

# Setting Direction for a Re-Engineering Project

---

# Forces affecting Reengineering Project

Burdened with a lot of interests pulling in different directions; political, technical, economic, management issues.

Complicated communication

The legacy system will pull you towards a certain architecture that may not be the best for the future of the system.

You will detect many problems with the legacy software, and it will be hard to set priorities.

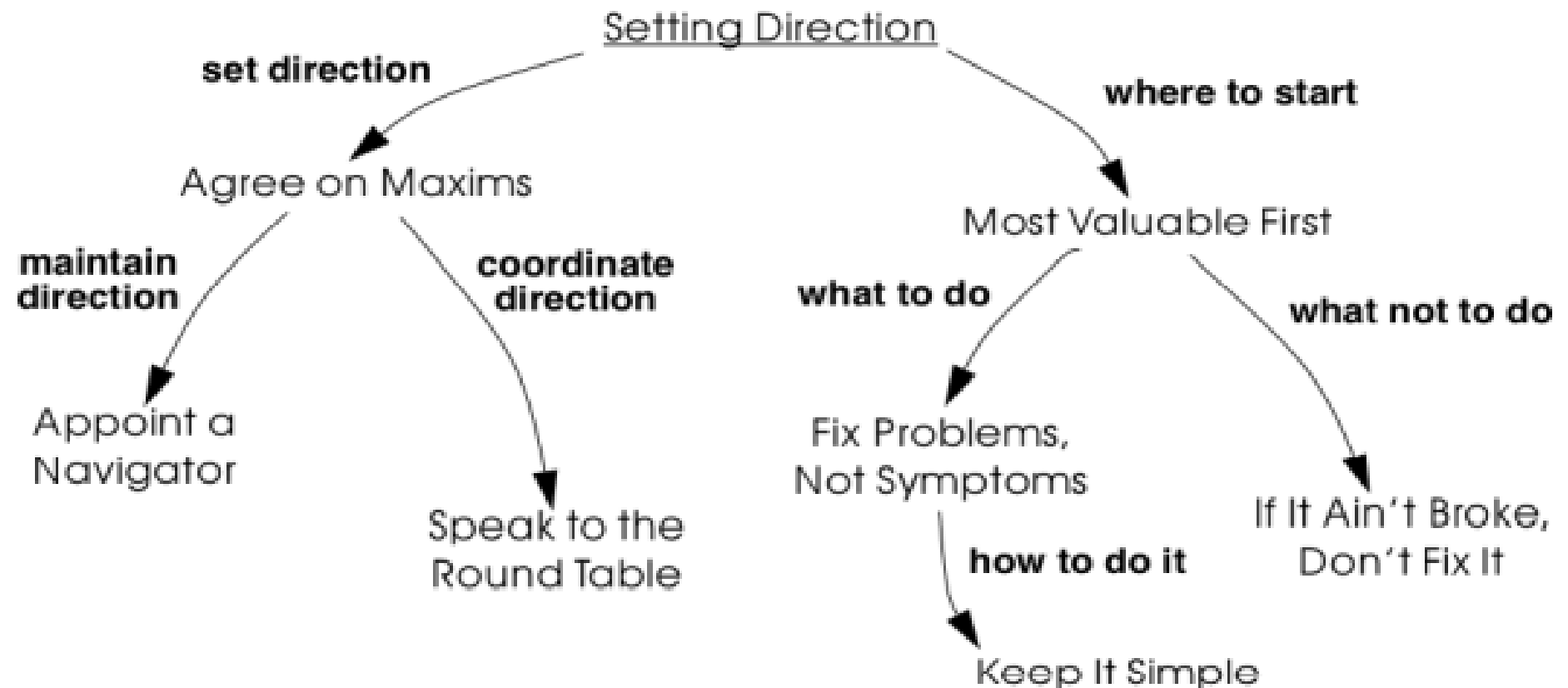
It is easy to get focused on the technical problems that interest you the most, rather than what is best for the project.

# Forces affecting Reengineering Project

It can be difficult to decide whether to wrap, refactor or rewrite a problematic component of a legacy system.

Over engineering problem.

# Setting Direction in Reengineering Project



# Agree on Maxims

## **Problem**

How do you establish a common sense of purpose in a team?

## **Solution**

Establish the key priorities for the project and identify guiding principles that will help the team to stay on track.

Answer various maxims:

Everyone is responsible for testing and debugging.

You cannot do it right the first time.

# Appoint a Navigator

## **Problem**

How do you maintain architectural vision during the course of complex project?

## **Solution**

Appoint a specific person whose responsibility in role of navigator is to ensure that the architectural vision is maintained.

# Speak to the Round Table

## **Problem**

How do you keep your team synchronized?

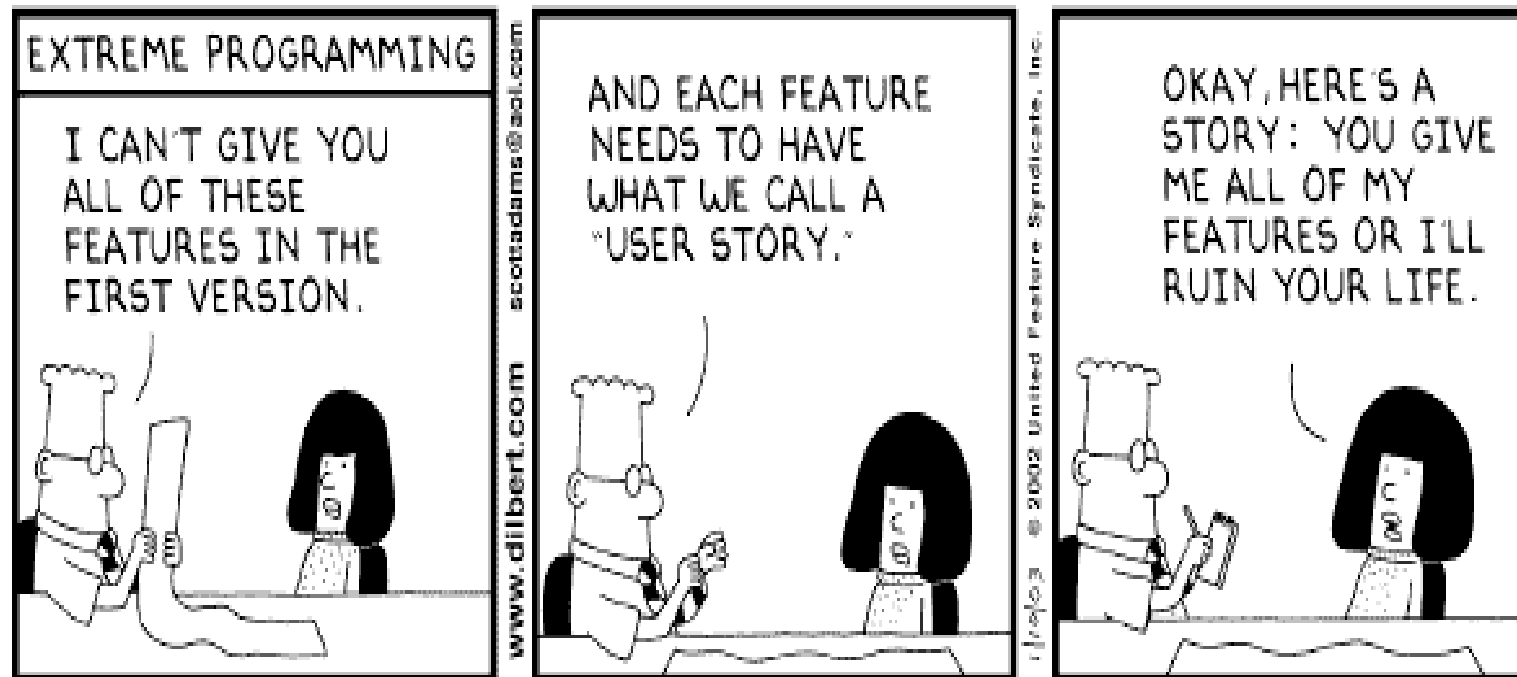
## **Solution**

Hold brief, regular round table meetings at least once a week.

- what they have done since the last meeting?
- what they have learned?
- what problems they have encountered?
- what they plan to do until the next meeting?

Record only decisions and actions to be taken.  
Stand up meetings.

# Most Valuable First



Copyright © 2003 United Feature Syndicate, Inc.



# Most Valuable First

## **Problem**

Which problems should you focus on first?

## **Solution**

Start working on the aspects which are most valuable to your customer.

By focusing on the most valuable parts first,  
you increase the chance that you will identify the right issues at stake,  
and that you will be able to test early in the project the most important decisions,  
such as which architecture to migrate to,  
or what kind of flexibility to build into the new system.

# Most Valuable First

How do you tell what is valuable?

- It can be difficult to assess exactly what is the most valuable aspect for a customer.
- Try to understand the customer's business model. This will tell you how to assess the value of the various aspects of the system.
- Try to determine what measurable goal the customer wants to obtain. For example, better response time, faster time to market of new features, easier tailoring to individual clients needs
- Try to understand whether the primary goal is mainly to protect an existing asset, or rather to add value in terms of new features or capabilities.

# Most Valuable First

How do you tell what is valuable?

- Examine change logs.
- Play the Planning Game.

# Most Valuable First



# Most Valuable First

What if the most valuable part is a rat's nest?

- Renovate the worst first.
- Determine whether to wrap, refactor or rewrite the problematic component by making sure you Fix Problems, Not Symptom.
- Involve users – Build confidence – Continuous feedback

# Fix Problems, Not Symptoms

## **Problem**

How can you possibly tackle all the reported problems?

## **Solution**

Address the source of a problem, rather than particular requests of your stakeholders

# Fix Problems, Not Symptoms

## Example 1

If the code of a legacy component is basically stable, and problems mainly occur with changes to clients, then the problem is likely to be with the interface to the legacy component, rather than its implementation, no matter how nasty the code is. In such a case, you should consider applying **Present the Right Interface** to just fix the interface.

## Example 2

If the legacy component is largely defect-free, but is a major bottleneck for changes to the system, then it should probably be refactored to limit the effect of future changes. You might consider applying **Split Up God Class** to migrate towards a cleaner design.

# Fix Problems, Not Symptoms

## Example 3

If the legacy component suffers from large numbers of defects, consider applying **Make a Bridge to the New Town** as a strategy for migrating legacy data to the new implementation.



# If It Ain't Broke, Don't Fix It

## **Problem**

Which parts of a legacy system should you reengineer and which should you leave as they are?

## **Solution**

Only fix the parts that are “broken” — those that can no longer be adapted to planned changes.

- components that need to be frequently adapted to meet new requirements, but are difficult to modify due to high complexity and design drift
- components that are valuable, but traditionally contain a large number of defects.

# Keep it Simple

## **Problem**

How much flexibility should you try to build into the new system?


## **Solution**

Prefer an adequate, but simple solution to a potentially more general, but complex solution.


Do the simplest thing that will work.



**Think Carefully**



You are part of a team developing a software system named proDoc which supports doctors in their da The main functional requirements concern (i) maintaining patient files and (ii) keeping track of the money to be paid by patients and health insurances. The health care legislation in Switzerland is quite complicated and changes regularly, hence there are few competitors to worry about. Nevertheless, a fresh start-up company has recently acquired considerable market-share with a competing product named XDoctor. The selling features of XDoctor are its platform independency and its integration with the internet. The system offers a built-in e-mail client and web-browser. XDoctor also exploits the internet for the transaction processing with the health insurances.



To ensure its position in the market, your company has purchased XDoctor and now wants to recover as much as possible from the deal. In particular, they want to lift the internet functionality out of XDoctor to reuse it into proDoc. You are asked to make a first evaluation and develop a plan on how to merge the two products into one. At the outset, there is very little known about the technical details of the competing product. From the original development team of four persons, only one has joined your company. His name is Dave and he has brought a large box to your office containing lots of paper (the documentation?) and two CDs. The first is the XDoctor installation disk containing an installer for Windows, MacOS and Linux. The other contains about 500,000 lines of Java code and another 10,000 lines of C code. Looking kind of desperately at this box sitting on your desk, you're wondering "Where on earth do I start?"



The End