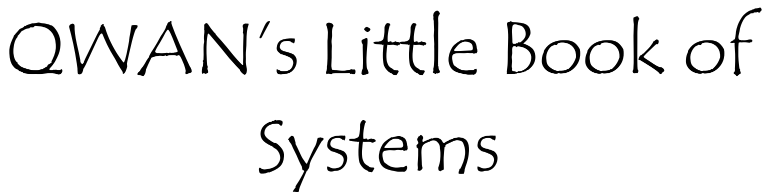

powered by Quality Without a Name

In steering your development process you adopt a simple set of practices that give you the feeling that you want. As development continues, you are constantly aware of which practices enhance and which practices detract from your goal. Each practice is an experiment, to be followed until proven inadequate.

from: Kent Beck, eXtreme Programming Explained 1st ed., p. 28

© 2017 QWAN – Quality Without a Name
www.qwan.eu

edition: June 2017

QWAN's little book of systems - Promise is debt

At QWAN, we like to think both big and small at the same time. We practice things like continuous delivery and test-driven development, because they make us go fast now, and help to keep our pace sustainable in the future.

Systems thinking with diagrams of effects helps us make sense of the practices we explore and the challenges we face. It also helps us build shared understanding, within our teams and with our customers.

Workshop in a box

For this little book of systems we have adapted a story about technical debt we have used to teach systems thinking. We have added the steps we use to create diagrams of effects from someone's story, so you can create your own diagrams for your stories. It's like a workshop in a box.

The story we use is called Promise is Debt - promising too much to customers leads to technical debt, which makes it harder… to fullfil promises to customers.

Willem van den Ende, Marc Evers & Rob Westgeest

© 2008-2017 QWAN – Quality Without a Name

Revised edition August 2017

Introduction

Did you ever. . .

. . . feel you have no grip on the situation?

. . . try to solve problems but the team seems to be stuck in a vicious circle?

. . . put out fire after fire, where putting out one fire seems to ignite the next one?

We tell the story of a team making promises to their customer in such a way that it becomes almost impossible to fulfil them. . . This creates a downward spiral of making promises, breaking promises, and making new promises to compensate their customers' disappointment. In the end, both the team and the customers lose trust and the team loses its credibility.

We will show what the underlying causes are and how you can really tackle the situation, using storytelling and systems thinking with diagrams of effects.

Systems thinking sees a system in terms of variables that influence each other. Systems thinking focuses on the interdependence of parts instead of linear cause-effect relations. It is about seeing the whole, and looking at the dynamics and change, with feedback playing an essential role.

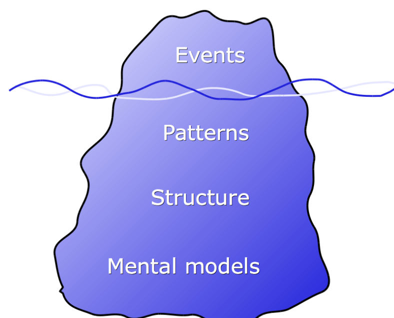

Figure 1: We only see events on the surface, the rest is hidden from us.

When we react, we tend to see only *events* - things that are happening right now or that have happened recently. It is hard to make sense of something when we only look at events. If we do that, we have knee-jerk responses and fight fires, instead of looking at underlying patterns, structures, and mental models.

Our mental models are very influential in what happens around us, but often hard to identify - the fish is always the last to see the water. Understand and (slowly. . .) change your mental models, and change the world.

Systems thinking with diagrams of effects helps to make mental models of different stakeholders more explicit and to see not-so-obvious effects and self-reinforcing loops. This makes it easier to find effective interventions.

Disclaimer: it is not necessarily easy to do though. Understanding what you did not understand before by surfacing tacit knowledge can be painful.

A process to tell a story and get a diagram

If you are in a difficult situation it is hard to make heads or tails from it. Telling a story helps, making a visual with the story encourages you to find more details, to ask more questions. Once you understand the situation better, you can come up with more effective interventions. The diagram(s) you and your group come up with can sometimes be useful in explaining the situation and your proposed solutions to others. Be aware though that the story and the diagrams only reflect the understanding of those who were in the room at the time of making it, so be open for suggestions and other views when presenting the results.

These steps do not have to be followed linearly, as you will see in our story below. We do recommend brainstorming variables before moving on to the next step, even if you end up discarding most of them later on, because it is easy to get stuck in a particular definition of 'the problem' early on.

1. *Tell the story*. Ask questions, determine scope. It helps if you do this in a group, one person tells the story, others listen and make notes, and in step 2:
2. *Collect variables*. We are looking for behaviour that is observable or measurable, and changes over time. We don't need to quantify everything. We often do this

on post-it notes or index cards. To prevent group think, it can help to have individuals collect variables on their own first, and then share with the group.

3. *Determine cause effect relations*. Find relations between the variables. Does one affect the other, is it more or less? You can use pieces of string to connect the notes from step 2, or draw arrows on the notes. Usually stuck on a flipchart for easy presentation.
4. *Look for loops*. Find loops that are self-reinforcing (vicious or virtuous cyles) or stabilizing.
5. *Simplify*. Remove unrelated variables. If necessary, split up the diagram.
6. *Identify possible interventions.* We see if we can change how variables influence each other or try to find new variables to influence 'bad' variables.
7. *Tell the story to someone else.* Use the diagram(s) as support. Telling it again will give you new insights and you will get feedback on your understanding of the situation, plus suggestions on possible solutions from your audience.

Contraindications

This is not a bullet-proof process or a diagramming technique magically suited to all situations. It has sometimes happened to us that we could not even identify useful variables. We have had situations where we were looking at the wrong problem, or did not have enough information. It might also be that you find yourself in a chaotic situation, or surrounded by unknowable unkowns, where there is no discernable relation between cause and effect just yet. For these situations, the Cynefin framework or the Satir Change Model can help to determine where you are.

On to step one

Remember, step one was:

> *Tell the story*. Ask questions, determine scope. It helps if you do this in a group, one person tells the story, others listen and make notes.

The story

Once upon a time ... there was a small IT organisation, consisting of a few developers (Paul, Mary, and Martin) and Jeff, the group's manager. Jeff does marketing and sales as well.

They have been working for over a year now on 'their' product, an Innovative Web System (IWS). They already have three customers, represented by Angela, Fred, and Brian. The IWS product is partially generic, to keep maintenance costs down. They provide some custom-built features for each customer, because the team values 'customer intimacy' and likes to use feedback to improve IWS further and make it attractive for more customers.

All three customers are enthusiastic: they see the system's possibilities, although it does not meet all their requirements yet. Each customer still needs specific functionality, but they are confident that the team is going to deliver it.

Because the team works with monthly releases, the customers continuously see progress. Some releases show more progress than others, but the product as a whole evolves steadily.

What is the matter?

The developers have just finished doing a release. They have delivered almost everything they had planned. A new plan has been created for the next release. As a team, they have a working agreement that for the next release, they won't promises more features than they actually delivered in the previous release.

During the most recent planning session, Jeff tried to persuade the team into promising an extra feature, but the team held firm. "Let's just do what is realistic. If we go faster than expected, we can always add some extra features. That is better than promising too much and then failing to deliver," according to Martin.

In the morning of the second day after they started working on the new release, Jeff enters the development room: "Yesterday, I have talked to Ronald again, and I finally managed to convince him! Our fourth customer! I had to promise him feature *FR53i*

however. He insisted that we build it specifically for his organisation and deliver it this release."

"But we have already planned enough for this release and this *FR53i* feature is a lot of work" objects Mary.

"This customer is of strategic importance! Ronald is someone who will start selling the system to others once he is convinced. That will get us a lot of extra customers and sales. We just have to work a little harder this release, and then everything will work out!"

"Then we will have to cut corners, I strongly doubt whether the code quality will remain acceptable," says Paul reluctantly, "I'm afraid we will experience defects that will be hard to track down."

"No problem, you can just refactor a bit extra during the next release and everything will be all right. I see you all understand the importance of going the extra mile, so then it's a deal!" Jeff quickly leaves for his office.

The developers have their doubts, but they have also become enthusiastic about getting a fourth customer on board. Ronald is a difficult person to persuade, having him on board is quite a big thing. So everyone works like mad to build the extra feature on time. Towards the end of the release, pressure and overtime increase. . . .

Step two: identify variables

Variables are properties of the system that we can observe or measure (circles). We denote variables in the story by making them **bold**.

Sitting together with our post-its, we find from the story above:

- **Customer satisfaction** (measurable)
- **Features promised** (measurable)
- **Customer expectations** (observable)
- **Features planned** (measureable)

When discussing variables (e.g. "what happens if **Customer satisfaction** takes a nosedive?"), you'll inevitably find yourself thinking about relations. Try not to write down the relations at once.

Step three: find relations

Causality can work in the same or in the opposite direction. If it works in the opposite direction, an increase in one variable means a decrease in another variable. In the diagram below we indicate that by drawing a dot on the edge; you can also use a minus sign.

What relations do we have?

- When **customer satisfaction** is low, Jeff decides to **promise features**.
- The promises raise **the customer's expectations**.
- To meet these expectations, the number of **features promised** increases,
- and when that happens, **customer expectations** also increase.

So we have two causalility relations that work in the same direction, and one that works in the opposte direction - when **customer satisfaction** *de*creases, the **number of features** promised *in*creases.

The next diagram shows the variables and relations we have identified so far. Things that we can measure are shown by circles, those things we can observe but not measure are drawn as clouds, in this case *customer expectations*. The edges show causality relations.

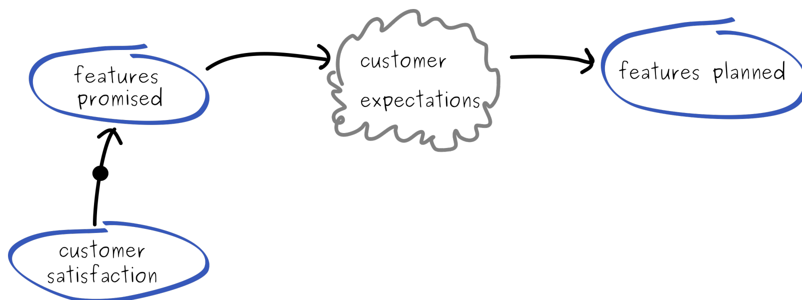

Figure 2: Customer satisfaction vs planned features

So the diagram we have so far makes us wonder, what happens when more features are planned? Let the story continue...

Iterate to step one, the story continues

The pizzas that Jeff brings in at evenings compensate for much. The feeling of doing something important, pleasing a customer, and being part of a close team gives a kick.

They do cut corners and they're not satisfied with the quality of their work. Fortunately, they will be able to make up for it (like adding all those missing unit tests) during the next release, when things have hopefully quieted down a bit.

> We add **corners cut** to our list of variables. Working in the evenings means the **workload** is high.

Mission accomplished...

The release is successful, all planned features as well as feature *FR53i* have been completed. Everyone is tired and stressed out. Jeff drops by in the development room: "Well done! I told you so: you're able to do more than you think. I'm proud of you all!"

At the next planning meeting, the team has trouble restricting the number of features to be scheduled. Jeff would like to schedule as much work as they just delivered.

Martin sticks to his guns: "The high velocity is distorted: although we completed more, we didn't do it in a sustainable way. We have put in a lot of overtime, skipped unit testing and refactoring, and didn't do any code reviews. I don't know how long we can keep going like this. Moreover, Jeff, you promised us extra time during this release to catch up with all the corners we've cut. We can probably do a bit more, but let's be sensible and work in a way we can sustain over time."

Jeff gives in, reluctantly.

> We add **corners cut** to the list of variables, as well as **pressure to deliver**. We considered adding **velocity** but could not do much with it at this point.

The next release

Work starts slowly. The team requires time to recover and is hardly productive during the first week. They try to repair some of the **corners cut**, but they're just too tired to think clearly and don't accomplish much. In the second week, with some pressure from Jeff, they start working on the planned features. Slowly they get up to speed.

> At this point we could add **fatigue** but believed later that **workload** covered it sufficiently.

Then the different customers start reporting **defects**. Over the last year, they had only 2 or 3 defects in each release, now they suddenly have 4 defects in a week. They also receive an angry e-mail from Ronald, stating his annoyance about a nasty bug he has found. The team immediately starts solving the defects, to prevent losing Ronald as their customer.

> We add **defects** to our list of variables. It increases the more **corners are cut**, and decreases **customer satisfaction**
>
> Because of the large **workload** and the **pressure** that Jeff puts on the team, the developers are more inclined to **cut corners** and choose quick and dirty solutions, thinking they'll catch up later. This causes more and more technical debt as well as more defects. More defects lead to lower customer satisfaction.
>
> We add **design debt** to the list of variables. It does not influence anything yet, but we feel there is more to it, which we will investigate later. The next diagram shows the variables and cause-effect relations we have so far.

By the end of the release, they are significantly behind. They still try to complete as much as possible, but they eventually deliver only half of what they had planned. Their customers are slightly disappointed.

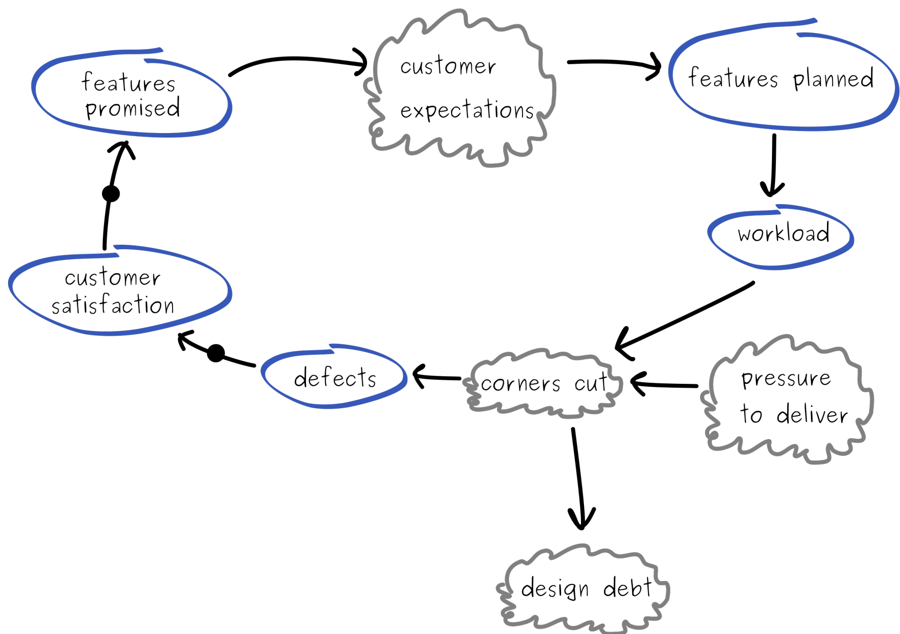

Figure 3: Cutting corners reduces customer satisfaction.

Better luck next time

Ronald begins to openly express his doubts about the system. Jeff pays him a visit in an attempt to placate him. "We just had some bad luck this time, the developers had a touch of flu at the start of the release, and things just went a bit slowly. I've talked to them sternly, so trust me, they will do a better job next time. We will make sure the next release also includes feature *8RTv91x*, with the *HH05* extension."

Ronald decides to give them another chance. Jeff visits the other customers as well and assures them that the *defects* and the non-delivery of features were only incidents. Next time, everything will go as promised.

Jeff pressures the team to complete *8RTv91x* in the next release – failure is not an option! He schedules a series of features all labelled as "essential". The team members notice that the amount of work is much more than the velocity of the previous release; it is even more than their velocity from the time they didn't have all these problems. They submit: they know in their hearts they won't succeed, but believe they have no choice.

On the way up?

The team members notice that the new features take them longer. The quick and dirty solutions they used for the previous releases are a royal pain in the backside. It takes more and more time to understand their own code, to add unit tests, and to find causes of defects. Meanwhile, new defects keep on coming in, most of them in the new features they delivered recently. After Jeff's lecture, they primarily focus on feature *8RTv91x*. They manage to complete it, at the cost of other features, so much less then they promised.

This time, Ronald is partially satisfied: "I'm glad you have finished *8RTv91x*, but I expected the *x80y8* to be finished as well, that's what Jeff promised." The other customers start complaining. "You have underdelivered, again! It's as if bugs are the only thing you people deliver these days. . . " Fred sighs in frustration.

> We add **fulfilled promises** to the list of variables. Quite often we go over the story and find more compact names as we go (the previous edition of

> this story had **Probability that promises will be fullfilled** as name for this).

"We will schedule *x80y8* right now!" Jeff promises to Ronald. To the other customers he says: "I'll have a firm chat with the team, I completely agree with you, things can't go on like this. The next release will be all right, we will deliver *Xnrg-4.5.4* as well." He knows all three customers are dying for that feature.

Jeff calls the team together in a conference room: "We need to work hard to regain the trust of our customers. I know you can do it, don't disappoint me! Just leave out refactoring, we don't have time for that. I get the impression that testing doesn't really contribute to productivity either. If everyone just builds features, everything is going to be all right."

He proceeds: "Now that this is clear, here's the schedule for the next release, including *x80y8* and *Xnrg-4.5.4*. I'm sure you can do it, although it may look to you like it is more than what you did before, I'm sure you will be fine. Well, we've spent enough time in this meeting, let's get back to programming now, so that we can make our customers happy."

Or on the way down?

The same thing happens for this release: completing features costs more and more time because of technical debt. New defects keep on coming in, predominantly caused by hasty fixes to previous defects. The team slowly lose their motivation. They try to rush through the features, to prevent being blamed by Jeff. When the release is over, the velocity turns out to be much lower, they are even unable to agree on whether some features are finished or not.

> We add **time per fix or feature** to the list of variables. **Design debt** will very slowly increase this. Adding **motivation** is tempting, but we leave it out for now.

> See that we just skipped step 4 and went for step 5? - *Simplify. Remove unrelated variables. If necessary, split up the diagram*.

After the initial brainstorm, simplification is something that happens more or less organically. We have it as a step to remind ourselves to do only have variables that influence our story in a meaningful way.

When Ronald tries out the *x80y8* feature he has been dying for, it works, but only partially. And the part that is least important to him is functional.

"This is the limit!" he screams at Jeff, "No more IWS for me!" He announces his decision loud and clearly, to everyone who wants to hear it.

Figure 4: Design debt reduces customer satisfaction after a while.

After some time, the **design debt** starts having a noticeable effect on the **time per fix or feature**. The number of **fulfilled promises** within the

> time estimated decreases. Not delivering what you've promised causes lower **customer satisfaction**.
>
> Time for step 4: *Look for loops. Find loops that are self-reinforcing (vicious or virtuous cyles) or stabilizing*
>
> This system contains two self-reinforcing loops: promising **extra features** indirectly causes even lower **customer satisfaction**. The system is not stable. Eventually, customers and developers will leave.

At the coffee machine, Angela meets Mary. Mary looks tired. "It's not going well with IWS, in my opinion," says Angela. "Indeed" says Mary, "I'm sorry for how things are going."

"No problem, it won't bother us much longer," Angela replies, "We're looking around for a replacement system and we have identified two suitable candidates. I feel sorry for you, I've always liked collaborating with the developers."

"Well, I'm also finished with this," Mary says, "I had a talk at QXD yesterday. They have a job that suits me better, I'll start next month."

"Congratulations! Too bad for IWS, but I'm happy for you!"

Doomed to fail?

If we plot *work completed* against *time*, we get the *release burndown chart* shown below. The solid line indicates the amount of work remaining, the slope of the line is indicative for the velocity. The dotted line represents the expectations based on the initial velocity. If the solid line deviates from the dotted line, then this is ground for further investigation and possible interventions.

In the first release, the team delivers what is expected. After that, however, the team delivers less and less. What causes this? What can we do to finish all the work in the near future?

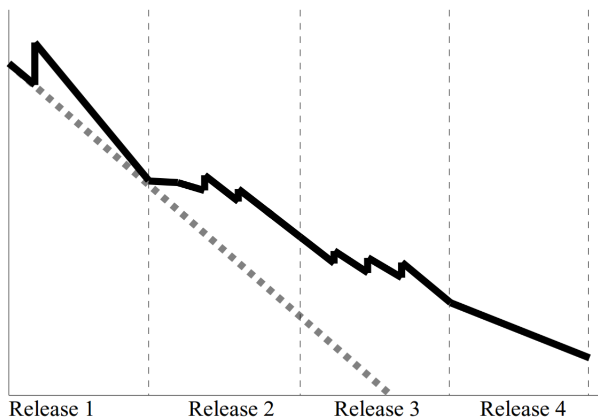

Figure 5: Features per release

The diagrams of effects give us insight into the underlying dynamics. How does this help us? More specifically, what does it tell us about possible interventions? We could promise less or more features, vary the amount of scheduled work, or vary the amount of pressure on the team. In terms of the model, this means changing the values of variables.

If we would only change the values of variables, but keep the loops, we don't touch the nature of the system and the system remains inherently unstable. We have seen

in practice that despite the instability, the system can continue to exist for quite some time:

- customers don't have a choice – at least, that's what they think – and give the team another chance, again and again,
- customers are afraid to speak up and and put up with this way of working for a long time,
- the product is not that important for the customer, so the impact of the problems is small,
- despite everything, the developers try to make the best of it. perhaps it would be wise to let things escalate early, so that clients get honest feedback about what is possible. But that goes against everyone's feeling of pride, professionalism, and craftsmanship.

Sooner or later however, things will go awry and the system will collapse: customers run away, people burn out, developers leave.

Identify possible interventions

Time for step 6. *Identify possible interventions.* We see if we can change how variables influence each other or try to find new variables to influence 'bad' variables.

The cause-effect relations we found between the different variables are not all "laws of nature" or carved in stone: some represent an implicit or explicit choice – a *management decision*. The relation between customer satisfaction and extra features promised is an example of this – if the customer satisfaction drops, it is up to the team and Jeff to decide if they want to promise extra features or not.

There are more places in the system where there is a choice:

- the amount of **pressure to deliver** that Jeff puts on the system; it is not sufficient to only intervene here, because the vicious circle remains intact
- the number of **features promised** when customer satisfaction changes: *Don't Overpromise*
- the number of extra **features planned** trying to meet **customer expectations**: *Limit Work in Process & Don't Overburden People & Process*
- the extent to which developers choose to **cut corners** when the **pressure to deliver** and the **workload** increase: *Developers Say No*

We have indicated these management decisions in the diagram below using squares. Each square represents a choice where the people involved choose explicitly if there is a positive or negative effect, or no effect at all.

Make sure not to overload the team. This is not easy once the system has started spiralling down the vicious circle: the team has to take a step back and lower their expectations. You know that you are going to disappoint one or more customers. It's better to do this consciously, instead of just letting it happen. You are going to have to take your medicine sometime. After that, you can make sure customer expectations remain realistic. You might lose a customer, but the alternative is much less attractive.

On the other hand, be careful with promising too little and exceeding expectations by far: this bears the risk that customers will expect you to always deliver much more than you promise...

Figure 6: Decisions remediate vicious cycles.

Another way to explore options for interventions is adding a variable and see how that might influence the system. Suppose the team does an experiment with pair programming. Pair programming often happens spontaneously when a defect needs to be fixed - developers seek each other out when they are stuck, even though it may take a while for someone to realise they are stuck.

Whether you add interventions or introduce new variables, be sure to choose a limited set, and consciously run any proposed change as an experiment for a set duration. Evaluate afterwards with all involved how it went, and what, if anything, you want to keep going forward. We find it significantly easier to get buy in for an honest experiment than a mandatory change.

Figure 7: An experiment with Pair programming may improve several variables

We chose **pair programming** as an example on purpose, because we have seen dramatic improvements in how long it takes to sustainably ship a new feature. It is not an easy change to try out, we usually introduce it after having run several other experiments successfully. Doing an experiment turned a major opponent of pair programming into one of its major proponents in the company. One never knows where support comes from. Note that we're not saying that **pair programming** is always the best way to do things, only that it may be very effective in this context.

Another way of intervening is trying to influence or stabilize the vicious loops so that they do not get out of control. We can work on technical debt explicitly, by scheduling time to work on refactoring and redesign. This affects the technical debt by reducing it. Depending on how bad the situation is, it might take some time before the pay-off becomes visible.

This intervention also affects the number of planned features: the more time we schedule to work on refactoring & redesign, the less time remains for working on features. This can be a tough trade-off to make, but we have seen it pay off in practice.

This also concludes step 7. *Tell the story to someone else*, which we've been doing all the time you were reading. Some of the interventions you've read about we found from audience suggestions when we presented this story.

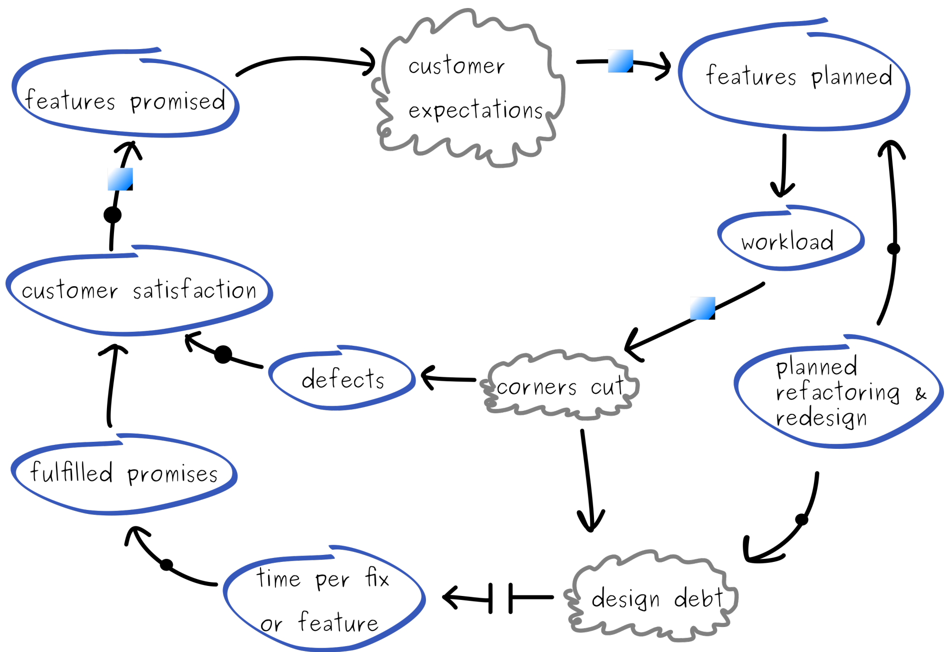

Figure 8: Planned refactoring & redesign stabilizes both loops

Conclusions

The “Promise is Debt” pattern, where someone overpromises to compensate for current problems, assuming they will catch up later, tends to defeat its purpose. Cutting corners appears attractive, but is counter-productive. The problem is that the effects are not immediately visible. Cause and effect are indirectly linked, separated in time, and influence each other mutually.

The combination of overpromising and cutting corners induces a vicious circle. The team slips into a destructive spiral of cutting more corners, delivering less and promising more.

We have observed this spiral in several different organizations. If it occurs, its causes are usually systemic and cannot be attributed to specific individuals. Looking for a scapegoat is pointless and will only make things worse.

Manipulation, in this case by Jeff, makes it difficult for people to say no or even to be aware of the fact that saying no is an option. Saying no should always be an option for every person involved. In fact, creating a culture where a grounded *no* at all levels is appreciated, is one of the most cost-effective interventions higher level management can make.

We hope our steps help you tell better stories, creating shared understanding that enables you and your co-workers to find lasting improvements.

By making your mental models and assumptions explicit and discussing them, you will see the available choices more clearly. This will let you identify those choices that make a structural improvement to the system dynamics.

Telling stories with diagrams of effects can make hard to pin down effects with delay and vicious circles visible and solvable. We wish you happy story telling!

References

Gerald M. Weinberg, *Quality Software Management*, volumes 1-4: *Systems Thinking, First Order Measurement, Congruent Action, Anticipating Change*

> *In depth application of systems thinking to all kinds of problems in software organisations. Republished on Leanpub as the Quality Software bundle.*

Gerald M. Weinberg: *More Secrets of Consulting, The Consultant's Tool Kit*

> *Introduction to some of Virgina Satir's tools.*

Peter M. Senge, *The Fifth Discipline: The Art & Practice of the Learning Organization*, 1994

> *Senge applies systems thinking to learning organisations. He discusses, among other things, causal loop diagrams and a number of archetypes – recurring systemic patterns.*

Donella H. Meadows, *Places to intervene in a system*, Whole Earth Magazine Winter 1997, www.developerdotstar.com/mag/articles/places_intervene_system.html

> *Essay providing an overview of different ways of intervening is a system.*

Donella H. Meadows, *Thinking in Systems* (2008)

> *Good introduction to system thinking from one of the pioneers in the field*

David J. Snowden, Mary E. Boone, *A Leader's Framework for Decision Making*, in: Harvard Business Review, November 2007

> *Introducing the Cynefin framework, which might help to determine what kind of systemic tools to use.*

Marc Evers, Nynke Fokma, Willem van den Ende, *Satir Change Model* www.satirworkshops.com/files/satirchangemodel.pdf

Two page summary of the Satir Change Model and some applications. Chaos is one of the stages in the change model.

Ward Cunningham, *OOPSLA '92 Experience Report - The WyCash Portfolio Management System*, March 26, 1992 – www.c2.com/doc/oopsla92.html

First paper we know of on technical debt.

Ward Cunningham, Ron Jeffries, and others, *Technical Debt* www.c2.com/cgi/wiki?TechnicalDebt

Discussion with examples on how technical debt accumulates in projects.

QWAN

Quality Without A Name

Authors

Willem van den Ende

Willem van den Ende is a consulting developer. He is always looking for better and more fun ways to develop software, and helping others do the same. Since 1999 he lets organisations in benefit from agile software development as an all-hands person: coach, developer and facilitator. Always active in the local and international community, he has served as board member of the Agile Alliance, and started several conferences.

Willem is based in Bath, UK.

E-mail: willem@qwan.eu | Phone: +44 743 8651 672 | Twitter: @mostalive

Marc Evers

Marc develops developers, himself and other roles. He works as an independent coach, trainer and consultant in the field of (agile) software development and software processes. Marc develops true learning organizations that focus on continuous reflection and improvement: *apply, inspect, adapt*. Marc is co-founder of the Agile Open and XP Days Benelux conferences, and the Agile Holland user group.

Marc is based in Nieuwegein, The Netherlands

E-mail: marc@qwan.eu | Phone: +31 6 44 55 000 3 | Twitter: @marcevers

Rob Westgeest

Rob is a developing consultant. After years of experience with object-oriented software development with UML, several development processes and project approaches as developer, trainer and project leader, Rob worked on his first XP project in 2000. He has supported projects and people in the application of agile practices, principles and values since then. He's made plenty of mistakes, so his teams don't have to.

Rob is based in Hilvarenbeek, The Netherlands

E-mail: rob@qwan.eu | Phone: +31 6 4577 6328 | Twitter: @westghost

QWAN

Quality Without A Name

About QWAN

We are Quality Without A Name, a partnership of pragmatic practitioners. We develop software, teams, and individuals. We do so by coaching, mentoring, and training tailored to the needs of our clients. We don't shy away from taking responsibility, and are only happy when you are successful in the short term as well as the long term. We specialise in agile and lean software development. And we develop software ourselves - above all, we are software developers.

Talk to us, it might help...

Would you like to know what systems thinking (and doing!) can do for you and your organisation? We look forward to investigating your situation with you, for example through a workshop or mentoring. Feel free to contact us, we're always interested in new stories and tough problems.

Email: info@qwan.eu | www.qwan.eu

UK Office:	NL Office:
Living Software Ltd The Guild High Street Bath BA1 5EB United Kingdom Companies House for England and Wales: 08849005	Elsakkersstraat 25 5081 GL Hilvarenbeek The Netherlands Chamber of commerce Tilburg: 18071671

Get your Code Smells & Refactorings Card decks from:
www.qwan.eu/shop

www.qwan.eu