Conscientious Software

Richard P. Gabriel
Sun Microsystems Laboratories
Menlo Park, California, USA
rpg@dreamsongs.com

Ron Goldman
Sun Microsystems Laboratories
Menlo Park, California, USA
ron.goldman@sun.com

Abstract

Software needs to grow up and become responsible for itself and its own future by participating in its own installation and customization, maintaining its own health, and adapting itself to new circumstances, new users, and new uses. To create such software will require us to change some of our underlying assumptions about how we write programs. A promising approach seems to be to separate software that does the work (allopoietic) from software that keeps the system alive (autopoietic).

Categories and Subject Descriptors D.2.5. [Software Engineering] Testing and Debugging—*Error handling and recovery*; D.2.11. [Software Engineering] Software Architectures; D.3.0. [Programming Languages] General; H.1.2. [Models and Principles] User/Machine Systems—*Human factors*

General Terms Design, Human Factors, Languages, Reliability

Keywords Robustness, software, self-sustaining systems, emergence, autopoiesis, stigmergy, continuous (re)design, self-testing, feedback, software complexity, repair

1. Introduction

Software systems today are produced according to a manufacturing model: A finished product is constructed at the factory and shipped to its final destination where it is expected to act like any other machine—reliable but oblivious to its surroundings and its own welfare. Responsibility for testing and devising how to install the software rests with the development team. Once deployed, software is on its own, along with the people who must use it. The result of this way of doing development has been brittle, buggy software where the only recourse for end-users is to hope the next release will fix the problems (and not add too many new ones).

We believe that software needs to grow up and become responsible for itself and its own future. Moreover, the people using the software every day must be able to shape and customize it without reliance on the software's original developers. In this paper we will argue that future innovations in software will need to produce systems that actively monitor their own activity and their environment, that continually perform self-testing, that catch errors and automatically recover from them, that automatically configure themselves during installation, that participate in their own development and customization, and that protect themselves from damage when patches and updates are installed. Such software systems will be self-contained, including within themselves their entire source code, code for testing, and anything else needed for their evolution.

Furthermore, we need to make the state of each component more visible and make our components interact more richly & more fully. We need to use softer, more dynamic architectures that support adding or replacing modules after deployment (e.g. plug-ins, dynamic loading of classes) and architectures where objects can be repaired *in situ*, methods changed / added, internal state restructured, and object hierarchies rewired). We also need new types of languages to describe the architecture of our systems. We will not "program" in these new languages—they probably won't have arithmetic operators, data structures, or low-level control statements—but instead express things like when to create and kill components, where components are run, when to run tests, and which components interact with each other.

As our software becomes ever more interdependent, with applications relying on (remote) services developed by other organizations, it also must paradoxically become more independent, able to maintain its integrity in a changing environment. Interfaces between components need to become less brittle and more accommodating—not requiring an exact match like a key in a lock, but based more on pattern recognition such as phenotropics [20]. We also need to better isolate components from each other—in biology this is referred to as *spatial compartmentalization* [19], for computing it may translate into components not sharing common memory or other resources.

Recent applications are beginning to exhibit these properties and this trend will increase, reflecting the need for software systems to change continuously over their lifetimes. Many of the ideas we present in this paper are based on the changes we see occurring in current software. We use the term "conscientious software" to describe code that takes responsibility for itself and its future. The following is our vision for the future.

2. Software Complexity

Over the last forty years we have attempted to create ever more ambitious systems, but their complexity often exceeds our abilities as witnessed by how brittle most software is and how many software projects fail. While our current methods have many good characteristics they do not successfully address the complexity

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License.

of modern software systems. We need to adopt new principles and approaches if we are to progress.

Software complexity arises in large part from the inherent complexity of the problems we are trying to solve. We lack good models to describe the organization of complex systems. We know how to decompose large problems into smaller ones, but we are not so good at describing the interactions among subcomponents. For example, we do not have a good vocabulary to discuss multiple feedback loops and the emergent behavior they can generate.

We also have trouble when different aspects of a problem require fundamentally different decompositions. For example, when designing a spacecraft control system there is a "natural" decomposition into subsystems and devices. However when dealing with the limited power budget or the attitude control system, there are "a tangle of special-case subsystem-to-subsystem couplings behind a façade of modular decomposition" [7]—consider that spinning up a disk drive consumes power, generates heat, and acts as a gyroscope. This *interaction of concerns* goes beyond the idea of *separation of concerns* that aspect-oriented programming (AOP) tries to address.

The sheer complexity of the problem and its solution makes it practically impossible to capture such interactions in a static specification. The experience at Tandem Computers of trying to create reliable systems found that many failures were due to faults in the specifications. Specifications are always incomplete and often just plain wrong—moreover requirements are always changing. Methodologies with an incremental approach of *continuous (re)design* are a better match to changing specifications. Related to this is the realization that errors are unavoidable, due to bugs in the implementation and also from unexpected inputs from the environment, so we must design our systems to continually run tests and repair errors.

Going from problem requirements to system design creates an explosion of derived requirements (the requirements for a particular design solution); the list of implicit requirements can be 50 times greater than the original problem-focused requirements. [11] Lots of details need to be filled in when translating "what to do" (describing the problem to be solved) into "how to do" (describing how the problem is to be solved). This includes translating the problem into computer terms, specifying required components, and even mapping the work to the organization of the people developing the solution. Again we lack good models and higher-level abstractions to talk about complex, interacting processes.

Another major source of complexity is unplanned interactions between components—cross-talk where bugs or changes in one piece of code affect quite distant, seemingly unrelated pieces of code. These implementation combinatorics make it impossible to thoroughly test software: Even 100% test coverage cannot account for code interactions, e.g. an error manifests only when a particular combination of logic paths is executed. And beyond the lowest level of unplanned direct interactions there are the unplanned indirect ones. Examples would be feedback loops created which cause new and complex behavior to emerge. Another would be system-wide interactions which could cause oscillations (again this involves feedback)—sometimes via unplanned and unthought-of shared paths like indirectly shared resources. Lastly would be emergent behavior resulting from new interaction patterns. Hardware is generally more robust because it is more modular (and less state dependent). Where it relies on history / state information hardware too is subject to bugs (e.g. the Pentium math bug [28] in which the results of the division algorithm depend on intermediate partial results and the path taken through a faulty lookup table).

As if our problems with current systems were not enough, even bigger systems are on the drawing board. Plans for ultra-large-scale (ULS) systems consisting of billions of lines of code spread over thousands of widely distributed multi-cpu computers are now being proposed. These systems must run continuously, even as individual components / computers come and go. In order to create such systems we need new approaches to how we design software. [35]

3. Continuous (Re)Design

The practice of continuous (re)design will move from the factory to the field and large lump development will fade away. Development strategies have changed over time as it's become clear that it's not possible to get static requirements right the first time around. From heavy design-before-coding methodologies to agile, people have tried to capture via processes the elusive nature of getting things right. The trend has been toward entangled interactions with end-users and customers in order to find out what the software needs to do and how it is most congenially used. However, the watchword is still getting things right.

But because requirements are static doesn't mean that software viewed over time stands still. Programs evolve and change, but only through a development process back at the factory. Bug reports, comments, suggestions and requirements determined by a marketing group, bright ideas, and new technology combined with a development group produce the next version of the software. Static requirements change over time, and in this narrow sense, it's possible to view software as a living entity—adaptable, flexible, malleable, resistant to failure. But not self-sustaining.

Design is less a result of *pre hoc* planning than of slowly dawning insights—insights derived from seeing how the thing turns out and is used. Continuous (re)design will move out into the field because the field is where the observations are immediate and changes can be tested rapidly *in situ*.

Already some programs offer personalization and customization allowing users to participate in the design of the software. Some basic looks, feels, and behaviors can be changed by tailoring menus, adding keyboard shortcuts, and defining macros and scripts that add behavior and provide workarounds. Many of the options are available through a mechanism called "preferences," which is a set of attributes whose values can be set, such as colors, field placement, and the like. Preferences are like wall coverings and color, window treatments, artwork, and, to a degree, furnishings. (Furnishings are perhaps more like plug-ins.)

4. Soft / Dynamic Architectures

With the advent of softer / more dynamic architectures, change to programs will be welcome, even encouraged. Today, software development is a process of building static artifacts—static meaning the program will not change once delivered, remaining inflexible or subject only to foreseen adaptations. A good metaphor is a highway. Constructing a highway takes a lot of effort including expensive and disruptive land purchases. Once the dimensions and course of a highway are set, they are stable and static until a major change is made. Word processors are near the extreme of

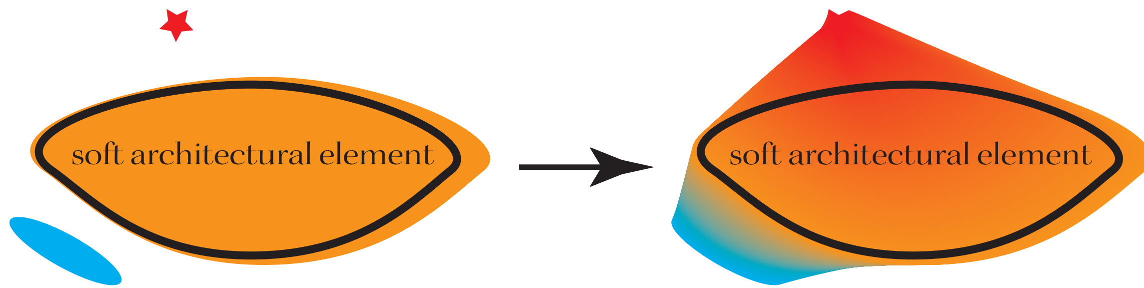

Figure 1: Soft Architecture

such static programs: useful for writing and revising documents with a wide variety of styles both visual and artistic, but its own future on a writer's computer is rigid—nothing changes.

Not all software is totally static, and such programs begin to point the way to change. Plug-in architectures and web-service-based applications can be changed once installed by adding modules that provide services based on defined protocols and interfaces. Adobe Photoshop™ is the classic example; and browsers, integrated development environments, and operating systems are built this way. Here the good metaphor is a house, which is also (usually) carefully planned (by an architect), but which can be and frequently is customized by its occupants in both small and large ways.

Stewart Brand looks at buildings as having six layers: the physical site, the load-bearing structure, the exterior surfaces, the guts of services like wiring and plumbing, the interior layout, and the stuff in the building like desks and phones. The timescale and effort needed to make a change varies across these layers; site is eternal, while stuff can easily be changed daily. [3] Our computer systems also consist of layers: programming languages are at the bottom,[1] then hardware, operating system, libraries, and applications. Applications generally consist of one or more levels of compiled code (e.g. a framework and application logic) with a few preference settings so that the user can modify the look and feel. On top of that is the layer of the user data, corresponding to Brand's stuff in the building—though in many applications the "desks" are bolted to the floor.

When thinking of software architecture, you probably are imagining an architecture diagram made of bubbles with lines between them—perhaps a framework (a big bubble maybe) with components (the little ones) where the lines mean communication through a small pipe, like an interface. This is what we have today. But what would it be like to imagine a diagram like this where two components (red star and blue ellipse) have melded with the framework (orange blob)? [Figure 1]

[1] Brand's layers correspond to permanence. For software, changing to a programming language with semantics not like Fortran, C, or Java would (rationally) require a new hardware platform. In fact, hardware (even instruction sets) has changed more than the basic semantic structures of our programming languages, and so the languages form the most basic layer for us. Sometimes the stack is drawn as an hourglass with hardware at the bottom, programming languages at the narrow waist, and applications at the top. This emphasizes how applications are decoupled from the hardware through the use of a standardized programming language.

Some applications, such as spreadsheets, enable their users to collaborate by adding new macros and sharing templates. Not everyone can write macros, but one or two tinkerers in a company can make extensions to the spreadsheet primitives that others can then easily use in their formulas. [26] An architecture that supports scripting or a macro language adds another layer that enables users to modify an application's behavior without needing to send it back to the factory for the professional programmers to rework. If such an architecture also supports adding plug-in modules then that opens the door for the do-it-yourself user to really customize the application to his or her local needs.

Beyond such softnesses are software architectures that make changes to themselves over time: dynamic adjustments to both state and behavior as the stuff (components / architecture) adapts to the environment in which it finds itself. We can imagine objects augmenting themselves, changing their algorithms, evolving through competition,[2] and rewiring their relationships as they evolve.

5. In(ter)dependent Software

Software will be constructed from components that want to form a community. Systems will be components—either within the same address space or not—that interact as needed, even supposedly / originally monolithic systems. This will enable us to treat every system as if it were a distributed system, which might mean that systems can be more robust to single-point failures.

Components will be constructed to minimize or ameliorate version differences and calling convention mismatches. Today, strange as it seems, components are written—such as dynamic link libraries (DLLs)—with inflexible interfaces so that slight mismatches halt reasonable operation of a system. The result is almost as if the designers of the components (or more properly, the overall OS-level system architecture) were more interested in their code and designs punishing simple mistakes rather than trying to make that code work around those mistakes to get something running.

An approach like Lanier's phenotropics could help. Phenotropic computing uses "surfaces" for interfaces rather than direct argument-based / protocol-based interfaces. Each component has a surface on which is displayed information about what it is doing and that it would like to communicate to another component (or other components). Two components interacting observe

[2] Digital software evolution

each other's surfaces and react to what they "see" or "sample." Approximation replaces perfection, and the result can be more accommodating but perhaps less optimized behavior. This is in contrast to our current use of rigid, minimalistic APIs where one agent essentially reaches inside another and commands it to do some function (i.e. a remote procedure call). Instead the first agent would present a request that the second could interpret and deal with as best it can. Other agents might also choose to participate by transforming / translating the original request, forwarding the request to an appropriate agent, or working on it themselves. The agent making the request need not even know which agent(s) will eventually handle it.

6. Self-Installation

Software in the future will take an active role in its own installation. At present, many programs are considered properly installed once the correct bits are in the right locations. And even this problem is not as well addressed as it could be: There are numerous interdependencies that go into proper installation. In recent years—with software updates—the installation scripts have become more sophisticated, but when more permanent layers (in the Brand sense) are installed, the process can require extensive human in(ter)vention. In fact, the outermost (applicationmost) layers are the places where installation has become easier for the installer.

Ignored as part of installation are the sometimes dozens of adjustments people make to the entire set of installed programs to make the computing / digital nest comfortable (again).[3] First are the obvious, such as: Although there are a handful of commands for moving through text, the gestures a person likes for various of them—though knowable—are never addressed. If it's even possible to make a program that manipulates text use Emacs[4] command gestures, why should a person have to figure out how to program / customize / personalize the software? Gestures are just part of it. The look and feel should be adaptable too. As stylesheets for application look and feel become more prevalent, these can form the basis for adapting to a preferred use scheme.

Each program should endeavor to discover such customizations. Further, it should be possible for one person to transfer his or her customizations to someone else who could then apply some or all of them either temporarily or permanently.

In the same vein, other aspects of the preferred use of software can be deduced and accommodated. For example, when installing a web browser, the browser can notice that one of its potential users is also an ardent user of Adobe™ products and to configure the browser to use Acrobat™'s PDF browser plug-in instead of some other one. The overall user color scheme can be intelligently guessed at and compatible colors can be chosen. A program that manipulates text can try to determine how spellchecking is done around here.[5] For example, I (one of the authors), like many in my generation, like to use Emacs for manipulating text. Many programs manipulate text. I am not particularly interested in the next person's bright idea of how to do it (unless it is a fabulous idea), so why can't I use not only the style I like (Emacs commands as mentioned before), but the very same Emacs in every situation, which I may have customized beyond recognition?

Another approach, of course, is a systemwide preference mechanism where specific preferences would be looked up in a hierarchical structure. This is the standardization approach. Although this would work fine were it to happen, the realistic likelihood of that is low. To get the majority of operating system / application writers to use such a system would require a mandated standard, which is unlikely for user experience. Moreover, for a long time there will be a significant population of legacy code that will not use the standard. And then there are the renegades.

Moreover, there is a real question about whether and when standardization is the right solution to a problem. Standardization requires universal adoption, and, when achieved, tends to promote inflexibility. And once standardization is taken as the key, it is typically taken too far, so that we end up, for example, with a monoculture.

Better, we think, is *ad hoc* and independent flexibility, recognizing that multiple standards / dialects will always exist, each of which will change over time.

7. Continual Installation

In our vision, the installation process never ends: As new software and hardware are installed, already installed software should continue the installation process, learning how the interests and habits of its users are changing over time. What it learns—not the personal things like where its users surf and the kinds of books and videos he or she prefers—can be sent back to the sleepy developers back at the plant who can move their agile fingers into gear to keep progress happening. Later or someday, this knowledge can be used to self-adapt the software.

Related to installation is the massive upgrade: buying that new heavy-lifting desktop machine or the spry and miniscule portable laptop / palmtop / fingertop / pintop. When a new computer system is acquired, all that should need be done is to point the old system at the new and have them "sync." "Sync" in the sense of compatibly installing everything and readjusting to the new surroundings. Most software manufacturers treat the upgrade as an opportunity (provided for free) for the users to clean house—to reinstall everything that was previously installed.[6] However, the massive upgrade is not a fresh start: It's an increment over the old environment because people expect the continuity of their lives to dominate the stutter of technological progress. And a new system should consider itself (anthropomorphically) to be installing itself in an old, established environment. Both the old and new must adapt.

Of course, none of this should be irrevocable. Any customizations and installations should be trivial to back out of or alter. It should always be the case that any installation can be completely and accurately undone.

[3] And this is the problem of (re)installing inner, permanent layers.

[4] Emacs is a text editor widely used by programmers. It is highly customizable but has a distinct set of default key bindings.

[5] At one extreme this can be taken as an example of the idea of common modules, typically selected by the user for some specific advantage.

[6] Apple's OS X has a Migration Assistant which performs the easier half of this when a new version of the operating system is installed—it copies files from the old system, ignoring files supplied by the new version. It doesn't do any "readjustments."

8. Beyond Installation

Someday the manufacturing model may become passé. Consider the notion of no installation or certainly no re-installation at all. In such a world, perhaps the only starting point would be a single object which would, like a seed, begin to use resources available to it to compose the "application," if it even makes sense to use that word. The seed would assemble components, download clones, etc. to custom build and wire itself into the world in which it lives. When an update was desired, the objects themselves might morph, according to instructions received from the original authors, while retaining customizations they had adaptively designed. In this way, objects change *in situ* and the system / application never really restarts nor does it lose the benefit it has had from living in the world with its users.

9. Buildable Packaging

We imagine that software will be more completely packaged. Instead of just binaries and some support files, each piece of software will contain everything it needs to be further developed by local software developers.[7] Today, if any sort of local development is possible, it's because the source code and required supporting material can be loaded onto a local machine or such a machine is accessible over the internet. In general, only open source and other source-available systems present this option—and even when the source code is available, the right compiler might not be, etc.

A system delivered completely packaged like this would be easy to modify for those capable of it; those whose knowledge and skills are limited will still be able to make stylesheet-based and other configuration-file–based changes. Some of the expected customizations include the ability to adapt the code more precisely to the local environment, perhaps by the encapsulated development system being able to sense its environment and shape the code a little better.

How to implement this is problematic at the moment because there are many sorts and levels of dependencies software can have on its build and execution environments. For example, how can the dependence on a particular version of a library or of the operating system for that matter be captured? These sorts of difficulties can be handled by packaging up all the dependencies (and perhaps storing them in a central location for all instances that share dependencies), but such an approach doesn't handle the executables, like the compilers. The problem is that a compiler needs to execute on a machine that exists, and it needs to produce code for the target machine. This is probably a case where standardization can help: A given platform is very likely to have a compiler for a popular or mandated standard programming language. Another possibility for executables is a virtual machine that executes the compiler; porting the virtual machine should be relatively easy and possibly standardly done.

There can be other advantages that derive from complete packaging. For example, the existence of slightly different versions of some software—which could very well end up interacting with each other—raises such questions as what constitutes the essence of a program and what kinds of variations are permitted while retaining its identity. With such a self-enclosed mechanism comes the reality of a population of individuals[8]—individual instances of the same program, software, component, or system—and with this it becomes possible to think about advancing the species by selective crossbreeding and other means typically used in husbandry. Whereas today a monoculture is highly valued, in the future not only will it have lesser or no value but it may become obsolete. In such a world, the value of common, nonprogram-specific standards for data, control, and behavior exchange will increase.

Another good result is the possibility of eliminating the problem of lost source code, which is a surprisingly pervasive problem with legacy software. The company that created the software you run goes out of business and no one cares about the source code or thinks it's important; an outside group wrote the software for your company and it's backed up on a tape that's then lost. This would never happen were software always packaged with everything you need. And there would be fewer problems re-establishing the work environment.

10. Software as an Active Collaborator / Participant

Software should take responsibility for its own future—not meaning software broadly construed but meaning each recognizable program, package, and application. After all, software can act, it can sense some part of its environment, and it can react to changes, so why should software remain passive once it's been unleashed to the world? The attitude that a program's actions should be limited to what was planned at the factory reminds us of what Marvin Minsky wrote in "Why Programming Is a Good Medium for Expressing Poorly-Understood and Sloppily Formulated Ideas":

> *There is a popular, widespread belief that computers can do only what they are programmed to do. This false belief is based on a confusion between form and content. A rigid grammar need not make for precision in describing processes. The programmer must be very precise in following the computer grammar, but the content he wants to be expressed remains free. The grammar is rigid because of the programmer who uses it, not because of the computer. The programmer does not even have to be exact in his own ideas—he may have a range of acceptable computer answers in mind and may be content if the computer's answers do not step out of this range. The programmer does not have to fixate the computer with particular processes. In a range of uncertainty he may ask the computer to generate new procedures, or he may recommend rules of selection and give the computer advice about which choices to make. Thus, computers do not have to be programmed with extremely clear and precise formulations of what is to be executed, or how to do it.*
>
> Marvin Minksy [25]

Software should be able to examine its environment prior to installation, monitor changes to its operating conditions and adapt to them as best it can, observe the state of its own health

[7] Such a packaging has numerous copyright and intellectual property issues, similar to those found in open source software.

[8] This already is true but not widely recognized. Any real running system runs a variety of release and patch levels.

and attend to it, pay attention to how its human users use it and become easier to use, provide for its own improvement at the hands of local developers (perhaps by noticing and remarking on places where users had difficulties such as frequent requests in one spot to undo an operation), accept extensions and customizations, and, finally accept and provide for its own death and replacement.

In short, software should act like a living collaborator in its own future and not like a manufactured machine waiting for obsolescence to overtake it—there isn't even any minimally valuable scrap metal or plastics to salvage. Just about everything else we buy as consumers is subject to being repaired or tinkered with given enough knowledge and/or bravery—this is the legacy of the physical world: It's just not possible to wall off easily the business end of most macroscopic physical machines the way it's possible to wall off from reasonable intrusion the parts of software (and microscopic machines like CPUs) that make it readily malleable. That is, the source code for a corporeal machine is not behind a firewall on its makers' server.

Some languages—Lisp [23, 24, 36], Smalltalk [12], Self [37], and others—provided some steps in this direction by being *reactive*. Terms used in those times to describe such systems / environments / languages include "exploratory programming," "rapid application development," and "interpreted." In a reactive environment changes are immediate—the response to a change is instantaneous—so not as much "bravery" is needed to dive in and see what happens. Such systems increase the possibility of salvaging an old bit of software. How often has it been just one thing you'd like to change about an application's behavior? In a reactive world you could just make that change and see what happens.

11. Failure is Common

We will assume it. This has been substantiated by numerous studies of software failures and famously long bug lists for most software systems. David Hovemeyer and William Pugh have reported:

> *we have found that even well tested code written by experts contains a surprising number of obvious bugs.*
>
> David Hovemeyer [14]

To write a program or system assuming that nothing will go wrong—even in the parts being written at that very moment—is foolish, and software written this way in a few years will not be tolerated and people who design and program that way will be barred from practicing the art. Here Joseph Weizenbaum's point of view is illuminating:

> *The psychological situation the compulsive programmer find himself in while [fixing a bug in his code] is strongly determined by two apparently opposing facts: first, he knows that he can make the computer do anything he wants it to do; and second, the computer constantly displays undeniable evidence of his failures to him. It reproaches him. There is no escaping this bind. The engineer can resign himself to the truth that there are some things he doesn't know. But the programmer moves in a world entirely of his own making. The computer challenges his power, not his knowledge.*
>
> *Indeed, the compulsive programmer's excitement rises to a fevered pitch when he is on the trail of a most recalcitrant error, when everything ought to work but the computer nevertheless reproaches him by misbehaving in a number of mysterious, apparently unrelated ways. It is then that the system the programmer has created gives every evidence of having taken on a life of its own, and certainly, of having slipped from his control. This too is the point where the idea that the computer can be "made to do anything" becomes most relevant and most soundly based in reality. For, under such circumstances, the misbehaving artifact is, in fact, the programmer's own creation. Its very misbehavior can, as we have already said, be the consequence only of what he has done.*
>
> Joseph Weizenbaum [40]

To understand the source of invasive and pervasive failure consider that no apple failed to fall because someone forgot to tell it to.

12. Seek Out and Repair Errors

We expect software will become like living organisms. We are talking about self-sustaining and self-repairing programs—running code that is buggy but coping with those bugs and perhaps repairing some of them[9] or repairing the damage done. Moreover, we are talking about software that continually adapts to its environment, both at installation time and afterward.

In most software there are data structures which are used by the processes the software embodies. Typically these data structures are by-products of the computation but sometimes they are used to direct parts of it. Such data can be tested for integrity, consistency, and, sometimes, correctness—repairing data can be a potent form of self-repair.

Self-sustaining and self-repairing programs are possible and necessary. Today, safety-critical systems are deployed for which stopping and awaiting human intervention after a failure has occurred is simply out of the question. Even a system for which shutdown is reasonable needs to shutdown safely. Some researchers (Rinard: [31], [32]; Patterson & Fox: [5], [6], [27]; Evans: [10]) have been exploring ideas for self-repair, rapid recovery from errors, and failure tolerance. To many practitioners today, the idea of recovering from an error goes against the grain: the error should never have happened—it's the result of bad design or bad coding. Or bad requirements or specifications, or because the software ends up in an unexpected environment, perhaps because the environment is being updated / upgraded.

The changes we foresee have as much to do with vocabulary and metaphors as with new techniques and technologies. Let's look at one of the first instances of a program that used self-repair rather than absolute correctness.

<story>

In 1958, John McCarthy was thinking about a symbolic differentiation program in a programming language that was later to become Lisp. He was concerned about the "erasure problem":

[9] Self-repair of code can be approached variously. One promising vein is to think about building software through generation—from some model or specification or by some process whose initial conditions can be varied and the code regenerated.

no-longer-needed list structure needs to be recycled for future use. In subsequent languages, such problems were handled either by the structure of the program being restricted to trees (stack allocation of data and its trivially automatic deallocation through stack popping) or by explicit allocation and deallocation (malloc/free). His comment on erasure / explicit deallocation:

> *The recursive definition of differentiation made no provision for erasure of abandoned list structure. No solution was apparent at the time, but the idea of complicating the elegant definition of differentiation with explicit erasure was unattractive.*
>
> John McCarthy [24]

It's worth a pause to notice the style of research described. McCarthy and his colleagues were trying to design a programming language. Part of their methodology was to write the program they thought should be able to do the job and not the program that a compiler or execution system would require to make the program run well. In fact, the *beauty* of the program was foremost in their minds, not correctness down to the last detail. Beauty. Remember that word.

Eventually the first Lisp implementers decided to ignore the bug—the fault of not explicitly erasing abandoned list cells, causing the error of unreachable cells accumulating in memory, leading to a failure to locate a free cell when one is expected—until the failure occurred and to repair it then. This avoided the problem of entangling a common set of functionality (keeping available all the memory that should be) with a pure and clear program (symbolic differentiation). The failure the fault eventually caused was repaired, along with a lot of other similar errors in a process named at the time and still called *garbage collection*.

</story>

The entanglement of erasure code with the symbolic differentiation code reminds us of the problem aspects address:

> *We have found many programming problems for which neither procedural nor object-oriented programming techniques are sufficient to clearly capture some of the important design decisions the program must implement. This forces the implementation of those design decisions to be scattered throughout the code, resulting in "tangled" code that is excessively difficult to develop and maintain.*
>
> Gregor Kiczales et al [16]

The combination of inelegance and the foolishness of pursuing perfection seem to combine:

> ***The Goal of Perfection is Counterproductive:*** *The aspiration to eliminate as many programming errors as possible creates a development process that diffuses the focus of the project, wastes engineering resources, and produces brittle software that is helpless in the presence of the inevitable errors or faults. A more productive aspiration is to develop systems that contain errors and sometimes behave imperfectly, but remain within their acceptable operating envelope.*
>
> ***Flawed Software Has Enormous Value:*** *The most productive path to better software systems will involve the combination of partially faulty software with techniques that monitor the execution and, when necessary, respond to faults or errors by taking action to appropriately adjust the state or behavior.*
>
> Martin Rinard [31]

The moral is not that Lisp researchers scooped the aspects and recovery-oriented programming guys by about 40 years but to note that the metaphor of conscientious computing can be effective if thoroughly adopted. The people who wrote the first garbage collectors weren't bogged down by the belief that a program is manufactured in one place for use in another, but believed that a programming language was based at least in part on a runtime environment in which the software would continually evolve—possibly in response to changing requirements or the changing nature of interdependent software.

In this case, the simplicity and elegance of the program (symbolic differentiation was the example that drove the thinking) was not to be compromised by the intrusion of an unrelated aspect of the program (its memory management needs). Instead the program was allowed to fail (to run out of storage because the programmer "forgot" to deallocate used memory cells), and by a process of observing its environment (noticing it was out of memory), it was able to avert disaster and initiate a process of repair to fix the mess the programmer's faults created. Importantly, the memory management concern has become isolated in a separate module—separate from the program the programmer is interested in, and in fact, separate from every other program.[10]

This is the sort of thought process we believe will become prevalent in the future, but executed with more gusto (and—let's face it—more guts, too).

Garbage collection is not the only example of the sort of programming practices we are talking about. We include as examples utilities that repair and defragment disks, that rotate log files and otherwise cleanup from the normal (and abnormal) activities of a complex system, and that are used as preventatives. Virus-protection software similarly looks from the outside at messages and other downloaded or created files for signs of errors or pathology.

Some of these utilities are triggered by events (like a file being downloaded) and others are scheduled (like log file rotation). But in all cases the problem-detection and repair code are independent of the code that experiences the problem. The code repaired and the repair code are decoupled, which is central in the same way that by being decoupled from the design and implementation of the tested code, testing code is effective. Such a decoupling, though, is not effective when there are similar or common design / implementation points. If both the repair and repaired code need to (correctly) implement the same difficult-to-imple-

[10] A related idea is leasing: A resource is allocated to a particular process or program, but only for a specified time. If the process or program "desires" to retain the resource beyond its lease duration, it must renew the lease before it runs out. If the process or program fails—crashes, goes into a loop—the lease will not be renewed and the resource will be returned to the system. This is like the garbage collection scenario except it is not the failure of a particular component within a system that is at stake but the failure of the entire system. And in the garbage collection case, a program notices that it has allowed its memory to fall into disrepair, while in the leasing situation, the overall system notices the failure only through a form of insurance policy—or one might view it as a form of emergence: nothing is specifically looking for the failure of a component, but the effects of such a thing happening are handled by a mechanism that assumes failure.

ment algorithm, then perhaps the same errors will appear in both pieces of code, and the repair will not be effective.

Moreover, some of the preventatives, such as anti-virus software, perform automatic self-updates over the internet. This is in concert with our belief that software is continually installing itself and actively improving. Such updates are provided by people, but they need not be: They might be from a design farm that is continually improving software through a process of digital software evolution or other automatic mechanisms.

Errors are typically handled by exception handlers—code that is activated when a failure occurs. But the ultimate cause of the failure—the fault—might not be apparent. Therefore, errors must be sought out, through aggressive means like continual testing and other bad smell detection. And as some researchers are now discovering (and the inventors of garbage collection discovered a long time ago), repairing damage and the results of erroneous execution can be an effective way to keep a program or system operating properly.

12.1 Don't Seek Out and Repair Errors

Some errors don't need to be fixed, and in fact, the very act of fixing them can cause more harm than good. [32] Sometimes it makes more sense to observe and note problems, perhaps trying to keep track of coincidental events, files open, etc., with the idea that perhaps with this information—if the error turns out to be serious enough to fix—repairs can be made.

As noted by Rinard and his colleagues, code can be partitioned into *forgiving* and *unforgiving* regions. Errors in unforgiving regions lead to fatal errors or unacceptable results. Errors in forgiving regions may result in bad or marginal results, but typically useful results. The example they give is of software that decodes and plays mpeg files. The part of the code that locates important metadata in the input stream—which enables the rest of the code to properly display the video stream—is unforgiving; an error here causes an infinite loop, early termination, or no video to be displayed. The code that displays video information from the input stream—once that information is found—is forgiving because generally errors there cause only degraded video / audio quality.

Ideally, the system would be able to determine for itself which errors are bad enough to be fixed and which not, perhaps by observing the amount of pain the error causes in users of the software or in other parts of the system.

13. Write Tests and Continually Run Them

Components of a system will be continually running tests—self tests, environment tests, communications tests, etc. This will form a matrix within which various feedback loops will keep the overall system running well. At the moment, most programs and software operate like bulls in a china shop: They blast ahead assuming everything will go their way. Instead, well-designed software in the future will constantly be adjusting to circumstances, righting itself, keeping clear of bad situations. Such feedback loops will exist within the software and also between the software and the software's environment, correcting small errors and making incremental improvements to the organization and performance of the system.

Self-tests and other sorts of tests will help a system discover problems early, so that their correction or amelioration is effective;[11] such tests can be useful to determine whether the environment is changing or degrading. In such a regime, tests should look outward as well as inward—out toward the environment as well as measuring the effects of the environment on the software. When an update is downloaded—along with its new tests—all tests will be (eventually) run and the results sent back to wherever the changes reflected in the update were made. This way, the software developers will have the benefit of the testing environments of all instances of the code, and there won't be as much need for an extensive and varied testing farm. Almost every installation is different, because the exact needs of users depends on the complete local computing environment: different hardware, different drivers, different optional or custom packages and libraries, different patches and patch levels, and on and on. Perhaps even the order of installation makes a difference, particularly when software is customized based on what is already in the system. We don't need to imagine some complicated learning environment—the person or people using a system can customize it, and how they do so depends on what they know, the experiences they have, and the work they do, which are all influenced by what is already installed on the computer.

Many software developers don't like to write tests, although those in the agile software development movement put writing tests at the center of design and implementation. For them, the tests are executable requirements. Their fervor over testing doesn't go far enough: Tests should be designed to run all the time after the software is installed. If the tests are also continually sampling and testing the environment, they can help a program avoid disasters and alert the users of the host computer of problems. Such testing is part of the concept of feedback, a principle on which life depends.

When testing is part of the culture of programs, and once software is shipped with everything needed to modify it, the possibility arises of users creating tests that reflect their usage patterns. Such tests would be useful to the original programmers.[12]

14. Exercise

Many of the behaviors we expect to see in future software come from the idea of ***exercise***. When we currently think about "exercising software," we imagine simply using it or testing it—or perhaps it's closer to the details of walking the dog. But for people, exercise is a way to strengthen, and when we exercise our minds our mental faculties can be improved. Many of our expectations for the future involve the idea of experience changing software. We have known for decades that one of the deep difficulties of

[11] Coupling tests with repair means that the effect of a software bug can be corrected or mitigated (possibly through rebooting, reinitializing a data structure, or returning an acceptable value), but the actual bug remains in the code until fixed by developers.

[12] To the first order, unit testing does its job: Components are verified to work according to the designer's notion of what the component is intended to do. Problems occur when the component is used in an unanticipated way and/or the component becomes a piece of a larger complex system. The system dynamics can cause unexpected behavior resulting from a combination of unanticipated use of components and interactions between components. State and behavior are stored in unlikely places in the system (like the communication pathways between the components); our hope is that a new paradigm of testing could find ways to exercise / test such system dynamics.

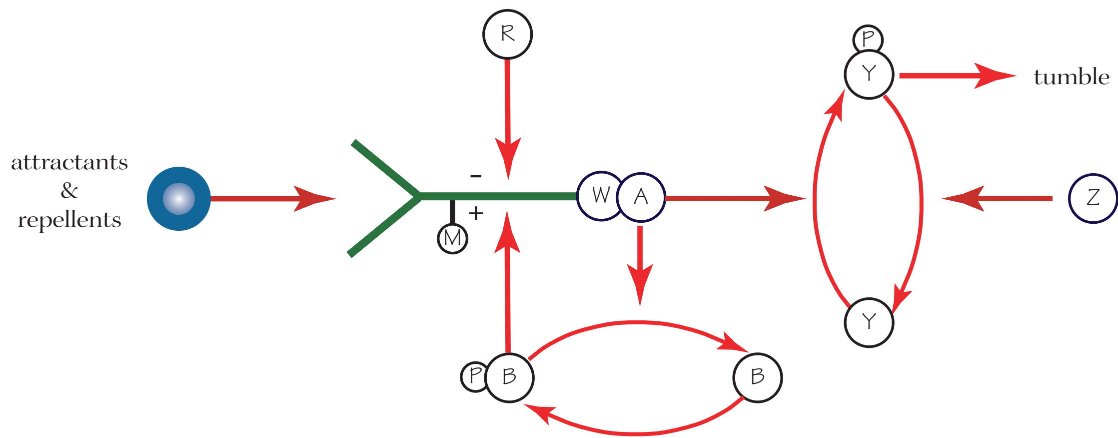

Figure 2: Chemotaxis

producing software is that discovering requirements is hard and trying to do so before some working version of the software is available is impossible. So why should the discovery of requirements stop once the software is installed when the requirements that depend on the local context are all around?

This requires writing software differently—in a way that permits changes *in situ*, something that is difficult but not impossible. [12, 23, 24, 36, 37]

15. Use Feedback

Software should adapt to its surroundings and prevailing conditions. Feedback is a mechanism found in nature and in numerous mechanical designs to provide stability and adaptation in the face of changing conditions.

In biological systems, feedback plays a decisive role in maintaining cellular functions. Chemotaxis [Figure 2] is an excellent example of nested feedback being used by bacteria such as *E. coli* to bias their swimming motion toward food sources and away from toxins. The outmost feedback loop involves sensors that control when the bacteria changes its direction. An inner feedback loop causes the sensors to adapt so they stay sensitive to small differences over a very wide range of concentrations. Together the two loops enable *E. coli* to robustly follow a gradient. [1]

Some algorithms and software that control physical devices use feedback, but otherwise it is a rare component of software. That is, no programming language incorporates feedback as an essential mechanism, but it is, instead, relegated to the programmer to explicitly construct out of low-level language features.[13] Building feedback requires variables, numbers, comparisons, and loops. Therefore, software developers—especially junior ones—don't think in terms of feedback.

—Even though feedback is essential to living systems.

16. Make Things Visible

System components will be able to see out into their environments and into other components. When software has just installed itself, it doesn't have enough experience (or enough time) to scope out how the system is being used, so it needs to come up in a default mode—with text editing, spellchecking, etc., exactly as some developer imagined them back at the factory. By observing other software in action or by looking at the recorded histories of other software running on the system, the newly installed software can learn about the usage patterns and predilections of its users.

Visibility into the environment is restricted today—not because of privacy concerns or inadequate design, but because the concept is not in play. Encapsulation inadvertently is in effect in both directions. It is not desirable to look into a software component by design and it is not possible to look outside of a component (from within it) because it's not a concept in contemporary computing. Communication across a software cell barrier is through an interface—sometimes through arguments as in procedural code or through method calls (or message-passing) in object-oriented code. But as any software developer knows, once inside the conceptual boundary between inside and outside, the programmer (and one might also say, the code) has a specific vocabulary and set of concepts and ontologies, as does the environment outside the component. The bridge that is normally / exclusively built between the two is the signature (the name and shape of the communication) and the types of the pieces of information that go back and forth. This is a very information-poor medium. The very narrowness of the communication channel is the source of many advances in conventional computing. When correctness and optimality is foremost, the narrower the channels of communication the better.

Visibility requires descriptions that are continually updated[14]—for example, descriptions of what's inside a system's software components, how a running system is currently configured and what it's working on, which users use which software and in what ways, etc. And visibility benefits from abundance—a rich

[13] Constraint languages—which are not widely used for application programming—may be exceptions, though feedback is typically buried in control algorithms.

[14] A description is not a representation. A representation can sometimes be thought of as a model of something else, usually in the real world. Manipulations of the representation typically mean something regarding that model. A description cannot be manipulated to create a change in the thing described. A description can be understood, compared to other descriptions, and actions can be taken based upon them. A representation is typically efficient in some way—representing just what is necessary—while a description can be extravagant and mention a number of apparently inessential things.

interface where perhaps pattern matching and some degree of understanding is possible.

This sort of visibility is partly what Lanier is aiming at with phenotropic computing. Lanier thinks of surfaces that can be pattern-recognized or sampled. We are not particular how this is accomplished—perhaps only (translatable) blackboards can suffice with simple textual pattern matching. Or even extensions of something like Common Lisp's keyword / optional argument lists and calling conventions. Or even just posting XML documents.

Making continual testing effective is another reason for visibility: Such things as the overall state of the system, what user-visible windows and processes are available, and what the user's physical environment is like should be visible and, in some cases, alterable. For example, as I (one of the authors) type this sentence, I am also digitizing some old band tapes (the authors were in a rock band together). It would nice if the program I'm writing this paper with knew about the recording software and that it was running, so that it wouldn't interfere with the program capturing all the data in the bitstream from the digitizing hardware, which can happen because the operating system, OS X, doesn't know, aside from some heuristics, how I differentially value the operation of the two programs, nor, more importantly, how disruptions affect the flow of the document-preparation program (typically, not at all) and my perception of my interaction with it (minimal or not at all). The document preparation program could easily surmise that the real-time–like recording is likely to be a lot happier if the program responding to typing were to get a little less priority—such a diminishment would be unnoticeable to me but degradation of the data stream would be noticeable on the recording to everyone who listens to it. And the recording software could likewise look out for itself, noticing I have a multi-CPU machine and conversing with the operating system about how to handle the situation.

17. Continual Noticing

Systems will be aware. For a system to maintain itself, it needs to see itself, it needs some level of self-awareness.[15] [33, 34] It must be possible for a programmer who wishes to write software that operates on behalf of its own well-being to write that code. A current trend is *telemetry*. A system sends a stream of data to another which is monitoring its health. A model like this is part of the solution, but only part. Telemetry represents a time-varying look at one or a few aspects of a system. The visibility required for continual noticing includes the entire status of the system—hardware and software—as well as some information about each component: exceptions thrown, errors trapped, population of inputs handled (for example).

This principle also enables the use of ***stigmergy*** as a self-organizing mechanism. According to the Wikipedia:

> *Stigmergy is a method of communication in decentralised systems in which the individual parts of the system communicate with one another by modifying their local environment.*
>
> [http://en.wikipedia.org/wiki/Stigmergy]

Stigmergy is a concept proposed by Pierre-Paul Grassé in the 1950's to describe the indirect communication taking place among individuals in social insect societies. [13] A termite will pick up some mud (for instance) and invest it with her pheromones. This mudball will tend to attract other termites carrying their own mudballs, and the local effect will be to concentrate the mudballs in a single place, and the global effect will be to create towers and arches which are then made into termite palaces. See also Prigogine [29] and Camazine [4]. In computing an example of this would be a process that writes data into a JavaSpace that other processes can then read, do some computation based on what they've read, and then possibly write new data into the JavaSpace.

An explicit description of the state and history of the software system would enable the construction of self-organizing software systems—not necessarily systems that would self-organize to accomplish their designed functions[16] but that would self-organize to stay alive and healthy.

18. Autopoiesis / Allopoiesis

Programmed systems will become as living. One way that biological systems use feedback, visibility, and continual noticing is as part of their self-generating nature: Many of the feedback loops in a living system are involved with the regulation of production of components, proteins, and other biochemical material that regulates the production of other things including the regulators just mentioned. This is essentially the concept of ***autopoiesis***. Autopoietic systems are

> *systems that are defined as unities, as networks of productions of components, that recursively through their interactions, generate and realize the network that produces them and constitute, in the space in which they exist, the boundaries of the network as components that participate in the realization of the network.*
>
> Humberto Maturana [21] (see also [22])

(Simply:) An autopoietic system is one that is continually (re)creating itself. It is an interrelated network of components that build that interrelated network of components—that is, itself. And even though there might be deep connections to an outer environment, the boundaries of the system are distinct.

In living systems, living is the ultimate goal. And so the production of more living stuff and its adaptation of that living stuff to the surrounding conditions is in many ways more important than the nature of the living stuff itself—as long as it can self-perpetuate (more or less). Our programmed systems are not thought of this way. What is important about a programmed system is what it does. In this sense the software systems we produce are ***allopoietic***: Allopoiesis is the process whereby a system produces something other than the system itself. We are interested in this other stuff and don't care about the mechanism that creates it beyond the fact that it creates it. The health and stability of the mechanism is beside the point as long as the desired production happens. Allopoiesis is the manufacturing process abstracted.

This explains a lot about our currently fairly fragile systems—but to see it requires examining longevity. Until recently, not many programs needed to run for extensive periods of time—operating systems, certain equipment- and human-health–monitor-

[15] "Self-aware" does not require AI or consciousness. A thermostat has awareness. A car's electronic ignition system is self-aware.

[16] Most programs have requirements that prevent them from naturally being written as self-organizing systems, but this shouldn't stop us from trying to look for ways to write such systems.

ing software, and some embedded systems—and therefore our educational system has become geared toward teaching how to design and code programs that are used once or for short periods of time. With the advent of the Web we've seen more systems that need to run for long periods. For example, web servers. These operate on essentially a regenerative basis: Each server request is handled as if it were the first and only request, and the context for the execution is created afresh. That is, no state is carried over from request to request, and it's as if the server were being created each time it is needed.[17]

When a complex, programmed system needs to live for a long time, living becomes the ultimate goal. Coupling this with the need to produce the right answers, we face the following daunting problem: How can we structure a system which needs to recursively generate, realize, and produce itself as well as correctly produce something other than itself? —That is, how do we combine the correct and efficient function of an allopoietic system with the urge to live of an autopoietic one that is continually recreating itself?

If we try to make a system more robust by adding lots of explicit exception handlers and error detection code the program becomes hard to understand and very difficult to maintain. It goes beyond what people can really write.

An obvious alternative answer—one that was pursued a bit during the late 1970s and early 1980s, mostly in the artificial intelligence community—is to create autopoietic systems instead of the allopoietic ones that are too fragile and don't know about living, and for that autopoietic system to also produce the allopoietic result.

One way of doing this requires the process that produces ultimate results (the allopoietic stuff) also to be aware of itself, the state its in, and how well it is producing both itself and its results. The simple garbage collection example demonstrates this. The allocation of fresh storage is part of the correct functioning of the program, while the details of the garbage collection algorithm are concerned with properly and reasonably locating unreachable storage for reclamation. Some garbage collection algorithms don't locate all such storage, leading to memory leaks[18] or use less-than-perfectly timely reclamation. That is, some collectors do a good job, but not a perfect job. Most importantly, the operation of the code that does symbolic differentiation (for example) is separate from that which does the memory management / erasure / garbage collection.

Even this simple example demonstrates the fundamental problem with this approach: Techniques and languages suitable to create an autopoietic system—such as the less-than-perfect garbage collectors—are (probably) not suitable for writing correct, predictable, and deterministic code; code that correctly does banking, for example.

Moreover ... it is instructive to note that the concept of garbage collection was resisted by mainstream language designers until the mid-1990s on the grounds that it was too slow or that the interruptions were too unpleasant or intolerable. Today—almost 50 years after the first garbage collector was written—research papers on how to improve or ameliorate these bad effects of separate garbage collection flood peer-reviewed venues, signaling the growing acceptance of the idea. Nevertheless, software developers today will sometimes still prefer to write a program that will leak memory until the program halts from memory exhaustion over using an automatic system.

The lesson from this is that there will be resistance to the creation of an autopoietic mechanism if it is slow or, worse, buggy.[19]

The flaw in the earlier research programme was to attempt to define a *single* programming language or programming system to serve both purposes—to operate both at the level of arithmetic and logic, procedure calls and method invocations, and also at the level of feedback, visibility, noticing, and living. Any programming language that can operate at the allopoietic level is necessarily fragile, and so injecting those elements in the autopoietic system will break it as well.

Moreover, the principles of the two sorts of systems are incongruous. For example, think about visibility. There is an elegance, or perhaps it's just a symmetry, that the principle of visibility would apply to the autopoietic part of a system while the principle of *information hiding* would apply to the allopoietic part:

> *In computer science, the principle of information hiding is the hiding of design decisions in a computer program that are most likely to change, thus protecting other parts of the program from change if the design decision is changed.*
>
> [http://en.wikipedia.org/wiki/Information_hiding]

The purpose of information hiding plays no role in the execution of a program—no role whatsoever. Only a role during the development of the software.[20] Therefore, information hiding is part of the process of software creation, which takes place within something called the programming environment (in the old days) or the integrated development environment (nowadays). There is no reason that the software cannot be visible (not necessarily alterable) during execution—or even while the software is idle. Here's an example of why.

When a new piece of software is installed, it might wonder—as mentioned earlier—how spellchecking is done in these parts. One way would be to inspect programs that are used a lot and have been used frequently to see whether they provide spellchecking and whether there is a common module or at least a dictionary that the user has augmented. Then the new software could adopt (/ adapt to) the existing preferred method.

That is, visibility / invisibility should be valued differently when viewed vis-à-vis the two different aspects of a program: autopoietic and allopoietic. Invisibility (ignorance of what's really inside a module, a function, an object, etc.) is valuable when a software development team is working to produce a system that will then, while executing, go on to produce something else—there are two allopoietic processes in play: the design / implementation team producing the system, and the system producing its results.[21] Limited visibility—of the module interfaces—at system

[17] To some approximation.

[18] A memory leak occurs when a program allocates a data structure using storage that is never re-used (i.e. deallocated or reclaimed) once the structure is no longer needed.

[19] Even if there is a simplification in the expression of the mechanism. Note that it takes less effort to write code in a garbage collected language, but programmers didn't / don't accept it because of performance worries.

[20] There are, however, execution-time repercussions.

[21] But note: During the design process, the insides of a module are visible

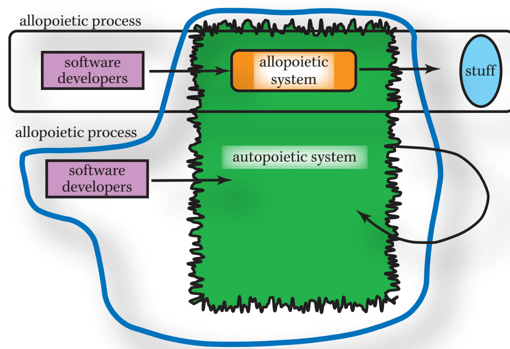

Figure 3: Autopoietic Software System

construction time and at execution time is valuable because it reduces the possibility of change to known interfaces, and the use of an interface enables the ability of the potentially changed module to supplant the old one. When interfaces are visible, it is readily noticeable when they change—compile-time, link-time, and execution-time errors are timely.

Visibility is valuable to the operation of the autopoietic part of a software system, though it may be a hazard to the allopoietic creation of the initial autopoietic system. Perhaps it would make sense to separate the allopoietic part of a system from the autopoietic part; schematically [Figure 3].

The encompassing autopoietic system needs to observe itself and its environment to know how to change and recreate itself. It must also have some ability to observe and affect the operation of components of the allopoietic system.

Other principles are similarly in opposition [Table 1].

Because of this, the requirements of the two types of systems don't line up, and so designing an autopoietic language / environment for allopoietic purposes is unlikely to succeed. The figure [Figure 3] contains the answer. The allopoietic part of the system must remain in an allopoietic language (and runtime when that's appropriate) while the autopoietic part can be in a separate language—designed to do different things and with radically different characteristics.

The autopoietic language must (perhaps) have no ingredients that can cause errors of a nature as to crash a program written in it. A way to think of this is that it must be impossible to write a program with a certain class of bugs in it. Naturally, it's not possible for a person to write a program with no bugs, so what we mean by this must be elaborated. Some bugs are about the ultimate problem, such as getting differential equations wrong, updating cells in the wrong order, or forgetting parts of a specification, while others seem centered around the business of programming *per se*, such as typos, fencepost errors, failing to initialize variables, etc. These two classes of errors feel very different. One is when we are surprised or discouraged that the program doesn't do what we want and the other is when we are upset the program falls over, perhaps after getting nowhere. One is a failure to write the right program and the other is a failure to rightly write the program (we intended).

to some people: Design decisions made within a module are visible to the local decision- and software makers.

Autopoietic Systems	Allopoietic Systems
visibility / transparency	information hiding
dynamic / flexible interfaces	static / rigid interfaces
self-generating / decentralized	command & control / hierarchical
diversity	uniformity
reactive	manufactured
abundance / redundancy	parsimony / efficiency
emergent (no reification)	reification of properties
loosely coupled interaction	tightly coupled interaction
pattern-based	signal-based
local rules	global reasoning
lithe languages	Java™, C++, C#, usual suspects

Table 1: Autopoietic and Allopoietic Principles

In a lithe language, it must not be possible to write a program that falls over. As an example, consider the game of life [9]. It has 3 simple rules for the birth, death and survival of counters on an infinite checkerboard:

- *Survivals.* Every counter with two or three neighboring counters survives for the next generation.
- *Deaths.* Each counter with four or more neighbors dies (is removed) from overpopulation. Every counter with one neighbor or none dies from isolation.
- *Births.* Each empty cell adjacent to exactly three neighbors—no more, no fewer—is a birth cell. A counter is placed on it at the next move.

We have all seen animations of life configurations. A "program" in the game of life is just any initial configuration of counters on an infinite 2-d plane. Some configurations don't do anything interesting at all: They grow forever or die off or fall into a cyclic state. Others are quite interesting—in fact, the mathematician John Conway proved life is universal: That is, it can simulate a Turing machine. But, consider that no initial configuration can have a fatal bug in it in the sense that the game of life will halt or experience some failure. The configuration might not do what you want (due to a fault in placing an initial counter), but *any* initial configuration will run. This is not true in a general-purpose programming language.

Of course, this is what the type theorists aim at with (allopoietic) type theory, but they will fail in the sense that fatal bugs will always be possible in the languages they work on because, after all, those languages deal with numbers and arrays and other inflexible and insensible data structures—data structures that don't, for example, enforce application and domain-specific semantics.[22] And if they were to succeed, well, we would simply use what they produce.

But we see autopoietic languages being lithe. Lithe:

Readily bent; supple. Marked by effortless grace.[23]

[22] This is true of essentially all programming languages—both statically and dynamically typed.

[23] This definition is adapted from "The American Heritage® Dictionary

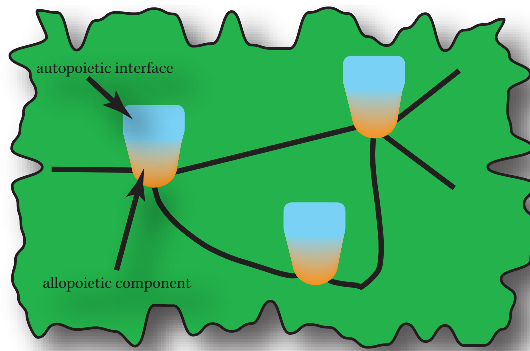

Figure 4: Epimodules

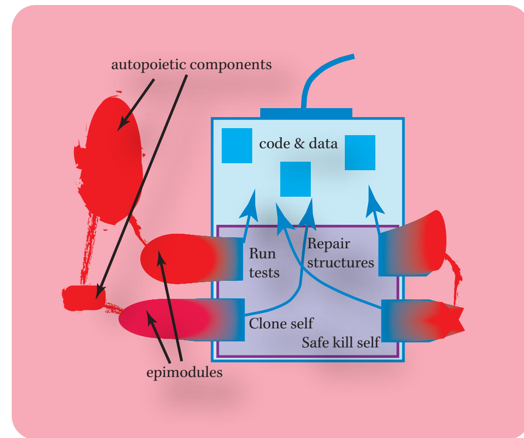

Figure 5: Autopoiesis / Allopoiesis

The structure of a system made this way would be of the allopoietic part of it—the part that does what the designers intended, such as banking or web serving—embedded in the autopoietic part, which would be responsible, in a sense, for keeping the overall system alive. In one possible metaphor, it would be like Einstein's body keeping his brain / mind working properly.

`<digression>`

We imagine a possible structure for this by the attachment (we favor this way of thinking of it rather than as a traditional interface, for reasons described later) of a monitor or autopoietic interface—we will call it an ***epimodule***—to each or some allopoietic components in an allopoietic system. The purpose of the epimodule is to monitor the behavior of the component and to alter it when needed. For example, in a conscientious allopoietic system, the component would include all its test code. The epimodule would be able to pause the component, execute its tests, and examine the results. If a new version of a subcomponent becomes available—say a new text processing component—the epimodule can direct the component to unload / unpatch / ignore the old version, install the new one, run tests, and then either proceed or undo the whole operation.

One of the more important sets of capabilities of epimodules would be to be able to clone and then kill a module. Imagine a scenario in which a component / module has a memory leak or other error whose bad effects accumulate. The autopoietic system through the intermediation of the epimodule can observe and track this problem and at some point decide to clone the component. If the component has no persistent state, the traffic to the old component can be shunted to the new one. If the component has persistent state, the old instance of the component can be directed to clone itself, copying over the persistent state. In some cases—in fact, in most cases—this will have the effect of doing a stop-and-copy garbage collection of the internal state of the component, thereby cleaning up / leaving behind the accumulated damage to the component.[24]

Instead of killing the module, it could be instructed to self-destruct. This mechanism could have similar advantages as ***apoptosis***, or programmed cell death, does in living systems. In apoptosis, the destruction of a cell is triggered by an internal or external signal, and the way the apoptotic process is executed—how the cell commits suicide—facilitates the safe disposal of cell corpses and fragments. Likewise, a system module asked to kill itself can cleanly stop (if possible) by explicitly releasing any system resources it has acquired, thereby working toward the health of the system.

Epimodules, taken in the aggregate, should be able to rewire and hence recreate an entire allopoietic system, and by continually regenerating it, keep it healthy and alive. The requirement, though, is to keep the allopoietic system operating properly: It is performing some precisely specified task, and that must continue. Perhaps some degree of tolerance for uneven performance exists, but even that might not be permitted. The autopoietic system is there to make the overall system more robust, less fragile, more accommodating to users, and its own correct functioning should not become a factor in the correct functioning of the allopoietic system.

Schematically [Figure 4].

A more radical and interesting example suggested by Armando Fox [6] is for an epimodule to observe a module's *mutterings*. A module can be instrumented with a series of mutter statements which when captured as telemetry provides the module's inner dialog. When punctuated by input and output boundaries, the mutterings can be considered messages, and a Bayesian learning process can be applied to a testing corpus to learn what are good and bad messages. Then the epimodule can run the Bayesian network over the messages of a live module to get a sense of whether the module is acting normally, regardless of whether its behavior is correct. These judgments can lead perhaps to a cloning and subsequent death of the original component.

Another possible way to envision the allopoietic / autopoietic structure is to look at a speculative, different, more detailed picture [Figure 5]. The blue boxes represent allopoietic code—the code that does the real work and red blobs indicate autopoietic code. The main big blue box is a component in a larger (allopoietic) system. The bump and string on top represent its normal interface and connections—think of it as the method signature and call graph. The darker (lower) bluish box inside can be thought of as the epimodule interface. Its job is to implement health- and repair-related operations within the component. We see things like the ability to run tests—this includes packaging up the tests

of the English Language, Fourth Edition."

[24] Maybe .... If the problem is in a persistent part, other techniques will be needed.

themselves and some way of characterizing the results—to clone the component, to safely kill the component, or to repair data structures. There could be others as well, such as a tap into the local exception handling and event environments.

The red shapes outside the box, including the encompassing reddish rectangle with the rounded corners, represent autopoietic code. The odd shapes that change shapes and shades from the outside to the inside of the component (from oval to square and from red to blue or from a darker grey to a lighter one) represent the epimodules that assist that component by keeping it healthy and keeping healthy the entire system within which that component exists.

Epimodules can either be developed by the original developers of the allopoietic code[25] or be developed / supplied by the developers or users of the installation site. There might be a standard ecology of health maintenance code at the local site that application systems tie into.

Further, there is no need for epimodules to execute on the same computer systems as the allopoietic code it oversees. Epimodules can execute on another computer system or systems onsite or offsite, perhaps where the developers of the allopoietic system are located. This will reduce the performance impact of the architecture on the implementation.

Epimodule execution can also be asynchronous with respect to allopoietic execution. For example, the epimodules (the autopoietic system) can include extensive simulations to set parameters (e.g. recompiling an allopoietic component with different compiler switches to better tune it for actual use), determine the best configuration, design and implement more appropriate glue, monitoring, or protection code.[26]

The red, blobby, autopoietic components communicate with each other through the reddish background substrate and perhaps by the action of autopoietic entities that operate in the substrate but which are not attached to specific allopoietic components.

We don't know much about what the autopoietic parts should be like. And in particular we have not much of an idea what lithe languages will be like. There is probably a continuum of likely possibilities. But our intuition says that nudging, tendencies, pressure, and influence will be more important than exchanging control and data or traditional control structures and composition techniques.

At the conservative end, lithe languages could be declarative, mostly descriptive—perhaps a set of wiring diagrams for putting together the system from its components, and specifications of the conditions under which to run tests, to clone components, to do internal repair, etc. These descriptions and specifications would be interpreted much as a constraint system is to maintain the functioning of the system and its components. As such, the descriptions might be ineffective but never broken.

At the more speculative (and intriguing) end, the lithe language might be able to express only programs that are dimly aware of themselves and their surroundings, but aware nevertheless. The red, blobby epimodules could be like living cells or other living organisms that are attracted to components either generally or specifically—that is, some epimodules might be tailored for certain components while others could be general. The epimodules would attach themselves to components and sense conditions, both inside and outside the component, and exert pressure on the component to run tests, etc. Not interact through a traditional interface, but sense as if sensing the density of chemicals and exert pressure as if by a suggestion or a push.

The substrate could be thought of as a medium for "smells" that would influence the activities of the epimodules. So, performance monitoring of a component could exude a hotspot smell, which when sensed by other epimodules would cause a migration of a set of components to a stronger cluster or a faster machine. Perhaps it would be useful for the substrate to have a geometry so that concepts of distance and expanding signals could be used to model accumulating evidence or disregard irrelevant, anomalous spikes.

At this end of the spectrum, concepts like population and crossbreeding could make sense so that the epimodules could "learn" or "evolve" to be more effective. Perhaps epimodules would live from nourishment provided by a healthy system and its components. Perhaps epimodules would effect the initial installation using surrogates[27] to optimize placement and connectivity, the real installation taking place after some initial experimentation—or perhaps the surrogates would assemble components or grow them from found / sought parts and pieces in a sort of in-place development / growth process. And so on.

`</digression>`

The concept of autopoiesis flows, in a way, from Immanuel Kant's analysis, where he compares living organisms to mechanical devices. Living organisms:

> *… In such a natural product as this every part is thought as owing its presence to the agency of all the remaining parts and also as existing for the sake of the others and of the whole… the part must be an organ producing the other parts, each consequently reciprocally producing the others.*
>
> Immanuel Kant [15]

Mechanical devices:

> *In a watch, one part is the instrument by which the movement of the others is affected, but one wheel is not the efficient cause of the production of the other. One part is certainly present for the sake of another, but it does not owe its presence to the agency of that other. For this reason also the producing cause of the watch and its form is not contained in the nature of this material. Hence one wheel in the watch does not produce the other and still less does one watch produce other watches by utilizing or organizing foreign material. Hence it does not of itself replace parts of which it has been deprived.*
>
> Immanuel Kant [15]

Allopoietic systems are like watches: The components are there for each other, but not by each other. In other words, each component does not create the others. Order, perhaps, requires continual remaking because the nature of the universe is to unmake order. The digital world is thought to be immune from such effects, being timeless, but what is sometimes forgotten is the fallibility of the makers—the software designers, developers,

[25] Perhaps as a paid service.

[26] Modulo bandwidth constraints. Note that a multicore CPU could minimize even this problem.

[27] We imagine a sort of digital stem cell.

and programmers—who continually create flaws or fail to grasp the fullness of the requirements. And these flaws accumulate and operate on the whole and each other rendering errors over time. This is why rebooting is a common resort.

19. More Principles

Although people have talked about dynamic languages, reflection, meta-models, and the like, conscientious software written as an embedded allopoietic system within an autopoietic system has never really been attempted before. The trip-up has been that while the initial impulse for a programming model might be toward an autopoietic system, the needs of the allopoietic requirements eventually come to the fore and any attempts at a radical design fold back into a traditional mold.

The autopoietic should not harm the allopoietic system. This might mean that the epimodules should not be enabled to actually kill a module. One approach to this is to create a design language for autopoietic systems that is not Turing complete, and in fact quite dramatically limited in its abilities. The autopoietic system might be programmed in terms of ***flows***, where information seeps through the system rather than being instantaneously transmitted from point to point. Markets work because currency flows, because goods flow. The immune system works because blood and other fluids flow. eBay works because reputation (relating to buyers and sellers) flows.

In many cases, flow is slow / through diffusion. A web server component can emit a slowly dispersing overload or hotspot "smell"—perhaps its input queues are overflowing. If the overload persists, the smell around the component will grow stronger and as it disperses it transmits its distress to autopoietic components that, when triggered by a suitable threshold, might replicate the stressed component to relieve the overload. On the other hand, if the overload diminishes, the smell diminishes and doesn't trigger the replication.

Then ***gradients***. This is related to locality. A flow can follow a gradient, perhaps from some region where self-recognizers / non-self–killers are created to an area where non-self has appeared. Another example is in a network: The autopoietic system can send out two flows that disperse at different rates from a persistent store. One flow is to enable a copy of the store to create a replicated site and the other flow disables copying. A copy is made at a site with a probability proportional to the ratio of the concentration of enabler to disabler. In effect, replicants are made not too close, not too far, and not in a predictable manner. [10]

Think in terms of ***layers***. A system doesn't have to be monolithic or based on interacting components. A system based on stigmergy—reacting to elements of the environment—can react to events, to reactions, to reactions to reactions, etc. A different concern can be handled by a separate layer. It can be useful to think in terms of layers rather than components, because a component is an allopoietic concept.

Diversity creates innovation and safety. We learned this from biology and from any creative activity. We've seen many instances in ordinary life of the fact that a diverse base creates fewer points of catastrophic failure. Even Charles Babbage recognized this in the realm of computation:

> *...if care is demanded from the attendants for the insertion of the numbers which are changed at every new calculation of a formula, any neglect would be absolutely unpardonable in combining the proper cards in proper order, for the much more important purpose of constructing the formula itself...*
>
> *When the formula is very complicated, it may be algebraically arranged for computation in two or more distinct ways, and two or more sets of cards may be made. If the same constants are now employed with each set, and if under these circumstances the results agree, we may then be quite secure of the accuracy of them all.*
>
> Charles Babbage [2]

This is not about fault tolerance of the allopoietic part of the system, though there may be some lessons of value there too. An approach to fault tolerance that has been explored and seems like it is an application of diversity is the use of *n*-version programming and voting ([17], for example). Such an approach might be an example were the diversity in the versions to be true diversity.[28] What we are concerned about is keeping the autopoietic part of the overall system working. Therefore, it is not correctness of result that is important but survivability.

When survivability is essential, ***redundancy*** may help. Redundancy achieves two things: tolerance of single fatal errors and the possibility of randomness. When there are numerous members of a population that in the aggregate are achieving some purpose, the loss of a fraction of that population due to error is not important—this is obvious; randomness is less so. For an anthill to forage for food, a number of individuals search each using a random path but leaving a pheromone trail. If one finds a food source, she returns with some food by following her trail back—and adding pheromones to the trail. When other ants sense the trail they follow it, further reinforcing it with pheromones. In this way, the redundancy enables a form of creativity or problem solving.

20. Pandora

> *Hope sole remain'd within, nor took her flight,*
> *Beneath the vessel's verge conceal'd from light.*
>
> "Works and Days," Hesiod [8]

Aside from parts of some operating systems—including virus protection and some of the other examples we've given—we inhabit a world with no autopoietic software. The fear is that with software that is more like a living system—including software that exploits digital software evolution—we (people) will lose control of our software, both as individuals and as a society we might become frustrated that our software takes our requirements more as suggestions than commands. In the worst case, software that is obeying its own imperatives could run amok. This is reminiscent of Pandora's problem.

---

[28] Knight and Leveson report on an experiment where a group of students from two universities are given the same specification, have the nature of the experiment (to study n-version programming) explained to them, and are required to use the same programming language, compiler, and platform—but any methodology they wished—to implement the specification. True diversity would require (vastly) different languages, different language implementations, relatively independent or even poorly related specifications, and design and implementation groups that not only don't know of each other, but are oblivious to the situation. And even then, whether by voting or some other means the *correct* answer could more reliably be gotten is not certain.

Pandora was created by the gods and given as a gift—along with a jar—to Epimetheus, Prometheus's brother. The gods were upset that the brothers had brought fire to humankind, and they placed disease, death, and sorrow in the jar, and told Pandora never to open it. But they had created her with an unconquerable curiosity. Eventually Pandora could not hold back any more and she opened the lid, letting loose the evils inside. Shocked, she closed the lid—but not before the evils had escaped.

Fear of control loss has always been the issue for robots and other intelligent software from the sci-fi point of view. We feel we are in control of our current software applications because they are the result of a conscious design process based on explicit specifications and they undergo rigorous testing. But we know that those applications contain many bugs, some of which may be quite destructive. We have learned to live with this, establishing a certain level of comfort based on the effort put in to testing and debugging the software.

Self-sustaining systems because of their more active nature will require us to get comfortable with a new set of criteria to determine that a program will do what we expect it to. In addition to explicit testing, we will need to rely on the various feedback loops and constraints defined to keep the system in check.

Moving forward with any technology that has the potential of being independent of human control is a touchy affair. One consolation comes from the ending of Pandora's story The gods, in an unusual fit of compassion, had placed Elpis—Hope—in the box. The evils had escaped but Elpis was still in the jar. Seeing Elpis, Pandora let her out and from then on there was the possibility / hint / (false?) hope of cure, of life, and of happiness.

We should start slowly and constantly build systems that self sustain. Feedback loops at all levels of scale can help.

21. Getting There

Getting to conscientious software can be done only in steps, though some of them require breakthroughs and new thinking. Already some steps are being taken: some software automatically updates itself; S.M.A.R.T. hard drive technology monitors disks and predicts errors; some systems support global preferences; and some operating systems engage in a small degree of self-repair, using telemetry or events; web servers spawn small server processes with fixed lifetimes. These are just examples.

Several researchers have been working in this area for a while, and research is getting more daring. A couple of prominent researchers in the area have already been referred to: Martin Rinard, David Evans, Armando Fox, and David Patterson. Others include Tom Knight and Gerry Sussman [18, 38, 39]. Several companies, notably IBM, have had projects in this area whose purpose has been to create near-term mechanisms and products.

More importantly, as we continue to make larger and larger software distributed over a variety of computers in a network, producing more and more variability of co-existing versions, the results based on our current old-fashioned approaches will only get worse. This is not pessimism speaking but an observation of the ways in which the need for accelerated work in this area will become apparent.

We can start on the ideas in this paper by designing systems with epimodules based on software services that are like plug-ins or that operate remotely (like telemetry) to provide some self-repair capabilities. Another simple step would be to make more of the insides of modules visible from the outside in order to observe normal and anomalous behaviors and take corrective action.

22. Conclusions

We are starting to make progress; and once progress is being made, perhaps it will accelerate. We are still in the infancy of computing. It is foolish to think that we have already found all the central concepts. For years we have been struggling to come up with the best metaphor for developing software: is it like writing, or engineering, or like architecture, like horticulture, like voodoo? Few people are ready to realize that programming is not like other things, but other things are like it. Programming is unlike anything people have done before, so the ways are limited in which it is like other things.

It is therefore still ok to pull ideas from lots of disciplines and areas of interest. We have been pulling from a variety of sources to come up with our vision for future software directions.

Our software is like children. For a time parents must provide everything and be prepared to step in to prevent the next disaster. It's not uncommon for a waking child to never be out of one of its parents' sight for years on end. We expect that after a time the child will mature, will grow up, will be able to take care of itself, to solve problems, to cope, and perhaps to contribute something new. Initially selfish—for what other options are there?—the child becomes responsible. With luck or persistence or as the result of good upbringing, the child may become conscientious.

Shall we hope similarly for our software?

Acknowledgements

Thanks: Gary T. Leavens, Kristen McIntyre, Meg Withgott.

23. References

[1] Alon, U., Surette, M. G., Barkai, N., Leibler, S., "Robustness in Bacterial Chemotaxis," Nature, Vol. 397, pp. 168–171, January 14, 1999.

[2] Babbage, C., "On the Mathematical Powers of the Calculating Engine," in *The Origins of Digital Computers: Selected Papers (3rd ed.)*, Brian Randell (Ed.), Heidelberg, Springer-Verlag, 1982.

[3] Brand, S., *How Buildings Learn: What Happens After They're Built*, Penguin, 1994.

[4] Camazine, S., Deneubourg, J., Franks, N., Sneyd, J., Theraula, G., Bonabeau, E., *Self-Organization in Biological Systems (Princeton Studies in Complexity)*, Princeton University Press, 2003.

[5] Candea, G., Brown, A., Fox, A., Patterson, D., "Recovery Oriented Computing: Building Multi-Tier Dependability," IEEE Computer, Vol. 37, No. 11, November 2004.

[6] Candea, G., Fox, A., "Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel," Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), Schloss Elmau, Germany, May 2001.

[7] Dvorak, D., "Challenging Encapsulation in the Design of High-Risk Control Systems," Proceedings of Onward! at OOPSLA 2002, November 2002.

[8] Elton, C., "The Remains of Hesiod translated from the Greek into English Verse," in *The Works of Hesiod, Callimachus, and Theognis*, London, G. Bell, 1879.

[9] Gardner, M., "The Fantastic Combinations of John Conway's New Solitaire Game 'Life'," Mathematical Games, Scientific American, 223, pp. 120–123, October 1970, http://ddi.cs.uni-potsdam.de/ HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm.

[10] George, S., Evans, D., Marchette, S., "A Biological Programming Model for Self-Healing," First ACM Workshop on Survivable and Self-Regenerative Systems, 2003.

[11] Glass, R., "Practical Programmer: Sorting Out Software Complexity," Communications of the ACM, Vol. 45, No. 11, pp 19–21, 2002.

[12] Goldberg, A., Robson, D., *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.

[13] Grassé, P., "La Reconstruction du nid et les Coordinations Inter-Individuelles chez Bellicositermes Natalensis et Cubitermes sp. La theorie de la Stigmergie: Essai d'interpretation du Comportement des Termites Constructeurs," Insectes Sociaux, 6:41–81, 1959.

[14] Hovemeyer, D., Pugh, W., "Finding Bugs is Easy," SIGPLAN Notices (Proceedings of Onward! at OOPSLA 2004), December 2004.

[15] Kant, I., *The Critique of Judgment*, 1790; Prometheus Books, 2000.

[16] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J., "Aspect-Oriented Programming," 11th European Conference on Object-Oriented Programming, LNCS 1241, pp. 220–242, 1997.

[17] Knight, J., Leveson, N., "An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming," IEEE Transactions on Software Engineering, Vol. SE-12, No. 1, pp. 96–109, January 1986.

[18] Knight, T., Sussman, G., "Cellular Gate Technology," Proceedings of UMC98, First International Conference on Unconventional Models of Computation, Auckland, NZ, January 1998.

[19] Krakauer, D., "Robustness in Biological Systems: A Provisional Taxonomy," Santa Fe Institute Working Paper #03-02-008, 2003.

[20] Lanier, J., "Why Gordian Software has Convinced Me to Believe in the Reality of Cats and Apples," in Edge 128, November 20, 2003, http://www.edge.org/documents/archive/edge128.html.

[21] Maturana, H., "Autopoiesis," in *Autopoiesis: A Theory of Living Organization*, Milan Zeleny (ed.), pp. 21–30, New York, North Holland, 1981.

[22] Maturana, H., Varela, F., "Autopoiesis: The Organization of the Living," in *Autopoiesis and Cognition: The Realization of the Living*, (1980), pp. 59–138, 1973.

[23] McCarthy, J., Abrahams, P., Edwards, D., Hart, T., Levin, M., *LISP 1.5 Programmer's Manual*, MIT Press, Cambridge, Massachusetts, 1962.

[24] McCarthy, J., "History of Programming Languages," The First ACM SIGPLAN Conference on the History of Programming Languages, pp. 217–223, Los Angeles, 1978.

[25] Minsky, M., "Why Programming Is a Good Medium for Expressing Poorly-Understood and Sloppily Formulated Ideas," a slight revision of a chapter in *Design and Planning II—Computers in Design and Communication*, Martin Krampen and Peter Seitz (Eds.), Visual Committee Books, Hastings House Publishers, New York, 1967, http://rafael_es_son.typepad.com/ metainformaciones/files/minsky_essay_1967.pdf.

[26] Nardi, B., *A Small Matter of Programming: Perspectives on End User Computing*, MIT Press, 1993.

[27] Patterson, D., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kiciman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., Traupman, J., Treuhaft, N., "Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies," UC Berkeley Computer Science Technical Report UCB CSD-02-1175, March 15, 2002.

[28] A good summary of the Pentium math bug by Ivars Peterson can be found at http://www.maa.org/mathland/mathland_5_12.html.

[29] Prigogine, I., Stengers, I., *Order Out of Chaos: Man's Dialogue with Nature*, Bantam Books, New York, 1984.

[30] Randell, D., "Facing Up to Faults," Department of Computing Science, University of Newcastle upon Tyne, January 2000.

[31] Rinard, M., "Automatic Detection and Repair of Errors in Data Structures," Companion to the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, Anaheim, CA, pp. 221–239, 2003.

[32] Rinard, M., Cadar, C., Nguyen, H., "Exploring the Acceptability Envelope," Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, San Diego, California, 2005.

[33] des Rivières J., Smith, B., "The Implementation of Procedurally Reflective Languages," Proceedings of the ACM Symposium on LISP and Functional Programming, 1984.

[34] Smith, B., *On the Origin of Objects*, A Bradford Book, The MIT Press, Cambridge, 1996.

[35] The Software Engineering Institute (SEI), "The Software Challenge of the Future: Ultra-Large-Scale Systems," June 2006, http://www.sei.cmu.edu/uls/.

[36] Steele Jr, G., Gabriel, R., "The Evolution of Lisp," ACM Conference on the History of Programming Languages, II,

published in ACM SIGPLAN Notices, Vol. 28, No. 3, March 1993,http://dreamsongs.com/NewFiles/HOPL2-Uncut.pdf.

[37] Ungar, D., Smith, R., "Self: The Power of Simplicity," Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Orlando, Florida, 1987.

[38] Weiss, R., Knight, T., Sussman, G., "Genetic Process Engineering," in *Cellular Computing*, Martyn Amos (Ed.), pp. 43–73, Oxford University Press, 2004.

[39] Weiss, R., Knight, T., Sussman, G., "Cellular Computation and Communication Using Engineered Genetic Regulatory Networks," in *Cellular Computing*, Martyn Amos (Ed.), pp. 120–147, Oxford University Press, 2004.

[40] Weizenbaum, J., *Computer Power and Human Reason: From Judgement to Calculation*, W.H. Freeman & Company, 1976.

Principles for Conscientious Software

- *Continuous (re)design happens over the full lifecycle of a piece of software.*
- *Soft / dynamic architectures are needed to allow adding / changing code at runtime.*
- *Software is in(ter)dependent: It must maintain its own integrity while it depends on services provided by other software.*
- *Continual customization and adaptation is required if software is to meet the user's needs in the local computing environment.*
- *Make software self-contained: Everything necessary to build the software should be packaged with it, e.g. source code, tests, etc.*
- *Software should actively participate in its own development, customization, and operation.*
- *Failure is common.*
- *Seek out and repair errors, or tolerate them gracefully.*
- *Write tests and continually run them.*
- *Use feedback.*
- *Make things visible.*
- *Software needs to be aware of itself and of other software it connects with.*
- *Software consists of an allopoietic part and an autopoietic part.*
- *Autopoiesis involves flow, gradients and layers.*
- *Software needs to support diversity and redundancy.*