Software Engineering

Computer Science Tripos 1B
Michaelmas 2011

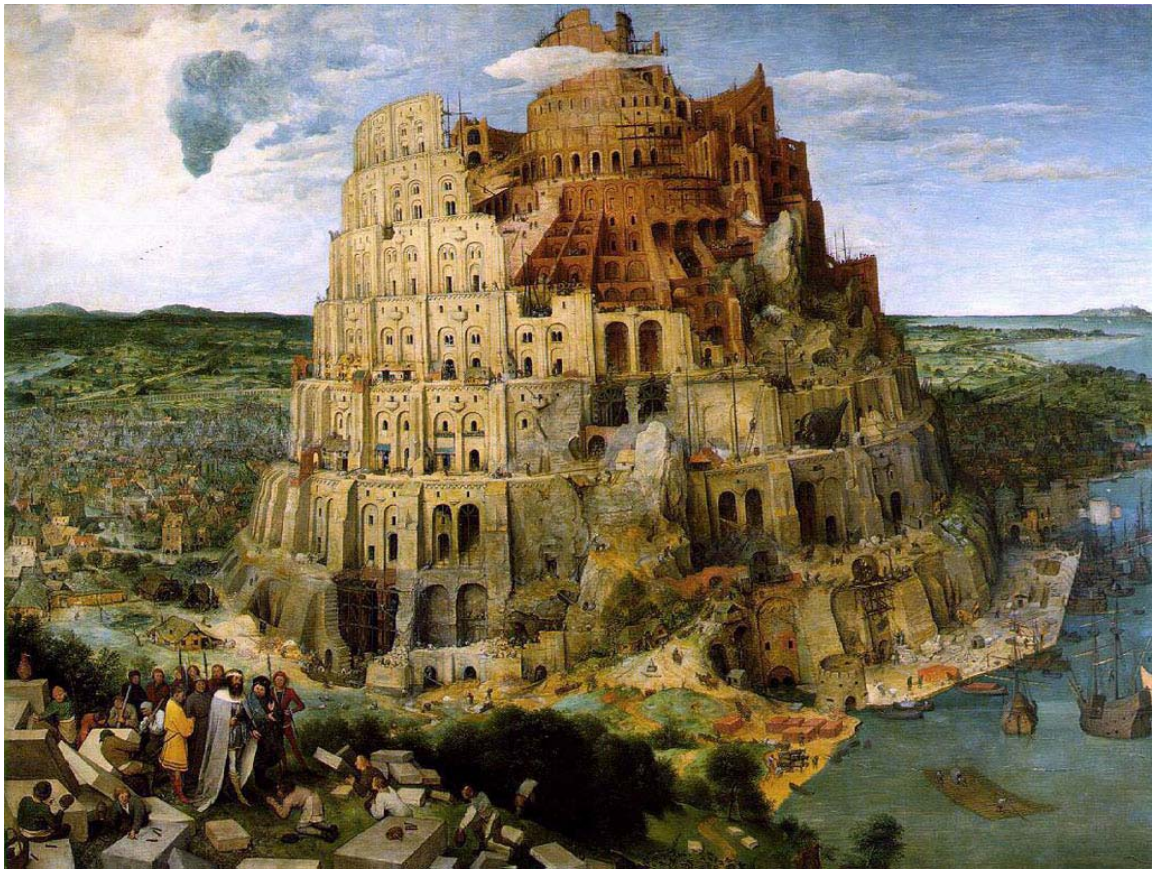
Richard Clayton

Lecture Two

Managing complexity

- Software engineering is about managing complexity at a number of levels
- At the micro level, bugs arise in protocols, algorithms etc. because they’re hard to understand
- As programs get bigger, interactions between the components grow at $O(n^2)$ or even $O(2^n)$
- ...
- With complex socio-technical systems, we can’t predict reactions to new functionality
- Most failures of really large systems are due to wrong, changing, or contested requirements

Project failure, c. 1500 BC



Complexity, 1870 – Bank of England

Complexity, 1876 – Dun, Barlow & Co

Complexity, 1906 – Sears, Roebuck

No. 23 MAKING A RECORD OF THE CUSTOMER'S ORDER.
SEARS, ROEBUCK & CO., Chicago, Ill.

No. 26 STENOGRAPHIC DEPARTMENT.
SEARS, ROEBUCK & CO., Chicago, Ill.

- Continental-scale mail order meant specialization
- Big departments for single book-keeping functions
- Beginnings of automation with dictation to recording cylinders and a typing pool to create outgoing letters

Complexity, 1940 – First National Bank of Chicago

1960s – The ‘software crisis’

- In the 1960s, large powerful mainframes made even more complex systems possible
- People started asking why project overruns and failures were so much more common than in mechanical engineering, shipbuilding, bridge building etc.
- Software Engineering as a field dates back to two NATO Science Committee run conferences in 1968 and 1969
- The hope was that we could get things under control by using disciplines such as project planning, documentation and testing

How is software different?

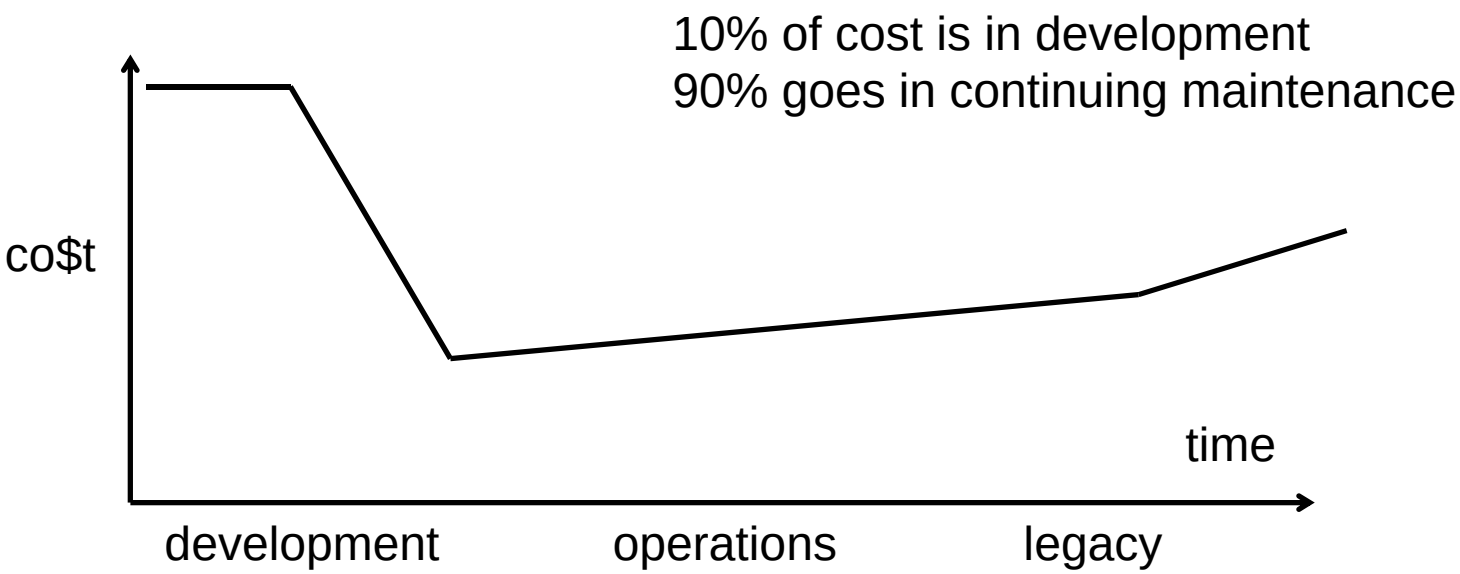
- Many things that make writing software fun also make it complex and error-prone:
  - the joy of solving puzzles and building things from interlocking moving parts
  - the stimulation of a non-repeating task with continuous learning
  - pleasure of working with a tractable medium, ‘pure thought stuff’
  - complete flexibility – you can base the output on the inputs in any way you can imagine
  - satisfaction of making stuff that’s useful to others

Software complexity

- Large systems become qualitatively more complex, unlike big ships or long bridges
- The tractability of software leads customers to demand ‘flexibility’ and frequent changes
- Thus systems also become more complex to use over time as ‘features’ accumulate
- The structure can be hard to visualise or model
- The hard slog of debugging and testing piles up at the end, when the excitement’s past, the budget’s spent and the deadline’s looming

The software life cycle

- Software economics can get complex
  - consumers buy on sticker price, businesses on total cost of ownership
  - vendors use lock-in tactics, with complex outsourcing issues
- First let's consider the simple (1950s) case of a company that develops and maintains software entirely for its own use:



What does code cost?

- First IBM measures (60s), in lines of code per man-year
  - 1,500 LOC/my (operating system)
  - 5,000 LOC/my (compiler)
  - 10,000 LOC/my (app)
- AT&T measures
  - 600 LOC/my (compiler)
  - 2,200 LOC/my (switch)
- So only a handful of lines of code per day
  - this of course includes all specification, debugging, testing
- Alternatives
  - Halstead (entropy of operators/operands)
  - McCabe (graph entropy of control structures)
  - Function point analysis

First-generation lessons learned

- There are huge variations in productivity between individuals
  - my own experience (1970s-1990s), at least a factor of 10 between ‘the best’ and those you are not prepare to dismiss as ‘useless’
- The main systematic gains have come from using an appropriate high-level language
  - high level languages take away much of the accidental complexity, so the programmer can focus on the intrinsic complexity
- It’s also worth putting extra effort into getting the specification right, as it more than pays for itself by reducing the time spent on coding and testing

Development costs

- Barry Boehm, 1975:

	Spec	Code	Test
C3I	46%	20%	34%
Space	34%	20%	46%
Scientific	44%	26%	30%
Business	44%	28%	28%

- So the toolsmith needs to focus on more than just code!

The ‘Mythical Man Month’

- Fred Brooks (manager of the IBM OS360 programme) debunked interchangeability:
- Imagine a project which is going to take 3 men 4 months, with one month of design and 3 months of code and test
- But if the design work takes an extra month, we only have 2 months left to do 9 man-months of work.
- But if training someone takes a month, we must add 6 men
- But the work 3 men could do in 3 months just can’t be done by 9 men in 1 month! Interaction costs maybe $O(n^2)$
- Hence Brooks’ law:
  - adding manpower to a late project makes it later!

Software engineering economics

- Boehm, 1981 (empirical studies after Brooks) developed the COCOMO model which has seen wide commercial use
  - COCOMO = Constructive COst MOdel
- Boehm's cost-optimum schedule time to first shipment
  - $T = 2.5 * (\text{man-months})^{1/3}$
  - exponent of '1/3' varies slightly with the experience of team and type of project. Total man months clearly varies rather more!
  - with more time for the project, cost rises slowly
  - with less time available, it rises sharply
- Hardly any projects succeed in less than 3/4 T
- Other studies show that if people are to be added, you should do it early rather than late
- Some projects fail despite huge resources!

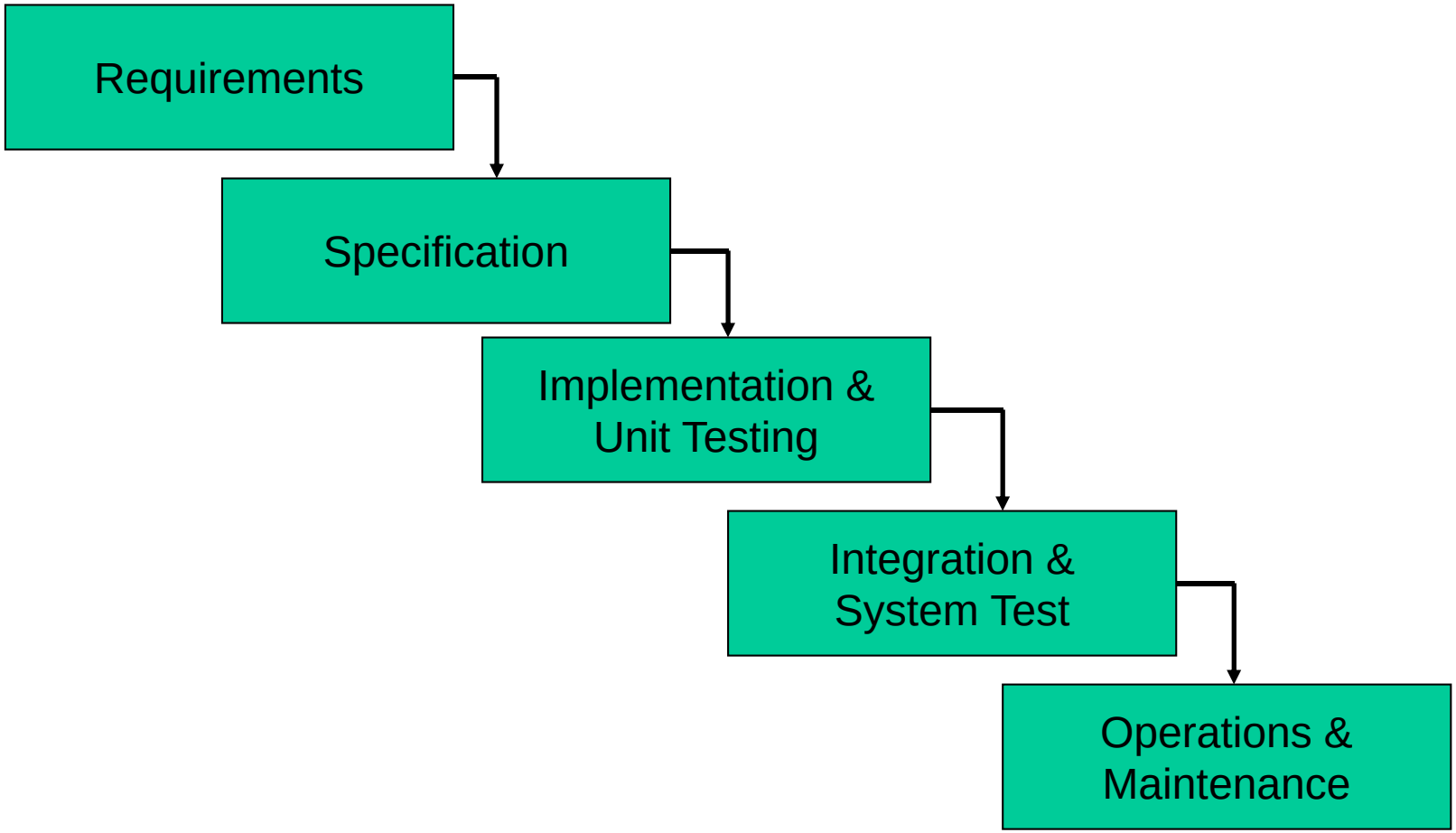
The software project ‘tar pit’

- You can pull any one of your legs out of the tar …
- Individual software problems all soluble but …

Structured design

- The only practical way to build large complex programs is to chop them up into modules
- Sometimes task division seems straightforward (bank = tellers, ATMs, dealers, …)
- Sometimes it isn’t
- Sometimes it merely appears to be straightforward …
- Quite a number of methodologies have been developed (SSDM, Jackson, Yourdon, …)

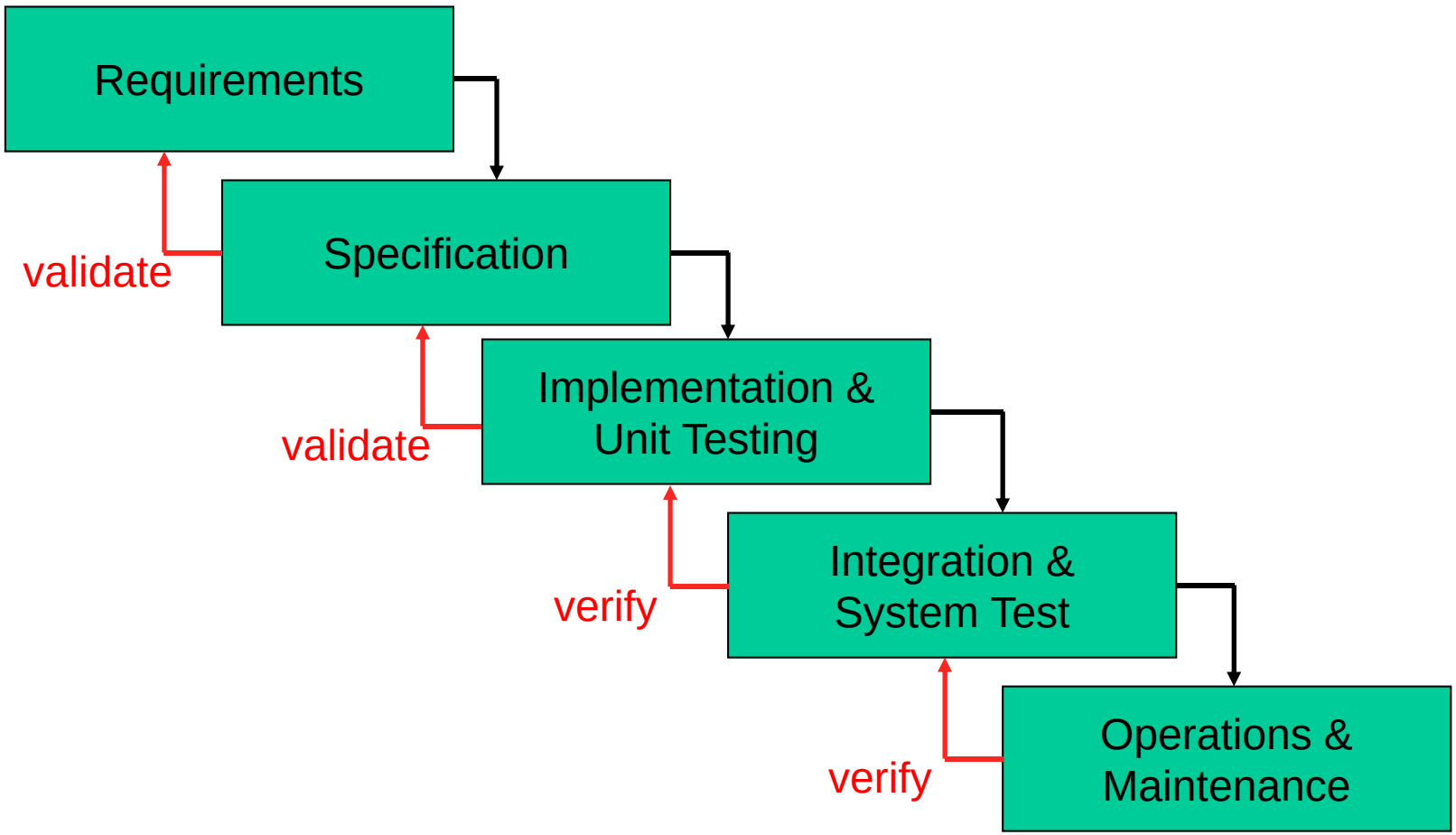
The Waterfall Model I



The Waterfall Model II

- Requirements are written in the user's language
- The specification is written in system language
- There can be many more steps than this – system spec, functional spec, programming spec …
- The philosophy is to progressively refine the description of what the user wants
- Warning – when Winton Royce published this in 1970 he cautioned against naïve use
- But it become a US DoD standard …

The Waterfall Model III



The Waterfall Model IV

- People often suggest adding an overall feedback loop from operations back to requirements
- However the essence of the waterfall model is that this isn't done
- It would erode much of the value that organisations get from top-down development
- Very often the waterfall model is used only for specific development phases, e.g. adding a new feature
- But sometimes people use it for whole systems

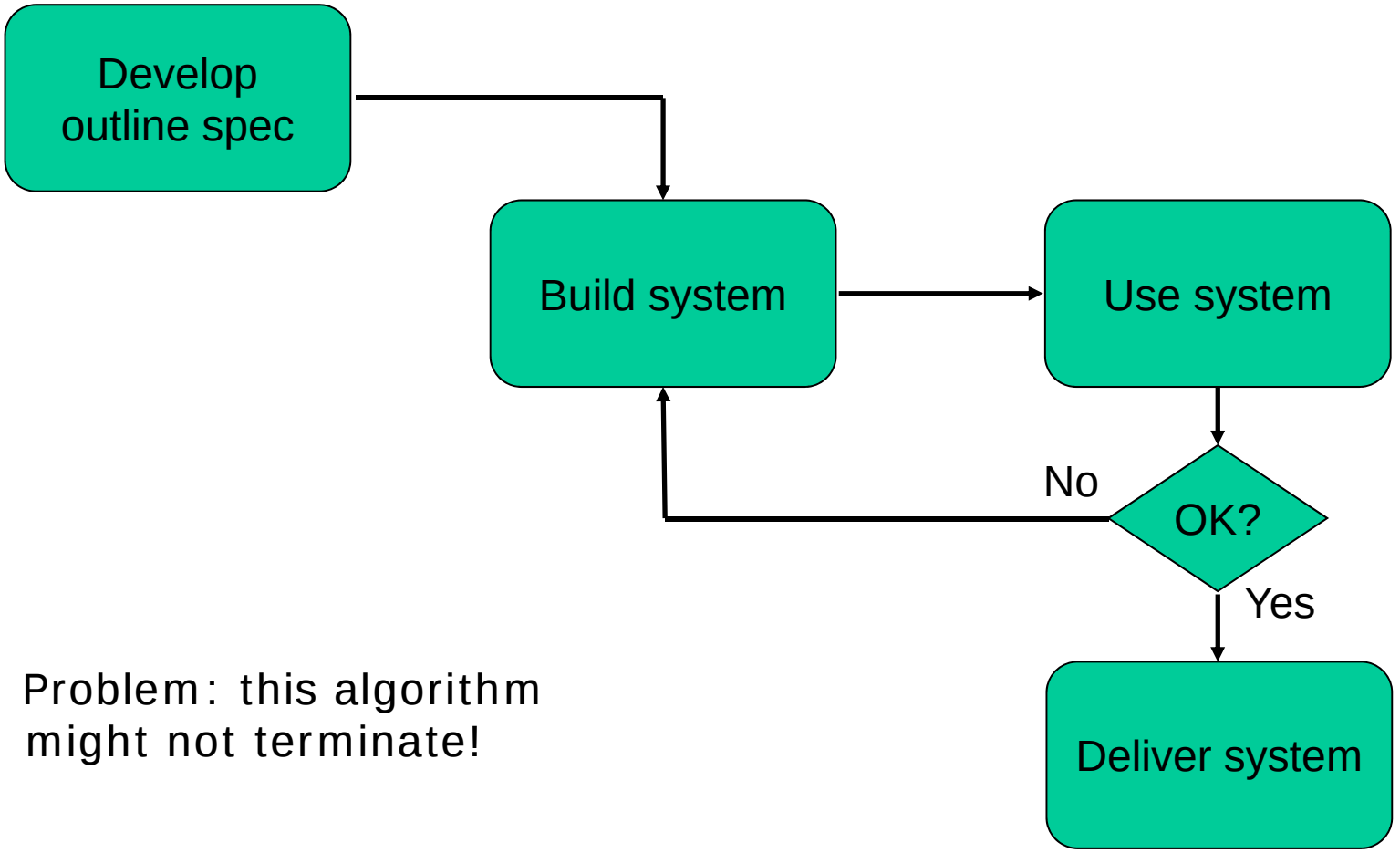
Advantages of the Waterfall Model

- Compels early clarification of system goals and is conducive to good design practice
- Enables the developer to charge for changes to the requirements
- It works well with many management tools, and technical tools
- Where it’s viable it’s usually the best approach
- The really critical factor is whether you can define the requirements in detail in advance. Sometimes you can (Y2K bugfix); sometimes you can’t (HCI)

Objections to the Waterfall Model

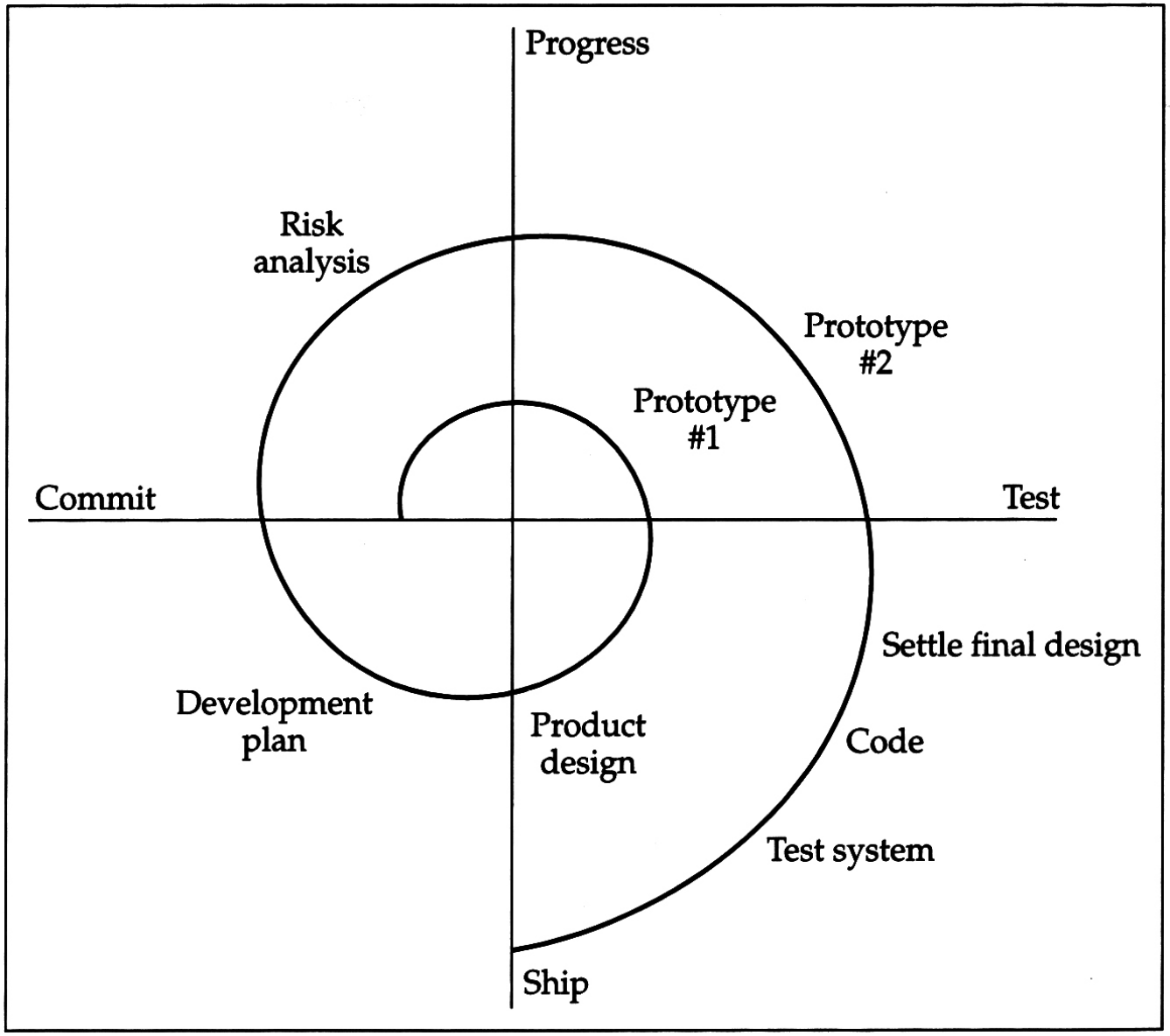
- Iteration can be critical in the development process:
  - requirements not yet understood by developers
  - or not yet understood by the customer
  - the technology is changing
  - the environment (legal, competitive) is changing
- The attainable quality improvement from the focus on getting design right from the beginning may be unimportant over the system lifecycle
- Specific objections from safety-critical, package software developers

Iterative Development



Problem: this algorithm might not terminate!

Spiral Model I



Spiral Model II

- The essence of the spiral is that you decide in advance on a fixed number of iterations
  - e.g. engineering prototype, pre-production prototype, then product
  - cf Brookes “Plan to throw one away, you will anyway”
- Each of these iterations is done top-down
- “Driven by risk management”, i.e. you concentrate on prototyping the bits you don’t understand yet
- Important to resist pressure from customers who may not understand engineering limitations of the prototypes and want to see one ‘just tidied up’ and delivered

Evolutionary Model

- Products like Windows and Office are now so complex that they ‘evolve’ (MS tried twice to rewrite Word from scratch and failed)
- The big change that has made this possible has been the arrival of automatic regression testing
- Firms now have huge suites of test cases against which daily builds of the software are tested
- The development cycle is to add changes, check them in, and test them (“check in early, check in often”)
- The guest lecture will discuss this
- NB: you may not use the same methodology everywhere … you may be able to build device drivers “bottom up” to conform to a static pre-determined design – but include them in an top-down development of an evolving system