Living Structure and the Software Garden

Russell Holt
holt-research
9 Golf View Drive
Lafayette Hill, PA 194444
(484) 686-1278
rsl@holt-research.com

ABSTRACT
Complex structures cannot be built; they must grow. Natural living systems grow and are both the products and creators of the environment; nothing exists in isolation. To create software that can approach the complexity of natural forms, we will have to figure out how to grow our software as well. This begins with changing the way we represent it so that changes are graceful and fundamentally stable. One way to do it, which I am developing, is to represent software as a connected web of relationships among the parts. If this representation can be integrated with its environment - that is, all the information that surrounds the creation of the software, all the more likely the system as a whole will reflect reality and pave the way for automated processes to work together with humans in the same environment in the software garden.

Categories and Subject Descriptors
D.2.6 [Software Engineering]: Programming Environments.

D.2.10 [Software Engineering]: Design – representation.

K.6 [Management of Computing and Information Systems]: software development.

General Terms
Management, Documentation, Design, Experimentation

Keywords
Evolution, complexity, life, living structure

1.INTRODUCTION
In this presentation I will talk about software complexity in comparison to natural life. I’ll find that not only do we have a lot of problems at our vastly smaller scale, but that complex systems must be grown, so if we are to achieve complexity comparable to natural life at any scale, we will have to learn to grow our software. And I’ll talk about work I’m doing.

2.SOFTWARE IS DEAD
Compared to my backyard garden, software today is as dead as a doornail. It’s not much more complex than a doornail, either - relative to my garden, that is. Which I’ll talk about later.

The deadness of software was once considered an essential virtue. When it was an extension of mathematics, a program, like a theorem, could be a correct and forever unchanging Truth, free from the decay, uncertainties and contradictions of our lives. But as we have discovered, our needs are not well served by the search for optimal abstractions. Our lives are ruled by messy details and we don't really care all that much about apparent contradictions. We all know how easy it is to create the wrong thing for any number of reasons. But even if we start out on the right track, the world can change. Is that a problem - or is it just reality? Can’t we just nail down the specs?

Well in any case, we build software and it sits there while the world changes. Structural changes are immensely difficult. The only way our software can change, at any level of detail (let’s be honest here) is through our meticulous effort and rigorous testing. Is it not incredible, amusing, and depressing how problems can be so elusive in the fundamentally finite, digital world of software, even on a small scale? It’s no wonder we have to pursue a state of transcendental consciousness to read (and write) good old-school spaghetti code. We know our limitations and they are deep.

It is almost embarrassing, for instance, that here in 2005 we still do version control on text files rather than, at some level, the actual software itself. So when I’m looking at a for loop: why can’t I look at the history of it all by itself? Why can’t that fence-post error bug report be linked to the historical version of the loop? Why should changing a comment be considered... well, a change when it doesn’t affect functionality? Indeed, why can’t anybody add a comment or have a discussion in consideration of some object? How about adding whitespace, or rearranging supposedly equal things - your favorite language aside, something is wrong when there is an ordered sequence to the declaration of things that are considered equal.

This shouldn't be complicated, but it is because software is intractably brittle. Everyone knows that only a tiny change is required to completely destroy a software system, and yet, in contrast, how so very hard it is to kill weeds.

We have a lot of problems and we're still only working on relatively small systems. But complexity on a vast scale surrounds us at all times. Perhaps we can learn something from it.

3.COMPLEXITY AND GROWTH
For software, the troubling thing about complexity is that it seems complex systems must grow. They simply cannot be built the way we think of construction, ie, the blueprint model. We design, then we build what we have designed, and it sort of works for us since our software systems are so very small. Nature doesn't work this way. But the tree that grows around the rock, the fence, the bicycle, the pavement - does it follow a Grand Plan in its growth? Does the tree throw an exception when it encounters a rock, when the wind breaks a branch, when bugs eat the leaves? Is there something or someone writing specifications for the branches, for the leaves?
Imagine if we had to take the tree offline and update its DNA, rigorously test the new process, then put it back online. Ridiculous. But actually that's not so bad: imagine if our actions were the only way the tree could change at all - say, by our piling cell up on cell, or manually reaching in and dividing the cells. Such a thing would not be a tree, it would be a fake. It would be dead. I know this has been said many times before. I'm a fan of Christopher Alexander, too.

Consider also that plants and things that grow both create and are products of their environment. That includes patterns of sun, wind, rain, soil - everything. The process of growth under local conditions significantly affects the resulting structures. In a way, the environment is shared experience among the individuals which, in software, are the running processes. Each individual plant is partially formed by and changes the environment in which it grows. It is a kind of a permanent, ongoing memory shared among all individuals. Imagine if each instance of our programs could benefit from the experience of all (or many, localized) previous instantiations, and could be different (if not better). We don't have anything like this in software.

If we are to create software systems that are as complex as natural life - if only at a very small scale - our systems must be able to grow. They must act like living structures. They must be self-organizing. Life undergoes continuous change while the structures remain stable even as they grow and change from the beginning and stop only when they come to the end.

This property must be fundamental to our new software material.

Unfortunately, approximately nothing in our software development experience has prepared us for this. For instance, even the very idea of source code is the antithesis of living structure. When we edit source code we're not editing the software itself, but instead an abstract description of it. And during the actual edit, the code exists in an unstable state: it cannot be used to create or recreate the executable until the edit is complete. And we honestly don't know if the change will work or not until it is tested. By analogy, it would be as if we stopped time to update the plants in our garden, which might or might not exist after the update.

4. PROPOSAL

So let's fix it. Let's design a system that lets us grow the software. We begin with the idea of the environment, as we have none today. This is the real question - if you have something that grows, how do you get it to be and do what you want? Well the things that surround software software development are, in no particular order - requirements, bug reports, test suites and results, feature requests, patterns, algorithms, references, schedules, tasks, business drivers, email, discussions - and of course, the code itself and all its versions, past present and future - everything. This is the context in which the code is created. This is the software environment, the shared memory which hatches the programs - and which, of course, is changed by their evolution.

We already create and store this information with computers. But it is all isolated and disconnected. Worse, our programs do not have access to it. I can't be the first to say that so-called "reflection" in software is deeply unsatisfying without access to all this other information. Shouldn't a truly reflective program be able to ask such questions about its own context? Have I known bugs? Who is responsible for me? What requirements don't I satisfy? Is this data I've been given likely to produce errors? Need I add: why do I exist? When I say that things need to change with the world, this is what I mean: all of this information we are already creating is, in effect, the description of the world, and it is already under continuous change and micro evolutionary growth, through our tiny actions all day, everyday. A manager adds a task, requirements are being encoded, bugs are being reported and fixed, discussions are constant. This is the world that is under continuous change.

But it needs to be connected; it's not an environment yet. It needs to be cohesive, and it needs to include the program representation, in a kind of information collective. Its structure is that of a web, with links placing things in the context of others. It is useful to see, when looking at a chunk of code, that there is a known bug in it, for instance, or that it is related in certain ways to other pieces of code - a functional back-link, perhaps. We will thus need to change the idea of application as a creator of isolated data into a creator of context. When changes are made to any kind of data, it changes the world, however small, and this affects things. Applications will then be windows into the entire information space, from varying perspectives, placing emphasis on different types of information and relationships.

Here's how it can work, at least, here's what I'm working towards: everything is an addressable object - everything in the information environment I've mentioned and the code in all its versions. Not files and documents; only the granular objects they today contain. We don't think about code, we think about the abstract structure of the software in the past, present, and future. Now we link it all together as appropriate. In code, you can begin with the functional relationships we already have, ie, containment and invocation. But it gets more fun when we mix it into the greater environment. This for loop has a bug! This function is due tomorrow! What we know as comments are merely objects linked - and can grow into real threaded discussions, independently of what we consider executable code. The code as represented in this system can be executed itself or it can act as a source for other tools.

These potentially billions of objects are active - they are sensitive to the environment and may react when things change. Objects that comprise the structure of your software become individually reactive to changes in the environment. They may contain a kind of meta-code which you may use to decide what to do in specific cases. It may do work in the background, so to speak, such as implied by my reflective questions earlier. What should happen when a bug is reported? What should happen when this function produces errors, is too slow, and so on. I would like my programs to notice parts that are too slow or error-prone, and make adjustments. Why not, for some function f, make variations on f, and try them in parallel, perhaps in the background as is reasonable, whenever f is invoked? Surely, by adding information to the environment that is today hidden away in software tools today like performance data, our software ought to, with experience, anticipate, and thus react. But only if it can understand its context.
5. CONCLUSION

If we are to create software of a complexity on the order of natural systems, our software must grow. It must have stable, continuous change over time, with an environmental memory that spans program invocations, perhaps even from site to site, worldwide, among many different programs. Otherwise there is no cycle of life, so to speak, and our software structures will remain isolated, dead.