Thinking for a Change

Origins

The "Thinking Processes" originated from the Theory of Constraints, the ideas for process improvement developed by Elyahu Goldratt. He realized that he was becoming a bottleneck in the dissemination of the ideas behind the Theory of Constraints. The Thinking Processes are a set of tools and heuristics that Goldratt uses.

The Theory of Constraints' process optimisation technique "The 5 focusing steps" is easily applied to physical, logistical processes like manufacturing, because the bottleneck and flows are visible. Applying the same ideas to more abstract problems in knowledge work or to improve rules and organisations is a lot more difficult. The Thinking Processes tools allow us to visualize this kind of situation.

The Thinking Processes were introduced in Goldratt's second business novel "It's Not Luck". "Thinking for a Change" is the title of a book about the Thinking Processes, written by Lisa Scheinkopf.

Goals of the tools

- Verbalize and make explicit intuition about systems and situations
- Allow a group to analyse and discuss situations, to come to a shared understanding
- A structured method to uncover hidden assumptions and question them in a constructive manner
- Create consensus before a major decision, by involving all affected stakeholders ("Nemawashi")
- Provide a structured, step-by-step approach to systems thinking that helps participants to focus on the goals to achieve.

The different tools

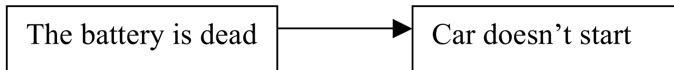
- **Current Reality Tree**: helps you to find one or a few root causes for problems you're facing. Now you know where to intervene to really solve the problems.
- **Future Reality Tree**: helps you to visualize the effects of a proposed intervention, including potential undesirable effects. Now you know if your intervention will result in the desired and effect. You know the extra interventions you will need to undo or avoid negative side effects.
- **Transition Tree**: allows you to map a path from where you are to where you want to be, by laying out a series of actions that will bring you closer to the goal, via a series of intermediate milestones.
- **Prerequisite Tree**: allows you to plan back from a desired state, by looking for actions that overcome obstacles.
- **Evaporating Cloud**: allows you to resolve conflicts between different courses of action, by surfacing and examining assumptions.

Simple Notation

Entity

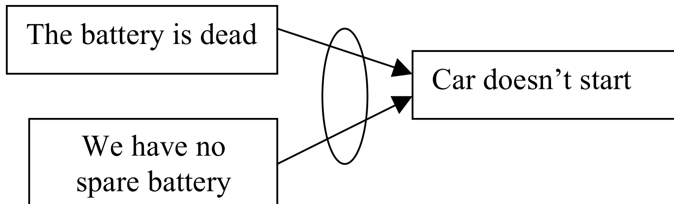
An entity is an element of the system. It describes a certain state.

Cause – Effect



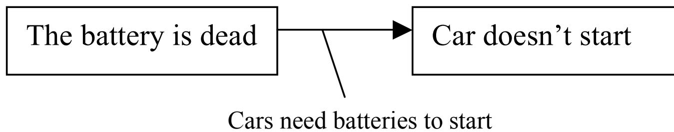
The car doesn't start (effect) BECAUSE the battery is dead (cause).

And Connector



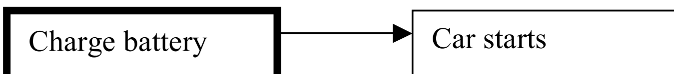
The car doesn't start BECAUSE the battery is dead AND we have no spare battery.

Assumption



The car doesn't start BECAUSE the battery is dead IS ONLY TRUE IF cars need batteries to start.

Action (or injection)



BECAUSE we've charged the battery, the car starts.

.

Making a Current Reality Tree

Find the root cause of undesirable effects

Step 1: Describe the system, its goal and the symptoms

1. Determine the scope of the system: what is the system we're analysing? What are its boundaries?
2. What is the goal of the system? Why does it (continue to) exist? What are the major measures of success?
3. Brainstorm a few (< 5) undesirable attributes of this system. What's bothering you? What could be done better? Don't analyse, just write them down. Use simple, definite sentences. These are your initial **entities**.

Example:

1. **System:** This is about the IT organisation (several hundred people) that supports the Belgian Postal system. More specifically, about the **development** teams that write the software and the **operations** teams (admins) that install and support the software.
2. **The goal of the system** is to create and maintain the IT systems that allow the business to offer its service and generate value. We can measure this by looking at "business value" generated vs cost.
   To make projects more manageable, more focused and to deliver value sooner, developers would like to make smaller releases, which are installed sooner and often, thereby increasing business value. However, this is not allowed: because installing software is difficult and risky, more frequent releases would increase costs for operations.
3. **The goal of the tree** is to find the root causes for the cost and risk of installations. If we can tackle those, we might be able to release more frequently. See the "Evaporating Cloud" later in this document.
4. **Initial undesirable entities:**
   - Installing is difficult
   - Installing is risky

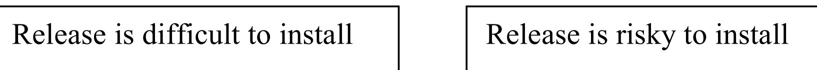
Step 2: Find effect-cause-effects. “Why does this happen?”

1. Start with the worst entity. Which one would you like to get rid of most?
2. Ask yourself: “Why <entity>?”
   - If the answer is a new entity, create it
3. Connect the cause to the effect
4. Repeat the question for the other effects to work in the breadth of the diagram
5. or ask the “Why” question for the causes to drill deeper
   - You might find more than one cause for an effect
   - You might find more than one effect from a cause

Note: in the “Toyota Way” there is a technique called the “5 Whys”, indicating that you should look for the root cause approximately 5 levels down from the original symptom.

Example:

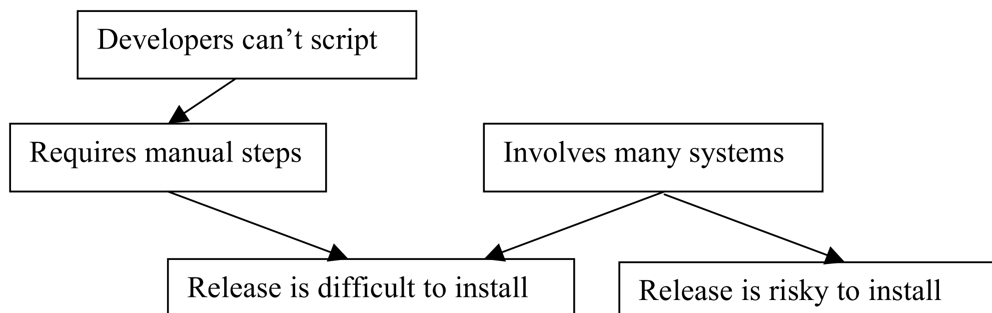
We start with the following entities:



Q: “Why is installing difficult?”
A: “Installing is difficult BECAUSE it requires many manual steps” (new entity)
A: “Installing is difficult BECAUSE it usually involves many systems” (new entity)

Q: “Why is installing risky?”
A: “Installing is risky BECAUSE it usually involves many systems”

Q: “Why do installs require many manual steps?” (Digging deeper)
A: “Installs require many manual steps BECAUSE developers don’t know how to automate tasks using scripts”.



Step 3: Legitimate reservations, testing the model

The "legitimate reservations" are critical questions to ask when making a tree. When you've added a few entities and/or relations, stop to ask these questions, to clarify and simplify the tree. This is the moment to make assumptions explicit so that everybody participating in the exercise agrees on the current state of the tree, before going further.

Important: only the legitimate reservations are allowed. Don't accept any kind of complaint, "Yeah but" or "That won't work".

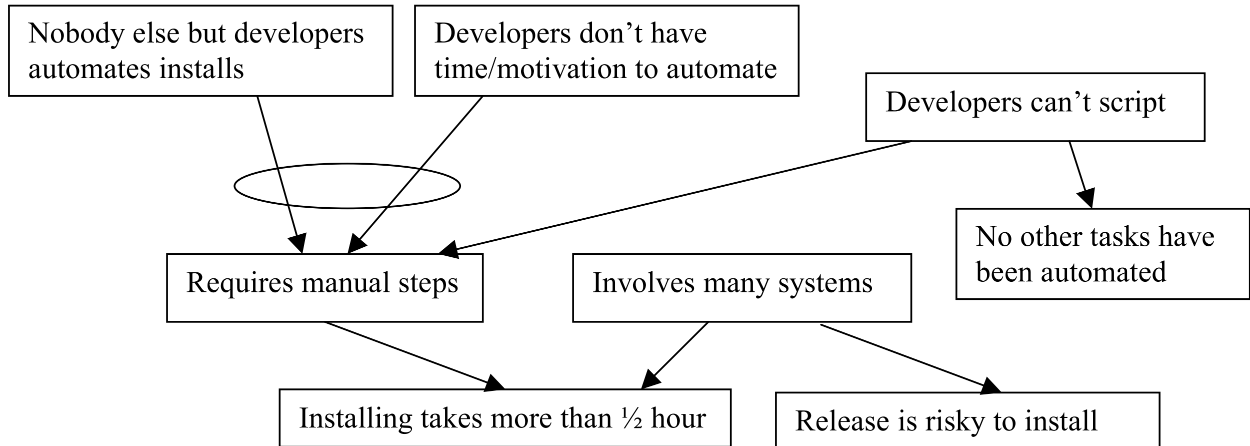
There are two categories of reservations. Test them in the given order.

1. **Level 1 reservations** involve a single entity or relation at a time
   a. **Clarity:** does everyone understand the entity description the same way? Can you make the description clearer, simpler, less ambiguous? Restate the entity in a different way to verify if everyone understands the entity like you do.
   b. **Entity existence**: does everyone agree that the entity exists? How can we "see" the entity? What proof do we have of its existence?
   c. **Causality existence**: is everyone convinced that the entity really causes the effect? What are the assumptions behind that relation?
2. **Level 2 reservations** involved more than a single entity and relation
   a. **Additional Cause**: Is the given entity the only possible cause for the effect? What else could have that effect? Could that additional cause also exist in the system? If so, how could we tell? Add the additional cause if you think it plays a role in creating the effect.
   b. **Insufficient Cause**: is the given entity sufficient to create the given effect or must it be combined with another entity? If so, add the other cause and indicate that they must occur together to cause the effect.
   c. **Predicted Effect**: can we imagine another effect caused by a given entity? If so, is this additional effect visible in the system? If it is, that strengthens the case for the existence of the entity. How could we disprove the existence of the entity? Can we perform (simulate) this test?

Example:

Clarity: "Installing is difficult" => "Installing takes more than ½ hour"
Existence: "Installing takes more than ½ hour" is easy to see. "Installing is risky" could be deduced from the number of installations that have to be redone.
Causality: "Installations have many manual steps" BECAUSE "developers don't know how to automate using scripts". Assumption: most of the steps in the installation can be automated using scripts. Verification: some applications use similar technology, yet have almost fully automated installs.
Additional Cause: "Installations have many manual steps" could also be caused by "Developers don't have the time/motivation to automate their installation".
Insufficient Cause: "Installations have many manual steps" BECAUSE "Developers don't automate them (for whatever reason)" AND "Nobody else but developers automates installs".

Predicted Effect: IF **“**Developers don’t know how to automate tasks using scripts” WE EXPECT THAT “no other tasks (e.g. builds) are automated”. Can we verify that?



Step 4: Digging deeper and pruning the tree to find the root cause

1. If an effect has multiple causes, verify the "weight" of each cause. If an effect is mostly caused by one or a few entities and rarely by other entities, prune the causes that do not contribute much to the effect. Use the 80/20 rule.
2. Dig deeper by asking WHY questions until you find one or a few entities that are responsible for causing most of the effects.
3. Take care not to create entities that are too abstract. Keep on applying the legitimate reservations.

Example:

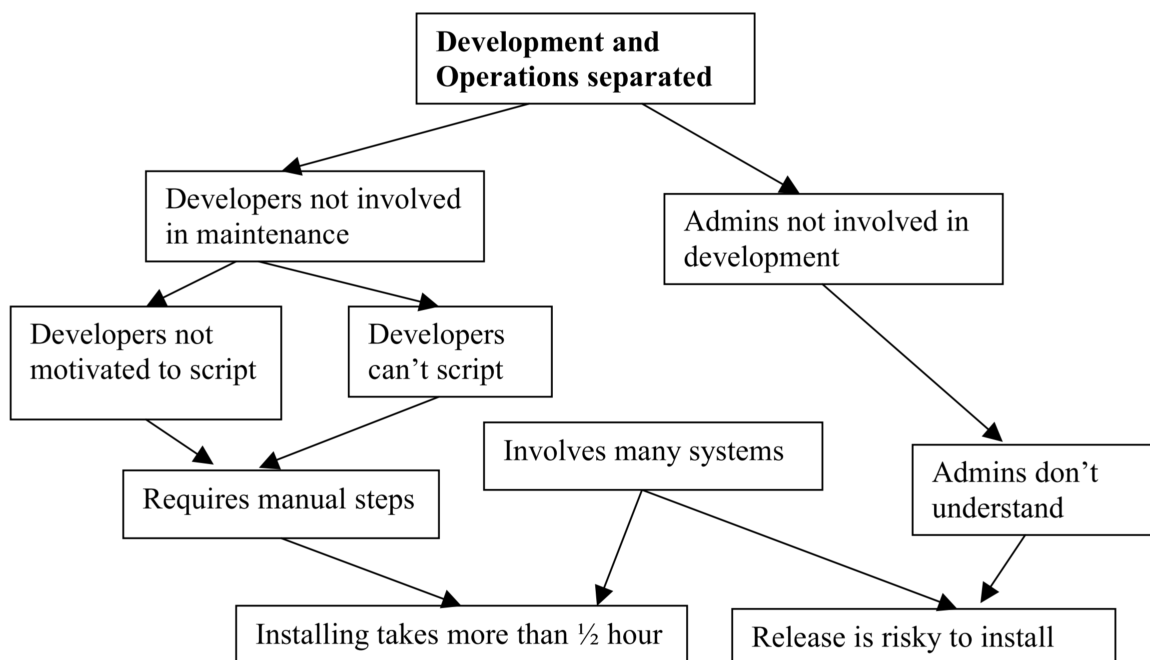
Q: "Why are installations so risky?"
A: "Because admins don't understand the applications they install and maintain well"

Q: "Why don't developers know how to automate tasks using scripts?"
A: "Because they're never involved (and don't know about) installing and maintaining servers"

Q: "Why are developers not involved?"
A: "Because the development and operational organisations are totally separate (separate management, separate budget)"

Q: "Why don't the admins understand the applications they install and maintain well?"
A: "Because they're not involved in the design, build and test of the application".
A: "AND Because the systems have many dependencies on other systems".

Q: "Why are admins not involved?"
A: "Because the development and operational organisations are totally separate (separate management, separate budget)"



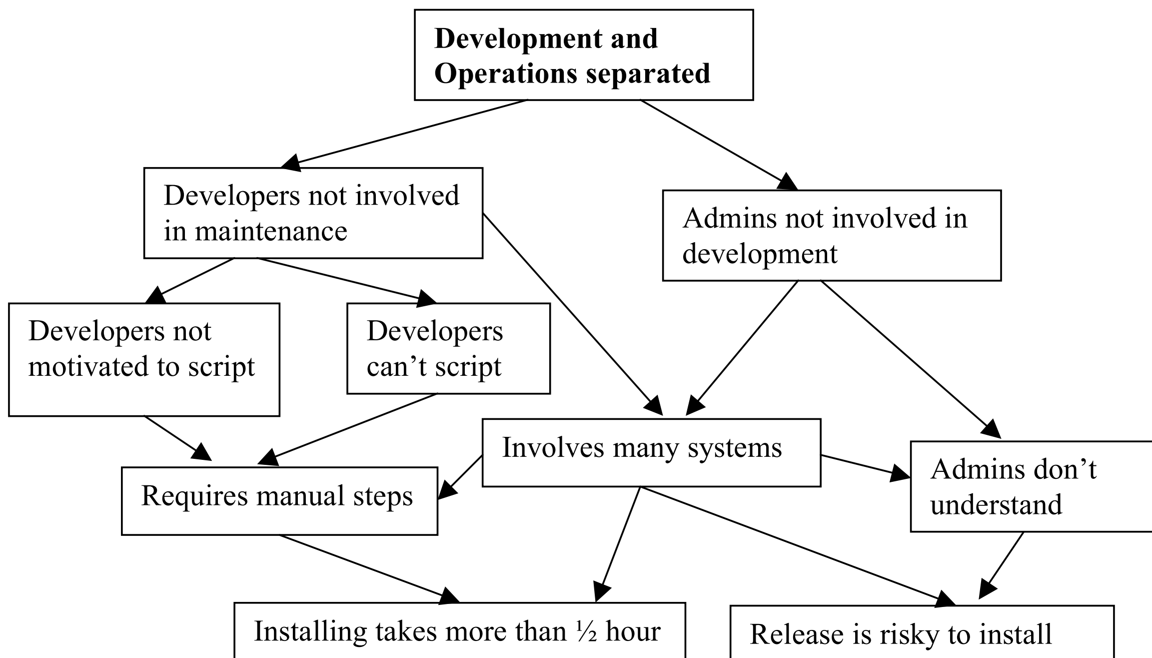
We've cleared away some entities that don't directly contribute to the problem. E.g. the predicted effect that no other tasks have been automated. This is indeed the case: teams that don't automate their install have no other automated tasks.

More importantly, we have found a core cause of many of the problems: the developers and admins are part of totally separate organisations, with separate budgets and management. Both organisations have different goals:

- The goal of the development organisation is to create valuable systems, as fast and cheap as possible. In Throughput Accounting terms: to **maximize Throughput** (business value), while **minimizing Investment**.
- The goal of the operations organisation is to keep maintenance costs as low as possible. In Throughput Accounting terms: to **minimize Operating Expense.**

If we look at the diagram again, we can see another potential root cause: the architecture of the systems is very complicated, with many dependencies. This makes the systems harder to understand and harder to automate installs (as that might involve many servers). We can tie this back to the separation of the organisations:

- As admins are not involved in architecture and design, they can't influence the architecture.
- As developers are not involved in maintenance, they don't feel the pain of keeping these complicated architectures running.



This strengthens the case against the root cause. What can we do about this problem?

Making a Future Reality Tree

Explore the intended and unintended consequences of an action.

Step 1: Start the tree with an injection and a goal

1. Create an entity that represents **the goal** you want to reach. This could be the inverse of an undesired effect or root cause from a current reality tree
2. If you have more goals, state them as entities. Don't try to reach too many goals at once!
3. Don't compromise your goals, because you think they are unattainable! We're trying to find out if and how they can be attained. Don't admit defeat before you start.
4. Brainstorm a few actions you could take to achieve the goal(s).
5. Select the most promising action and create an entity that represents it. Write the entity as a simple sentence. This will help you imagine that you have already taken the action, so that you can explore its consequences. This entity is called the **injection**.
6. Put the **injection** entity at the bottom of the diagram
7. Put the **goal** entity (entities) at the top of the diagram.

Tip: write using present tense and don't use tentative phrasing (maybe, might, possibly…), this will help you imagine the future.

Example:

Goals:

Let's try do something about the problem described above. What are our goals?

- Releases are installed reliably, first time, each time.
- Installing a release takes less than ½ hour.

These are just the undesirable effects from the CRT, reversed.

Injection(s):

How can we bring about these goals? We can't do anything about the root cause (yet), because the way the company is organized is not something we can change (quickly). But… could we do something to involve developers in maintenance and admins in development?

I propose two actions:

- Developer and admin pair-install the system
- Admins review the architecture of the application

Release works

Release is fast

Pair install

Architecture review

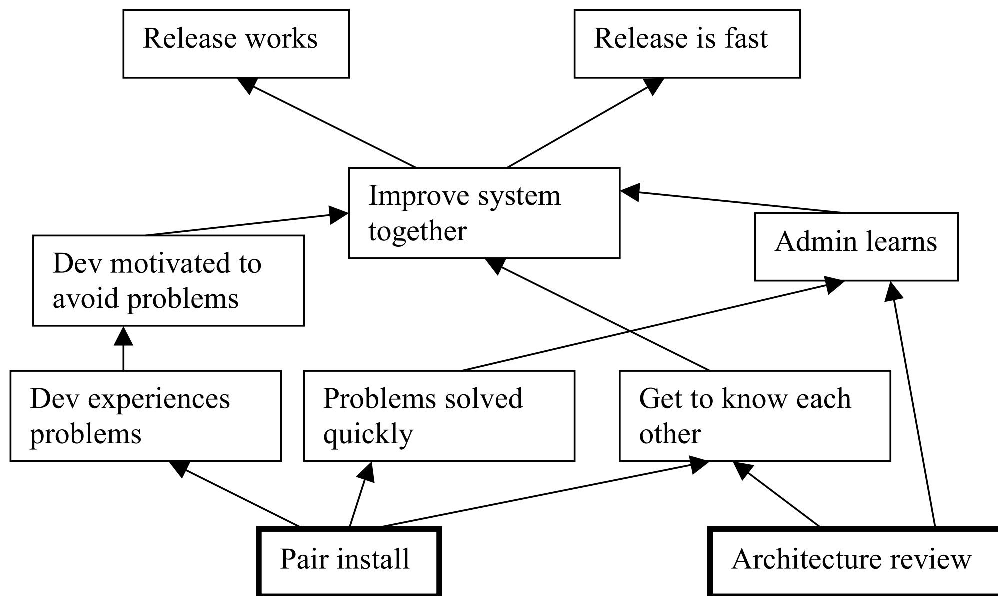
Step 2: List consequences of the action

1. Starting with the action, list the effects it has (remember, think and write in present tense).
2. Apply the categories of legitimate reservations after adding a few entities and relations.
3. If any of the effects are negative or undesirable, or if someone starts to raise objections, stop and examine the diagram:
    a. **"You can't do that!"** If an action has a desirable effect, but someone thinks it's impossible to perform that action, don't argue. **You have discovered an obstacle**.
    b. **"You don't want that to happen!"** If an action has an undesirable (side) effect, don't argue. **You have discovered a negative branch reservation**.
4. Note the obstacles and negative branch reservations; we'll revisit them in the next step.
5. If you get stuck reasoning forward from the actions to the goal(s), try to reason backwards from the goals and vice versa.

Examples:

What are some inferences we can draw from the actions?

- IF admin/developer pair-install THEN developer experiences installation problems firsthand"
- IF developers experiences installation problems THEN developer is motivated to avoid these problems
- IF admin/developer pair-install THEN installation problems get resolved quickly, because developer knows application well
- IF problems get solved during install THEN admin learns about the system
- IF developer and admin pair-install THEN they get to know each other
- IF developer and admin know each other THEN they work together to improve the system
- IF developer and admin work together to improve the system AND admin learns about the system AND developer is motivated to avoid installation problems THEN they will make next release's installation by automating more, by making the system simpler or by reducing configuration needs.
- IF developer and admin perform architecture reviews THEN they get to know each other AND the admin learns more about the system AND they can improve the system together.


Release works
Release is fast
Improve system together
Admin learns
Dev motivated to avoid problems
Dev experiences problems
Problems solved quickly
Get to know each other
Pair install
Architecture review

Step 3: Yeah, but… Dealing with obstacles and negative branch reservations

1. Dealing with **obstacles**: examine the reasoning behind the obstacle. Is there some other action you could take to remove the obstacle? If yes, add it to the diagram as an additional cause for the effect and note the assumption that this action removes the obstacle. If you see no immediate way to remove the obstacle, note the obstacle. You can try to apply a **Prerequisite Tree** to remove the obstacle.
2. Dealing with **negative branch reservations**: examine the reasoning leading to the negative effect using the legitimate reservations.
   a. If the effect requires more than one cause, is there a way to remove one of the causes by taking some action? If so, add the action and remove the unintended effect. Note the assumption that taking this action removes the cause and thus the effect.
   b. Add the opposite of the negative effect as a goal. Use the same techniques as for the other goals to find actions that bring about this goal. If you succeed in reaching the goal, you can leave off the undesirable effect. Add the new action as a prerequisite to the intended effects of the action that caused the undesirable effect you removed.

Examples:

Undesirable effects:

1. If developers are involved in the installation, there will be even more hacks than before during installations. **Installations will become even less repeatable. You don't want that**.
2. If developers are involved in the installation, the **developer spends time** doing (unplanned) work that's not in their job description. **You don't want that**.

Obstacles:

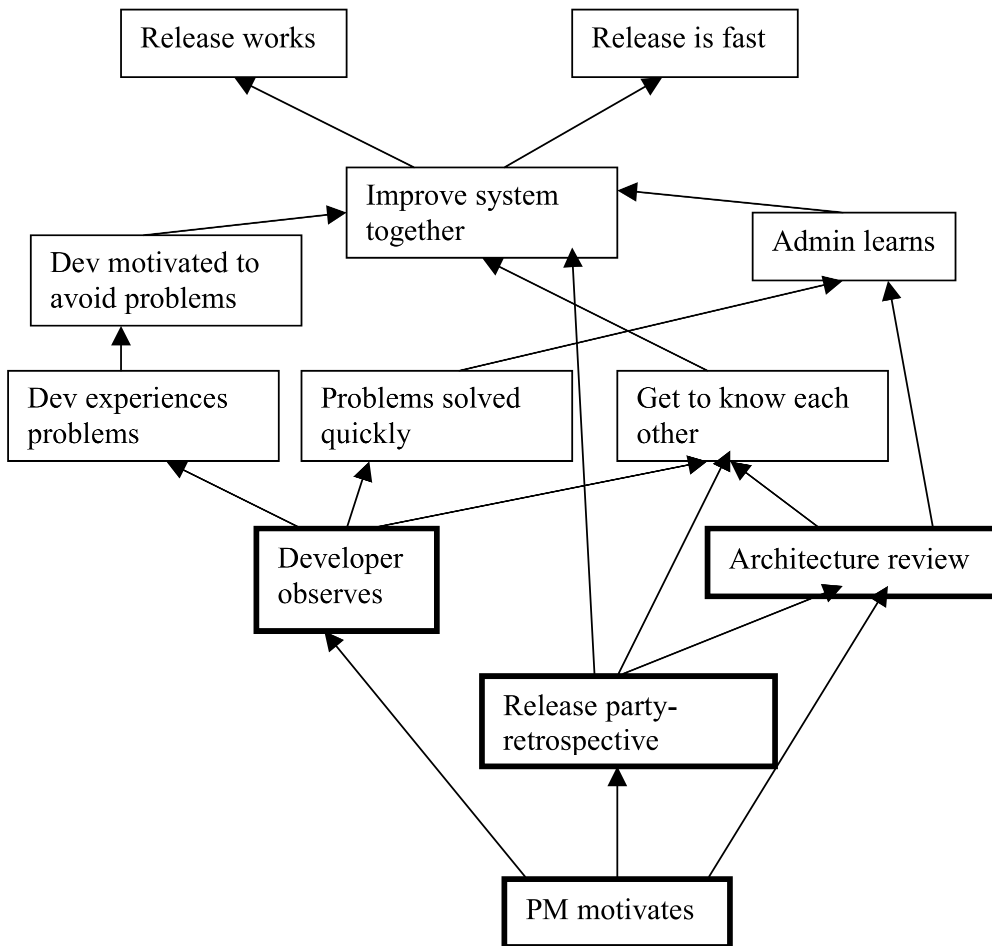
1. "**You can't pair install**", production servers are off-limits for developers, for obvious security and privacy reasons.
2. **"Developers and admins aren't motivated to work together"**. Because the two organisations are separated, there's an "over the wall" culture.

Resolving the objections:

- **Obstacle 1** can be resolved by changing the role of the developer: they are "observers". The observer responds to questions of the admin and notes where the installation instructions are unclear. In both cases, the observer then updates the installation document. => **Change injection "Pair install" to "Developer observes admin".**
- The previous change would also remove **Undesirable effect 1**: with this feedback, the installation document will become clearer and hacks will be required less often.
- To remove **Undesirable effect 2**, the PM would have to put this installation time in the plan. But even if he doesn't, the developers are always idle between two releases, so there's no real time loss. If our releases become faster, developers and admins have more time.

- To remove **Obstacle 2**, the PM would have to motivate or tell developers and admins to work together. That's feasible for the developers, but not the admins. A PM has no authority over people in other teams and organisations. There are two ways the PM can motivate developers and admins:
  1. Involve admins from the start of the project, so that they know what they're working on and their input is valued
  2. Throw a small release party to celebrate the successful release. Use the relaxed atmosphere to perform an informal retrospective, to improve the next installation of the release.

If we perform these actions, the tree looks like this.



Step 4: Getting to the goal and stabilizing with reinforcing loops

1. Keep on applying the steps above to get to the goal(s) you set.
2. You might have to backtrack, remove actions, add other actions.
3. When you reach the goal, try to find a reinforcing loop. A reinforcing loop is a causal relation from an entity high up in the tree (near or at the goal) to an entity lower in the tree (nearer the actions you want to take).
    a. Examine the entities from top to bottom, starting with the goal
    b. Check if this entity could cause an entity lower down, from bottom to top.
    c. Apply the legitimate reservations if you find a candidate relation.
4. Reinforcing loops can help keep a goal "alive" by reinforcing the actions that bring about the goal. However, too much of a good thing can be bad: be aware of possible negative effects from repeatedly performing an action or strengthening a goal.

Examples:

It's clear that there has to be a working release to have a party. Therefore, the release party should be at the top, caused by the working release. This is already a high up cause that has a low down effect. That's great to keep people motivated.
Still, how do we get the system started? The PM has to motivate developers and admins somehow. Involving admins early in the process, so that they really feel part of the team is a good way to do that.

Release party-retrospective
Release works
Release is fast
Improve system together
Admin learns
Dev motivated to avoid problems
Dev experiences problems
Problems solved quickly
Get to know each other
Developer observes
Architecture review
PM motivates

What we're doing here is to create a **"cross-functional virtual team"**. Some team members are permanent, like the developers of the project. Other members are part of the team for (part of) this release only, like admins or developers of other impacted projects. These people are part of many virtual teams.
The PM has no formal authority over them, so they have to motivate those people to want to work for a team. It's very important that every member has a clear view of the goal and knows how they participated in bringing this goal about.
By creating these virtual teams, we are dealing with the root cause of the problems: "development and operations are separated". We are in effect creating a matrix structure, which keeps the good parts of the separation (clear roles, security and privacy), but removes the bad results ("over the wall" mentality, poor knowledge and attention to detail by developers about installation and maintenance). Even though we can't do anything about the root cause, we can do something about it.

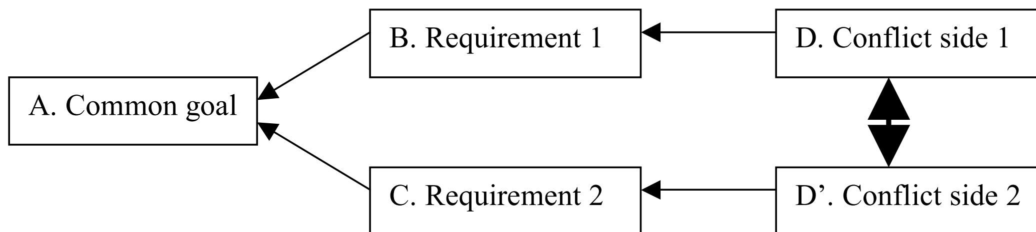
There is one dangerous point in this diagram: it's up to the PM to get this system started and to keep it going (with the help of the release party). What if this injection falls away? One way of dealing with this, would be to encode the other injections ("developer observes", "architecture review" and "release party-retrospective") in the standard process of the organisation. That would require some injections to spread the idea and to get it started. But afterwards, we hope the system becomes self-sustaining.

The evaporating cloud

Examine the reasoning behind two conflicting statements

Step 1: Articulate the problem: where's the conflict?

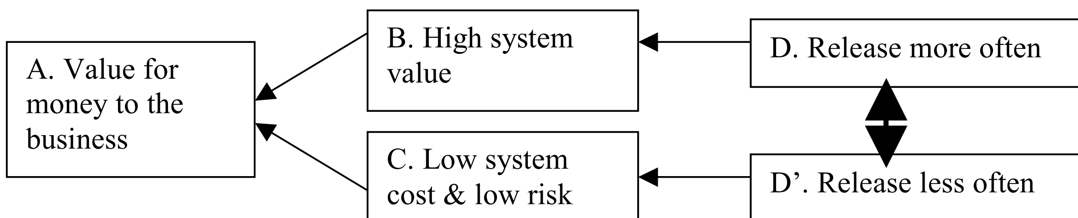
1. Describe the system and its goal, if you haven't already.
2. Are you sure you want to solve this problem?
3. State the two sides of the conflict as entities (D and D')
4. State the goal of the system as an entity (A)
5. Add an entity B, so that: in order to achieve A, we need B. In order to achieve B we need D.
6. Add an entity C, so that: in order to achieve A, we need C. In order to achieve C we need D'.
7. You should have a diagram like this:



You should be able to read the diagram out loud like:
"In order to have A, B must exist. We also need C in order to have A.
We can't get B, unless we have D. We must have D' in order to have C.
D and D' are mutually exclusive, they cannot coexist."

Example:

This is the IT organisation of a large company. Developers want to release more often to bring value sooner and to reduce risk. Admins want to install fewer releases to reduce costs and to reduce risk. Both of these departments together want to provide systems that provide the best value for the lowest cost to the business.



"In order to provide good value for money to the business, developers must provide systems that provide high value at low risk (B->A). The admins must also ensure that these systems are installed and maintained at low cost and low risk (C->A). In order to have higher value and lower risk, developers need to release smaller releases, more often (D->B). In order to lower maintenance and installation costs and risk, admins need to make fewer changes to the systems (D'->C). Releasing more often AND less often is mutually exclusive, they cannot co-exist" (D<->D').

Step 2: Examine the diagram with the legitimate reservations

1. Does the diagram satisfy the level 1 reservations:
   - Clarity ?
   - Entity existence ?
   - Causality existence ?
2. Does the diagram satisfy the level 2 reservations:
   - Additional cause ?
   - Sufficient cause ?
   - Predicted effect ?
3. Take note of any assumptions
4. Are D and D’ really mutually exclusive?
   - Why can’t D and D’ co-exist? Note any interesting assumptions.
   - Why aren’t we allowed to have D and D’? Note the assumption.
   - Is there any overlap between D and D’? If so, can you separate them more cleanly, while holding on to the common part?

Example:

Clarity: what does release more often/less often mean? Typical projects now take around 6 months. Developers would like to release every 2 months.

Entity existence:

Causality existence:

- Does releasing more frequently increase value? Yes: business people have been asking for shorter releases, to be able to react faster to the competition.
- Does releasing less frequently reduce cost? Yes: systems admins have to spend time planning and executing the change. There are often problems during or shortly after a release, so that admins have to perform emergency fixes.
- Does releasing less frequently reduce risk? Yes: if you leave the systems alone, you don’t risk downtime or regression problems.

Additional cause:

- Is there another way to deliver value sooner, without releasing more often? We could make the system more configurable by users, so that they could make more changes without involving IT. But this is insufficient to be able to support all the features in the new releases.
- Is there another way to reduce risk and cost of installations, except not releasing? **Maybe….**

Sufficient cause: does releasing often suffice to create value? No: we must also ensure that the release contains high value features and that they work. Let’s assume this is the case.

Predicted effect: can we disprove “releasing less often reduces cost and risk”. Yes, if we can find projects that release often, yet are not costly or risky to install. Is this the case? Yes, there are one or two such projects. We should examine what they do differently. Why is it that **most** projects are risky and costly to install? We can examine this problem using a Current Reality Tree (see start of document). If we can find a way to make releases cheap and safe to install, we can remove D’, thus resolving the conflict.

Bibliography

The Goal: A Process of Ongoing Improvement, Elyahu Goldratt (ISBN: 0566086654)
It's Not Luck – Elyahu M. Goldratt (ISBN: 0566076276)
Thinking for a Change: Putting the TOC Thinking Processes to Use – Lisa M. Scheinkopf (ISBN: 1574441019)
A Guide to Implementing the Theory of Constraints – http://www.dbrmfg.co.nz/
The Toyota Way: 14 Management Principles From The World's Greatest Manufacturer – Jeffrey K. Liker (ISBN: 0071392319)

For more books about the subject, see:
http://wiki.systemsthinking.net/Systemsthinking/BookList.html

Thank you for participating in this session.

Marc Evers
Piecemeal Growth
The Netherlands
http://www.piecemealgrowth.net
marc@piecemealgrowth.net

Pascal Van Cauwenberghe
Nayima
Belgium
http://www.nayima.be
pvc@nayima.be