The Software Principle of Locality

By Jamie Dobson

Abstract

The principle of locality in software development states that: the person working closely, in terms of both space and time, to an artefact is the most qualified person to remove defects associated with it. This paper examines breaches to the principle and how they affect productivity on a software project. The paper also examines techniques that, consciously or not, respect the principle and very positively effect productivity on a software project.

Introduction

In this paper I am going to talk about the Principle of Locality in terms of software engineering. I want to explore and examine what happens when the principle is breached and when it is honoured. I will first define the principle of locality. Then I will relate the principle to two key problems in software development: defects and hand-over. I will conclude with some general lessons that I hope will help readers to improve their own software processes.

The Principle of Locality Defined

In physics the principle of locality states that distant objects cannot have influence over each other. In computer science it states that if a memory location was referenced then it will be referenced again soon (temporal locality), or a location in the neighbourhood will be referenced (spatial locality).

In regards to software engineering the Principle of Locality, also called the principle for the purpose of this paper, states that the person or team working closest, in both space and time, to an artefact is the most qualified to remove trivial problems associated with it.

It seems rudimentary; however, breaches of this simple principle can be found easily both experientially and in the software body of literature. In 2002 Gilb reported a half day inspection revealing 19 logic errors per page in a 40,000 page specification [1]. When sampling specifications like this, teams are actually obeying the principle of locality. However, Gilb goes on to give an example of a team who did a similar review three years after the original specification was complete and development was under way. This time the team found 150 major defects per page of the specification. This second example is clearly a breach of the principle and it cost that team 2 years to rectify the problems found in the specification.

In September 1999 the Mars Climate Orbiter [2] burnt up entering the atmosphere of Mars. The root cause, as stated in the report into the disaster was:

“Failure to use metric units in the coding of a ground software file, ‘Small Forces,’ used in trajectory models”

This came about from two modules not working well together. The modules modelled the spacecraft velocity changes differently. The words “integration” and “test” spring to mind. This is a great example of a defect been found in beta instead of system testing. In this case, the defect proved costly indeed. One hundred and twenty five million dollars according to CNN [3].

CNN also report Edward Wieler as writing “The problem here was not the error, it was the failure of NASA's systems engineering, and the checks and balances in our processes to detect the error. That's why we lost the spacecraft". NASA breached the principle of locality. They did the right tests at the wrong time.

Defects and Handover

Defects are when things don't work and, as stated earlier, they are easier to remove when the principle of locality isn't breached. During hand-over moments waste comes from, amongst other things, the loss of tacit knowledge. When something is handed over the principle of locality is breached. Thus, not only is tacit knowledge lost, but the person most qualified to remove trivial problems is no longer working on the artefact.

Defects & Multi Tasking

When a human being has to switch from one thing to another they lose the ability to concentrate. When a tester finds a bug and shouts to a developer, "hey! This login thing doesn't work" the developer has to stop what they are doing and, at least, listen to the shout.

Rubinstein et al [4] show quite categorically that not only does multi-tasking waste time but the switch time is actually a function of the complexity of the tasks at hand. I.e. the more complex the task the longer the switch. Most programming tasks, by their nature, are complicated.

Defects represent an insidious type of distraction since most defects can be prevented. Here is a real world example taken from my project notes [5] that highlights the destructive nature of finding bugs at the wrong time. This case study's time span is over a morning. However, it still highlights the extravagant waste of time associated with breaching the principle of locality.

Case Study Defects

Bob is a developer on a team of 10. The team includes three testers and seven developers. Richard is a tester. At 9 am Bob introduces a bug into the system. The bug comes from the incorrect resizing of a list of customers. Bob's crude manual testing never revealed the problem because the test he used never added more customers than the list's initial size.

Bob checks the code in at 9.05, the system is built properly and deployed by 9.15. Richard starts to execute his acceptance tests against the new version of the system. His tests find nothing. Therefore, he grabs a celebratory cup of coffee and returns to do some manual exploratory testing.

When Richard tries to add his tenth new customer the system breaks. Richard does a quick shout out for all developers who have worked on anything to do with the customer objects or tables. All seven developers have. All seven are now disturbed by this interruption.

Richard opens up his acceptance tests and now tries to re-produce the bug. It takes him half an hour to rule out the database. Another hour to rule out the web-client. Finally, after three hours, he pinpoints the problem to somewhere in the domain model.

Richard now asks the team, who has touched the customer domain model? This time Bob says he did, this morning. He checked it in around about 9 O'clock. Bob and Richard now sit down and begin to look at Richard's acceptance test. Richard needs to bring Bob back up to speed with the suite, this takes 30 minutes. They decide to break for lunch because it is now 12.45.

When the guys return from lunch at 13.15 Bob re-deploys the application to the web-container and attaches a debugger. He has three ideas where the problem may lie and asks Richard to manipulate the GUI in the same way he had done earlier. The code breaks and eventually Bob tracks the problem to a single method that handles the list re-sizing.

They fix the code but Richard insists that Bob writes a unit test. Bob does, it consists of 10 lines of code and took him 30 seconds to write. They check their code in and everything works. They finish at 2.15.

Bob breached the principle of locality. He was the most qualified to write the unit test and reveal the problem in his own code. In the end 30 developer seconds became the time Bob was involved, 1 hour, added to the time Richard was involved, 5 hours. This is 720 times longer than was necessary (and this

number does not include the interruptions to the other team members). However, the problem is compounded because Richard was taken off his other activities for 5 hours. So, essentially, those 5 hours are lost. Richard now has a backlog of 5 hours work he needs to get through. One has to hope, for Richard's sake, that some slack was built into the plan.

In the same way NASA's procedures let them down, Richard's and Bob's are letting them down too. A simple project rule stating that each developer must write a unit test per public method would have saved a lot of problems here. In general, this is not a bad rule to enforce. Using Beck's Test Driven Development [6] is an even better one.

Defects & Maintenance

In the example of Richard and Bob we saw that skipping a unit test could cost 720 times more time to find and fix the problem. However, this cost becomes astronomical, no pun intended NASA, when the principle is breached much later. For example, during beta testing or when the system is in release.

Jerry Weinberg insightfully refers to maintenance of software systems as 'software brain surgery' [7]. His labelling highlights how hard to remove, dangerous, and costly, defects found later rather than sooner are.

Weinberg points out the chance of introducing an error has an inverse relationship to the amount of code that is altered. I.e. The less code touched, the higher the probability that something would have been broken. This is because small changes tend to attract less attention. Therefore, people are more sloppy.

Arguably, you could tighten up your maintenance procedures. You could supplement your maintenance procedures with a suite of automatic regression tests. And these things would help. But, tinkering with something as delicate as a live computer system is dangerous. Weinberg puts it well. "Just open up my skull, Doc, and remove that one little habit. And please do a quick and dirty job—it's only a small change! Just a little maintenance job!" [7]. There is no such thing as a little maintenance job. As a manager, you let defects slip downstream at your peril. When there is a choice, bugs should be fixed immediately.

The maintenance problem is compounded by the way companies organise their teams. Typically companies, including those who develop with agile methodologies, have a separate maintenance team. The team is usually under staffed with junior developers. So, not only are the maintainers a long way, both spatially and temporally, from the production of the code, they are also not skilled enough to make these brain surgery like changes. This is one reason why maintenance is the most expensive phase of any system development. Precision work is batched, then queued, and then thrown over the wall. Throwing anything over the wall is the ultimate in breaching the principle of locality.

There are many reasons this happens. The one is that managers cut corners and allow, deliberately, bugs to remain in the system. They do this in order to meet their own deadlines and come out of the project 'successfully'. Of course, these same managers could really be successful if they were smarter about how their system is developed. Tom Gilb again [1]:

"Management frequently allows extremely weak specifications to go unchecked into costly projects processes. They are obviously not managing properly."

You can replace "extremely weak specification" above with "poor code", "buggy software", "no tests", or anything that suits. If this is happening in your teams the conclusion is the same: your managers are not managing, and they are certainly not taking responsibility. Only in software engineering is such a slap-dash approach allowed and tolerated.

Adding Features During Maintenance

According to Glass maintenance is part of the software solution [8], not a problem unto itself. This is because features are added during maintenance. I would tend to agree with Glass; however, I would add that the principle still applies. The development team should become the maintenance team. It is as simple

as that. Not only does this not breach the principle but it does something else: it forces the team to take responsibility.

This following example is horribly too common. The team cannot be bothered to fix a certain bug, or the deadline is looming. A normal reaction is to say “leave it to maintenance”. Even though everyone knows the best time to fix a defect is now. Would this attitude be taken if the same people had to remove the bug later? I doubt it very much indeed.

There are many urban legends about soldiers who build bridges. When the bridge is complete they have to stand under it for a length of time as the first 100 battalions march over head. If the bridge breaks, the soldiers get squashed. This is a pretty good way to ensure the quality of the bridge. I always insist that the team that develops the code must maintain it. And if they leave too many bugs in the system it is they who get squashed under the weight of their own laziness or stupidity. Not some poor maintenance person who had nothing to do with the development of the system. The principle of locality can be re-phrased in a cruder manner: no person should have to clean up the mess of another.

Defects and Queues & Load

When a system gets put under too much strain it does not perform at 100% efficiency, despite what common management logic might dictate. Rather, it collapses. More traffic can shift between two cities when the connection, the motorway, is not overloaded. Now imagine a stretch of motorway running for 10 kilometres. At four points, the 2, 4, 6 and 8 kilometre marks there are off ramps. The off ramps take the drivers all the way back to the beginning of the motorway.

The first ramp is developer testing. The second ramp is acceptance testing. The third ramp is system testing. The fourth is beta-testing. The cars getting off at these different points and re-joining the motorway are defects. The amount of defects will have a direct relationship on the throughput (the amount of cars leaving the motorway per hour). When the motorway gets too full horrible things happen. The emergency services can’t clear away accidents as effectively; drivers get stressed and get off at the wrong exit by accident (they become a defect); and the motorway managers become very dis-heartened because they know that this motorway is capable of so much more. If only those stupid drivers would stop getting off at those exits… If only we could produce software with less bugs…

Work must flow throughout the pipeline. The queue of work, typically captured in a Scrum backlog, has to be loaded enough to keep the team busy for an iteration. The problems with load occur when work, in the form of defects, starts to come back. The backlog gets stacked up more. Developers, feeling they can fix that defect easily, take on more work. Congestion occurs and all the problems associated with an overloaded network kick in. Unfortunately there are other human problems associated with load such as irritability, lack of motivation, depressions, and anxiety, all of which come from the stress of a team having too much work to do.

Handover

Handover is really to do with the loss of tacit knowledge. We have seen already, with defects, that the principle of locality is costly. Bugs take longer to locate later partly due to tacit knowledge being lost during handover. Thus, what takes the original developer a minute to find and a minute to fix is likely to take the new guy an hour, or day, to find and fix.

One possible solution to the handover problem would be to just stop doing it. Don’t handover anything, ever… You avoid breaching the principle of locality by never allowing anything to be done non-locally. I.e. the person who analyses also develops. And the developer tests. And the developer is later the end user…

I make this last point to highlight the ridiculousness of this suggestion. However, there is some merit in actively only handing over what is necessary. Or in taking more care during the hand over moments. The blockages to successful handover are: knowledge and skill. For example, a developer cannot always carry out performance tests because they are not skilled in this area. Thus, we can attack the two main problems of handover with better skills, better handover, and not handing over anything unnecessarily.

Hand Over & Generalising Specialists

A generalising specialist is a term coined by Ambler [9]. A generalising specialist is an individual who is multi-skilled. For example, I am a good Java programmer, a good tester, and an alright analyst. This spread of skills enables me to contribute to many areas of a software project.

The power of the generalising specialist is two fold. Firstly, hand over can be reduced because, for example, a developer may go on to test their code. Later the same person may go onto carry out performance tests too. This one person is working locally with their own system so detecting and removing defects is done cost effectively.

Secondly, there are always different tasks on a software project open at a time. To use the same example, a developer may be able to switch to testing once all the development tasks are complete. This will help to keep work flowing, and hopefully not re-work flowing back. When this happens a team is less likely to throw something over the wall to the maintenance team since they have the skills, and hopefully the confidence and the commitment, to use them.

Note on Refactoring

Refactoring, as defined by Fowler [10] is “improving the design of the code after it has been written” . Rework is not Refactoring. The terms rework and refactoring are used by managers and developers who are too lazy to do the job properly. Or by people who are ignorant to how feedback and design work.

Let’s stick to our automobile theme. A car rolls of the production line and later has to be brought back because the anti-lock braking system doesn’t work. This is rework.

A clay model of a car is created and shown to a group of customers. The customers complain that they don’t like the headlights. Based on this feedback the designers remould the clay with a different design. This is refactoring.

In general, when anyone uses throw-away language like “oh, don’t worry, we’ll refactor it later” what they usually mean is “don’t worry, we’ll rework it later”. And, what that sentence means is “I don’t give a shit. This defect will not affect my success criteria but the fixing of it may. I don’t care if this will cost the company an exponential amount to fix later, as long as I am ok, that’s all that matters”.

Hand-Over & Collaborative Work

The final thing to help not break the principle of locality is to get people working together and handing stuff over more effectively. Good examples of this are modelling together, joint requirements sessions, writing tests together and pair programming. By taking advantage of this locality, team knowledge is increased and later, when hand-over between members take place, it should be more effective.

Cockburn calls this Osmotic communication [14] and suggests, rightly in my opinion, that it should be encouraged by the design of work environments that support this. Cockburn suggests that closeness of the individuals has a direct effect on their ability to pick up on the information flying around.

Cockburn does, however, mention that different types of negative chatter may affect the team. This especially comes when people, based on their speciality, form cliques. His solution, naturally, is to make sure that the seating is arranged to avoid the clique! This clique forming based on speciality is of course another reason to use generalising specialists. Even if, for example, an analyst does not program on the project, if they have programmed before they are much more likely to understand why a developer does what they do.

Managers and leads must also try to understand emotional contagion. Teams of humans, when left to their selves, will gravitate to a negative state of mind [11]. That is because fear is, in the short term, a life saver. When one team member is afraid, this will spread like a bad cold spreads. Senior managers very often don’t

understand why I arrange so many teams lunches; South Park afternoons; lectures; days out; and evening drinks. I do this to counter the emotional contagion of negative emotions.

That my senior managers don't get this is a damning indictment of the way we train our managers. It can be hard for them getting educated in the theory of constraints, queuing theory, motivation, emotional intelligence, planning, tracking, team structure & dynamics by a programmer.

Conclusion

I have borrowed the principle of locality from physics and computer science and tried to use it as a framework for understanding effective ways to develop software. I have also shown examples of breaches of the principle and how disastrous they can be to the software development effort.

As a software practitioner I am always looking for ways to decentralise control. Allowing the right people at the right time to do their jobs effectively has helped me, on many occasions, successfully deliver software and teams.

Many defects found by customers, testers, specialist performance tester teams, system integrators, and maintenance teams can be found with unit testing, code review, analysis review or any other downstream cheaper, more local, technique. Breaching the principle by, what McConnell calls, short changing quality assurance [12] is very short sighted.

Therefore, some general advice is:

1. Don't skip important quality activities
2. Create teams of people who can do multiple things, but don't let them do multiple things at the same time
3. Allow people to communicate locally by co-locating them, but manage rigorously to avoid negative emotional contagions and clique forming

I once described performance testing on an agile project as "theoretically, really easy. Change the model, get everyone motivated, and use a few XP practices. However, the hard part of anything agile is not knowing the path, but walking it. This takes dedication, energy, and the design of good processes that suit the context".

The same goes for the principle of locality. It's not difficult to grasp, it's just difficult to enforce.

Discipline and effort must be put into software quality as early as possible. No matter how hard it may seem the return on this investment will be high. This honouring of the principle of locality is a key factor to project success.

References

[1] Tom Gilb. The 10 Most Powerful Principles for Quality in Software and Software Organizations, CROSSTALK The Journal of Defense Software Engineering. 2002. http://www.stsc.hill.af.mil/CrossTalk/2002/11/gilb.pdf

[2] Mars Climate Orbiter Mishap Investigation Board, Phase I Report, Nasa. 1999.

[3] NASA's metric confusion caused Mars orbiter loss. Cable News Network. 1999. http://www.cnn.com/TECH/space/9909/30/mars.metric/

[4] Rubinstein et al. Executive Control of Cognitive Processes in Task Switching. Journal of Experimental Psychology: Human Perception and Performance. 2001.

[5] Dobson. Project Notes of Team CMS at Logica CMG. 2006/2007.

[6] Beck. Test-Driven-Development. Addison-Wesley. 2002.

[7] Weinberg. Innocent but Dangerous Language. Blog Entry, March 2007.
http://secretsofconsulting.blogspot.com/

[8] Glass. Software Maintenance is a Solution, Not a Problem (An Excerpt from Software Conflict 2.0) .
2006. http://www.developerdotstar.com/mag/articles/maintenance_solution.html

[9] Ambler. Generalizing Specialists: Improving Your IT Career Skills. Scott W, Ambler. 2003-2007.
http://www.agilemodeling.com/essays/generalizingSpecialists.htm

[10] Fowler. Refactoring. Addison-Wesley. 1999.

[11] Sigal G. Barsade, The Ripple Effect: Emotional Contagion In Groups.
Yale School of Management Working Papers ysm167, Yale School of Management. 2001.

[12] McConnell. Rapid Development: Taming Wild Software Schedules. Microsoft Press. 1996.

[13] Dobson, Performance Testing on an Agile Project. 2006.

[14] Cockburn. ASD book extract: "Communicating, cooperating teams". Cockburn. 2006. http://alista