IEC 61508-conformant software development with SPARK

Peter Amey

Agenda

- Introductions
- Some thoughts on standards
- Static analysis
- Unambiguous languages
- Formal methods
- Putting it all together

Agenda

- Introductions
- Some thoughts on standards
- Static analysis
- Unambiguous languages
- Formal methods
- Putting it all together

Praxis HIS

- specialists in the engineering of high-integrity and safety-critical, software-intensive systems
- delivers services by:
  - provision of tools such as the SPARK Examiner
  - outsourcing complete projects
  - capability enhancement
  - providing key consultancy
- 100+ technical staff and growing
- autonomous but part of the Altran engineering consultancy group
- founded specifically to put engineering into software engineering
  - first software house in the world to achieve ISO 9001 certification

Peter Amey

- Chief Technical Officer at Praxis HIS
- prime author of SPARK Examiner
  - 1992 onwards with Program validation Ltd
  - with Praxis since 1995
- worked on wide range of critical systems projects
  - avionics
  - transport
  - security
- aeronautical engineer, served in Royal Air Force
- worked on armament software safety and certification at (what was then the) A&AEE, Boscombe Down
  - (UK equivalent of Erprobungsstelle 61 at Manching)

Agenda

- Introductions
- Some thoughts on standards
- Static analysis
- Unambiguous languages
- Formal methods
- Putting it all together

A confession

- I am not very interested in 61508
  - despite being asked here to talk about it
- I am not very interested in DO-178B
  - despite being on the committee currently revising and updating it
- I am not very interested in Def Stan 00-55
  - despite helping write it

- Here is a standard I would sign up to:

My Standard

The fitness for purpose of a software program shall be established by logical reasoning

Something that met this standard would meet all of the others as well

Development for Different SILs

IEC 6 1508

	SIL1	SIL2	SIL3	SIL4
Specification	Informal	Informal	Semi-Formal	Formal
Prototyping	R	R	R	R
Coding	HLL Preferred	HLL	Safe-Subset HLL	Safe-Subset HLL
Defensive Code	-	R	HR	HR
Static Analysis	R	HR	HR	HR
Formal Proof	-	R	R	HR
Dynamic Testing	R	HR	HR	HR
Performance Testing	R	R	HR	HR

Development for Different SILs

IEC 6 1508

	SIL1	SIL2	SIL3	SIL4
Specification	Informal	Informal	Semi-Formal	Formal
Prototyping	R	R	R	R
Coding	HLL Preferred	HLL	Safe-Subset HLL	Safe-Subset HLL
Defensive Code	-	R	HR	HR
Static Analysis	R	HR	HR	HR
Formal Proof	-	R	R	HR
Dynamic Testing	R	HR	HR	HR
Performance Testing	R	R	HR	HR

What about actually doing it?

IEC 6 1508

	SIL1	SIL2	SIL3	SIL4
Specification	Informal	Informal	Semi-Formal	Formal
Prototyping	R	R	R	R
Coding	HLL Preferred	HLL	Safe-Subset HLL	Safe-Subset HLL
Defensive Code	-	R	HR	HR
Static Analysis	R	HR	HR	HR
Formal Proof	-	R	R	HR
Dynamic Testing	R	HR	HR	HR
Performance Testing	R	R	HR	HR

Agenda

- Introductions
- Some thoughts on standards
- Static analysis
- Unambiguous languages
- Formal methods
- Putting it all together

7.9 Software verification

…

7.9.2.12 Code verification: the source code shall be verified by static methods to ensure conformance to the specified design of the software module (see 7.4.5), the required coding standards (see 7.4.4), and the requirements of safety planning (see 7.3).

7.9 Software verification

…

7.9.2.12 Code verification: the source code shall be verified by static methods to ensure conformance to the specified design of the software module (see 7.4.5), the required coding standards (see 7.4.4), and the requirements of safety planning (see 7.3).

Disadvantages of Dynamic Testing

- Practical disadvantages
  - takes place late in development
  - hard to diagnose unexpected behaviour
  - frequently a bottleneck (e.g. shared use of test rig)
  - very expensive
  - significant source of project risk

- Theoretical limitations
  - high levels of confidence require mathematically infeasible amounts of testing

Theoretical Limitations of Testing

for ultra-high reliability ($<10^{-7}$ failures per hour)

- Bayesian mathematics definitively limits what we can claim from statistical testing
- Reliability growth models cannot provide necessary assurance
- Proofs:
  - Butler & Finelli. 1993 (see references)
  - Littlewood & Strigini. 1993 (see references)

Advantages and Disadvantages of Analysis

- Advantages
  - can be used early in the development process
  - can establish properties that cannot be demonstrated in any other way. e.g.
    - proof of absence of run-time errors
    - freedom from timing deadlocks
- Disadvantages
  - can only compare artefacts (e.g. code against specification)
  - what can be achieved is limited by precision of descriptions and notations used

Static analysis - the catch

- We need analysis that is:
  - sound
    - all errors of a particular kind found
    - low false alarm rate
  - fast
  - usable early in development
- What we can achieve depends on the properties of the language we are analysing
- Analysis of general purpose languages cannot achieve these goals

Agenda

- Introductions
- Some thoughts on standards
- Static analysis
- Unambiguous languages
- Formal methods
- Putting it all together

7.4.4.3 To the extent required by the safety integrity level, the programming language selected

shall:

...

b) be completely and unambiguously defined or restricted to unambiguously defined features;

...

d) contain features that facilitate the detection of programming mistakes; and

...

7.4.4.3 To the extent required by the safety integrity level, the programming language selected

shall:

...

b) be completely and unambiguously defined or restricted to unambiguously defined features;

...

d) contain features that facilitate the detection of programming mistakes; and

...

Unambiguous source code?

- But: all commonly used languages allow the construction of programs of uncertain meaning
  - ambiguities
- Most also have uncheckable rules
  - insecurities

Causes of Uncertainty

- Deficiencies in language definitions
  - Semantics of C integer division
  - Pascal named vs. structural type equivalence
- Implementation freedoms
  - term/expression evaluation order
  - parameter-passing mechanisms

Example of an Ambiguity

```
z := F(x) + G(y);
```

Suppose function **F** modifies **y** as a side-effect of its operation. In this case the meaning of the expression depends on whether **F** or **G** is evaluated first.

A rule stating "functions are not permitted to have side effects" turns the **ambiguity** into an **insecurity**: it does not solve the problem

A Simple C Ambiguity

```
i = v[i++];
```

Page 46 of the C++ Annotated Reference Manual states that this leads to the value of **`i`** being undefined.

A Simple Ada Ambiguity

```
procedure Init2(X, Y : out Integer)
is
begin
   X := 1;
   Y := 2;
end Init2;
```

What is the meaning of:

```
Init2(A, A);
```

Resolution

- Ambiguities are resolved .... by compiler authors
- Insecurities are left for the user to discover
- Possible solutions
  - Invent new languages without these problems
  - Work with dialects associated with compilers
  - Use logically coherent language subsets to overcome ambiguities and insecurities

Safe Subsets

- Potentially give us the best of both worlds:
  - logical soundness and predictability
  - access to standard compilers, tools, training, staff

But

- Level of integrity achievable depends on foundation language chosen
- Subsetting alone may not be enough for the highest integrity levels

Constructing a Safe Subset

- Selection of base language
- Removal of the most troublesome language features
- Limitations on the way remaining features may be used
- Introduction of annotations to provide extra information

The Subset Spectrum

- We can construct subsets that vary on 4 axes:
  - Precision (security and lack of ambiguity)
  - Expressive power
  - Depth of analysis possible
  - Efficiency of analysis process

The Subset Spectrum (contd.)

- Trade-offs quite complex; we are trying to avoid surprise: unexpected behaviour which we don’t find until test
  - removing problematic features may reduce this risk
  - increased precision may require reduction in expressive power but improves depth of analysis
  - we may be able to combine expressiveness with depth of analysis but at the cost of efficiency of analysis

The Subset Spectrum (contd.)

- Fundamental trade-off is between discipline we accept to reduce bug insertion and the effort we are prepared to make in bug detection
- For example:
  - unrestricted C provides little or protection from bug insertion
  - Ada requires extra discipline (e.g. strong typing) which reduces bug insertion rate
- A qualitative shift in what is possible only occurs when precision becomes exact

SPARK

- A sub-language of Ada with particular properties that make it ideally suited to the most critical of applications:
  - Completely unambiguous
  - Free from implementation dependencies
  - All rule violations are detectable
  - Formally defined
  - Tool supported
- SPARK subsets of both Ada 83 and Ada 95 are defined
- Language definition is open and widely available

"... one could communicate with these machines in any language provided it was an exact language ..."

"... the system should resemble normal mathematical procedure closely, but at the same time should be as unambiguous as possible."

Alan Turing, 1947

(lecture to the London Mathematical Society on the "Automatic Computing Engine")

SPARK

- A sub-language of Ada with particular properties that make it ideally suited to the most critical of applications:
  - Completely unambiguous
  - Free from implementation dependencies
  - All rule violations are detectable
  - Formally defined
  - Tool supported
- SPARK subsets of both Ada 83 and Ada 95 are defined
- Language definition is open and widely available

Constructing a Safe Subset - SPARK

- Selection of base language
  -
- Removal of the most troublesome language features
  -
- Limitations on the way remaining features may be used
  -
- Introduction of annotations to provide extra information
  -

Constructing a Safe Subset –SPARK

- Selection of base language
  - ANSI/MIL-STD-1815A-1983 & ISO-8652:1995
- Removal of the most troublesome language features
  - e.g. unrestricted tasking
- Limitations on the way remaining features may be used
  - e.g. limitations on placement of exit and return
- Introduction of annotations to provide extra information
  - core (e.g. Global) and proof (e.g. Post) annotations

Why Annotations?

- Annotations strengthen specifications
  - providing design-by-contract facilities
- Allows analysis without access to procedure-bodies
  - which can be done early during development
  - before programs are complete or compilable
- Erroneous constructs are efficiently detected

An example (detection of erroneous constructs)

```
procedure Inc (X : in out Integer);
--# global in out Callcount;
```

detection of function side-effect

```
function AddOne (X : Integer)
    return Integer is
  XLocal : Integer := X;
begin
  Inc (Xlocal);
  return XLocal;
end AddOne;
```

detection of aliasing

```
Inc (CallCount);
```

SPARK supports static verification

- These methods can ensure:
  - freedom from language misuse
  - freedom from data and information flow errors
  - freedom from run-time errors
  - specified safety properties are guaranteed
- Source is robust and contains few errors
- Source can be:
  - directly host-tested
  - directly cross-compiled to target
  - used to generate alternate language

Example: run-time error proof

```
type T is range 0 .. 100;

procedure Inc (X : in out T)
--# derives X from X;
is
begin
   X := X + 1;
end Inc;
```

Unsimplified verification condition

```
procedure_inc_1.
H1:    true .
H2:    x >= t__first .
H3:    x <= t__last .
        ->
C1:    x + 1 >= t__first .
C2:    x + 1 <= t__last .
```

Example: run-time error proof

```
type T is range 0 .. 100;

procedure Inc (X : in out T)
--# derives X from X;
is
begin
   X := X + 1;
end Inc;
```

Simplified verification condition

```
procedure_inc_1.
H1:     x >= 0 .
H2:     x <= 100 .
        ->
C1:     x <= 99 .
```

Can’t be proved - problem!

Solutions

```
type T is range 0 .. 100;

procedure Inc (X : in out T)
--# derives X from X;
is
begin
   if X < T'Last then
      X := X + 1;
   end if;
end Inc;
```

Defensive programming

```
type T is range 0 .. 100;

procedure Inc (X : in out T)
--# derives X from X;
--# pre X < T'Last;
is
begin
   X := X + 1;
end Inc;
```

Logical guard

SPARK Results - C130J - Lockheed

- “Very few errors have been found in the software during even the most rigorous levels of FAA testing, which is being successfully conducted for less than a fifth of the normal cost in industry”
- “This level A system was developed at half of typical cost of non-critical systems”
- Productivity gains: X4 on C130J compared to previous safety-critical projects, X16 on C27J with re-use and increased process maturity
- SPARK code was found to have only 10% of the residual errors of full Ada and Ada was found to have only 10% of the residual errors of C

Agenda

- Introductions
- Some thoughts on standards
- Static analysis
- Unambiguous languages
- Formal methods
- Putting it all together

IEC 61508: Software Detailed Design

0:	none
1:	structured methodology (CORE, JSD, MASCOT, Yourdon)
2:	+ semi-formal methods (function block diagrams, data-flow diagrams, pseudo code)
3:	+ formal methods (VDM, Z, CCS, CSP, HOL, OBJ, LOTOS, Petri nets, state transition diagrams)
4:	+ formal proof to establish conformance to software requirements specification.

IEC 61508: Software Detailed Design

0:	none
1:	structured methodology (CORE, JSD, MASCOT, Yourdon)
2:	+ semi-formal methods (function block diagrams, data-flow diagrams, pseudo code)
3:	+ formal methods (VDM, Z, CCS, CSP, HOL, OBJ, LOTOS, Petri nets, state transition diagrams)
4:	+ formal proof to establish conformance to software requirements specification.

What about actually doing it?

0: none

1: structured methodology (CORE, JSD, MASCOT, Yourdon)

2: + semi-formal methods (function block diagrams, data-flow diagrams, pseudo code)

3: + formal methods (VDM, Z, CCS, CSP, HOL, OBJ, LOTOS, Petri nets, state transition diagrams)

4: + formal proof to establish conformance to software requirements specification.

Formal Methods

- Rumours of the death of Formal Methods are much exaggerated:
  - we successfully use them
  - our customers successfully use them
  - they fit perfectly with SPARK and Correctness by Construction
- Only mathematically-based approaches offer the promise of bug prevention rather than bug detection

Some evidence

CDIS	VDM, CCS, C	10 year warranty, exceptionally trouble free
C130J	CORE, SPARK	Dramatic rise in quality and productivity
SHOLIS	Z, SPARK	00-55, “proof more cost-effective than testing”
MULTOS CA	Z, Ada, C++, SQL, SPARK	ITSEC E6. Formal spec adopted by customer
TCAS	RSML	Formal spec. is “official”. Informal spec. abandoned
HDLC	FormalCheck	Serious error found, model checking adopted

Agenda

- Introductions
- Some thoughts on standards
- Static analysis
- Unambiguous languages
- Formal methods
- Putting it all together

Correctness by Construction

- Formal specs, unambiguous languages and static analysis all help improve code quality and reliability
- If we use all of them and:
  - focus on bug prevention
  - use techniques that find bugs early
  - regard final testing as demonstration of correct behaviour rather than method of finding bugs

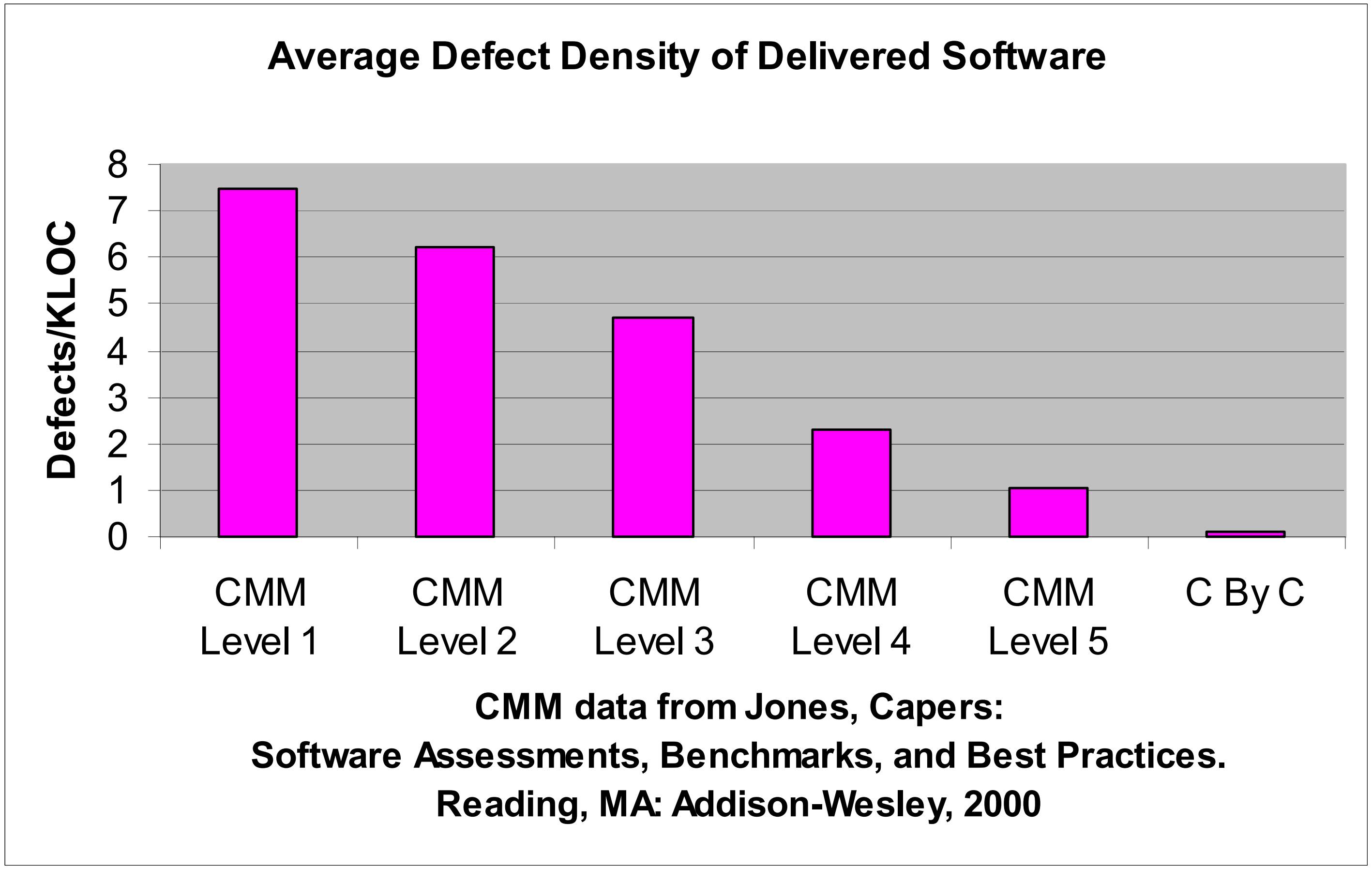
  then a wonderful synergy occurs:
  - we get higher quality at lower cost
  - we achieve Correctness by Construction

Low defects at high productivity rates

- Typical defect rate in industry is > 5 defects per KLOC
- Typical productivity rate in industry is < 10 LOC per day
- Sample Praxis rates (for deployed, certified code, including all lifecycle phases and management overhead):

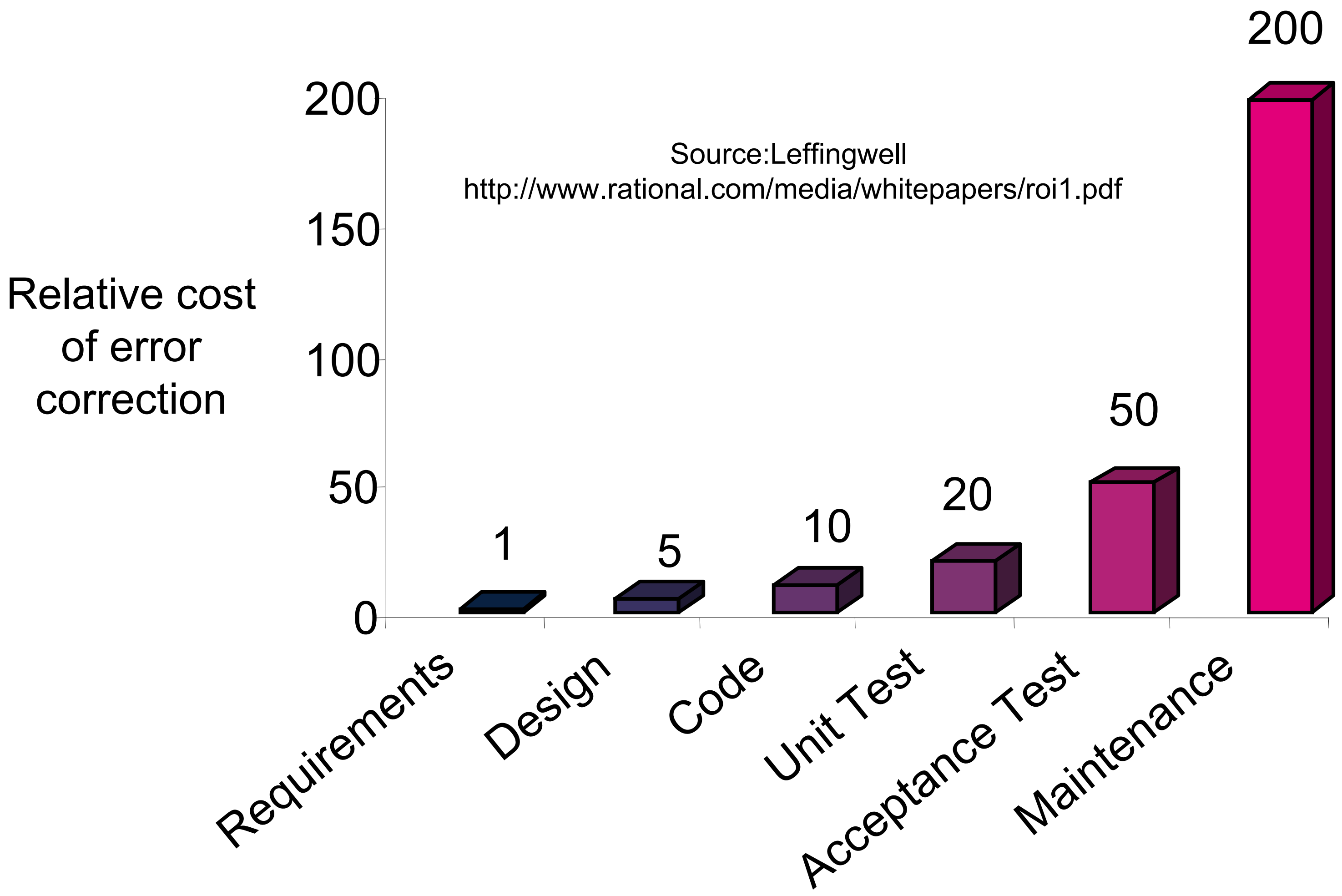
Year	Project	Integrity	Size	Defect/ ksloc	loc/day
1992	ATC display	SIL2	197,000	0.75	13
1997	Helicopter landing system	SIL 4	27,000	0.22	7
1999	Smart card security	ITSEC E6	100,000	0.04	29
2002	Aircraft test set	SIL 0	35,000	<0.1	28
2003	Secure biometrics	CC EAL 5+	10,000	0.00	38

Industry-beating Software Defect Rates



CMM data from Jones, Capers:
Software Assessments, Benchmarks, and Best Practices.
Reading, MA: Addison-Wesley, 2000

How? The Cost of Not Finding Errors

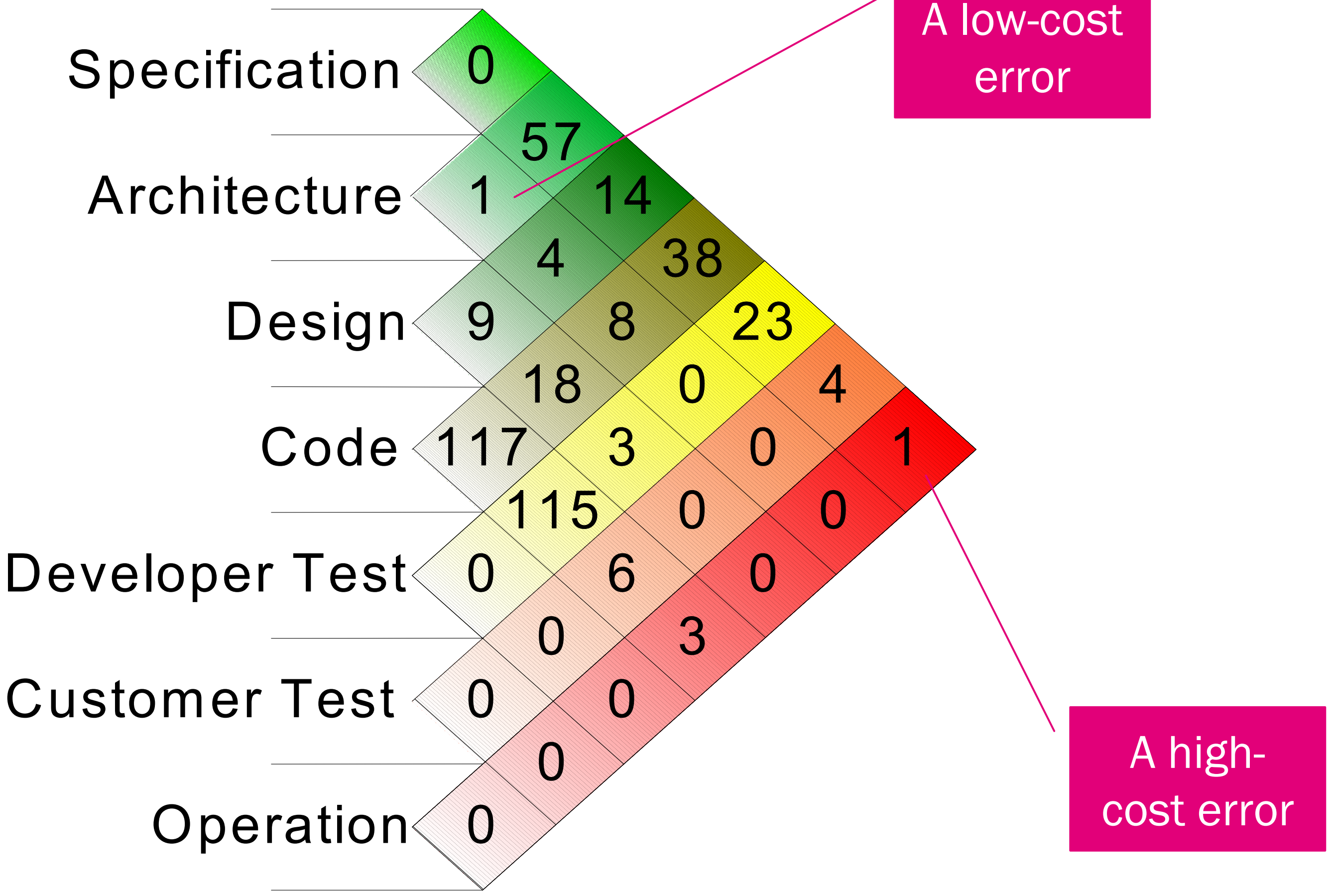


Time spent:

<20%	>80%

Metrics (from smart card project)

eliminating errors immediately after introduction



My Standard

The fitness for purpose of a software program shall be established by logical reasoning

DONE!

And it is wholly 61508 compliant as a free bonus!

and finally

"Real life problems are those that remain after you have systematically failed to apply all the known solutions"

Edsger Dijkstra, 1973

Any questions?

Praxis Critical Systems Limited

20 Manvers Street
Bath BA1 1PX
United Kingdom

Telephone:	+44 (0) 1225 466991
Facsimile:	+44 (0) 1225 469006
Website:	`www.praxis-cs.co.uk`
	`www.sparkada.com`
Email:	`peter.amey@praxis-his.com`

Resources

- Cook, David. *Evolution of Programming Languages and Why a Language Is Not Enough to Solve Our Problems*. Crosstalk Dec 99. pp 7-12 (http://www.stsc.hill.af.mil/crosstalk/frames.asp?uri=1999/12/cook.asp)
- Amey, Peter. *Correctness by Construction - Better Can Also be Cheaper.* Crosstalk March 2002 pp 24 -28. (http://www.praxis-his.com/pdfs/c_by_c_better_cheaper.pdf)
- ISO/IEC JTC1/SC22/WG9. Programming Languages - *Guide for the Use of the Ada Programming Language in High Integrity Systems.* (www.dkuug.dk/jtc1/sc22/wg9/n359.pdf)
- German, Andy, *Software Static Code Analysis Lessons Learned*. Crosstalk Nov 2003. pp 13-17. (http://www.stsc.hill.af.mil/crosstalk/2003/11/0311German.pdf)
- Hall, Anthony and Chapman, Roderick: “*"Correctness By Construction: Developing a Commercial Secure System“*, IEEE Software, Jan/Feb 2002, pp18-25 (http://www.praxis-his.com/pdfs/c_by_c_secure_system.pdf)
- King, Steve; Hammond, Jonathan; Chapman, Rod and Pryor, Andy: *"Is Proof More Cost Effective Than Testing?”*, IEEE Transactions on Software Engineering, Volume 26 Number 8 (http://www.praxis-his.com/pdfs/cost_effective_proof.pdf)

Resources (contd.)

- Butler, Ricky W., and George B. Finelli, eds. *"The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software."* IEEE Transactions on Software Engineering 19(1): 3-12. (http://shemesh.larc.nasa.gov/paper-nonq/nonq-paper.pdf)
- Littlewood & Strigini"*Validation of Ultrahigh Dependability for Software-based Systems*".. CACM Nov 1993 (http://www.csr.city.ac.uk/people/lorenzo.strigini/ls.papers/CACMnov93_limits/CACMnov93.pdf)
- Littlewood, B. *"The Problem of Assessing Software Reliability."* SCSS-2000. (http://www.csr.city.ac.uk/people/bev.littlewood/bl_public_papers/SCSS_2000/SCSS_2000.pdf)
- Amey, Peter. "*A Language for Systems not Just Software*". ACM SigAda 2001. (http://www.praxis-his.com/pdfs/systems_not_just_sw.pdf)
- Chapman, Rod., Amey, Peter. *"Industrial Strength Exception Freedom".* Proceedings of ACM SigAda 2002. (http://www.praxis-his.com/pdfs/Industrial_strength.pdf)

Resources (contd.)

- Amey, Peter,. and White, Neil. *“High Integrity Ada in a UML and C World”.* Lecture Notes in Computer Science 3063
  A. Llamosi, A. Strohmeier (Eds.): Reliable Software Technologies – Ada-Europe 2004 9th Ada-Europe International Conference, La Palma de Mallorca, June 2004, pp. 225-236. `(http://www.praxis-his.com/sparkada/pdfs/ada_uml_and_c.pdf)`

- See also www.sparkada.com