The Feyerabend Project: Redefining Computing

Ron Goldman
Sun Microsystems
The Feyerabend Project

• a fractal, grassroots effort
• three workshops so far, four more planned—one at Santa Fe Institute next week on Biological Framings of Problems in Computing
• over 75 people participating including:

  Richard P. Gabriel  Jim Highsmith  Jeff Eastman
  Dave Thomas        Martine Devos        Brian Marick
  Guy L. Steele Jr.  Jerry Weinberg       Dave West
  Walter Fontana     Ward Cunningham      Ken Anderson
The Problem with Computing

Brittle, complex computer systems that create more work for people

- difficult to create
- difficult to configure
- difficult to maintain
- difficult to change
- difficult to use
Evolution vs. Revolution

What’s needed to solve the problem?

• Can we build on existing work?
• Do we need to change fundamental values?

We’re leaning towards the need for revolution.
How did we get here?

Current state was created by intelligent people doing the rational, “right” thing, but in a world that no longer exists.

Old world: cpu time is expensive
limited memory, disk storage, etc.
programmer time is cheap
standalone computers
applications are independent

New world: computers are cheap
lots of memory, disks are huge, etc.
programming is expensive
networked computers
applications need to interoperate
Example: Databases

Old world: how to get high performance out of disks—how to best cache records, access disk, etc.

New world: in-memory databases are simpler solutions as memory exceeds disk storage in cost-performance (memory now costs ~100 times more per GB than disk, but is ~200,000 times faster for random access)

E.g. Google uses thousands of low cost PCs networked together to create a superfast search engine. Each PC has 4GB of memory and they do lots with in-memory databases.
Example: Security vs. Performance

Old world: code to check array bounds & memory to hold them is too expensive; static compile-time type checking

New world: security concerns & costs are paramount—normal performance exceeds user requirements

E.g. programming in C or C++ leads to buffer overruns, most common security hole.
Example: Robustness vs. Performance

Old world: programmer required to handle freeing objects when no longer needed—garbage collection too much overhead

New world: let computer do it via automatic garbage collection—debugging time and robustness bigger factors

Observation: C/C++ plus GC = Java?
Example: Flexibility vs. Efficiency

Old world: Monolithic application can be updated anytime (i.e. global recompilation possible)

Tight-coupling via API’s required for efficiency

New world: Net “applications” reside on many different machines and can never be updated simultaneously

Older devices/applications need to interact with newer ones—cannot break because the “interface” has changed

Loose-coupling via protocols & languages required for flexibility
Change in Values & Underlying Assumptions

Old world:  efficiency
            resource limitations
            performance
            monolithic
            standalone
            single author
            mathematics

New world: robustness
            flexibility
            adaptation
            distributed
            interconnected
            multiple authors
            biology
Partial Taxonomy of Life Forms

- Bacteria
- Archaea
- Protista
- Eukarya

- Fungi
- Plants
- Animals

- Flowering plants
- Conifers
- Ferns
- Algae
- Sponges
- Mollusks
- Insects
- Spiders
- Fish
- Amphibians
- Reptiles
- Birds
- Mammals
- Flatworms
Taxonomy of Computer Architectures
Limited Diversity of Computer Systems

In computer design, operating systems & programming languages our explorations have been quite limited.

Market forces push us toward winner-take-all solutions.

Single solution more prone to viruses, security breaches, etc.
Diversity vs. Commonality

At the lowest levels using a common solution can be ok:

- all life on Earth is DNA-based
- uses same encoding for amino acids
- shares basic structures (e.g. mitochondria)
- built using common chemical processes (e.g. photosynthesis via chlorophyll)

Innovation springs from diversity, but must be built on a solid foundation.
Programming Languages

Major software languages are basically all the same:

Fortran, C, C++, Java, Lisp, Smalltalk

Different syntax, but similar features.
While it is perhaps natural and inevitable that languages like Fortran and its successors should have developed out of the concept of the von Neumann computer as they did, the fact that such languages have dominated our thinking for twenty years is unfortunate. It is unfortunate because their long-standing familiarity will make it hard for us to understand and adopt new programming styles which one day will offer far greater intellectual and computational power.

—John Backus, 1981
Millions for compilers but hardly a penny for understanding human programming language use. Now, programming languages are obviously symmetrical, the computer on one side, the programmer on the other. In an appropriate science of computer languages, one would expect that half the effort would be on the computer side, understanding how to translate the languages into executable form, and half on the human side, understanding how to design languages that are easy or productive to use.... The human and computer parts of programming languages have developed in radical asymmetry.

—Alan Newell & Stu Card, 1985
Software Development

Moore’s Law: computer performance doubles every 18 months.

Fry’s Law: the speed of developing reliable software functionality doubles every 18 years (if that)!

programmer performance = human capabilities + programming tools
Software as Literature

- Programmers are taught theory and to write toy programs
- Great software is not available to be studied
- No critical tradition for software
Who Can Write Software?

Need to enable more people to “program”

Change what we mean by programming

Example: HTML opened up the web to non-programmers resulting in a great diversity of content.
Different Ways to Develop Software

- Open source model: lots of resources / room for system to evolve
- Piecemeal growth vs. master planning
- Need to include users in the design
Where do we go from here?

- Need to make some fundamental changes.
- Maybe need a new paradigm?
- Look at history of science.
...one of the most striking features of recent discussions in the history and philosophy of science is the realization that events and developments... occurred only because some thinkers either decided not to be bound by certain ‘obvious’ methodological rules, or because they unwittingly broke them.

This liberal practice, I repeat, is not just a fact of the history of science. It is both reasonable and absolutely necessary for the growth of knowledge. More specifically, one can show the following: given any rule, however ‘fundamental’ or ‘necessary’ for science, there are always circumstances when it is advisable not only to ignore the rule, but to adopt its opposite.

—Paul Feyerabend, Against Method
The Feyerabend Project

- Understand the limitations of our current computing paradigm
- Understand the limitations of our current development methodologies
- Bring users—that is, people—into the design process
- Make programming easier by making computers do more of the work
- Use deconstruction to uncover marginalized issues and concepts
- Looking to other metaphors
Feyerabend Project

- Homeostasis, immune systems, self-repair, and other biological framings
- Physical-world-like constraints—laws, contiguity
- Blackboards, Linda, and rule-systems—use compute-power
- Additive systems—functionality by accretion not by modification
- Non-linear system-definition entry—instead of linear text
- Non-mathematical programming languages
- Sharing customizations
- Language co-mingling and sustained interaction instead of one-shot procedure invocation in the form of questions/answers or commands
- Piecemeal growth, version skews, random failures
- Artists’ understanding, ambiguous truth
Join Us

http://www.dreamsongs.com/Feyerabend/Feyerabend.html