

Coping with Complexity

Albert Einstein wrote:

A clever person solves a problem. A wise person avoids it.

Source: <http://refspace.com/quotes/problem-solving>

- Introduction
- Defining the concept
 - What are examples of this?
 - Hurd
 - Elasticsearch
 - Google
 - What are other examples?
- Why do things become complex?
 - Intertwining
 - Dependency hell
 - Zawinski's Law
 - Coping with Zawinski's Law
 - Combinatorial explosion
 - Decomposition of complex problems in nearly-independent sub-problems is a complex activity itself
- How much complexity can we cope with?
- What symptoms should we look out for?
 - Innovation slowing down
 - Commit stats evolution
 - Code base growing faster than the community
 - Putting things into perspective
 - Releases become slower or more difficult
 - Developers increasingly want to work on other projects
- Arguments that we are still very far from being on the "second part of the chessboard"
- Yes, but it could happen eventually
- What does the Tiki model do well to cope with complexity?
 - All-in-one model reduces complexity
- Where are the risks and how to mitigate them?
- Things are becoming easier for web apps
- So what now?
 - Remaining use cases
 - Wiki Suite
- Other ideas
- Related links

Introduction

Tiki started in 2002 and it has been the [FLOSS Web Application with the most built-in features](#) since perhaps 2005 or 2006. Since the beginning, Tiki was designed/destined to have a lot of features. Ref: [SourceForge.net Project of the Month, July 2003](#).

Back in the days many web apps had just one "main" feature, for example, [PhpWiki](#) and [phpBB](#). Especially in the early years of Tiki, there were concerns voiced by some (in the community and well-wishers) that having all these features and such an ambitious goal was a huge risk. Essentially, the concern is: "yes, it can work now at a small scale, but as more features are added, the complexity increases and it will eventually implode/grind to a halt."

In June 2011, in the context of a discussion about then-named Tiki Suite (later renamed [WikiSuite](#)), someone pointed out these concerns. I hadn't heard them for years and I figured that the description of the Tiki [model](#) along with Tiki's [success](#) over the years had laid those concerns to rest. However, when one person expresses concerns, it very likely represents a larger number of people that think this way silently. Thus, this page is to

1. explain how Tiki historically copes with complexity
2. speculate on what it would look like if/when we reach difficulties, and
3. in general, offer ideas on how to avoid them in the future.

A Wikipedian was once quoted: "[The problem with Wikipedia is that it only works in practice. In theory, it can never work.](#)" Tiki has a lot of similarities and some people think it can never work, or it can't scale.

Note: For the purpose of this discussion, the terms "extensions", "modules", "add-ons" and "plugins" are all synonymous, referring to code that is not part of the main (core) code base. The use of these is very common for web applications.

Defining the concept

This is not just a code complexity issue. It is also a community management issue. It affects documentation, etc.

"At the heart of the argument is the distinction between accidental complexity and essential complexity. Accidental complexity relates to problems that we create on our own and which can be fixed; for example, the details of writing and optimizing assembly code or the delays caused by batch processing. Essential complexity is caused by the problem to be solved, and nothing can remove it; if users want a program to do 30 different things, then those 30 things are essential and the program must do those 30 different things."

http://en.wikipedia.org/wiki/No_Silver_Bullet

"In technology strategy, the second half of the chessboard is a phrase, coined by Ray Kurzweil, in reference to the point where an exponentially growing factor begins to have a significant economic impact on an organization's overall business strategy."

http://en.wikipedia.org/wiki/Wheat_and_chessboard_problem#Second_half_of_the_chessboard

"The software Peter principle is used in software engineering to describe a dying project which has little by little become too complex to be understood even by its own developers. It is well known in the industry as a silent killer of projects, and by the time the symptoms arise it is often too late to do anything about it. Good managers can avoid this disaster by establishing clear coding practices where unnecessarily complicated code and design is avoided. The name is (...) derived from the Peter Principle — a theory about incompetence in hierarchical organizations"

http://en.wikipedia.org/wiki/Software_Peter_principle

"Economies of scope are conceptually similar to economies of scale. Whereas economies of scale for a firm primarily refers to reductions in the average cost (cost per unit) associated with increasing the scale of production for a single product type, economies of scope refers to lowering the average cost for a firm in

producing two or more products."

http://en.wikipedia.org/wiki/Economies_of_scope

Related links:

- <http://en.wikipedia.org/wiki/Anti-pattern>
- http://en.wikipedia.org/wiki/Software_bloat
- <http://www.zdnet.com/blog/service-oriented/avoid-accidental-complexity-and-96-other-things-every-softw-are-architect-should-know/2436>

What are examples of this?

How did projects that experienced such problems cope? Did they drop some goals? Did they split the project? What can we learn to avoid the issues?

Hurd

Source: [Modular Design and the Development of Complex Artifacts: Lessons from Free/Open Source Software \(2003\)](#) by Alessandro Narduzzo & Alessandro Rossi

It's hard to compare a web app to a kernel, but if one can make the comparison, the unified approach of Tiki is closer to the Torvalds approach.

"The Mach microkernel imposed problems on the HURD developers that have been difficult to surmount, and despite the criticisms of Tanenbaum and others, the choice of a monolithic kernel for Linux made it easier to fulfill the first imperative of all free software developers, which was a working free operating system."

Source:

<http://www.h-online.com/open/features/GNU-HURD-Altered-visions-and-lost-promise-1030942.html?page=3>

Elasticsearch

“

Banon comes from a distributed systems background and says you want to run the compute next to the data. All the compute too; not search in one place, BI in another, machine learning in a third. You should do it in a single system.

Source:

http://www.theregister.co.uk/2014/12/23/elasticsearch_big_data_search_tool_fancy_an_elk_hunt/?page=2

Google

- <http://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/>
- <http://m.cacm.acm.org/magazines/2016/7/204032-why-google-stores-billions-of-lines-of-code-in-a-single-repository/fulltext>
- <https://www.youtube.com/watch?v=W71BTkUbdqE>
- <http://danluu.com/monorepo/>

What are other examples?

Why do things become complex?

Intertwingularity

EVERYTHING IS DEEPLY INTERTWINGLED. In an important sense there are no "subjects" at all; there is only all knowledge, since the cross-connections among the myriad topics of this world simply cannot be divided up neatly. Hierarchical and sequential structures, especially popular since Gutenberg, are usually forced and artificial. Intertwingularity is not generally acknowledged—people keep pretending they can make things hierarchical, categorizable and sequential when they can't. —Ted Nelson

<http://en.wikipedia.org/wiki/Intertwingularity>

http://shirky.com/writings/ontology_outrated.html

http://en.wikipedia.org/wiki/Ted_Nelson

While Ted Nelson coined "Intertwingularity" to express the "complexity of interrelations in human knowledge", it's the same problem for software. There will always be overlap for [features](#) and [use cases](#). For example, a [learning management system](#) has some feature overlap (ex.: user system and calendar) with a [groupware](#) so adding missing functionality (ex.: gradebook) to a groupware is less work than maintaining two apps. So, as Tiki adds features with each version, it progressively becomes easier and easier to cover various use cases (economies of scope).

Dependency hell

"Dependency hell is a colloquial term for the frustration of some software users who have installed software packages which have dependencies on specific versions of other software packages."

http://en.wikipedia.org/wiki/Dependency_hell

End-users of Tiki are pretty much immune to this because all the code is in the core. All features are released at the same time. However, the Tiki development community needs to deal with this as it includes over 100 software libraries from [Composer](#). The Tiki strategy is to try to keep trunk using the latest versions of software libraries and to use the latest requirements to be able to innovate. Ex.: As of 2017-07, WordPress requires PHP 5.2 while Tiki requires PHP 5.6. See: [Version Lifecycle](#).

Zawinski's Law

Zawinski's Law of Software Envelopment (also known as [Zawinski's Law](#)) relates the pressure of popularity to the phenomenon of software bloat.

"Every program attempts to expand until it can read mail. Those programs which cannot so expand are replaced by ones which can."

Coping with Zawinski's Law

When planes crash, do we blame gravity? We must cope with this!

- 37 signals in the book "Getting Real" : "Goodbye to bloat. Simple, focused software that does just what you need and nothing you don't"
- ["Focusing is about saying no" - Steve Jobs \(WWDC'97\)](#)

In a community FLOSS project, it'll be difficult to get consensus on what is "needed" and what is "bloat".

Paraphrasing Clay Shirky, the solvable problem is not feature overload, it's better filters.

Thus, your next option is to have an extension system like Drupal & Joomla! or an [all-in-one model](#) like Tiki.

After a while, some features get so many options that it becomes spaghetti (in the UI, the code or both). It's important to refactor when this happens. The Tiki model permits major changes between versions without abandoning part of the community see: [Adaptability](#).

Combinatorial explosion

"In mathematics, a combinatorial explosion describes the effect of functions that grow very rapidly as a result of combinatorial considerations."

http://en.wikipedia.org/wiki/Combinatorial_explosion

As the feature list grows and everything is supposed to interact with everything, the number of things that can go wrong increases quickly. When there are several extensions for similar purposes, this adds even more complexity.

Decomposition of complex problems in nearly-independent sub-problems is a complex activity itself

In reaction to complexity and software that has way too many features, some will point to the philosophy: Do one thing, and do it well.

"Literature both in management and in computer science has clearly pointed out the pros and cons of modular design and we have already discussed the undervalued difficulties that designers face when they invent modular architectures for complex systems. Along with Simon's perspective, it has been shown that the decomposition of complex problems in nearly-independent sub-problems (i.e. modules) is a complex activity itself (Marengo et al., 2001). At the beginning, designers do not know precisely how to conceptualize the modules of new artifacts; later, when a first conceptualization is reached, they still vaguely know how good is the chosen architecture, compared to the other that have not been considered."

Source: [Modular Design and the Development of Complex Artifacts: Lessons from Free/Open Source Software \(2003\)](#), by Alessandro Narduzzo and Alessandro Rossi

How much complexity can we cope with?

Hard to tell. There is no mathematical formula! But globally, the larger, more diverse, more vibrant and more collaborative the community, the more complexity we can cope with.

- More features and more code bring more complexity.
- More [eyeballs](#) help cope with complexity.
- More features bring more users and, thus, more eyeballs.
 - Either current users of the software stay with it instead of using something else, or the long feature list attracts people.
 - "Essential complexity is caused by the problem to be solved, and nothing can remove it; if users want a program to do 30 different things, then those 30 things are essential and the program must do those 30 different things." — http://en.wikipedia.org/wiki/No_Silver_Bullet

We also can't predict the adaptive capacity of collaborative communities. Wikipedia has succeeded in less than 20 years in gathering much more knowledge than anything else. Few people predicted success at the current levels.

'The problem with Wikipedia is that it only works in practice. In theory, it can never work.'

Tiki "only" has had [over 300 code contributors](#) (see the [list](#) of all code contributors) and many other projects have proven that FLOSS can scale up much much higher than [Dunbar's number](#).

What symptoms should we look out for?

It's one of those things it's hard to tell at which precise instant that complexity "took over" and maybe by the time you see it, it's hard to reverse course.

Innovation slowing down

What if adding new features becomes so time-consuming because of all the things to take into account?

In recent years, thanks to improvements in Tiki, additional building blocks and [things are becoming easier for web apps](#), it's easier than it ever was to innovate.

Commit stats evolution

If commits stats start going down (which is not the case for Tiki), is it because of complexity or because features are more stable?

Code base growing faster than the community

If you look at the lines of code (LOCs) count (which is not a great measure I agree, but just to illustrate), you will see that LOCs of Tiki has a progressive growth while the Drupal/Joomla!/WordPress projects have for the extensions/add-ons an exponential code base to deal with. My argument is that this is caused by feature duplication in the extensions (which brings complexity via the exponential combinations)

Tiki LOCs count is growing slowly with all the features that are added. We regularly proceed to refactoring which simplifies and reduces the code base. [Source Lines of Code](#)

Putting things into perspective

A Tiki 12.2 install contains 14,803 files and it's the [FLOSS Web Application with the most built-in features](#).

About half the code in Tiki is maintained by the Tiki community and the other half is re-using code from [external libraries](#) like Smarty, Zend Framework, jQuery, etc. These are generally in /vendor_bundled: **7511 Files** (over time, everything is being moved to [Composer](#), so the external code is less and less in SVN or SVN externals).

So the Tiki community maintains the remaining: **7,292 files**.

So say we maintain about 7,000 files. Sounds like quite a bit, but let's put this into perspective:

- [Joomla! has 8,351 "extensions"](#)
- [Drupal has 27,604 "modules"](#)
- [WordPress has 32 912 "plugins"](#)

Tiki covers the vast majority of features that these three systems offer via their thousands of extensions. So just about any project you could do with Joomla!, WordPress or Drupal, you could also do with Tiki.

Yet, they have more extensions to maintain than we have files! (and since they can't possibly maintain them all, it leads to dead-end extensions and disappointed end-users).

Also, [compare Tiki vs Drupal code base](#), and you will see that we are coping brilliantly with complexity.

Releases become slower or more difficult

Not only is Tiki the web app with the most built-in features, no other major CMS / Web app has released more major versions in the last 10 years:

- 2009-05: <http://doc.tiki.org/Tiki3> LTS
- 2009-11: <http://doc.tiki.org/Tiki4>
- 2010-06: <https://doc.tiki.org/Tiki5>
- 2010-11: <https://doc.tiki.org/Tiki6> LTS
- 2011-06: <https://doc.tiki.org/Tiki7>
- 2011-11: <https://doc.tiki.org/Tiki8>
- 2012-06: <https://doc.tiki.org/Tiki9> LTS
- 2012-12: <https://doc.tiki.org/Tiki10>
- 2013-06: <https://doc.tiki.org/Tiki11>
- 2013-10: <https://doc.tiki.org/Tiki12> LTS
- 2014-8: <https://doc.tiki.org/Tiki13>
- 2015-5: <https://doc.tiki.org/Tiki14>
- 2016-4: <https://doc.tiki.org/Tiki15> LTS
- 2016-11: <https://doc.tiki.org/Tiki16>
- 2017-7: <https://doc.tiki.org/Tiki17>
- 2018-1: <https://doc.tiki.org/Tiki18> LTS
- 2018-11: <https://doc.tiki.org/Tiki19>
- 2019-6: <https://doc.tiki.org/Tiki20>

It's so fast that part of our community prefers to use [LTS](#) versions!

To be fair, [WordPress](#) also had a similar number of major releases during the same approximate time frame (<https://wordpress.org/about/history/>) but it has way fewer features. More recently than Tiki, [Typo3](#) and [Joomla!](#) have moved to a 6-month release cycle. But in all these cases, with their extension-based model, not all features are ready at the same time. And sometimes, extensions are never ported to the next version. Thanks to the Tiki model, we have [inherent synchronized releases](#).

There is no sign of slowing down, etc. and even if development slowed down (because of maturity of features, fewer developers, etc.), there would still be a new version every six months, albeit with less innovation.

Releases have actually become easier because we have streamlined the process and, thanks to the release schedule, the whole community is synchronized.

Developers increasingly want to work on other projects

If the code base becomes so unpleasant to work with . . . it's important to have clean-up/refactoring projects.

The best answer to this is customer-financed projects with decent timelines. For example, the customer needs some new features and it's more cost-efficient to clean/refactor before adding it. So we must continue to be a great platform for projects and [Consultants/web shops](#).

Arguments that we are still very far from being on the "second part of the chessboard"

Many times, the chess board analogy is used to describe a huge increase in usage / sales / value coming from a network effect. In this case, the exponentially increasing complexity of a system is being highlighted.

If we were on the exponentially hard part, we would see

- an increasing effort for enhancements: instead, things are becoming easier and easier, as we re-use code that is in place and take advantage of better components (economies of scope)
- an increasing effort for releases: our packaging/release infrastructure is getting better and better (with more automation).

Yes, but it could happen eventually

This is really hard to debate as it's something that *may* happen some time in the future and no one can prove anything either way until it happens (regarding the success or the failure). Every year that goes by, one camp says: "I told you so". The other camp says: "wait, it's coming eventually."

It's important to listen to concerns and to act accordingly. Perhaps the warnings have in fact become a [self-defeating prophecy](#)?

What makes you make you think that the community won't adapt?

What does the Tiki model do well to cope with complexity?

- PHP / MySQL / Zend Framework / Smarty / jQuery and, more recently, Bootstrap are our base and we re-use a lot of code. So this reduces our workload. All of these are in evolution (notably ZF2, Smarty3) so our base is as future-proof as it can be. We have offloaded a lot of work to these components and avoided new work by reusing what they offer. In fact, more than half the code shipped in Tiki comes from an [external library](#) and we don't have to maintain it (although some times we need to help). We upstream fixes when we have some. Please see: [Source Lines of Code](#).
- A diverse community, including commercial ecosystem
- Easy to contribute to
- Our [Dogfood](#) is really good now. Tiki, as a community, relies on Tiki to collaborate. For example: our bug tracker was not very good at first and this was an added workload (not only fix bugs, but also improve the bug tracker). But now, Tiki as an application is powerful and mature. Thus, our community is more efficient (all other things being equal) than a community with a less powerful/integrated tool or a tool that they can't tailor to their needs.
- The Wiki Way is really good for dealing with complexity, as proven most notably by Wikipedia.
- A lot of people never thought Wikipedia would become as big as it is. Wikipedia is an example of spectacular volume and growth. It gets bigger and more complex but, with more people, they can address more. And Wikipedia has huge server costs to cover. In Tiki's case, even doubling the number of

features/developers/users won't cause a significant financial risk. We'll need a few more dedicated servers. Except for Google Summer of Code, all contributions to Tiki have been by the community (via consulting companies, IT departments, etc.). The GSoC contributions certainly made some features appear faster than they would have otherwise, but after the introduction, they are community-maintained, without any external funding.


- [Maintainability](#)
 - We proceed to undertake regular cleanups to, for example, push features to the browsers (ex.: spellcheckers). See: [Endangered features](#)
 - We dropped the goal of [database independence](#) to focus on MySQL to keep things simpler and to streamline.
- [Upgradeability](#)
- [Adaptability](#)
 - Tiki can make major changes between versions without abandoning part of the community. Examples of refactoring include
 - Themes in Tiki3
 - jQuery in Tiki3
 - Permissions in Tiki4
 - UTF-8 handling in Tiki5
 - Trackers in Tiki7
 - Comments in Tiki8
 - Bootstrap in Tiki13
 - Todo: (other more recent examples of refactoring will be added here)
- [Release early, release often](#)
- [Paraphrasing Clay Shirky, the solvable problem is not feature overload, it's better filters.](#)
- profiles.tiki.org

All-in-one model reduces complexity

Some people think that *all the code in core* increases complexity. That's because they are just looking at the core. A normal usage of a core + extension app necessarily means that you will use many extensions. Thus, you need to take this real-world usage into account.

Now, an all-in-one code base leads to more core re-use, less duplication, more code review, and avoids dependency hell. See more about the Tiki [model](#).

Where are the risks and how to mitigate them?

- Active Tiki community members know well where the [issues](#) are 
 - Regular refactoring and cleanups needs to happen.
- Too many branches. Ref: [lifecycle](#)
 - Fewer branches would be better
 - [Translation branching strategy](#)
- Hard for new devs to join
 - We could have better docs and more organized mentoring
 - [The influx of newcomers into an organization does not seem to increase defects in its software, perhaps because newcomers get simple tasks at the start.](#)
- Quick release schedule which "loses" part of the community.
 - We have LTS versions. See [lifecycle](#).

- Community SWOT
 - [SWOT](#)

Things are becoming easier for web apps

Software in general is getting better and expectations are higher. But in general, it's quite clear that things are easier every year for a FLOSS web application.

- Better overall ecosystem (better browsers, faster JavaScript, HTML5, CSS3) is making things easier.
- Git vs SVN vs CVS
- PHP5 vs PHP4, jQuery vs hand-coding JavaScript, etc.
 - All these components are generally good with backward compatibility.
- Hardware is faster and can cope with more (Moore's law).
- Internet connections are more readily available, faster and more reliable.
- When we [started our spreadsheet in 2004](#), browsers were slow and it was tough to make it cross-platform. This recent [revamp with jQuery is a day and night contrast](#).
- MapServer vs GoogleMaps/OpenStreetMaps is another example where things have become so much easier.
- Responsive web design with the Bootstrap framework is an advance from [jQuery Mobile](#) + modern handsets, which were a walk in the park [compared to supporting WAP phones](#).
- We tried to add WebDAV support several years ago. It worked but it was so slow that it was barely usable. Now, we are back with [eZ Components's implementation of WebDAV](#)
- UTF-8 support is not what it was in 2002 when Tiki started!

So what now?

Remaining use cases

If we look at the [use cases](#), [Roadmap](#), and [missing features](#), it's clear that it's proportionally a small number compared to what already has been done. And it's mostly just more of what we are really good at. For example, adding a [time sheet](#) to Tiki is not going to be [the straw that broke the camel's back](#)!

It should be repeated that Tiki (as is) is the FLOSS web application with by far the most built-in features. It is also among the fastest if not the fastest release cycle of comparable apps (Drupal, Joomla!, Plone, Typo3, TWiki, etc.). And more features attracts more people. With enough people and with collaboration, we are solidly on top of complexity challenges for Tiki.

Wiki Suite

Will [WikiSuite](#) take us to a tipping point where things start to break down? This is a legitimate risk/concern because WikiSuite is not "more of the same". It takes the Tiki community out of its comfort zone and it involves

- different communities
- different repositories
- different technologies
- different development philosophies
- different release schedules
- etc.

It also introduces [dependency challenges](#) that Tiki has mostly avoided until now.

This being said, every day, there are SysAdmins/IT architects that are building their own "Suite" and coping with the complexity. They are mostly doing this per organization. They may add a bit of re-contributed glueware here and there but it's essentially individual initiatives without community and sustainability.

How can a community be less efficient than these uncoordinated initiatives?

What is proposed is to federate the efforts of this type of person and to get the projects themselves to collaborate. [Tiki and BigBlueButton already have this type of collaboration](#). [Kaltura as well](#).

To reduce the risk:


1 Pick the right components

- Bring community and experience to the project.
- These deal with their own internal complexities.
 - We are not asking people to change their client OS, as all client apps are cross-platform.
- <http://wikisuite.org/Component-criteria>
- Use the same technology when possible.
 - ClearOS is the choice for WikiSuite because it's PHP/MySQL/jQuery like Tiki vs the otherwise excellent option Zentyal that is in Perl.

2 Attract critical mass of eyeballs

The same way some of the Tiki community members are hitting limitations and need something like ClearOS or Jitsi, some of the Jitsi community members need something like Tiki or ClearOS. So because of the complementary nature of the components, each community has those needs. And those energies are not currently canalized in an organized way.

ClearOS reports [118,000 registered systems](#): A good chunk will be interested in complementary features.

For Tiki, we don't have system registration, so we don't know how many installs, except that it's a lot. Most are on shared hosting so the Suite is out of reach (unless a SaaS option is readily available). However, even a small portion adds up to a lot of potential users 

3 Be strict on supported versions

To reduce [dependency hell](#), we'll be very strict on the version numbers that are supported. We'll likely start by following the Tiki release schedule and use whatever component version number is available at that time. We'll try to get the component communities to adopt [synchronized releases](#). And down the road, we'll make an LTS version.

4 Use loose coupling

http://en.wikipedia.org/wiki/Loose_coupling

5 Use/promote open standards

6 Incorporate glueware in the respective component projects (vs maintaining code at wikisuite.org)

mlp wrote:

comment: what is the problem exactly?

This comment would pose that complexity in itself is not a problem. The Tiki development process is basically an organic one - a self regulating self limiting process. Tiki grows the same way as a tree does, branching and growing continuously. The compliment of organic growth for a tree, as in Tiki, is that not all branches or features must live. Lots of branches (features, experiments) go in a direction that doesn't encounter much sunlight (usage). These die and eventually are pruned off, which is a healthy process for the tree overall.

It is certainly possible that Tiki will reach a stage where the growth is less upward and outward, but more of a thickening (cross feature integration). The risks of complexity though, by and large only come into play where complexity meets inflexibility. As a software project, Tiki only guarantees to be itself, what it is at the moment; unlike proprietary projects, it is not bound to serve or upgrade an existing client base; backward compatibility is viewed as important but not essential; projects can stay put on LTS releases. The openness to contributions accepts a lot more energy, and ultimately it is the volume of energy that determines the size of the project and the tree. The bottom line is that Tiki is what its current community of users and developers want it to be, and whatever level of complexity it has now is "just right" for that set of people.

Other ideas

- It would be nice to have a historical chart. I suspect it would show that the number of new features is slowing down and that our code base complexity is under control.
 - Lines of Code
 - Number of preference options
 - Number of active developers (like Openhub.net)

Related links

- <https://joind.in/talk/view/3484>
- <http://www.shirky.com/weblog/2010/04/the-collapse-of-complex-business-models/>
- [Modular Design and the Development of Complex Artifacts: Lessons from Free/Open Source Software \(2003\)](#), by Alessandro Narduzzo and Alessandro Rossi
- <http://css.csregistry.org/tiki-index.php?page=What%20are%20Complex%20Systems%20?>
- [Complexity](#)
- [A debate on complexity leads to a Drupal fork](#)
- <http://thenextweb.com/entrepreneur/2014/09/06/complexity-enterprises-biggest-debt/>

alias

[Complexity](#)