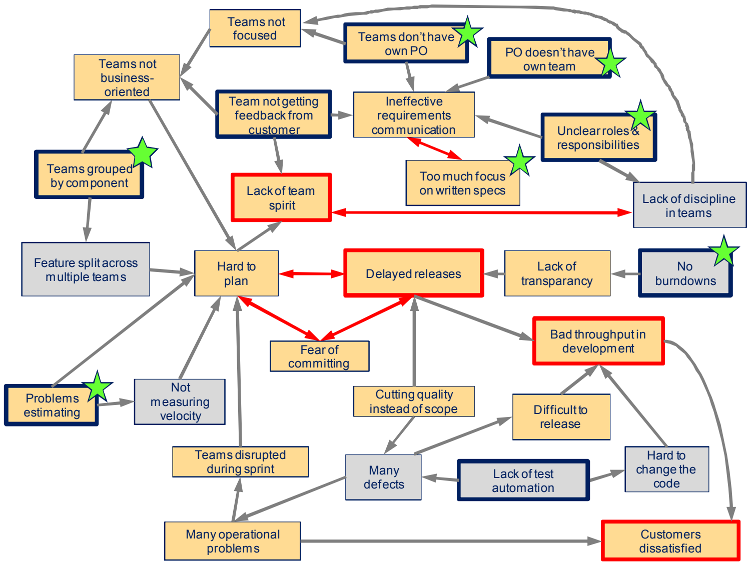
Cause-effect diagrams

A pragmatic way of doing root-cause analysis

Henrik Kniberg

Version: 1.1 (2009-09-28)


Teams not focused
Teams don't have own PO
PO doesn't have own team
Teams not business-oriented
Team not getting feedback from customer
Ineffective requirements communication
Unclear roles & responsibilities
Teams grouped by component
Lack of team spirit
Too much focus on written specs
Lack of discipline in teams
Feature split across multiple teams
Hard to plan
Delayed releases
Lack of transparancy
No burndowns
Fear of committing
Bad throughput in development
Problems estimating
Not measuring velocity
Cutting quality instead of scope
Difficult to release
Teams disrupted during sprint
Many defects
Lack of test automation
Hard to change the code
Many operational problems
Customers dissatisfied

Purpose of this article

Cause-effect diagrams are a simple and pragmatic way of doing root cause analysis. I've been using these diagrams for years to help organizations understand and solve all kinds of problems – technical as well as organizational.

The purpose of this article is to show you how cause-effect diagrams work, so you can put them to use in your own context.

Acknowledgements

I'd like to thank Tom Poppendieck for encouraging me to write about this, and for keeping my reading list constantly stocked with interesting books and articles about Lean and Systems thinking.

Solve problems, not symptoms

The key to effective problem solving is to first make sure you understand the problem that you are trying to solve - why it needs to be solved, how you will know when you've solved it, and what the root cause is.

Often symptoms show up in one place while the actual cause of the problem is somewhere completely different. If you just "solve" the symptom without digging deeper it is highly likely that problem will just reappear later in a different shape.

Problem: Smoke in my bedroom.
Bad solution: Open the window and go back to sleep.
Good solution: Find the source of the smoke and solve it. Whoops, there's a fire in the basement! Extinguish it, find out what caused the fire in the first place, install a fire alarm for earlier warning next time.

Problem: Hot forehead, tired.
Bad solution: Put ice on forehead to cool it down. Eat some sugar to wake up. Keep working.
Good solution: Take my temperature. Oh, I have fever! Go home and rest.

Problem: Memory leak in server.
Bad solution: Buy more memory.
Good solution: Find & fix the source of the memory leak. Implement tests to detect new memory leaks in the future.

Problem: Water in the boat.
Bad solution: Pump out the water & keep sailing.
Good solution: Find the source of the water. Ah, a hole! Fix it. Then pump out the water.

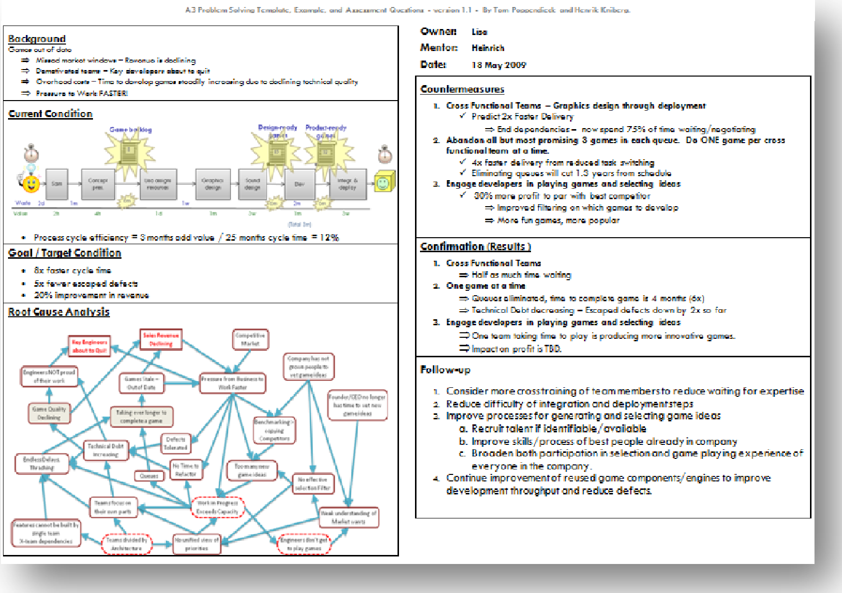
... and so on.

Most problems in organizations are systemic. The "system" (your organization) has a glitch that needs to be fixed. Until you find the source of the glitch, most attempts to fix the problem will be futile or even counterproductive.

A3 thinking - the Lean problem solving approach

One of the core tenets of Lean Thinking is *Kaizen* – continuous process improvement. Toyota, one of the most successful companies in the world, attributes much of their success to their highly disciplined problem solving approach. This approach is sometimes called A3 thinking (based on the single A3-size papers used to capture knowledge from each problem solving effort).

Here's an example & template:
http://www.crisp.se/lean/a3-template



A3 Problem Solving Template, Example, and Assessment Questions - version 1.1 - By Tom Poppendieck and Henrik Kniberg.

Background
Games out of date
⇒ Missed market windows – Revenue is declining
⇒ Demotivated teams – Key developers about to quit
⇒ Overhead costs – Time to develop games steadily increasing due to declining technical quality
⇒ Pressure to Work FASTER!

Current Condition

- Process cycle efficiency = 3 months add value / 25 months cycle time = 12%

Goal / Target Condition

- 8x faster cycle time
- 5x fewer escaped defects
- 20% improvement in revenue

Root Cause Analysis

Key Engineers about to Quit
Sales Revenue Declining
Competitive Market
Company has not grown people to vet game ideas
Engineers NOT proud of their work
Games Stale – Out of Date
Pressure from Business to Work Faster
Founder/CEO no longer has time to vet new game ideas
Game Quality Declining
Taking ever longer to complete a game
Benchmarking – copying Competitors
Defects Tolerated
Technical Debt Increasing
Endless Delays Thrashing
No Time to Refactor
Queues
Too many new game ideas
No effective selection Filter
Teams focus on their own parts
Work in Progress Exceeds Capacity
Weak understanding of Market wants
Features cannot be built by single team X-team dependencies
Teams divided by Architecture
No unified view of priorities
Engineers don't get to play games

Owner: Lisa
Mentor: Heinrich
Date: 18 May 2009

Countermeasures

1. **Cross Functional Teams – Graphics design through deployment**
   ✓ Predict 2x Faster Delivery
   ⇒ End dependencies – now spend 75% of time waiting/negotiating
2. **Abandon all but most promising 3 games in each queue. Do ONE game per cross functional team at a time.**
   ✓ 4x faster delivery from reduced task switching
   ✓ Eliminating queues will cut 1.3 years from schedule
3. **Engage developers in playing games and selecting ideas**
   ✓ 30% more profit to par with best competitor
   ⇒ Improved filtering on which games to develop
   ⇒ More fun games, more popular

Confirmation (Results)

1. **Cross Functional Teams**
   ⇒ Half as much time waiting
2. **One game at a time**
   ⇒ Queues eliminated, time to complete game is 4 months (6x)
   ⇒ Technical Debt decreasing – Escaped defects down by 2x so far
3. **Engage developers in playing games and selecting ideas**
   ⇒ One team taking time to play is producing more innovative games.
   ⇒ Impact on profit is TBD.

Follow-up

1. Consider more crosstraining of team members to reduce waiting for expertise
2. Reduce difficulty of integration and deployment steps
3. Improve processes for generating and selecting game ideas
   a. Recruit talent if identifiable/available
   b. Improve skills/process of best people already in company
   c. Broaden both participation in selection and game playing experience of everyone in the company.
4. Continue improvement of reused game components/engines to improve development throughput and reduce defects.

With the A3 approach, a significant amount of time (the left half of the sheet) is spent analyzing and visualizing the root cause of a problem *before* proposing solutions. A cause-effect diagram is only one way of doing a root-cause analysis. There are other ways too, such as value stream maps and Ishikawa (fishbone) diagrams. The sample A3 above contains a value stream map (top left) and a cause-effect diagram (bottom-left).

The nice thing about cause-effect diagrams is that they are fairly intuitive and self-explanatory (especially compared to fishbone diagrams). Another advantage is that you can illustrate reinforcing loops (vicious cycles), which is very useful from a systems thinking perspective.

The rest of this article describes how to effectively create and use these diagrams.

How to use cause-effect diagrams

Here’s the basic process:

1. Select a problem – anything that’s bothering you - and write it down.
2. Trace “upwards” to figure out the business consequences, the “visible damage” that your problem is causing.
3. Trace “downwards” to find the root cause (or causes).
4. Identify and highlight vicious cycles (circular paths)
5. Iterate the above steps a few times to refine and clarify your diagram
6. Decide which root causes to address and how (i.e. which countermeasures to implement)

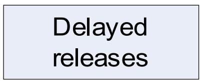
Later on, follow up. If your countermeasures work, then congratulations! If your countermeasures didn’t work, then don’t despair. Analyze why it didn’t work, update your diagram based on the new knowledge gained, and try some other countermeasures.

So a countermeasure is in fact an *experiment*, not a *solution*. Your *hypothesis* is that this countermeasure will solve (or mitigate) the problem, but you can never be sure. You are in effect prodding your system to see how it reacts. That’s why the follow-up is important.

Failure really just means that your system is trying to tell you something – so you’d better listen. The only *real* failure is the failure to learn from failure!

Example 1: Long release cycle

Let's say our problem is that we always miss deadlines. More specifically, our releases always occur at a later date than planned.

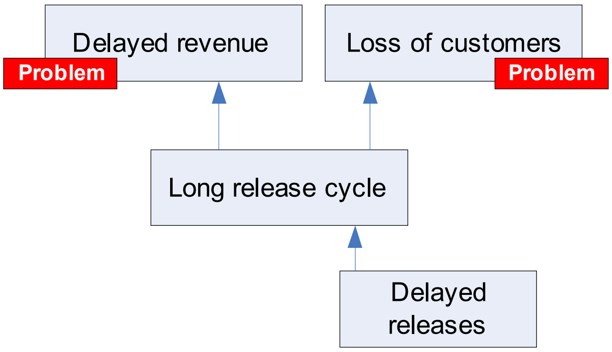


A problem is only a problem if it conflicts with your goal. So start by defining your goal, and think about the consequences of this problem in terms of your goal. This can be done by asking a series of "so what?" questions until you identify the visible damage.

Let's say our goal is to delight our customers and maximize our revenue. Our dialog might sound something like this:

Q: "Who cares if the releases are delayed? What is the consequence?"
A: "Delays make our release cycles long"
Q: "So what?"
A: "That delays our revenues, which messes up our cash flow. It also causes us to lose customers, since they are impatient and don't like waiting longer than necessary."

As we talk, we add boxes and cause-effect arrows to the diagram. Normally I try to go "upwards" from the original problem statement when mapping out consequences, but that isn't a strict rule.



So delayed releases isn't really the problem. The *real* problem is delayed revenue and loss of customers. At this point we should consider three things:

1. Are there any *other* issues that are causing loss of customers or delayed revenues? If so, are delayed releases the biggest culprit or should we turn our attention elsewhere?
2. Can we quantify the problem? How much revenue have we lost? How many customers have we lost? This data will help us evaluate how much effort it is worth spending to solve this problem.
3. How will we know when we've solved the problem? If a consultant comes in, does a noisy rain dance and then proudly proclaims says "I've solved the problem now", how will we call the bluff?

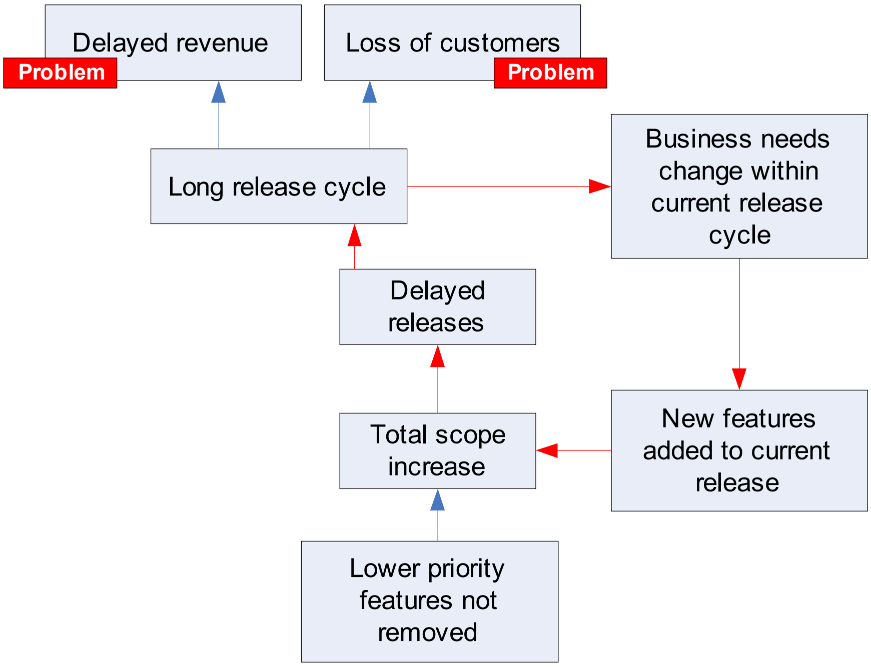
Once we've spent some time analyzing the consequences of the problem it is time to dig downwards – towards the root.

This is done by asking a series of "why" questions. Yes, this is the "five whys" technique that you've probably heard of if you've studied lean thinking.

Q: "Why are the releases delayed?"
A: "Because the scope keeps increasing"
Q: "Why?"
A: "Because the customers comes up with new features and insists that we add them to the current release, and refuses to allow us to remove lower priority features."
Q: "Why? Why not defer the features until next release?"
A: "Because the release cycle is so long, so new demands appear before the release is done."

OK, that was only 3 whys. But you get the picture.

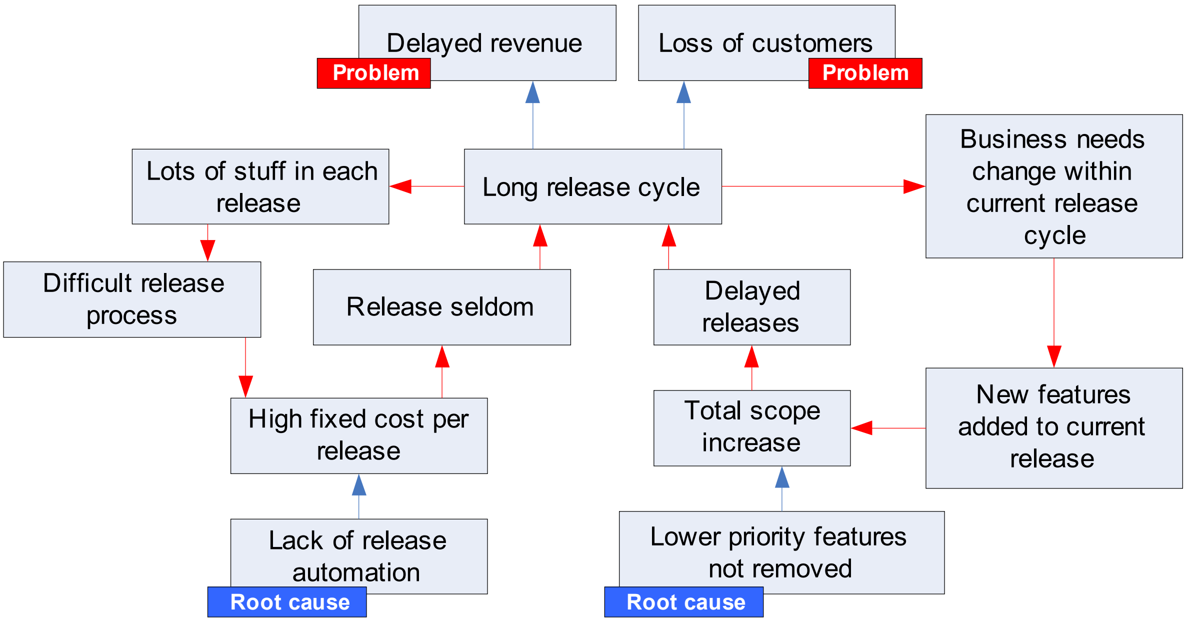
This dialog gives us the following picture:


Delayed revenue
Problem
Loss of customers
Problem
Long release cycle
Business needs change within current release cycle
Delayed releases
Total scope increase
New features added to current release
Lower priority features not removed

The vicious cycle (or re-enforcing loop) is highlighted with red arrows. Recurring problems almost always involve loops like this, but they may take some time to find. Spotting these will greatly increase your likelihood of solving the problem effectively and permanently!

Our goal is to identify the root cause(s) of this problem, so we can achieve maximum effect with minimum effort. It is easy to miss important causes on the first pass – so go back and ask a few more whys.

Q: "Why is the release cycle long? Are delayed releases the only cause?"
A: "Well actually, even without the delays our planned release cycles are quite long."
Q: "How long is your planned release cycle?"
A: "Once per quarter."
Q: "Why so long then?"
A: "Because releases are expensive and complicated."
Q: "Why?"
A: "Because there's so much stuff in each release, and because it's all manual work."


Delayed revenue
Problem
Loss of customers
Problem
Lots of stuff in each release
Long release cycle
Business needs change within current release cycle
Difficult release process
Release seldom
Delayed releases
High fixed cost per release
Total scope increase
New features added to current release
Lack of release automation
Root cause
Lower priority features not removed
Root cause

Look to the left, another vicious cycle (red arrows)! Long time between releases means lots of stuff in each release, which means releases are difficult and expensive, which makes us reluctant to have frequent releases.

As you see, I've decided to label two root causes. Now it's time to propose countermeasures:

Root cause	Countermeasure
Lack of release automation	Implement release automation
Lower priority features not removed	Negotiate a rule with the customer, allowing them to add new features to a release only if they remove lower priority features of corresponding size.

There’s no strict rule for determining which issue is the root cause, but here are some indicators.

- This issue has only arrows going out, and no arrows coming in.
- It doesn’t feel meaningful to dig further down (ask further “why” questions) from here.
- This issue is something we can address, and it will probably have a significantly positive effect on the problem.

The “five whys” technique is called so because it typically takes about five “why” questions to get to the root. We tend to stop asking too early, so keep digging!

Note that the originally identified problem – delayed releases – wasn’t really a problem or a root cause. It was just a symptom. We used that as a *handle* to dig upwards to identify the real problem, and then downwards to identify the root causes. This allows us to propose effective countermeasures in an informed way.

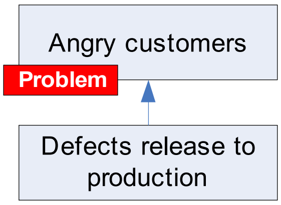
Without this type of analysis, we tend to jump to conclusions and execute ineffective and counterproductive changes. For example adding more people, even though head count had nothing to do with the problem. Or changing the incentive model (reward people for releasing on time or punish people for releasing late) even though the current incentive model had nothing to do with the problem. I bet you’ve already seen that happen a few times.

Example 2: Defects released to production

Let's say we are having problems with defective code being released to production.

Q: So what?

A: The defects make our customers angry



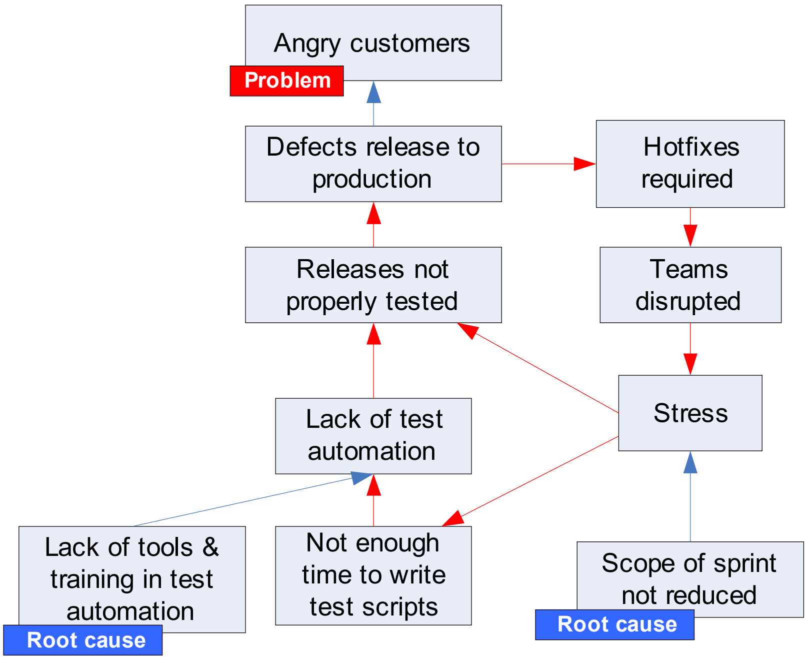
Q: Why are defects released to production?

A: Because they aren't properly tested before release.

Q: Why not?

(etc)

Here's where we ended up:


Angry customers
Problem
Defects release to production
Hotfixes required
Releases not properly tested
Teams disrupted
Stress
Lack of test automation
Lack of tools & training in test automation
Root cause
Not enough time to write test scripts
Scope of sprint not reduced
Root cause

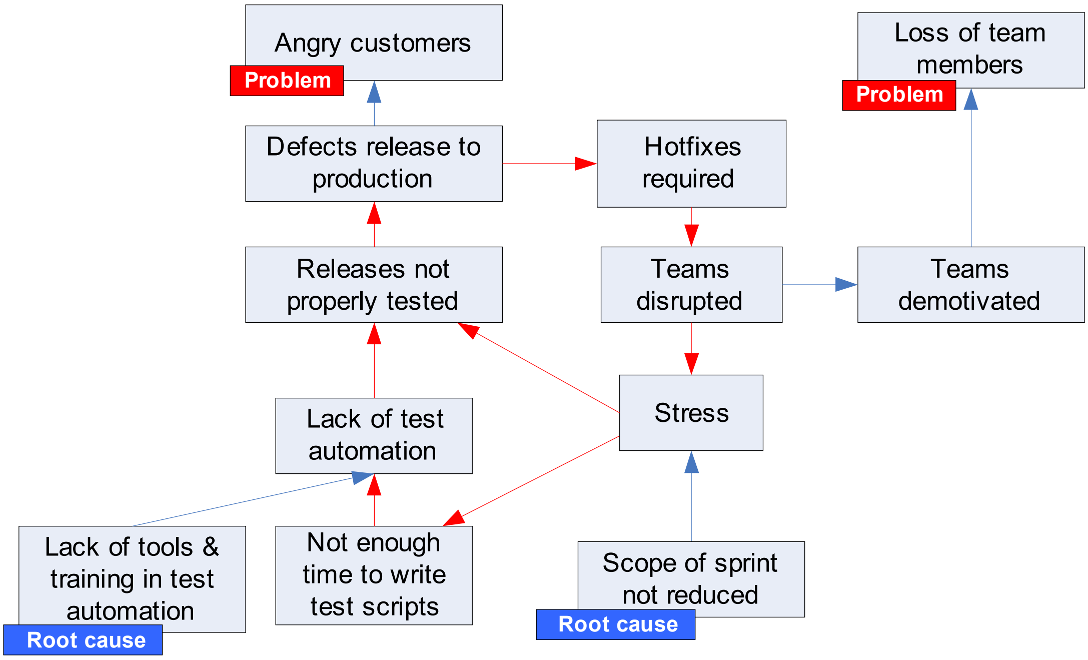
Look at that, two reinforcing loops! Check out the red arrows.

Loop 1 (inner loop): Defects in product causes hot-fixes, which disrupts the team. Since they aren't allowed to reduce scope, they are stressed and don't have time to test new releases properly. Which, of course, leads to more defects in production.

Loop 2 (outer loop): Because they are stressed, they don't have time to write automated test scripts either. This leads to an overall lack of test automation, making it harder and harder to regression-test new releases properly, which leads to defects in production and hot-fixes and ultimately more stress.

But wait, there's more!

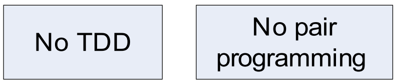
Teams hate being disrupted. This disturbs flow and, in the long run, ruins motivation. This might explain which the staff turnover rate has been high! So in solving the original problem (defects in production) we get the added bonus of reducing team turnover!


Angry customers
Problem
Loss of team members
Problem
Defects release to production
Hotfixes required
Releases not properly tested
Teams disrupted
Teams demotivated
Stress
Lack of test automation
Lack of tools & training in test automation
Root cause
Not enough time to write test scripts
Scope of sprint not reduced
Root cause

That's the nice thing about addressing the root cause. Root causes are usually the cause of more than one problem (that's why they are called "root"...).

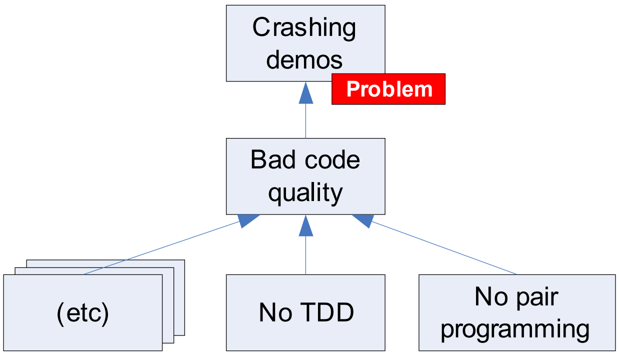
Example 3: Lack of pair programming

I was asked to help a client figure out why they weren't doing XP practices like pair programming and test-driven development. "We know that we should be doing it, but we aren't".



So is lack of TDD and pair programming really a problem? As usual, the things we call problems often turn out to be just symptoms.

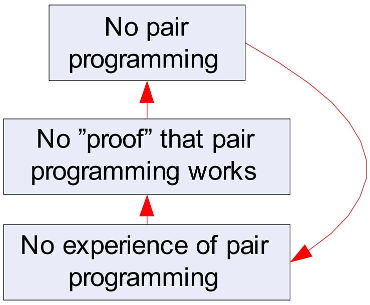
Q: What is the consequence of not doing pair programming and TDD?
A: We think we would have much better code quality if we did these things.
Q: What is the consequence of bad code quality? Have you encountered any actual problems due to bad code quality?
A: Yeah, we've had some crashing demos. We are a research company and demos are how we get business, so this is really a problem.



OK, let's take one of the issues and see if we can dig down to the root.

Q: Why aren't you pair programming then?
A: Because many people are afraid that it won't work, and that we will be wasting our time. We have no proof that it works.
Q: What kind of "proof" would you need?
A: Well, we've seen studies that indicate that it works. But nobody here has really tried it, so we aren't sure that it works.

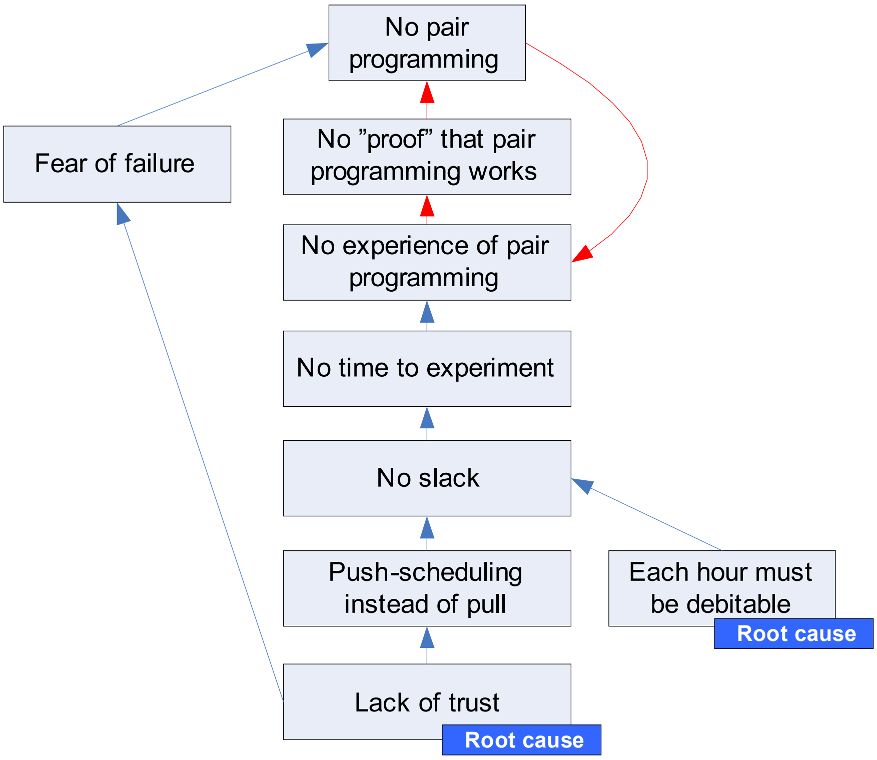
Well, there's the first loop:


No pair programming
No "proof" that pair programming works
No experience of pair programming

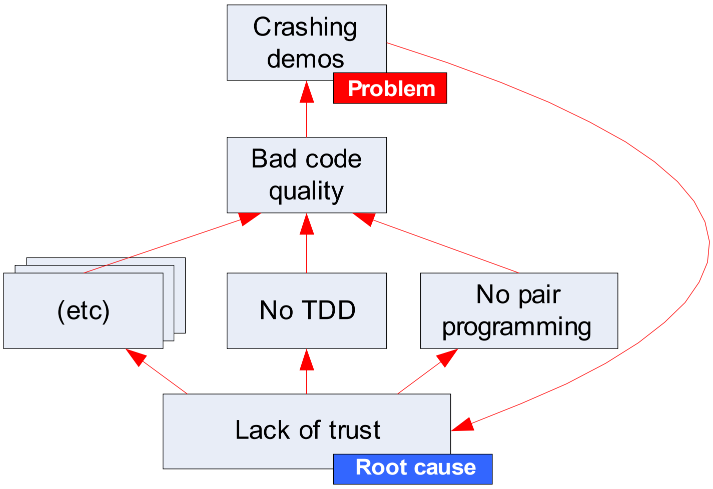
They don't want to do it because they don't know that it will work. And they don't know that it will work because they haven't tried it...

Q: Why haven't you at least given pair programming a try?
A: We don't have time to experiment.
Q: Why not?
A: Because we don't have any slack. Each hour is accounted for. Our customers keep piling work on us.
Q: Why don't they let you manage your own time, and let you pull in more work whenever you are ready?
A: They don't trust us to use our time effectively.

The lack of trust also leads to a general fear of failure, which of courses reduces the likelihood that they will try something new like pair programming without "proof" that it works.


No pair programming
Fear of failure
No "proof" that pair programming works
No experience of pair programming
No time to experiment
No slack
Push-scheduling instead of pull
Each hour must be debitable
Root cause
Lack of trust
Root cause

So there appears to be two big root causes: Lack of trust, and the management principle that every hour must be debitable. Let's fold this back into the big picture.



Lack of trust turned out to be the root cause of not doing XP practices such as TDD and pair programming. That causes bad quality which causes crashing demos. And guess what? Crashing demos reduce trust even further. There's a vicious circle for you!

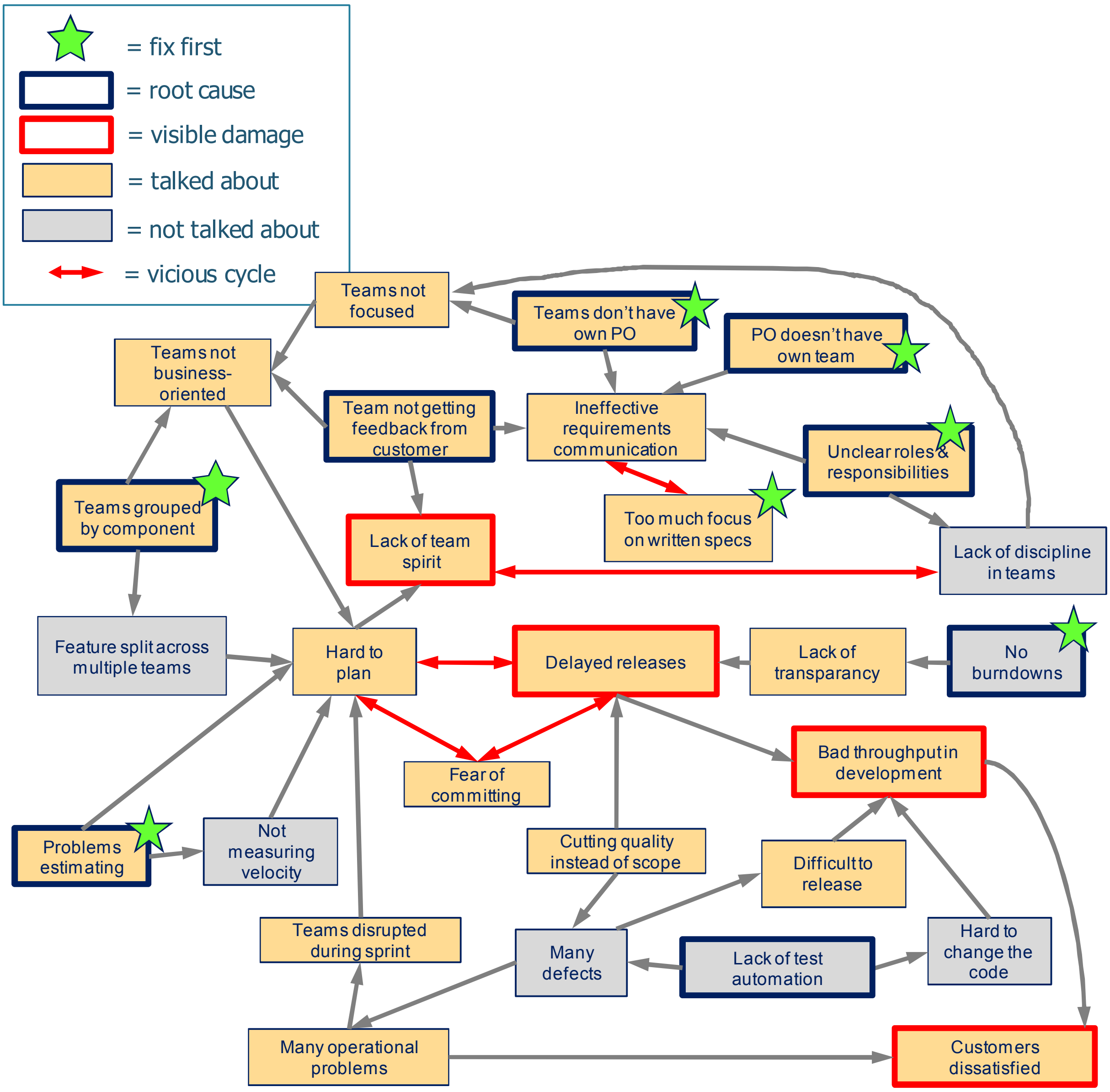
The interesting thing is that we did this in a 2-day workshop with about 25 people. At the beginning we were talking mostly about technical stuff – how to get started with TDD and pair programming. That didn't really get us anywhere, so we instead split into groups and had each group choose one problem and start drawing cause-effect diagrams and create problem solving A3s. The interesting thing is that several of the groups that were analyzing seemingly different problems came up with the same root cause – lack of trust! The diagram above was just one example of this.

So by the end of the day we were all talking about what we could do to increase the level of trust between the customer and the developers, which was a surprising turn of events.

For starters we agreed that we should invite "them" (the customers) to participate next time we do this type of workshop, which should lessen the use of terms like "us" and "them"...

Example 4: Lots of problems

Here's a bigger example. This organization was doing Scrum but was having some problems. After some interviews and workshops the cause-effect diagram that emerged showed that they weren't really doing Scrum correctly and that this was causing problems.



It became clear to everyone that many of the root causes would be addressed with a "proper" Scrum implementation (for example reorganizing into cross functional teams, and making sure each team has a dedicated product owner). This triggered organizational changes that ultimately fixed many of the root causes (green stars). The next step was to improve test automation.

Scrum isn't always the solution of course. In fact, sometimes Scrum itself is the problem and other techniques such as Kanban are the solution See my upcoming book "Kanban and Scrum – making the most of both" for more on that.

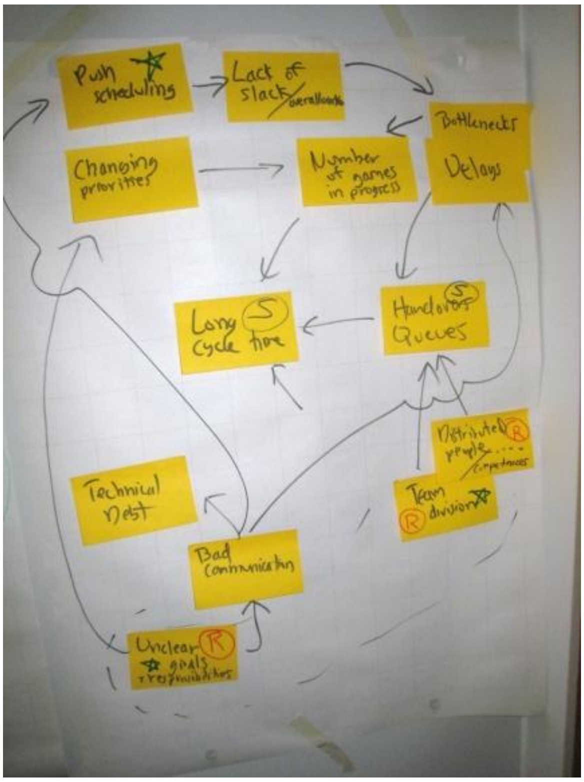
Practical issues – how to create & maintain the diagrams

Working alone

When creating the diagrams alone I find it easiest to work directly with a diagramming tool such as Visio or Powerpoint. It's nice to be able to move things around quickly, resize the boxes, and make quick backups when playing around with the picture.

Working in a small group (2 – 8 people)

Gather in front of a whiteboard or flipchart. Use stickynotes for issues, and draw arrows to connect them. Whiteboard is preferable, so you could erase and redraw the arrows as you move the stickynotes around. Make sure everybody is helping out, not just one person doing all the drawing. Make sure someone takes a high-resolution photo and sends to everyone after the meeting.



Working in a larger group (9 – 30 people)

Split the group into smaller teams, each focused around one specific problem. It is OK to have multiple teams working independently on the same problem – they may come to the same conclusion or different conclusions, and both cases are interesting. Each team works with a flipchart/whiteboard and stickynotes. Gather everyone together at regular intervals to share insights.

Long-term maintenance of a diagram

Let the diagram live in a tool such as Visio or Powerpoint. Whenever you get to a workshop setting, decide if the meeting is mostly for presenting the diagram, or for updating it. If presenting, use a projector to show the diagram directly in Visio (or whatever tool you use). If updating the picture, replicate it on a whiteboard/flipchart with stickynotes and arrows so that people can collaborate effectively. Then synchronize with the electronic tool after the meeting.

This type of synchronizing does take some time, but it is often worth it. Nothing can beat physical tools like whiteboards & stickynotes when doing team workshops.

Pitfalls

Too many arrows and boxes

Sometimes the diagram gets too messy to be readable. In that case you need to simplify it. Here are some techniques:

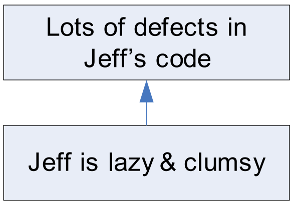
- Remove redundant boxes (i.e. boxes that don't add much value to the diagram).
- Focus on "depth first" rather than "breadth first". Don't write *all* causes of a problem, write only the most important one or two, and then keep digging deeper.
- Accept imperfections, a diagram like this will never be perfect.
  "All models are wrong, but some are useful" (George Box)
- Maybe your problem area is too broad, try to limit yourself to a more narrowly defined problem.
- Split the diagram into pieces, like I did in example 3 above.

Oversimplification

This type of cause-effect diagram is simple, intentionally so. It doesn't replace face-to-face communication. If you need something more advanced or formally defined read a book on systems thinking, such as "The Fifth Discipline" by Peter Senge. There are ways to distinguish between reinforcing loops and balancing loops, and ways of adding a temporal dimension (showing how X causes Y but with a delay). Just beware – even a "perfect" diagram is pretty useless if you need a doctor's degree to understand it.

Getting personal

Avoid "blame game" issues such as:



Problem solving works best if you assume that all problems are systemic. Sure there are clumsy people. But even if that is causing us significant problems then that is still a *systemic* problem – we have a system that assumes clumsy people aren't clumsy, or a system that lets extremely clumsy people in, or a system that doesn't help clumsy people get less clumsy, etc.

This point is worth emphasizing: Treat all problems as systemic!

Summary: Why to use cause-effect diagrams

- **Create a common understanding**
  - Team-based problem is extremely effective, but requires a common understanding of the problem. Cause-effect diagrams are a very practical collaboration technique.
- **Identify how problems affect the business**
  - So that you can focus on the most important problems first and make informed decisions.
- **Find root causes**
  - So that you can maximize the effect of your changes.
- **Find vicious cycles** (negative reinforcing loops)
  - So that you can break them, or turn them into positive reinforcing loops (good stuff leading to more good stuff, instead of bad stuff leading to more bad stuff).

Good luck!

/Henrik

Email: henrik.kniberg AT crisp.se
Web: http://www.crisp.se/henrik.kniberg
Blog: http://blog.crisp.se/henrikkniberg