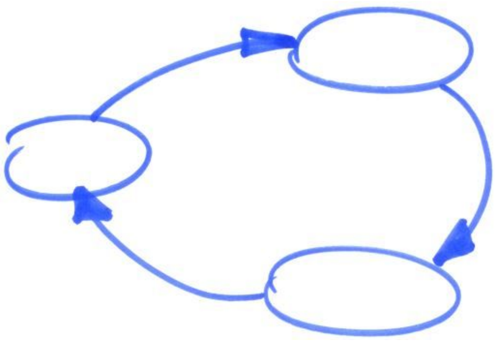
Promise is Debt



Marc Evers & Willem van den Ende

February 6, 2008

In this paper, we present the "Promise is Debt" pattern through the story of a team that gets stuck in a vicious circle of making promises to their customers, working hard trying to fulfil the promises, and making new promises when they fail.

Using systems thinking and diagrams of effects we uncover the dynamics of Promise is Debt. This helps to find the underlying causes and to break vicious circles.

The example is based on our experiences with organizations developing software and stories from participants of our workshops.

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 the ignite the next one?

In this paper, we describe a recognizable story from our experience – 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 the customers' disappointment. In the end, both the team and customers lose trust and the team loses its credibility.

We will show what the root causes are and how you can really solve the problem, using systems thinking with diagrams of effects.

Systems thinking is an approach where a (part of an) organisation or project is seen 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 about dynamics and change, with feedback playing an essential role.

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

Contents

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 – Angela, Fred, and Brian. The IWS product is partially generic, to keep maintenance costs down. They provide a number of 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 currently doesn't meet all their requirements yet. Each customer still needs specific functionality, but they're confident that the team is going to deliver it.

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

What's 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. Every feature is estimated by assigning a number of *feature points*. The number of feature points finished during the previous release – their *velocity* – is the maximum number they can plan for the next 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."

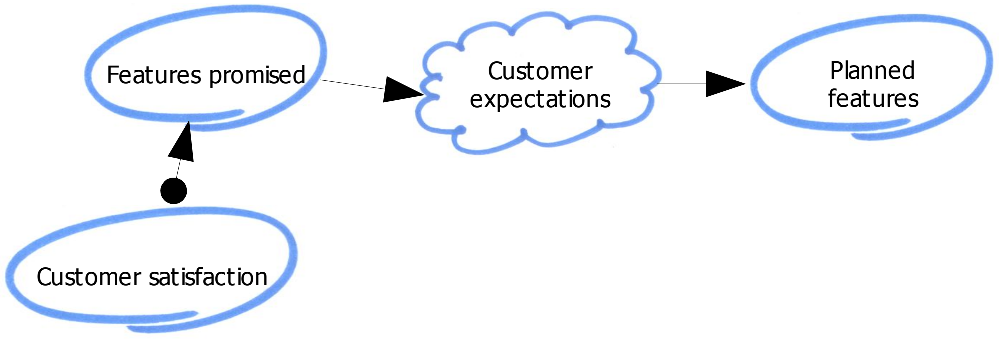
"But we have already planned enough for this release and this `FR53i` feature is a lot of work, at least 8 feature points!" objects Mary.

"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 their asses off to build the extra feature on time. Towards the end of the release, pressure and overtime increase....



If the customer is not satisfied enough, Jeff decides to promise extra features. The promises raise the customer's expectations. To meet these expectations, Jeff plans the extra features. The above diagram of effects shows this dynamic: the circles and clouds contain variables, the edges show causality relations.

Variables are properties of the system that we can observe (clouds) or measure (circles). Causality can work in the same or in the opposite direction (the latter indicated by the dot on the edge). If for example the number of features promised increases, customer expectations also increase. If customer satisfaction *de*creases, the number of features promised *in*creases.

The pizzas that Jeff brings in at evenings make up 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 during the next release, for instance by adding quite a few missing unit tests later.

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!"

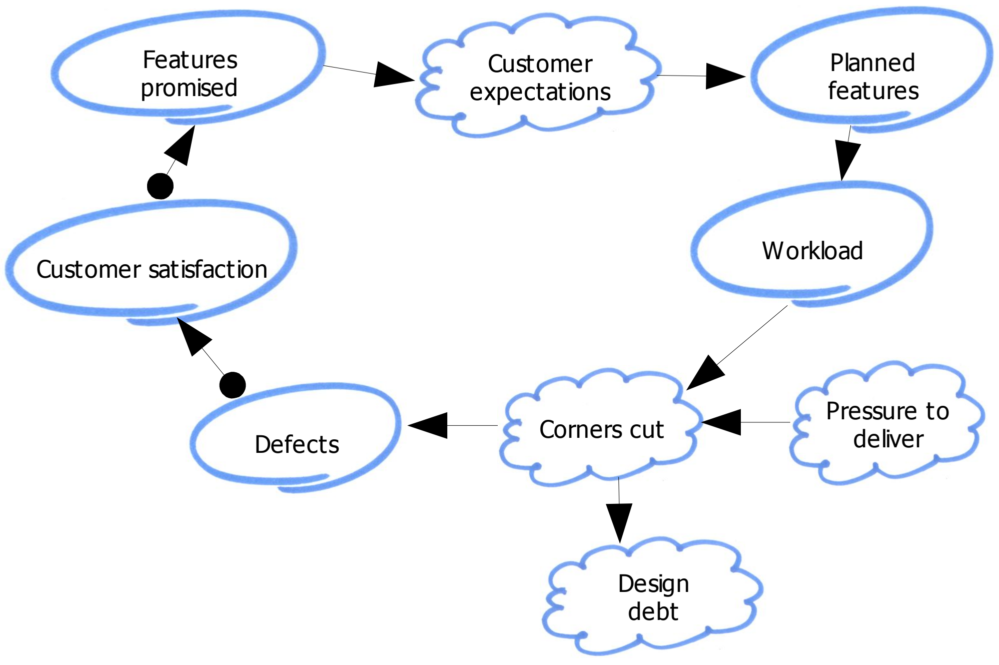
At the next planning meeting, the team has trouble restricting the number of features to be scheduled. Jeff would like to schedule as much work as they just delivered: 29 feature points. Martin sticks to his guns: "The high velocity is distorted: although we completed more, we didn't do it in a sustainable 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. One release earlier, we completed 21 feature points. This time we did 6 feature points extra. So now we can only promise 21 minus 6 is 15 feature points, to keep things sustainable. 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.

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 they cut, but they're just too tired too accomplish much. In the second week, with some pressure from Jeff, they start working on the planned features. Slowly they get up to speed.

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 who has found a nasty bug, explicitly stating his annoyance. The team immediately starts solving the defects, to prevent losing Ronald as their customer.



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 design debt as well as more defects introduced by working this way. More defects lead to lower customer satisfaction.

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: 11 feature points. Their customers are slightly disappointed.

Better luck next time

Ronald begins to openly express his doubts about the system. Jeff quickly 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 then 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 that the next release also includes feature `8RTv91x`, with the `HH05` extension.

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

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 planned work is much more than the velocity of the previous release (26 points); 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 ass. 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. They only finish 13 feature points worth of work this release.

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 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.

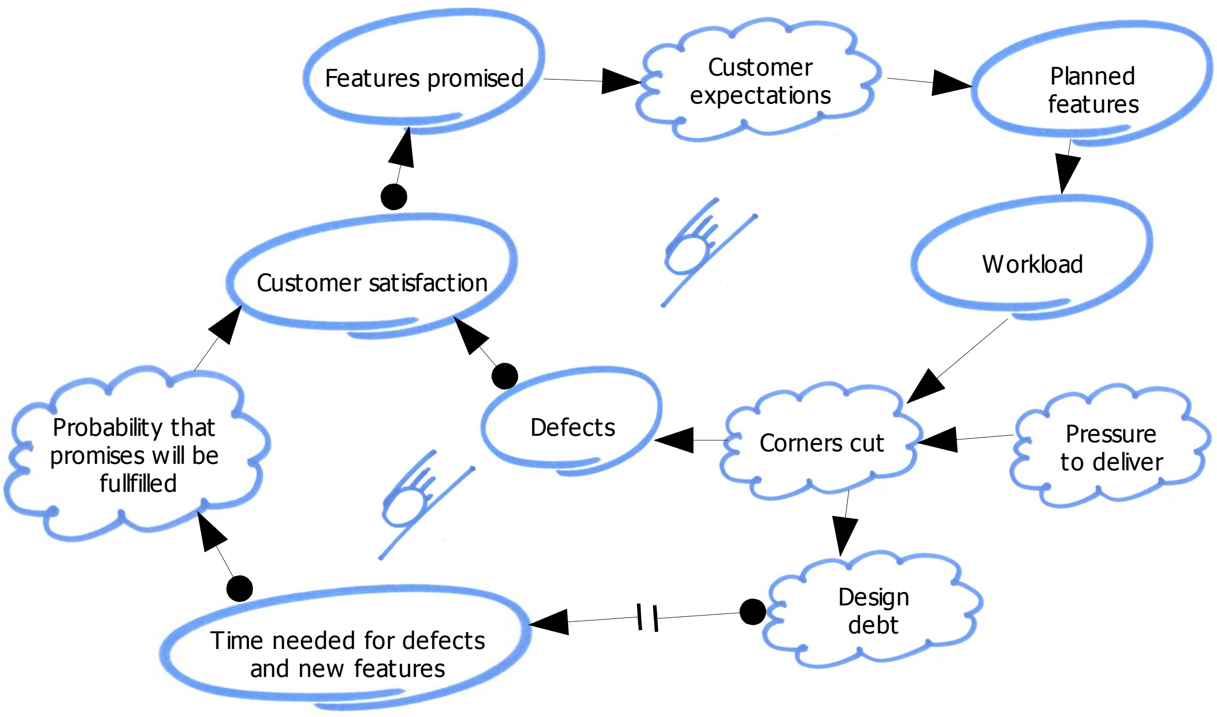
Jeff calls the team together in a conference room: "We need to work hard to regain the trust of our customers. I know you can do it, don't disappoint me! Just leave out refactoring, we don't have time for that. I get the impression that testing doesn't really contribute to productivity either. If everyone 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`. It's 24 feature points all together, that shouldn't be a problem, because you have done 29 once. I think your points are also subject to inflation, so it's ok to add some. 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 design debt. New defects keep on coming in, now predominantly caused by hasty fixes to previous defects. The team slowly loses its 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 11, or 10, because for 1 feature, they don't agree whether it's finished or not.

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.



After some time, the design debt starts having a noticeable effect on the time needed to solve defects or to build a new feature. The probability that promises will be fulfilled within the time estimated decreases. Not delivering what you've promised causes lower customer satisfaction.

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."

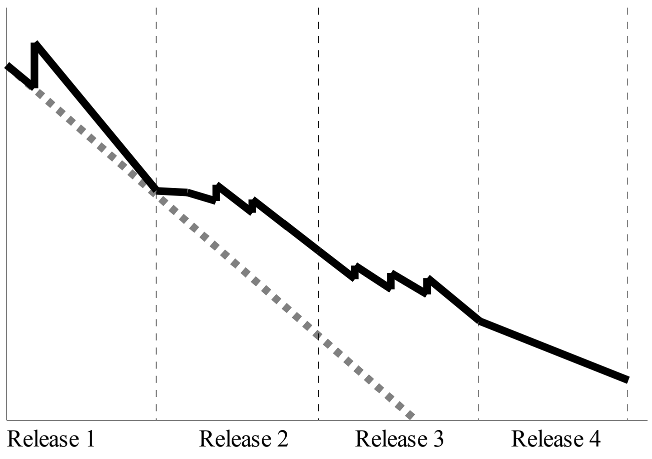
"No problem, it won't bother us much longer. We're looking around for a replacement system and we have identified two suitable candidates. I feel sorry for you, I've always liked collaborating with the developers."

"Well, I'm also almost 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 *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?

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 for instance 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, 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, 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 get burned out, developers leave.

Possible interventions

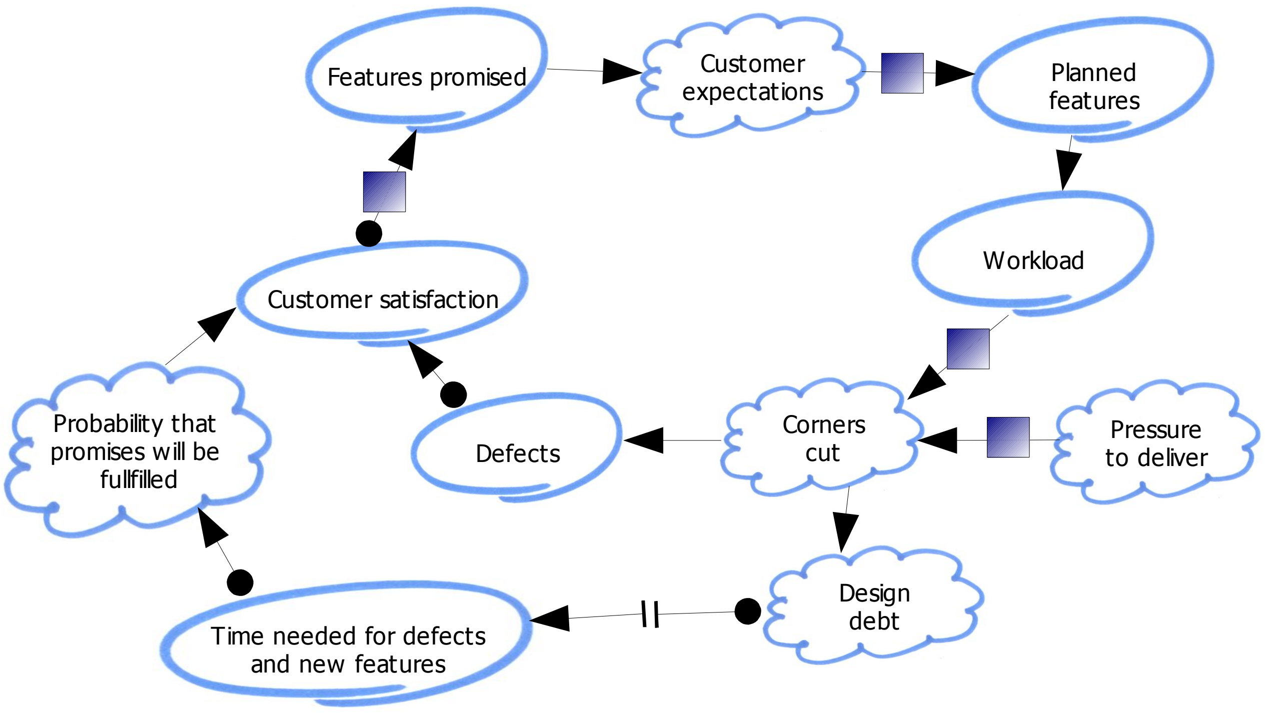
The cause-effect relations we found between the different variables are not all "laws of nature" or carved in stone: some represent an implicit or explicit choice – a *management decision*. The relation between customer satisfaction and extra features promised is an example of this – if the customer 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 that Jeff puts on the system; it is not sufficient to only intervene here, because the vicious circle remains intact;

- ✔ the number of extra features actually scheduled trying to satisfy the customer;
- ✔ the extent to which developers choose to work in a quick and dirty way when the pressure and the workload increase.

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.



By making your mental models and assumptions explicit in this way and by discussing them, you will see the available choices as well as those choices that make a structural change to the system dynamics.

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...

Conclusions

The "Promise is Debt" pattern, where someone overpromises to compensate for current problems, assuming they will catch up later, usually defeats 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 reinforce the vicious circle.

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.

Learning to observe well as a team and using diagrams of effects, help to make these indirect effects and vicious circles visible and solvable.

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.*

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*, in: 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.*

systemsthinking.net

> *Portal containing a wiki and a weblog aggregator, with writings from different systems thinkers within IT.*

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* design 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.*

Authors

Marc Evers (Piecemeal Growth)

Marc 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 also organizes workshops and conferences on agile and lean software development, extreme programming, systems thinking, theory of constraints, and effective communication. He is co-founder of the Agile Open and XP Days Benelux conferences.

Marc knows how to combine his real-world experience with knowledge that is out there to create novel solutions. He likes to add games to his highly-rated workshops, so participants have fun and learn from experience.

Phone: +31 6 44 55 000 3

Website: www.piecemealgrowth.nl

E-mail: marc@piecemealgrowth.nl

Piecemeal Growth

Willem van den Ende (Living Software BV)

Willem van den Ende is a Dutch eXtreme Programming pioneer. Since 1999 he guides organisations in the introduction of Agile Software development as an all-hands person: coach, developer and facilitator. Always active in the local and international community, he currently serves as board member of the Agile Alliance, host of systemsthinking.net and the European Agile Open conferences. Willem is an appreciated workshop facilitator at practitioners' conferences like XP(Day), Software Practice Advancement and Agile200*.

Willem's sharp vision, his broad knowledge, and twenty years of experience as programmer and coach enable him to adopt a very flexible and improvising attitude during workshops. He has the ability to let people see things differently.

Phone: +31 6 413 06 965

Website: www.livingsoftware.nl

E-mail: willem@livingsoftware.nl

Living Software

Would you like to know what systems thinking (and doing!) can do for you and your organisation? We look forward to helping you, for example through a workshop or mentoring. Feel free to contact us.