The Software Development Process

A personal view

Dr Robert Brady

CTO, Brady plc

Science Park

Cambridge

r.brady@bradyplc.com

Key references:
"Debugging the Development Process"
S Maguire, Microsoft Press

"Showstopper"
G Pascal Zachary, Macmillan

The three most important things in software development

1. Bugs

2. Bugs

3. Bugs

My agenda today:-

- Why is software development hard?
- Managing the code
- Development team ground-rules
- Making the management decision to ship

Why is software development hard?

How to lose $70bn

In the late 1980s, IBM lost $70 billion of stock-market value, and gave an entire market away to a previously small company called Microsoft.

According to the popular book “Big Blues”, this was, amongst other things, because it couldn’t write software effectively.

But IBM “did it right”. It followed all the standard rules taught in computer science courses at the time:

- Get the design right before you write the code
- Write complete documentation
- Get it right first time
- Use formal methods, design walk-throughs etc. to satisfy yourself that the code is bug-free

So what went wrong?

Size is important

Bytes

100b -1kb	Typical punch-card program (The IBM development method was probably developed for this type of program)
2kb-10kb	Typical software module Typical computer science project(?)
16kb	Operating system of Sinclair Spectrum
200Kb	Our first software product – 1986
18 Mb	Human Genome – active code (30k genes * protein size 800)
64Mb	Xbox RAM
100Mb	Our current software product (code)
750Mb	Human genome - including rubbish code (3×10^9 base-pairs)
1Gb – 2 Gb	Windows 2000 with associated products
40Gb	Storage on small laptop

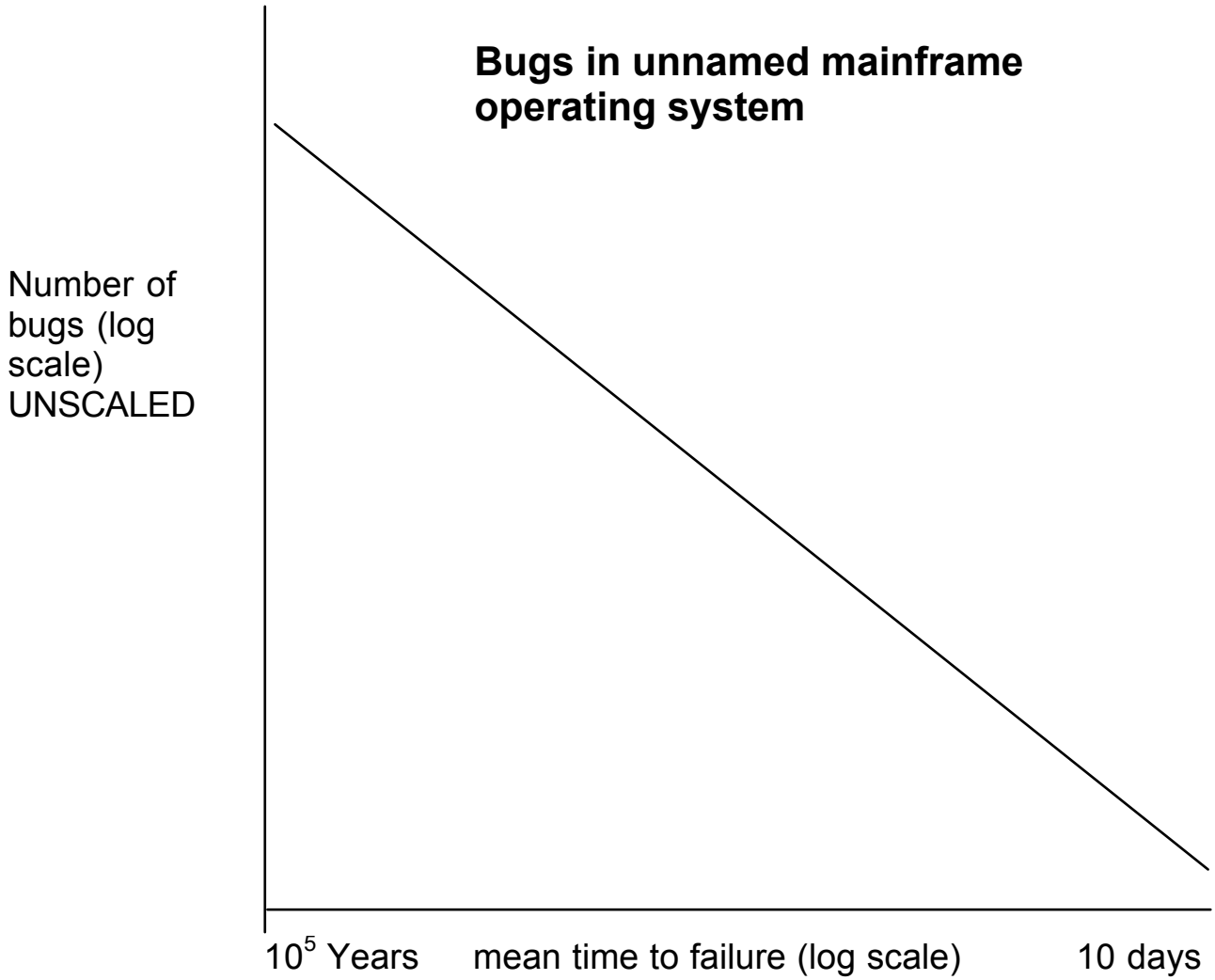
How size affects the basic assumptions

	Punch-card program	2kb of code	Large program
Complete the design in advance	Almost essential	Difficult	Too complex - not possible
Complete the documentation in advance	Highly desirable	Difficult	Too complex - not possible
Prove it is bug-free	Very difficult mathematical challenge	Too complex - not possible	Too complex - not possible
“Right first time”	A worthy goal	Too complex - not possible	Too complex - not possible

Conclusion: bugs are inevitable

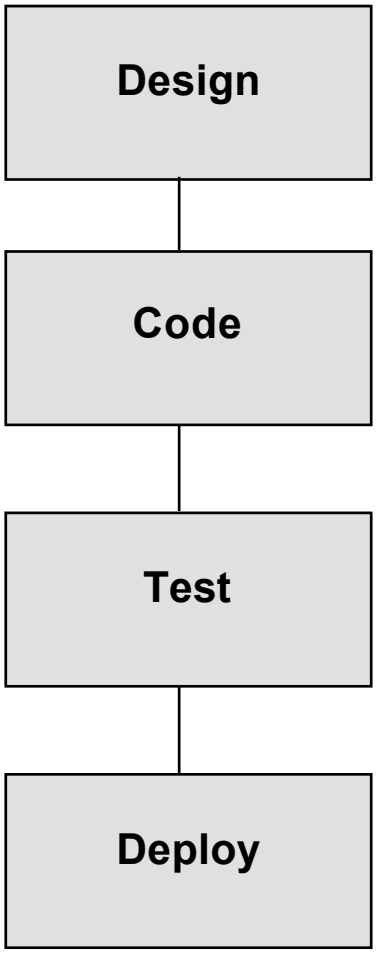
IBM's depressing research on bugs

There are lots and lots of very obscure bugs that are very hard to find.



Managing the code

Waterfall model

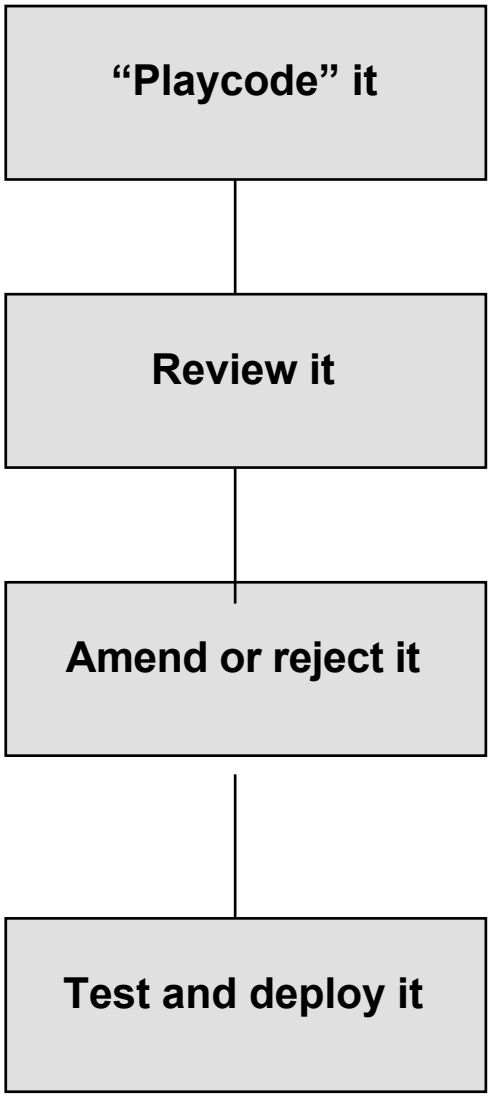


Strengths and pitfalls of the Waterfall Model

Good for small modules or sub-units, particularly if you can have simple and well-specified interface.

- IBM implemented this model by having DIFFERENT people in each stage. This gave people posh-sounding job titles (“Analyst” etc.), but caused very bad communication that killed their projects.
- Like Microsoft, we have a policy: “We do not have any programmers” We have developers. They are responsible for seeing the whole thing through.

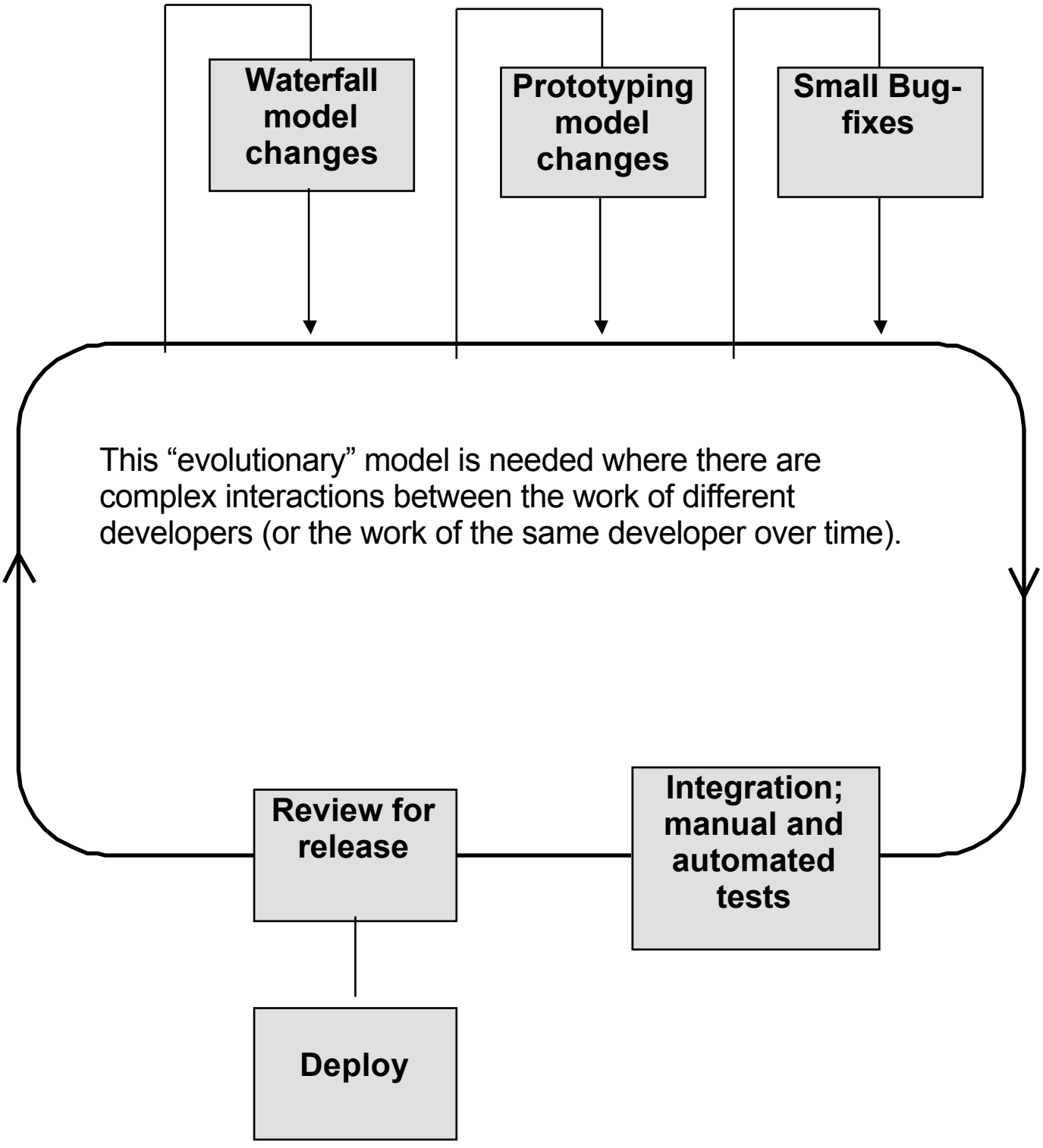
The “Prototype” Model



Strengths and pitfalls of the “Prototype” model

- Good where there are significant project risks or unknowns - e.g. external software, new techniques or methods, or can’t decide between alternatives.
- Not very predictable (a big problem in contracted developments)

The “evolutionary” model
- what everyone does in practice
(whatever they call it)



Development team ground-rules

A quiz

You are the manager of a small (2 person) software development/test team. They come to you with a problem and a proposed solution. Do you approve it?

Problem

- We need to implement 10 features. We have reviewed the designs, we now need to code and test them.
- Time is very tight. We will have to pull out all the stops to do it by the contracted deadline of next month
- John (the developer) is the best person to do the coding
- Richard (the test engineer) is the best person to do the testing

Proposed solution

- John and Richard work closely together to accelerate the development phase
- John codes the features and makes quick releases to Richard during development
- Richard provides testing feedback during development
- After this development phase, the software goes into the normal release cycle for testing/bugfix

If you approve the plan

- You will send a message to your developers that bugs don't matter – you can "throw them over the wall" and someone else will find them for you

- You will accelerate developers who produce sloppy code and slow down developers who produce good code

- The process will be inefficient, eg
  - the developer has a rough idea which areas will be buggy, he can home in on these
  - The developer has tools ("debuggers") to find bugs which the tester doesn't have
  - The developer will have to constantly communicate with the tester on what's changed, this slows them both down
  - The tester will be inefficient because silly bugs will stop him running his automated tests

- When you get to the original deadline
  - your project will probably have all the features
  - but the product probably won't work well enough to run the automated tests, so you cannot ship
  - You won't be able to advise the customer of the new ship date, because the automated tests don't work and they might (or might not) uncover something when they do run
  - It will be too late to take corrective action

If you reject the plan (developer has to test his code before release)

- Your team will be forced to make the hard project decisions, eg
  - Go back to the design stage for feature number 3 – can we implement it more simply?
  - Cut feature number 6 – it's not strictly in the specification
  - Advise the customer there is a risk. Does he want a delay or does he want feature number 7 in a later release?
  - Request more resources (a long shot…)

- Your team will work more efficiently
  - The tester will always work on code that is basically stable (so he can develop his regression tests etc.)
  - The developer will be rewarded for producing quality code, not for producing features that destabilise the product

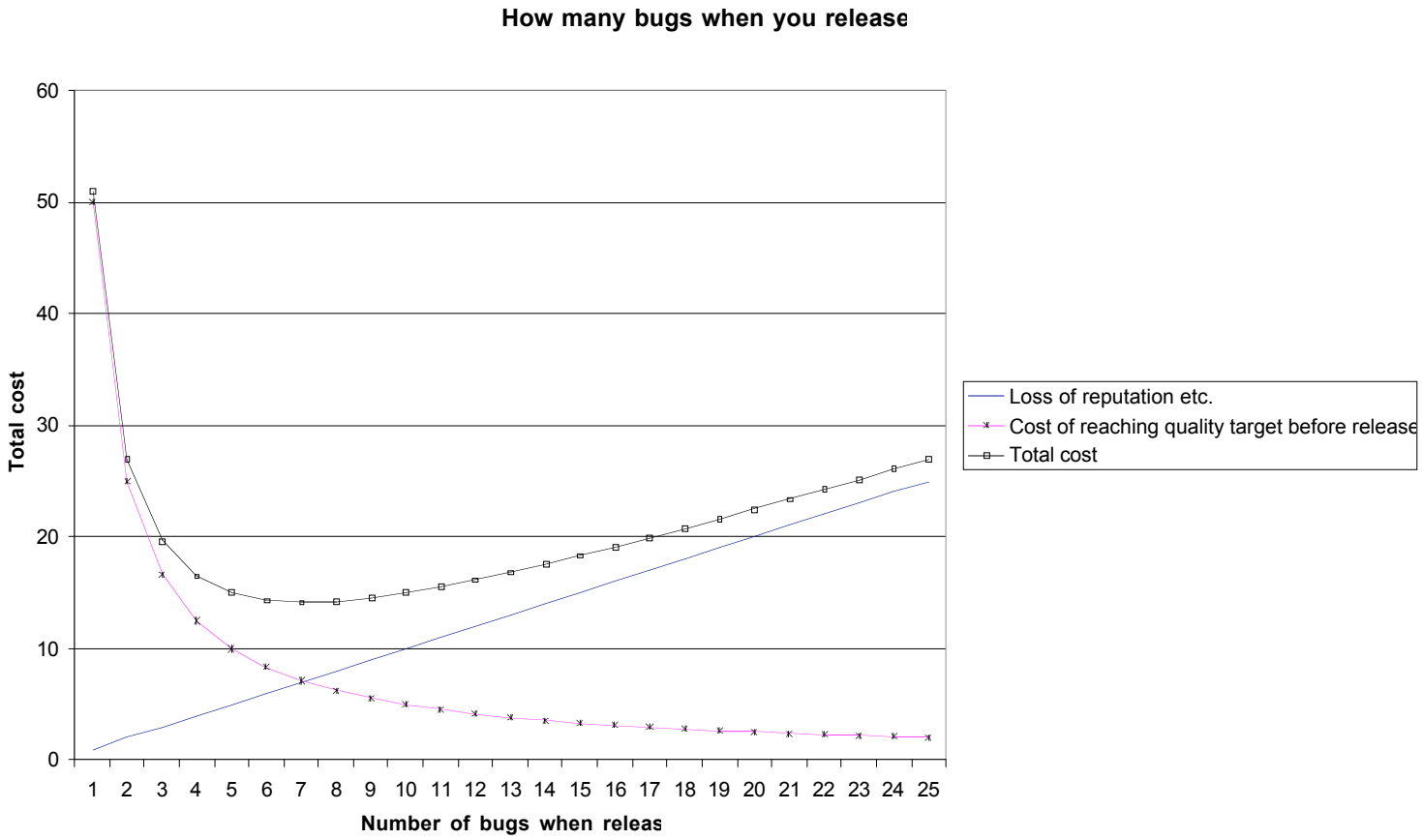
- Your team will be able to plan the project
  - If a feature is in the product then it will "basically work"
  - The team (and you) can now monitor progress
  - You can get test results and customer feedback early on the features you have implemented
  - Management can make the decision to ship with a more predictable freeze-time

Making the management decision to ship

Standard quality engineering decision

Release at time of lowest total cost

"Handbook of software quality engineering"



Windows NT version 1

Probably released at lowest cost point (as above)

5.6M lines of code

1 bug for every 10 lines = 560K bugs to fix
1 bug for every 100 lines = 56K bugs to fix

Management's major activity was prioritising bugs
- Showstopper (always fixed)
- Priority 1 (fixed except in late stages of release)
- Priority 2 (deferred)

Date	Release	"Serious or showstopper" bugs
12 Oct 1992	Beta 1	2,000 known on release
8 Mar 1993	Beta 2	0 known on release
		263 found in first 6 days
26 Jul 1993	Final	0 showstoppers known at release

Reference: "Showstopper" (G Pascal Zachary)

The decision to ship (3)

My personal view

Avoid the "first release" syndrome altogether if possible

- Make the first release very small
- Make regular upgrades (SP or beta versions)
- Typically monthly or quarterly releases
- Each release contains only small changes

Essential to limit risks OF EACH RELEASE

- Invest in automated regression tests
- The risk of each change is the primary focus
- Manage higher risk changes by breaking them up
  - eg Separate the code from existing code
  - eg Have the ability to switch the risky part on/off
  - eg implement a big change in smaller bits

Get real customers for each release

- Forces focus on what the customer really requires
- Gets real-world feedback that no lab can reproduce
- Problems? Add to automated regression tests

Criterion for each release

- "better than the previous"
- Pass automated regression tests
- Pass manual tests of new functionality