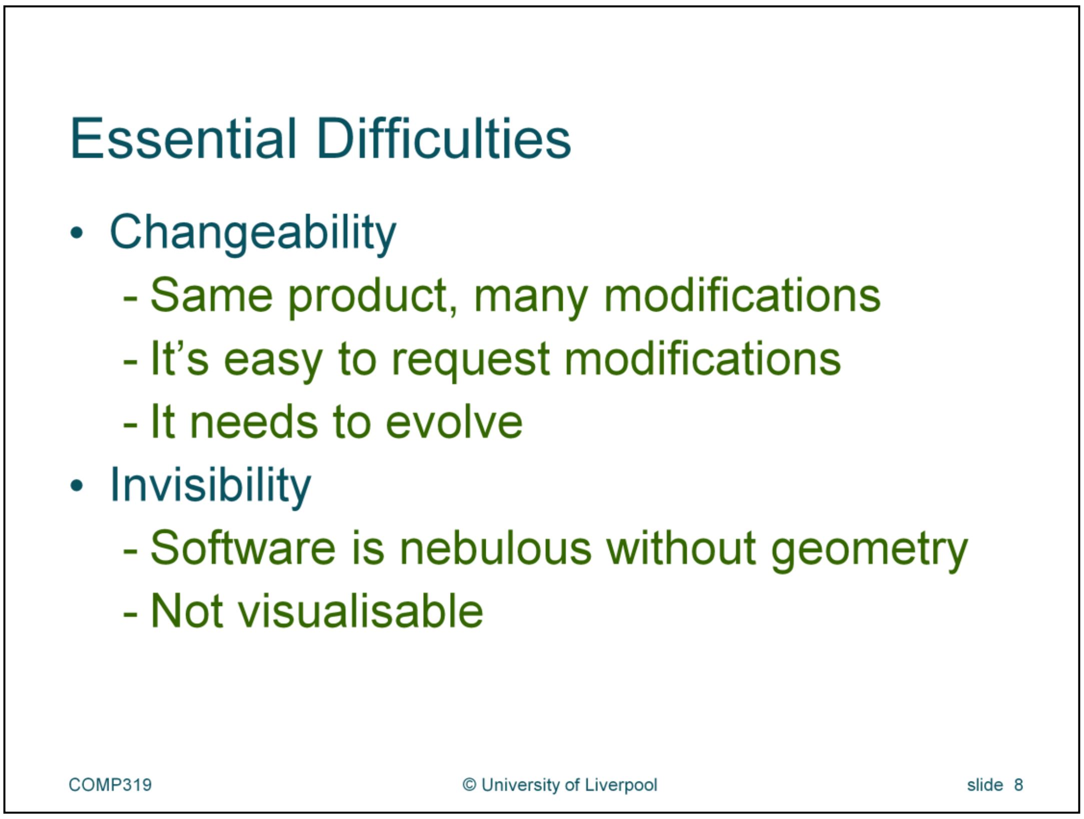
Thus we have a hard task that is inherently difficult to solve – what are the difficulties?

Brooks in NSB suggests 4 essential difficulties.

It is complex because there are more parts involved than any single human can handle without resorting to memory aides and simplifications. This is loosely measured in terms of “elements” although we could substitute “module” or “object” without losing the general idea.

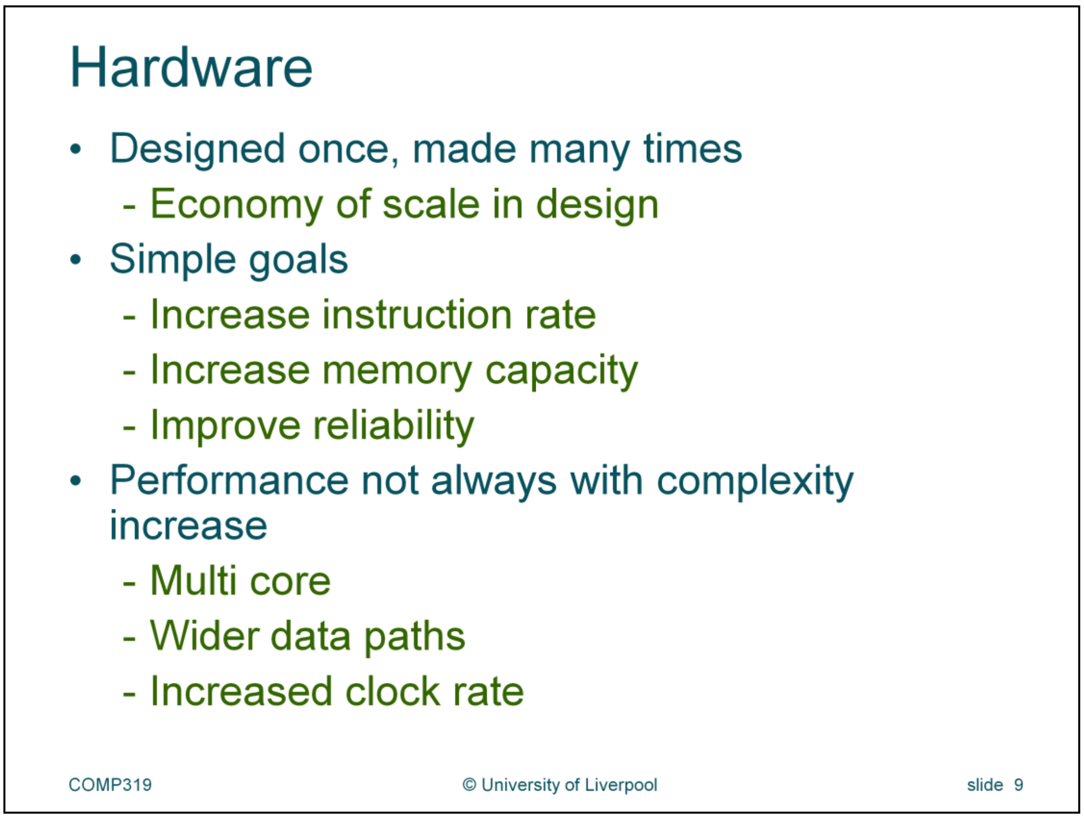
The complexity is inherent in that a conceptual structure is required which is far more complex than any other structure that man regularly constructs. Brooks compares it to constructing a building or a bridge where spatial issues help keep the level of complexity under control. In contrast in software engineering there is no spatial geometry that we can rely on.

The second difficulty is associated with the number of interfaces and standards which the software must comply with. This conformity requirement is serious problem that has to be solved. Every interface must be defined and agreed between those on either side, and all agreements must be documented. Standards are required because the software is part of an environment defined by others in which it must operate. These external standards spawn internal ones – which also need to be agreed and documented.



Essential Difficulties

- Changeability
  - Same product, many modifications
  - It’s easy to request modifications
  - It needs to evolve
- Invisibility
  - Software is nebulous without geometry
  - Not visualisable



Hardware

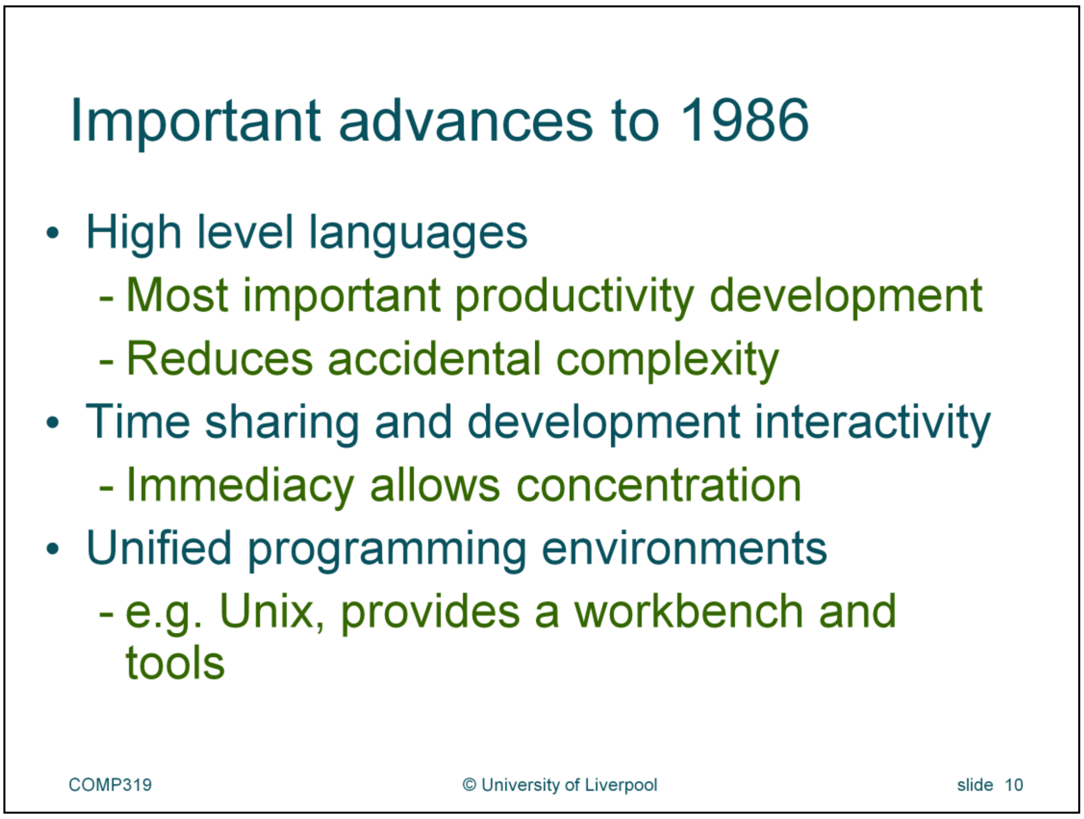
- Designed once, made many times
  - Economy of scale in design
- Simple goals
  - Increase instruction rate
  - Increase memory capacity
  - Improve reliability
- Performance not always with complexity increase
  - Multi core
  - Wider data paths
  - Increased clock rate

COMP319 © University of Liverpool slide 9

It is worth noting that hardware seems to manage improved performance year on year where software does not. This is due to a number of reasons:

A CPU is designed once and then many copies of the same chip are fabricated, this allows the chip designers to throw a lot of resources at that 1 design. Once the design is confirmed as working it is not changed, this means that changes and bugs cannot creep in (unlike software that is expected to change throughout its lifetime).

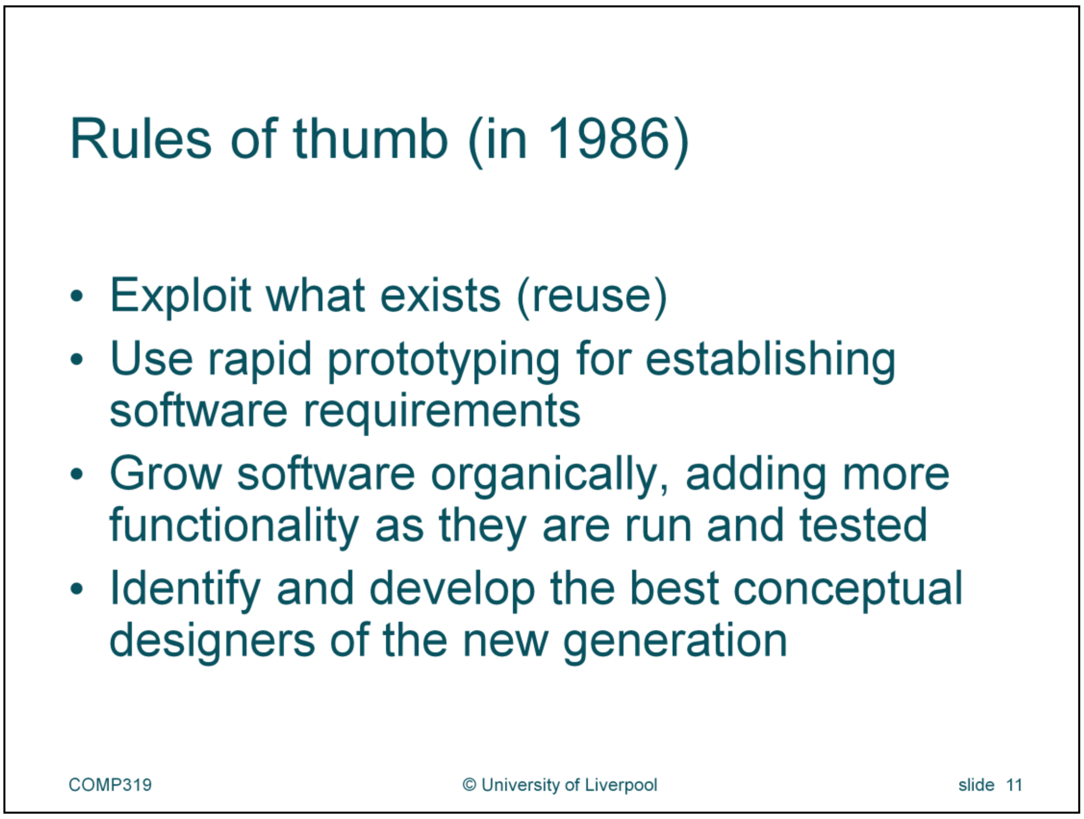
The other reason that hardware can achieve its improved performance easier than software can, if that performance can often be improved without increasing the complexity of the design, for example increasing the clock rate or using multiple processing units are relatively simple approaches to increasing the throughput.


Important advances to 1986

- High level languages
  - Most important productivity development
  - Reduces accidental complexity
- Time sharing and development interactivity
  - Immediacy allows concentration
- Unified programming environments
  - e.g. Unix, provides a workbench and tools

COMP319 © University of Liverpool slide 10

Brooks noted in 1986 that what had been successful up to then could be summarised in terms of 3 significant developments. The most important was the development of high level languages. Brooks argued that the improvement in HLL was responsible for a five-fold increase in productivity over the use of assembly level languages. It did this by reducing the risk of including **accidental** complexity. The program consists of conceptual constructs operations, data-types, sequences, and communications rather than machine constructs such as bits, registers, conditions, branches, channels, disks, etc. Eliminating this accidental complexity frees the programmer to solve the conceptual issues. He notes that by 1986 however that there was the risk that languages could not improve further because the sophistication they introduced exceeded what was useful or could be absorbed by the programmer. Interactive development of programs is credited with a two to four-fold increase in productivity. The reason was not necessarily because more code could be written but because simple syntax errors could be easily spotted and immediacy means that the minutiae of a program could be held in human memory long enough to be usefully employed in spotting semantic and logic errors. As system response times fell below about 100 milliseconds Brooks noted that further benefit from this aspect could not be expected. Finally, integrated programming environments such as Unix and Interlisp had improved productivity by a factor of about two to four. This was because "… they attack the accidental difficulties of using programs *together,* by providing integrated libraries, unified file formats, pipes, filters, etc." [MMM p 187] The development of the programmer workbench approach follows naturally where new tools can be easily incorporated and made available to everyone participating. In total these methods offer an order of magnitude improvement in productivity – however, in the same period hardware speed and storage efficiency increased by several orders of magnitude



Rules of thumb (in 1986)

- Exploit what exists (reuse)
- Use rapid prototyping for establishing software requirements
- Grow software organically, adding more functionality as they are run and tested
- Identify and develop the best conceptual designers of the new generation

COMP319 © University of Liverpool slide 11

In 1986 these were the rules of thumb for effective software engineering. While I go through them think if any of them have changed or are now obsolete.

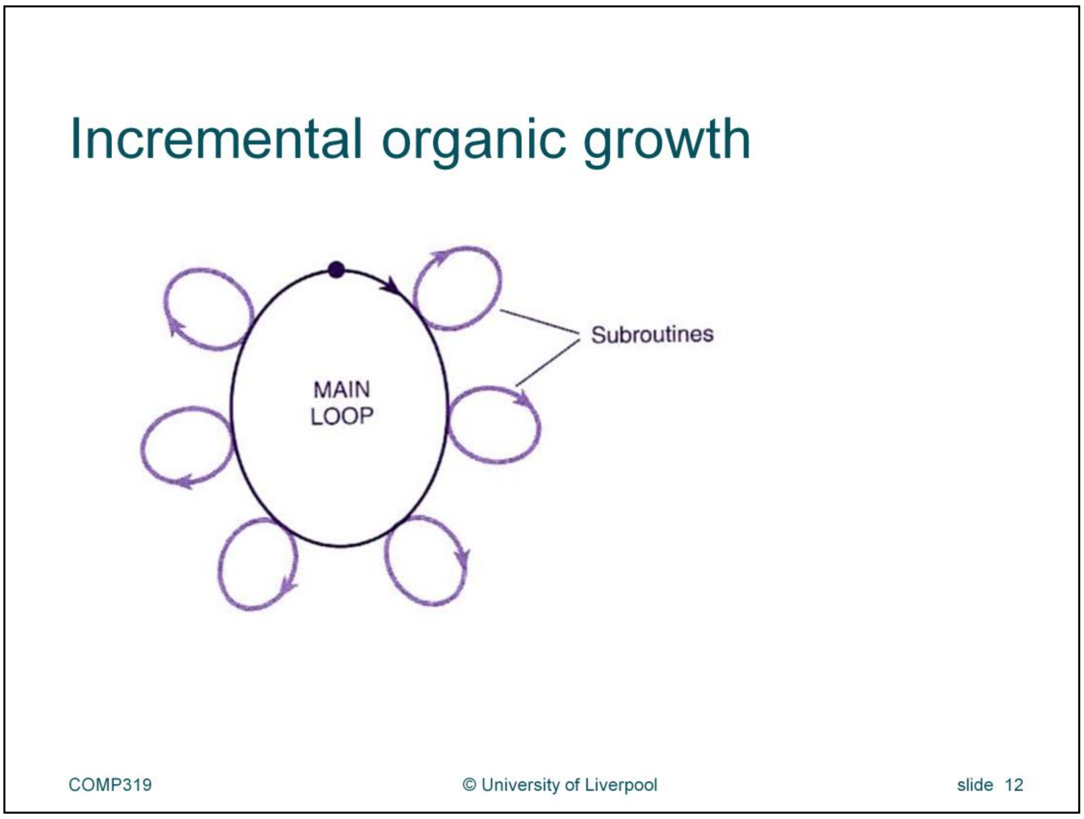
Simply stated the first is to avoid re-inventing the wheel.

Software requirements are best established by rapidly building prototypes that allowed customers and those involved in the development process (mainly programmers) to see what sort of interfaces and functions are required.

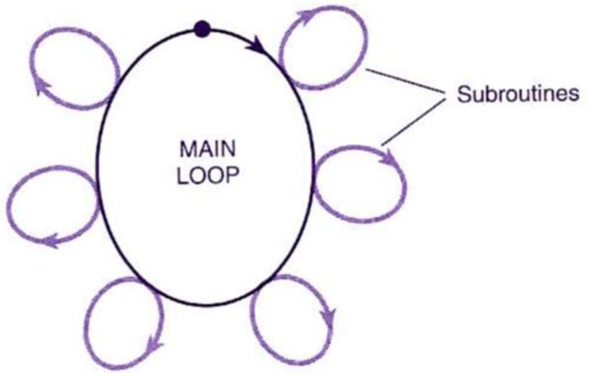
Grow software by adding more and more functions as they run, are used, and get tested.(see next slide)

Finally and most importantly identify the most productive conceptual designers, pay them and groom them to do their best

How have things changed since 1986 … ? We will look at this later in this module.



Incremental organic growth

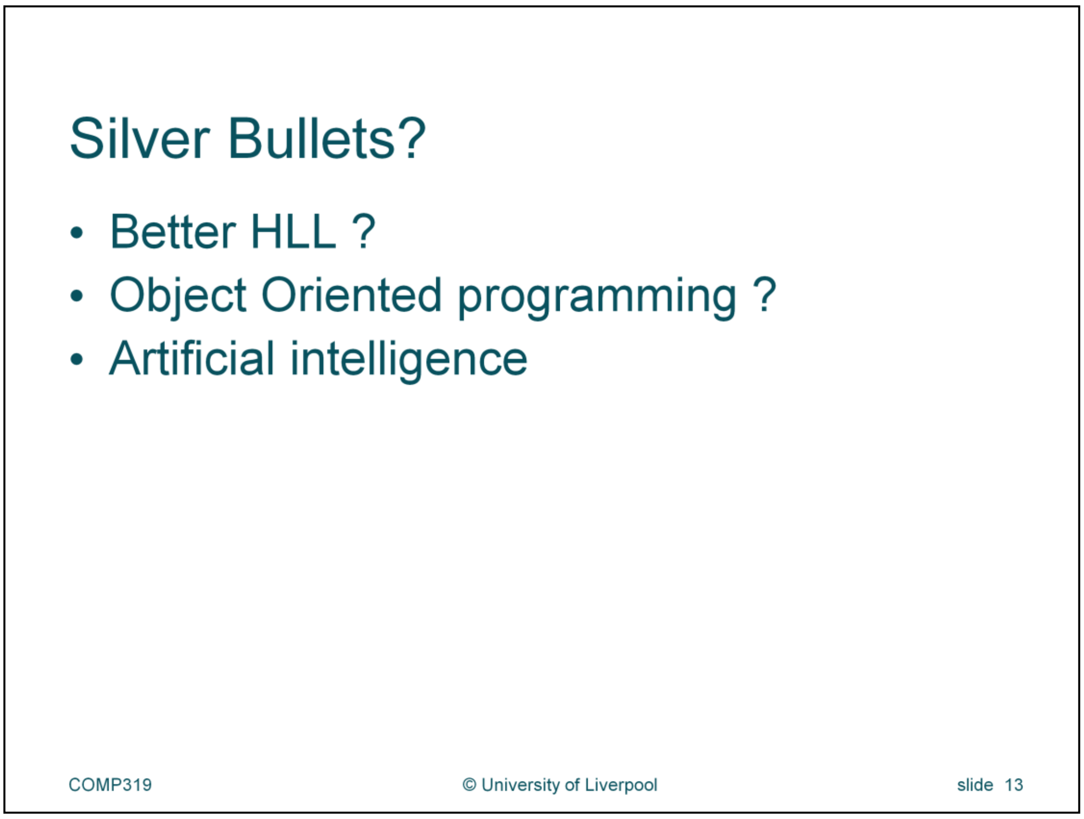


COMP319 © University of Liverpool slide 12

Basic idea is that you rapidly prototype the main interaction loop.

To this you add functions (called subroutines by Brooks, but could also be modules and other similar terms) and change the interaction main loop as required.

```
While not finished
  funtion1
  function2
  function3
  .
  .
  functionN
End while loop
```


Silver Bullets?

- Better HLL ?
- Object Oriented programming ?
- Artificial intelligence

COMP319 © University of Liverpool slide 13

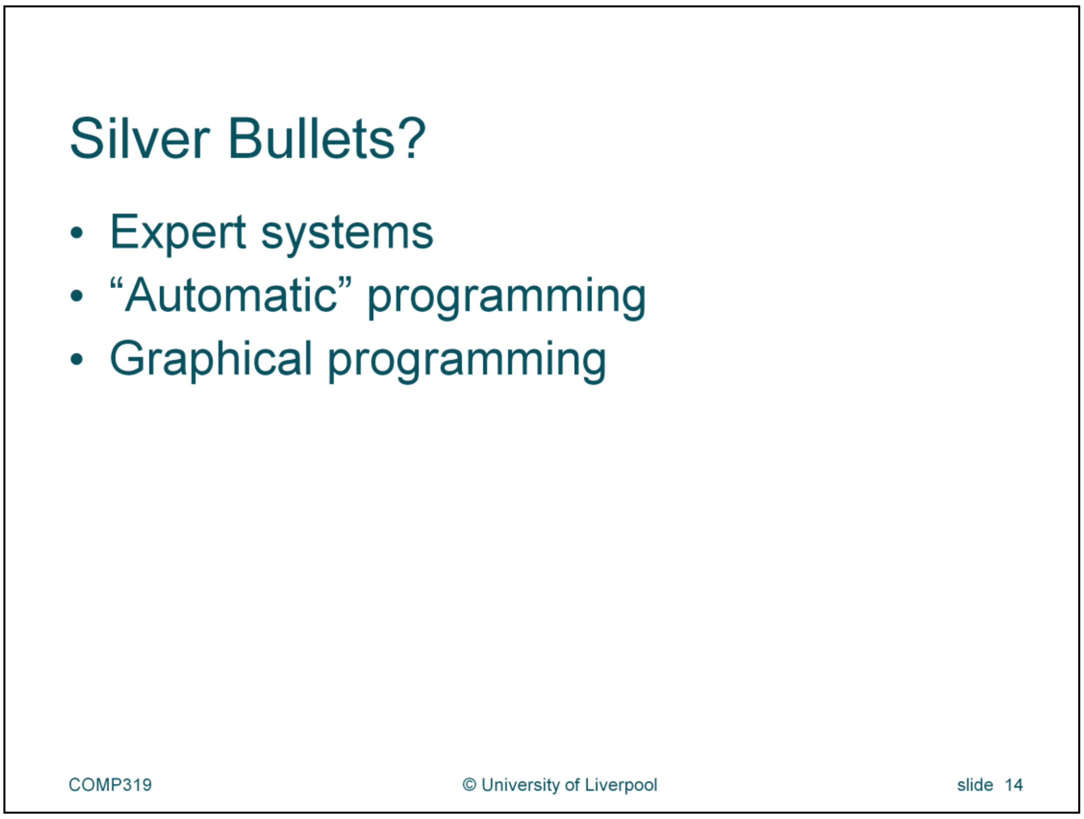
Brooks identified the problem (werewolf/software productivity) and then looked at the potential solutions (silver bullets) that were around in 1986.

HLLs like Ada add nothing more to the HLLs that already exists.

OO held the most promise in 1986. However, most of the then existing OO languages added only one feature (abstract data type as classes with methods) and while this did reduce some accidental complexity no order of magnitude gain in productivity could be seen by that addition.

(The most popular current OO language Java certainly does nothing more in 2004!)

In 1986 it was thought that order of magnitude productivity gains would come from AI. However, AI apart from Expert Systems, have looked at systems such as human speech and image recognition and tried to emulate them. However, it is hard to see how the lessons learnt in either of these could improve software engineering.



Silver Bullets?

- Expert systems
- “Automatic” programming
- Graphical programming

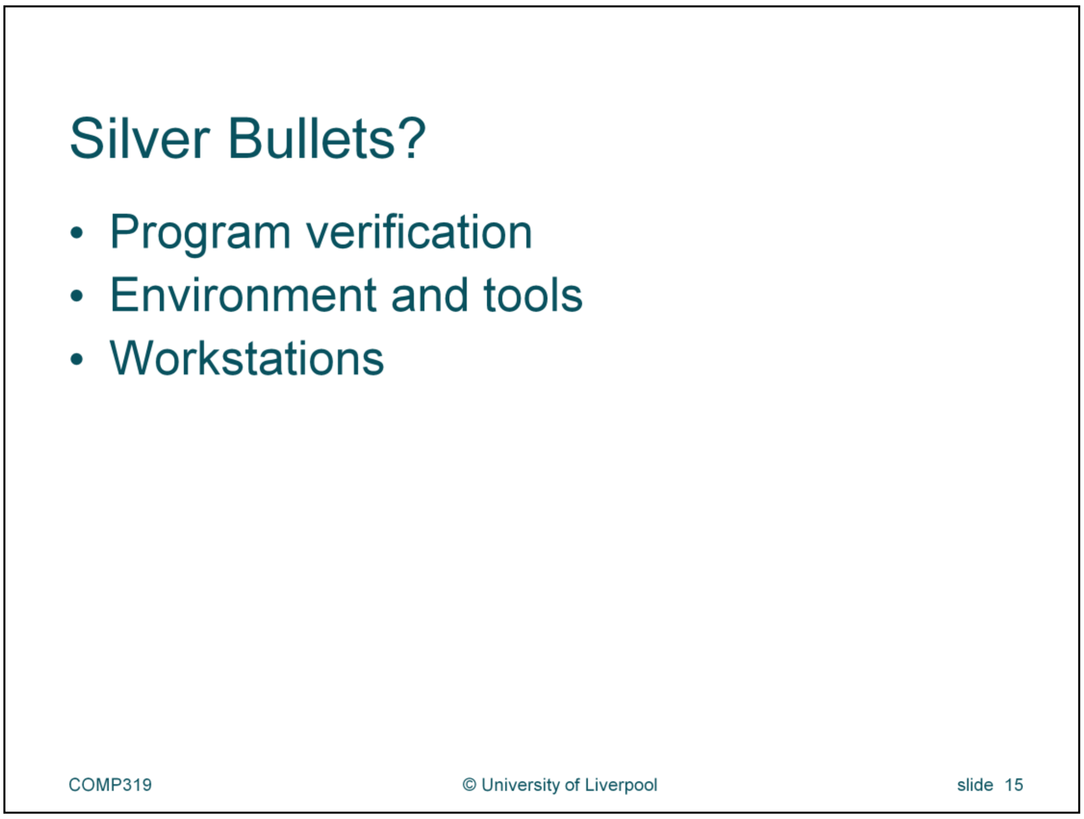
COMP319 © University of Liverpool slide 14

“An expert system is a program containing a generalised inference engine and a rule base. It is designed to take input data and assumptions and to explore the logical consequences through the inferences derivable from the rule base, yielding conclusions and advice, and offering to explain its results by retracing its reasoning for the user. The inference engine typically can deal with fuzzy or probabilistic data and rules in addition to purely deterministic logic” … Brooks MMM, 1986.

The inference engine is a general and highly advanced tool that can be carried unchanged from domain to domain. The real power of Expert Systems are captured in their rule bases. These in software engineering capture interface rules, test strategies, typical bugs, optimisation, etc. Such expert systems reduce the labour in producing implementations and beefs up the productivity of inexperienced programmers by capturing the best practice of experts – however acquiring this knowledge from the experts and analysing why they do what they do is hard, and from the outset is doomed only to repeat what that expert can expect to do quickly. Brooks thought it unlikely that order of magnitude gains in software productivity could be obtained.

“Automatic” programming is the generation of a program from a statement of the problem specification. In 1986 Brooks noted that the technique worked where it was possible to generate an exhaustive specification involving limited sets of parameters and known solution methods exist, e.g. for sorting or integration of differential equations. Such systems do not generalise well and even in the cases outlined have not contributed to increasing productivity.

Graphical programming works on the premise that some diagram can capture the program structure. Brooks argues that this is bound to fail because of the size limitations of the visible screen and the multidimensional and unvisualisable nature of programs. The analogy with VLSI chip design is flawed because chips are layered and two-dimensional where the geometry can reflect the essence of the connections that are required. Software systems do not reflect



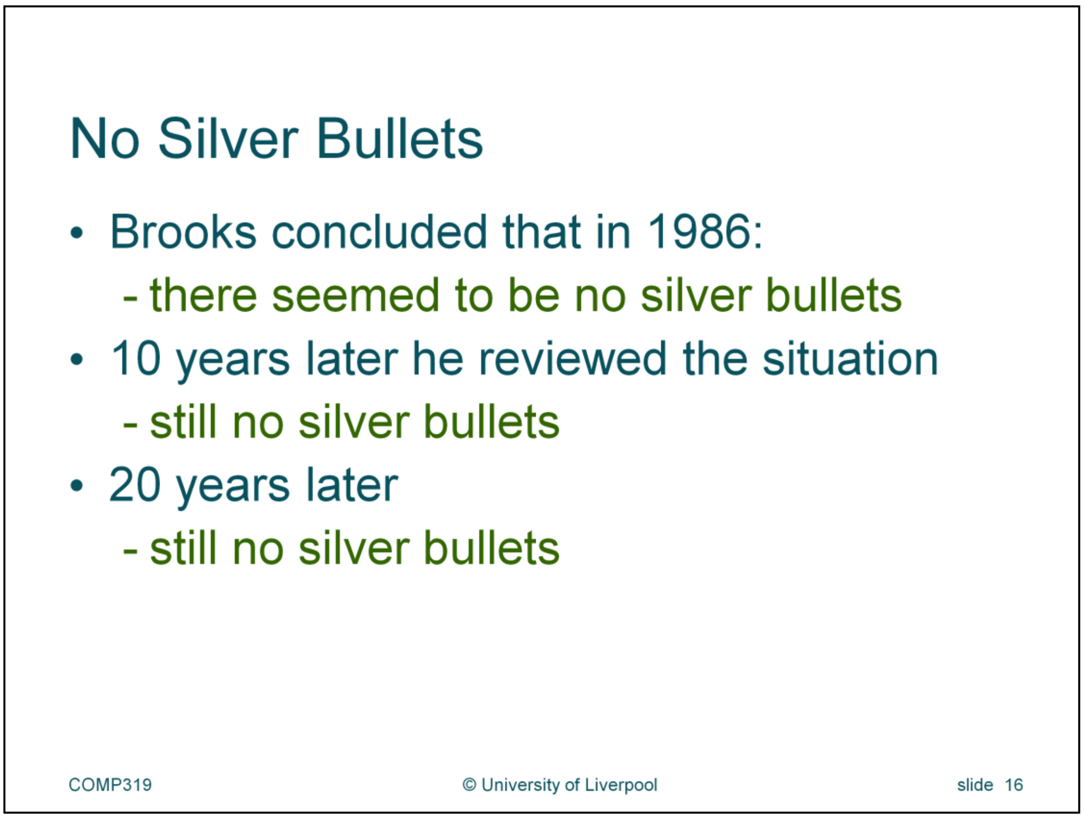
Silver Bullets?

- Program verification
- Environment and tools
- Workstations

COMP319 © University of Liverpool slide 15

Program verification prior to coding in 1986 (and now) was/is the subject of much research. The premise is that if you can verify that the design is free of bugs productivity and product reliability is enhanced. The immense effort involved in testing programs is avoided if such a verification step can be done as part of the design step. Brooks noted that while this was a powerful concept, especially for system kernels and secure operating systems the method does not save effort. Verification involves so much work that only a few crucial programs have ever been verified. Similarly mathematical proofs too except in very simple cases seem always to be faulty – leading to the conclusion that while verification might reduce the program testing load, it cannot eliminate it, and cannot eliminate it in predictable ways. At best program verification can only establish that a program meets its specification – the hardest part is arriving at that specification in the first place. Much of the essence of building a program is in fact the debugging of the specification.

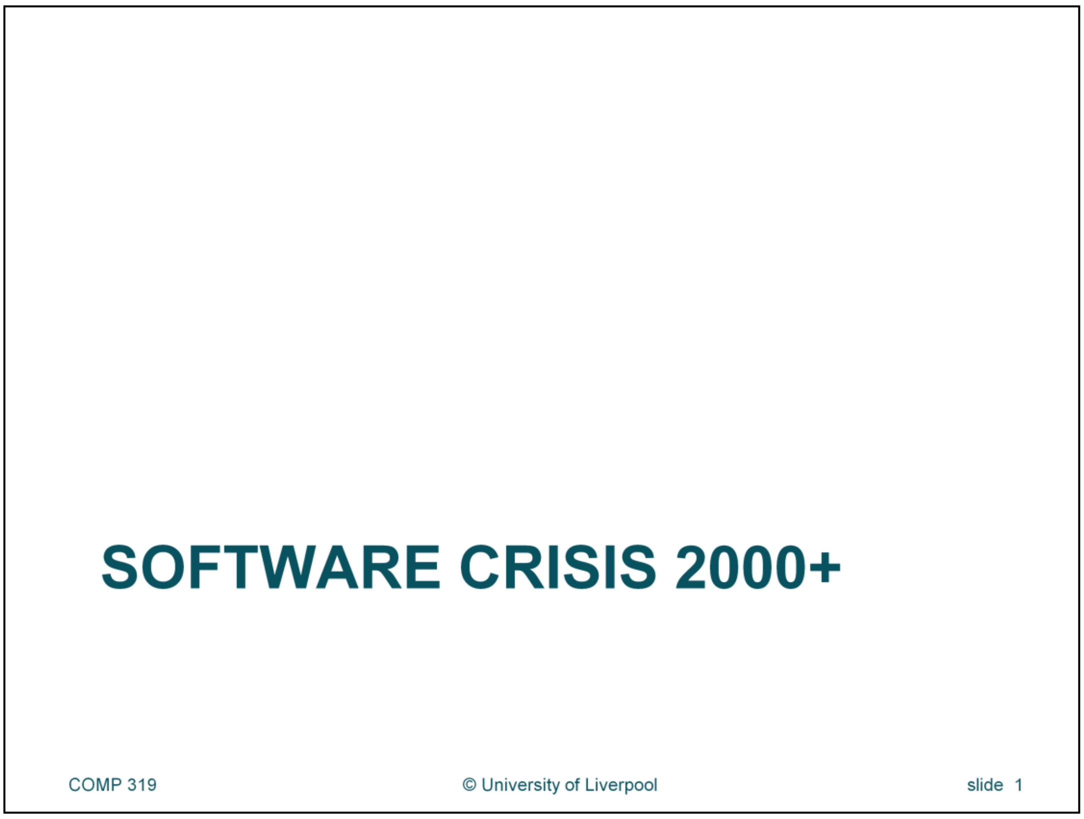
New programming environments and software tools provide scope for the biggest gains in productivity. The use of language specific editors and integrated databases keeping track of detail for recall by the programmer and his colleagues working on a software project was estimated by Brooks as being very worthwhile, but the productivity grains were likely to be marginal.



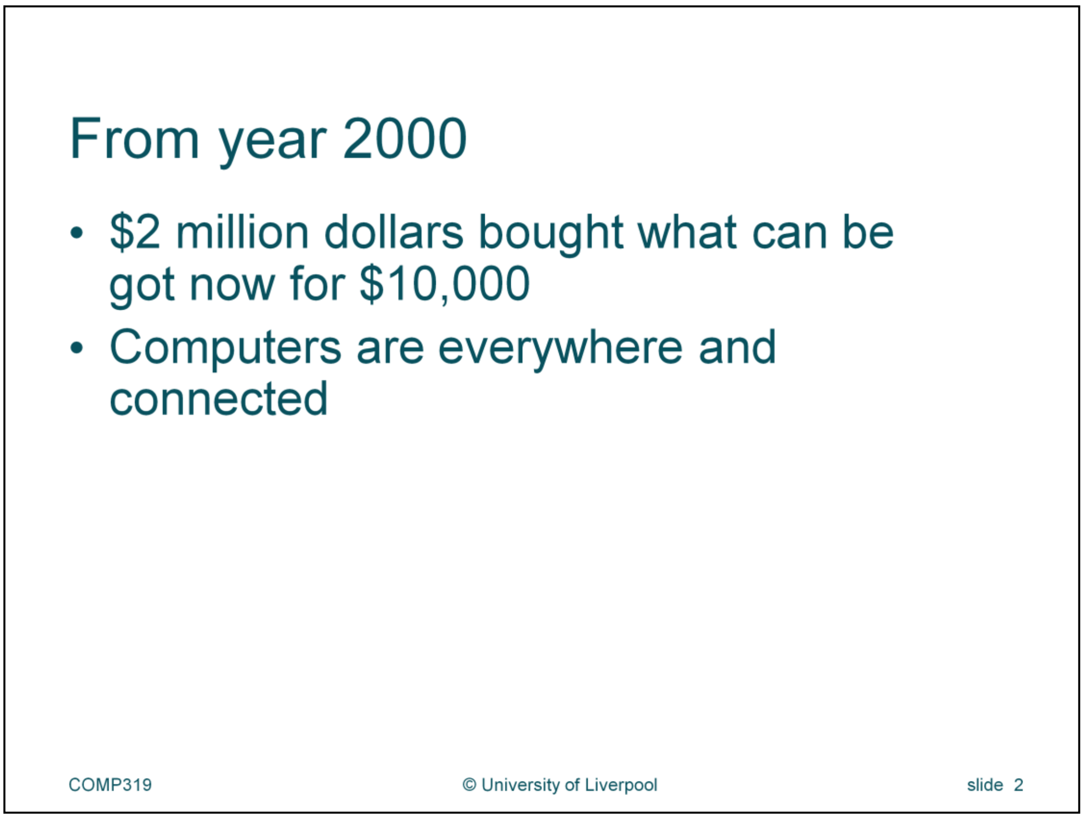
No Silver Bullets

- Brooks concluded that in 1986:
  - there seemed to be no silver bullets
- 10 years later he reviewed the situation
  - still no silver bullets
- 20 years later
  - still no silver bullets

COMP319 © University of Liverpool slide 16



SOFTWARE CRISIS 2000+



From year 2000

- $2 million dollars bought what can be got now for $10,000
- Computers are everywhere and connected

COMP319 © University of Liverpool slide 2

This side of the millennium what is different in computing, and what is different in software engineering?

The most important thing, without doubt, is the cost of the products.

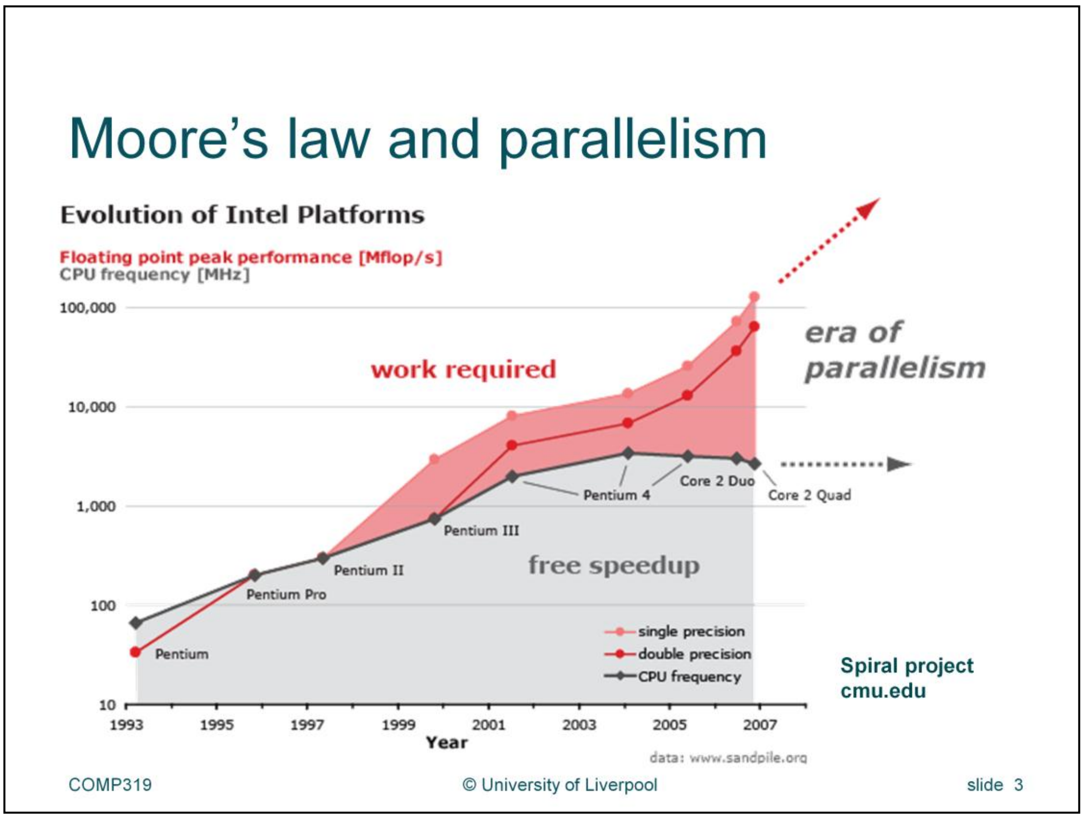
In 1972 the Michigan Terminal System was implemented on an IBM model 360/68 that cost about £1.5 million, of which £60K was for the building modifications and water cooling system.[Accidental tasks included handling fungus in the cooling pipes – bugs indeed!].

Now for about £1,000 I have about the same processing power and about the same disk space on this laptop. The key difference is that only I use the laptop whereas the 360 had about 250 simultaneous users at any time 24 hours a day, 7 days a week. It also had access to gigabytes of tape storage – though it needed a team of about 7 computer operators to keep it fed, watered, milked, serviced and taped.

The second change has come about because of the first. Computers are now everywhere. These are essentially still "computers", but soon the computer will vanish into your telephone, car, television, radio, clothes, container, under your pet's skin, and probably into a small cavity in your skull.

A part of this change, and probably inseparable from it, is that many of these computers are connected. The connections are by wire or some part of the electro-magnetic spectrum: e.g. radio, microwaves, infra-red, sound, and laser light.

The software has changed too.



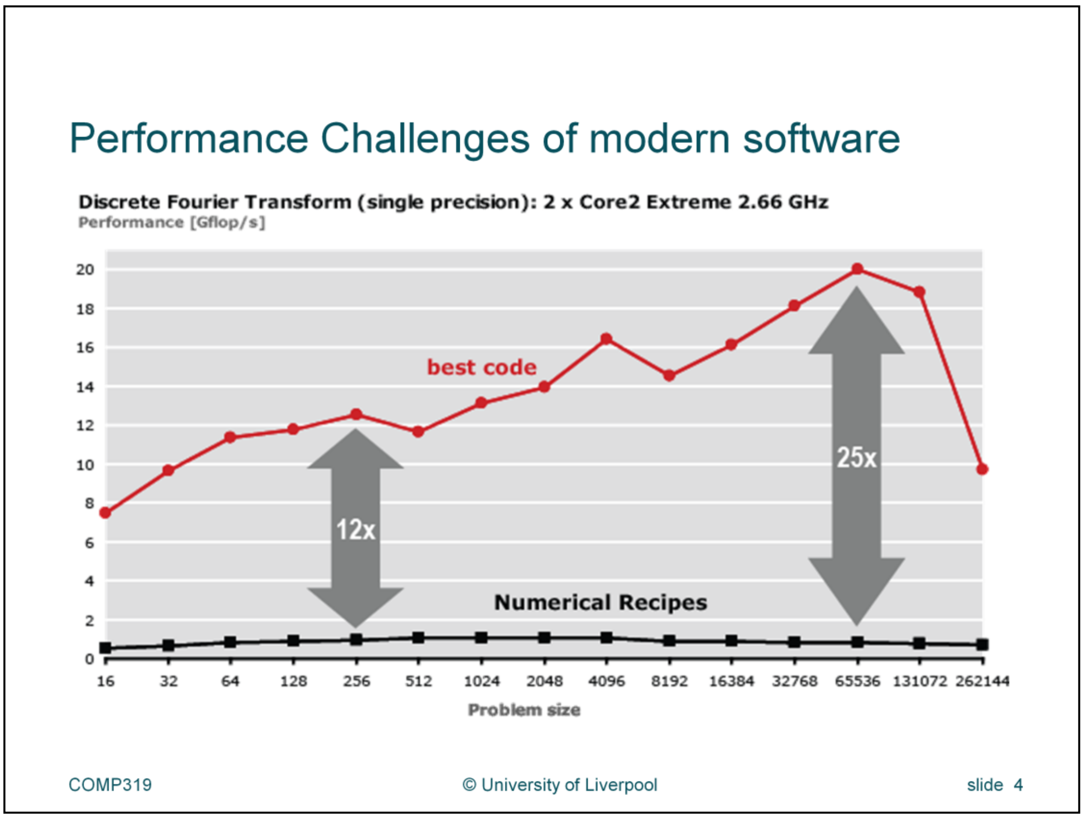
Moore's law and parallelism

COMP319 © University of Liverpool slide 3

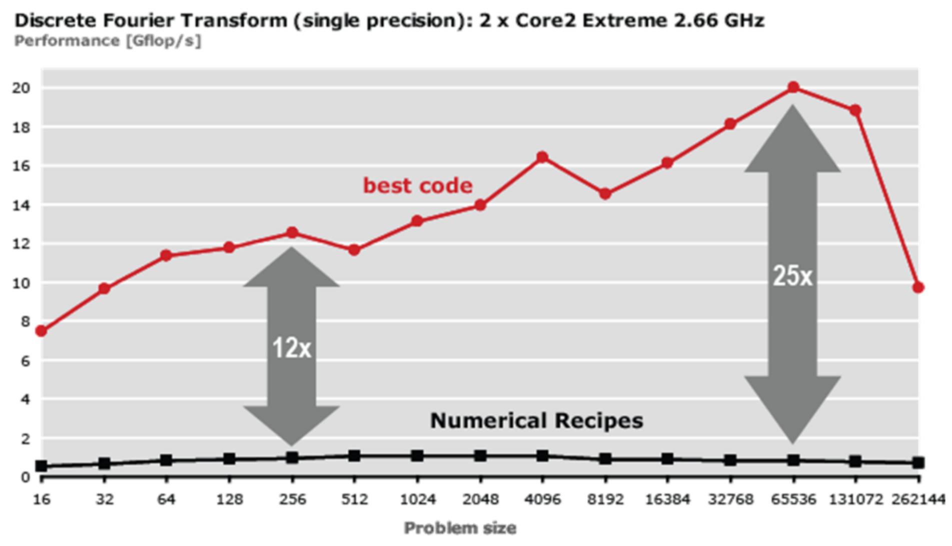
We can see that processor clock speed for current silicon technologies is hitting a wall in terms of "free" speed up of software execution.

New processors rely on using multi-core technology to improve performance.

Unless the software has been optimised for running in parallel then no performance increase will be achieved, by adding more processors.



Performance Challenges of modern software



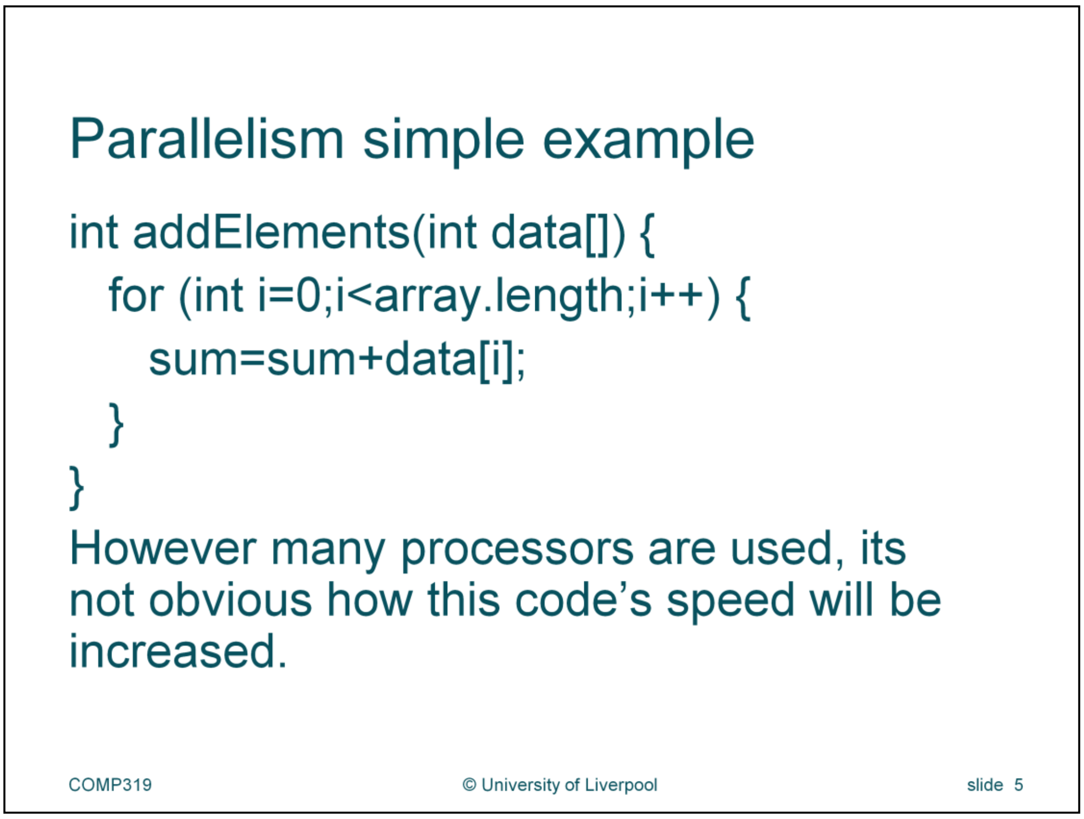
COMP319 © University of Liverpool slide 4

We can see how the larger the problem size is (in terms of processed data) the larger the difference between non-optimised and optimised code is.

These figures are from the Spiral project at Caregia Mellon University.

www.spiral.net.

Their core research question is “Can we teach computers to write fast libraries?”

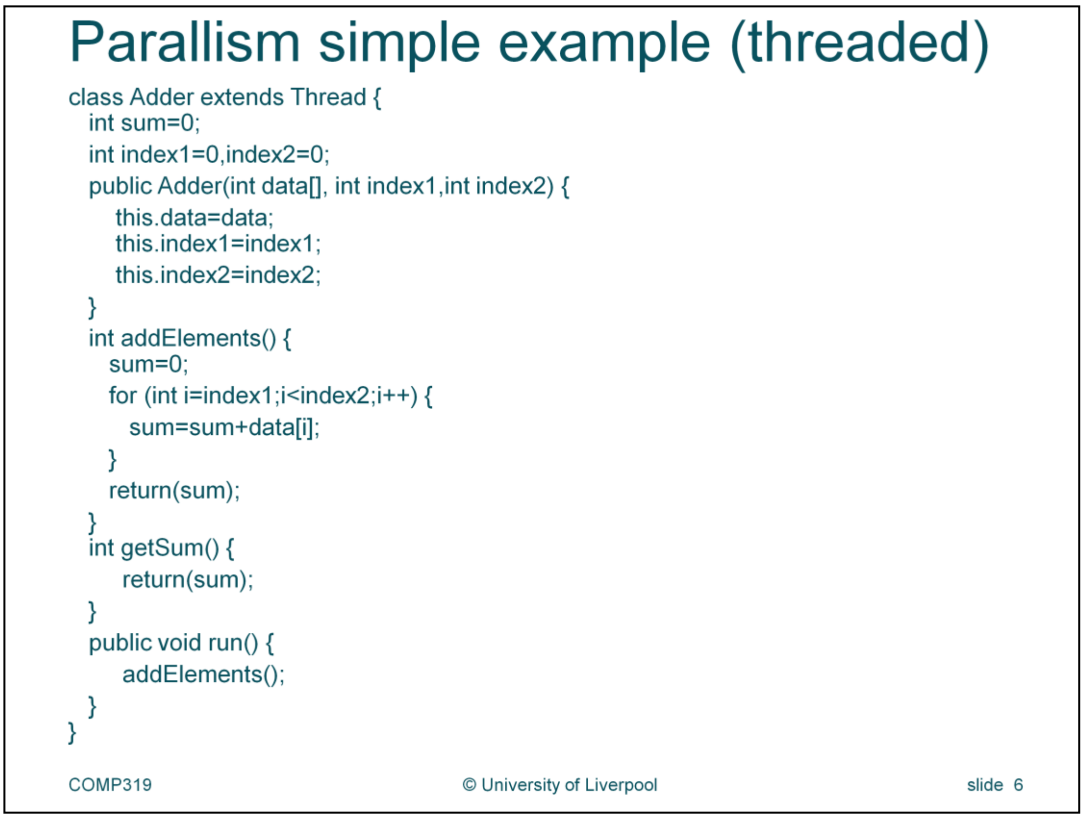


Parallelism simple example

```
int addElements(int data[]) {
  for (int i=0;i<array.length;i++) {
    sum=sum+data[i];
  }
}
```

However many processors are used, its not obvious how this code’s speed will be increased.

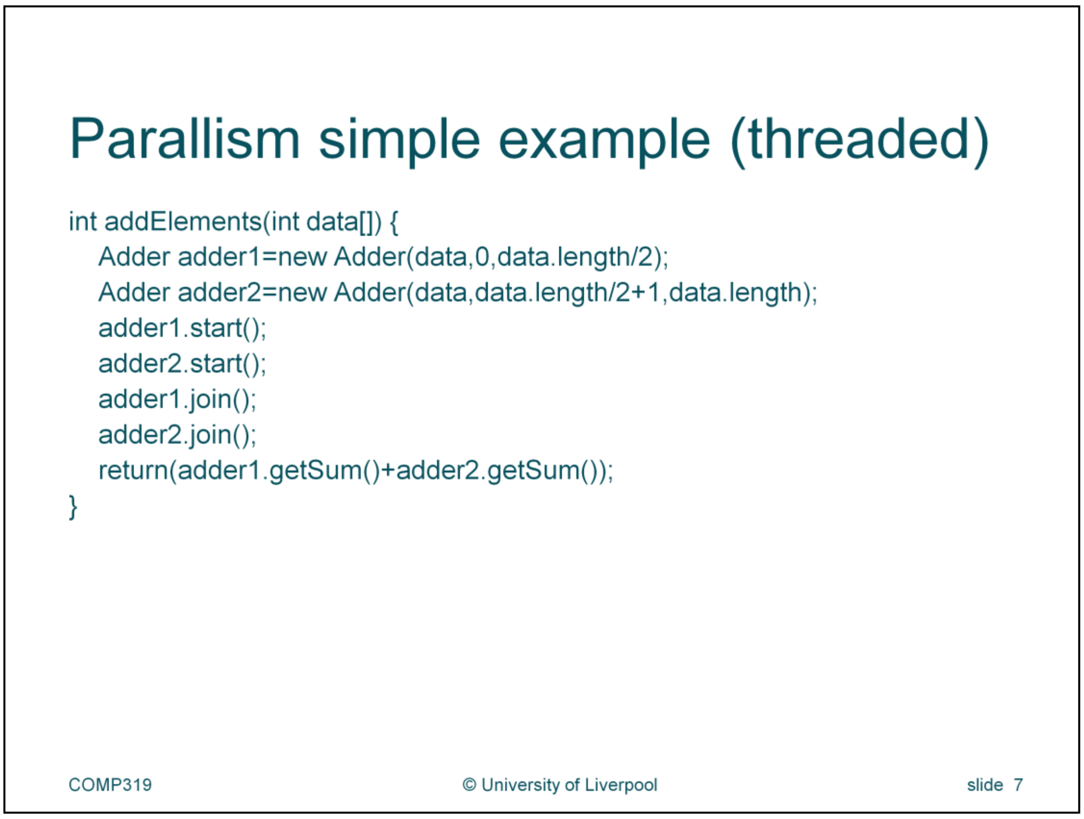
COMP319 © University of Liverpool slide 5



Parallism simple example (threaded)

```
class Adder extends Thread {
   int sum=0;
   int index1=0,index2=0;
   public Adder(int data[], int index1,int index2) {
       this.data=data;
       this.index1=index1;
       this.index2=index2;
   }
   int addElements() {
      sum=0;
      for (int i=index1;i<index2;i++) {
         sum=sum+data[i];
      }
      return(sum);
   }
   int getSum() {
        return(sum);
   }
   public void run() {
        addElements();
   }
}
```

COMP319 © University of Liverpool slide 6

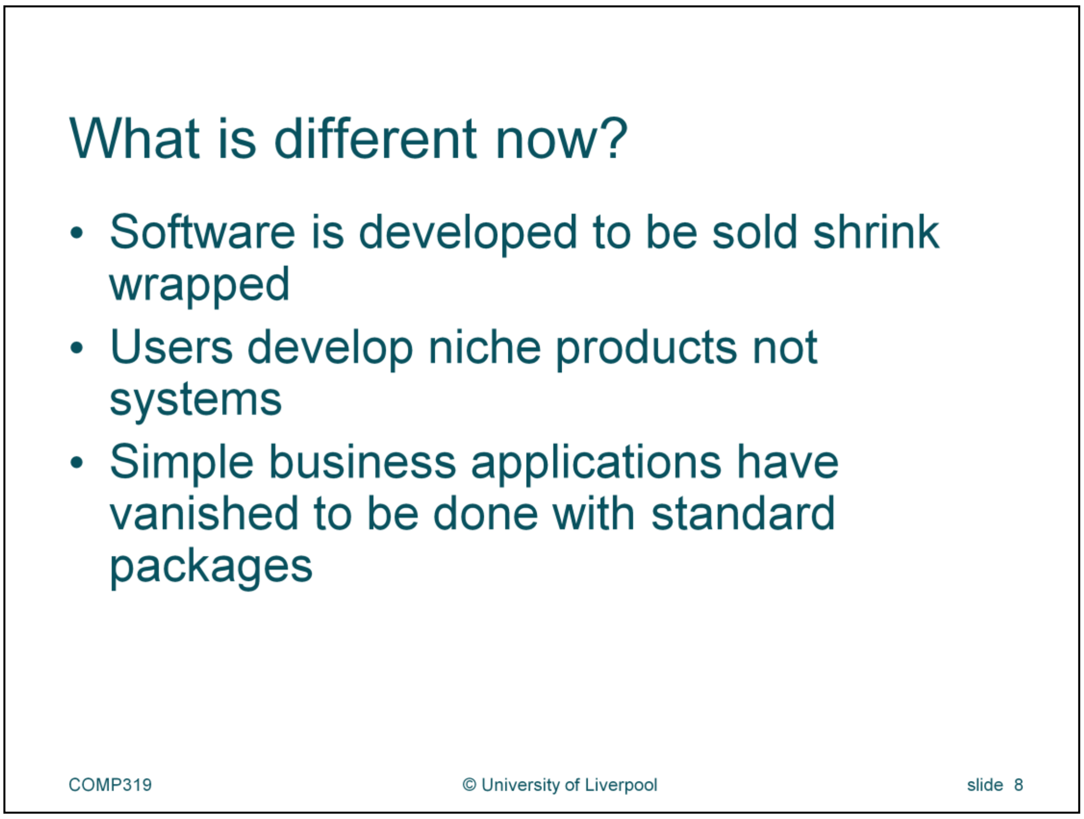


Parallism simple example (threaded)

```
int addElements(int data[]) {
   Adder adder1=new Adder(data,0,data.length/2);
   Adder adder2=new Adder(data,data.length/2+1,data.length);
   adder1.start();
   adder2.start();
   adder1.join();
   adder2.join();
   return(adder1.getSum()+adder2.getSum());
}
```

COMP319 © University of Liverpool slide 7

We can see that the amount of code and the code complexity has to be increased considerably to make use of more than 1 processor. To optimise the code further it needs to be able to handle more than 2 processors. Image doing the same optimization with a sort routine, what kind of sort algorithm would work well?


What is different now?

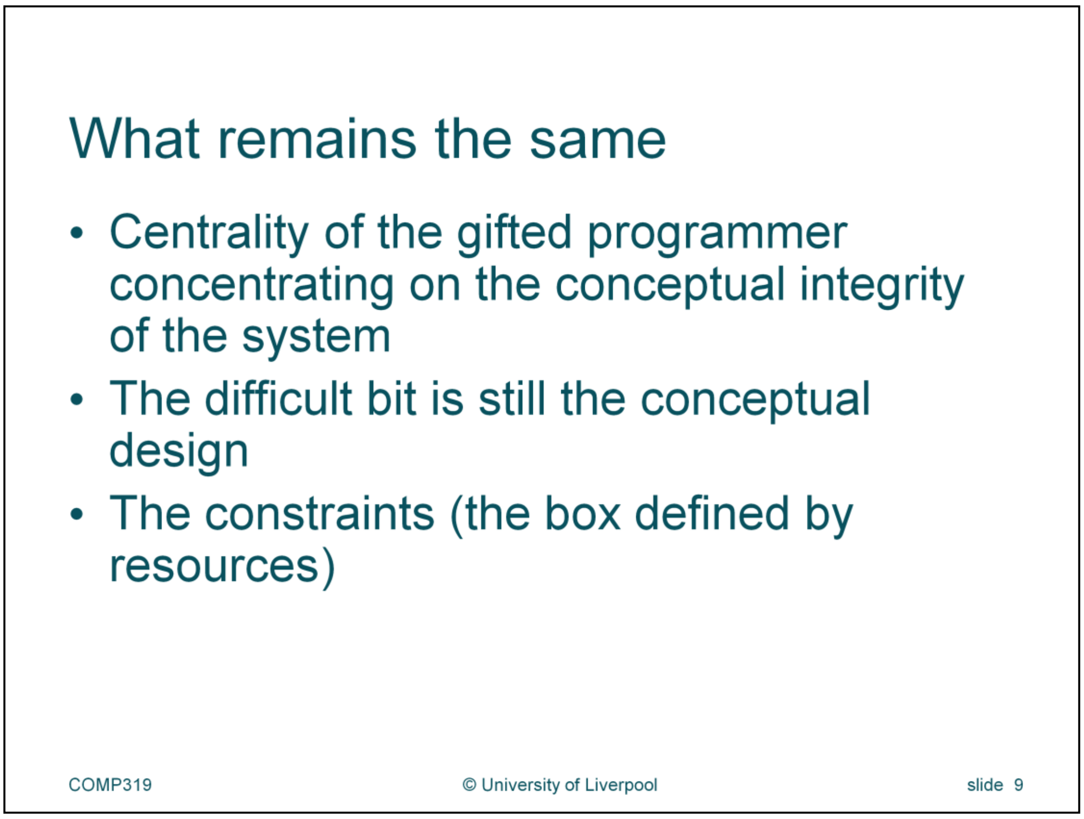
- Software is developed to be sold shrink wrapped
- Users develop niche products not systems
- Simple business applications have vanished to be done with standard packages

COMP319 © University of Liverpool slide 8

Software is now boxed and shrink wrapped or delivered automatically through some connection method. (Does everyone know how to use zip/unzip, etc?)

The users have changed too. They are everywhere not just behind business work-desks. Their requirements are for general tools, to be sure, but increasingly for niche products: computer games, genealogy software, finite element packages, music and image manipulation, DNA pattern search tools, etc. In each of these areas several producers exist – and where necessary the user can write the niche stuff himself using both general tools and niche targeted tools.

However, the main difference is that the main driver for application development is no longer business it is the home and niche markets. Much of what business does can be achieved with general tools, indeed the general tools which now exist: the word processor, spreadsheet, database were originally business specific tools.



What remains the same

- Centrality of the gifted programmer concentrating on the conceptual integrity of the system
- The difficult bit is still the conceptual design
- The constraints (the box defined by resources)

COMP319 © University of Liverpool slide 9

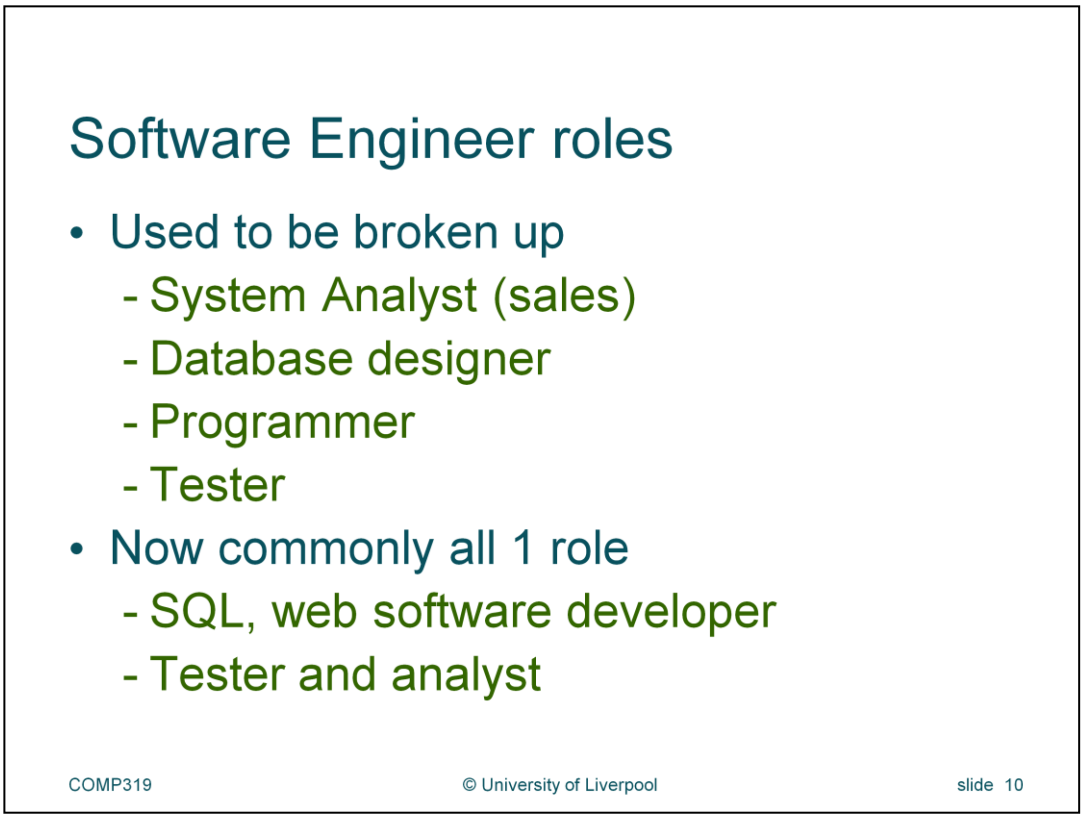
But what is new and different comes because of new products.

The software engineering process, as predicted by NSB, has not changed significantly. The key agent is still the talented programmer concentrating on the "conceptual integrity" of the product. There are just many more of them out there and not all of them have been snapped up by Microsoft or Oracle.

The programmer is still battling with the same creative tasks and accidental problems and still having problems with his conceptual design.

However, in some niche areas think say of games production, a multitude of tools and techniques have evolved that aim to ensure that a "new" product, say a Harry Potter game, can be brought to market in about 3-9 months.

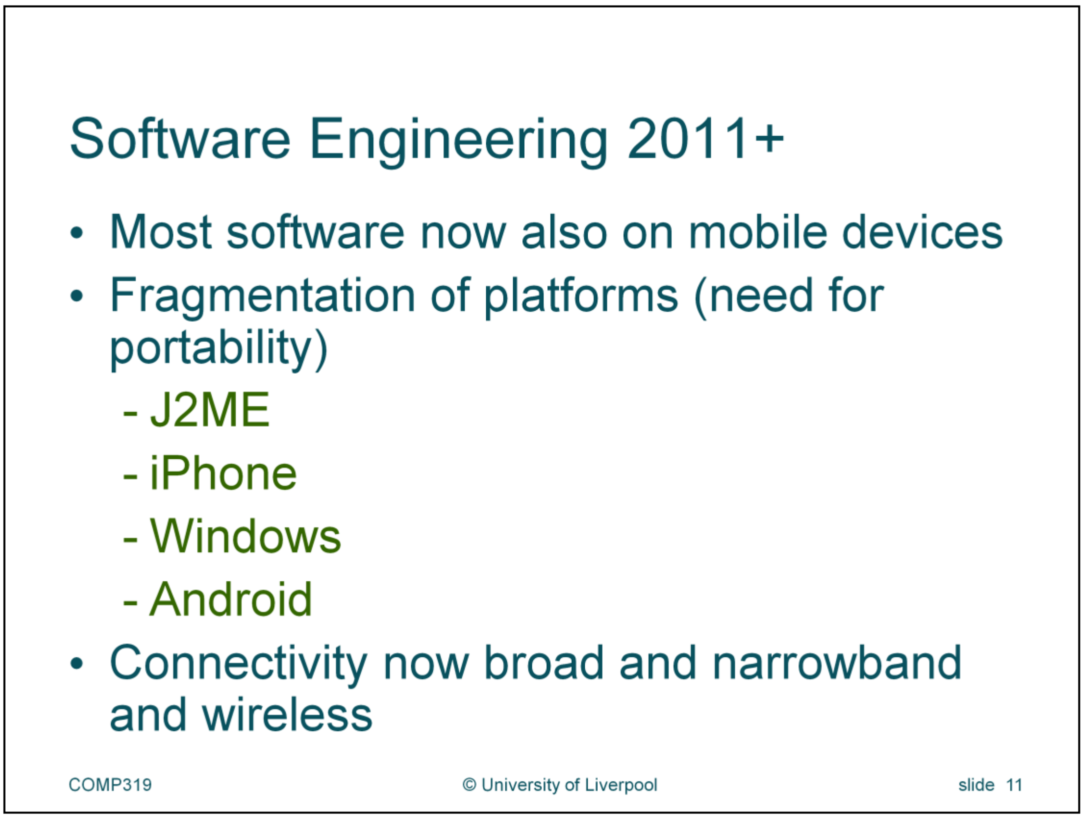
Development constraints and deployment constraints still remain although the numbers (e.g. of memory available to do the development or in the target computer, might increase



Software Engineer roles

- Used to be broken up
  - System Analyst (sales)
  - Database designer
  - Programmer
  - Tester
- Now commonly all 1 role
  - SQL, web software developer
  - Tester and analyst

COMP319 © University of Liverpool slide 10

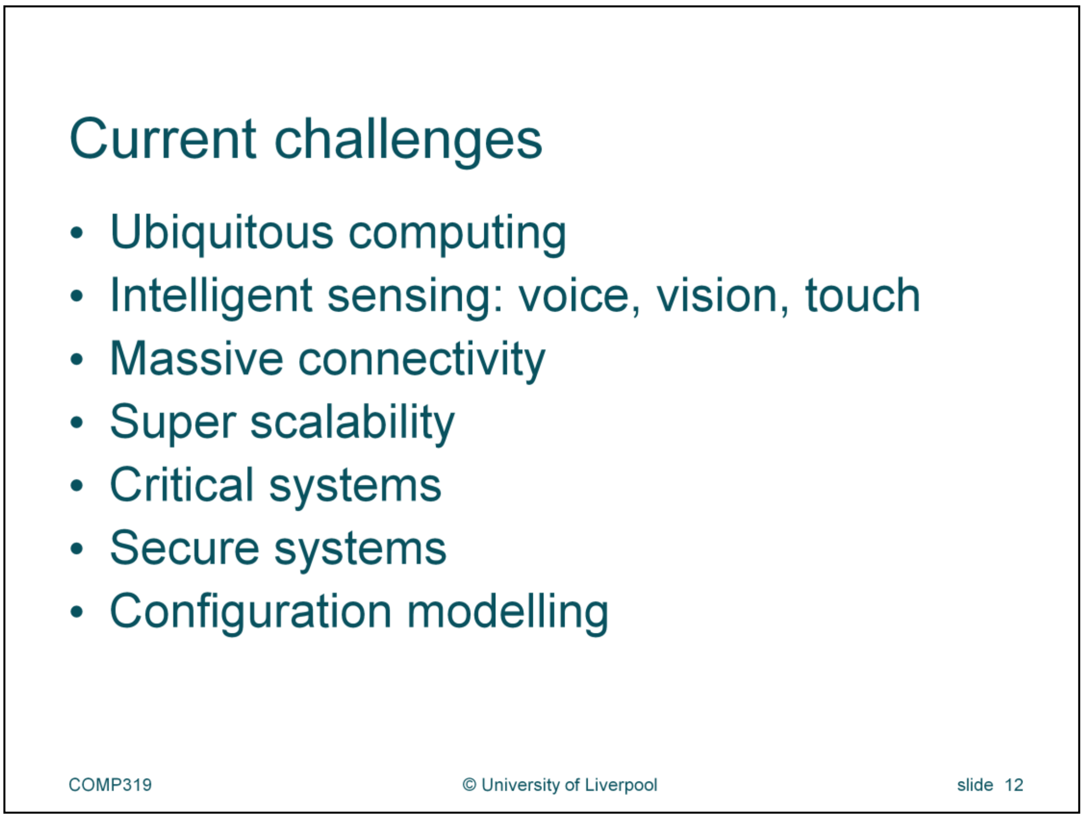


Software Engineering 2011+

- Most software now also on mobile devices
- Fragmentation of platforms (need for portability)
  - J2ME
  - iPhone
  - Windows
  - Android
- Connectivity now broad and narrowband and wireless

COMP319 © University of Liverpool slide 11

It is common now for software to require porting to mobile devices. This has particular challenges, in terms of platform fragmentation and therefore porting between platforms. Network connectivity for mobile applications is wireless, this is power hungry and often unreliable. Software has to be optimised for this unreliable network service.



Current challenges

- Ubiquitous computing
- Intelligent sensing: voice, vision, touch
- Massive connectivity
- Super scalability
- Critical systems
- Secure systems
- Configuration modelling

COMP319 © University of Liverpool slide 12

Ubiquitous computing – is defined as being "computing everywhere" . Perhaps the most exciting being the idea of wearable computers, in the very fabric of your clothes.

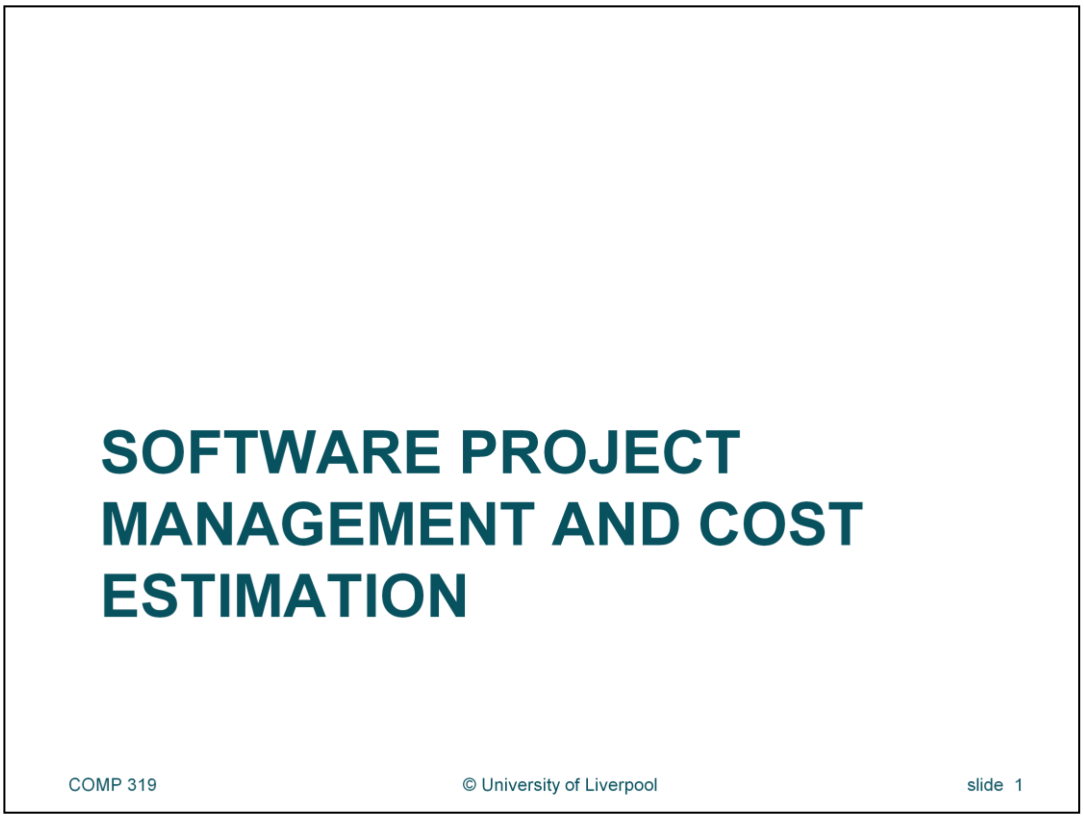
Intelligent sensing and processing, especially to process voice (e.g. voice interfaces), vision (I.e. computers that can see), and devices that relay touch to and from users (haptic devices).

Massive connectivity achieved by using all forms of communication and then using the connectivity and spare processing capacity to solve inherently difficult problems (includes grid computing).

Critical systems and dependable software. This is one of the key areas for software engineering. All the skills: fault avoidance, fault detection and recovery, and fault tolerance computing are required and are currently research issues. Examples are: power system monitoring, medical systems, telecommunication switches, aircraft auto-piloting.

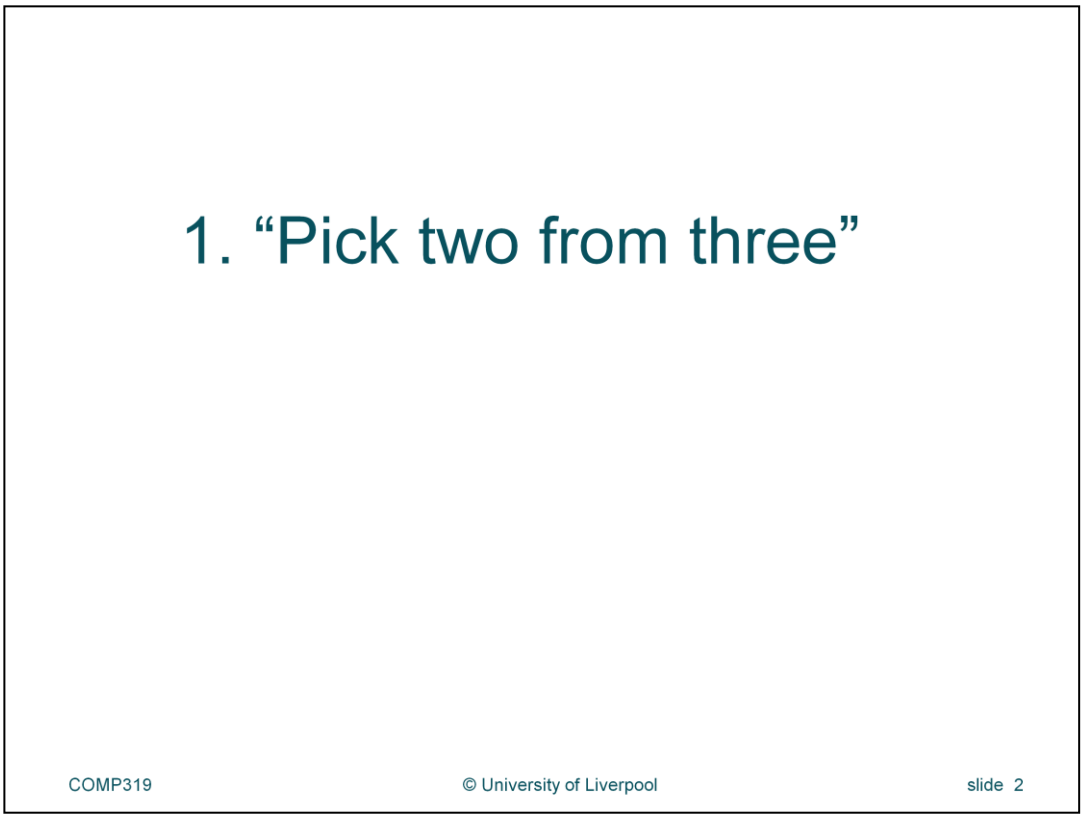
Secure systems include banking and finance systems that should be impervious to accidental and malicious damage. These systems are often also critical systems.

Configuration modelling is the area that configures equipment and services to meet specific criteria. Used in, e.g. telecommunications, to configure systems based on information such as equipment availability, usage traffic, cost, and services required.



SOFTWARE PROJECT MANAGEMENT AND COST ESTIMATION

COMP 319 © University of Liverpool slide 1

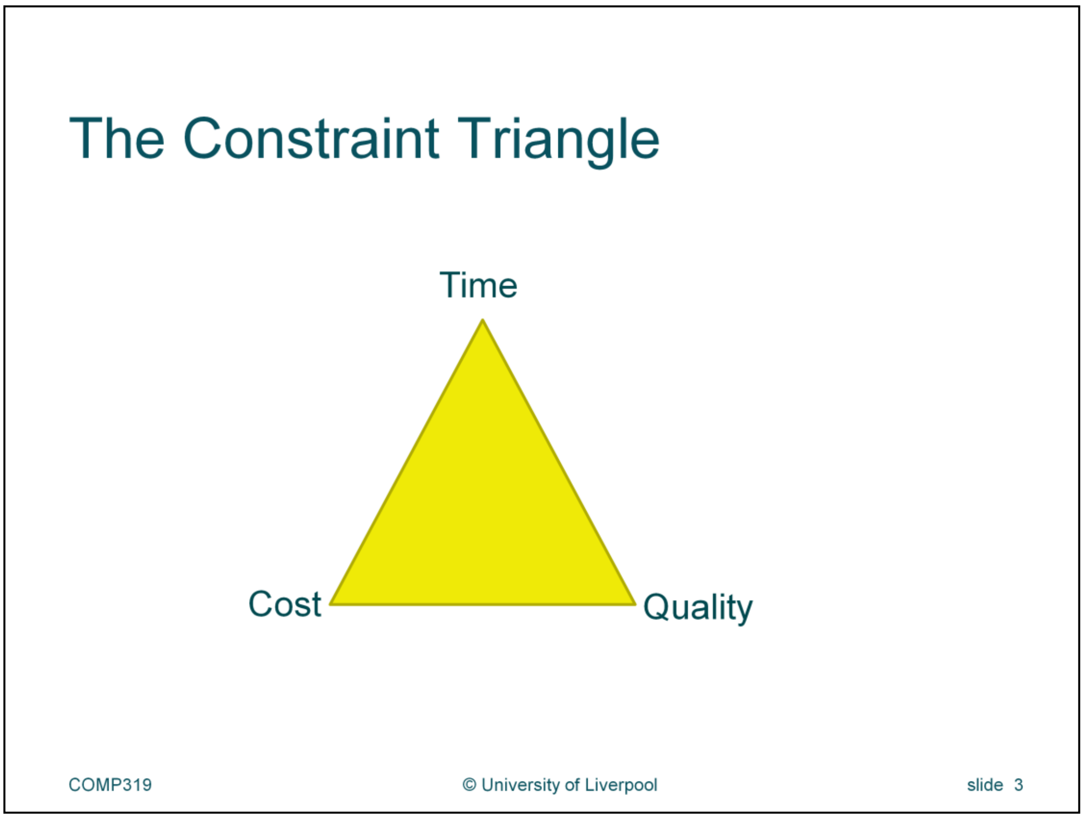


1. “Pick two from three”

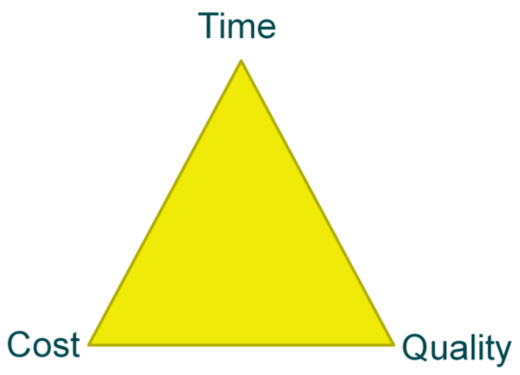
COMP319 © University of Liverpool slide 2

Management of a software project involves managing the resources – crudely these are time, people, and quality.

The tasks involved in project management follow naturally from this. Project planning is the management of all three. Risk management is management of quality. Project scheduling is management of time. People management is management of cost though of course other elements of cost (e.g. equipment) must be considered too.



The Constraint Triangle



COMP319 © University of Liverpool slide 3

If there is one thing that you take away from this course, it should be this diagram.

(Quotes here are from Rudy Rucker's book "Software engineering and computer games", 2003, Addison-Wesley Pearson Education Ltd)

Software engineering is all about working within the constraints imposed by time, cost (usually in terms of people) and quality (in terms of features and/or bugs).

"In a fantasy world we'd like our projects to be done instantly, to cost nothing, and to be of infinitely good quality. However, in the real world we must compromise – it is impossible to achieve all three goals of zero time, zero cost and infinite quality.

You can decrease the time needed but will need to add more staff and/or reduce the quality.

You can reduce the cost by using fewer staff, but the project will take more time and/or reduce in quality

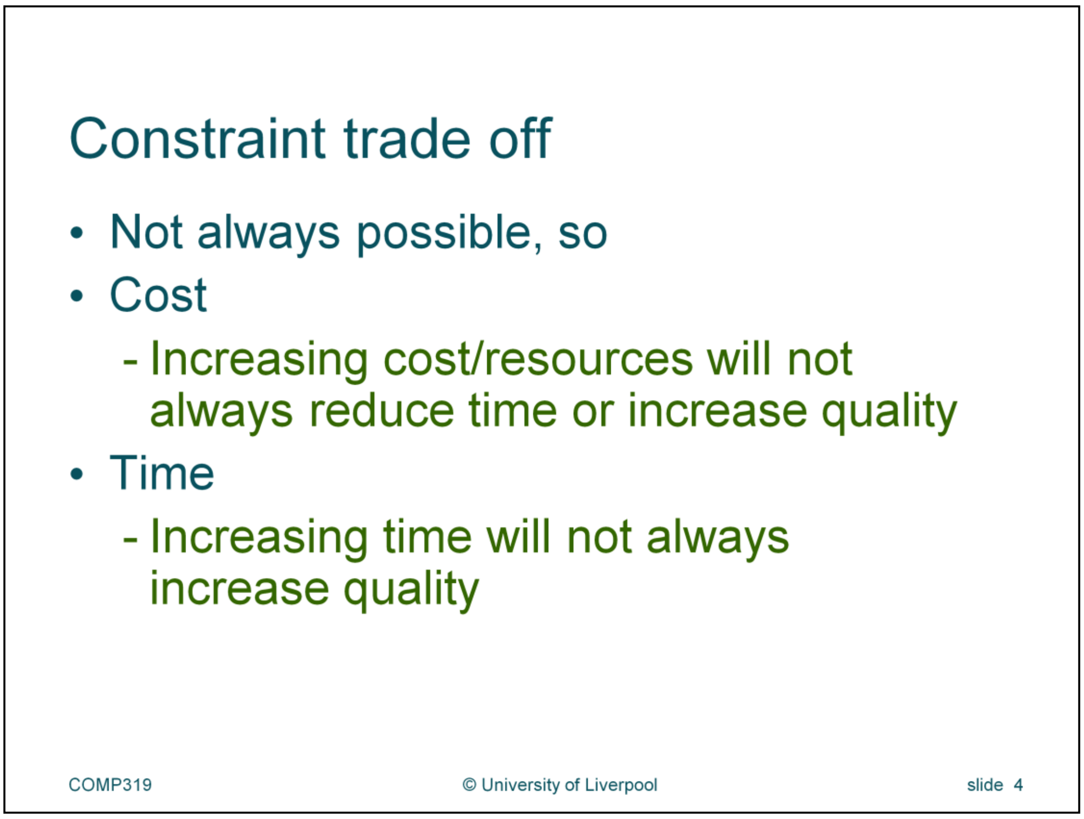
You can opt for high levels of quality, but this means it will cost more and/or take more time.

In general any change in one goal must be compensated for by changes to one or both of the other goals.

If you let your customer (or your manager) specify all three corners of the constraint triangle, your project is doomed to fail."

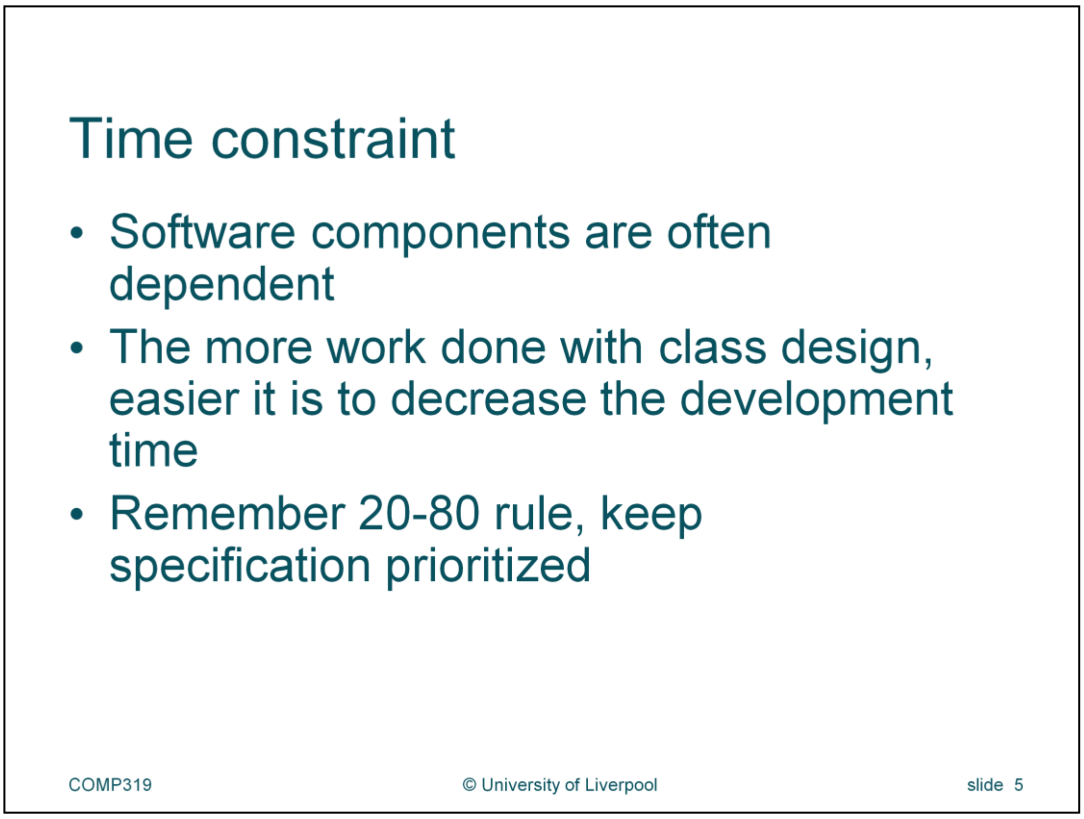
Rucker notes that NASA briefly adopted the slogan ‘Faster, cheaper, better’. However, after a series of disastrous projects the slogan was quietly but quickly abandoned. The lesson they had learned was that such a slogan is impossible to satisfy.

Wags in the software engineering world modified the slogan to ‘Faster, cheaper, better: pick two out of three” …


Constraint trade off

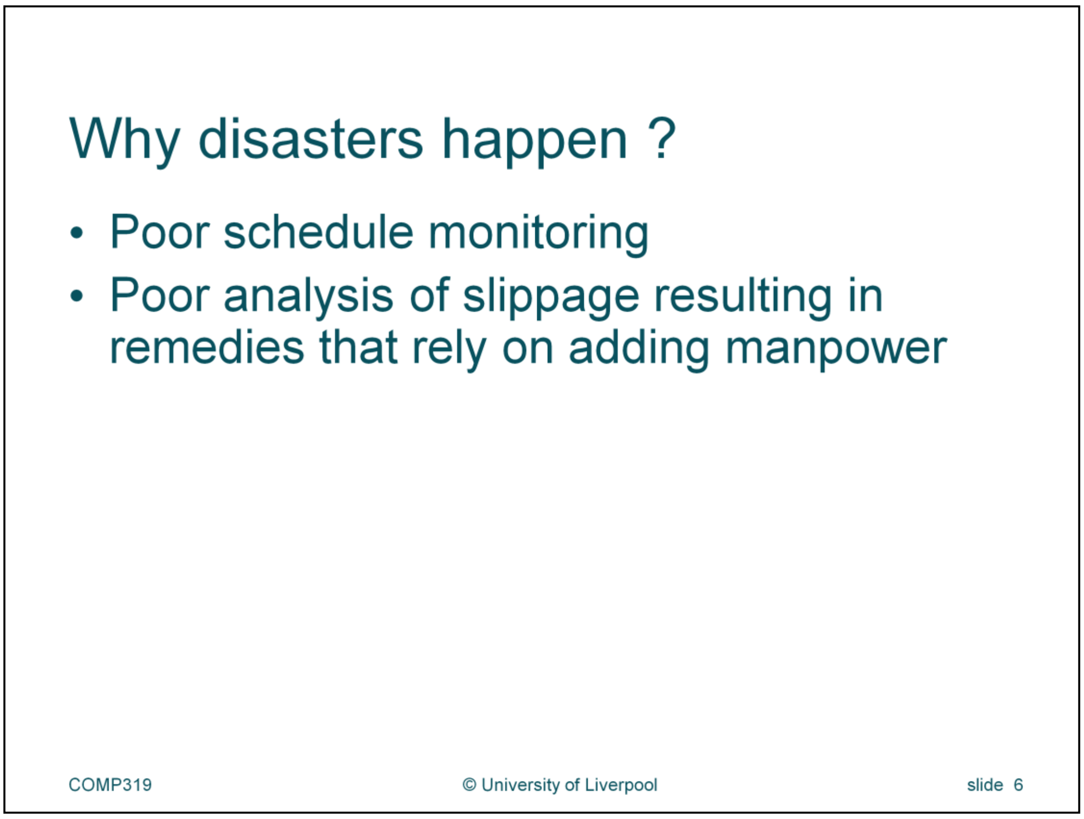
- Not always possible, so
- Cost
  - Increasing cost/resources will not always reduce time or increase quality
- Time
  - Increasing time will not always increase quality

COMP319 © University of Liverpool slide 4


Time constraint

- Software components are often dependent
- The more work done with class design, easier it is to decrease the development time
- Remember 20-80 rule, keep specification prioritized

COMP319 © University of Liverpool slide 5



Why disasters happen ?

- Poor schedule monitoring
- Poor analysis of slippage resulting in remedies that rely on adding manpower

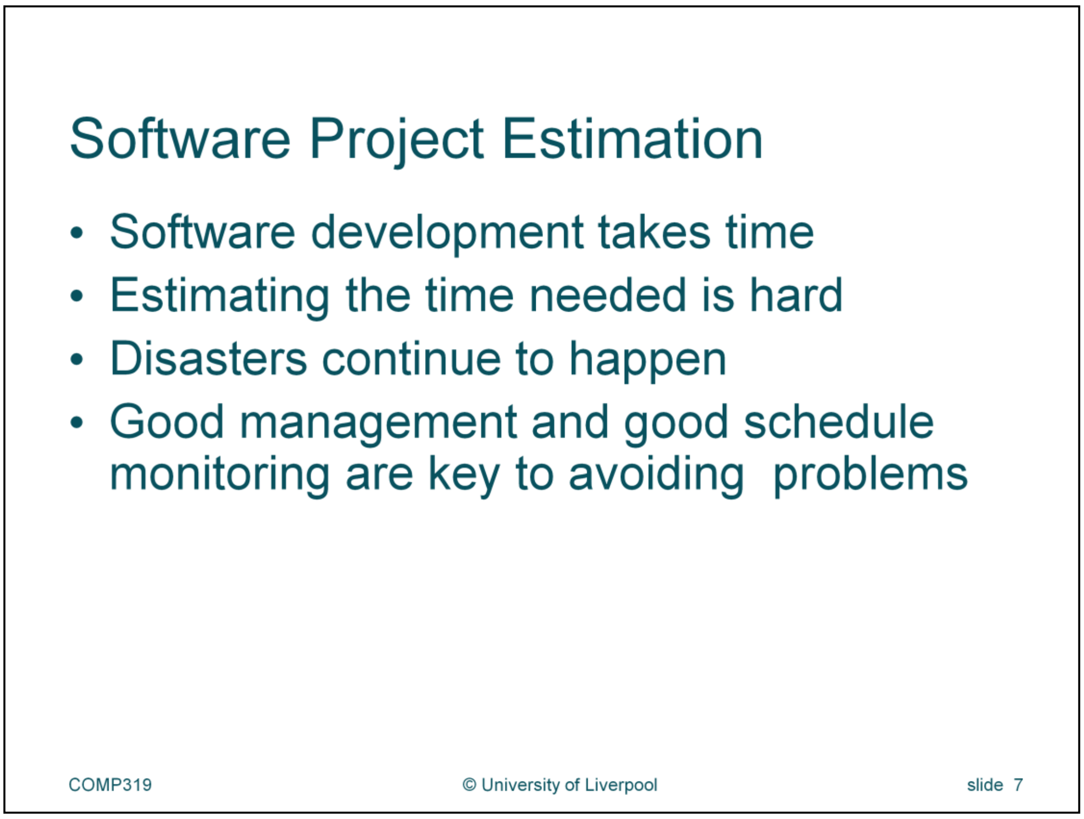
COMP319 © University of Liverpool slide 6

All schedules need to be monitored – failure to monitor the schedules is as bad as not having a schedule in the first place – and leads to disaster.

Similarly, it is no good monitoring that a schedule is slipping if there are no remedies to solve the problem – and for reasons we shall see, just adding manpower is often not a solution. Indeed it may compound the problems.

Finally what should the schedule contain: work blocks; deliverables; tasks; milestones; – or all all of them. Clearly the answer is as many are required for the project in hand.

To monitor the project the schedule one of the most useful is the milestone – a time point where the others (blocks, deliverables and tasks) will be assessed.



Software Project Estimation

- Software development takes time
- Estimating the time needed is hard
- Disasters continue to happen
- Good management and good schedule monitoring are key to avoiding problems

COMP319 © University of Liverpool slide 7

We can see that failure to observe the rules of the constraint triangle leads to software engineering disasters – which we can define as projects over time, over budget, and/or not doing what they should do or being full of bugs.

One way to avoid such problems is to have realistic project estimation and to have reliable strategies in place for when the estimates begin to wobble – which they surely will do.

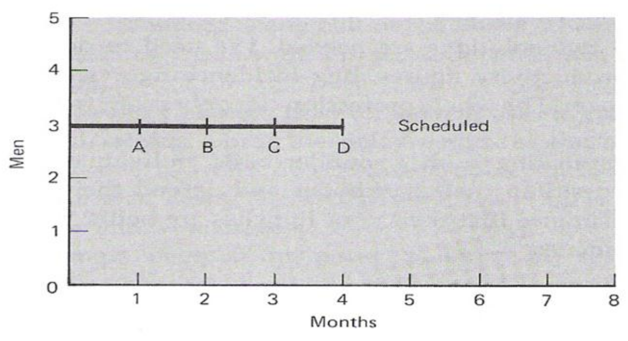
(Put another way, there is a tendency to underestimate the resources required to deliver a particular level of quality).

Last week we noted that software development was hard to do because of the inherent complexity and for this reason developing software takes time.

Because of the complexity we get unpredictability; and for this reason estimating project costs is also hard to do.

To avoid the disasters it is necessary to have good (honest) management, to develop realistic time schedules, to have extensive project monitoring in place and some reliable strategies to resolve the inevitable creeping delays and costs that find their way into the project.

Schedule slippage



COMP319 © University of Liverpool slide 8

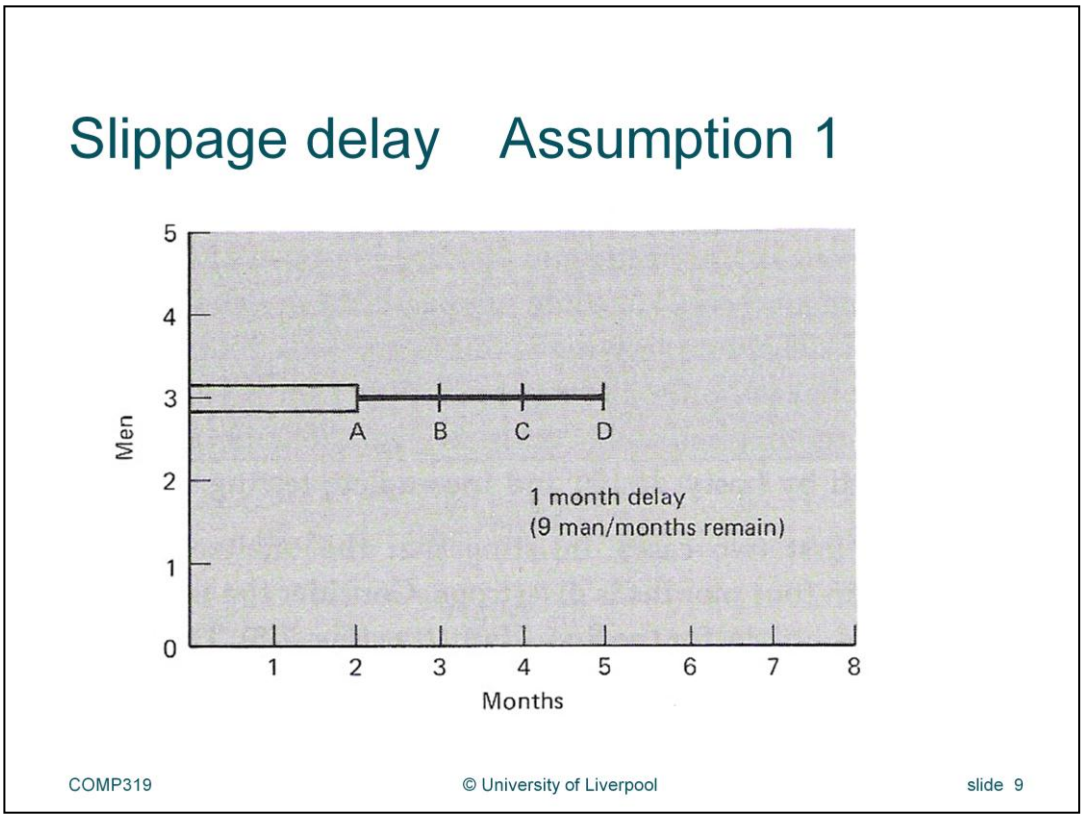
Imagine this schedule: (taken from Brooks MMM, p22-26)

There is a task estimated to take 12 man months.

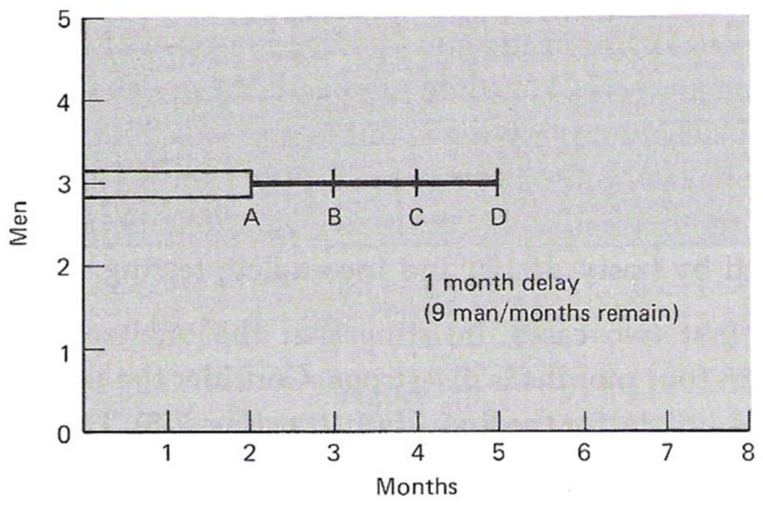
It is assigned to 3 men for four months.

There are 4 mileposts, falling at the end of each month.

Note that months are along the x-axis, and men allocated to the project along the y-axis.



Slippage delay Assumption 1



COMP319 © University of Liverpool slide 9

Here we illustrate that the the first milepost is not reached until two months have elapsed.

The open line indicates the past, the simple line the future.

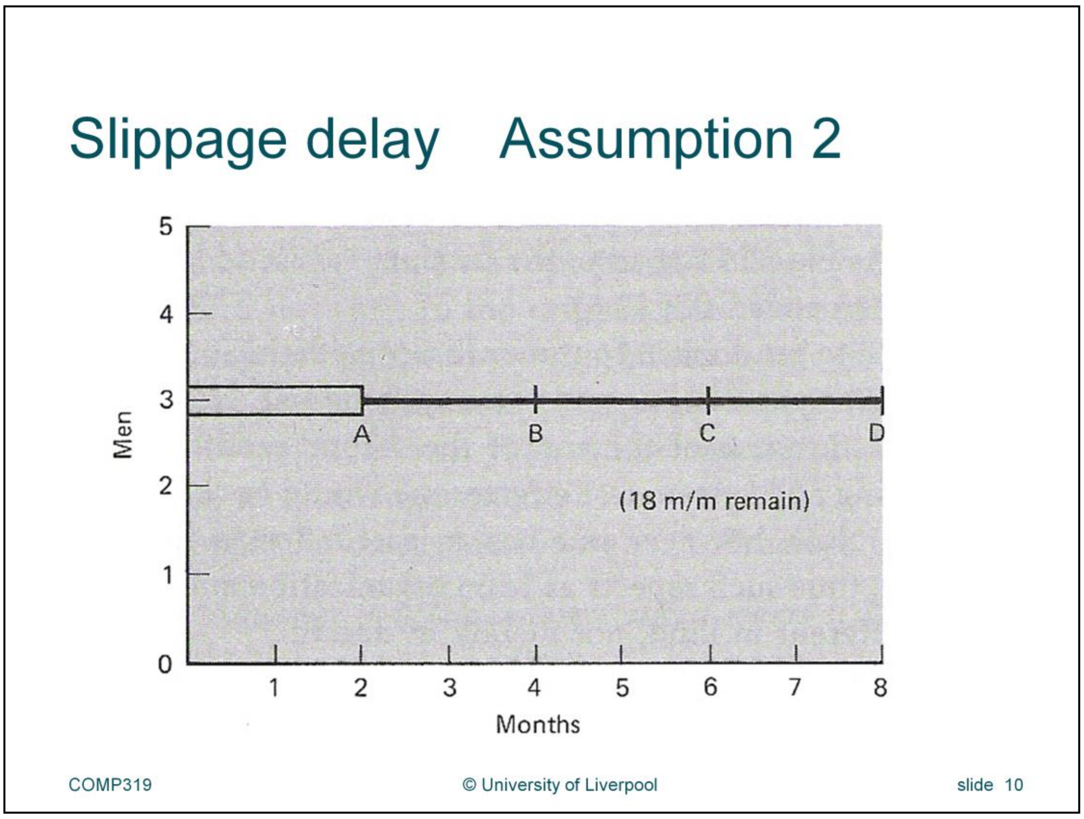
There has been slippage of one month.

Notice, incidentally, that because of the milestone we have at least spotted the slippage.

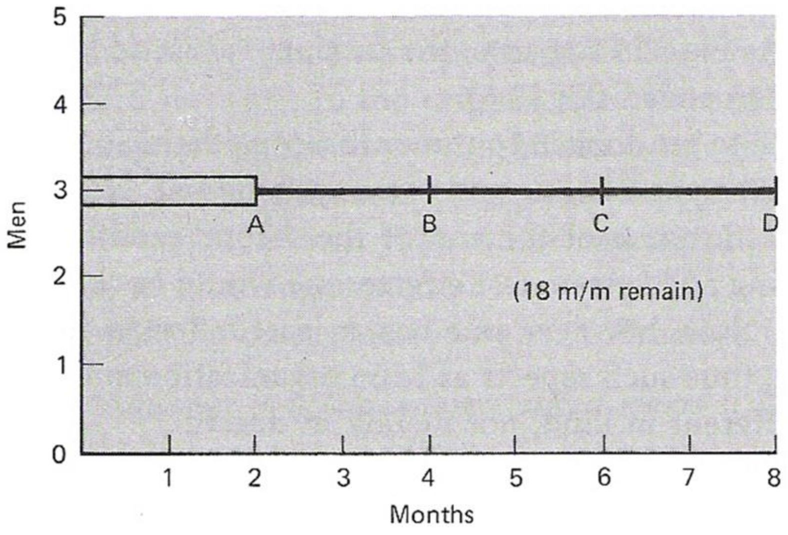
What do we do ? and

What strategies are available to us to remedy the situation ?

Assume that the task must be done on time. Assume that only the first part of the task was miss-estimated so the figure tells the story accurately. This is one months delay for 3 men leaving 9 man months of estimated effort remaining and 2 months for completion. 9/2 = 5 men are now required to complete the task, that is 2 extra.

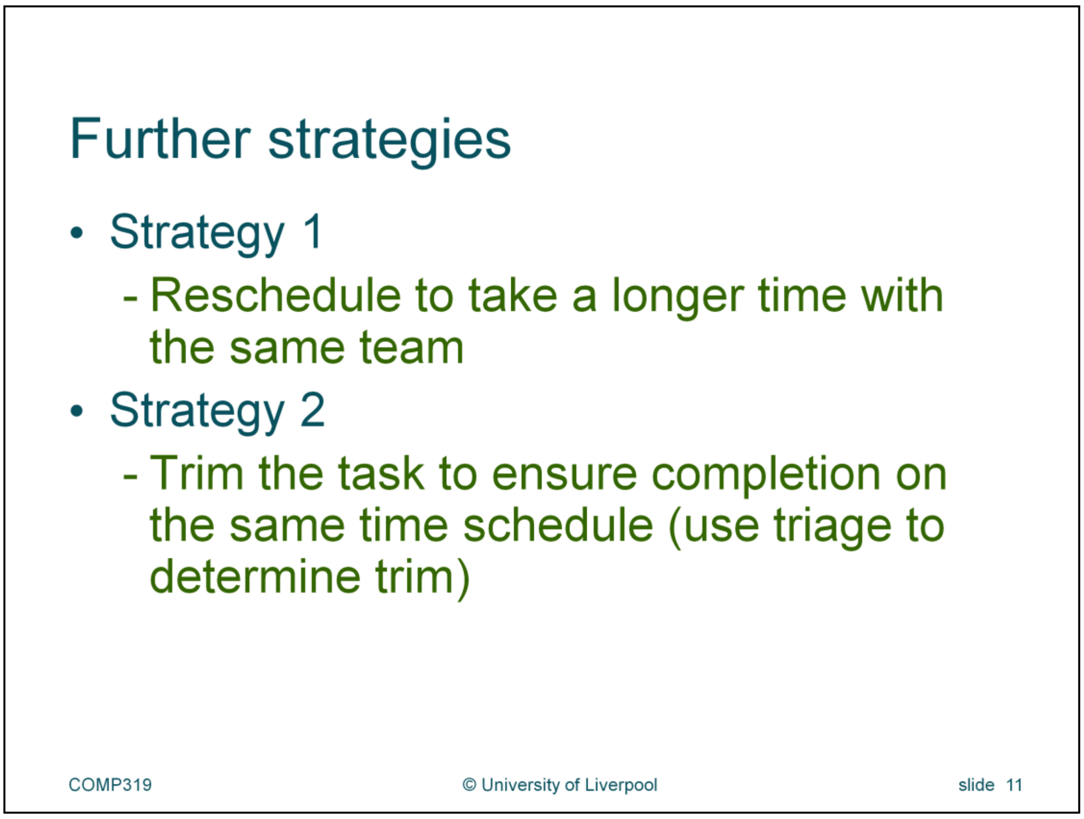


Slippage delay Assumption 2



COMP319 © University of Liverpool slide 10

2) Assume that the task must be done on time. Assume that the whole estimate was uniformly low, so that the figure above is really the situation. That is, it has taken 3 men two months (6 man months) to get here, there are 3 phases left requiring 6*3 = 18 man months of effort. The time left is 2 months, thus 18/2 = 9 men are required; we need to add 6 to the 3 already on the project.

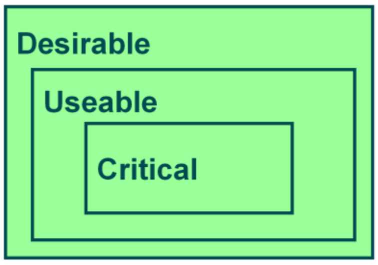

Further strategies

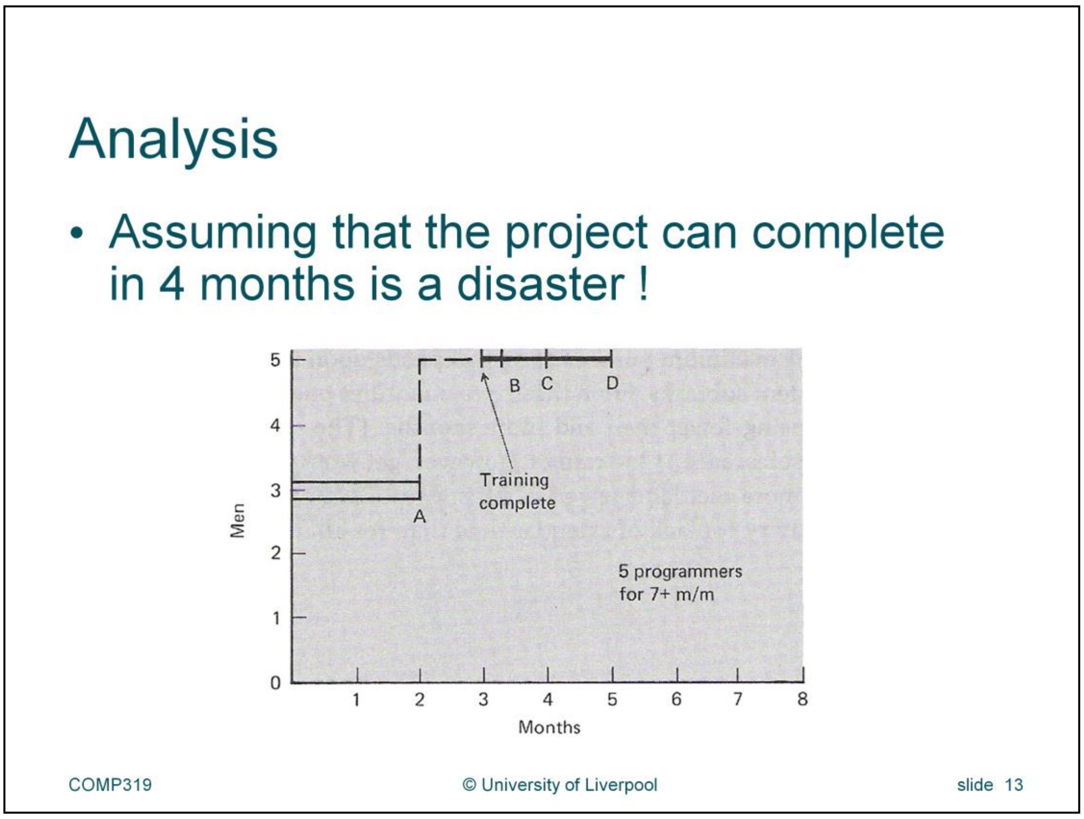
- Strategy 1
  - Reschedule to take a longer time with the same team
- Strategy 2
  - Trim the task to ensure completion on the same time schedule (use triage to determine trim)

COMP319 © University of Liverpool slide 11

Triage

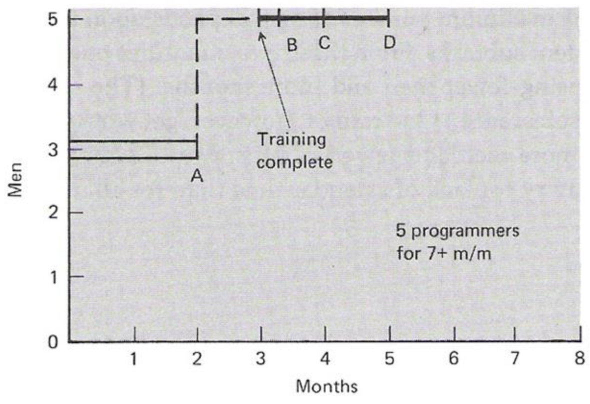
- Feature triage
  - Must do, good to do, nice to do
- Testing/debug triage
  - Must fix, good to fix, nice to fix





Analysis

- Assuming that the project can complete in 4 months is a disaster !



COMP319 © University of Liverpool slide 13

The problem is that there is a lag in implementing any solution.

In this case the assumption that we must complete in 4 months is disastrous. Adding people, however talented and assuming rapid recruitment … will divert one of the existing people (say, for a month) wasting his time and one man months effort from each of the new men. (either 3 or 7 m/m).

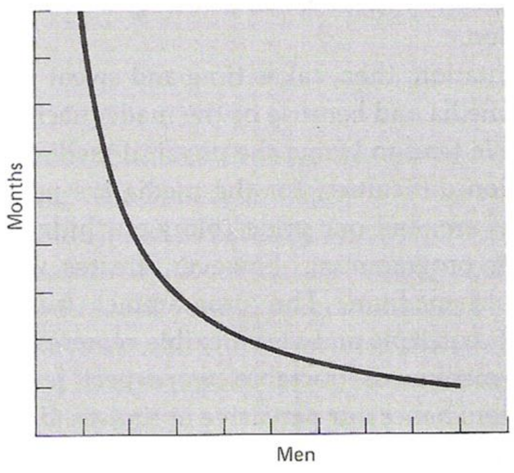
Further, work already done will be lost and extra work will need to be done by way of testing because more men are now working on the project, and introducing bugs. One solution might be to add yet more men. Using the assumptions of Strategy 1 a total of 7 men would now be required; given training time, etc … the project would be as late as if no further men were added.

The same is true, with even more disastrous cost implications if Strategy 2 were used.

This leads to Brooks' Law: ***Adding manpower to a late software project makes it later***.

Doing the diagrams is one way to appreciate this. But what are the problems.

Before moving on; it is important to note that working with milestones in this way does not need to affect the other aspects of the planning namely; the work packages, tasks, and deliverables involved but often does so – simply because of the nature of the constraint diagram.



Time versus number of workers—perfectly partitionable task

COMP319 © University of Liverpool slide 14

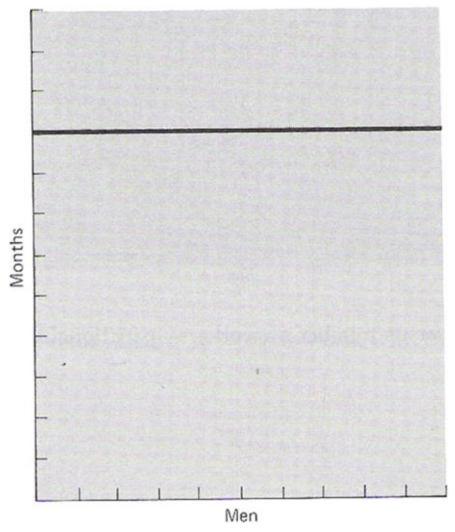
The first problem is that while cost does vary as the product of the number of men and the number of months … project progress does not.

The man-month as a unit for measuring the size of a job is dangerous, and worse, deceptive.

Men and months are interchangeable only when a task can be partitioned with the assumption that there is no communication among the workers.

True of (say) gathering crops not even approximately true in software engineering.

Sequential constraints



Time versus number of workers—unpartitionable task

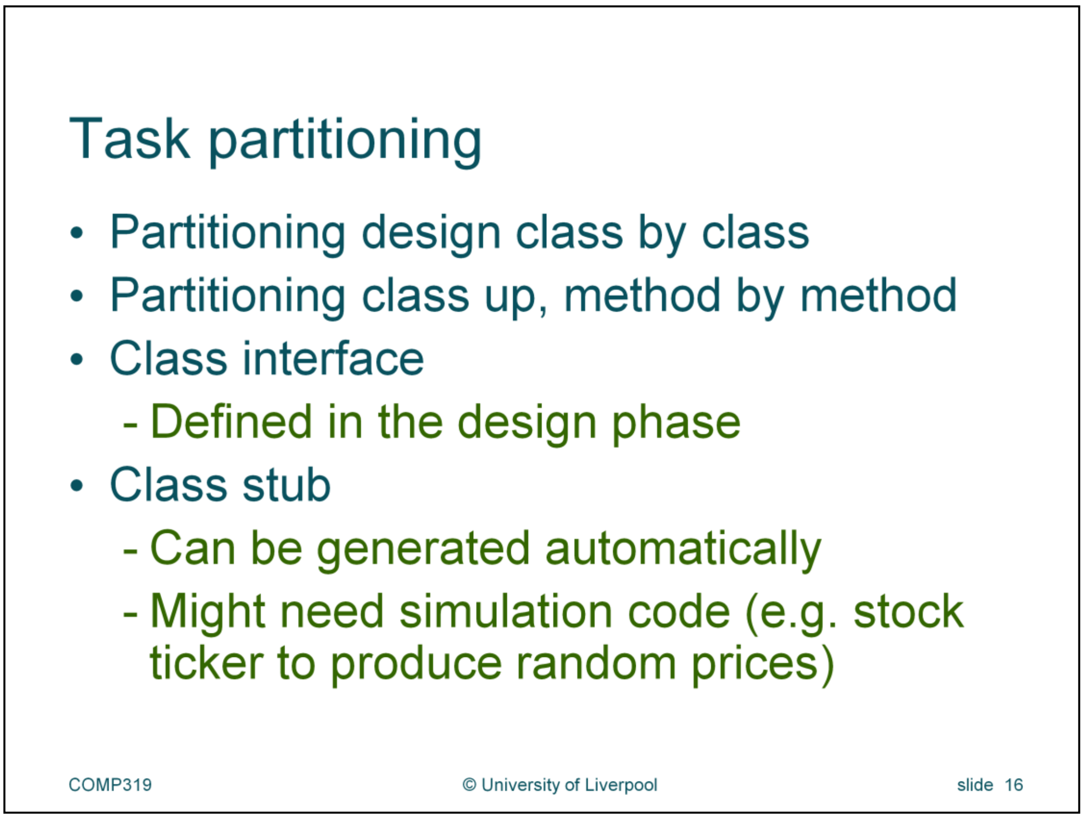
COMP319 © University of Liverpool slide 15

When a task cannot be partitioned, because of sequential constraints, the time versus men graph is a flatline, the application of more effort has no effect on the schedule.

True of childbirth which takes 9 months, and true of many software engineering tasks such as debugging, because of the sequential nature of the task.

Thus, it is important to know how sequential the task is. If it is highly sequential, then it is highly constrained in terms of time required.

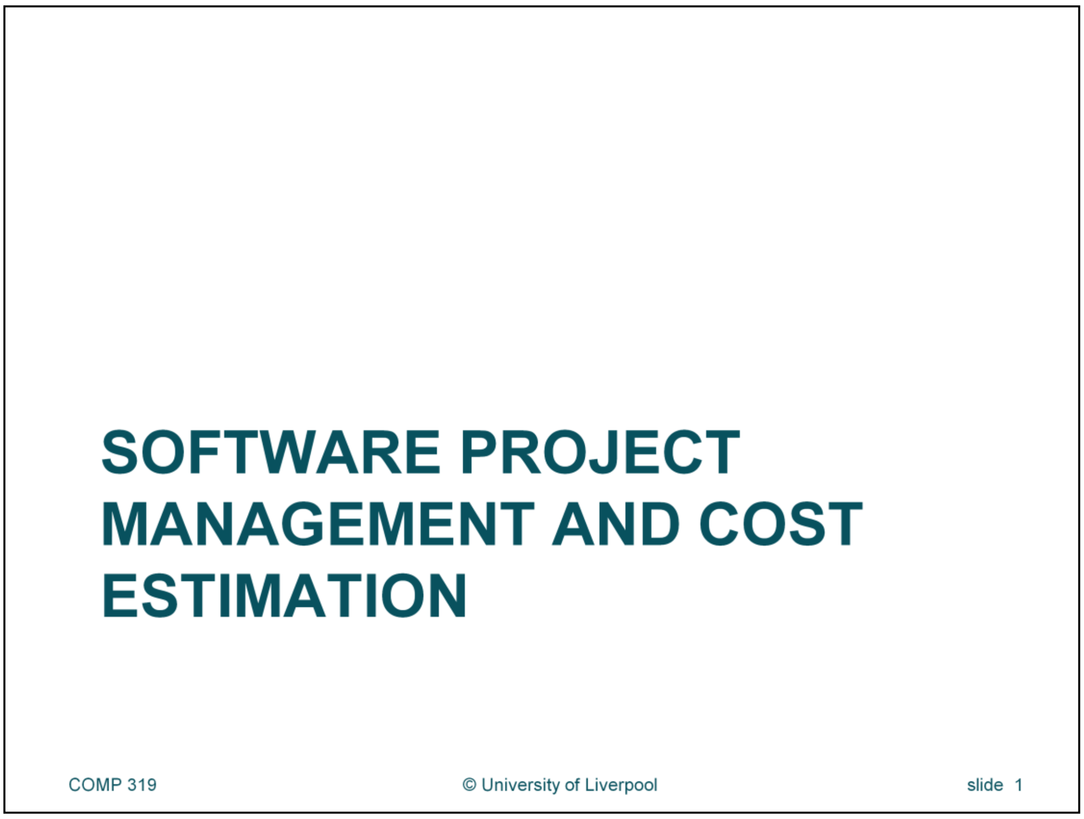
Partitioning tasks is evidently desirable so that more development effort can be added to the project if its running late.



Task partitioning

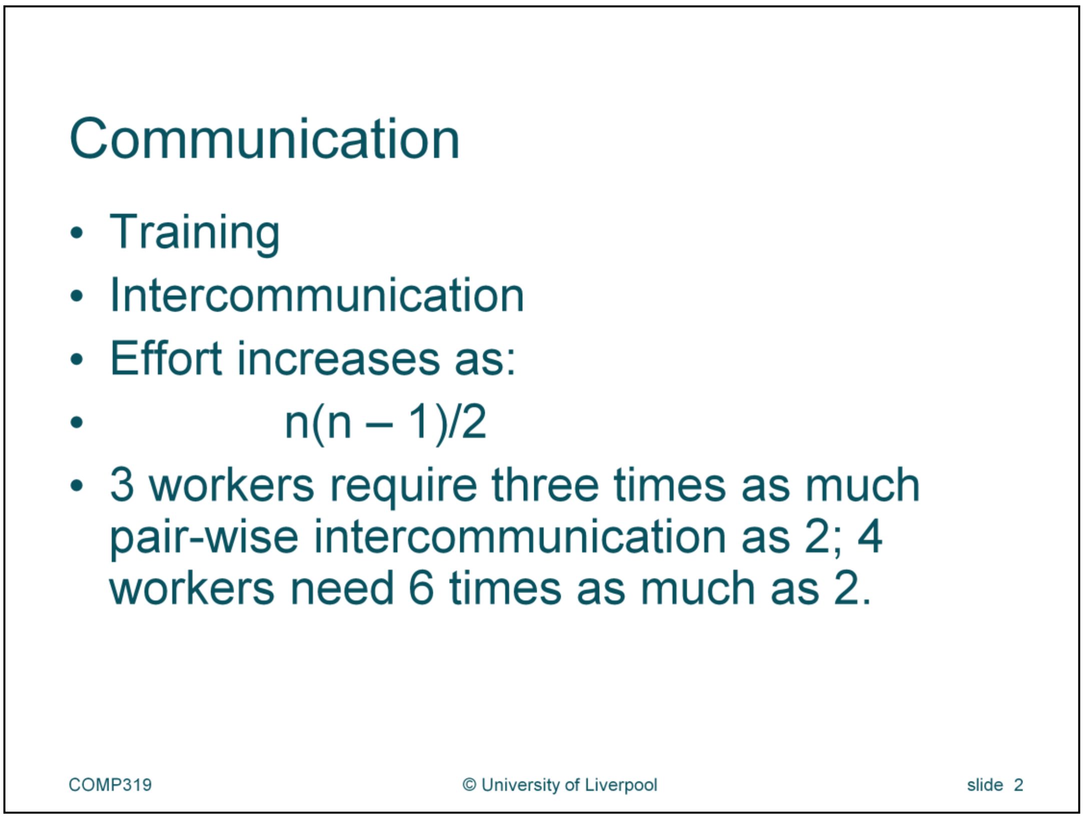
- Partitioning design class by class
- Partitioning class up, method by method
- Class interface
  - Defined in the design phase
- Class stub
  - Can be generated automatically
  - Might need simulation code (e.g. stock ticker to produce random prices)

COMP319 © University of Liverpool slide 16



SOFTWARE PROJECT MANAGEMENT AND COST ESTIMATION

COMP 319 © University of Liverpool slide 1



Communication

- Training
- Intercommunication
- Effort increases as:
- $n(n - 1)/2$
- 3 workers require three times as much pair-wise intercommunication as 2; 4 workers need 6 times as much as 2.

COMP319 © University of Liverpool slide 2

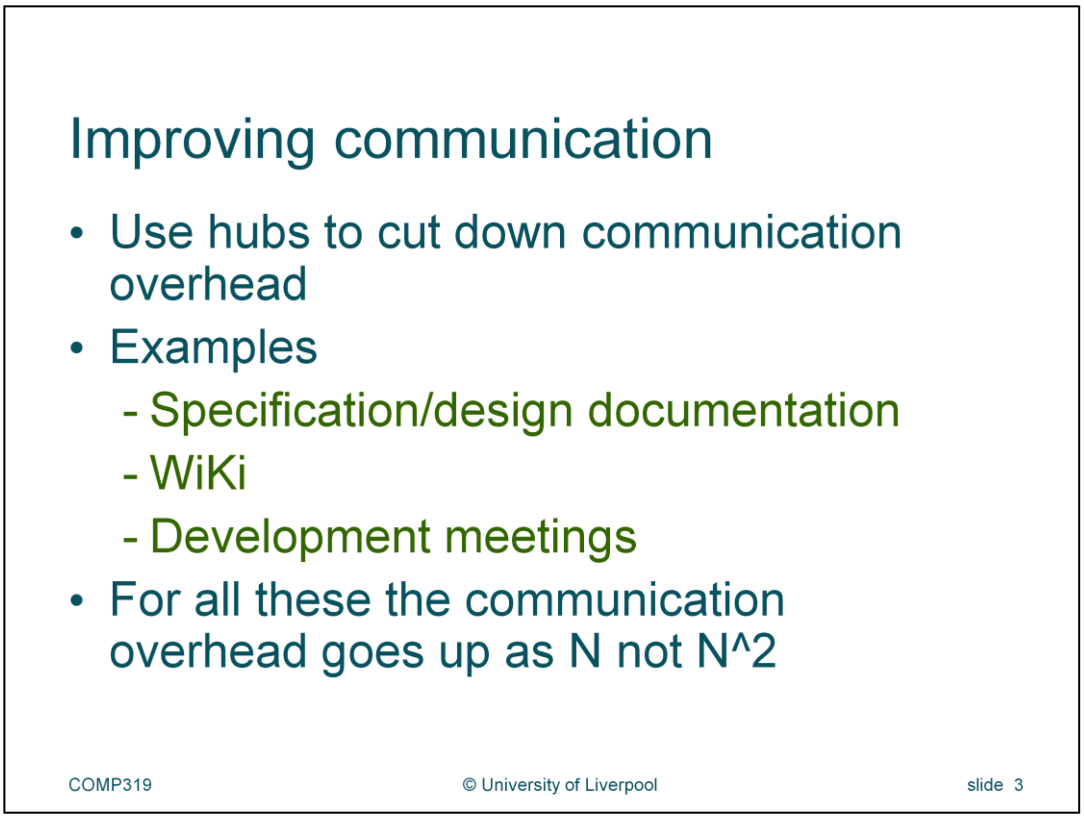
Communication is in two parts: training and intercommunications

Training is required: in the technology, the goals of the project, the overall strategy and the plan of work. Training cannot be partitioned each new man must go through it individually.

Intercommunication is required in more complex tasks such as software engineering. It assumes that to complete the task workers must have pairwise communications.

If each part of the task must be separately co-ordinated with every other part the effort required increases in a complex way.

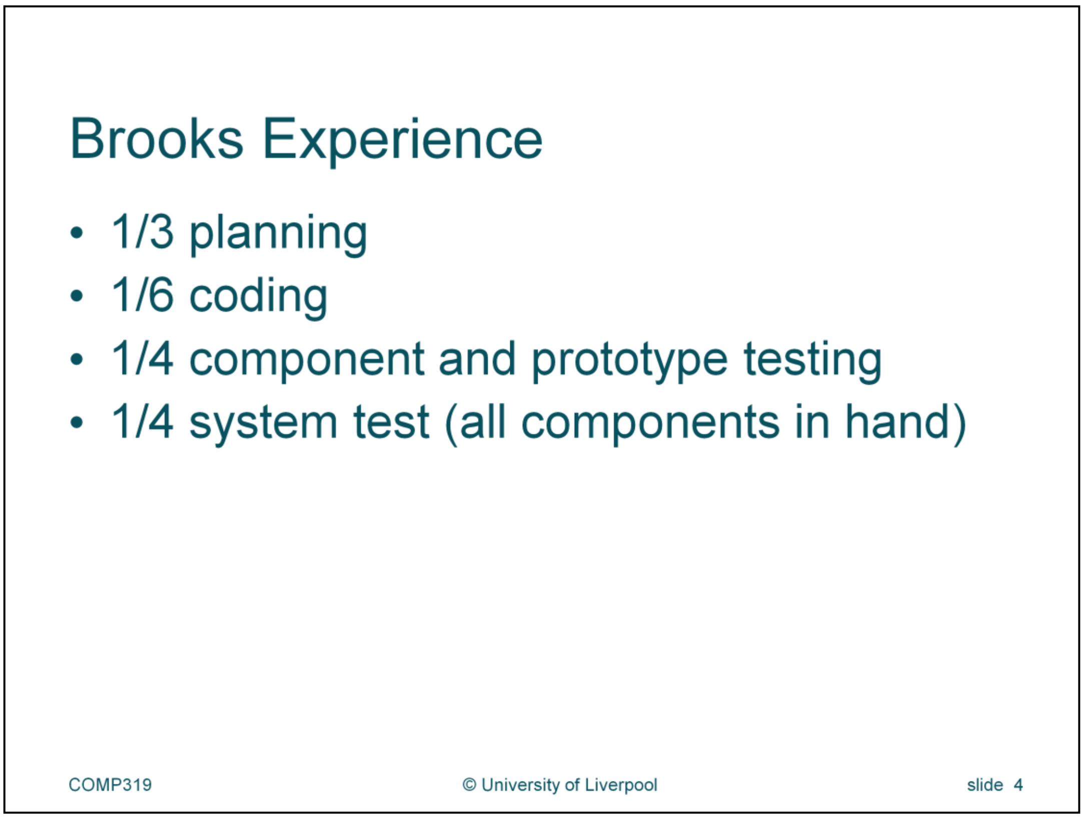
Software construction is inherently a team effort and the communication time required is one that cannot be ignored and quickly dominates any partitioning strategy and project costing.



Improving communication

- Use hubs to cut down communication overhead
- Examples
  - Specification/design documentation
  - WiKi
  - Development meetings
- For all these the communication overhead goes up as N not N^2

COMP319 © University of Liverpool slide 3


Brooks Experience

- 1/3 planning
- 1/6 coding
- 1/4 component and prototype testing
- 1/4 system test (all components in hand)

COMP319 © University of Liverpool slide 4

The first question is what are we guessing at … well it's time. More particularly, it is time to write some standard number of lines of code, or some function (as a subroutine or sub-process). This is fearsomely complicated because it involves target language, methodology, skills involved, learning required, etc. etc.

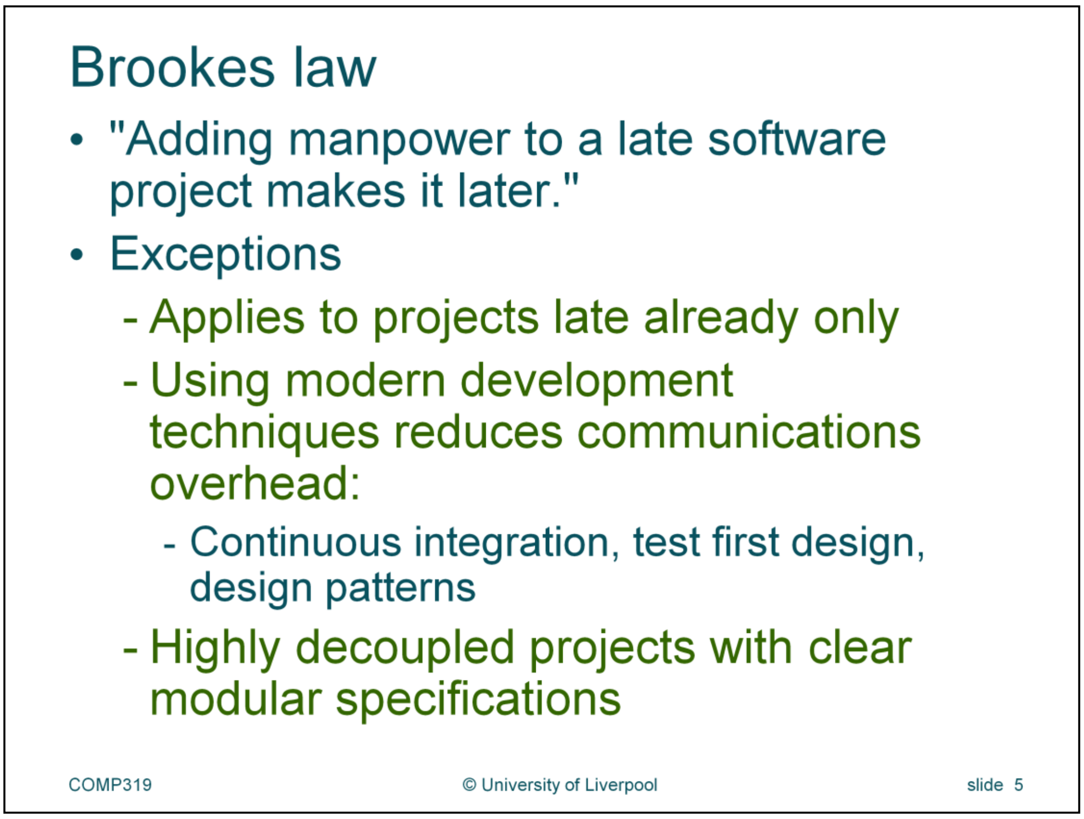
Another problem is, what is the goal. Is it the completed prototype?, the alpha release (alpha-1 is considered the first field testable version), or the beta release (beta-1 is the first released version).

Brooks, in MMM presents his rules of thumb for scheduling a software task as about half on debugging of completed code and about 1/6 on coding (the bit which is easiest to estimate).

Planning is given 1/3 of the total time, however, he notes that this does not include research time or exploration of new techniques. Where the learning curve is steep (e.g. in honours projects ? This part can consume more than 50% of the time and lead to nothing being delivered …)

The other key point of experience is that failure to allow enough time for testing is particularly disastrous. Because, this comes at the end of the schedule near the delivery milestone, slippage affects customers and badly reflects on the project as a whole.

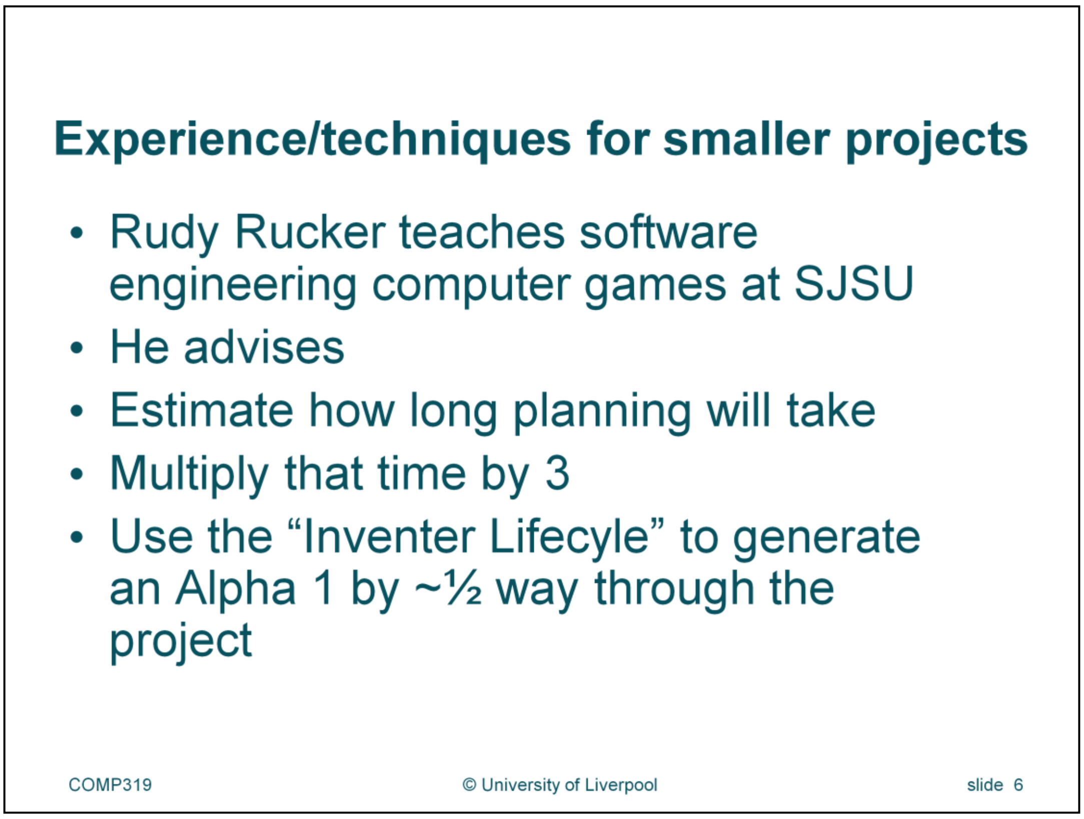
We thus see the tricky balance between design at the beginning taking too much time – and full system testing at the end.



Brookes law

- "Adding manpower to a late software project makes it later."
- Exceptions
  - Applies to projects late already only
  - Using modern development techniques reduces communications overhead:
    - Continuous integration, test first design, design patterns
  - Highly decoupled projects with clear modular specifications

COMP319 © University of Liverpool slide 5



Experience/techniques for smaller projects

- Rudy Rucker teaches software engineering computer games at SJSU
- He advises
- Estimate how long planning will take
- Multiply that time by 3
- Use the “Inventer Lifecyle” to generate an Alpha 1 by ~½ way through the project

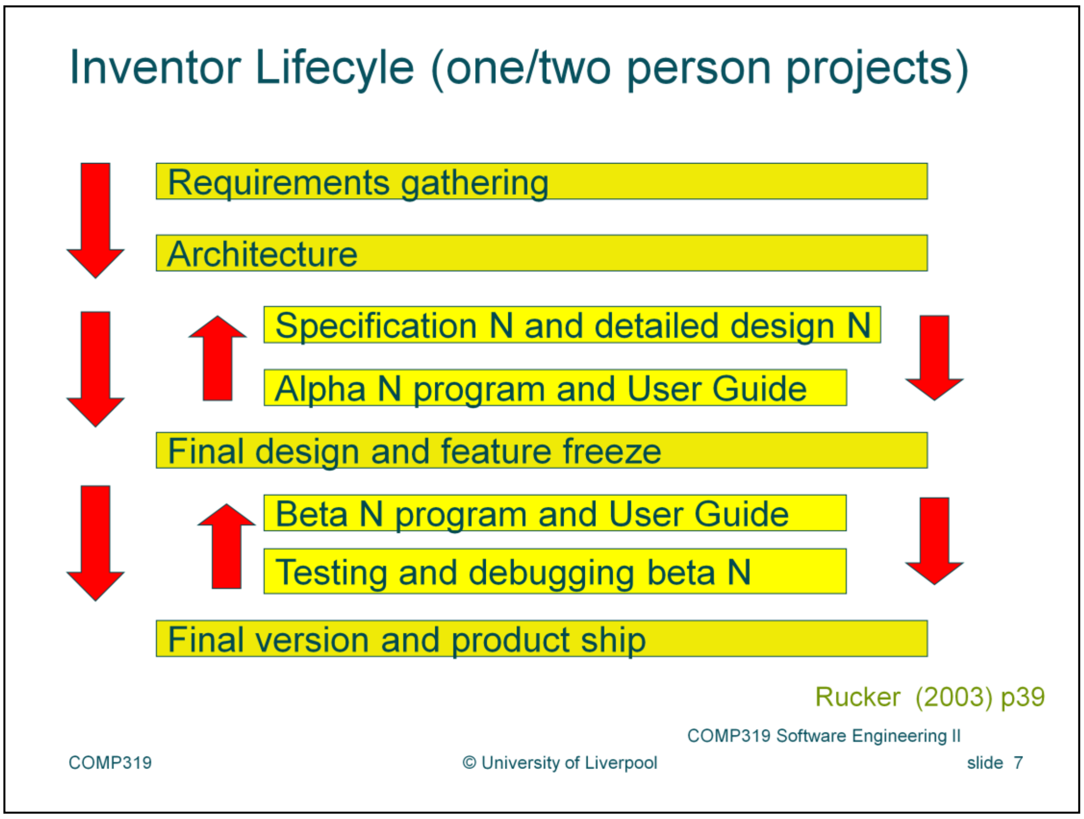
COMP319 © University of Liverpool slide 6

Rudy Rucker (San Jose State University) graduate level course. His book (“Software engineering and computer games”, 2003, Addison-Wesley Pearson Education Ltd) is a good one if: a) you are a student; b) you are embarking on a software project to be completed with about 2-3 man months of effort (i.e. roughly an honours project); c) you are planning to write software games.

Rucker develops Brooks idea of using a 1/3 of the time for planning, by suggesting that planning the architecture (essentially the user interface – which we will return to later in the module) should be the goal. He notes that time spent on good architecture will save time in coding and debugging.

He also suggests not doing the waterfall thing, nor really the spiral thing, but a much more concrete thing based on producing small field releases until a final design and feature freeze can happen.

The calls this the “Inventor Lifecyle” … but note this is for students on short 1 to 4 man projects.



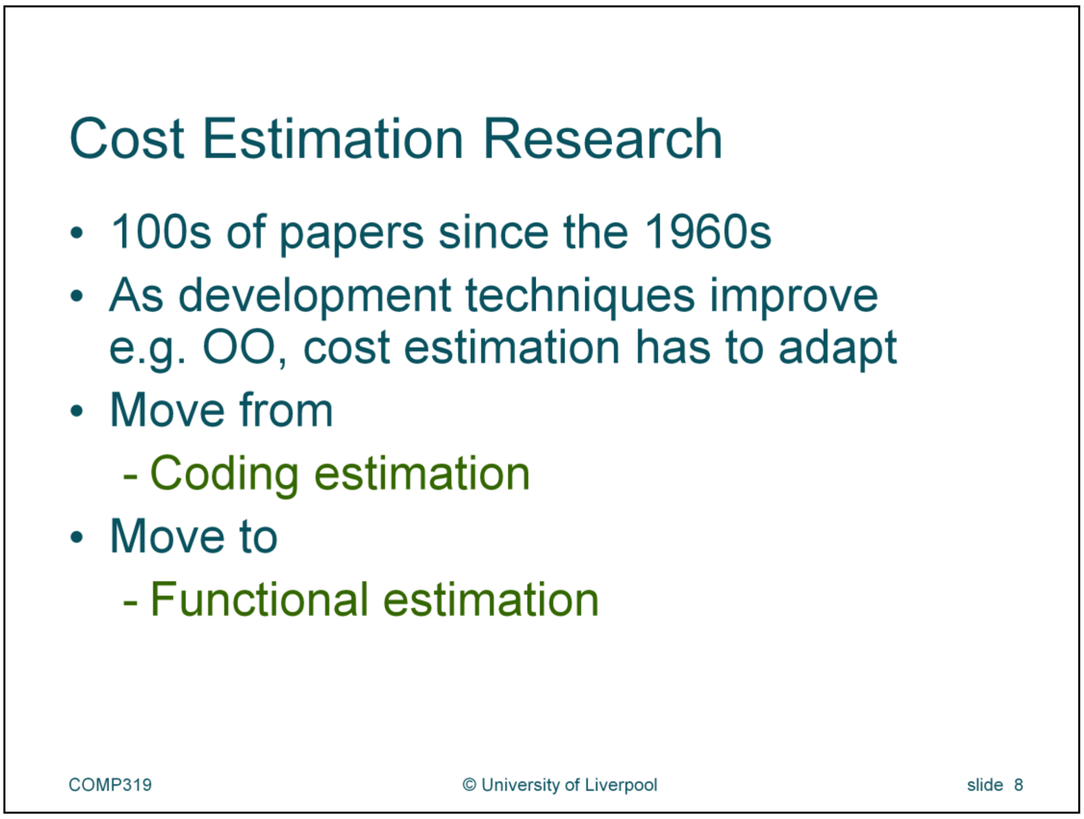
Inventor Lifecyle (one/two person projects)

Requirements gathering
Architecture
Specification N and detailed design N
Alpha N program and User Guide
Final design and feature freeze
Beta N program and User Guide
Testing and debugging beta N
Final version and product ship

Rucker (2003) p39

COMP319 Software Engineering II

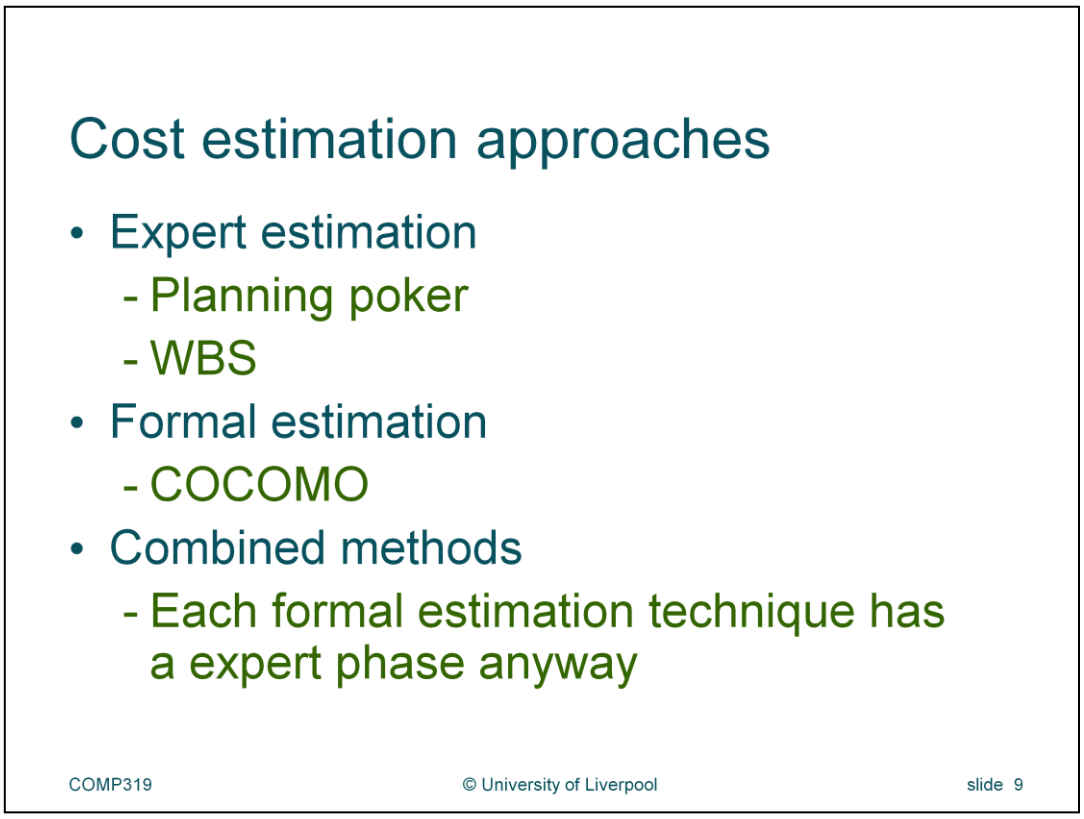
COMP319 © University of Liverpool slide 7



Cost Estimation Research

- 100s of papers since the 1960s
- As development techniques improve e.g. OO, cost estimation has to adapt
- Move from
  - Coding estimation
- Move to
  - Functional estimation

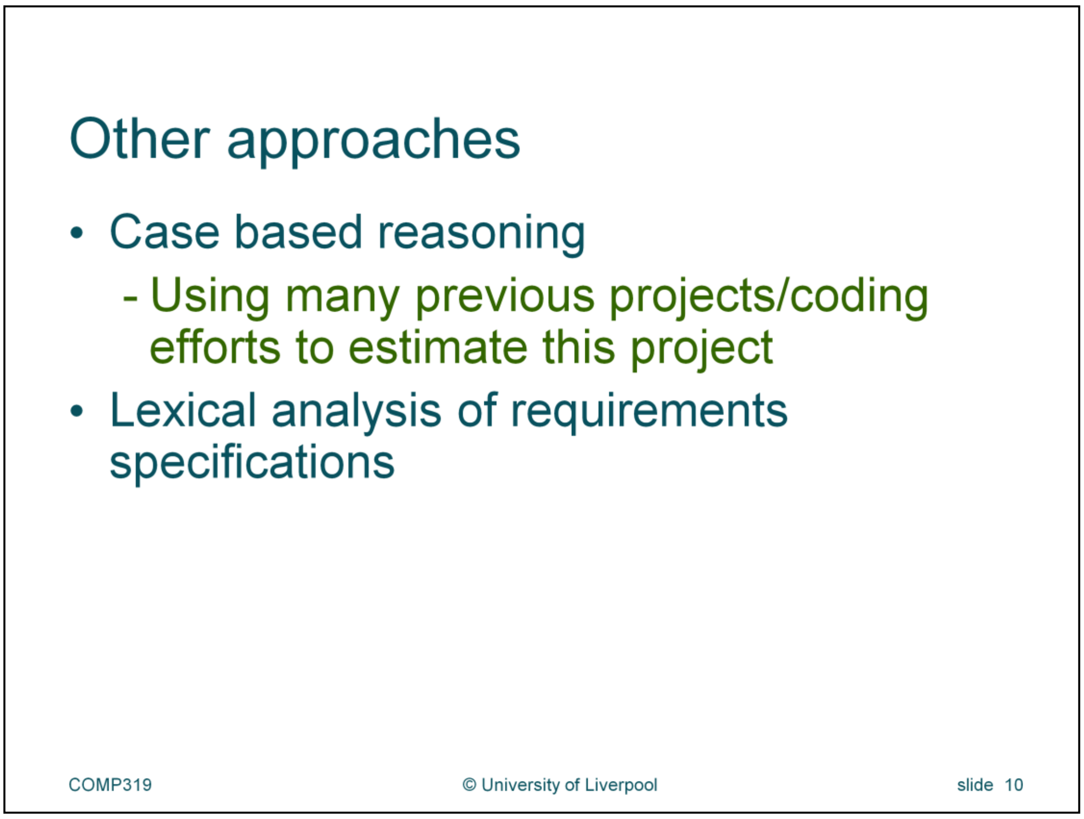
COMP319 © University of Liverpool slide 8



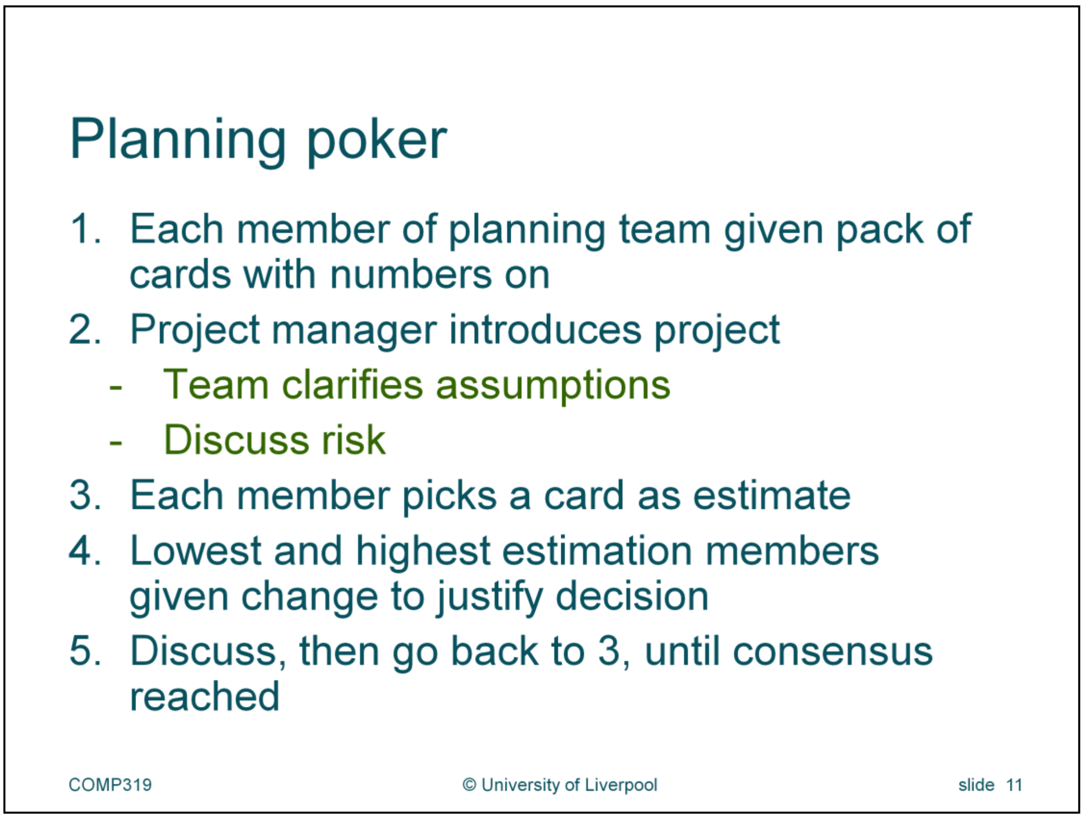
Cost estimation approaches

- Expert estimation
  - Planning poker
  - WBS
- Formal estimation
  - COCOMO
- Combined methods
  - Each formal estimation technique has a expert phase anyway

COMP319 © University of Liverpool slide 9

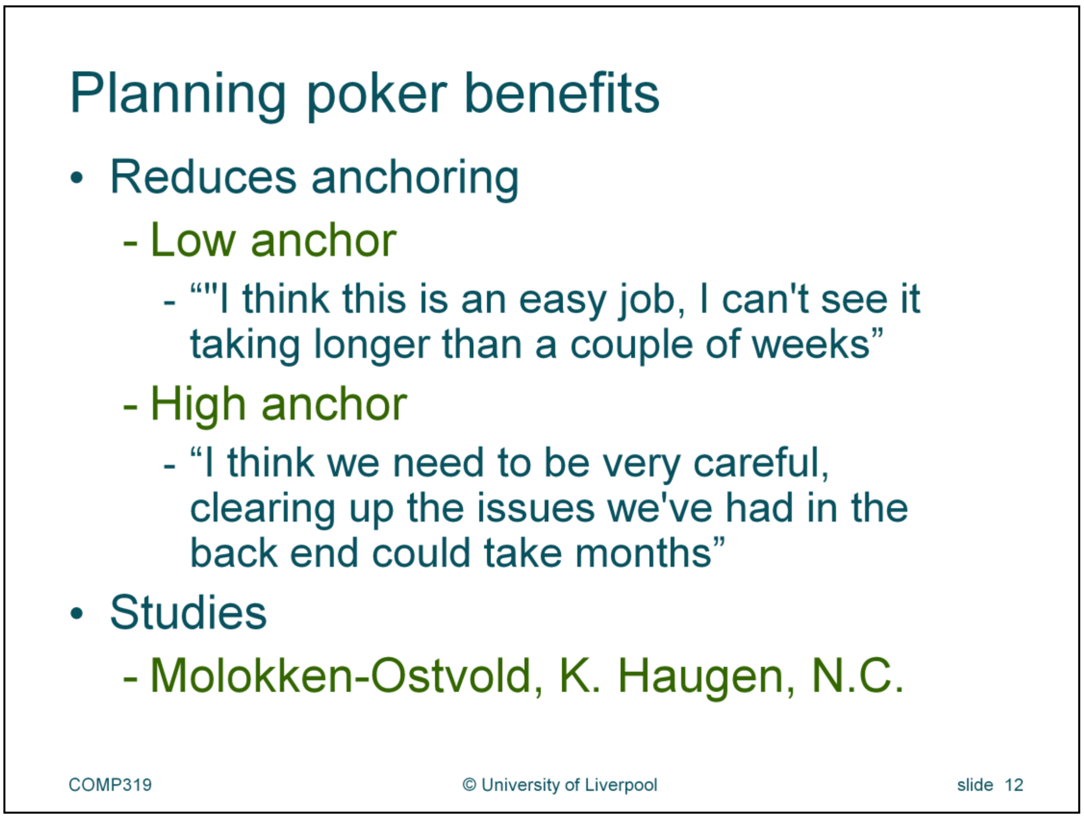

Other approaches

- Case based reasoning
  - Using many previous projects/coding efforts to estimate this project
- Lexical analysis of requirements specifications



Planning poker

1. Each member of planning team given pack of cards with numbers on
2. Project manager introduces project
   - Team clarifies assumptions
   - Discuss risk
3. Each member picks a card as estimate
4. Lowest and highest estimation members given change to justify decision
5. Discuss, then go back to 3, until consensus reached



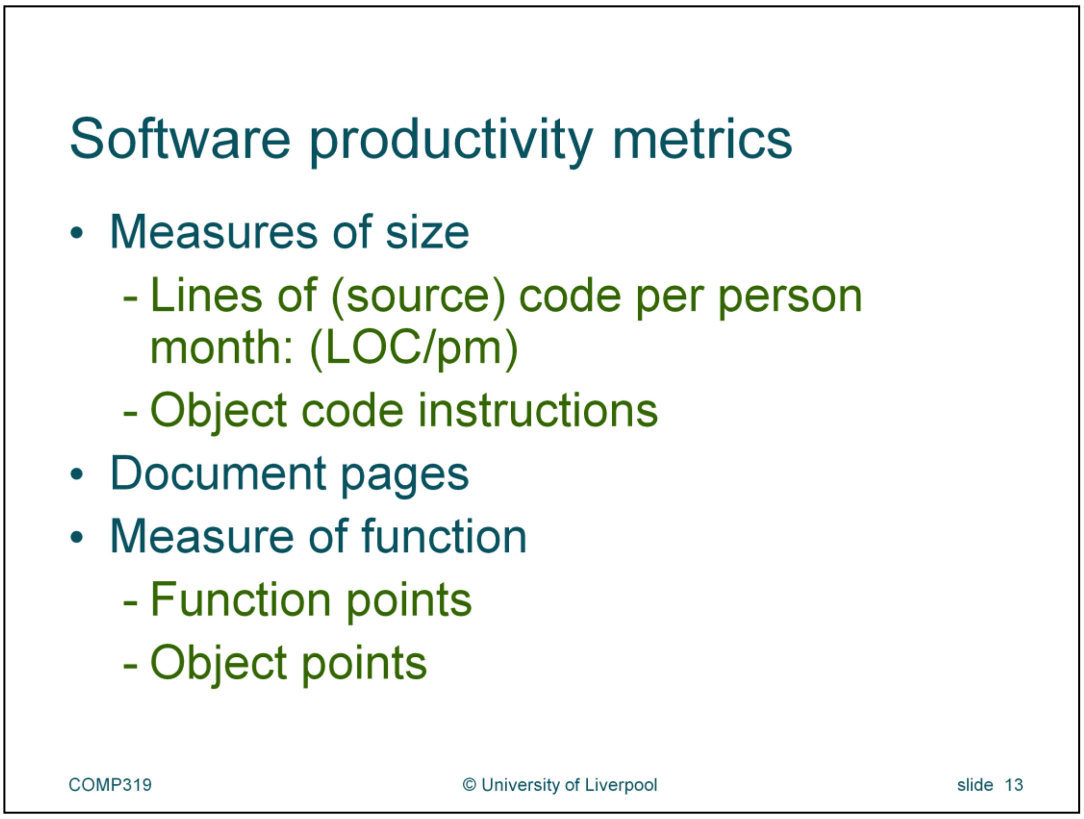
Planning poker benefits

- Reduces anchoring
  - Low anchor
    - “"I think this is an easy job, I can't see it taking longer than a couple of weeks”
  - High anchor
    - “I think we need to be very careful, clearing up the issues we've had in the back end could take months”
- Studies
  - Molokken-Ostvold, K. Haugen, N.C.

COMP319 © University of Liverpool slide 12

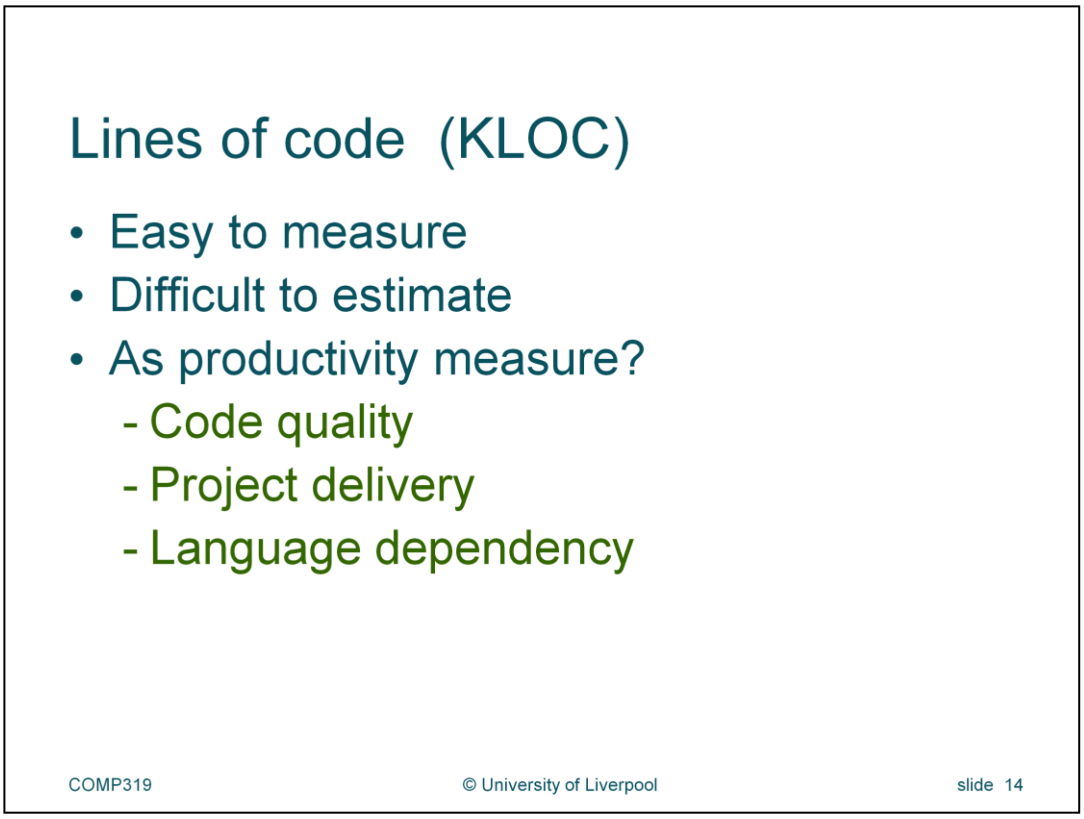
Molokken-Ostvold, K. Haugen, N.C. (13 April 2007). "Combining Estimates with Planning Poker--An Empirical Study". IEEE

This showed the planning poker technique produced estimates were less optimistic and more accurate than estimates obtained through other approaches such as mechanical combination of individual estimates for the same tasks.

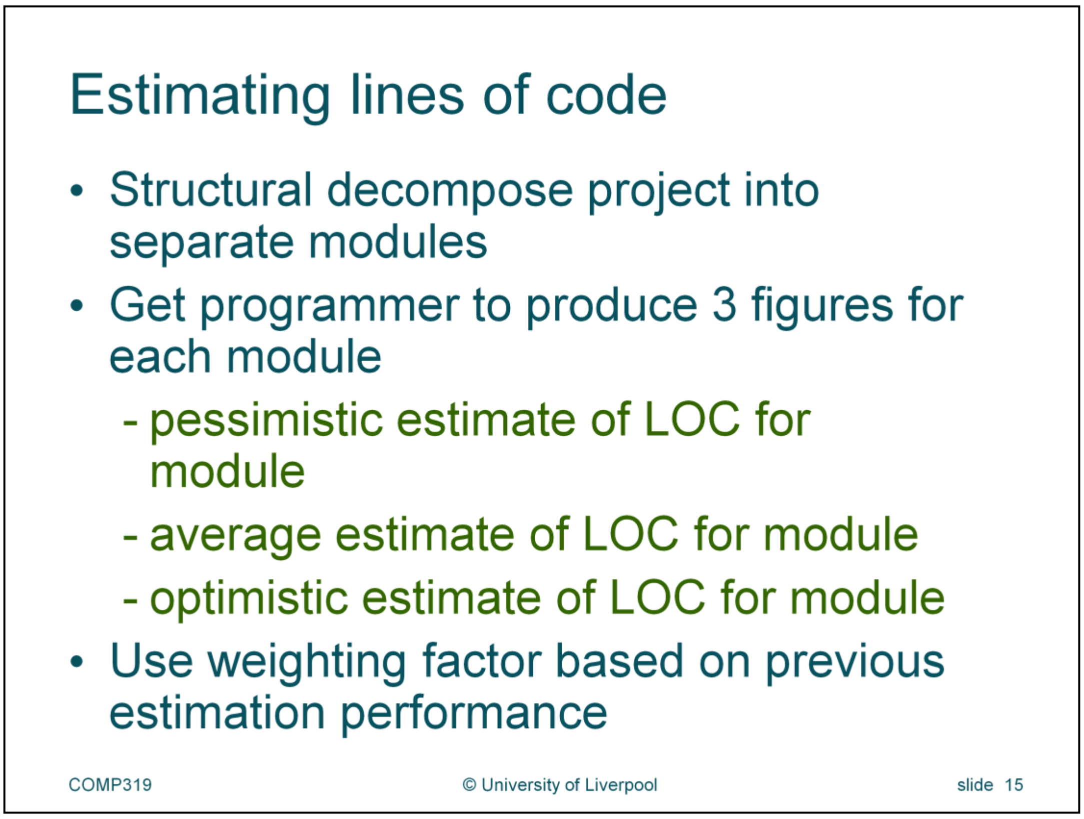


Software productivity metrics

- Measures of size
  - Lines of (source) code per person month: (LOC/pm)
  - Object code instructions
- Document pages
- Measure of function
  - Function points
  - Object points

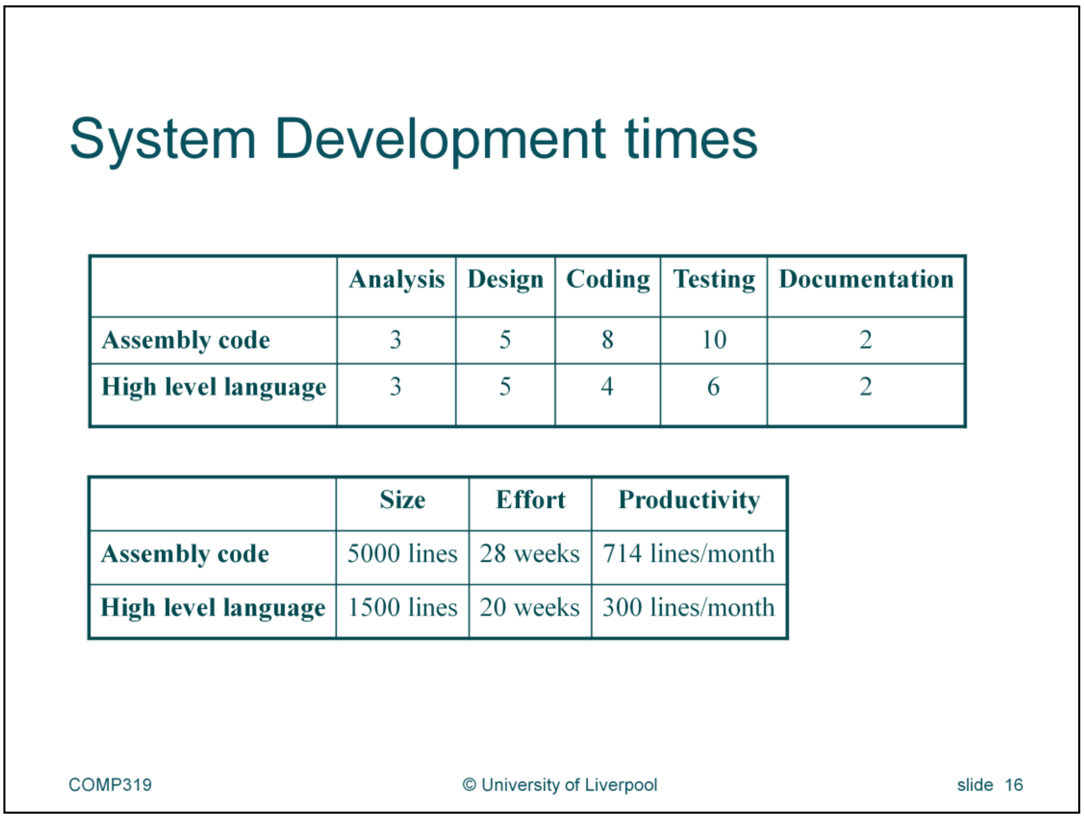

Lines of code (KLOC)

- Easy to measure
- Difficult to estimate
- As productivity measure?
  - Code quality
  - Project delivery
  - Language dependency


Estimating lines of code

- Structural decompose project into separate modules
- Get programmer to produce 3 figures for each module
  - pessimistic estimate of LOC for module
  - average estimate of LOC for module
  - optimistic estimate of LOC for module
- Use weighting factor based on previous estimation performance

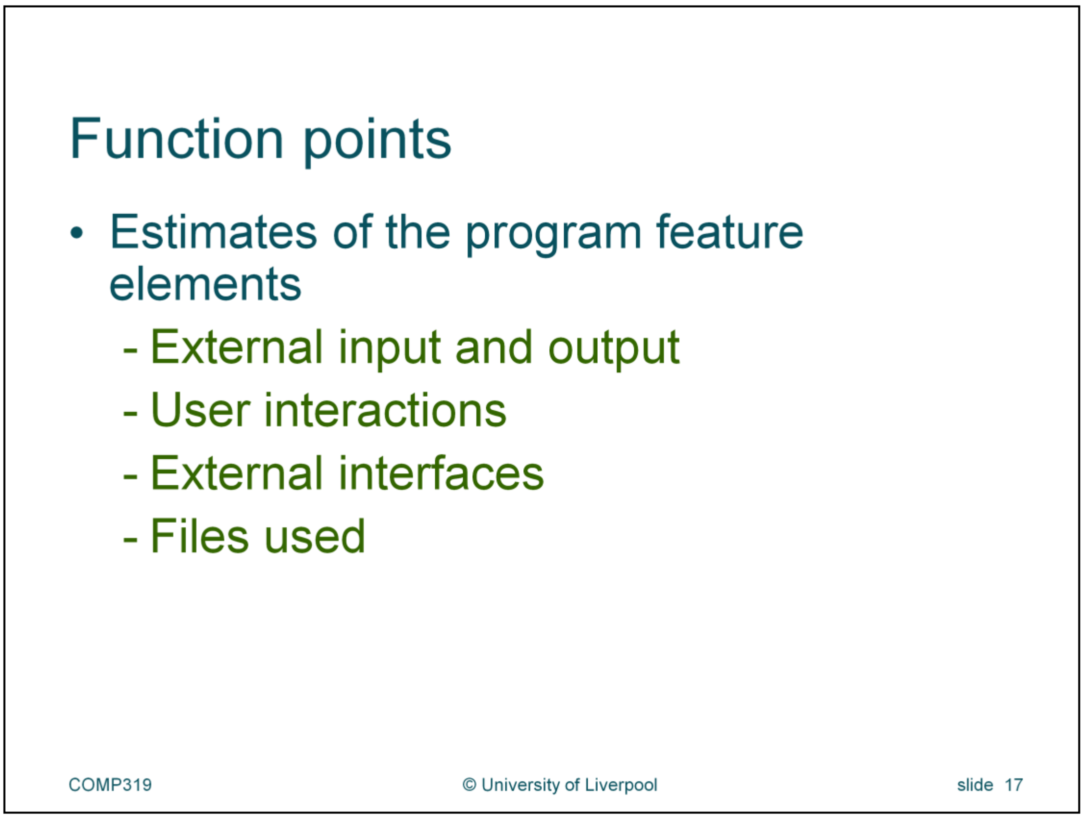
COMP319 © University of Liverpool slide 15



System Development times

	Analysis	Design	Coding	Testing	Documentation
Assembly code	3	5	8	10	2
High level language	3	5	4	6	2

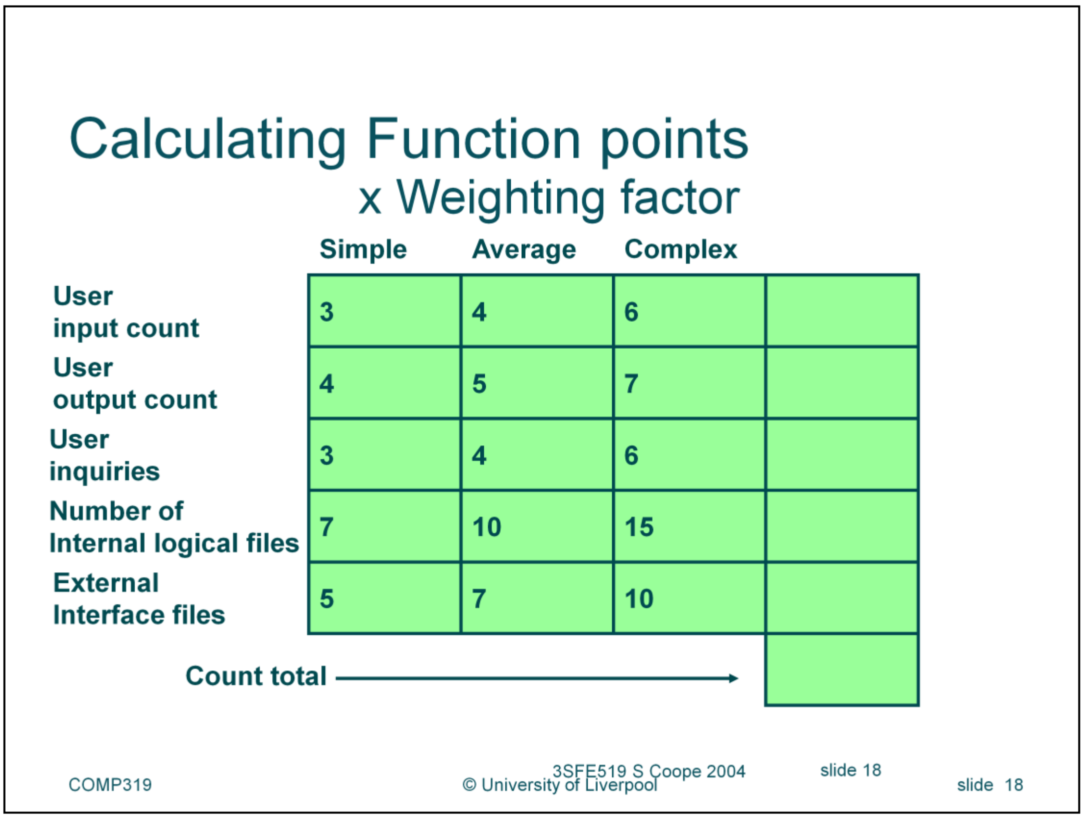
	Size	Effort	Productivity
Assembly code	5000 lines	28 weeks	714 lines/month
High level language	1500 lines	20 weeks	300 lines/month



Function points

- Estimates of the program feature elements
  - External input and output
  - User interactions
  - External interfaces
  - Files used

COMP319 © University of Liverpool slide 17

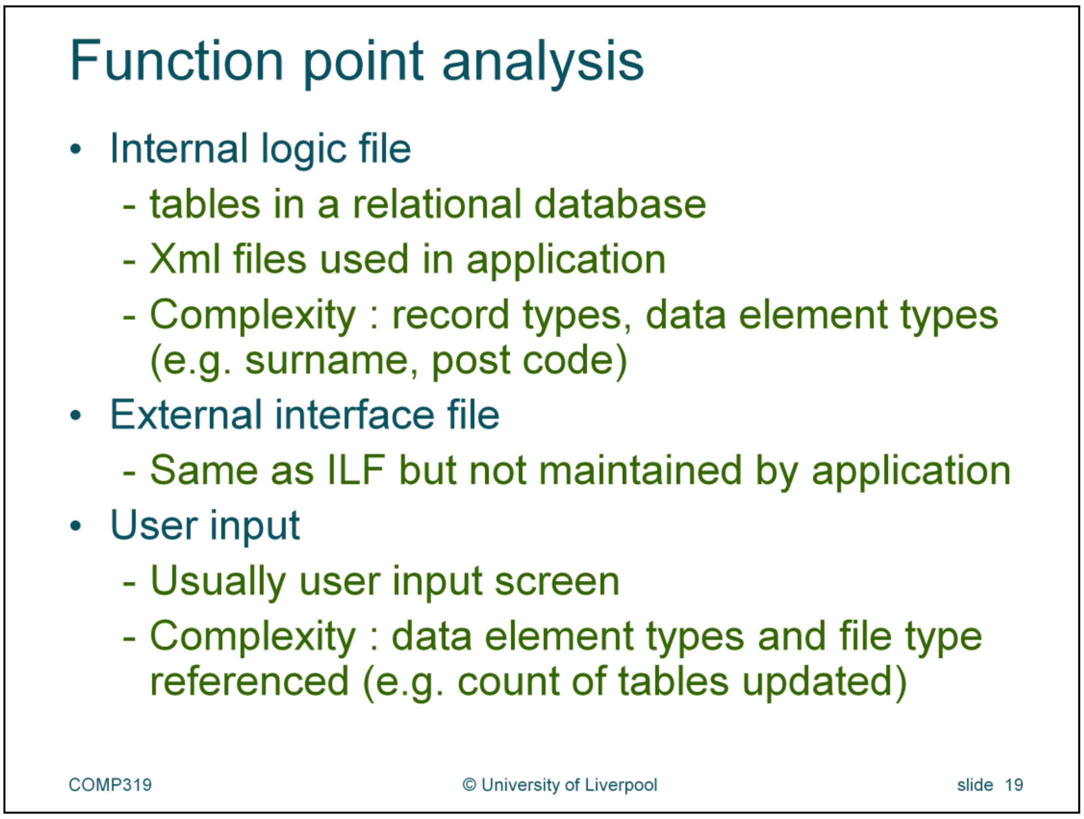


Calculating Function points

x Weighting factor

	Simple	Average	Complex	
User input count	3	4	6	
User output count	4	5	7	
User inquiries	3	4	6	
Number of Internal logical files	7	10	15	
External Interface files	5	7	10	
Count total →				

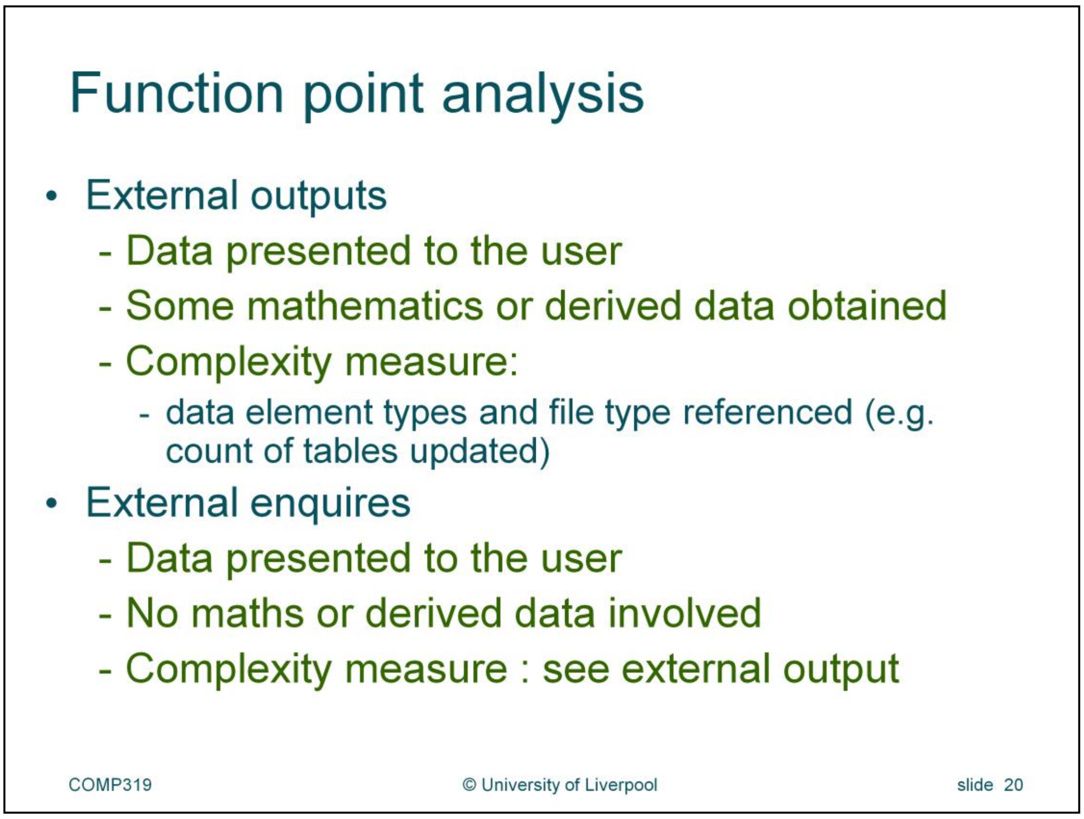
COMP319 © University of Liverpool 3SFE519 S Coope 2004 slide 18 slide 18



Function point analysis

- Internal logic file
  - tables in a relational database
  - Xml files used in application
  - Complexity : record types, data element types (e.g. surname, post code)
- External interface file
  - Same as ILF but not maintained by application
- User input
  - Usually user input screen
  - Complexity : data element types and file type referenced (e.g. count of tables updated)

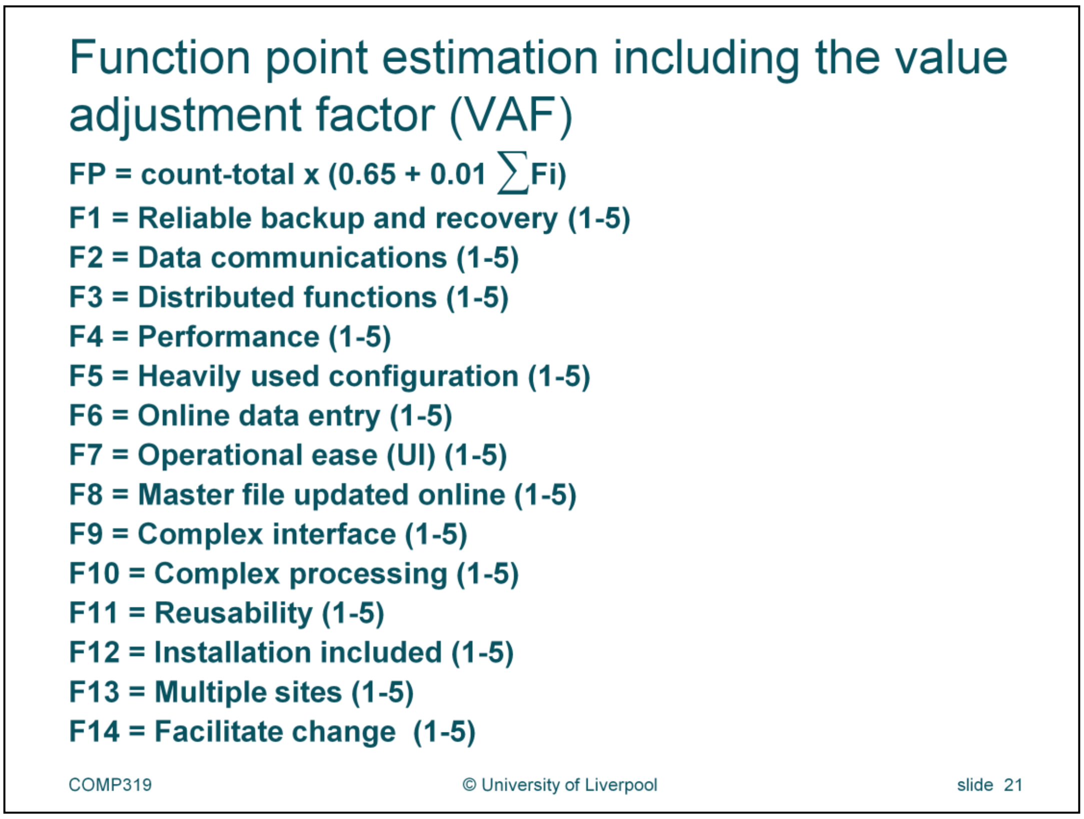
COMP319 © University of Liverpool slide 19



Function point analysis

- External outputs
  - Data presented to the user
  - Some mathematics or derived data obtained
  - Complexity measure:
    - data element types and file type referenced (e.g. count of tables updated)
- External enquires
  - Data presented to the user
  - No maths or derived data involved
  - Complexity measure : see external output

COMP319 © University of Liverpool slide 20



Function point estimation including the value adjustment factor (VAF)

FP = count-total x (0.65 + 0.01 $\sum$Fi)

F1 = Reliable backup and recovery (1-5)
F2 = Data communications (1-5)
F3 = Distributed functions (1-5)
F4 = Performance (1-5)
F5 = Heavily used configuration (1-5)
F6 = Online data entry (1-5)
F7 = Operational ease (UI) (1-5)
F8 = Master file updated online (1-5)
F9 = Complex interface (1-5)
F10 = Complex processing (1-5)
F11 = Reusability (1-5)
F12 = Installation included (1-5)
F13 = Multiple sites (1-5)
F14 = Facilitate change (1-5)

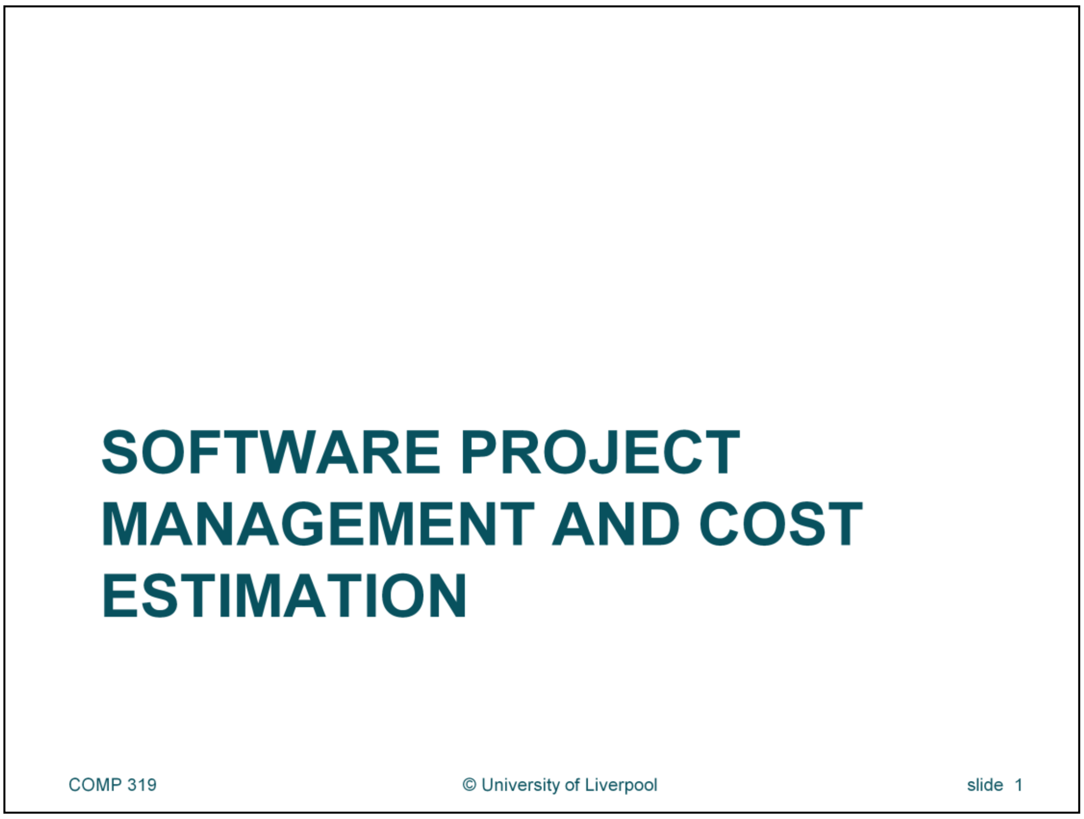
COMP319 © University of Liverpool slide 21

Note this is often not used. The VAF will adjust the raw FP count by a factor of about 2 either way. In general function point tool use the raw FP count. From devdaily.com/FunctionPoints

“I can't tell you much about the history of the VAF, but what I can tell from the conversations I've had with many other users is that they don't use the VAF. This stems from at least two reasons that I can determine:

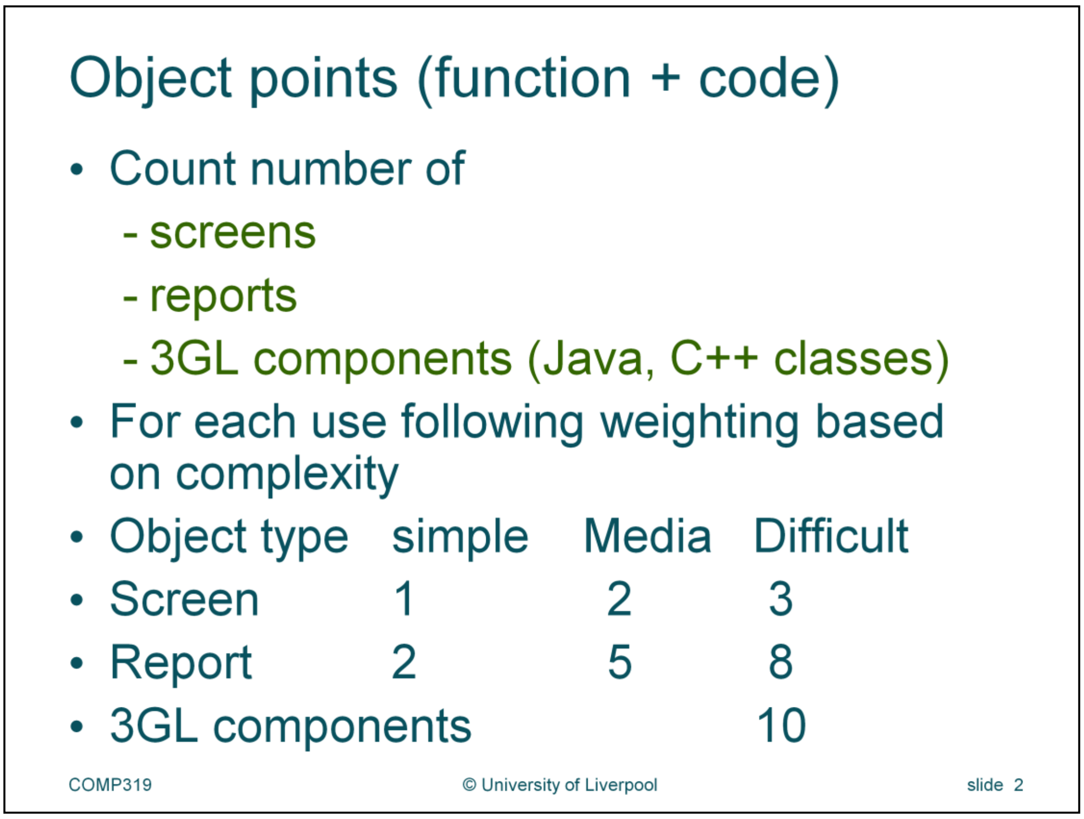
Most users count function points to derive a number that they can plug into another piece of software to determine a cost estimate. Those other software applications usually have their own equivalent of the VAF, and in fact, instruct you to supply the ``raw FP count". So, in this case, the VAF competes against these vendor tools.

Some users don't feel that the GSCs are flexible enough. I tend to agree, and I think it's an easy argument. When you look at the math below, you'll see that for two applications under consideration, if both start with the same function point count - let's say 1,000 FPs - after adjustments the hardest application in the world would be rated at 1,350 FPs, and the easiest possible application would be rated at 650 FPs when adjusted. Let's say the hardest application in the world had to run on 10 different operating systems in 15 languages and be distributed electronically to 1 million users, and the easiest would be written in Microsoft Access, run on Windows, and be used by only one user, the author of the program. Do you really think the first application is only about twice as hard to deliver as the second? No, I certainly don't, and this is why I don't use the VAF.”



SOFTWARE PROJECT MANAGEMENT AND COST ESTIMATION

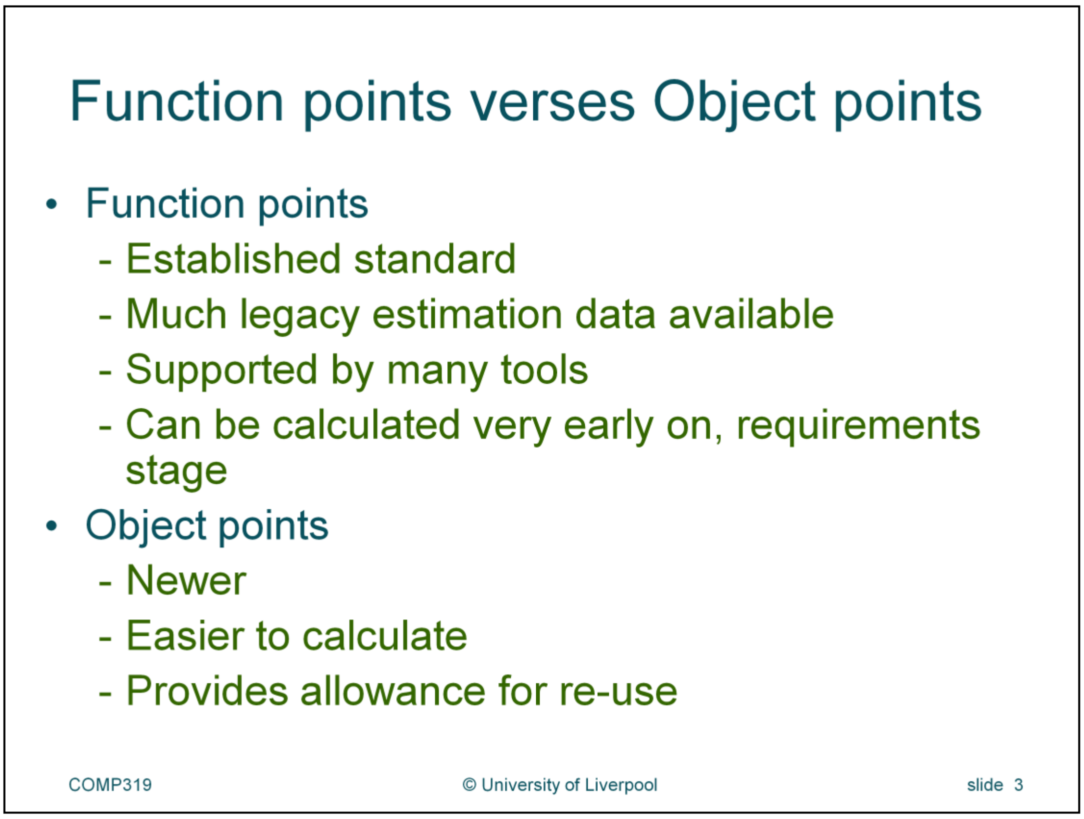
COMP 319 © University of Liverpool slide 1



Object points (function + code)

- Count number of
  - screens
  - reports
  - 3GL components (Java, C++ classes)
- For each use following weighting based on complexity

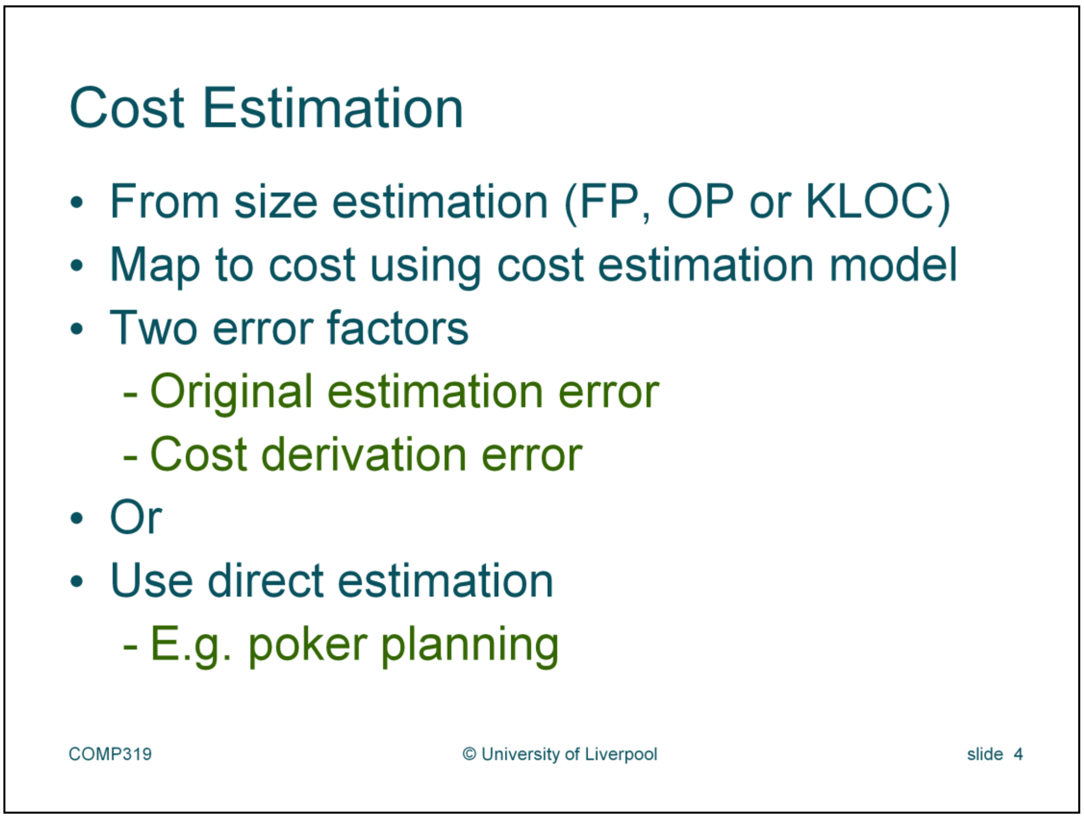
Object type	simple	Media	Difficult
Screen	1	2	3
Report	2	5	8
3GL components			10



Function points verses Object points

- Function points
  - Established standard
  - Much legacy estimation data available
  - Supported by many tools
  - Can be calculated very early on, requirements stage
- Object points
  - Newer
  - Easier to calculate
  - Provides allowance for re-use

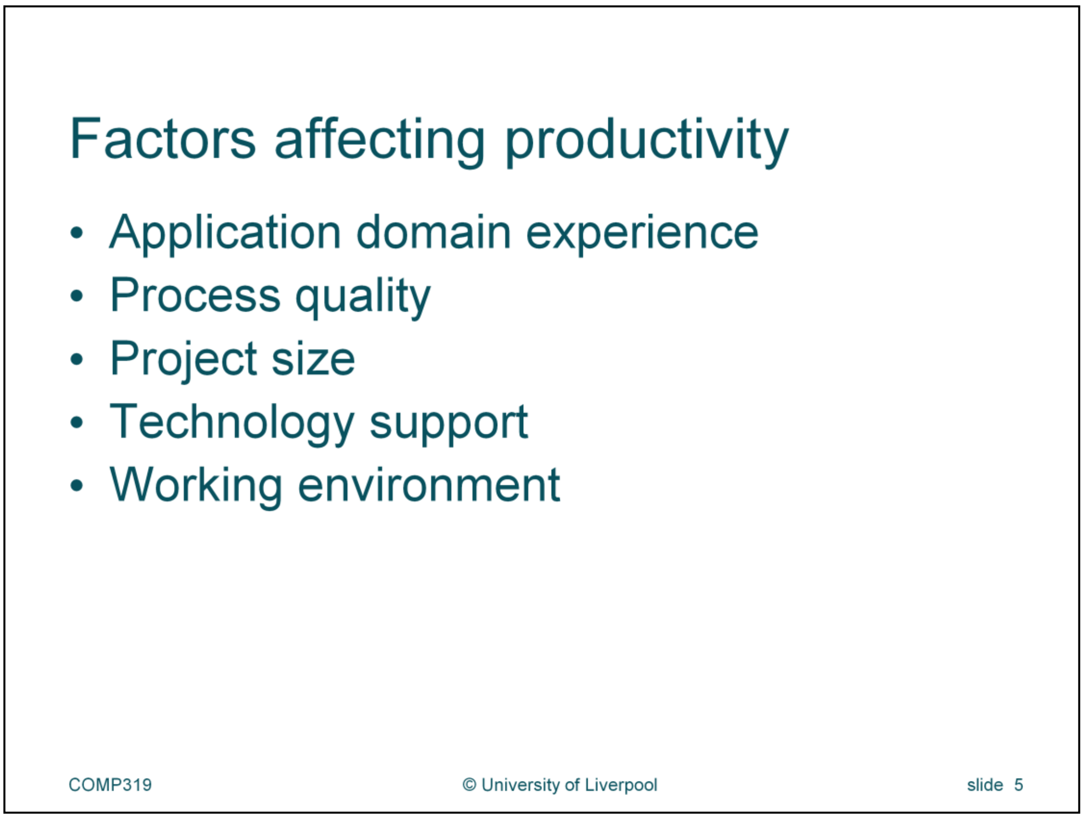
COMP319 © University of Liverpool slide 3



Cost Estimation

- From size estimation (FP, OP or KLOC)
- Map to cost using cost estimation model
- Two error factors
  - Original estimation error
  - Cost derivation error
- Or
- Use direct estimation
  - E.g. poker planning

COMP319 © University of Liverpool slide 4



Factors affecting productivity

- Application domain experience
- Process quality
- Project size
- Technology support
- Working environment

COMP319 © University of Liverpool slide 5

Individuals in a project may vary in their productivity by a factor of 10 (Boehm, et al. 1995), that is from 4 object points per month to 50.

The composition of the team and other factors must be taken into consideration in the overall estimate being made.

These other factors are:

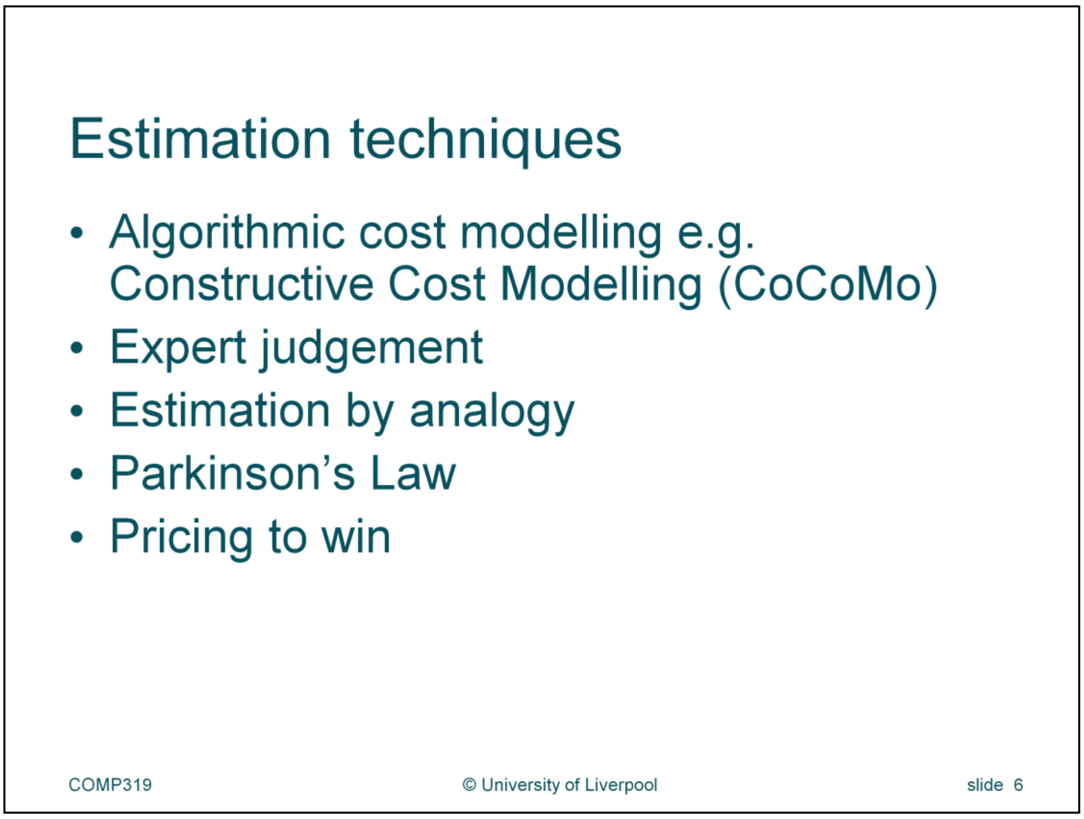
Knowledge of the application domain

The software process used in development – if these are good productivity is higher (we return to this later)

Large projects need communication with less time for development and high productivity

Good CASE tools, configuration management system etc improve productivity

Quietness and private areas contribute to better productivity



Estimation techniques

- Algorithmic cost modelling e.g. Constructive Cost Modelling (CoCoMo)
- Expert judgement
- Estimation by analogy
- Parkinson's Law
- Pricing to win

COMP319 © University of Liverpool slide 6

Using historical cost information an equation is developed that relates some software metric (usually size) to cost. Estimates of the metric then predict the cost/effort required.

Multiple experts estimate, then discuss and agree a compromise cost.

The cost of new project is estimated by analogy with comparable completed projects.

Parkinson's Law: work expands to fill the time. If there are 5 people and the software must be delivered in 6 months then the effort estimate is 30 person months.

The customers budget determines the cost. Makes sense if the system can be shipped to a subsequent customer.

However, most methods assume that will be no significant changes in the way software development is done. Changes in the past 10 years have shown that this is not a safe assumption. Examples of changes that affect estimation include:

Distributed and grid based systems

Web facilities

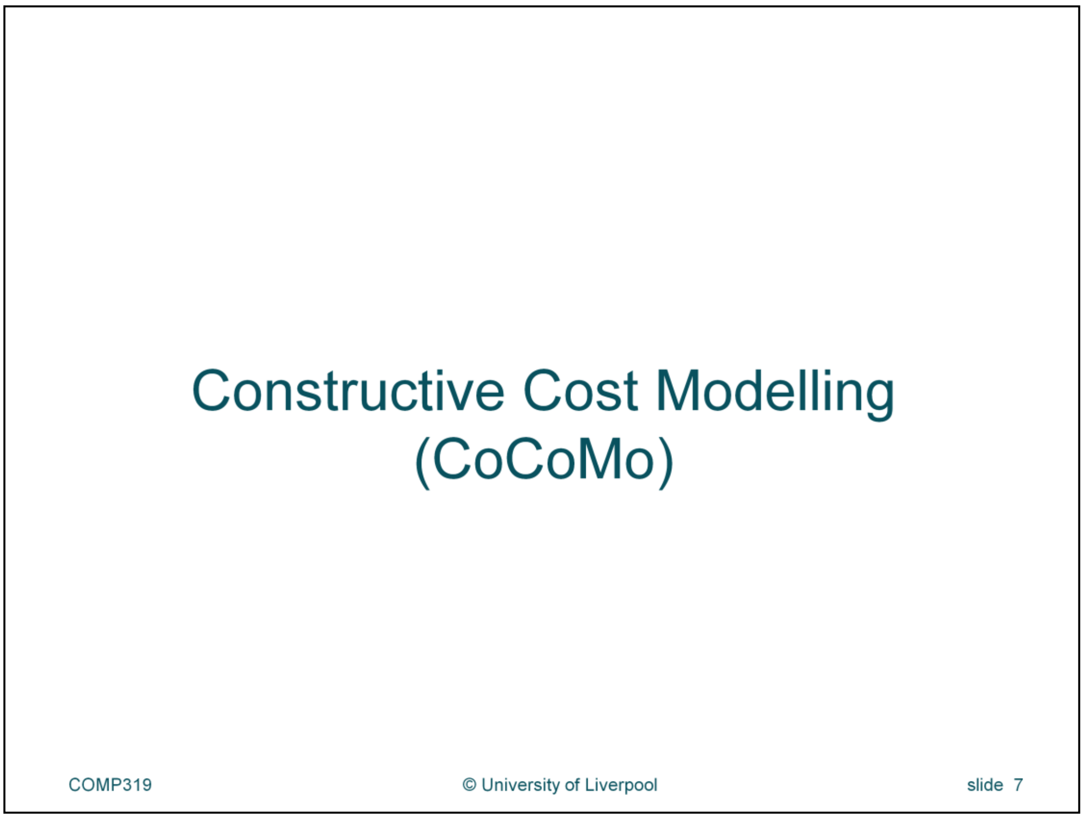
Entity Resource Planning or db centred systems

Shrink wrapped software

Module reuse

Scripting facilities

CASE tools



Constructive Cost Modelling (CoCoMo)

COMP319 © University of Liverpool slide 7

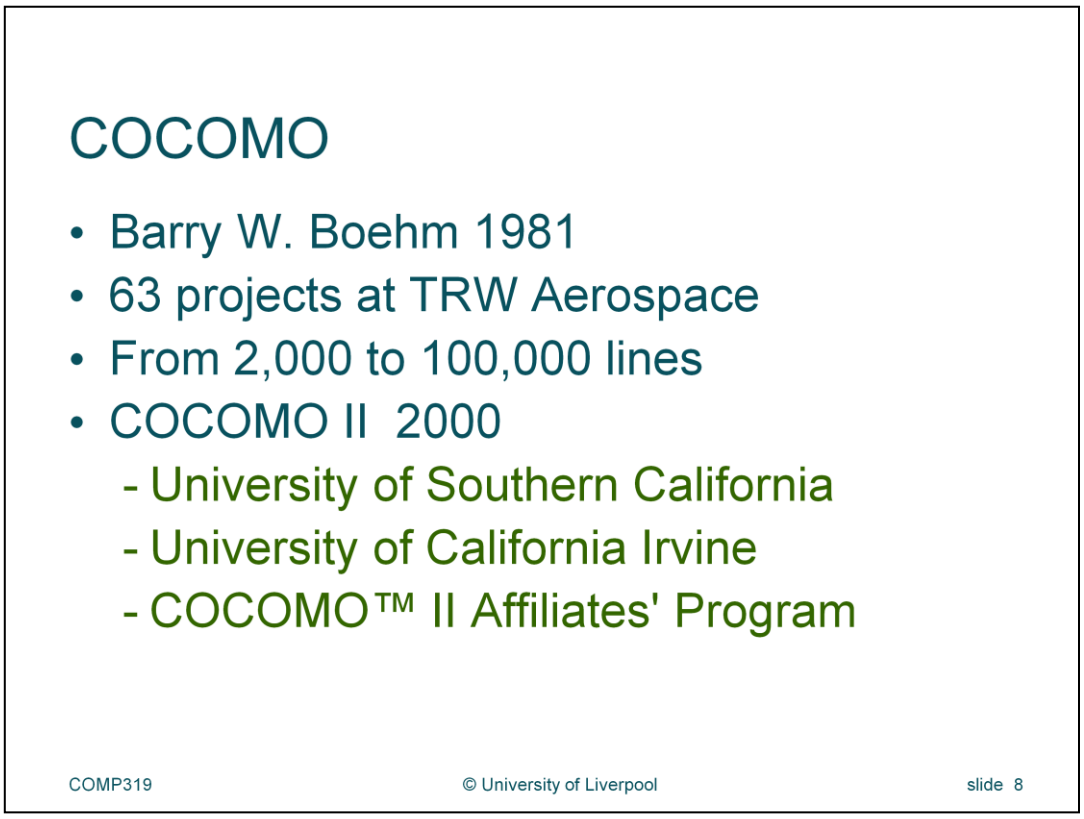
Constructive Cost Modelling (COCOMO) is an algorithmic model that is well known, widely used as a commercial tool, and software for it is in the public domain.

It was first proposed by Boehm, B. in his 1981 book “Software engineering economics”, Prentice-Hall. The definitive book for the method is Boehm, B. (2000) “Software Cost Estimation with COCOMO II”, Prentice-Hall.

Summary of the COCOMO II method is in the paper Boehm, B., Clark, B. et al. (1995) “Cost models for future software life cycle processes: COCOMO II”, Annals of Software Engineering, Vol 1, p57-94.

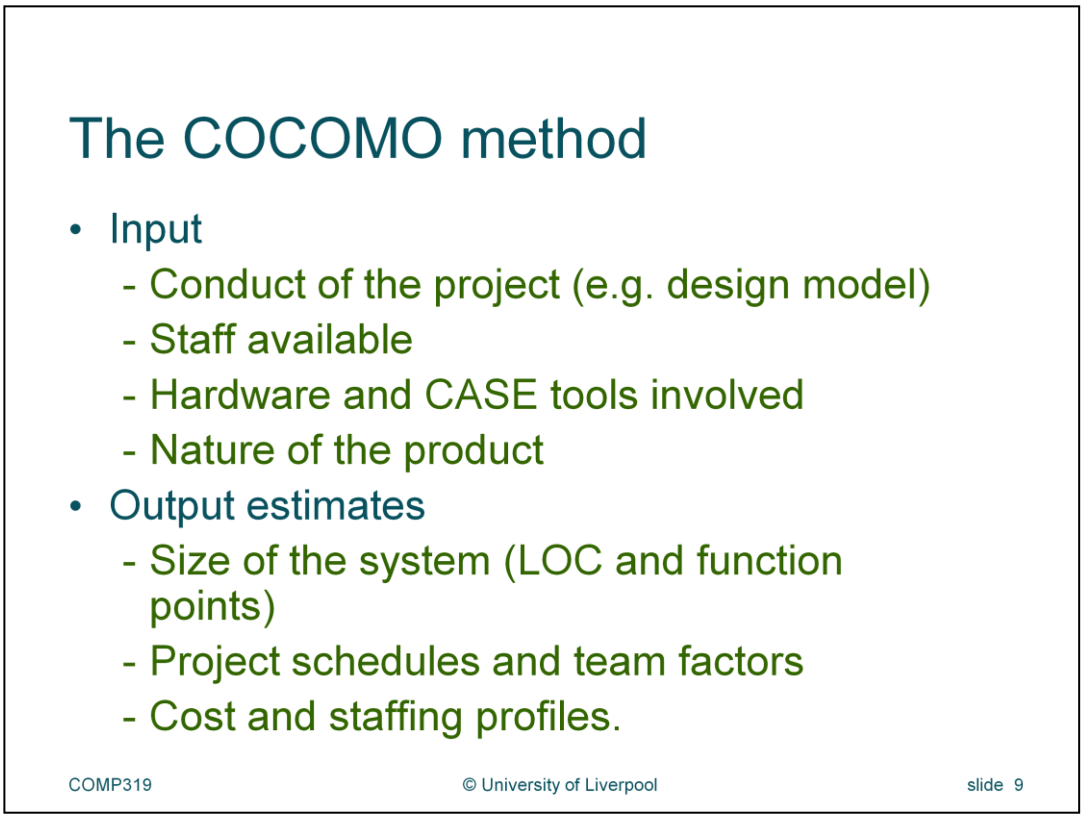
COCOMO assumes a stable software house or software department for which historical measures of software productivity exist. Where this is not available, the published data from finished software projects may be used.

In 1981 COCOMO was based on the waterfall model using a programming language. Now, it can cope with changes in method (e.g. the spiral or incremental model etc), use of shrink wrapped components such as Oracle that provides both a DBMS and full database programming language, and the use of various CASE tools and hardware.


COCOMO

- Barry W. Boehm 1981
- 63 projects at TRW Aerospace
- From 2,000 to 100,000 lines
- COCOMO II 2000
  - University of Southern California
  - University of California Irvine
  - COCOMO™ II Affiliates' Program

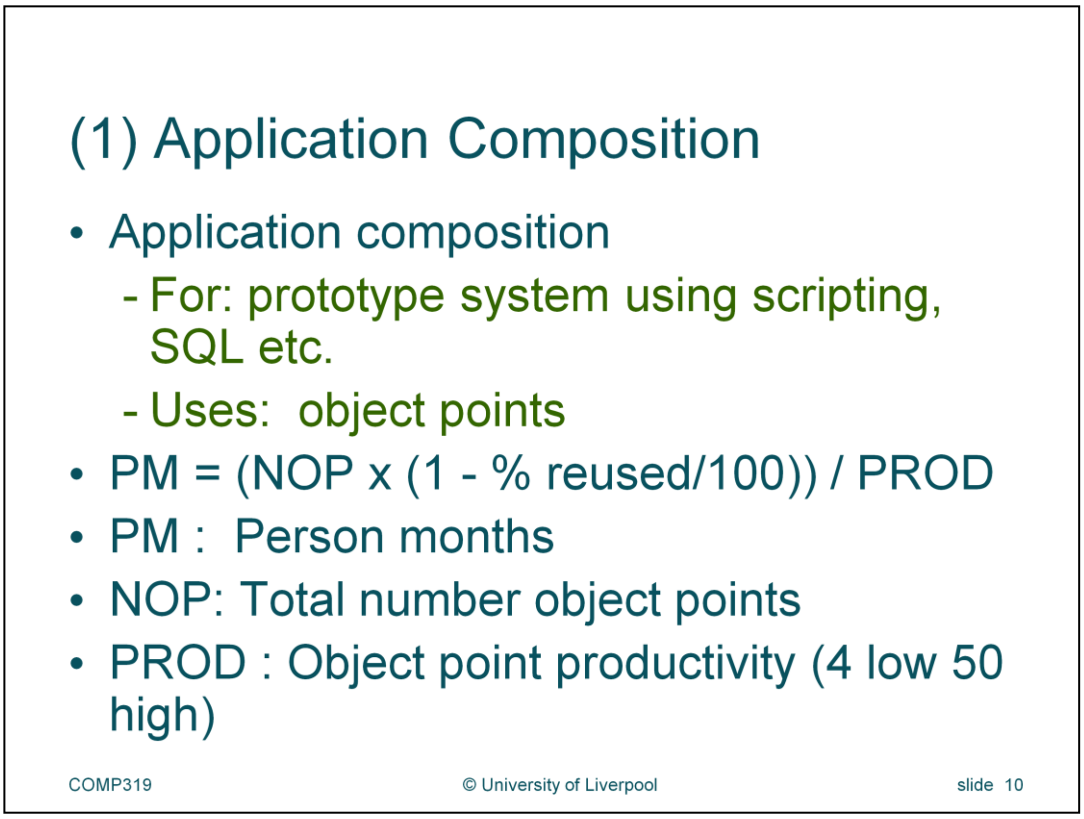
COMP319 © University of Liverpool slide 8



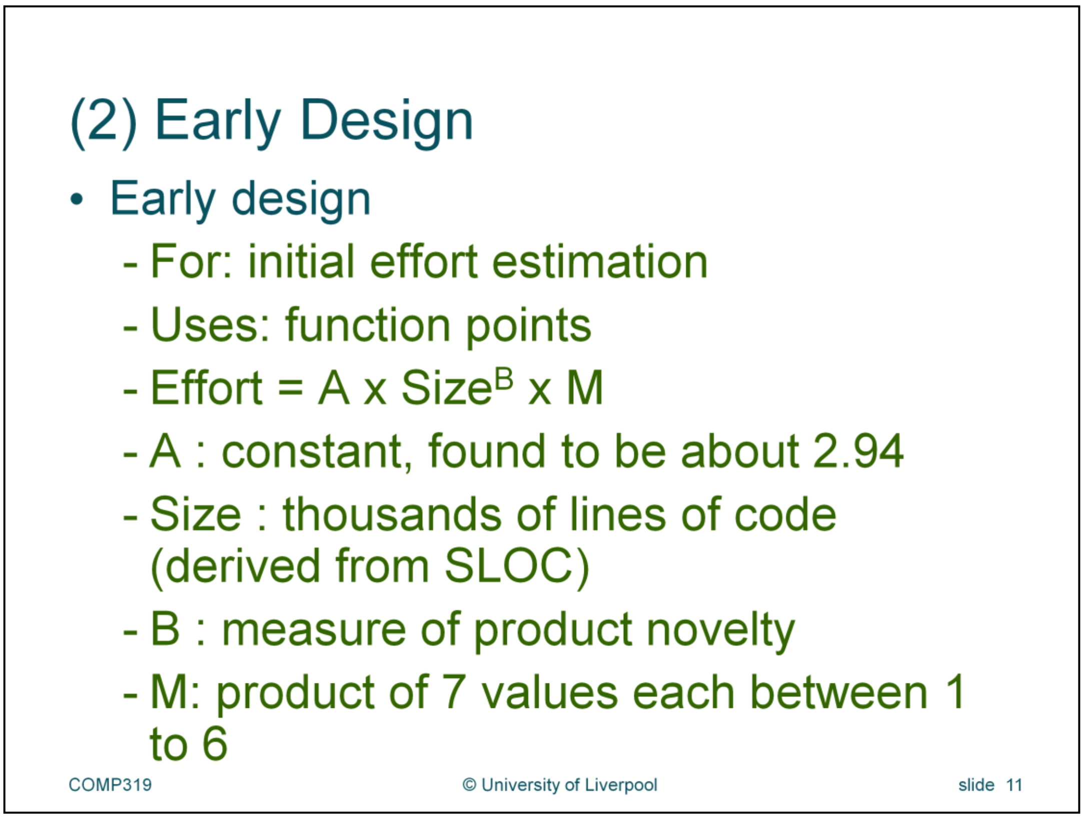
The COCOMO method

- Input
  - Conduct of the project (e.g. design model)
  - Staff available
  - Hardware and CASE tools involved
  - Nature of the product
- Output estimates
  - Size of the system (LOC and function points)
  - Project schedules and team factors
  - Cost and staffing profiles.

COMP319 © University of Liverpool slide 9


(1) Application Composition

- Application composition
  - For: prototype system using scripting, SQL etc.
  - Uses: object points
- PM = (NOP x (1 - % reused/100)) / PROD
- PM : Person months
- NOP: Total number object points
- PROD : Object point productivity (4 low 50 high)



(2) Early Design

- Early design
  - For: initial effort estimation
  - Uses: function points
  - Effort = A x Size^B x M
  - A : constant, found to be about 2.94
  - Size : thousands of lines of code (derived from SLOC)
  - B : measure of product novelty
  - M: product of 7 values each between 1 to 6

COMP319 © University of Liverpool slide 11

Early Design model

Used once the requirement is finalised and an architecture design is required with an estimate of cost. It is based on the

formula:<read>

A is a constant which based on Boehms data is 2.94; The size is thousands of lines of source code : KSLOC – obtained by calculating the function points required (there are look-up tables to do the relationship between function points and KSLOC).

The exponent (B) varies from 1.1 to 1.24 and covers an estimate of the novelty of the project. As novelty increases the number of lines of code and thus effort increases.

The constant M is the product of seven project and process characteristics measured on a scale from 1 (very low) to 6 (very high).

RCPX product reliability and complexity

RUSE reuse required

PDIF platform difficulty

PERS personnel capability

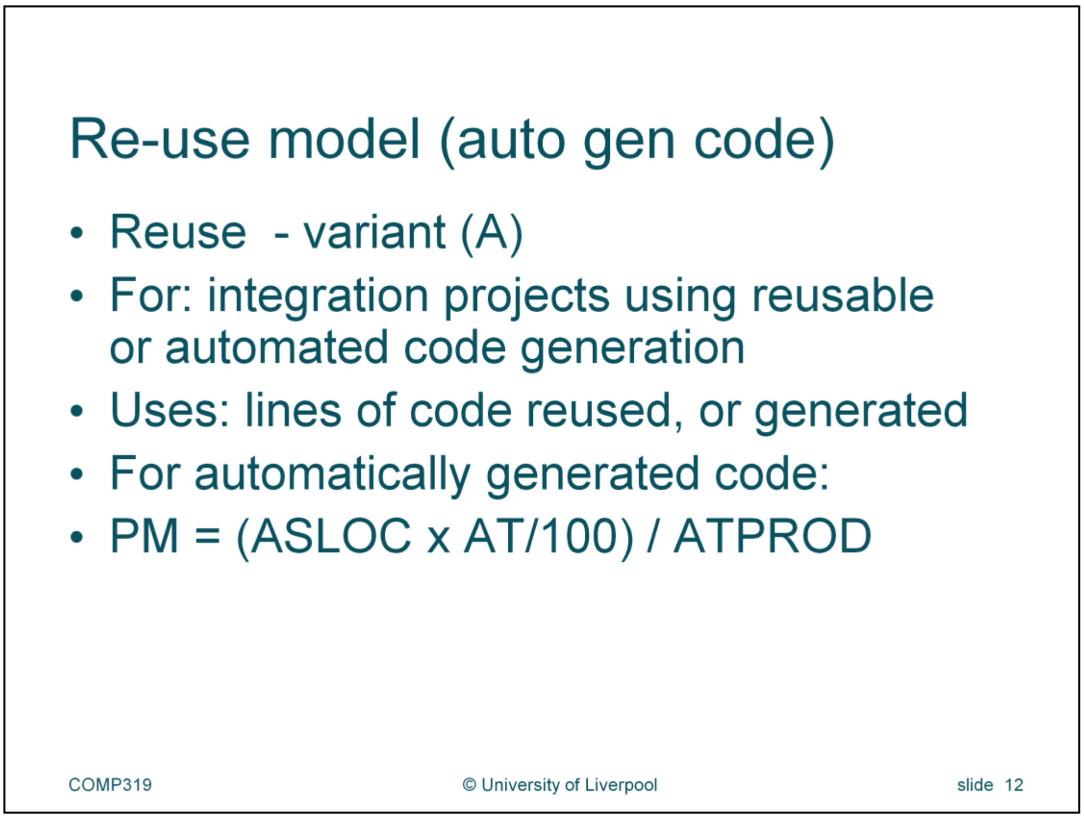
PREX personnel experience

SCED schedule

FCIL support facilities.

From this we get:

PM = 2.94 x Size**B x (RCPX x RUSE x PDIF x PERS x PREX x SCED x FCIL)



Re-use model (auto gen code)

- Reuse - variant (A)
- For: integration projects using reusable or automated code generation
- Uses: lines of code reused, or generated
- For automatically generated code:
- PM = (ASLOC x AT/100) / ATPROD

COMP319 © University of Liverpool slide 12

Reuse Model (A)

Two variants exist based on whether all code is automatically generated (A) or some is automatically generated and some newly written (B).

Reuse of code is perhaps best illustrated in computer game software, where new story lines lead to new games – e.g. Harry Potter.

Code that does not need to be understood to reuse it is termed black box code, and the development effort associated with it is deemed to be zero.

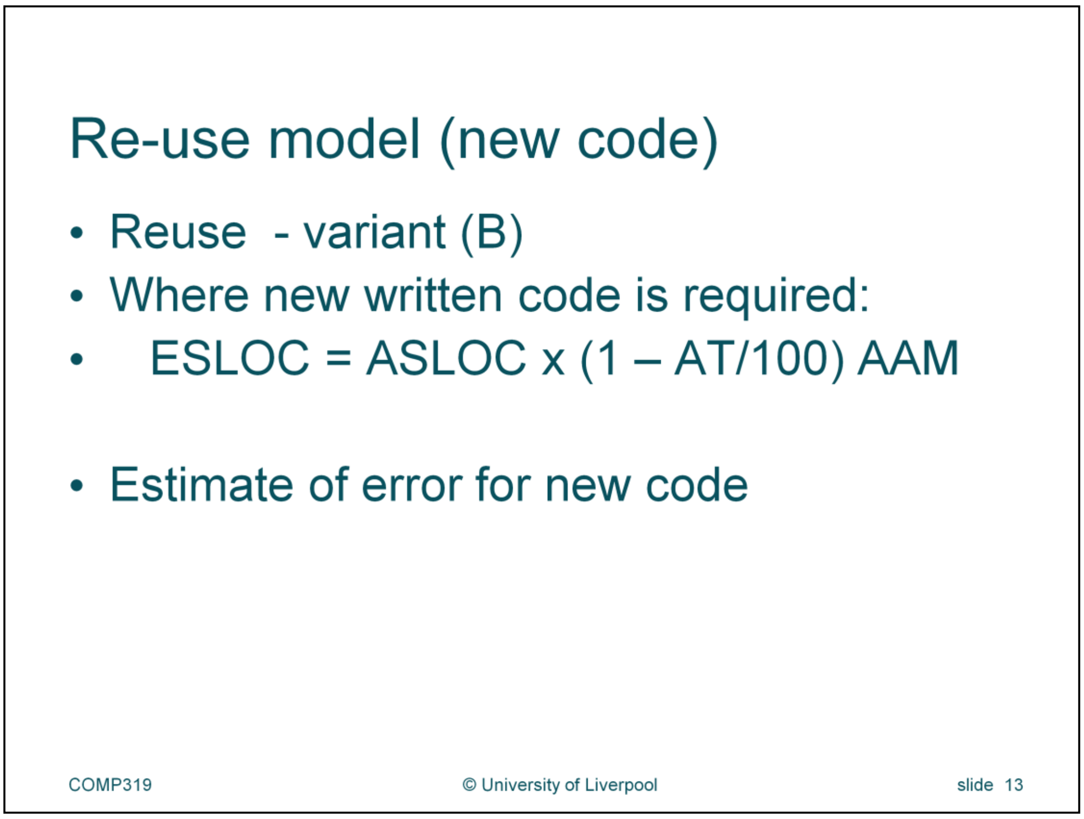
Code that has to be modified to be reused, is termed white box code, and effort is required to understand and modify it.

In addition code may be automatically generated – adding a second form of reuse.

(A) For code automatically generated the formula used is:

ASLOC is the number of lines of code in the component that have to be adapted. AT is the percentage of adapted code that is automatically generated and ATPROD is the integration productivity of staff. ATPROD is currently assumed to be about 2,400 source statements per month (Boehm et al., 2000).

e.g. If there is a total of 20,000 lines of white-box reused code in a system and 30% is automatically generated, then PM(auto) is: (20,000 x 30/100) / 2400 = 2.5 person months.


Re-use model (new code)

- Reuse - variant (B)
- Where new written code is required:
- ESLOC = ASLOC x (1 – AT/100) AAM

- Estimate of error for new code

COMP319 © University of Liverpool slide 13

Reuse Model

(B) In the reuse model where there is some new code and some reused code, the effort required is calculated indirectly. Thus, based on the number of lines of code reused, it calculates a figure that represents the number of lines of new code.

e.g. With 30,000 lines of code to be reused, the new equivalent size estimate might be 6,000; or put another way 6,000 new lines are required to reuse the 30,000. This calculated figure is added to the number of lines of new code to be developed in the COCOMO II post-architecture model.

The estimates in this reuse model are thus:

ASLOC – number of lines of code in the component that have to be adapted

ESLOC – equivalent number of lines of new source code

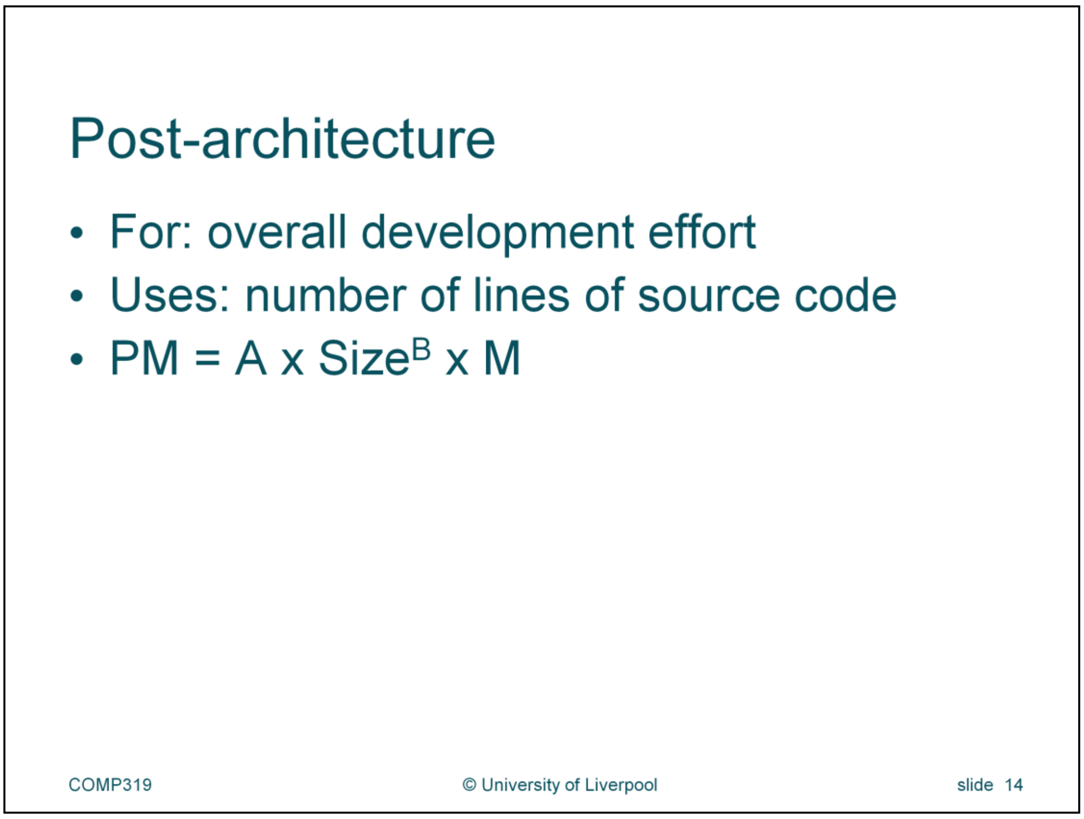
The ESLOC figure summarises the effort required in making changes to the reused code and for making changes to the system to integrate the code. It also takes into account the automatically generated code where the calculation is as above.

We now can calculate the equivalent lines of source code as: <read>

ASLOC is reduced according the the percentage of automatically generated code.

AAM – Adaptation Adjustment Multiplier. A sum of: the cost of making the changes, cost of understanding the code, and an assessment factor which determines whether the code can be reused.

This model is non-linear, as more and more reuse is contemplated, the cost per code unit reused drops.


Post-architecture

- For: overall development effort
- Uses: number of lines of source code
- PM = A x Size^B x M

COMP319 © University of Liverpool slide 14

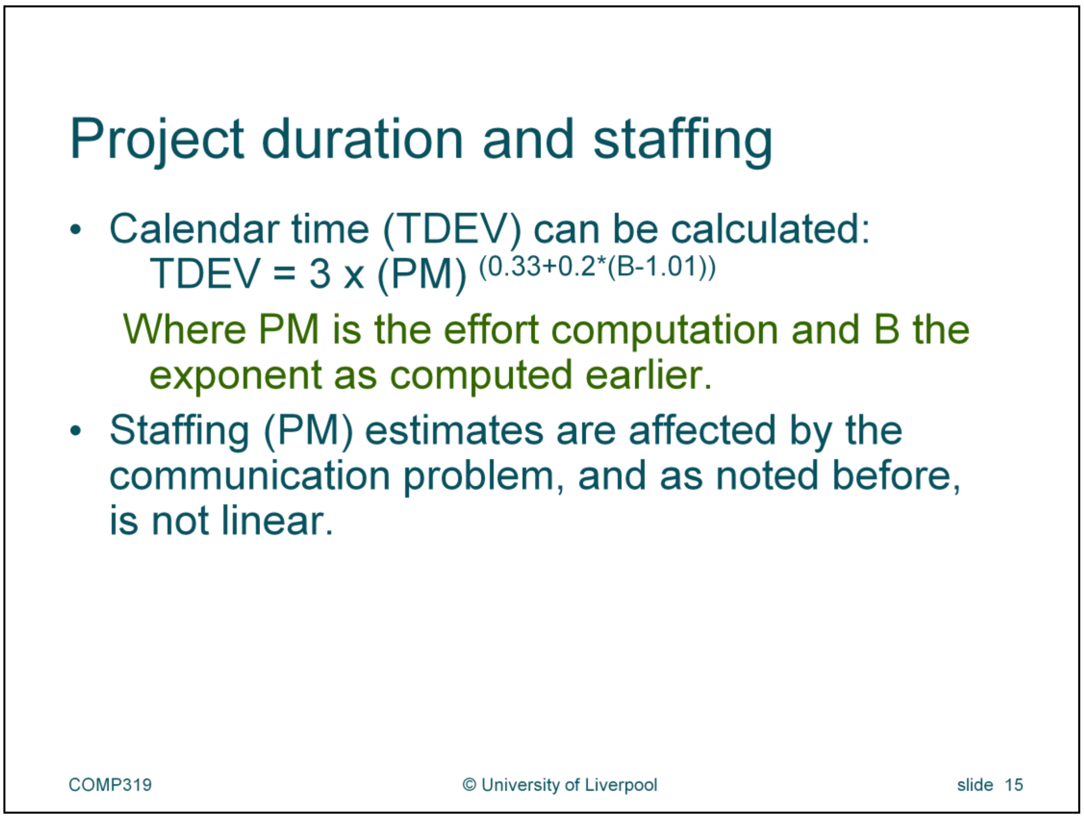
Post-Architecture Model

Once an initial architectural design is available and the structural units are known the post-architectural COCOMO II model can be used. It uses the same exponent based formula seen before: It is assumed this will now be more accurate and uses a more extensive set (17) of product, process, and organisational attributes as more information is now available. The model uses an estimate of:

1. the total number of lines of new code to be developed;
2. the equivalent number of source lines of code (ESLOC) needed calculated using the reuse model and
3. 3. the number of lines of code that have to be modified because of changes in the requirements.

These values are added to give KSLOC. B (which is continuous, as before) is made up of 5 scale factors rated on a six point scale from very low to extra high (5 to 0). The ratings are summed and divided by 100 and added to 1.01 to get the actual exponent used. The factors cover: the previous experience with this type of project – a new project scores low (say 4); freedom from client involvement – no client involvement rated very high (say 1); risk analysis done – no risk analysis rated very low (say 5); team cohesion and experience of working together – rated nominal (say 3); and process control maturity – some process control present, rated nominal (say 3).

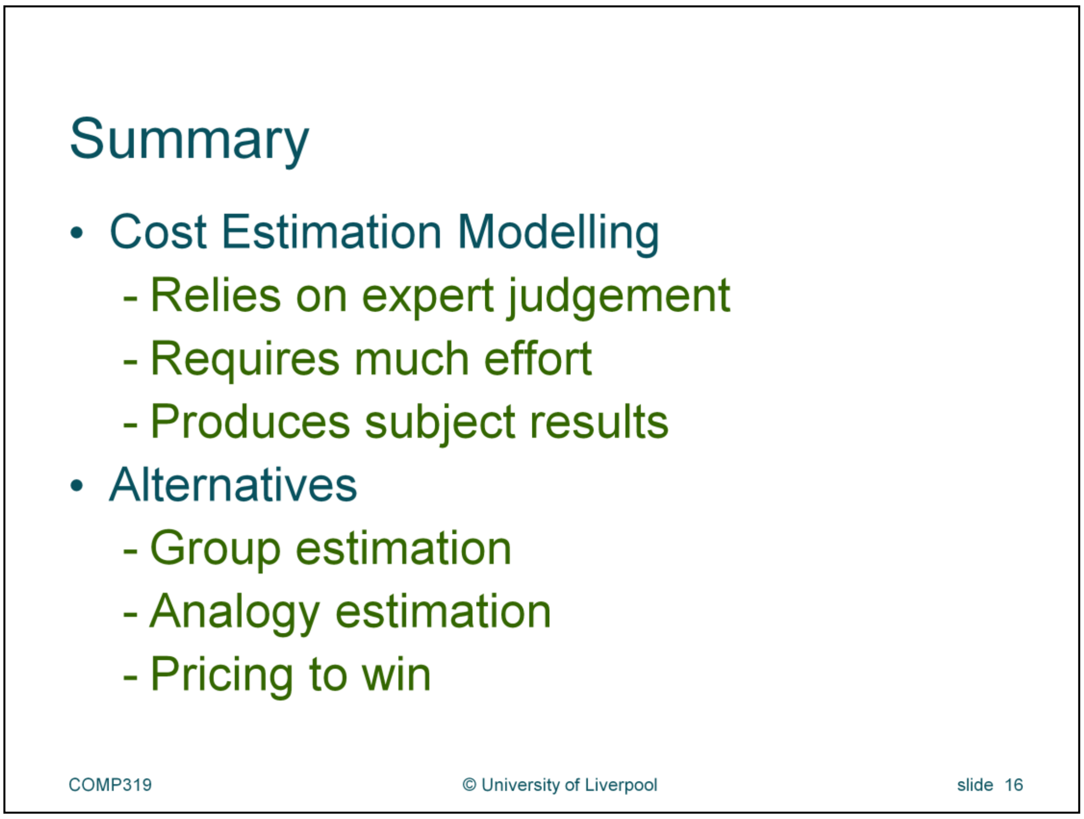
In this example the sum is 16 which we divide by 100. Add 0.16 to 1.01 to give a value for B of 1.17. M is calculated using 17 project cost drivers covering the product, the hardware, personnel, and the project (7 covered earlier). The are estimated based on experience and in practice are difficult to use with any accuracy.


Project duration and staffing

- Calendar time (TDEV) can be calculated:
  TDEV = 3 x $(PM)^{(0.33+0.2*(B-1.01))}$

  Where PM is the effort computation and B the exponent as computed earlier.
- Staffing (PM) estimates are affected by the communication problem, and as noted before, is not linear.

COMP319 © University of Liverpool slide 15



Summary

- Cost Estimation Modelling
  - Relies on expert judgement
  - Requires much effort
  - Produces subject results
- Alternatives
  - Group estimation
  - Analogy estimation
  - Pricing to win

COMP319 © University of Liverpool slide 16

OBJECT ORIENTATION AND OBJECT PATTERNS

Language levels

Object-oriented language (e.g. C++, Smalltalk, Java, C#)

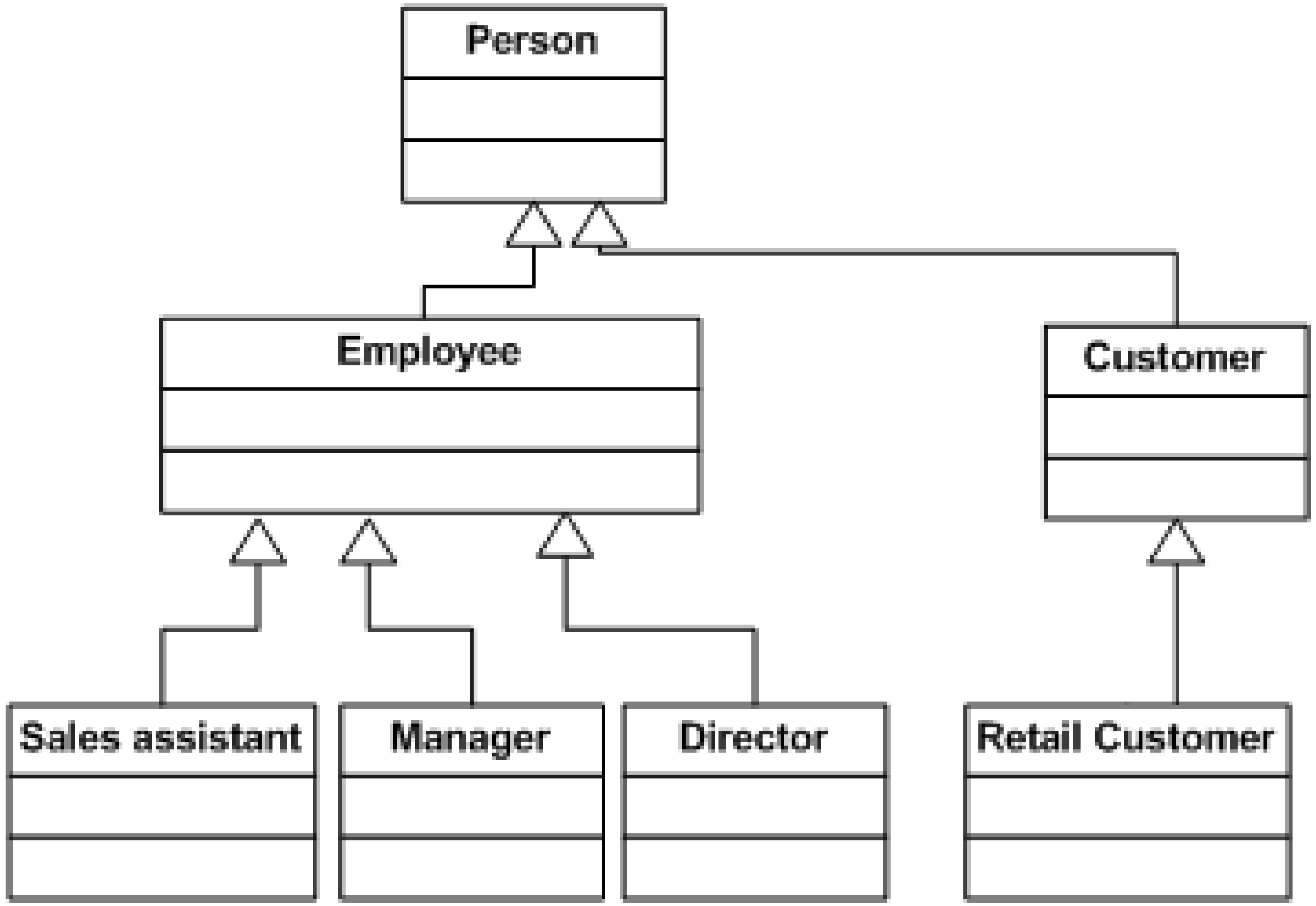
High-level language (e.g. C, Pascal)

Assembly languge (e.g. IBM or Intel assembly language)

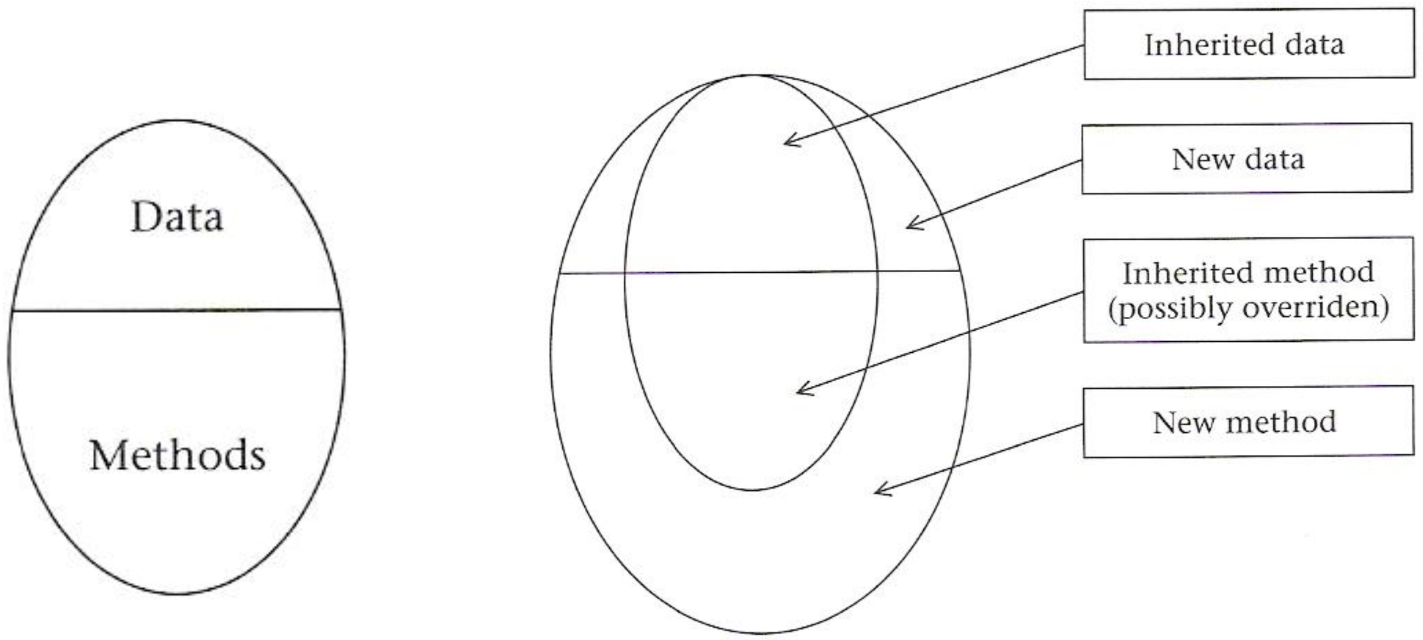
Machine language (ultimate output of compiler and/or assembler)

Microcode (tells the processor how to interpret machine language instructions)

Classification



Encapsulation & Inheritance



Benefits of OO approach

- Inheritance - classes
- Encapsulation - classes + methods
- Polymorphism - function
- good Cohesion
- good Coupling

OO Analysis   (!= OO design)

“ … is figuring out how to arrange a collection of classes that do a good job of representing your real-world problem in a format which a computer programmer finds easy to deal with.”

- Input
  - Thinking effort
  - Pencil, paper and Notebook
  - Observations
- Output
  - Answer to “which classes to use?”
  - UML diagrams

Object Orientated design

“ … *is about what kinds of data and method go into your classes and about how the classes relate to each other in terms of inheritance, membership and function calls.”*

- Input
  - Thinking effort + Pencil Paper, Notebook
  - OOA diagrams
- Output
  - UML diagrams
  - Header files (e.g. *.h files)

Role of documentation

- Central communication
  - Cut down on communication overhead
- Control
  - If it’s not in the specification, it won’t be built
- Annotation
  - Particularly of code but also design
- Operational
  - User/system manuals

Types of Documentation

- UML diagrams
- User Guides
- System Guides
- Management documents
- Requirement and Specification
- Schedule and Budget
- Organisation and Planning

Design Patterns

Software Evolution → Patterns

Software design patterns

Object-oriented language (e.g. C++, Smalltalk, Java)

High-level language (e.g. C, Pascal)

Assembly languge (e.g. IBM or Intel assembly language)

Machine language (ultimate output of compiler and/or assembler)

Microcode (tells the processor how to interpret machine language instructions)

Design patterns

- Repeatable approaches to problem solving in software design
- Not locked into any 1 language (but often use OO concepts)
- Speed up development
- Increase software flexibility
- Make software more readable
- Can be implemented as components which will move from reusable design to reusable code

Design Pattern types

- Architectural (approach to designing the whole system) example MVC
- Creational
- Structural (one class/method wrapping another)
- Behavioural
    - Example : call backs, persistence
- Concurrency
    - Controls multiple threads

Model View Controller

- Problem
  - Many different GUI APIs
  - GUI code can be very complex and messy
  - Porting GUI code between platforms is hardwork

MVC Components

- Splits the code into
  - Model
    - Stores, retrieves and manipulates the data
  - View
    - Renders the data on the screen
    - View fetches data from model
  - Controller
    - Processes user input, passing events to model
    - Controller can instruct view to render

Model

- Provides the following
  - business logic, rules (e.g. who can access a student's transcript)
  - validation (can also be in controller)
  - persistence
  - application state (session)
    - shopping cart for user
    - address book, contact list
    - logged in user id

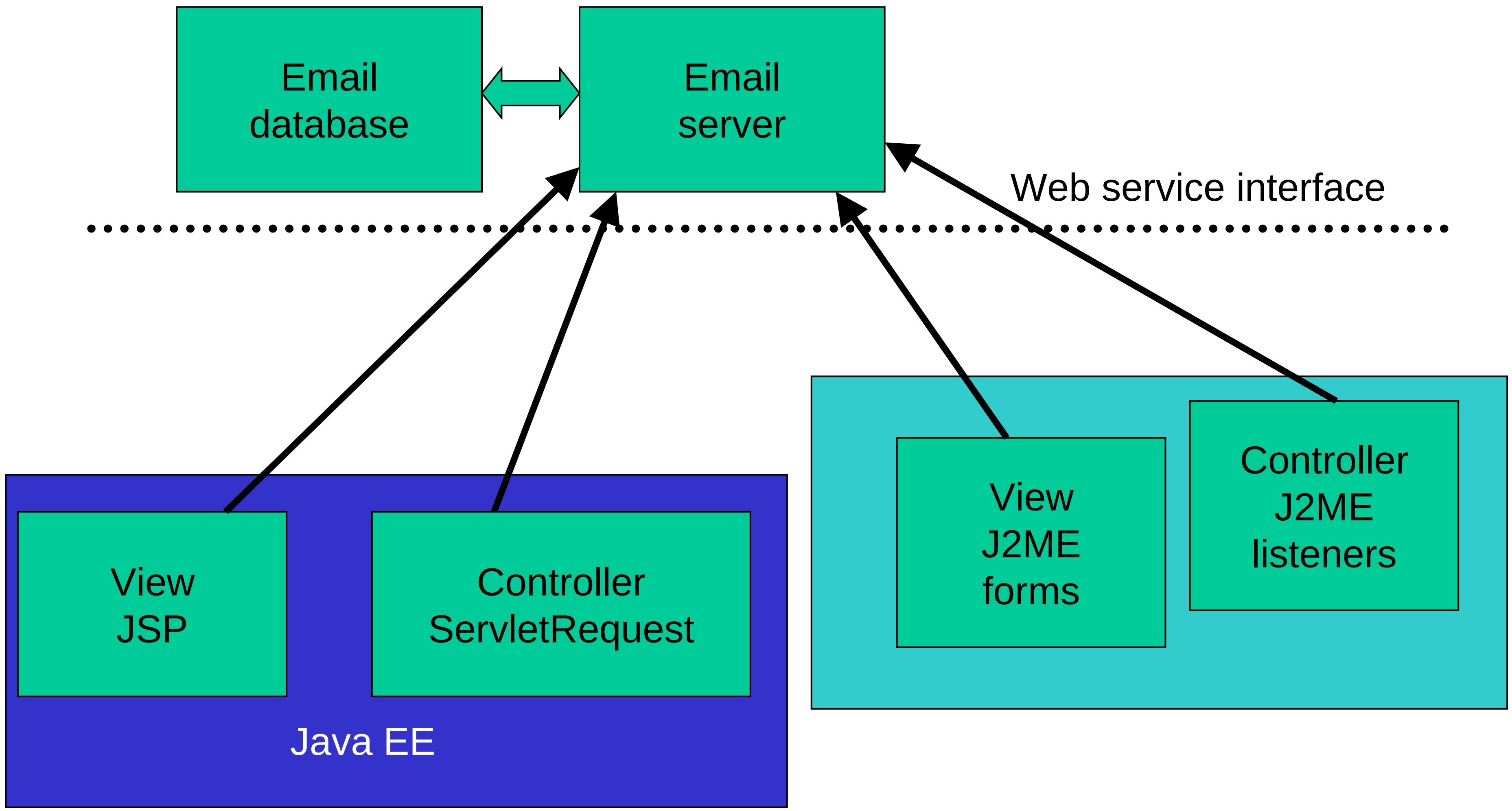
View

- Presents the information to the user
- Example View technologies
  - JSP   allows user to use Java to generate web pages
  - CSS web page presentation
  - HTML/XML
  - .aspx   Microsoft dynamic web technology

View/Controller options

- Java servlets and JSP   (browser client)
  - Java EE (Tomcat or Glassfish)
- .NET aspx pages   (browser client)
  - Microsoft Server
- J2ME MIDP
  - Mobile Java
- Java AWT/Swing
  - Java SE

MVC Example


Email database
Email server
Web service interface
View JSP
Controller ServletRequest
Java EE
View J2ME forms
Controller J2ME listeners

Model code example

Plain old Java class

```
class Customer {
    private String surname;
    private String forenames;
    private Date dateOfBirth;
}
```

Note this class can be ported to any platform that supports Java

View code example

```
Class CustomerForm extends Form {
    private TextField tfSurname; // text field input surname
    private TextField tfForenames; // forenames input
    private DateField dfDateOfBirth; // date of birth input
    private Command ok;
}
```

Controller Code (J2ME)

```
CustomerFormListener implements CommandListener {
   CustomerForm customerForm;
   public void commandAction(Command c, Displayable
     displayable) {
       if ( (c.getCommandType()==Command.OK)) {
                     Customer customer=customerForm.getCustomer();
                     customer.Save();
       }
}
```

MVC Model View Controller

- Benefits
  - Clear seperation of concerns
  - Easier to port software UI platform to UI platform
- VC code
  - Can be implemented by GUI specialist
- Team working
  - Web, Mobile App (iOS, Android), Mobile Web
  - Business logic

Command pattern

- Command
  - general abstraction for controller type interactions
  - allows controller API to change and keep business logic the same
- Code example

```
interface Command {
    void    OnExecute();
}
```

Command interface detail

```
public abstract class Command {
    private Hashtable <String,Object> callParameters=new Hashtable();
    private Hashtable <String,Object> returnParameters=new Hashtable();
    protected abstract void OnExecute();

    protected void setCallParameter(String name,Object object) {
        callParameters.put(name, object);
    }

    public void setCallParameters(Hashtable parms) {
        this.callParameters=parms;
    }

    protected Object getCallParameter(String name) throws
     ParameterNotFoundException {
        if (callParameters.containsKey(name)) {
            return(callParameters.get(name));
        }
        throw(new ParameterNotFoundException());
    }
}
```

CommandManager

```
public class CommandManager {
public void Execute(Hashtable parameters) throws
NoSuchCommandException,CommandNameMissingException {
        String packageName="patterns.commands";
         if (!parameters.containsKey("name")) {
            throw (new CommandNameMissingException());
        }
        String name=(String)parameters.get("name");
        String commandName=packageName+name;
        try {
           Class commandClass=Class.forName(commandName);
           Command commandObject=(Command)commandClass.newInstance();
           if (parameters!=null) {
              commandObject.setCallParameters(parameters);
           }
           commandObject.OnExecute();
        } catch (Exception exc1) {
            throw (new NoSuchCommandException(name));   // problem with command
class
        }
    }
```

HttpCommandManager extends CommandManager

```
 public void Execute(HttpServletRequest request) throws
NoSuchCommandException, CommandNameMissingException {
      Enumeration allNames=request.getParameterNames();
      Hashtable <String,Object> parameters=new Hashtable
<String,Object> ();
      while (allNames.hasMoreElements()) {
         String pname=(String)allNames.nextElement();
         String parmValue=request.getParameter(pname);
         parameters.put(pname, parmValue);
      }
      Execute(parameters);
   }
```

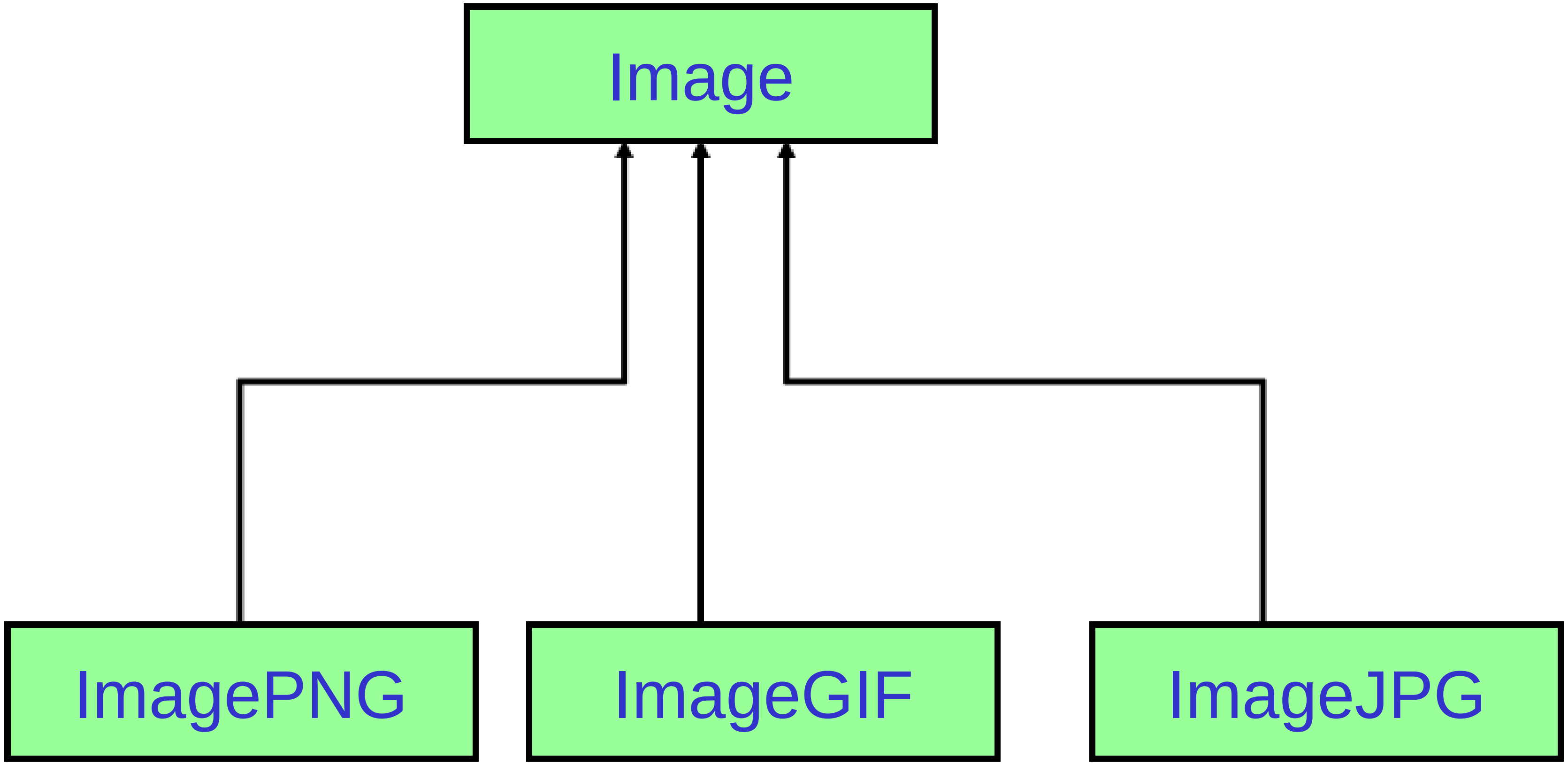
Factory class

- Factory method constructs instances of a class
- Problem
- Constructing a Image class
  - Image format could be png, gif, jpg
  - Each format could have different image class

  - Calling code needs to use different class depending on image type
  - ImagePNG image=new ImagePNG("/picture.png");
  - Type may not be know till runtime

Factory example

- Solution
  - Use inheritance from abstract class Image



```
public static  createImage(String fname) throws
Exception {
   if (fname.endsWith(“.gif”)) {
      return( (Image) new ImageGIF(fname) );
    }
    if (fname.endsWith(“.png”)) {
       return( (Image) new ImagePNG(fname) );
    }
    if (fname.endsWith(“.jpg”)) {
       return( (Image) new ImageJPG(fname) );
    }
    throw new Exception("Unknown image type
for file "+fname);
}
```

Singleton

- Single instance of class
- Constructor is private
- static final Class instance constructed when application loads
- or loaded only when need (lazy initialization)
- Examples of usage
  - to access database so that all threads go through one control point
  - Font class keeps memory load low

Singleton Example in Java

```
public class DbaseConnector {
   private static final DbaseConnector instance=new
   DbaseConnector();
   private DbaseConnector() {
           // database construction code…..
   }

   public static DbaseConnector getInstance() {
       return(instance);
  }
}
```

Singleton Example (lazy initialization)

```
public class DbaseConnector {
    private static DbaseConnector instance;
    private DbaseConnector() {
            // database construction code…..
    }
    public static DbaseConnector synchronized getInstance() {
        if (instance==null) {
            instance=new DbaseConnector();
        }
        return(instance);
    }
}
```

Wrapper classes

- Problem
  - Different external technologies to connect to
  - Example for database connection
    - ODBC (Microsoft)
    - JDBC (Java standard)
  - Other examples
    - External Credit card payment
    - Network connection (Java and Microsoft)
    - Data structure libraries

Wrapper classes

- Problem with coding directly
  - Code will end up messy
  - Hard to port
  - Hard to understand
- Benefits of wrapping code
  - easier to swap modules (e.g. CC function)
  - easier to implement standard functions (e.g. accountancy, error logs)

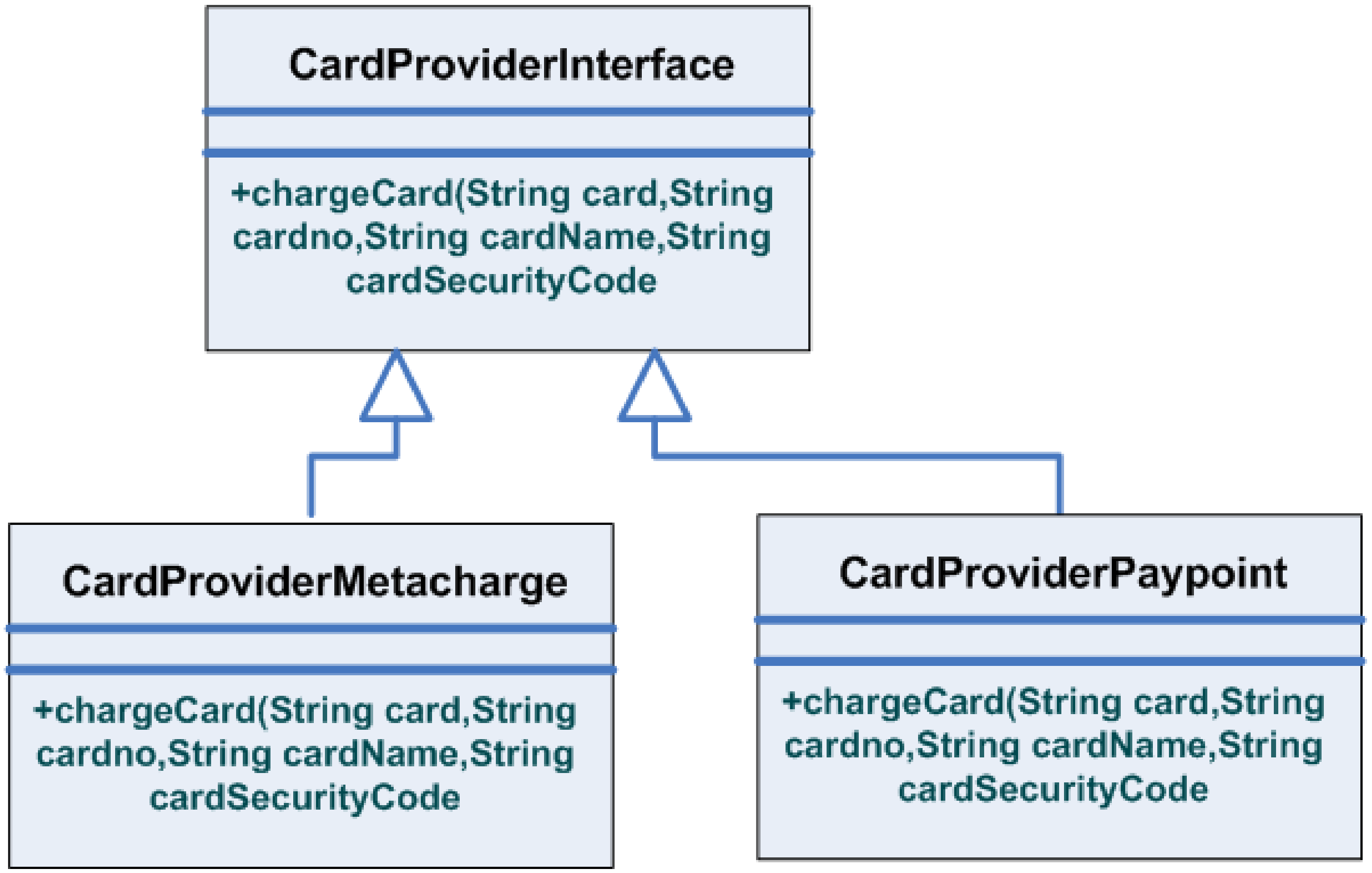
Wrapper example (unwrapped code)

```
String sql="select * from customers";
    try {
        java.sql.Statement
s=dbConnection.createStatement();
        int rows=s.executeUpdate(sql);

     } catch (Exception e) {
        status=sql+" "+e.toString();

     };
```

Wrapped code

```
public class SQLHelper {
  public void executeSQL(String sql) {
        try {
              java.sql.Statement
   s=dbConnection.createStatement();
              int rows=s.executeUpdate(sql);
        } catch (Exception e) {
          status=sql+" "+e.toString();
      };
}
}
```

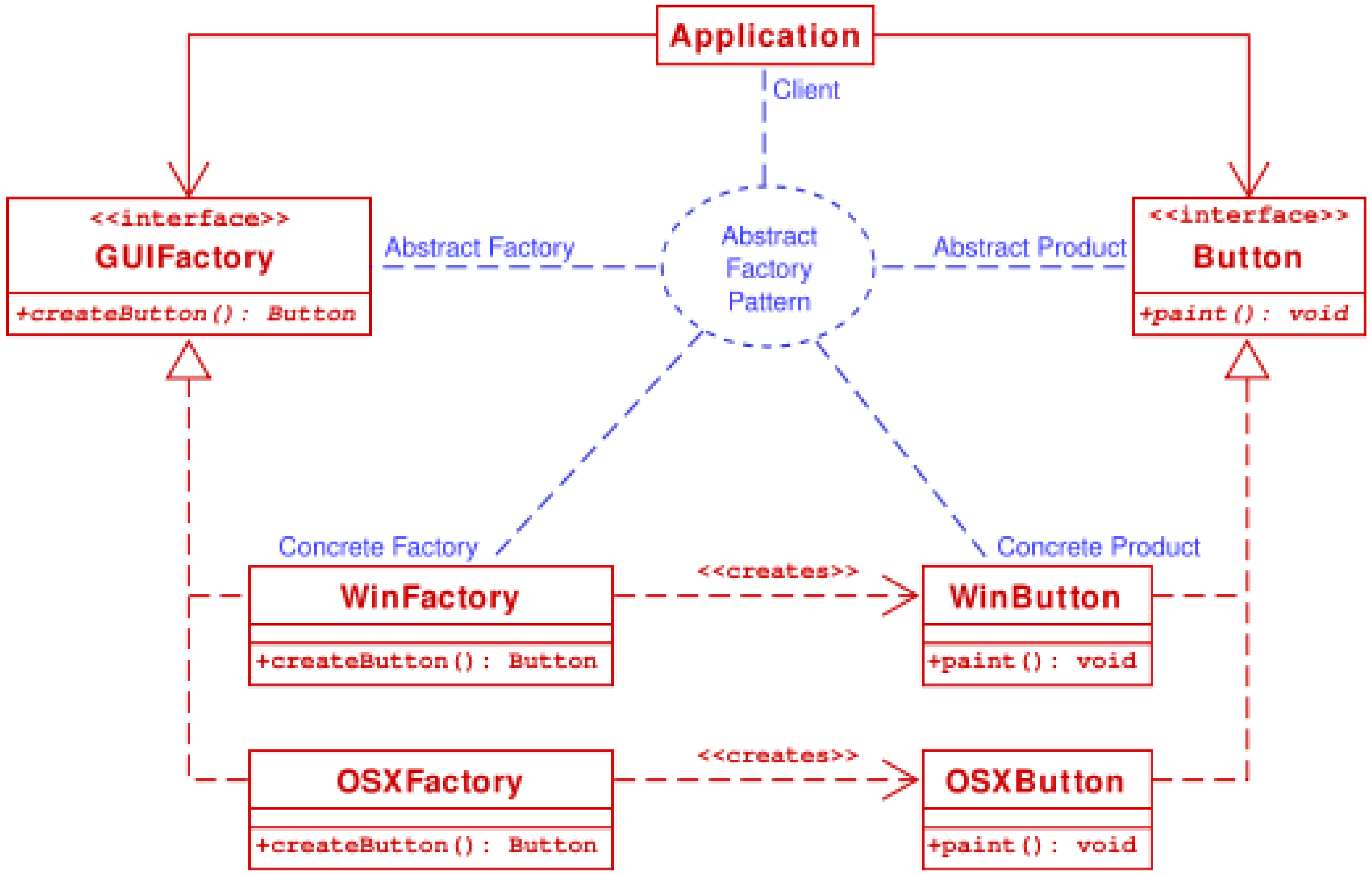
Adapter class diagram example


CardProviderInterface
+chargeCard(String card,String cardno,String cardName,String cardSecurityCode
CardProviderMetacharge
+chargeCard(String card,String cardno,String cardName,String cardSecurityCode
CardProviderPaypoint
+chargeCard(String card,String cardno,String cardName,String cardSecurityCode

Abstract factory

- Used when you have an associated set of object types to create, but the actual class to create is decide at run time
- Example:
  - Sets of encryption algorithms from different providers
  - User interface components for different OS UI API

Abstract Factory class diagram



Abstract factory code example

```
interface SecurityFactory {
   public Encryptor createEncryptor();
}
class LowSecurityFactory implement  SecurityFactory {
    public Encryptor createEncryptor() {
           return(new ShortKeyEncryptor());
    }
}
class  HighSecurityFactory implement SecurityFactory {
    public Encryptor createEncryptor() {
           return(LongKeyEncryptor());
    }
}
```

Abstract factory example

```
class Application {
        private Encryptor encryptor;
        public Application(securityFactory sfactory) {
            encryptor=sfactory.createEncryptor();
        }
}

class Start {
   public static void main(String argsv[ ]) {
      Application application;
      if (professionalVersion) {
          application=new Application(new HighSecurityFactory());
      } else {
          application=new Application(new LowSecurityFactory());
      }
   }
}
```