C v C++ in embedded control systems

Les Hatton

October 4, 2010

A little about this talk

- This is the second talk I have done which apart from its beamer interface used no Microsoft software whatsoever.
- It was constructed on Linux using the Kile interface to Latex and the Beamer package.

Setting the scene

- Does programming language matter in systems design ?
- Who to - the engineer or the end-user ?
- If it matters, what should we choose ?
- If it doesn't matter, what should we do ?
- I hope to convince you (with evidence) that ...
  - It doesn't matter
  - Language choice has never mattered
  - What does matter is how well you understand the language you use
  - ... and a few other things. :-)

Setting the scene

- Does programming language matter in systems design ?
- Who to - the engineer or the end-user ?
- If it matters, what should we choose ?
- If it doesn't matter, what should we do ?
- I hope to convince you (with evidence) that ...
  - It doesn't matter
  - Language choice has never mattered
  - What does matter is how well you understand the language you use
  - ... and a few other things. :-)

Setting the scene

- Does programming language matter in systems design ?
- Who to - the engineer or the end-user ?
- If it matters, what should we choose ?
- If it doesn't matter, what should we do ?
- I hope to convince you (with evidence) that ...
  - It doesn't matter
  - Language choice has never mattered
  - What does matter is how well you understand the language you use
  - ... and a few other things. :-)

Setting the scene

- Does programming language matter in systems design ?
- Who to - the engineer or the end-user ?
- If it matters, what should we choose ?
- If it doesn't matter, what should we do ?
- I hope to convince you (with evidence) that ...
  - It doesn't matter
  - Language choice has never mattered
  - What does matter is how well you understand the language you use
  - ... and a few other things. :-)

Setting the scene

- Does programming language matter in systems design ?
- Who to - the engineer or the end-user ?
- If it matters, what should we choose ?
- If it doesn't matter, what should we do ?
- I hope to convince you (with evidence) that ...
  - It doesn't matter
  - Language choice has never mattered
  - What does matter is how well you understand the language you use
  - ... and a few other things. :-)

Setting the scene

- Does programming language matter in systems design ?
- Who to - the engineer or the end-user ?
- If it matters, what should we choose ?
- If it doesn't matter, what should we do ?
- I hope to convince you (with evidence) that ...
  - It doesn't matter
  - Language choice has never mattered
  - What does matter is how well you understand the language you use
  - ... and a few other things. :-)

Setting the scene

- Does programming language matter in systems design ?
- Who to - the engineer or the end-user ?
- If it matters, what should we choose ?
- If it doesn't matter, what should we do ?
- I hope to convince you (with evidence) that ...
  - It doesn't matter
  - Language choice has never mattered
  - What does matter is how well you understand the language you use
  - ... and a few other things. :-)

Setting the scene

- Does programming language matter in systems design ?
- Who to - the engineer or the end-user ?
- If it matters, what should we choose ?
- If it doesn't matter, what should we do ?
- I hope to convince you (with evidence) that ...
  - It doesn't matter
  - Language choice has never mattered
  - What does matter is how well you understand the language you use
  - ... and a few other things. :-)

Setting the scene

- Does programming language matter in systems design ?
- Who to - the engineer or the end-user ?
- If it matters, what should we choose ?
- If it doesn't matter, what should we do ?
- I hope to convince you (with evidence) that ...
  - It doesn't matter
  - Language choice has never mattered
  - What does matter is how well you understand the language you use
  - ... and a few other things. :-)

Overview

- Evidence
- Some guidelines
- Conclusions

Evidence

I will consider evidence from ...

- Defect studies
- Abstraction and power-laws
- Language Standardisation

Defects

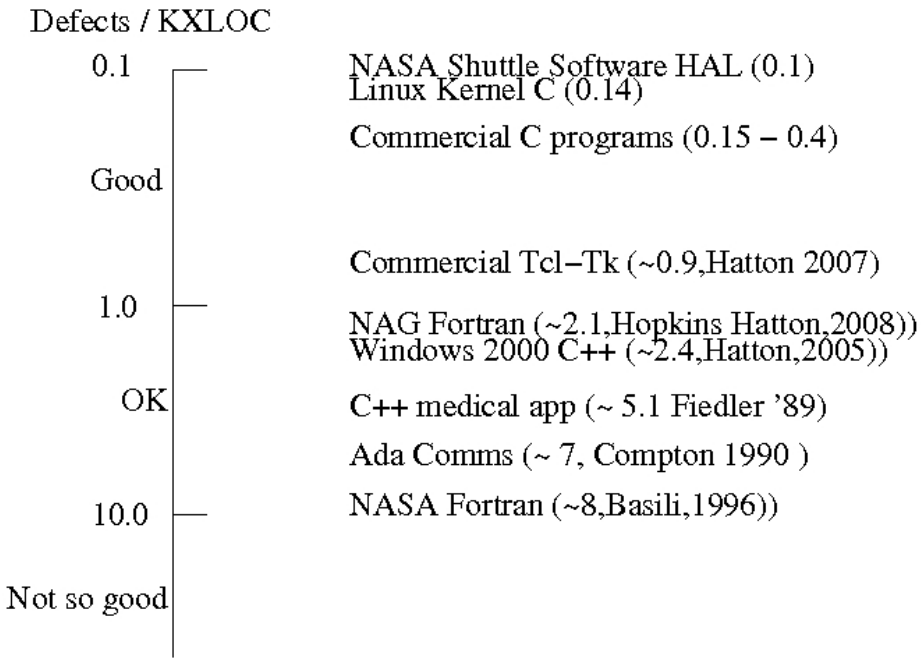

Figure: Some examples of quoted defect densities. *There is no systematically observable relationship between programming language and resulting defect density.*

Abstraction and power-laws 1

- Since we seem to make mistakes at a rate independent of programming language, how about abstraction ?
- Abstraction is believed to have the following properties
  - Allows large systems to be built like small systems - from many standardised components.
  - Adds (currently unquantifiable) benefits in design
  - Produces smaller systems so if defect density is about the same, abstracted systems would have less defects.

Abstraction and power-laws 1

- Since we seem to make mistakes at a rate independent of programming language, how about abstraction ?
- Abstraction is believed to have the following properties
  - Allows large systems to be built like small systems - from many standardised components.
  - Adds (currently unquantifiable) benefits in design
  - Produces smaller systems so if defect density is about the same, abstracted systems would have less defects.

Abstraction and power-laws 1

- Since we seem to make mistakes at a rate independent of programming language, how about abstraction ?
- Abstraction is believed to have the following properties
  - Allows large systems to be built like small systems - from many standardised components.
  - Adds (currently unquantifiable) benefits in design
  - Produces smaller systems so if defect density is about the same, abstracted systems would have less defects.

Abstraction and power-laws 1

- Since we seem to make mistakes at a rate independent of programming language, how about abstraction ?
- Abstraction is believed to have the following properties
  - Allows large systems to be built like small systems - from many standardised components.
  - Adds (currently unquantifiable) benefits in design
  - Produces smaller systems so if defect density is about the same, abstracted systems would have less defects.

Abstraction and power-laws 1

- Since we seem to make mistakes at a rate independent of programming language, how about abstraction ?
- Abstraction is believed to have the following properties
  - Allows large systems to be built like small systems - from many standardised components.
  - Adds (currently unquantifiable) benefits in design
  - Produces smaller systems so if defect density is about the same, abstracted systems would have less defects.

Abstraction and power-laws 2

Many software systems show power-law behaviour (Hatton, 2009 and lots of others), *independently of programming language.*

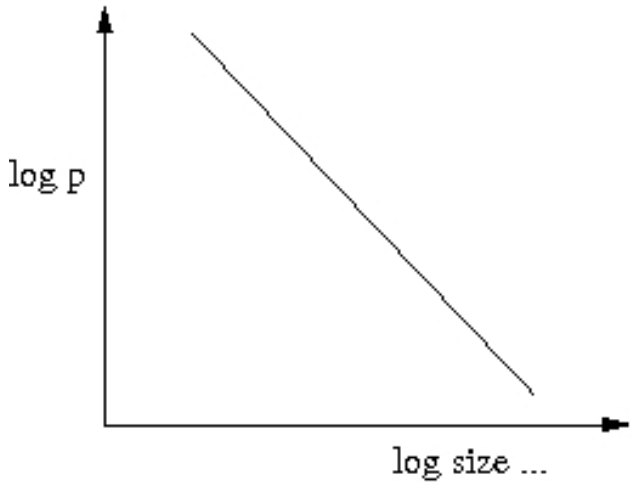

Figure: A power law $p_i \simeq s^{-\beta}$ plotted on a log-log chart.

Abstraction and power-laws 3

- If abstraction was working, big systems would be built like small systems
- This would mean ...
  - A majority of small objects re-used by programs of all sizes
  - Instead, objects obey power-laws, just like non OO language components, leading to the conclusion ...
  - *"This challenges the perceived wisdom of OO design ... objects within large programs have no characteristic scale"*, Potanin et al (2005). (Java, C++ or Smalltalk).

Abstraction and power-laws 3

- If abstraction was working, big systems would be built like small systems
- This would mean ...
  - A majority of small objects re-used by programs of all sizes
  - Instead, objects obey power-laws, just like non OO language components, leading to the conclusion ...
  - *"This challenges the perceived wisdom of OO design ... objects within large programs have no characteristic scale"*, Potanin et al (2005). (Java, C++ or Smalltalk).

Abstraction and power-laws 3

- If abstraction was working, big systems would be built like small systems
- This would mean ...
  - A majority of small objects re-used by programs of all sizes
  - Instead, objects obey power-laws, just like non OO language components, leading to the conclusion ...
  - *"This challenges the perceived wisdom of OO design ... objects within large programs have no characteristic scale"*, Potanin et al (2005). (Java, C++ or Smalltalk).

Abstraction and power-laws 3

- If abstraction was working, big systems would be built like small systems
- This would mean ...
  - A majority of small objects re-used by programs of all sizes
  - Instead, objects obey power-laws, just like non OO language components, leading to the conclusion ...
  - *"This challenges the perceived wisdom of OO design ... objects within large programs have no characteristic scale"*, Potanin et al (2005). (Java, C++ or Smalltalk).

Abstraction and power-laws 3

- If abstraction was working, big systems would be built like small systems
- This would mean ...
  - A majority of small objects re-used by programs of all sizes
  - Instead, objects obey power-laws, just like non OO language components, leading to the conclusion ...
  - *"This challenges the perceived wisdom of OO design ... objects within large programs have no characteristic scale"*, Potanin et al (2005). (Java, C++ or Smalltalk).

Abstraction and power-laws 4

- This turns out to be inevitable.
- In (Hatton (2008), (2009)), I proved that, for a given vocabulary of programming tokens (symbols and identifiers) a_i and a fixed size, the most likely probability distribution of component sizes obeyed

$$p_i \sim (a_i)^{-\beta} \tag{1}$$

for any programming language.

- This is a natural consequence of variational calculus and Shannon's information theorem.

Abstraction and power-laws 4

- This turns out to be inevitable.
- In (Hatton (2008), (2009)), I proved that, for a given vocabulary of programming tokens (symbols and identifiers) a_i and a fixed size, the most likely probability distribution of component sizes obeyed

$$p_i \sim (a_i)^{-\beta} \tag{1}$$

for any programming language.

- This is a natural consequence of variational calculus and Shannon's information theorem.

Abstraction and power-laws 4

- This turns out to be inevitable.
- In (Hatton (2008), (2009)), I proved that, for a given vocabulary of programming tokens (symbols and identifiers) a_i and a fixed size, the most likely probability distribution of component sizes obeyed

$$p_i \sim (a_i)^{-\beta} \tag{1}$$

for any programming language.

- This is a natural consequence of variational calculus and Shannon's information theorem.

Abstraction and power-laws 5

- However, abstraction which *reduces the number of lines of code for a given functionality* is different.
- At the open community level, it appears exceptionally valuable in the scripting languages, Perl, Python, Rexx and Tcl, when compared with C, C++ or Java. (Prechelt, 2000)
- In terms of higher-level functionality, both C and C++ have fallen behind, probably because they are not community based.

Abstraction and power-laws 5

- However, abstraction which *reduces the number of lines of code for a given functionality* is different.
- At the open community level, it appears exceptionally valuable in the scripting languages, Perl, Python, Rexx and Tcl, when compared with C, C++ or Java. (Prechelt, 2000)
- In terms of higher-level functionality, both C and C++ have fallen behind, probably because they are not community based.

Abstraction and power-laws 5

- However, abstraction which *reduces the number of lines of code for a given functionality* is different.
- At the open community level, it appears exceptionally valuable in the scripting languages, Perl, Python, Rexx and Tcl, when compared with C, C++ or Java. (Prechelt, 2000)
- In terms of higher-level functionality, both C and C++ have fallen behind, probably because they are not community based.

Abstraction and power-laws 6

- An example, a Perl e-mail reader starts off ...
- use Getopt::Std, Socket, MIME::Parser, MIME::Base64, MIME::WordDecoder, Unicode::Map8, HTML::Strip, IO::File, String::Random, CAM::PDF;
- With these and less than 200 lines of Perl, I can read any e-mail of any format and language including Word and PDF attachments.

Abstraction and power-laws 6

- An example, a Perl e-mail reader starts off ...
- use Getopt::Std, Socket, MIME::Parser, MIME::Base64, MIME::WordDecoder, Unicode::Map8, HTML::Strip, IO::File, String::Random, CAM::PDF;
- With these and less than 200 lines of Perl, I can read any e-mail of any format and language including Word and PDF attachments.

Abstraction and power-laws 6

- An example, a Perl e-mail reader starts off ...
- use Getopt::Std, Socket, MIME::Parser, MIME::Base64, MIME::WordDecoder, Unicode::Map8, HTML::Strip, IO::File, String::Random, CAM::PDF;
- With these and less than 200 lines of Perl, I can read any e-mail of any format and language including Word and PDF attachments.

Language standardisation

- We have real problems here.
  - C
    - C99 is a technical and commercial disaster
    - C1X is building on it. It is currently 670 pages, (3 times bigger than C90 and 50% bigger than C99).
  - C++
    - C++0X has *1317 pages* in Draft N3126 including over 250 examples of undefined behaviour.
- Both have failed their respective communities in my opinion.

Language standardisation

- We have real problems here.
  - C
    - C99 is a technical and commercial disaster
    - C1X is building on it. It is currently 670 pages, (3 times bigger than C90 and 50% bigger than C99).
  - C++
    - C++0X has *1317 pages* in Draft N3126 including over 250 examples of undefined behaviour.
- Both have failed their respective communities in my opinion.

Language standardisation

- We have real problems here.
  - C
    - C99 is a technical and commercial disaster
    - C1X is building on it. It is currently 670 pages, (3 times bigger than C90 and 50% bigger than C99).
  - C++
    - C++0X has *1317 pages* in Draft N3126 including over 250 examples of undefined behaviour.
- Both have failed their respective communities in my opinion.

Language standardisation

- We have real problems here.
  - C
    - C99 is a technical and commercial disaster
    - C1X is building on it. It is currently 670 pages, (3 times bigger than C90 and 50% bigger than C99).
  - C++
    - C++0X has *1317 pages* in Draft N3126 including over 250 examples of undefined behaviour.
- Both have failed their respective communities in my opinion.

Language standardisation

- We have real problems here.
  - C
    - C99 is a technical and commercial disaster
    - C1X is building on it. It is currently 670 pages, (3 times bigger than C90 and 50% bigger than C99).
  - C++
    - C++0X has *1317 pages* in Draft N3126 including over 250 examples of undefined behaviour.
- Both have failed their respective communities in my opinion.

Language standardisation

- We have real problems here.
  - C
    - C99 is a technical and commercial disaster
    - C1X is building on it. It is currently 670 pages, (3 times bigger than C90 and 50% bigger than C99).
  - C++
    - C++0X has *1317 pages* in Draft N3126 including over 250 examples of undefined behaviour.
- Both have failed their respective communities in my opinion.

Language standardisation

- We have real problems here.
  - C
    - C99 is a technical and commercial disaster
    - C1X is building on it. It is currently 670 pages, (3 times bigger than C90 and 50% bigger than C99).
  - C++
    - C++0X has *1317 pages* in Draft N3126 including over 250 examples of undefined behaviour.
- Both have failed their respective communities in my opinion.

Some guidelines

- Portability
- Changes to software process
- Know your chosen language well
- Be careful with your compiler choice
- Test the arithmetic
- Do not rely on bureaucracy to produce good systems

Portability 1

- Portability is a very important system attribute
- It is central to reducing costs and increasing impact, (van Genuchten and Hatton, IEEE Software 2009).
- Portability is less easy to achieve in C++ than C. The failure of C99 means that C systems are very often developed close to the old C90 standard which was simple and therefore still very portable.

Portability 1

- Portability is a very important system attribute
- It is central to reducing costs and increasing impact, (van Genuchten and Hatton, IEEE Software 2009).
- Portability is less easy to achieve in C++ than C. The failure of C99 means that C systems are very often developed close to the old C90 standard which was simple and therefore still very portable.

Portability 1

- Portability is a very important system attribute
- It is central to reducing costs and increasing impact, (van Genuchten and Hatton, IEEE Software 2009).
- Portability is less easy to achieve in C++ than C. The failure of C99 means that C systems are very often developed close to the old C90 standard which was simple and therefore still very portable.

Portability 2

Compare these two architectures

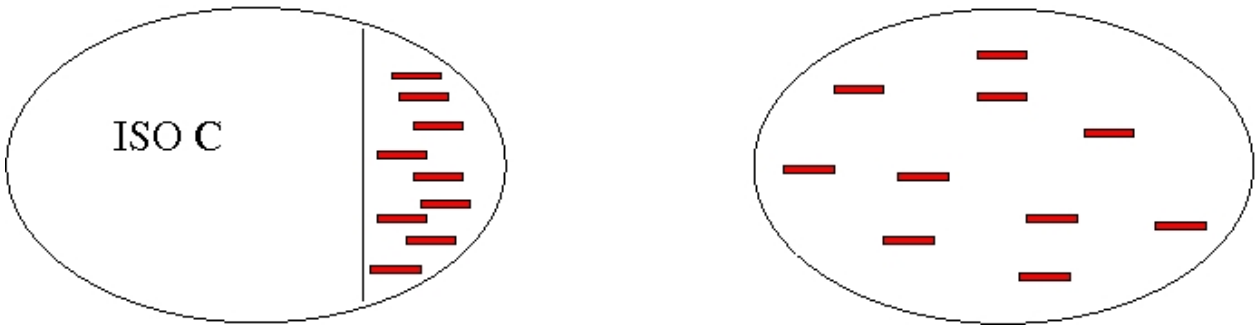

Figure: Two system architectures for dealing with non-standard language extensions. The left hand side produces a considerably cheaper system whatever language is being used.

Changes to software process

- Some things to watch out for in OO development
  - Inheritance is a defect attractor (defect densities around 3 times higher, Cartwright and Shepperd 2000).
  - OO delays some static decisions to dynamic ones, (Humphreys 1995, Hatton 1997) making defects found at system testing far more expensive to find, so early defect detection processes (for example code inspections) are even more important for OO than non OO systems.

Changes to software process

- Some things to watch out for in OO development
  - Inheritance is a defect attractor (defect densities around 3 times higher, Cartwright and Shepperd 2000).
  - OO delays some static decisions to dynamic ones, (Humphreys 1995, Hatton 1997) making defects found at system testing far more expensive to find, so early defect detection processes (for example code inspections) are even more important for OO than non OO systems.

Changes to software process

- Some things to watch out for in OO development
  - Inheritance is a defect attractor (defect densities around 3 times higher, Cartwright and Shepperd 2000).
  - OO delays some static decisions to dynamic ones, (Humphreys 1995, Hatton 1997) making defects found at system testing far more expensive to find, so early defect detection processes (for example code inspections) are even more important for OO than non OO systems.

Know your chosen language well

- All the evidence suggests that language choice has no statistically observable effect on externally facing system properties such as defect density.
- By far the most significant factor seems to be how well the engineer understands the language and technology they actually use.

Know your chosen language well

- All the evidence suggests that language choice has no statistically observable effect on externally facing system properties such as defect density.
- By far the most significant factor seems to be how well the engineer understands the language and technology they actually use.

Choose a compliant compiler if possible

- Compliance to ISO standards is at the heart of safer subsets such as MISRA. However Hatton (2007b) found that ISO C failures occurred at about the rate of about 1 every 20 lines or so in some compilers.

Test your arithmetic

- Some combinations of chip and embedded system compiler are still failing basic arithmetic tests.
- Make sure you check it, Hatton (2005).

Do not rely on bureacucracy to produce good systems

- We are developing a quality culture where box-ticking has replaced engineering insight and experience
- Example. The RAF Nimrod explosion, Afghanisation 2006. The result of the enquiry in October 2009 said ...
- *"The Nimrod Safety Case was a lamentable job from start to finish. It was riddled with errors, it missed key dangers, its production is a story of incompetence, complacency and cynicism."* Charles Haddon-Cave, QC.

Do not rely on bureacucracy to produce good systems

- We are developing a quality culture where box-ticking has replaced engineering insight and experience
- Example. The RAF Nimrod explosion, Afghanisation 2006. The result of the enquiry in October 2009 said ...
- *"The Nimrod Safety Case was a lamentable job from start to finish. It was riddled with errors, it missed key dangers, its production is a story of incompetence, complacency and cynicism."* Charles Haddon-Cave, QC.

Do not rely on bureacucracy to produce good systems

- We are developing a quality culture where box-ticking has replaced engineering insight and experience
- Example. The RAF Nimrod explosion, Afghanisation 2006. The result of the enquiry in October 2009 said ...
- *"The Nimrod Safety Case was a lamentable job from start to finish. It was riddled with errors, it missed key dangers, its production is a story of incompetence, complacency and cynicism."* Charles Haddon-Cave, QC.

Conclusions

- Language choice appears to have no significant effect on systems quality
- However, engineering skills do, so never stop trying to improve them.
- Lots more at http://www.leshatton.org/
- Thanks.

Conclusions

- Language choice appears to have no significant effect on systems quality
- However, engineering skills do, so never stop trying to improve them.
- Lots more at http://www.leshatton.org/
- Thanks.

Conclusions

- Language choice appears to have no significant effect on systems quality
- However, engineering skills do, so never stop trying to improve them.
- Lots more at http://www.leshatton.org/
- Thanks.

Conclusions

- Language choice appears to have no significant effect on systems quality
- However, engineering skills do, so never stop trying to improve them.
- Lots more at http://www.leshatton.org/
- Thanks.

Bibliography

- Cartwright and Shepperd (2000) “An Empirical Investigation of an Object-Oriented Software System”, IEEE TSE 26(8)
- Compton and Witrow (1990) “Prediction and control of Ada software defects”, JSS, 12(3).
- Basili, Briand and Melo (1996) “A Validation of Object-Oriented Design Metrics as Quality Indicators”, IEEE TSE, 22(10)
- Hatton (1998) “Does OO sync with the way we think ?”, IEEE Software, 15(3).
- Hatton (2005) “Embedded System Paranoia: a tool for testing embedded system arithmetic”, IST 47(8)

Bibliography 2

- Hatton (2007) “How accurately do software engineers predict maintenance tasks ?”, IEEE Computer, 40(2).
- Hatton (2007b) “Language subsetting in an industrial context: a comparison of MISRA C 1998 and MISRA C 2004”, IST 49(5)
- Hatton (2008) “Power-Law distributions of component sizes in general software systems”, IEEE TSE, Jul 2009.
- Hatton (2009) “Power-laws, persistence and the distribution of information in software systems”, http://www.leshatton.org/variations_2010.html.

Bibliography 3

- Hopkins and Hatton (2008) “Exploring defect correlations in a major Fortran numerical library”, http://www.leshatton.org/NAG01_01-08.html
- Humphreys (2005) “A discipline of software engineering”, Addison-Wesley
- Potanin, Noble, Frean and Biddle, (2005) “Scale-free Geometry in Object-Oriented Programs”, Comm. ACM., May
- Prechelt (2000) “An Empirical Comparison of Seven Programming Languages”, IEEE Computer 33(10)
- van Genuchten and Hatton (2009) “Software: Whats in it and whats it in ?”, IEEE Software 27(1).