

Darwinian Software Development

Stefano Mazzocchi

<stefano@apache.org>

<stefanom@mit.edu>

Warning

This talk is NOT about genetic algorithms!

so, don't complain later...

Nutrition Facts

- 0% software and algorithms.
- 0% licenses and/or philosophies.
- 40% distilled front-line experience.
- 25% community dynamics.
- 20% my boss/colleague/friend should read this.
- 10% oh, now I get it.
- 5% take it with a grain of salt.

Section I
A little bit of history

What is a model?

A schematic description of a system, theory, or phenomenon that accounts for its known or inferred properties and may be used for further study of its characteristics.

The American Heritage Dictionary
of the English Language, Fourth Edition

Why a model?

- Used as a metaphor, helps people understand.
- If applicable in more fields, can be used as a bridge, allowing discoveries in one field to be moved to the other.
- Simplifies reality and highlights particular aspects.

Previous Models

- The Cathedral and Bazaar Model (Eric Raymond)
- The Mob Software Model (Richard P. Gabriel & Ron Goldman)
- The Cooking Pot (Rishab Aiyer Ghosh)

Why a new model?

- There is no model that helps people with the following questions:
 - how does an open development effort change over time and why?
 - what can I do better?
 - can I apply this F/OSS secret sauce to other environments?

Section II

A Few Definitions

Evolutionary Systems

Synonym: Darwinian Systems

Systems of whatever material base
that undergo evolutionary processes.

Principia Cybernetica

Evolutionary Process

A trial-error process of variation
and natural selection of systems
at any level of complexity.

Principia Cybernetica

Trial-Error Process

The creation of random combinations of matter, with the subsequent struggle for existence, as a result of which some combinations survive and proliferate, while other perish.

Principia Cybernetica

BVSR

- “Blind Variation and Selective Retention”
- Introduced by Donald T. Campbell as a formal mathematical model of the darwinian evolutionary process.
- Based on three phases:
 - blind variation
 - asymmetric transition
 - selective retention

Blind Variation

At the most fundamental level,
variation processes "do not know"
which of the variants they produce
will turn out to be selected.

Principia Cybernetica

Asymmetric Transition

A transition from an unstable configuration to a stable one is possible, but the converse is not.

Principia Cybernetica

Selective Retention

Stable configurations are retained,
unstable ones are eliminated.

Principia Cybernetica

Section III

Applying to Software

Parallels Software/Biology

- Software source code \Leftrightarrow DNA
- Software binary code \Leftrightarrow Organisms
- Software projects \Leftrightarrow Species

Further Parallels I

- Software Copying \Leftrightarrow Birth
- Software Patching \Leftrightarrow Mutation
- Software Forking/Back Incompatibility \Leftrightarrow Change in Specie
- Software Dependencies \Leftrightarrow Symbiosis/Synergy
- Software Systems \Leftrightarrow Ecosystems

Further Parallels II

- Quality \Leftrightarrow Ability to replicate
- Impossibility to copy with or without modifications \Leftrightarrow Sterility
- Free/Open Licensing \Leftrightarrow All individuals can replicate
- Closed licensing \Leftrightarrow Only few individuals can replicate

Main Difference

- Random variation does not apply (no monkey-typing code development)
- But blind variation does apply!
- Note how the main reason why many people don't buy the theory of evolution (mostly creationists) is the idea of order emerging from pure chaos and lack of intentionality. This debate does not apply in software (yet?).

Section IV

Embracing Previous Models

Release Early/Release Often

- Increases the nativity rate.
- Faster adaptation cycle.

Scratching Itches

- Only features that are really critical for the environment get implemented.
- Keeps the efficiency at maximum.

KISS

- Allows more people to be able to scratch their itches.
- Increases the ability for environmental mutations to get fed back.

Commit then Review

- Mutations get out there faster.
- Quality control is done by the environment, not by you.

Test-Driven Programming

- Too many mutations can make an organism unable to survive enough for the environment to test its fitness.
- Make sure your organism can still survive out there, or it's a waste of resources.

Community is more important than Code

- software does not mutate itself, so without a community software is sterile.

Good Ideas and Bad Code build communities faster

- good ideas maximize usage and fitness
- bad code increases feedback

Can't start a community without working code

- there is no evolution cycle without organisms being released in the environment

Code/idea ownership is bad

- it slows evolution and reduces the social noise that empowers it

Section V
Bridging Lessons

Reproduction Models

- Most biological species follow a model where all individuals are fertile.
- But there are examples of species (for example, bees) where only selected individuals can replicate.
- Lesson to learn: both development models have reasons to exist (and coexist).

Fertility as Feedback

- If a sterile organism lives or dies, the fitness information about its mutations is lost.
- Fertility of all individuals increase feedback.
- Lesson to learn: closed development needs special channels for fitness information feedback, open development doesn't.
- Note: this also bridges the concept of “user-driven innovation”.

Incubation Periods

- Smaller organisms tend to have smaller incubation periods and lower reproduction costs.
- They also tend to be much more adaptable to radical environmental changes.
- Lessons to learn: higher modularity improves adaptability, because evolution can happen in parallel now.

Adaptability

- Biological species with a high rate of reproduction tend to adapt faster to environmental changes.
- On the other hand, fossils indicate how some animals achieved perfect fit ages ago and didn't require changes overtime.
- Lesson to learn: the environment drives software development, not the other way around.

Mutation Rates I

- Mutation rates should balance the environmental changes.
- Both too low and too high mutation cause extinction.
- Lesson to learn: release often but not too often, on the other hand, don't aim at perfection: its the environment that will judge it, not you.

Mutation Rates II

- Changes should be small and incremental to allow the environment to judge them independently.
- Lesson to learn: avoid revolutionary redesign as a very last resort.

Adaptability vs. Quality

- The “value” of an organism is never absolute but it’s always relative to the environment it evolved to fit in.
- Lessons to learn: there is no such thing as absolute software quality

Extinction

- The greatest majority of biological species is now extinct.
- Lesson to learn: prepare to let go

That's all folks!

Kudos to:

James Duncan Davidson
The Apache Software Foundation
Ben Hyde
Karim Lahkani

Available at

<http://www.betaversion.org/~stefano/papers/oscon2004.pdf>