Principles of Antifragile Software

Martin Monperrus
University of Lille & Inria, France
martin.monperrus@univ-lille1.fr

April 11, 2014

Abstract

The goal of this paper is to study and define the concept of "antifragile software". For this, I start from Taleb's statement that antifragile systems love errors, and discuss whether traditional software dependability fits into this class. The answer is somewhat negative, although adaptive fault tolerance is antifragile: the system learns something from errors to errors, and always gets better. Then, I claim that automatic runtime bug fixing (changing the code in response to errors) and fault injection in production (injecting errors in business critical software) correspond to antifragility. Finally, I hypothesize that antifragile development processes are better at producing antifragile software systems.

1   Introduction

Nassim Nicholas Taleb has invented the concept of *antifragility* in his eponymous book "Antifragile" [16]. *Antifragility* is a property of systems (whether natural or artificial). In a nutshell, a system is antifragile if it thrives and improves when facing errors. Taleb has a wide acceptation of "error": it can be volatility (for financial systems), attacks and shocks (for immune systems), death and dishonesty (for human systems), etc.

The central triad of Taleb's book [16] is "fragile", "robust" and "antifragile". Taleb revisits many areas of our life and society through this prism. In the software domain, we know both concepts of fragility and robustness. *Software fragility*, also known as "soft-

ware brittleness" [15] refers to the fact that many software applications crash when they encounter minor changes in their data, code or environment.

Software robustness is part of the more general concept of "software dependability"[2]. Dependability is a generic property including reliability, availability, safety, etc. One definition of "software robustness" is the ability of not crashing given erroneous inputs [2].

Consequently, software engineers naturally feel comfortable with the two first pilars of Taleb's triad, especially if one replaces or equates robustness with dependability. What is then *software antifragility*?

In this position paper, I explore the relations between antifragility and software:

- I claim that "software antifragility" provides new and fresh ideas that complement the body of knowledge on robustness and fault tolerance: robust software resists to failures while antifragile software extracts the intrinsic value of errors.

- I argue that automatic runtime bug repair and fault injection in production correspond to Taleb's characteristics of antifragility. The research community will surely find other antifragile software properties in a near future.

Automatic runtime bug fixing refers to techniques that automatically change the program code at runtime in presence of errors. Recent work has been conducted in this direction [9, 12]. Fault injection in production consists of embedding fault injection in the field, with real scale, real inputs, real users during business critical operations. That may sound

1

crazy, but there are good reasons to believe it is valuable as I will discuss in Section 3. Indeed a successful company called Netflix uses this technique [6, 17]. The antifragile *"loves errors"*, and software is not an exception to the rule:

> *Antifragile software* loves errors.

The rest of this paper is structured as follows. Section 2 briefly outlines software fragility and its counter-measures. Section 3 aims at defining some characteristics of antifragility in software. Section 4 discusses the antifragility of the development process as opposed to the antifragility of the software product.

2 Software Fragility

Anecdotes There are many pieces of evidence of software fragility. For instance, the inaugural flight of Ariane 5 ended up with the total destruction of the rocket, because of an overflow in a sub-component of the system. At a totally different scale, in the Eclipse development environment, a single external plugin of a low-level library providing optional features crashes the whole system and makes it unusable (this is a recent example of fragility from December 2013[1]). More generally, it happens that a small memory error crashes the whole software application with a segmentation fault or a crashing exception. In software design, there is even a well-known problem called the "fragile base class problem" [10]. Software fragility seems independent of scale, domain and implementation technology.

Robustness: Counter-measures to software fragility There are means to combat fragility: fault prevention, fault tolerance, fault removal and fault forecasting [2]. As software engineers who strive for dependability, we do our best to prevent, detect and fix errors. We prevent bugs by following best practices, we detect bugs by extensively testing them and

---

[1] https://bugs.eclipse.org/bugs/show_bug.cgi?id=334466

comparing the implementation against the specification, we fix bugs reported by testers or users and ship the fixes in the next release. In other words, we do our best to create dependable, hence robust software systems

Let us assume that software robustness has become a standard property. Would generalized robustness be enough for resisting the full range of errors that happen in production? The fact is that, up to today, software robustness is not generalized. This may only be due to lack of education, the weight of legacy systems, or the economic pressure for writing cheap code. However, I think that today's software robustness lacks key ideas that Taleb's antifragility crystallizes. To some extent, Taleb's book shows that errors contain some intrinsic value. If that holds, one can do better than just resisting to errors with robust software. *One could extract the latent value of software errors.* This is the goal of software antifragility.

3 Software Antifragility

A antifragile system loves errors. Software engineers do not. First, errors cost money: it is time-consuming to find and to fix bugs. It can even threaten the reputation of companies and eventually their business viability. Second, they are unpredictable: one can hardly forecast when and where they will occur, one can not precisely estimate the difficulty of handling them. Software errors are a plague.

Possibly, instead of damning errors, one can see them as an intrinsic characteristics of the systems we build. Complex systems have errors: in biological systems, errors constantly occur: DNA pairs are not properly copied, cells mutate, etc. Software systems of reasonable size and complexity naturally suffer from errors. Model-checking fails to prove them error-free because of this very size and complexity [15].

But there is still one step to "loving errors". We need a change in perspective. Instead of putting "developers" as subject to Taleb's words *"loves errors"*, let us put "software" as subject.

3.1 Software as subject

When we put software as subject, this gives a different perspective. Software prevents errors by keeping a clean runtime environment, for instance by freeing the memory taken by disposed resources. Software detects errors with error detection code. Software tolerates some errors if it embeds error recovery code and fault tolerance techniques, such as recovery blocks [13].

Error-detection code (aka self-checking [20]) notices that the system is in an incorrect state or failed to perform an action. For instance, a function can warn the caller that an error has happened using an error code or an exception. Another way of detecting errors is runtime assertions. In some programming languages, there is even a programming language concept for error detection: in Java it is the "assert" keyword. Error-recovery code recovers from errors and provide the ability to tolerate some faults. It enables the system to continue providing some service, despite the occurrence of the error. There are many kinds of recovery: best-effort, alternative methods, reboot, etc. and different ways of implementing it. A standard way of implementing recovery in a programming language with exceptions is to insert catch blocks.

Software with error detection and error recovery self-detects and self-recovers from errors if one considers "self" as "no human in the execution loop". However, the location (in which functions?) and the content of this code (what piece of code?) is usually both fixed and manually written. However, error-detection and error-recovery can be partially automated. Automated error detection and error recovery mechanisms is the foundation of research on self-healing [9] and fault-tolerance in general [2].

3.2 Fault-tolerance = antifragile?

Is software antifragility simply error prevention, detection and tolerance? In Taleb's view, a key point of antifragility is that an antifragile system is better and stronger under continuous attacks and errors than without. The immune system, for instance, has this property: it requires constant pressure from microbes to stay reactive. According to this definition, preventing bugs is not antifragile, software may prevent a lot of bugs, it won't make it preventing more. Detecting bugs is not antifragile, software may detect a lot of erroneous states, it won't make it detecting more.

For fault tolerance, the frontier blurs. If the fault tolerance mechanism is static there is no advantage of having more faults. If the fault tolerance mechanism is adaptive [7] and learns something from errors to errors, the system always gets better. We hit a first characteristic of software antifragility: adaptive fault tolerance is antifragile.

> A software system embedding adaptive fault tolerance is antifragile: exposed to faults, it continuously improves.

3.3 Automatic runtime bug fixing

Avizienis, Laprie and Randell say that fault removal, i.e. bug fixing, is one means to attain dependability [2]. Let us take again the self-* perspective and put "software" as subject. Let us imagine software that fixes its own bugs at runtime and call the technique "automatic runtime repair" (also called self-healing [9]).

I propose to distinguish two kinds of runtime software repair: state repair and behavioral repair [11]. *State repair* consists in modifying the program state during the execution (the registers, the heap, the stack, etc.). Demsky and Rinard's paper on data structure repair [5] is an example of such state repair, as well Lewis and Whitehead's paper [8] on repairing event-driven programs. *Behavioral repair* consists in modifying the program behavior, with kinds of runtime patches. The patch, whether binary or source, is synthesized and applied at runtime, with no human in the loop. For instance, the application communities of Locasto and colleagues [9] share behavioral patches for fixing faults in C code.

Traditional fault tolerance is mostly state repair (with rollback, compensation or rollforward [2]). Most self-healing proposals also work on the application state, whether by isolation, reconfiguration or

reinitialization. To this extent, fault tolerance and self-healing are conceptually close, and this has been noted before us [4].

As said in the previous section, a fault-tolerant or self-healing system is antifragile as long as it learns something from bugs that occur. If one defines "fixing" as repairing the code, i.e. behavioral repair, *automatic runtime bug fixing* leads to antifragility, since each fixed bug results in a change in the code, and the system gets better and better. Now let us come back to *"loves errors"*. A software system with runtime bug fixing capabilities loves errors because they trigger continuous improvements of the system itself.

> A software system with dependable runtime bug fixing capabilities is antifragile, in face of bugs, it continually improves.

3.4 Fault injection in production

If you really *"love errors"*, you always want more of them. In software terms, one can create errors – as many as one wants – using a technique called fault injection. A fault injector is a piece of software that creates many kinds of errors. For instance, it may change the value of a variable (changing $i = 0$ to $i = 1$), it may temporarily change a piece of code (calling $increment()$ replaced by $decrement()$. So, literally software that *"loves errors"* self-injects faults continuously. Does it make sense?

By self-injecting faults, a software system constantly exercises its error-recovery capabilities. If the system resists to those injected faults, it will likely resist to similar real-world faults. For instance, in a distributed system, servers may crash or be disconnected to the rest of the network. In a distributed system with fault injection, a fault injector may for instance randomly crash some servers (an example of such an injector is the Chaos Monkey discussed below [6]).

Ensuring the occurrence of faults has different positive effects on the system. First, it forces the engineers to think of error-recovery as first-class engineering elements: the system must be able to at least resist to the injected faults [17]. Second, it gives engineers and users some confidence about the error recovery capabilities; if the system can handle those injected faults, it is likely to handle real-world natural faults of the same nature. Third, by injecting faults randomly, it enables one to test the error-recovery mechanisms on a variety of diverse runtime states, each injection giving the opportunity to learn something on the system itself and the real environmental conditions.

Because of these three effects, injecting faults in production makes the system better. This corresponds to the main characteristic of antifragility: "the antifragile loves error". Compared to adaptive fault tolerance and automatic runtime bug fixing, it is not purely the injected faults that improve the system, it is the impact of injected faults on the engineering process. I will come back on the profound relation between the product and the process in Section 4.

The idea of fault injection in production is unconventional but not new. In 1975, Yau and Cheung [20] proposed to insert fake "ghost planes" in an air traffic control system. If all those ghost planes land safely while interacting with the system and human operators, one can really trust the system. Recently, a company named Netflix released a "simian army" [6], whose different kinds of monkeys inject faults in their services and datacenters. For instance, the "Chaos Monkey" randomly crashes some production servers, and the "Latency Monkey" arbitrarily increases and decreases the latency in the server network. This company is healthy and makes large benefits. When fault injection is done in production on a special day under full control (as opposed to automatically at any arbitrary point in time), it is called a GameDay exercise [1].

Automated fault injection in production, which I call *"self-injection"*, complements the other self-* properties (e.g. self-healing) discussed above. This concept is not mentioned in the cornerstone report by Avizienis, Laprie and Randell. [2]. Would these authors admit that fault self-injection in production is a dependability technique? I think so, since they define dependability as *"the ability to deliver service that can justifiably be trusted"*, which exactly fits our trust argument.

> A software system using fault self-injection in production is antifragile, it decreases the risks of miss, mismatch, and rot of error-handling code by continuously exercising it.

Beyond the vision, in reality, there is probably a balance between the dependability losses (due to injected system failures) and the dependability gains (due to software improvements) that result from using fault injection in production. There is probably a mirror tradeoff between the short term business losses due to injected faults and the long term business gains resulting from having a more dependable software system.

4 Software Development Process Antifragility

On the one hand, there is the software, the product, and on the other hand there is the process that builds the product. In Taleb's view, antifragility is a concept that also applies to processes. For instance, he says that the Silicon Valley innovation process is quite antifragile, with stochastic tinkering at all levels, from inventors to investors. The process deeply admits errors, and inventors and investors both know that many startups will eventually fail. I now review some aspects of the software development process that can be considered as antifragile.

4.1 Test-driven development

In test-driven development, developers write automated tests for each feature they write. When a bug is found, a test that reproduces the bug is first written; then the bug is fixed. The resulting strength of the test suite gives developers much confidence in the ability of their code to resist changes.

Concretely, this confidence enables them to put "refactoring" as a key phase of development. Since developers have an aid (the test suite) to assess the correctness of their software, they can continuously refine the design or the implementation. They refac-

tor fearlessly, having little doubts that they can break anything that will go unnoticed.

Furthermore, test-driven development allows continuous deployment, as opposed to long release cycles. Continuous deployment means that features and bug fixes are released in production in a daily manner (and sometimes several times a day). Again, it is the trust given by the automated tests that allow continuous deployment: developers do not need long manual testing phase before releasing.

What is interesting with test-driven development is the second order effect. With slow releasing, errors can be hard: they happen all together, they interact one with each other in a possible catastrophic way, they require to maintain an old version of the source code to fix them. With continuous deployment, errors are made smaller. When an error is found in production, the new version can be released very quickly since the process supports very quick verification and deployment phases. When an error is found in production, it applies to a version that is close to the most recent version of the software product (the "HEAD" version). Fixing an error in HEAD is usually much easier than fixing an error in a past version, because the patch can seamlessly be applied to all close versions, and because the developers usually have the latest version "loaded in mind". Both properties (ease of deployment, ease of fixing) contribute to minimize the effects of errors. We recognize here a property of antifragility: *If you want to become antifragile, put yourself in the situation "loves errors"—to the right of "hates errors"—by making these numerous and small in harm.* (Taleb [16]).

4.2 Bus factor

In software development, the "bus factor" measures to what extent people are essential to a project. If a key developer is hit by a bus (or anything similar in effect), could it bring the whole project down? In dependability terms, such a consequence means that there is a failure propagation from a minor issue to a catastrophic effect.

There are management practices to cope with this critical risk. For instance, one technique is to regularly move people from projects to projects, so that

nobody concentrates essential knowledge. To the extreme "If a programmer is indispensable, get rid of him as quickly as possible" [19]. On the short-term, moving people is sub-optimal. From a people perspective, they temporarily lose some productivity when they join a new project, in order to learn a new set of techniques, conventions, and communication patterns. They may even feel frustrated and unhappy because of this. From a project perspective, when a developer leaves, the project does experience a small slow-down. The slow-down lasts until the rest of the team grasps the knowledge and know-how of the developer who has just left.

However, from a long-term perspective, it decreases the bus factor. In other terms, moving people transforms rare irreversible large errors (project failure) into lots of small errors (productivity loss, slow down). This is again antifragile.

4.3 Conway's law

In programming, Conway's law states that the *"organizations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organizations"* [3]. Raymond famously puts this as *"If you have four groups working on a compiler, you'll get a 4-pass compiler"* [14]

More generally, the engineering process has an impact on the product architecture and properties. In other terms, some properties of a system emerge from the process set to build it. Since antifragility is a property, there may be software development processes that hinder antifragility and others that foster it. The latter would be "antifragile software engineering".

I tend to think that the engineers that set up antifragile processes better know the nature of errors than others. I tend to think the developers enrolled in an antifragile process become imbued of some values of antifragility (Tseitlin'paper "The Antifragile Organization" refers to the same idea [17]). Because of this, I hypothesize that *antifragile software development processes are better at producing antifragile software systems.*

5 Conclusion

Many software applications are fragile. Software fragility may be due to an error by commission (too many things, too much) or an error by omission (too few things, too little). This is Taleb's opposition of Book VI [16] between addition and substraction.

Software fragility by commission means that there exists design principles and implementation techniques that are fragile per se. Identifying them and subsequently eradicating them would decrease software fragility. Software fragility by omission means that there may exist neglected design principles and implementation means that would result in the opposite of fragility, that is antifragility. Those would be antifragile principles.

In this paper, I focused on the latter to contribute to defining of antifragile software. I compared it against traditional fault tolerance and highlighted two means for achieving antifragile software, automated runtime bug fixing and fault-injection in production.

This is only the beginning, there surely are more principles of antifragile software, and the software community need to translate them into execution environments, libraries, etc. Since the legacy software code base is huge, a research avenue is to invent ways to develop antifragile software on top of existing brittle programming languages and execution environments. That would be a 21th century echo to Van Neuman's dream of building reliable systems from unreliable components [18].

References

[1] J. Allspaw. Fault injection in production. *Communications of the ACM*, 55(10):48–52, 2012.

[2] A. Avizienis, J.-C. Laprie, B. Randell, et al. Fundamental concepts of dependability. Technical report, University of Newcastle upon Tyne, 2001.

[3] M. E. Conway. How do committees invent? *Datamation*, 14(4):28–31, 1968.

[4] R. de Lemos. ICSE 2003 WADS panel: Fault tolerance and self-healing. `http://www.cs.kent.ac.uk/events/conf/2003/wads/panelReportWADS2003.pdf`, 2003.

[5] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. *ACM SIGPLAN Notices*, 38(11):78–95, 2003.

[6] Y. Izrailevsky and A. Tseitlin. The Netflix simian army. `http://techblog.netflix.com/2011/07/netflix-simian-army.html`, 2011.

[7] Z. T. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant. Chameleon: A software infrastructure for adaptive fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):560–579, 1999.

[8] C. Lewis and J. Whitehead. Repairing games at runtime or, how we learned to stop worrying and love emergence. *IEEE Software*, 28(5), 2011.

[9] M. E. Locasto, S. Sidiroglou, and A. D. Keromytis. Software self-healing using collaborative application communities. In *Proceedings of the Symposium on Network and Distributed Systems Security*, 2006.

[10] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *Proceedings of the European Conference on Object-oriented Programming*, pages 355–382. Springer, 1998.

[11] M. Monperrus. A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the International Conference on Software Engineering*, 2014.

[12] J. H. Perkins, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, M. Rinard, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, and S. Sidiroglou. Automatically patching errors in deployed software. *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems*, 2009.

[13] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220 –232, june 1975.

[14] E. S. Raymond et al. The jargon file. `http://catb.org/jargon/`, last accessed Jan. 2014, -.

[15] M. Shaw. Self-healing: softening precision to avoid brittleness. In *Proceedings of the first workshop on self-healing systems*, 2002.

[16] N. N. Taled. *Antifragile*. Random House, 2012.

[17] A. Tseitlin. The antifragile organization. *Commun. ACM*, 56(8):40–44, Aug. 2013.

[18] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies*, 1956.

[19] G. M. Weinberg. *The psychology of computer programming*. Van Nostrand Reinhold New York, 1971.

[20] S. Yau and R. Cheung. Design of self-checking software. In *ACM SIGPLAN Notices*, volume 10, pages 450–455. ACM, 1975.