Editor: **Cesare Pautasso**
University of Lugano
c.pautasso@ieee.org

Editor: **Olaf Zimmermann**
University of Applied Sciences of Eastern Switzerland, Rapperswil
ozimmerm@hsr.ch

Software Retrofit in High-Availability Systems

When Uptime Matters

Thomas Ronzon

Performing software maintenance on business-critical legacy systems requires not only software engineering but also archeologist, detective, and caretaker skills. In this installment of Insights, Thomas Ronzon shares road stories from more than 20 years of professional services for clients in the logistics industry. From his experiences, he has distilled a stepwise approach to software retrofit for understanding, stabilizing, and improving systems without disrupting the day-to-day business operations depending on them.
—Cesare Pautasso and Olaf Zimmermann, department editors

THE TIMES WHEN software was developed from scratch have passed almost everywhere. But what if you need to functionally enhance or renew high-availability software owing to obsolete technology? Such problems worsen if you're dealing with "ghost town software": software that's still in use, but all its developers have left.

I've often found software like this in companies in which IT isn't a main competency—for example, a company that uses a software-intensive system to run a high-bay warehouse for spare parts. In such companies, everybody uses the software system daily. However, no one cares how it works until it no longer works or must be modified because the world around it, including the business context and technical environment, has changed.

To address such problems of business-critical systems that are no longer maintainable, I've applied a *software retrofit*. In engineering, retrofit is a common term that describes, for example, improving power plant efficiency by renewing an old plant instead of building a new one.

What Software Retrofit Is and Isn't

A software retrofit changes old software step-by-step while daily operations are going on. The goal is to reestablish a system you can easily extend and maintain with state-of-the-art development and operations technologies. Because systems such as the high-bay warehouse management system must run 24/7, uptime matters. You can't stop them for a time period that's longer than a lunch

0740-7459/16/$33.00 © 2016 IEEE

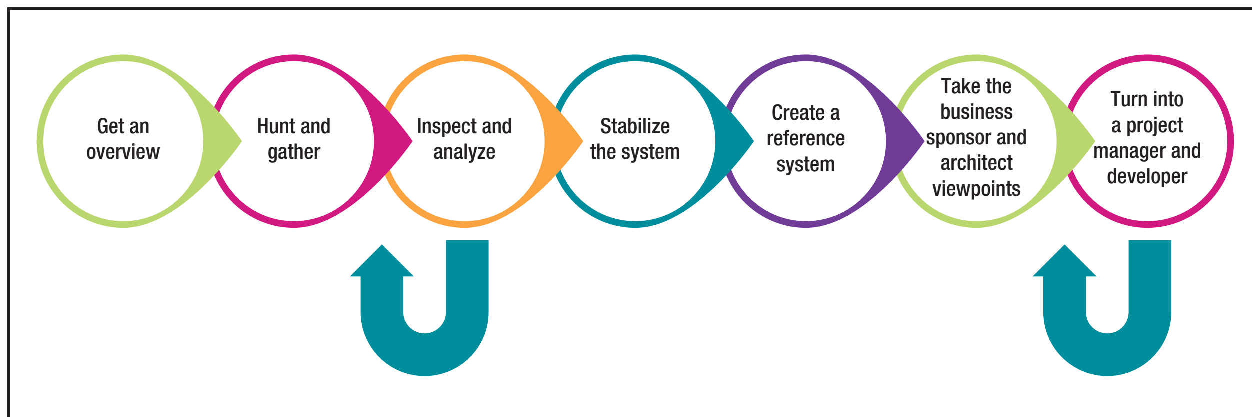


FIGURE 1. Software retrofit's seven steps. A retrofit changes old software while daily operations are going on. The goal is to reestablish a system you can easily extend and maintain with state-of-the-art development and operations technologies.

break. So, you must conduct many incremental changes that are as small as possible.

A common approach that's similar to software retrofit is "hard reengineering"—porting software from one OS to another.[1] Similarly, in "soft reengineering,"[1] the technologies remain the same but the system is reengineered to make it fit for new functional or nonfunctional requirements.

Software retrofit is more than that. It renews the entire system even as its users are accessing it. It shouldn't be confused with tinkering with the system or extending the system without a significant architectural impact (such as by building a new GUI that merely wraps the underlying code).

The Problem to Solve

In many organizations, software such as that in my warehouse example has been running for years. However, the know-how about the software is codified in the software but not in the documentation (if any documentation exists at all).[2]

Assuming you can reconstruct the knowledge buried in the software, availability is the key design concern in such a business context. If the business must be stopped during a software update, the company loses money directly. Failure to deliver certain goods can lead to a temporary standstill of the company, or—maybe even worse—it can force customers to buy from a competitor. In many of my project assignments, the total costs (including personal costs and penalties to customers) have been 10 times higher than the software costs. For example, a standstill of a high-bay warehouse can cost several hundred thousand dollars per shift because a stoppage of goods delivery implies a complete stoppage of production.

The Seven Steps

Software retrofit involves seven steps (see Figure 1). While practicing them, you'll find a lot of lost knowledge about the system, so you'll be able to find a way inside the mind of the original developers. With this knowledge, you'll be able to remove technical debt and use new techniques to implement new requirements. At each step, you can apply many standard software engineering (and reengineering) concepts, from small practices such as code refactoring to comprehensive offerings such as the TOGAF Architecture Development Method or the Object Management Group's Architecture-Driven Modernization approach. Here, I feature only those practices that have served me well on my retrofit projects.

Step 1: Get an Overview

First, get a feeling for the software. It's not enough to consult the code to understand what the software is doing.

The following questions can help you get an initial overview:

- What's the software good for?
- What hardware does the system use? Where is it hosted?
- Who works with the system? How many users access it concurrently?
- Who owns the software (copyright and legal responsibility)?
- What contextual dependencies exist, and what's the software's operating range?[3]

Step 2: Hunt and Gather

Next, visit the customer. At this stage, you morph into a hunter-

gatherer. You have much data to compile. Anything might be useful—not only source code:[4]

- Talk to people who might know something about the system under retrofit.
- Visit their offices to find anything that might be useful.
- Look at the history of the server's command shell (if you have access).
- Look in cupboards and desktops near the system to see whether you can find something useful, such as documentation (digital, on paper, and so on).
- Be creative. Where would you note useful things about such a system?

Figure 2 shows a sheet I found on a notice board on the customer's premises. It was the only existing written documentation of a conveyor system in an automated warehouse. It helped me a lot because it showed the internal naming of relevant conveyor system elements, which we later found while sniffing the network traffic.

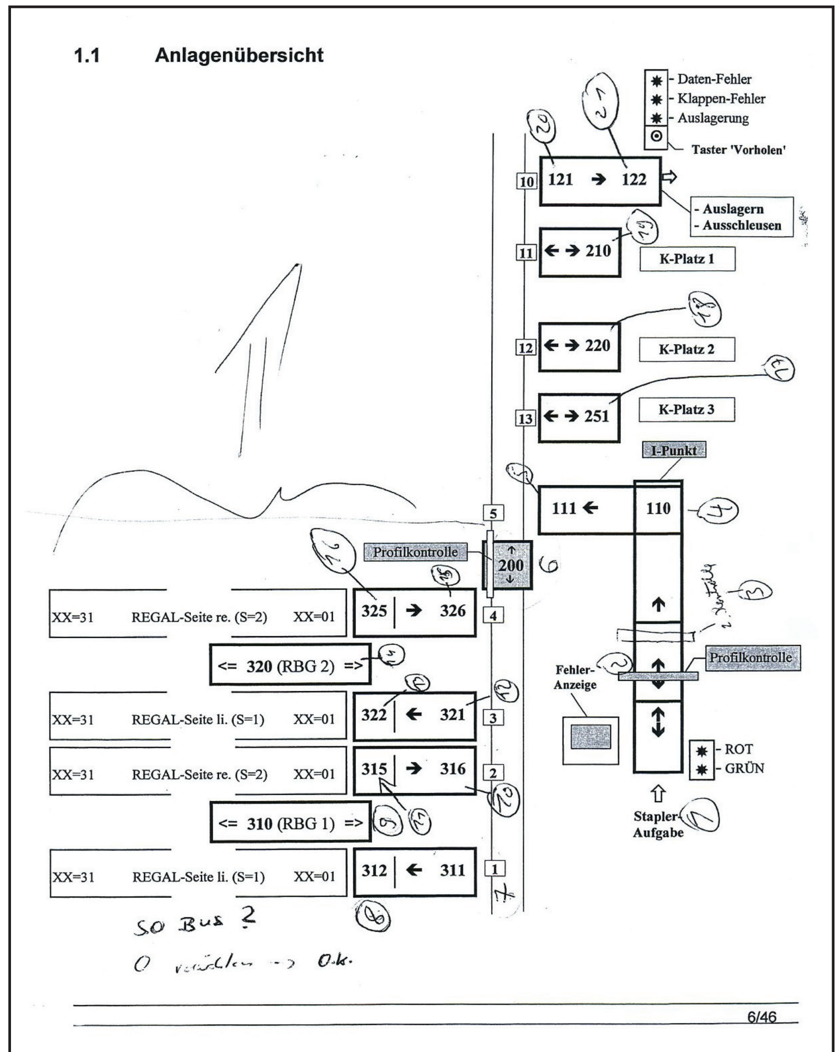


FIGURE 2. This sheet, which I found on a notice board on the customer's premises, was the only existing written documentation of a conveyor system in an automated warehouse. It helped me a lot because it shows the internal naming of relevant conveyor system elements, which we later found while sniffing the network traffic.

Step 3: Inspect and Analyze

Now try to group the information you found into technical and organizational parts. A mind map[5] can help a lot here; Figure 3 shows an example. Divide the information items such that every tree level has a maximum of five leaves. Then, compare your findings with what you expected to find (identifying gaps is an important skill for successful reviewers):

- What business does the system drive?
- Who runs the system (people in operations)?
- What are the organizational limiting conditions (for example, regular breaks or accepted downtime)?
- What information about the software did you find?
- What information was missing?

For example, you know that data must be stored somewhere. But be careful. Not everything is actually what it seems at first glance. For instance, just because a database exists doesn't mean it stores all the data.

Also watch out for things you normally wouldn't expect. I once found a router that was intended as a service gateway for the system's old maintainers (I had received this information from the IT department).

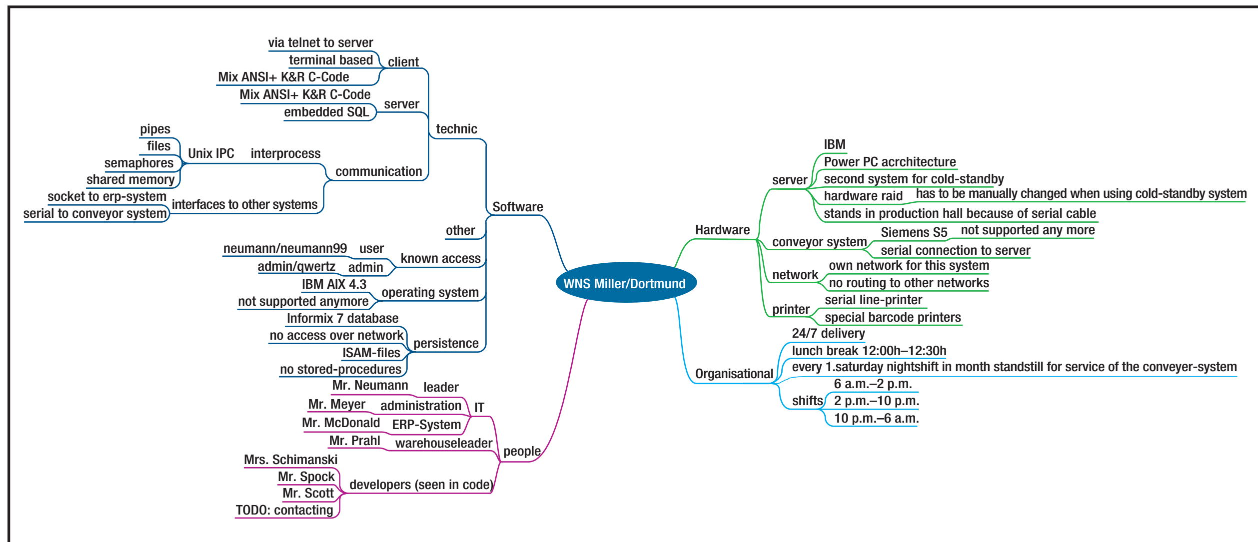


FIGURE 3. A mind map of information related to a system for running a warehouse. You can use such maps to group the information you find into technical and organizational parts. You can create multiple levels of subtopics to further organize these parts.

That intent was fine, but its realization had one small problem. The router had only one LAN cable attached to it (and there was no WLAN card). So, which networks could it possibly connect? A deeper investigation unveiled that the real service gateway was an FTP server configured with an anonymous account on the production machine, which I found after a port scan of the server. This obviously was a severe IT infrastructure security exposure and threat.

If you can't find any information, start with another topic from your mind map. Write down all things that aren't clear at the moment (that is, keep a "discovery backlog"). To find more information, you can either go back to step 2 and revisit the system or find a creative solution. Regarding that database I mentioned earlier, you could try to find out whether a database connection is configured on the computer.[6]

You often can find people who remember these things when you ask directly. Try to find out who worked in a certain organizational unit when the system was initially installed (for example, by looking at old project minutes or org charts). Maybe he or she is still in the company but has a different role. Be friendly to these people; they'll talk to you if their memories are fond (see social hacking[7]). Try to improve your mind map step-by-step.

Step 4: Stabilize the System

With your mind map, you should be able to identify the top three risks. Often, you can easily stabilize or mitigate them.

For example, if no backup exists, try to configure one. If you encounter broken hardware, be creative. A customer had an old server that wasn't on the company network. All terminals had a serial connection, and there was no "free terminal" for access to the server in the computer room. Several keys on the server's console keyboard were defective. This was a special keyboard, and I couldn't find a replacement on the Internet. The company's IT folk had tried to repair it, but it was still broken. So, I took an old PC, installed Linux, built a small network to the server, and logged onto the server via Telnet (instead of the console) to administer it remotely.

If there's a permanent problem (one you can't fix or work around), try to find out what went wrong, why, and what the impact will be on development and maintenance. (In other words, conduct retrospective risk management.[8]) If you can't easily stabilize the system, try to define fallback scenarios. These can involve manual handling if necessary. But ensure you have everything for these scenarios in case of failure. What do you think could happen?

For example, one of my customer's old programmable logic controllers (digital computers used for automation) broke down, so the warehouse cranes couldn't drive anymore. As a workaround, I created a picking list with a small script (which read the

picks directly from the database) so that the employees could print it and get the most important goods by hand, using a safety harness and a ladder. Caution: safety first!

Step 5: Create a Reference System for Testing

Working directly on a running system is dangerous. Just because the system is stabilized doesn't mean you can change everything easily without errors.

So, build a test system that's as similar as possible to that system. An easy way is to create a virtual machine directly from the working machine—if the virtualization software supports the OS. If this doesn't work, still try to build a system that comes as close as possible. Often, you won't have your own test hardware for the underlying programmable controller. In my retrofit projects, I use software that emulates the controllers as faithfully as possible. The same applies to interfaces to other software. Try to build mock-ups.

Again, be creative:

- What hardware do you need?
- Which part of the system do you want to test?
- What third-party software do you need (don't forget licenses)?
- What information should the mock-ups return (good and bad cases)?
- How can you compare the reference system's behavior with that of the working system?

I remember a server with corrupted network-attached storage (NAS). The NAS's disc controller was broken. A standby system stood beside the server. Because of the corrupted controller, the standby server couldn't be attached to the storage. So, I mounted disk space via the Network File System (NFS) from an old PC to the standby hardware. I changed the standby system to a test system with a shell script that changed hundreds of symbolic links that had previously linked to the corrupted storage.

If you can't build a test system, use as small steps as possible for your modifications and try to define small test environments for each step. Write down every step and how to get back afterward before the actual execution, so that you can go back in case of failure.

> Always plan a change from two sides: goal and fallback scenario. Uptime matters!

Step 6: Take the Business Sponsor and Architect Viewpoints

This step defines the new system's technical and business goals:

- What should the system look like when you're done?
- What service-level agreements (SLAs) should the new system guarantee?
- What business-level key performance indicators (KPIs) can you determine? If no KPIs have been defined, define some to prove that the retrofitted system is at least as powerful as the old one.
- What hardware and software technologies should the system use?
- What quality-of-service characteristics must the system achieve?

Maybe it seems a bit late to establish goals at this point, but what would you do with a high-end goal if the system isn't working anymore?

Often, in enterprise IT, technical details change constantly. As IT professionals, we sometimes forget that the business also changes over the years. Normally it doesn't change as fast as IT (at least not in most industry sectors I've seen). However, for long-running systems, the business context rarely stays the same during a software system's lifetime.

Having conducted steps 1 to 5, you have the chance to define how business could or should be conducted (and supported by the software system). This means you can redefine the system—what it should look like when you're done. These new business and system goals don't have to be defined as strictly as in a classic specification. They should only provide guidelines for the team so that everybody knows which way to go. The real goal will gradually become clearer while the project runs. As in agile development of a new system, the goal is to establish a vision of where to go eventually; intermediate goals and the final destination can change during the project.[9]

For example, suppose you have an old warehouse management system running on SCO Unix, written in ANSI C. One high-end goal could

PROS	CONS
• Software retrofit occurs while the software is running, so business can go on. • You need only a small budget to get started. • It provides a safe way for migration. • The total cost of ownership is less than for building a new system.	• The development effort might be higher than that for building a new system (owing to development of interim solutions that will be deleted later). • It requires additional roles. For example, the project team needs at least one system steward (for the old parts) and a future visionary (for the new parts).

FIGURE 4. Software retrofit's pros and cons.

be to integrate your e-commerce solution so that ordered goods will be delivered just in time. For this purpose, you want to change the old system into a Java application on a Linux cluster (whose SLA guarantees high availability).

Step 7: Turn into a Project Manager and Developer

Here, you start changing the system. Define a low-level goal, and perform project planning for it.

For example, one small goal could be for other applications to be able to access the data. In a new system, this step might be small, but what if the server isn't on the network or is in a special subnet? I've found servers in which changing an IP address required recompiling the OS kernel.

Always plan the change from two sides: define what should be done to achieve the goal, and describe the fallback scenario (remember, you're dealing with a high-availability system):

- What would you like to change?
- How must this part interact with the old system?
- What must you prepare for going live (for example, special tests)?
- What could go wrong? How do you recover from errors?
- When and how do you sunset the replaced parts?

For a case such as the warehouse example, it/could be okay to change just the IP address during a lunch break because compiling the kernel would take some time. On the other hand, you should save the old kernel so that you can use it to restart the system without disrupting business if the kernel compilation goes wrong. After finishing, go back to step 6 and rethink what to change next.

The system will become increasingly easier to handle. So, after you've planned and executed several low-level retrofitting steps, the development process could continue in a normal forward-engineering way.

Because you normally don't work on such a project every day, a complete retrofit will likely take some time. I've worked on projects in which a retrofit took six years, although the actual time was much less. Over that period, we had more than 250 installations on or modifications to the system. Sometimes there were four or five a week; sometimes there were none.

I can't say now how often we went through the seven steps, but certainly more than 50 times. Admittedly, a six-year-old system is considered outdated in IT—but remember the system you started with. Again, uptime matters. The main issue and cost driver isn't the time developers spend but the time when the system and therefore production are at a standstill.

Figure 4 summarizes software retrofit's pros and cons. I strongly believe that software retrofit will become increasingly popular because yesterday's new software will become tomorrow's legacy software.[10] Don't be afraid—software retrofit is an interesting journey from old to new technologies. The steps and supporting techniques I shared here are applicable beyond the logistics industry. When succeeding with a retrofit, you'll be the customer's hero because you were tough enough to manage a task that nobody else dared take on. And playing the detective is also fun.

References

1. M. Eric and M. Stevanović, "Comparative Characteristics of TQM and Reengineering," *Proc. 2nd Int'l Quality Conf.*, 2008; www.cqm.rs/2008/pdf/2/29.pdf.

2. "Software Archaeology with Dave Thomas," *Software Eng. Radio*, episode 148, 2 Nov. 2009; www.se-radio.net/2009/11/episode-148-software-archaeology-with-dave-thomas.
3. F. Torres, "Context Is King: What's Your Software's Operating Range?," *IEEE Software*, vol. 32, no. 5, 2015, pp. 9–12; www.computer.org/csdl/mags/so/2015/05/mso2015050009-abs.html.
4. G. Robles, J.M. Gonzalez-Barahona, and J.J. Merelo, "Beyond Source Code: The Importance of Other Artifacts in Software Development," *J. Systems and Software*, vol. 79, no. 9, 2006, pp. 1233–1248.
5. A. Binstock, "Mind Maps: The Poor Man's Design Tool," *Dr. Dobbs*, 2 Oct. 2012; www.drdobbs.com/tools/mind-maps-the-poor-mans-design-tool/240008292.
6. S.W. Ambler, "Agile Legacy System Analysis and Integration Modeling," *Agile Modeling*, 2012; www.agilemodeling.com/essays/agileLegacyIntegrationModeling.htm.
7. C. Hadnagy, *Social Engineering: The Art of Human Hacking*, John Wiley & Sons, 2010.
8. C. Alberts and A. Dorofee, "Mission Risk Diagnostic Method Description," tech. note CMU/SEI-2012-TN-005, Software Eng. Inst., Carnegie Mellon Univ., Feb. 2012; http://resources.sei.cmu.edu/asset_files/TechnicalNote/2012_004_001_15431.pdf.
9. M. Fowler, "Continuous Integration," 1 May 2006; http://martinfowler.com/articles/continuousIntegration.html#TestInACloneOfTheProductionEnvironment.
10. R. Hopkins and K. Jenkins, *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*, IBM Press, 2008.

THOMAS RONZON is a project manager and senior software developer at w3logistics AG. Contact him at ronzon@w3logistics.de.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.

ACM - IEEE CS ECKERT-MAUCHLY AWARD

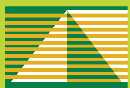
Call for Award Nominations • Deadline: 30 March 2016 • www.computer.org/awards • awards.acm.org

ACM and the IEEE Computer Society co-sponsor the **Eckert-Mauchly Award**, which was initiated in 1979. The award is known as the **computer architecture community's most prestigious award**.

The award recognizes outstanding contributions to computer and digital systems architecture. **It comes with a certificate and a $5,000 prize**.

The award was named for John Presper Eckert and John William Mauchly, who collaborated on the design and construction of the Electronic Numerical Integrator and Computer (ENIAC), the first large-scale electronic computing machine, which was completed in 1947.

TO BE PRESENTED AT



ISCA 2016
The 43rd International Symposium on Computer Architecture

Nomination Guidelines:
• Open to all. Anyone may nominate. • Self-nominations are not accepted. • This award requires 3 endorsements.

Questions? Write to IEEE Computer Society Awards Administrator at **awards@computer.org** or the ACM Awards Committee Liaison at **acm-awards@acm.org**