Object Oriented Reengineering

Olivier Liechti
HEIG-VD
olivier.liechti@heig-vd.ch

MSE | MASTER OF SCIENCE IN ENGINEERING

License

- These slides are based on a book authored by Stéphane Ducasse, Serge Demeyer and Oscar Nierstrasz.
- http://scg.unibe.ch/download/oorp/

Object-Oriented Reengineering Patterns

http://scg.unibe.ch/download/oorp/

Dealing with Legacy Systems

New or changing requirements will gradually degrade original design…
…unless extra development effort is spent to adapt the structure.

New Functionality

Quick Hack?

Remember Lehmans's laws of Software Evolution?

- Duplicated code
- Complex conditionals
- Abusive inheritance
- Large classes and methods

*Take a **loan** on your software*

- Refactor
- Restructure
- Reengineer

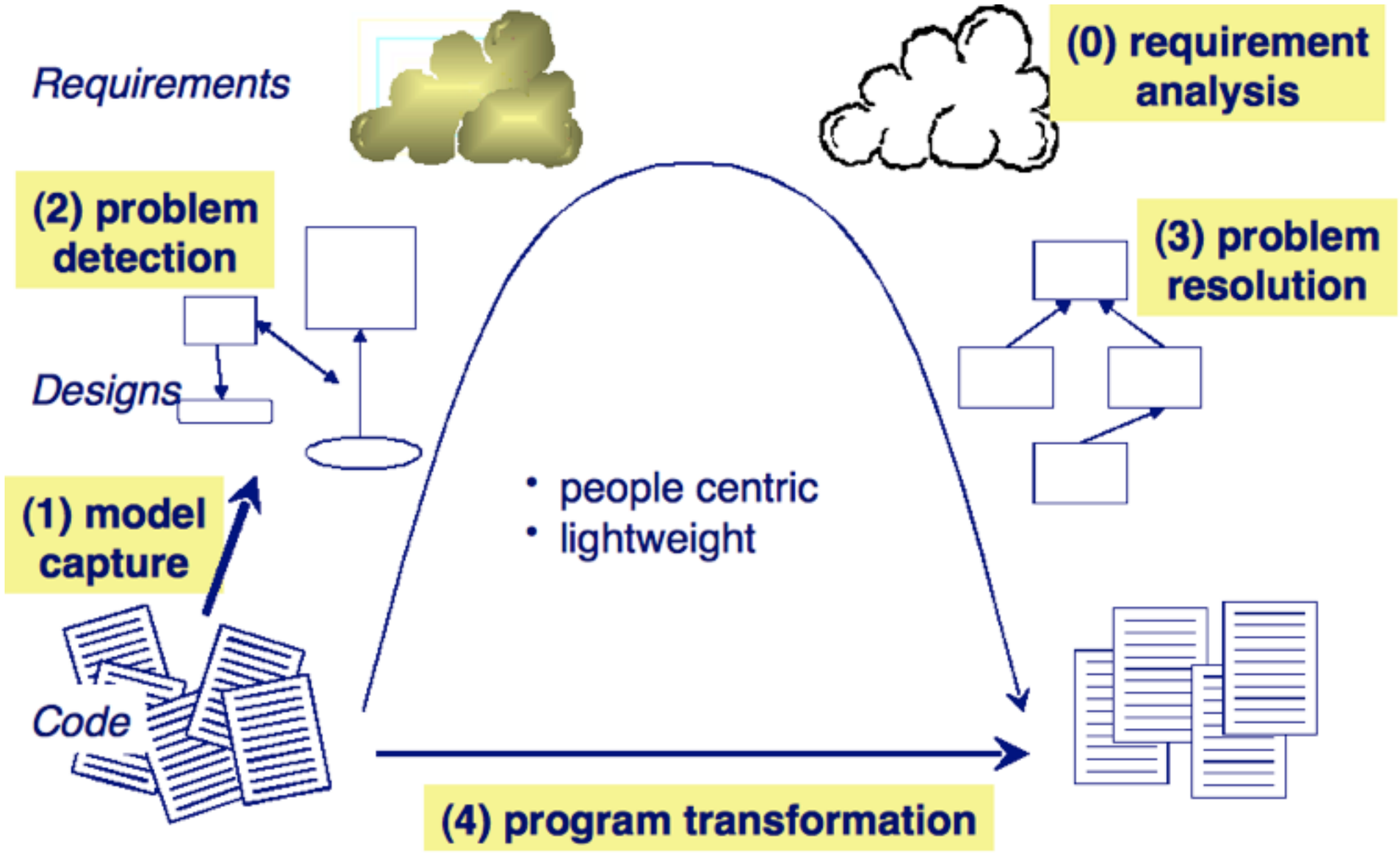
***Invest** for the future*

Terminology

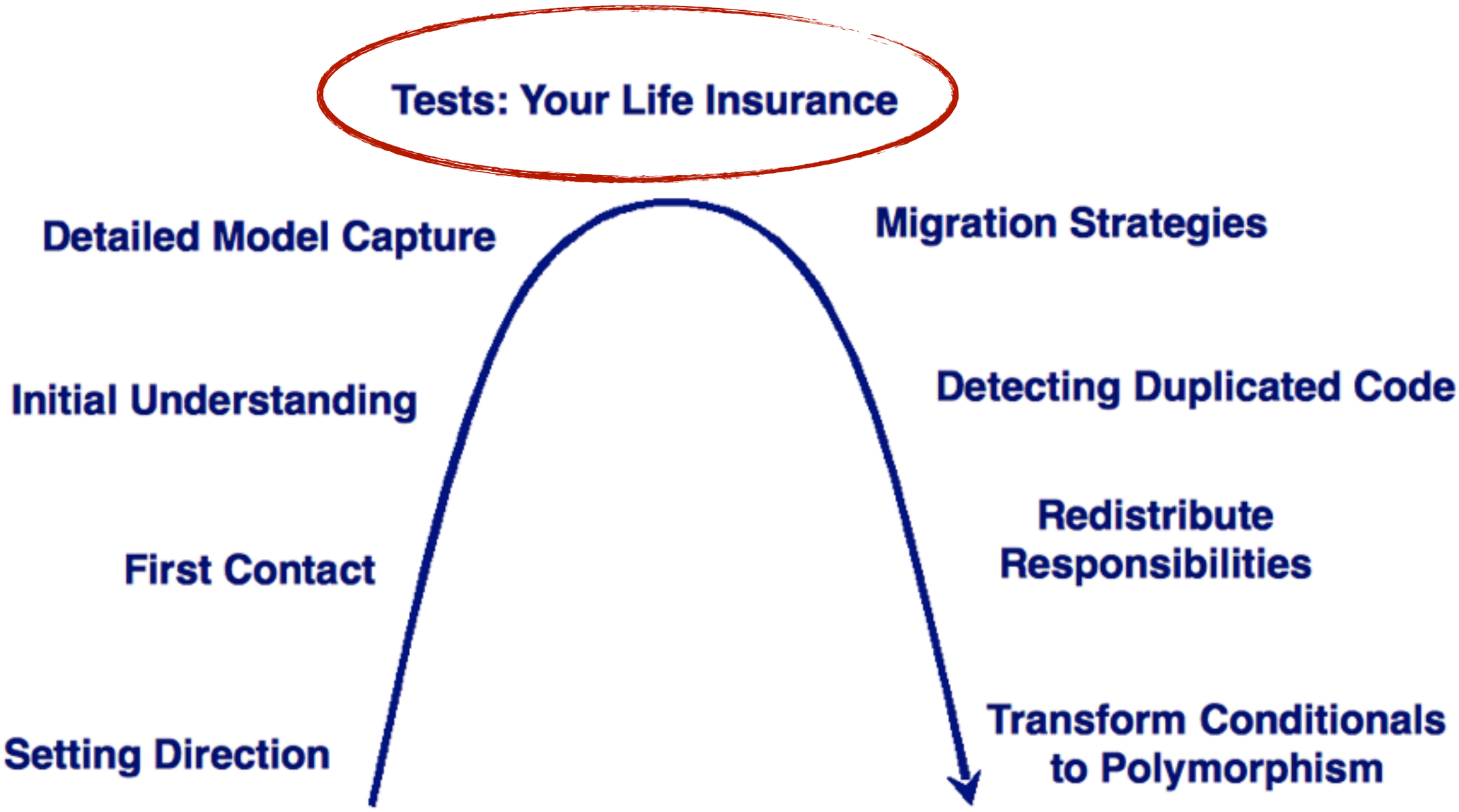
- **Forward Engineering**
  - traditional process of moving **from high-level abstractions** and logical, implementation-independent designs **to the physical implementation** of a system.
- **Reverse Engineering**
  - process of analyzing a subject system to identify the system's components and their interrelationships and **create representations of the system** in another form or at a higher level of abstraction.
- **Reengineering**
  - examination and alteration of a subject system to **reconstitute it in a new form** and the subsequent implementation of the new form.

Chikofsky and Cross, 1993

The Reengineering Life-Cycle

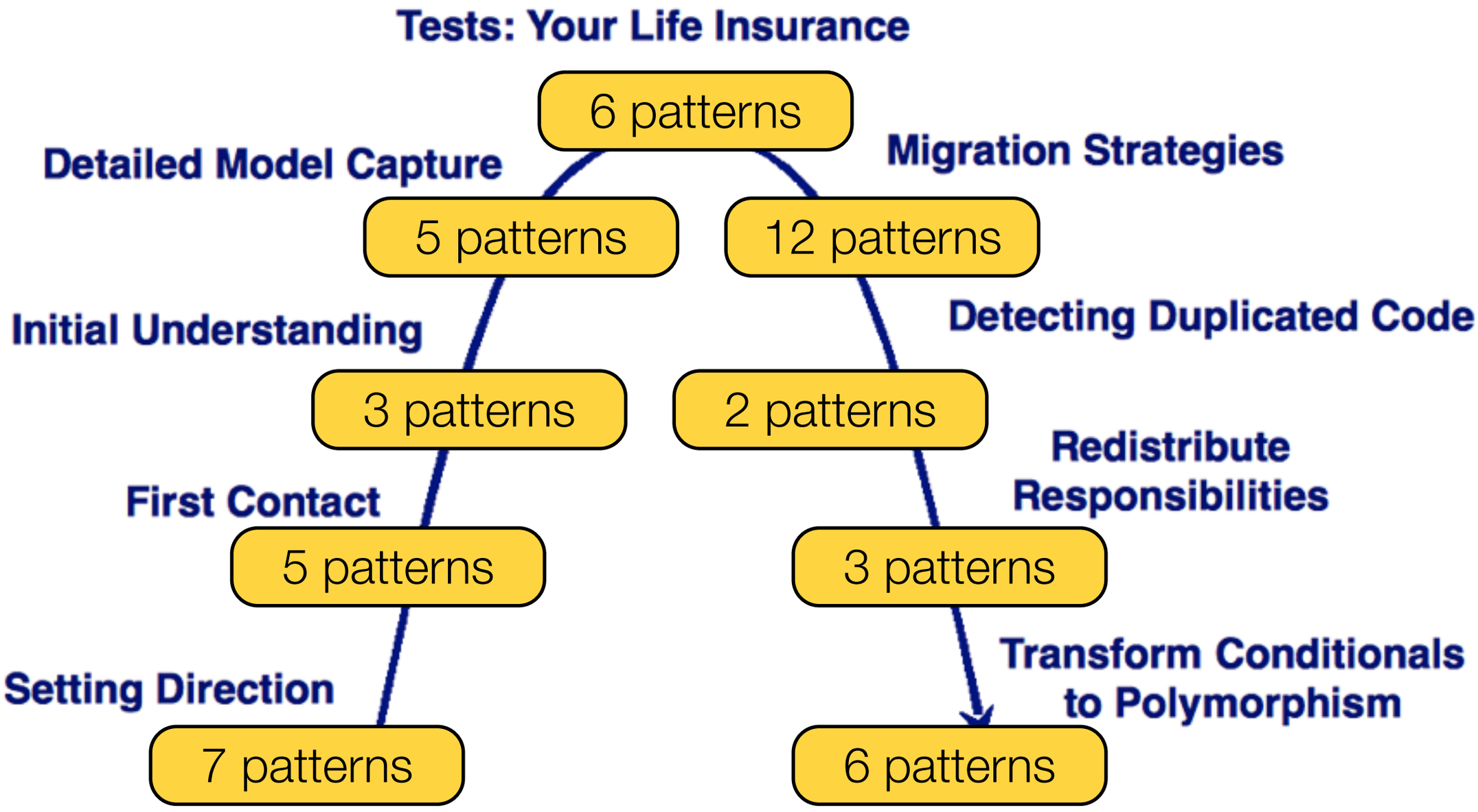


A Map of Reengineering Patterns



Based on https://www.iam.unibe.ch/scg/svn_repos/Lectures/OORPT/

A Map of Reengineering Patterns



Based on https://www.iam.unibe.ch/scg/svn_repos/Lectures/OORPT/

MSE | MASTER OF SCIENCE IN ENGINEERING

Reverse Engineering Patterns

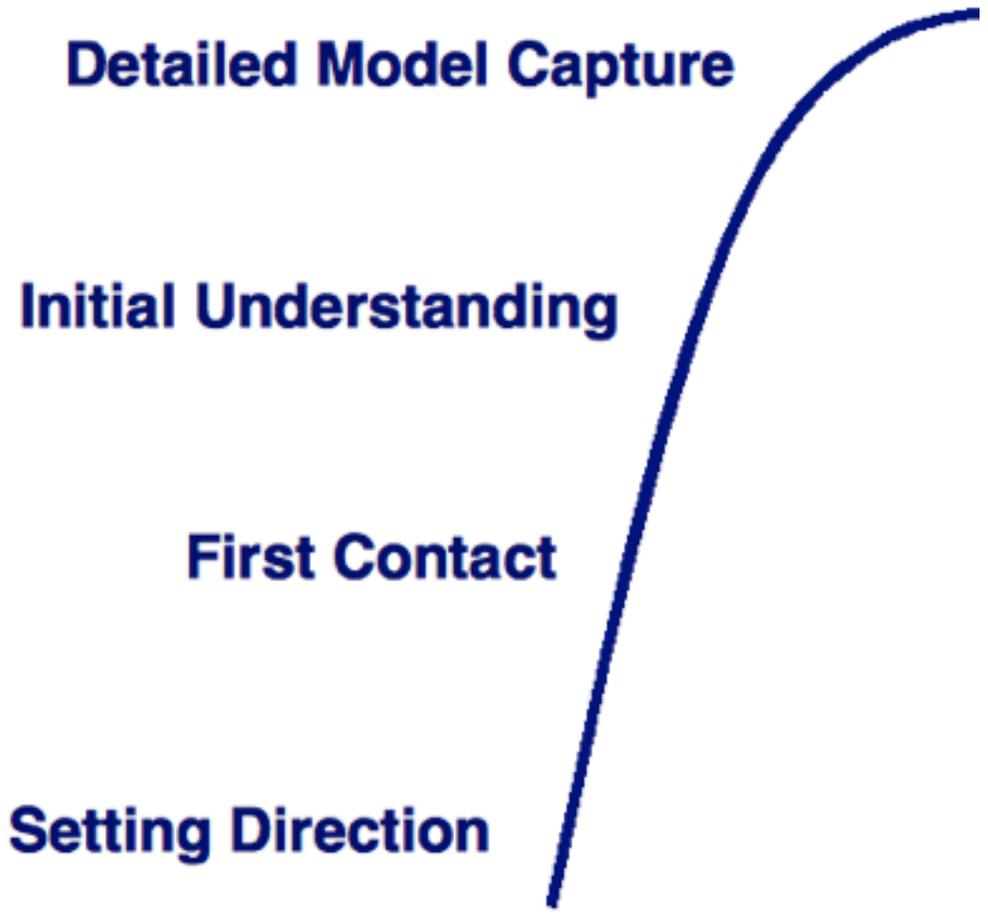
Reverse engineering patterns encode expertise and trade-offs in ***extracting design*** *from* ***source code***, ***running systems*** *and* ***people***.

Mission Impossible

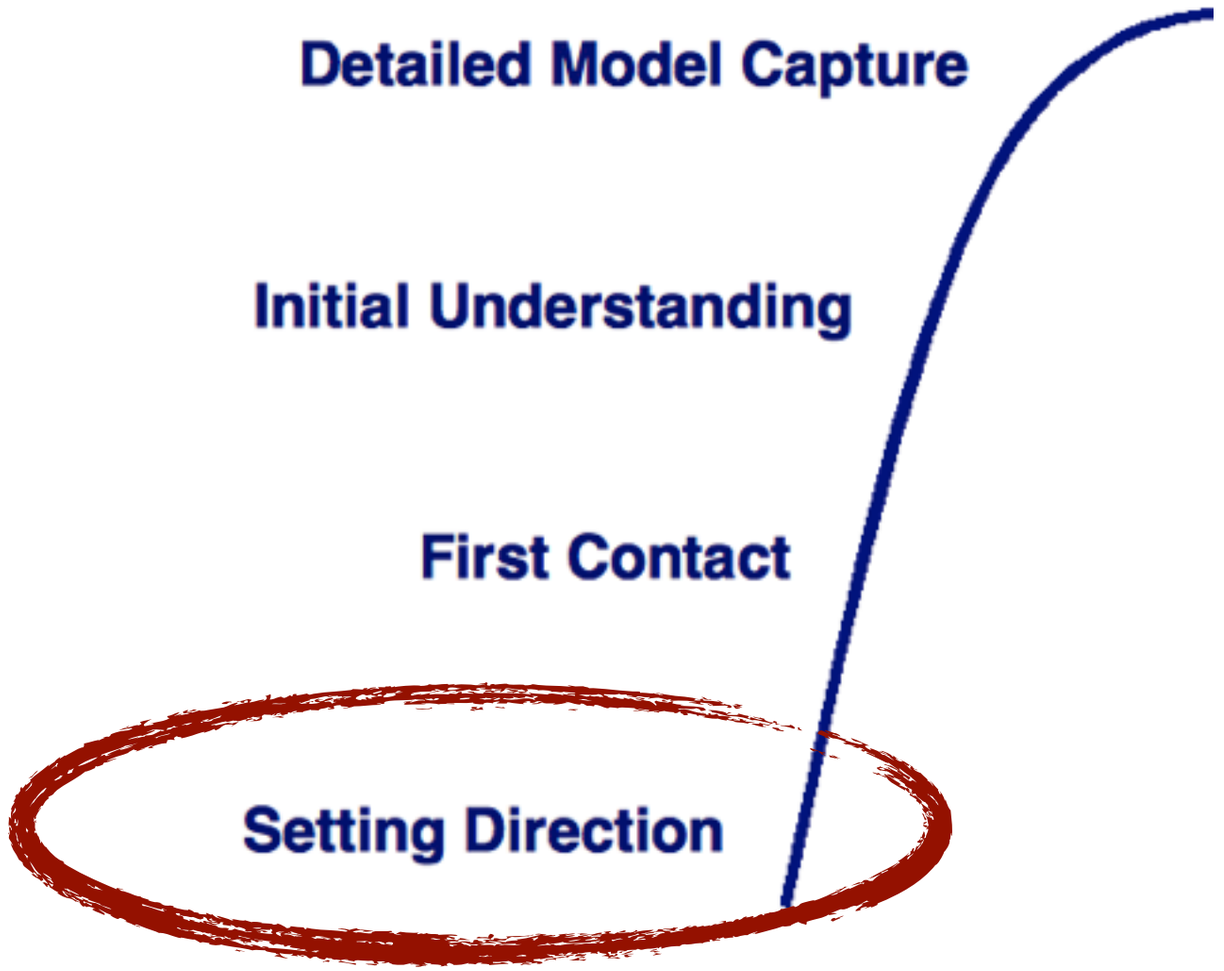
- **“Allo, PK Consulting”?**
  - “Yes”
- **“We have got a small problem with our fancy social networking site...”**
  - “Yes?”
- **“As soon as there are 20 concurrent users, it implodes...”**
  - “Yes...”
- **“Can you help us?”**
  - “Yes.”

Reverse Engineering Patterns

- When you start a reengineering project, you will be pulled in **many different directions**, by management, by the users, by your own team.
- How do you **set the direction** of the reengineering effort, and how do you **maintain direction** once you have started?



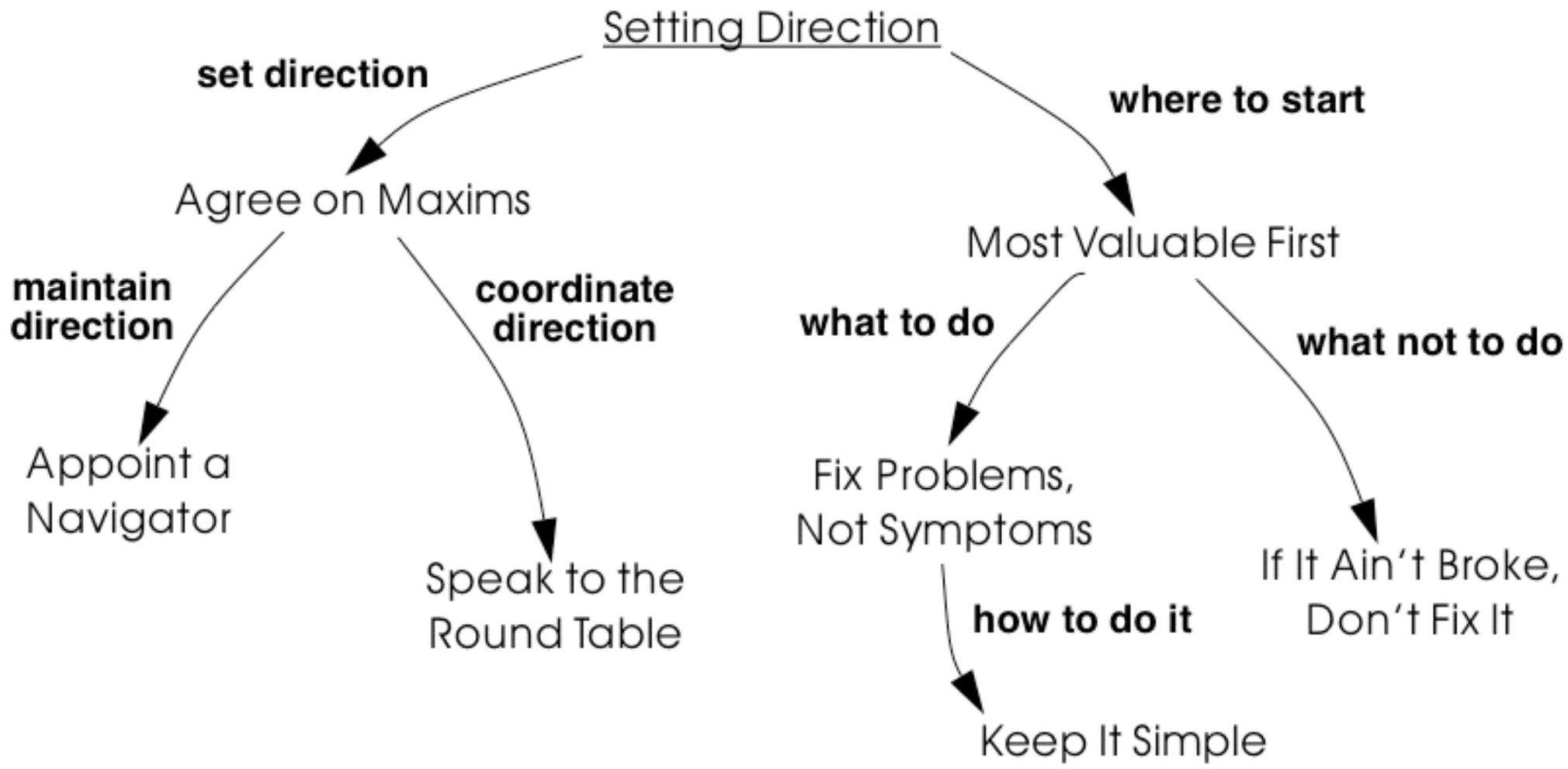
Setting Direction



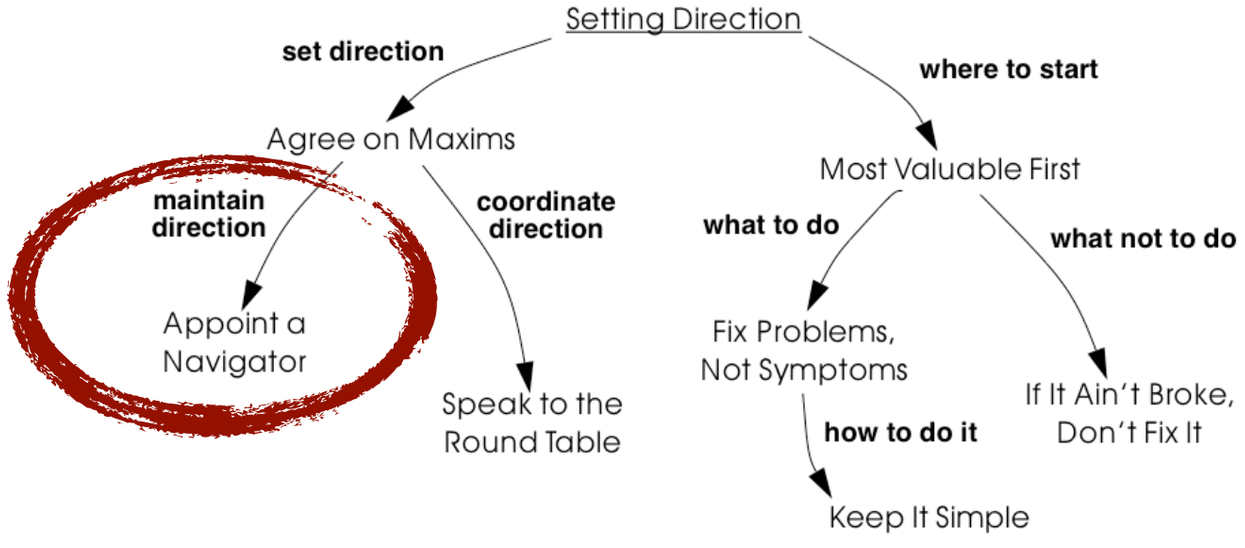
Setting Directions: Forces

- A typical reengineering project will be burdened with a lot of interests that pull in **different directions**.
- Technical, ergonomic, economic and political considerations will make it difficult for you and your team to **establish and maintain focus**.
- Communication in a reengineering project can be complicated by either the presence or absence of the **original development team**.
- 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 seduced** by focussing on the technical problems that interest you the most, rather than what is best for the project.
- It can be difficult to decide **whether to wrap, refactor or rewrite** a problematic component of a legacy system. Each of these options will address different risks, and will have different consequences for the effort required, the speed with which results can be evaluated, and the kinds of changes that can be accommodated in the future.
- When you are reengineering the system, you **may be tempted to over-engineer** the new solution to deal with every possible eventuality.

Setting Directions: Patterns



Appoint a Navigator

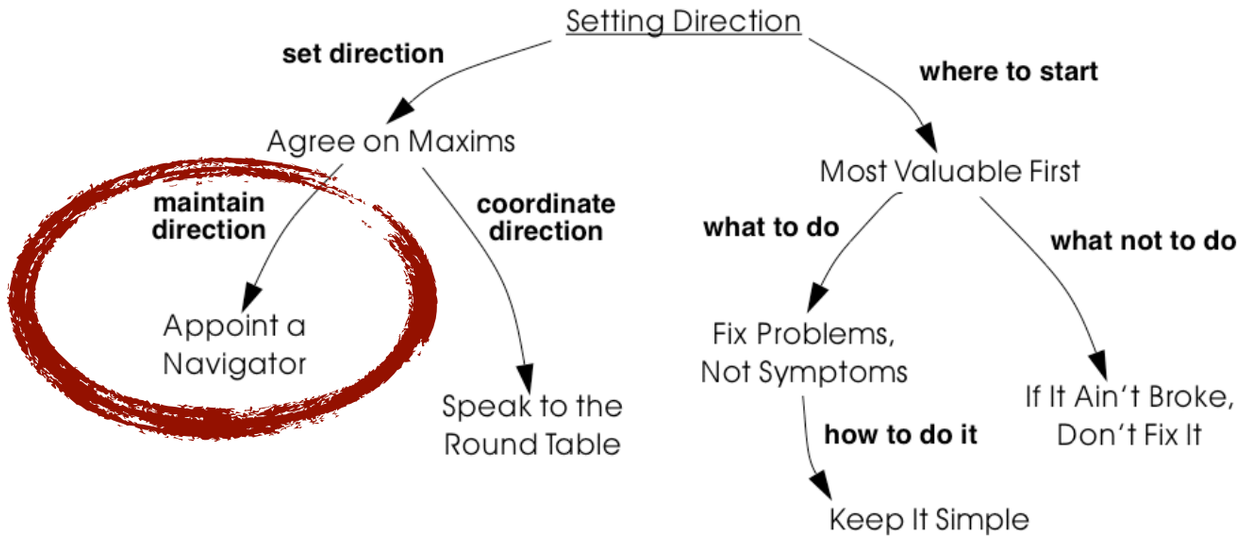


Setting Direction: Appoint a Navigator

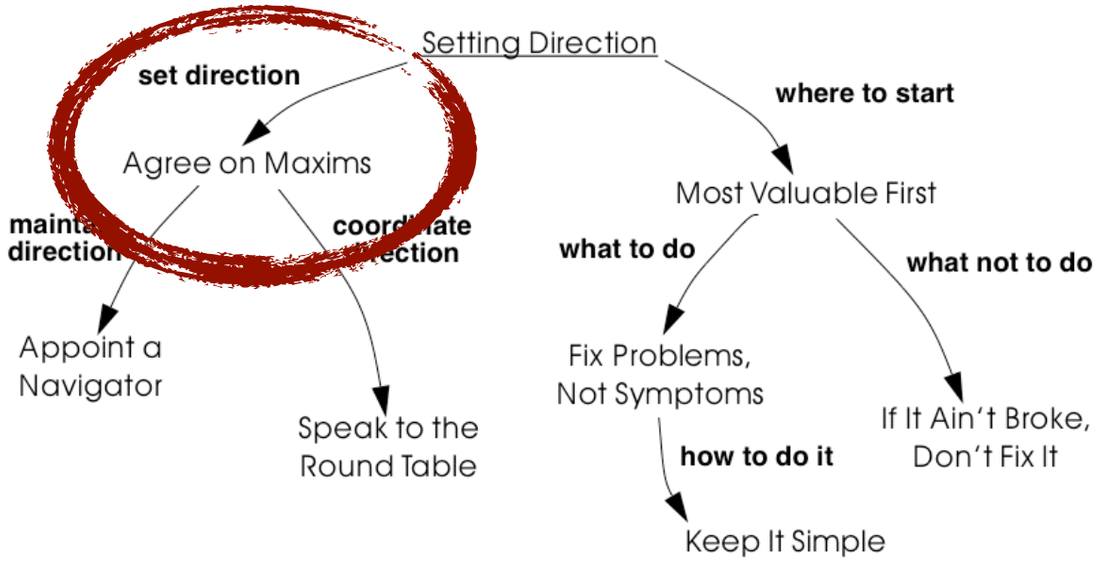
- **Problem**
  - How do you maintain **architectural vision** during the course of a complex project?

You should **Appoint a Navigator** to maintain the architectural vision

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



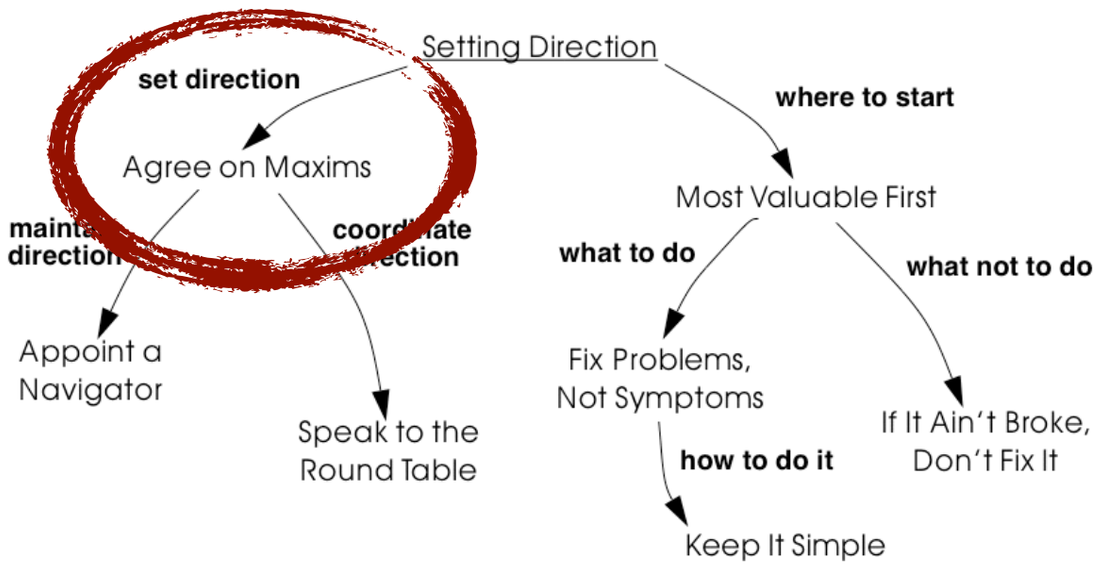
Agree on Maxims



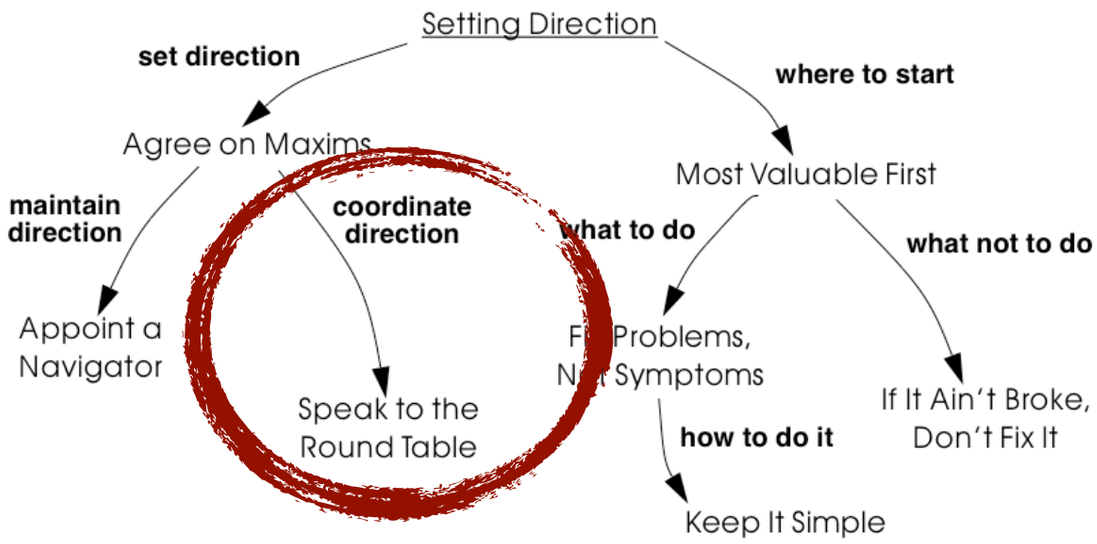
Setting Direction: Agree on Maxims

- **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**.
- **Examples**
  - “The central subsystem can be taken down without impact on end-users”
  - “Every external system must be simulated with a mock system”

You should **Agree on Maxims** in order to establish a common understanding within the reengineering team of what is at stake and how to achieve it.



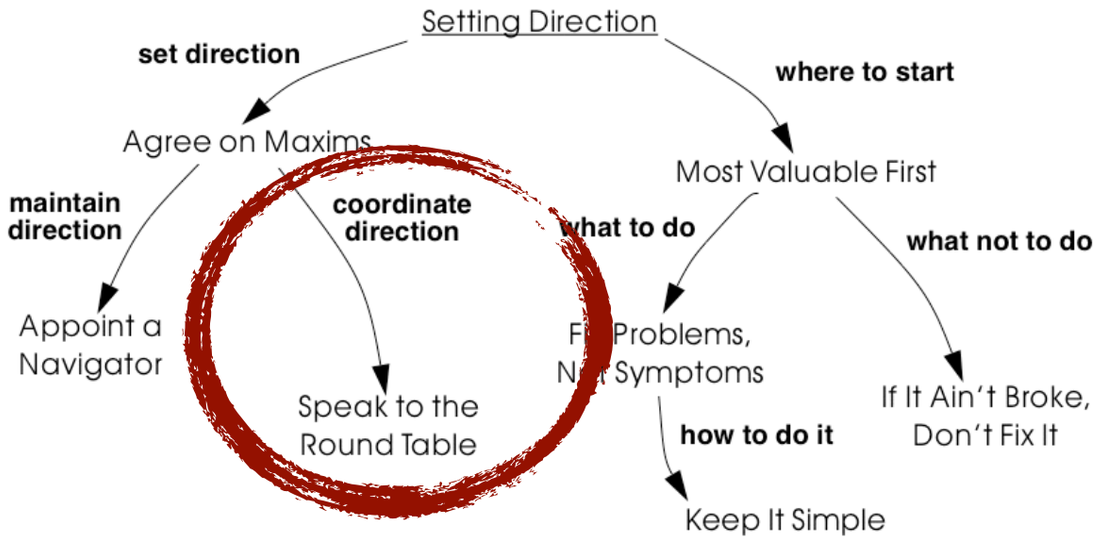
Speak to the Round Table



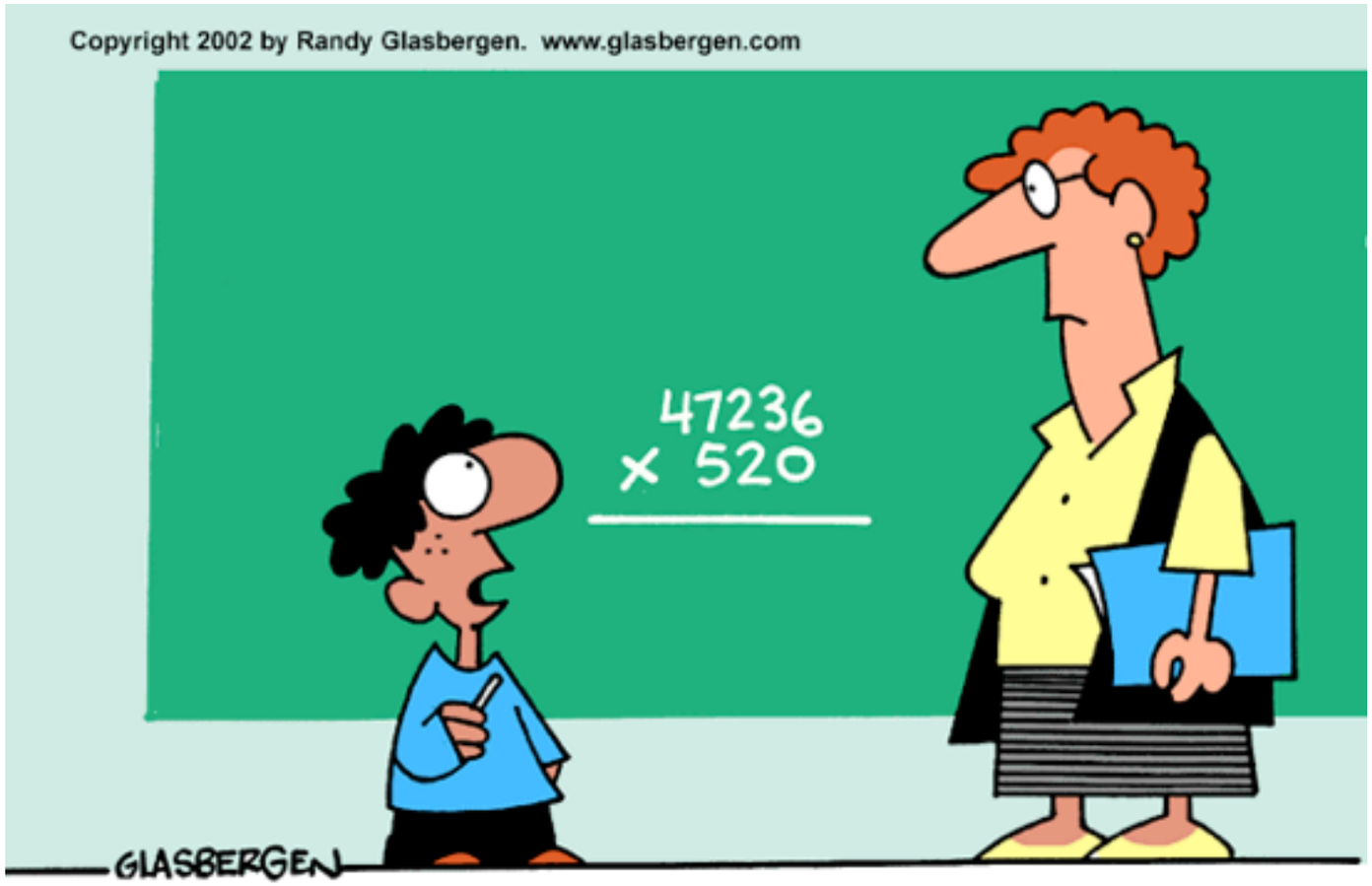
Setting Direction: Speak to the Round Table

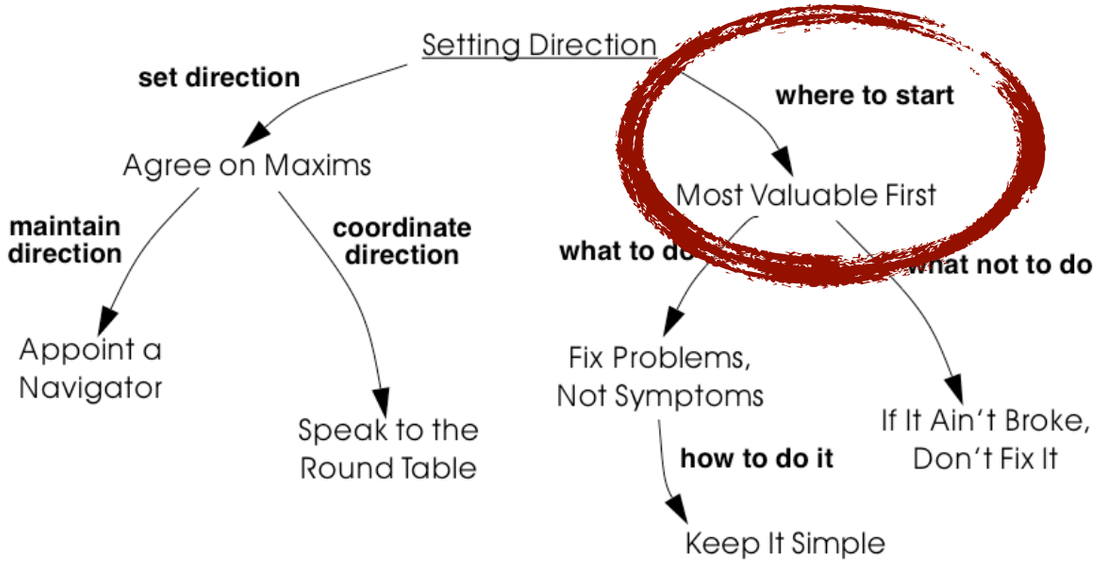
- **Problem**
  - How do you keep your team **synchronized**?
- **Solution**
  - Hold **brief**, **regular round table meetings**.

Everyone should **Speak to the Round Table** to maintain team awareness of the state of the project.



Most Valuable First



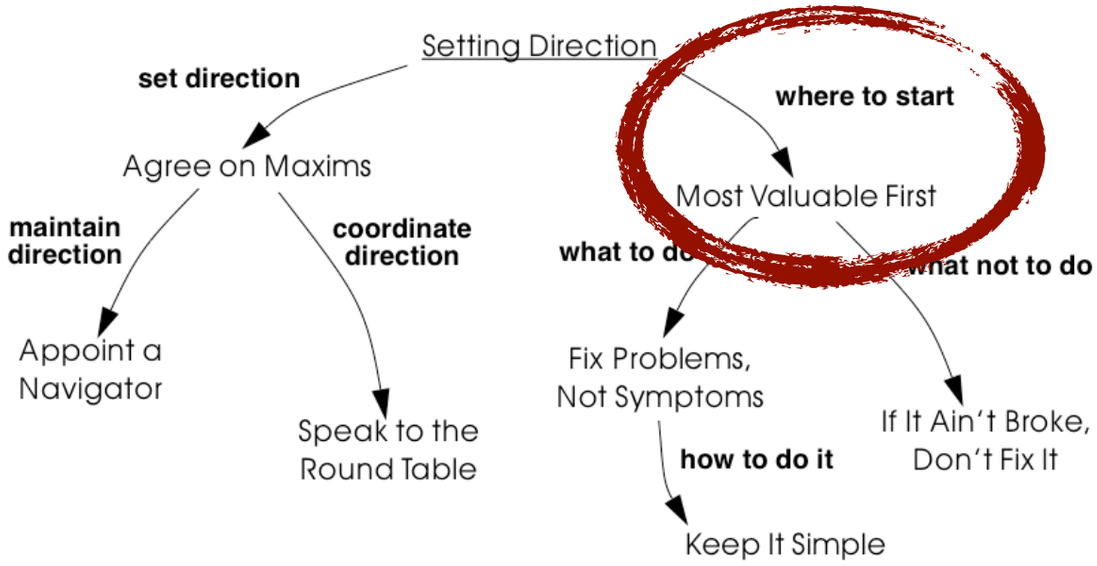


Setting Direction: Most Valuable First

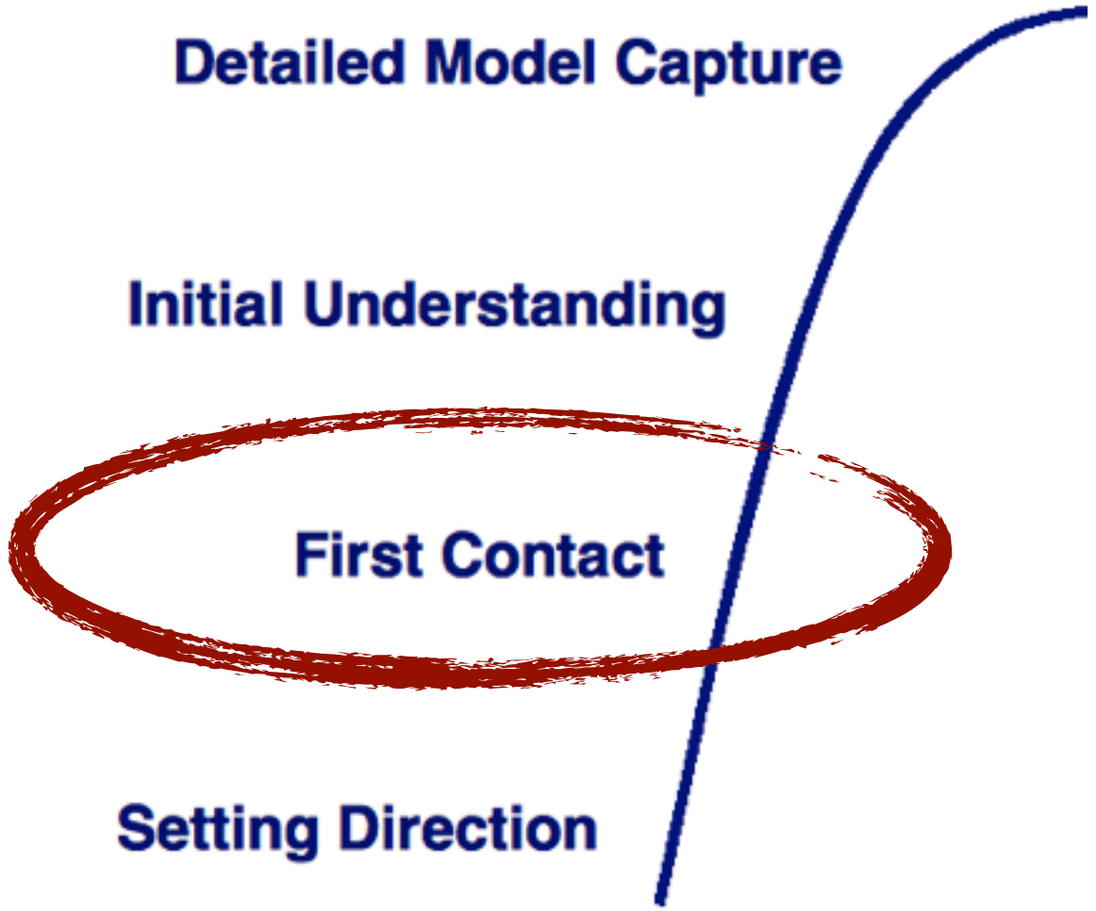
- **Problem**
  - Which problems should you focus on **first**?
- **Solution**
  - Start working on the aspects which are **most valuable** to your **customer**.
- **Who is the your customer?**
  - Many stakeholders, but who makes decisions?
- **What is the most valuable?**

- try to understand the customer's **business model**
- try to determine what **measurable** goal the customer wants to obtain

To help you focus on the right problems and the critical decisions, it is wise to tackle the **Most Valuable First**.



First Contact

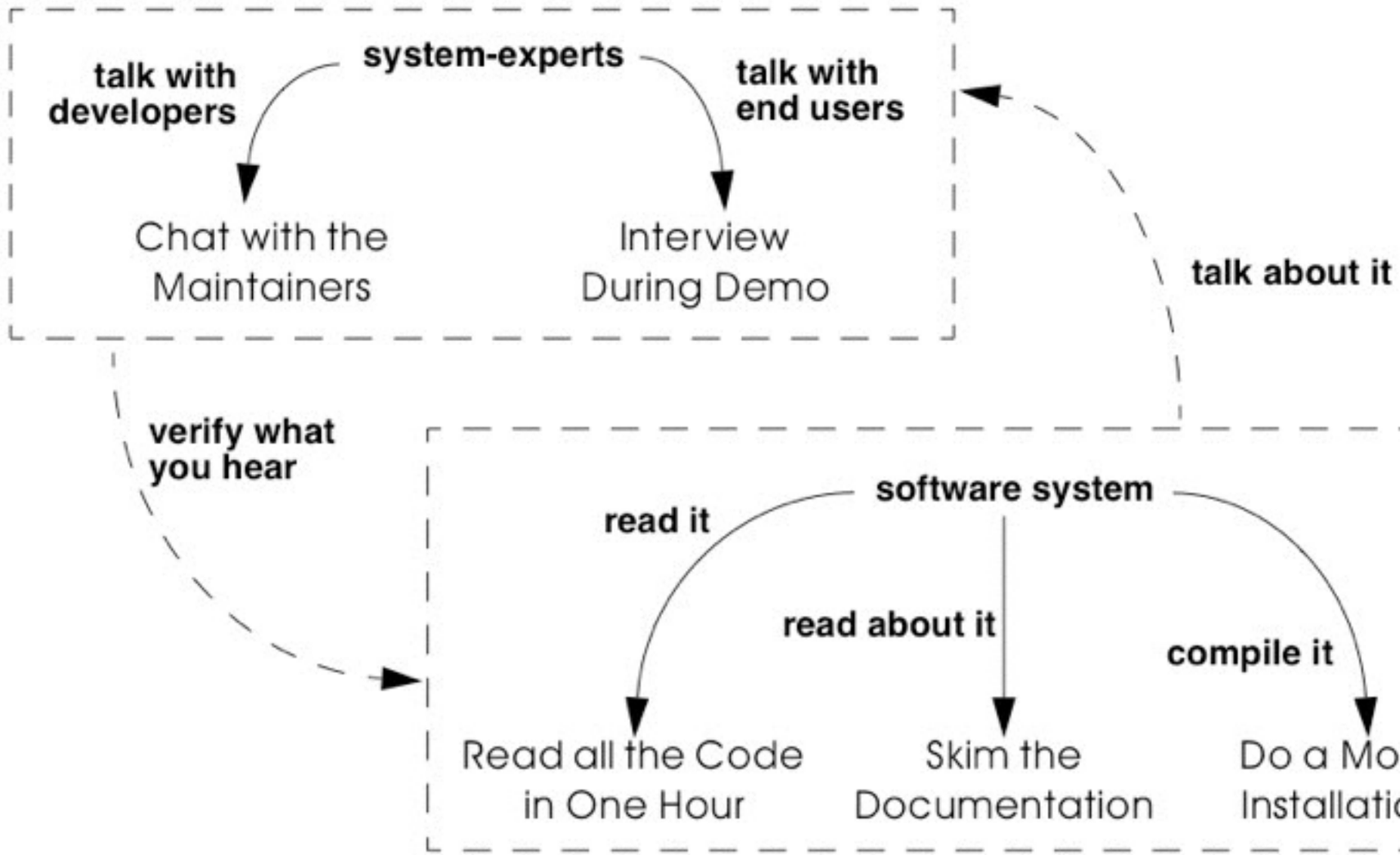


*“All the patterns in this cluster can be applied to the very **early stages** of a reengineering project: you’re facing a system that is **completely new for you**, and **within a few days** you must determine whether something can be done with it and **present a plan** how to proceed.”*

First Contact: Forces

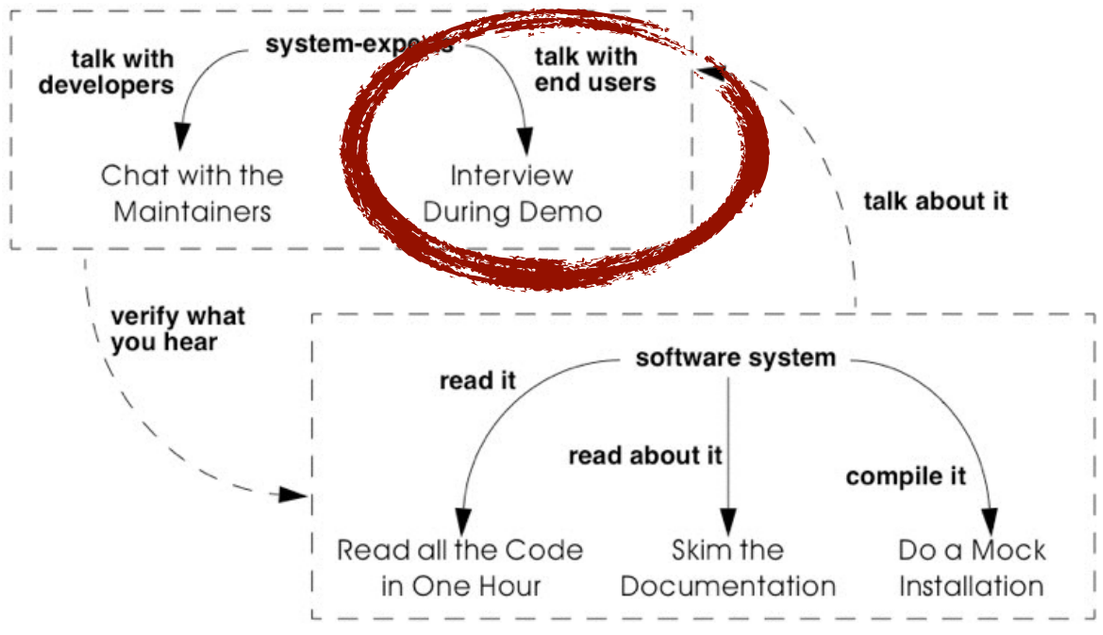
- **Legacy systems are large and complex**. Consequently, **split** the system into manageable pieces, where a manageable piece is one you can handle with a single reengineering team.
- **Time is scarce**. It is tempting to start an activity that will keep you busy for a while instead of addressing the root of the problem.Therefore, **defer all time-consuming activities** until later and use the first days of the project to **assess the feasibility** of the project's goals.
- **First impressions are dangerous**. There is no way to avoid that risk during your first contact with a system, however you can minimize its impact if you always **double-check your sources**.
- **People have different agendas**. In order to make the project a success, you must keep convincing the **faithful**, gain credit with the **fence sitters** and be wary of the **skeptics**.

First Contact: Patterns



*"**Wasting time** is the largest risk when you have your first contact with a system, therefore these patterns should be applied during a short time span, **say one week**. After this week you should **grasp the main issues** and based on that knowledge **plan** further activities, or — when necessary — cancel the project."*

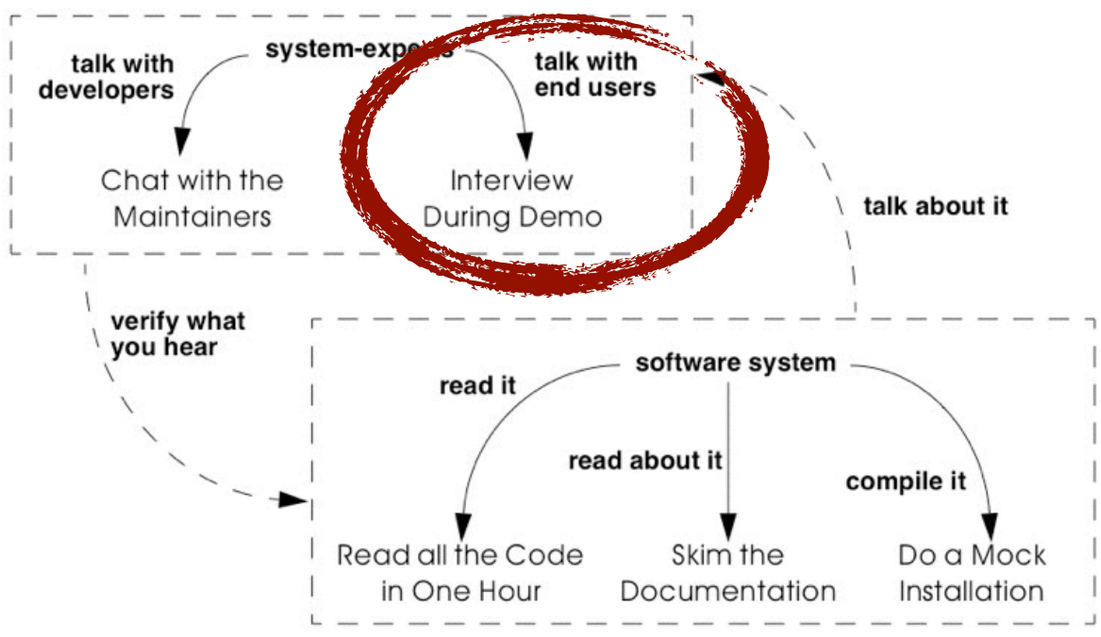
Interview During Demo



First Contact: Interview During Demo

- **Problem**
  - How can you get an idea of the typical **usage scenarios** and the **main features** of a software system?
  - Typical usage scenarios vary quite a lot depending on the **type of user**.
  - If you ask the users, they have a tendency to complain about **what's wrong**, while for reverse engineering purposes you're mainly interested in **what's valuable**.
    - You can **exploit the presence of a working system and a few users** who can demonstrate how they use the software system.
- **Solution**
  - Observe the system in operation by seeing a demo and interviewing the person who is demonstrating.
  - **Note that the interviewing part is at least as enlightening as the demo**.
  - Write a short report.

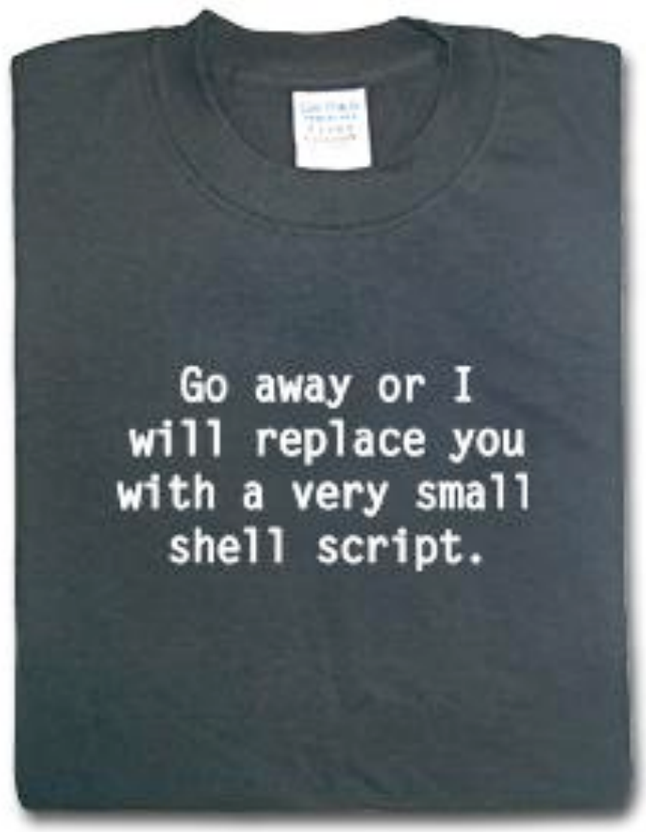
Obtain an **initial feeling for the appreciated functionality** of a software system by seeing a demo and interviewing the person giving the demo.

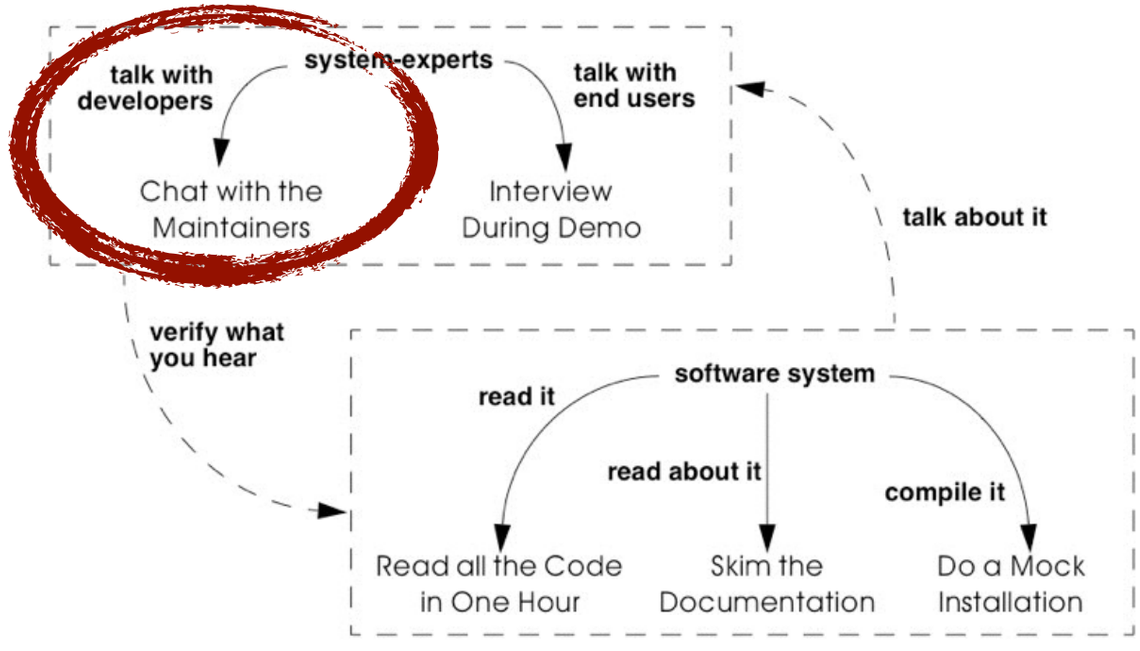


First Contact: Interview During Demo

- **Hints**
  - An **end-user** should tell you **how the system looks like from the outside** and explain some detailed usage scenarios based on the **daily working practices**.
  - A **manager** should inform you how the system fits within the **rest of the business domain**.
  - A **person from the sales department** ought to compare your software system with **competing** systems.
  - A **person from the help desk** should demonstrate you which features **cause most of the problems**.
  - A **system administrator** should show you all that is happening behind the scenes of the software system. **Ask for past horror stories** to assess the reliability of the system.
  - A **developer** may demonstrate you some of the subsystems. Ask how this subsystem communicates with the other subsystems and why (and who!) it was designed that way. Use the opportunity to **get insight in the architecture of the system** and the trade-offs that influenced the design.

Chat with Maintainers



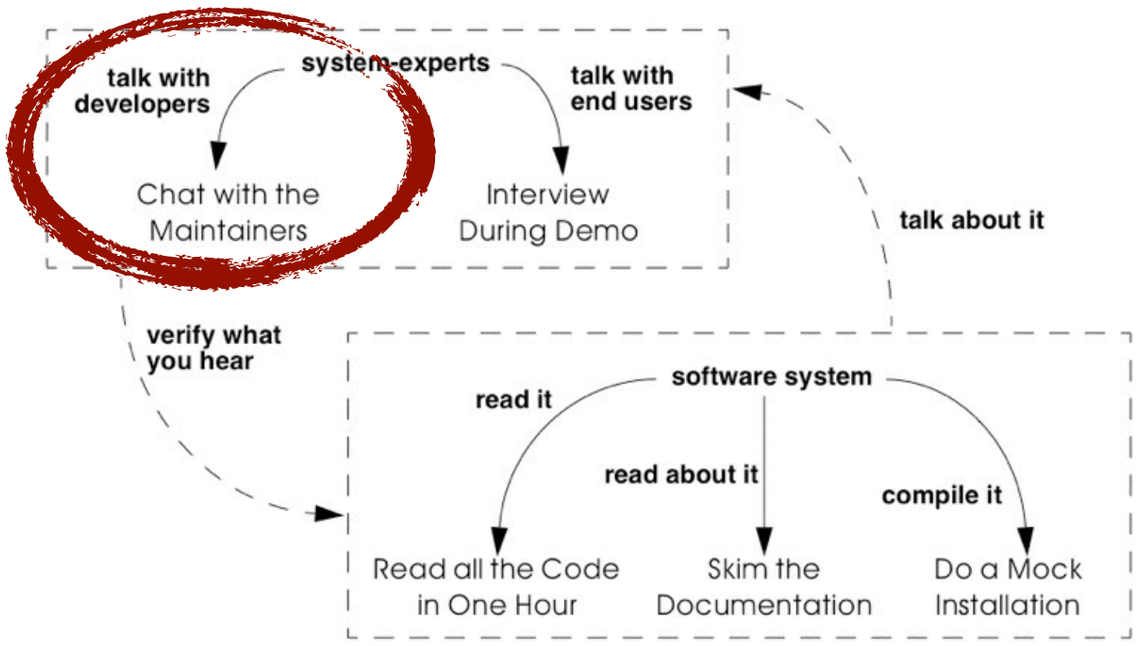


First Contact: Chat with the Maintainers

- **Problem**
  - How do you get a good perspective on the **historical** and **political** context of the legacy system you are reengineering?
- **Solution**
  - Discuss with the system maintainers. As **technical people** who have been intimately involved with the legacy system, they are well aware of the **system's history** and the **people-related issues** that influenced that history.

- To avoid misleading information, treat the maintainers as "**brothers in arms**".

Learn about the **historical** and **political context** of your project through discussions with the people maintaining the system.



First Contact, Chat with Maintainers: Questions (1)

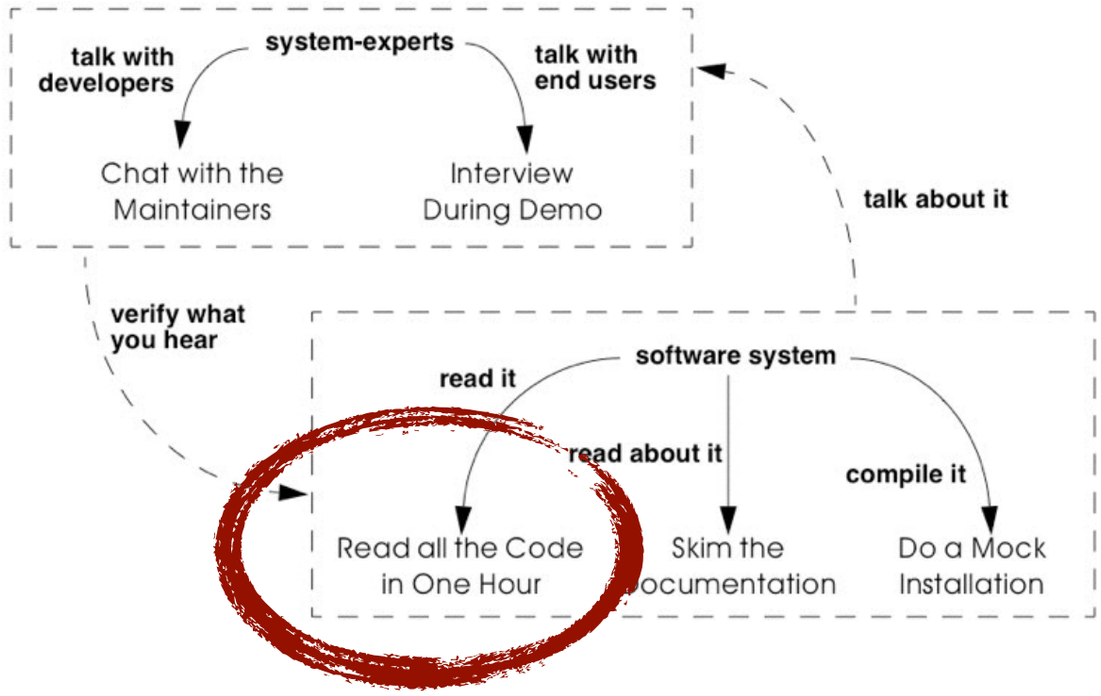
- **What was the easiest bug you had to fix during the last month? And what was the most difficult one?**
  - Good starters because they **show that you are interested in the maintenance work**.
  - Will provide you with **some concrete examples** of maintenance problems you might use in later, more high-level discussions.
- How does the maintenance team **collect bug reports** and **feature requests**? **Who decides** which request gets handled first? Is there a **version or configuration management system** in place?
  - Help to **understand the organization** of the maintenance process and the internal **working habits**.
  - Helps to assess the relationships **within the team** and **with the end users**.

First Contact, Chat with Maintainers: Questions (2)

- **Who was part of the development/maintenance team during the course of years? How did they join/leave the project? How did this affect the release history of the system?**
  - It is a good idea to ask about persons because people generally have a good recollection of former colleagues.
- **How good is the code? How trustworthy is the documentation?**
  - You will have to verify their claims yourself afterwards
- **Why is this reengineering project started? What do you expect from this project? What will you gain from the results?**
  - It is crucial to ask what the maintainers will gain from the reengineering project as it is something to keep in mind during the later phases.

Read all the Code in One Hour

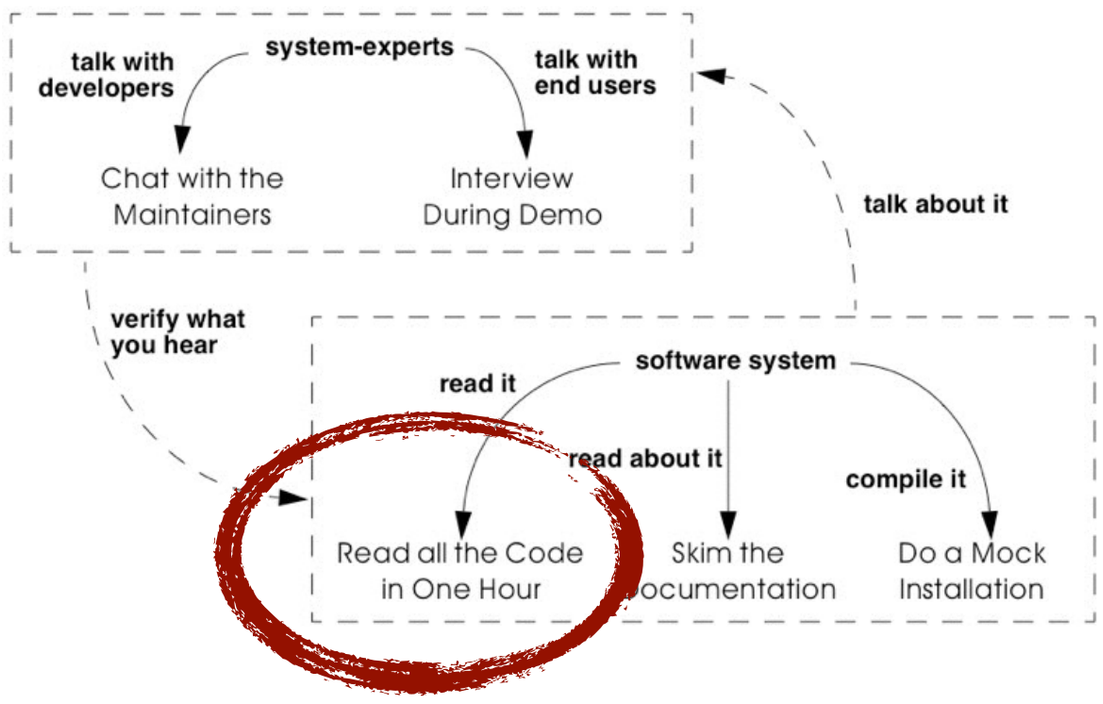
"My short-term goal is to bluff my way through this job interview. My long-term goal is to invent a time machine so I can come back and change everything I've said so far."



First Contact: Read all the Code in One Hour

- **Problem**
  - How can you get a **first impression** of the quality of the source code?
  - The system is large, so there is **too much data** to inspect for an accurate assessment.
  - You're unfamiliar with the software system, so you do not know **how to filter out what's relevant**.
  - You have **reasonable expertise** with the implementation language being used.
  - Your reengineering project has a **clear goal**, so you can assess the kind of code quality required to obtain that goal.
- **Solution**
  - Take a short time (i.e., approximately one hour) to read the source code.
  - Produce a report (general assessment, entities which seem important, suspicious coding styles - "smells", parts which needs to be investigated further)

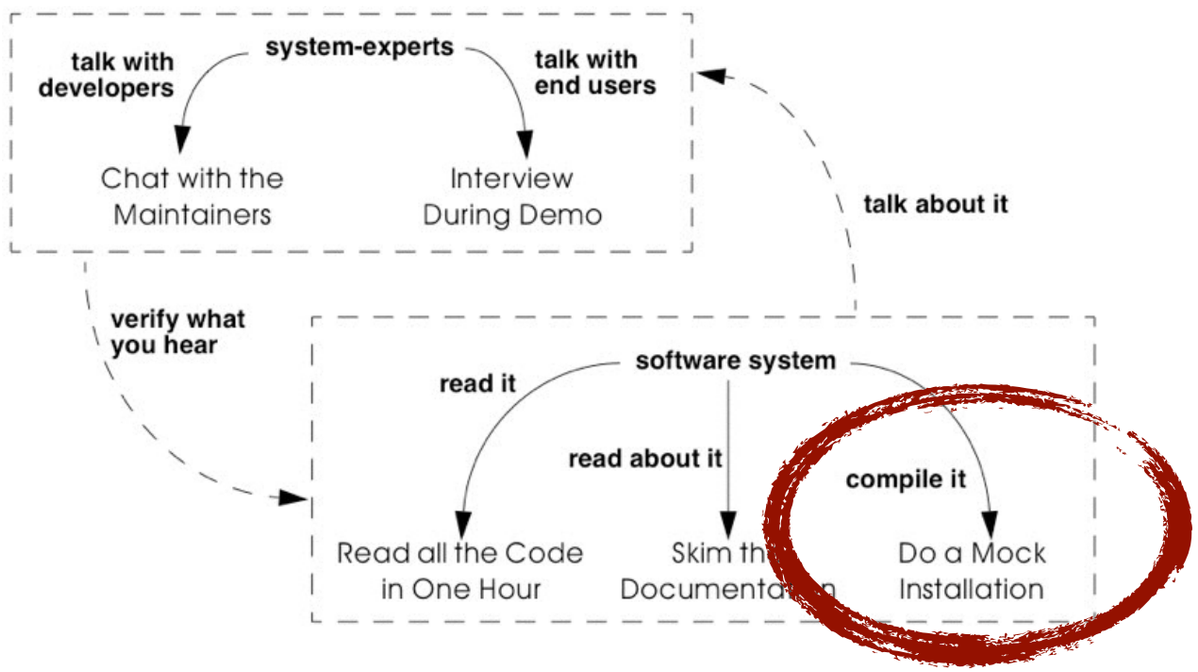
Assess the state of a software system by means of a **brief, but intensive code review**.



First Contact: Read all the Code in One Hour

- **Pros**
  - **Start efficiently**. Reading the code in a short amount of time is very efficient as a starter. Indeed, by limiting the time and yet forcing yourself to look at all the code, **you mainly use your brain and coding expertise to filter out what seems important**.
  - **Judge sincerely**. By reading the code directly you **get an unbiased view of the software system** including a sense for the details and a glimpse on the kind of problems you are facing.
  - **Learn the developers vocabulary**. Acquiring the vocabulary used inside the software system is essential to understand it and communicate about it with other developers.
- **Cons**
  - **Obtain low abstraction**. Via this pattern, you will get some insight in the solution domain, but only very little on how these map onto problem domain concepts.

Do a Mock Installation

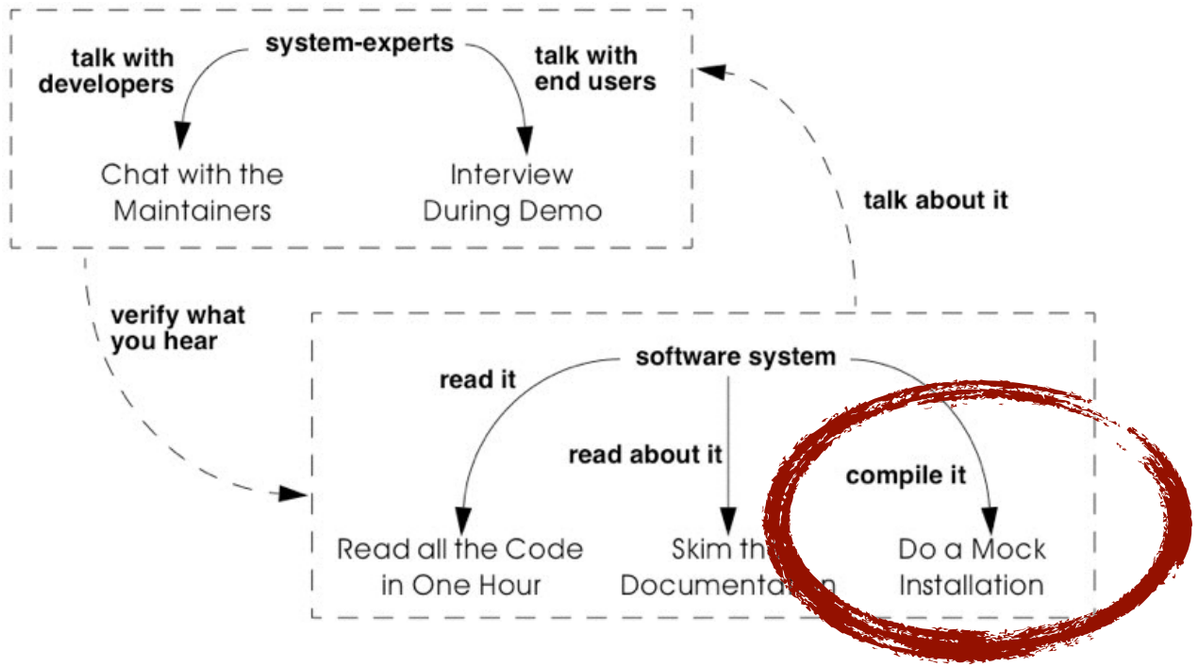


First Contact: Do a Mock Installation

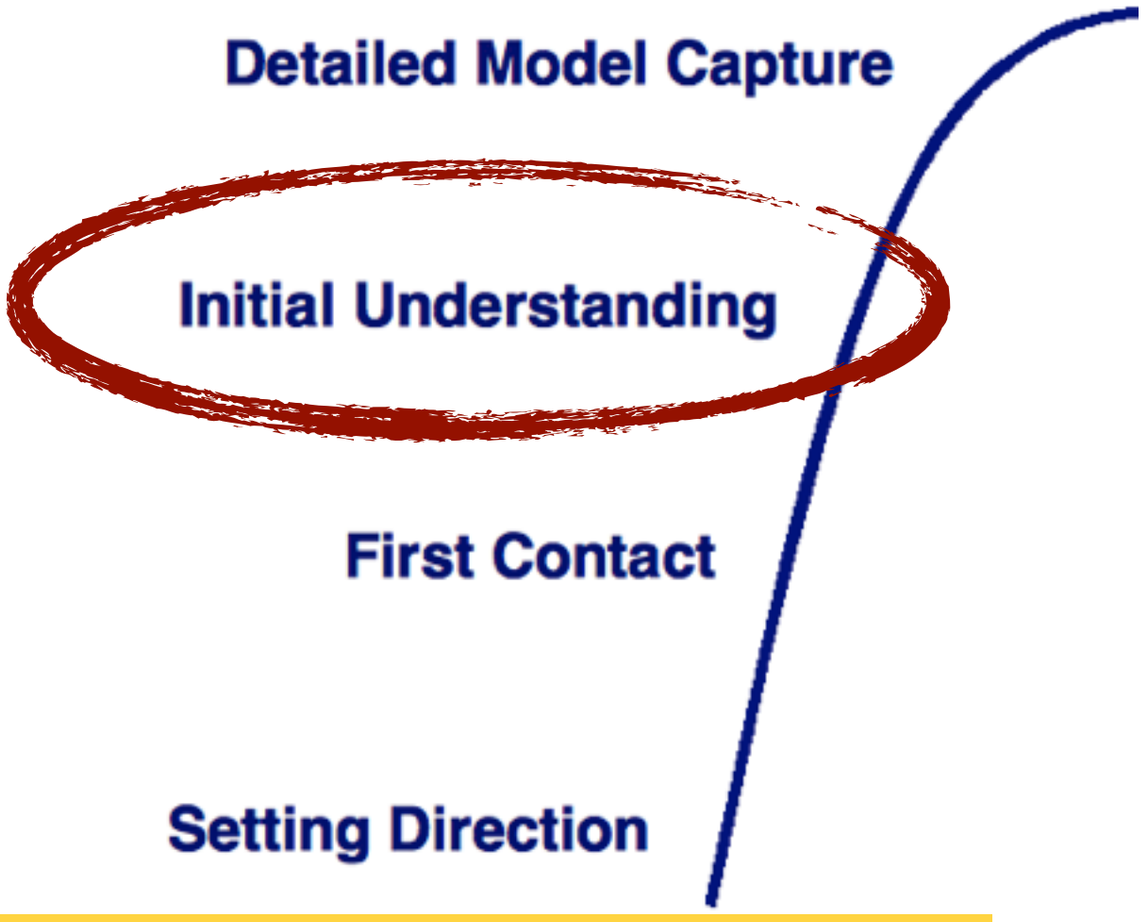
- **Problem**
  - How can you be sure that you will be able to (re)build the system?
  - The system is new for you, so you do not know which files you need to build the system.
  - You have access to the source code and the necessary build tools (i.e., the makefiles, compilers, linkers).
  - Maybe the system includes some kind of self test, which you can use to verify whether the build or install succeeded.

Check whether you have the necessary artefacts available by installing the system and recompiling the code.

- **Solution**
  - Try to install and build the system in a clean environment during a limited amount of time (at most one day). Run the self test if the system includes one.

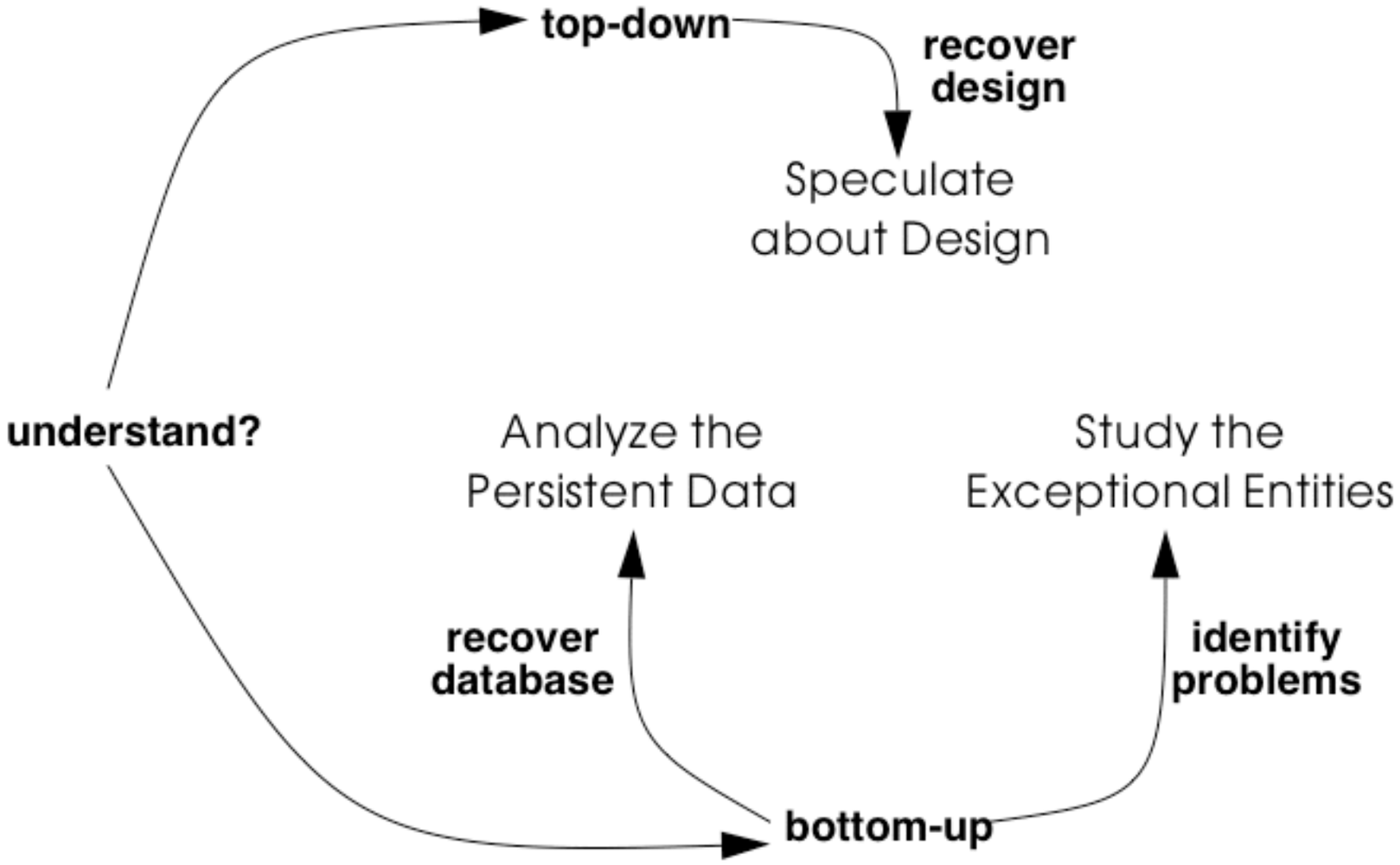


Initial Understanding



“The patterns in First Contact should have helped you to get some first ideas about the software system. ***Now is the right time to refine those ideas into an initial understanding and to document that understanding*** *in order to support further reverse engineering activities”*

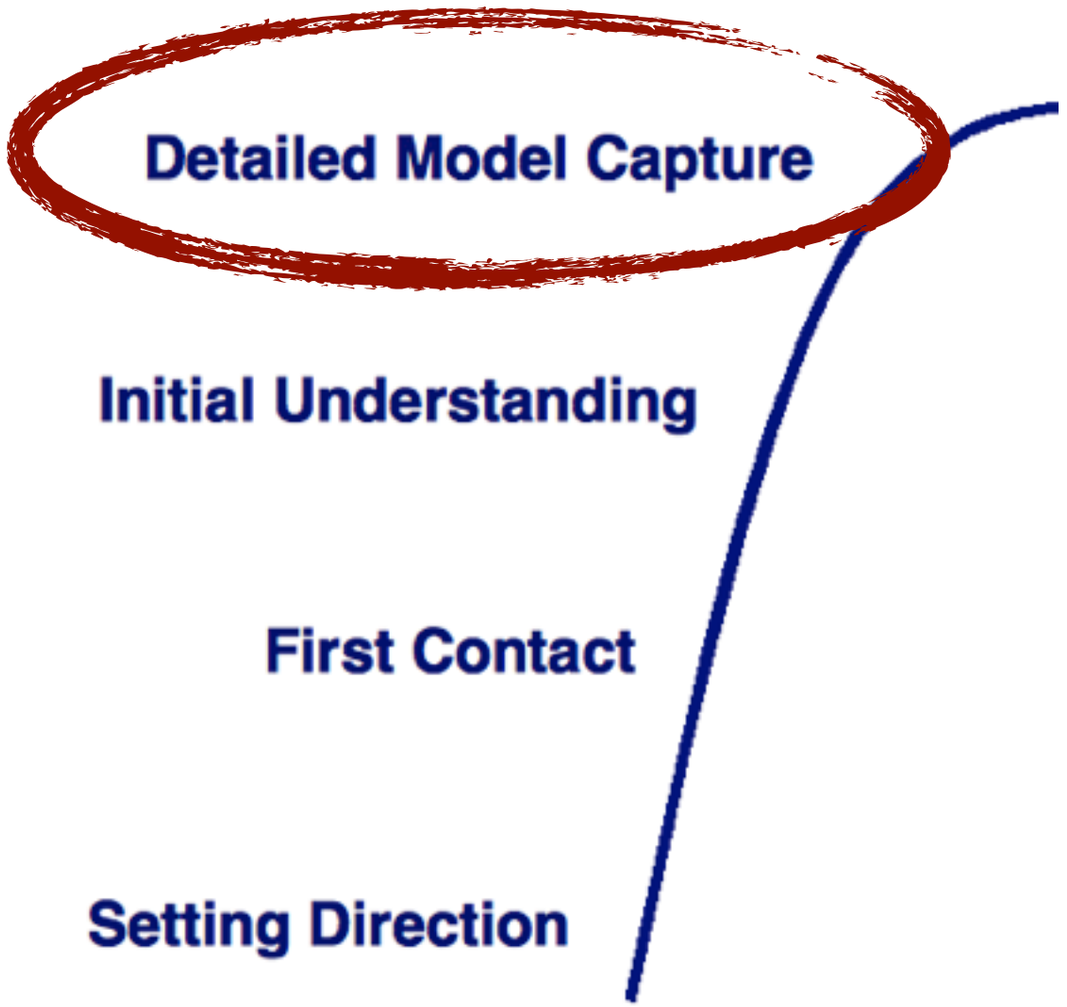
Initial Understanding: Patterns



Initial Understanding: Overview

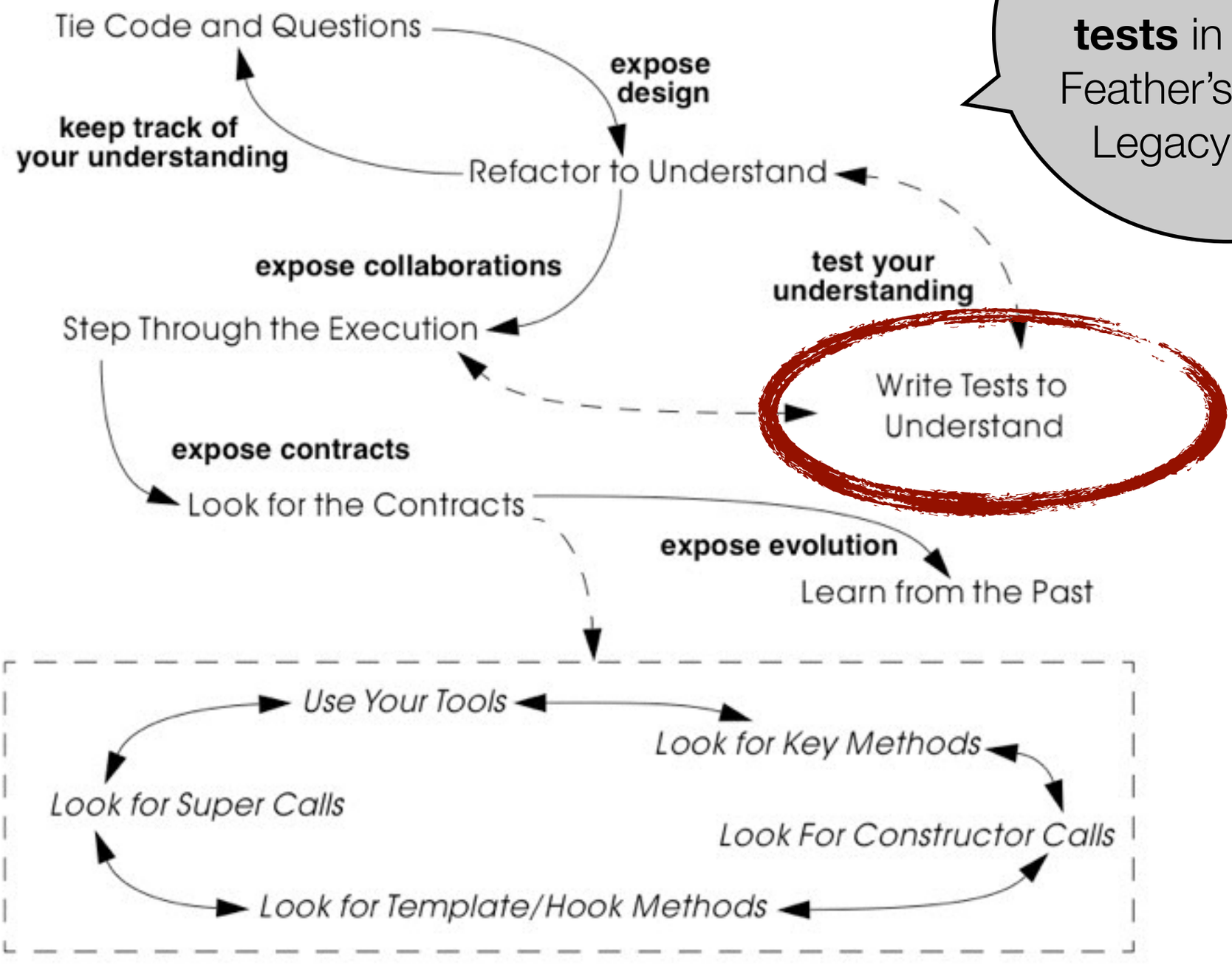
- These patterns rely mainly on **source code** because this is the only trustworthy information source.
- Two approaches for studying source code (both are valuable):
  - **Top-down**: start from a high-level representation and verify it against the source-code (see Speculate About Design)
  - **Bottom-up**: start from the source-code, filter out what's relevant and cast the relevant entities into a higher-level representation (see Analyze the Persistent Data and Study the Exceptional Entities).
- There is not a unique order in which to perform these activities.

Detailed Model Capture



*"Your main priority now is to build up a **detailed model** of **those parts of the system** that will be important for your reengineering effort. Most of the patterns concerned with Detailed Model Capture entail considerably **more technical knowledge**, use of **tools** and investment of **effort** than the patterns we have applied up to now"*

Detailed Model Capture



See **characterization tests** in Michael Feather's "Testing Legacy Code"

MASTER OF SCIENCE
IN ENGINEERING

Additional Slides

Agenda

- **Introduction**
  - Software evolution, legacy systems, reverse engineering and reengineering.
- **Reverse Engineering Patterns**
  - Setting Direction
  - First Contact
  - Initial Understanding
  - Detailed Model Capture

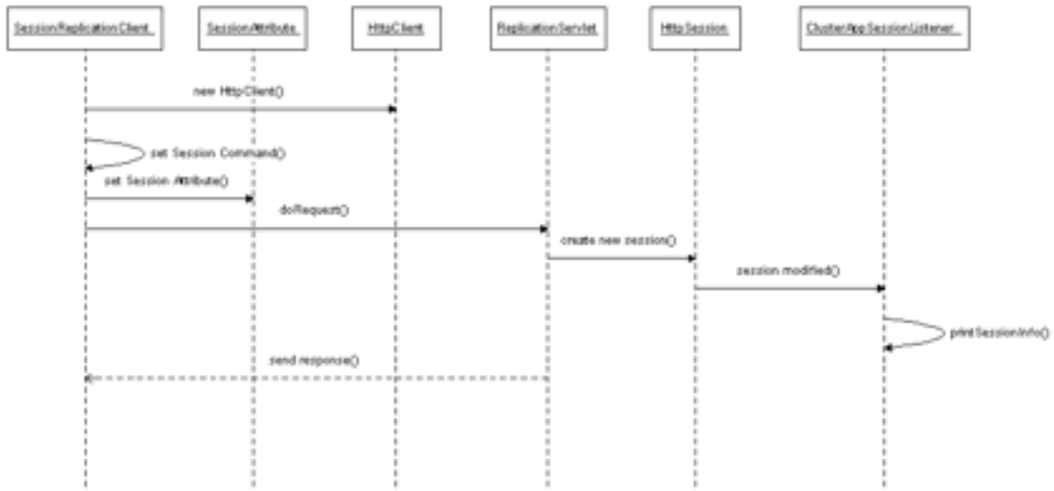
Goals of Reverse Engineering

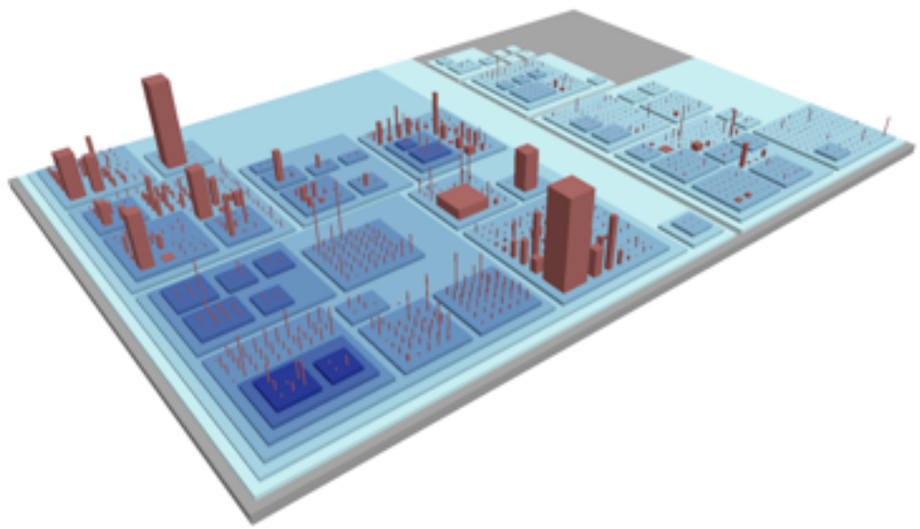
- **Cope with complexity**
  - need techniques to understand large, complex systems
- **Recover lost information**
  - extract what changes have been made and why
- **Detect side effects**
  - help understand ramifications of changes
- **Synthesize higher abstractions**
  - identify latent abstractions in software
- **Facilitate reuse**
  - detect candidate reusable artifacts and components

Chikofsky and Cross, 1993

Reverse Engineering Techniques

- **Redocumentation**
  - pretty printers
  - diagram generators
  - cross-reference listing generators
- **Design recovery**
  - software metrics
  - browsers, visualization tools
  - static analyzers
  - dynamic (trace) analyzers





Goals of Reengineering

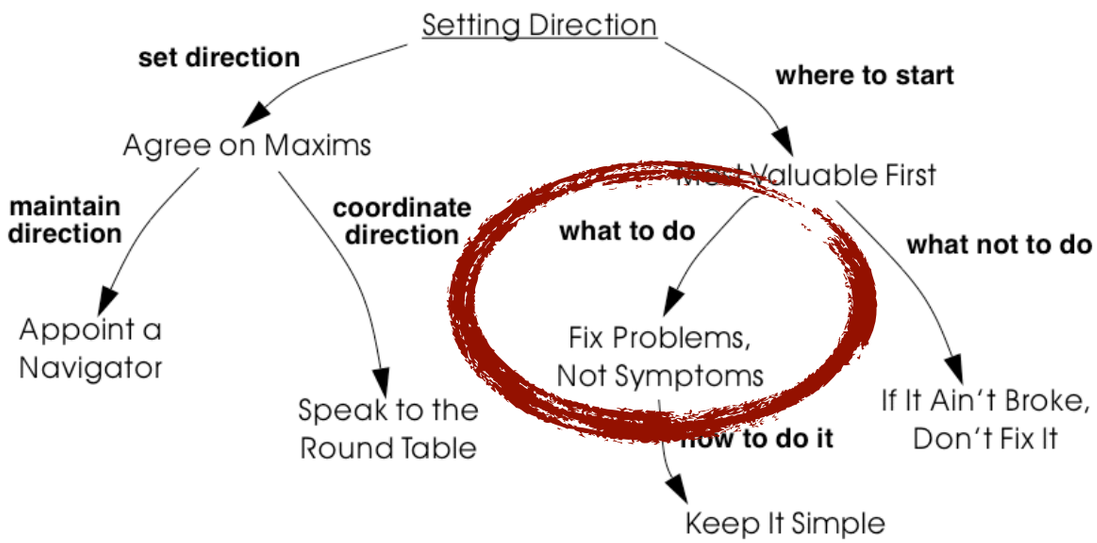
- **Unbundling**
  - split a monolithic system into parts that can be separately marketed
- **Performance**
  - “first do it, then do it right, then do it fast” — experience shows this is the right sequence!
- **Port to other Platform**
  - the architecture must distinguish the platform dependent modules
- **Design extraction**
  - to improve maintainability, portability, etc.

Mission Impossible

- **“Jim”**
  - “Yes?”
- **“You just finished your last client engagement, right...?”**
  - “Yes”
- **“You know Facebook and Twitter, right..?”**
  - “Yes…”
- **“Good. Your mission is to go fix a big social networking site. This message will self destruct in 10 seconds...”**
  - “Youpi.”

Fix Problems, Not Symptoms

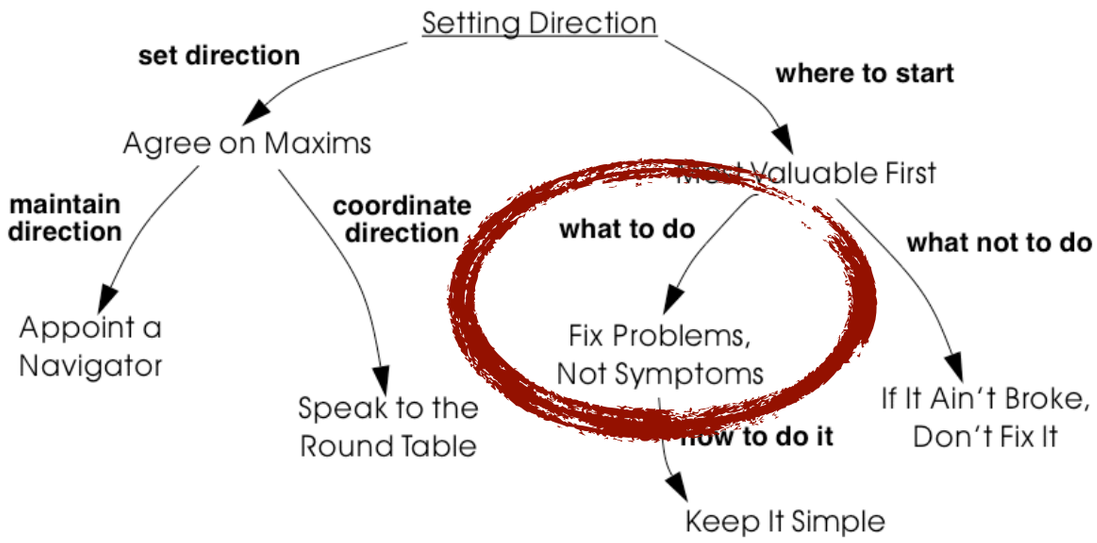
"My team has created a very innovative solution, but we're still looking for a problem to go with it."



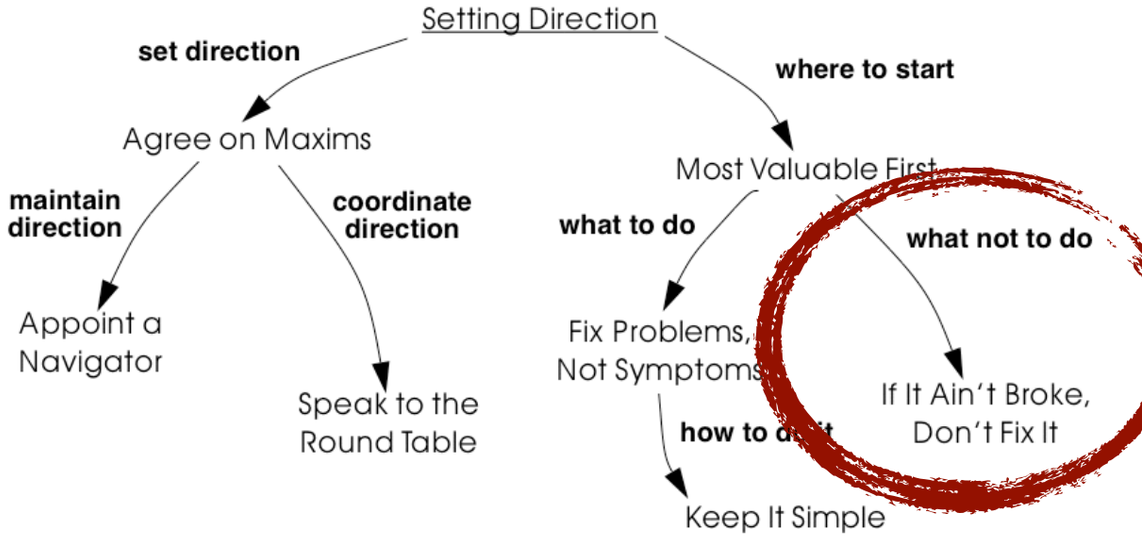
Setting Direction: Fix Problems, Not Symptoms

- **Problem**
  - How can you possibly tackle all the reported problems?
- **Solution**
  - Address the **source of a problem**, rather than particular requests of your stakeholders.

In order to decide whether to wrap, refactor or rewrite, you should **Fix Problems, Not Symptoms**.



If It Ain’t Broke, Don’t Fix It

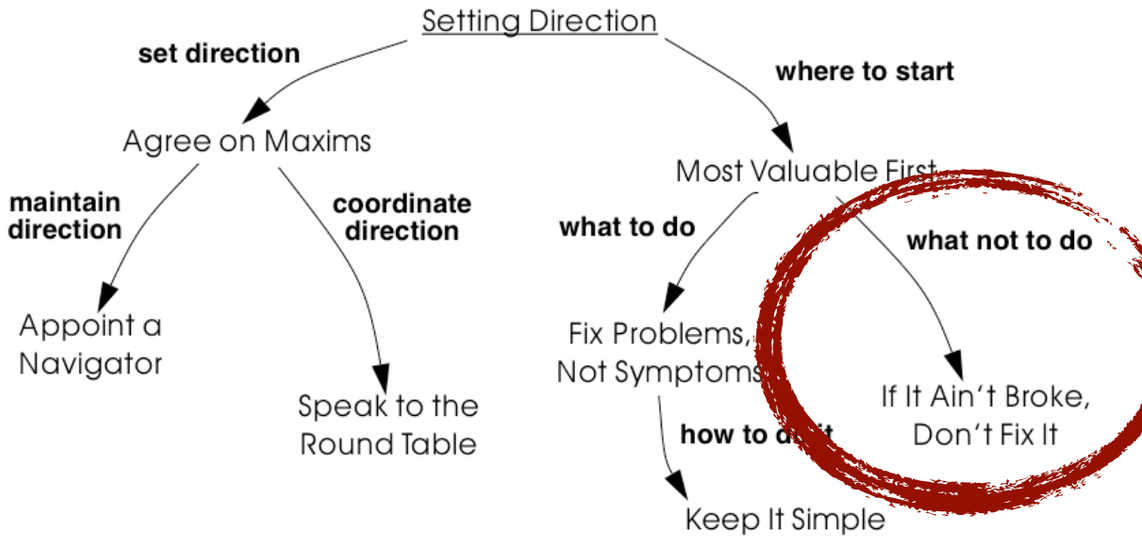


Setting Direction: If It Ain’t Broke, Don’t Fix It

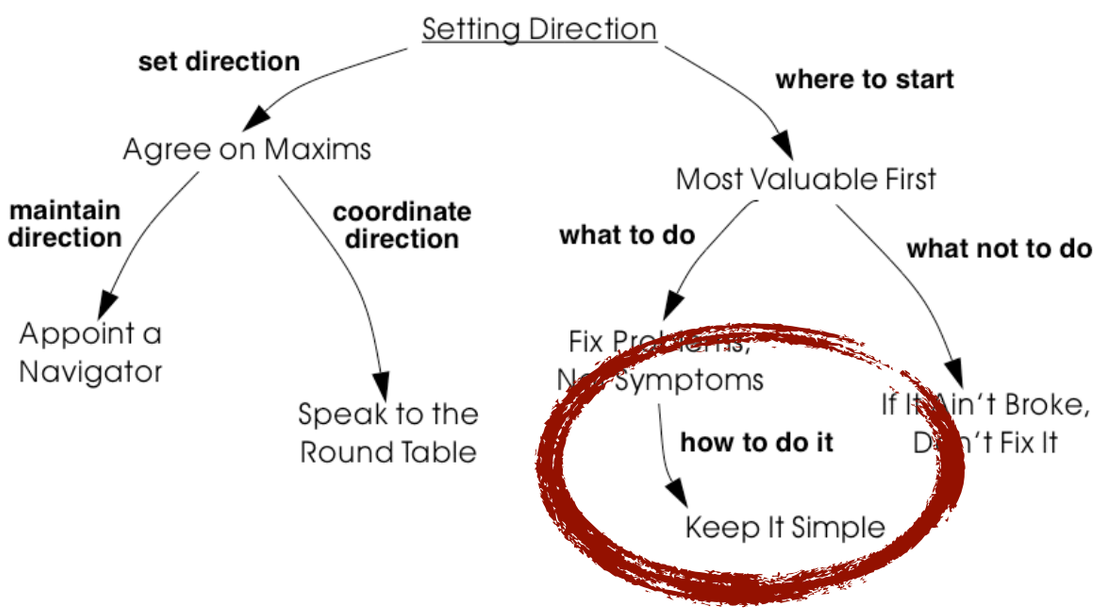
- **Problem**
  - Which parts of a legacy system should you reengineer and which should you leave as they are?
- **Solution**
  - Only fix the parts that are “broken” — those that can no longer be adapted to planned changes.
- **Which components are broken?**

Change for change’s sake is not productive, so **If It Ain’t Broke, Don’t Fix It**.

- 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**.



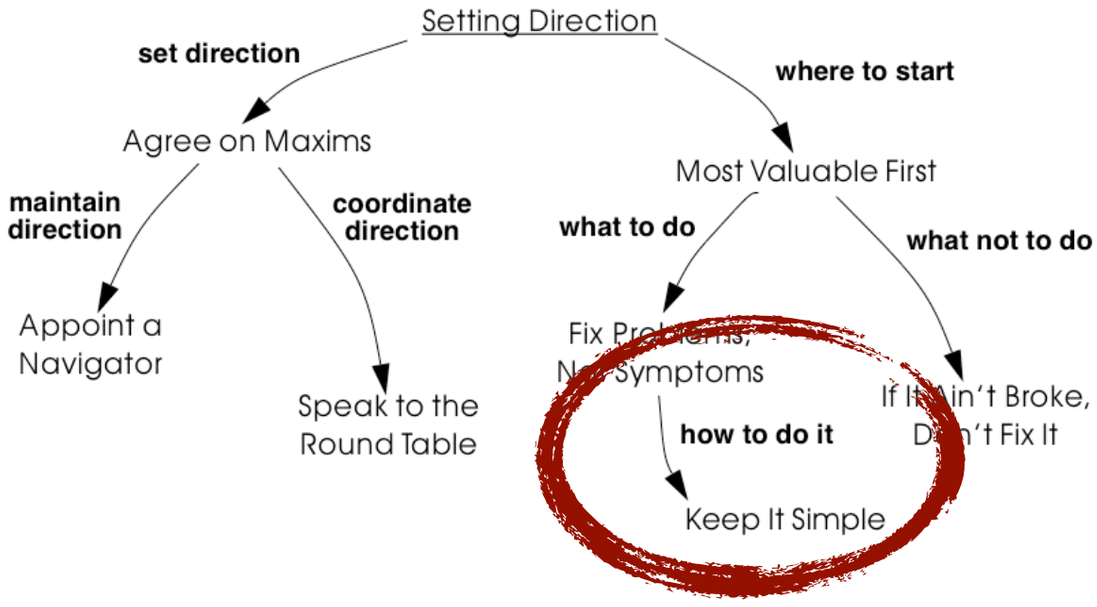
Keep It Simple



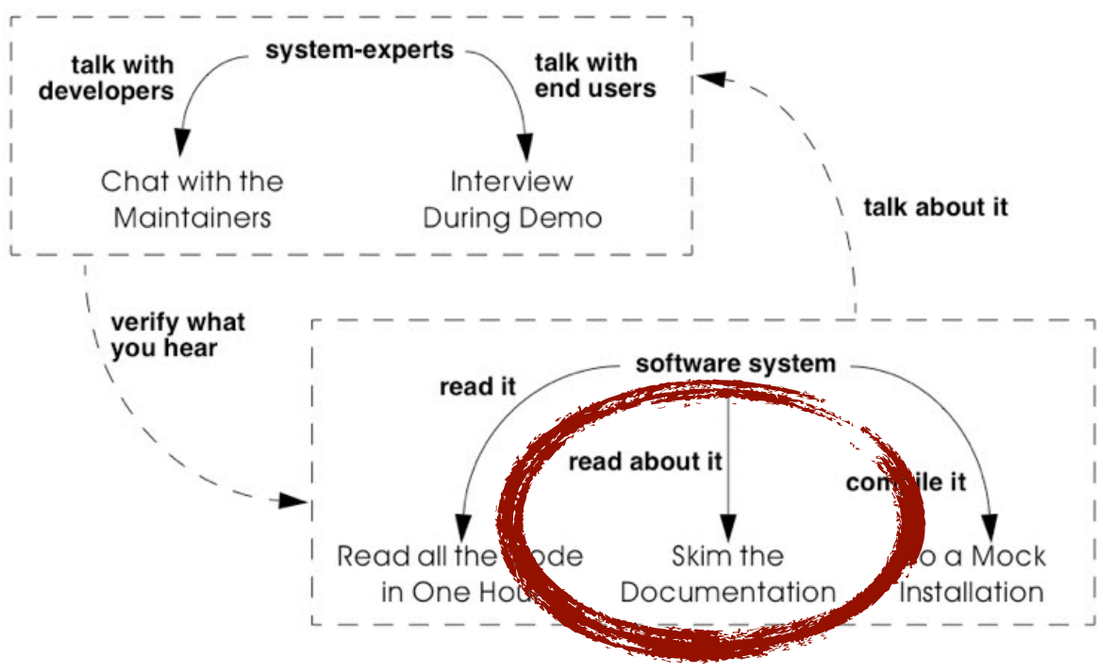
Setting Direction: Keep It Simple

- **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.
- **Discussion**
  - Flexibility is a double-edged sword. An important reengineering goal
  - is to accommodate future change. But too much flexibility will make the new system so complex that you may actually impede future change.

Although you may be tempted to make the new system very flexible and generic, it is almost always better to **Keep It Simple**.



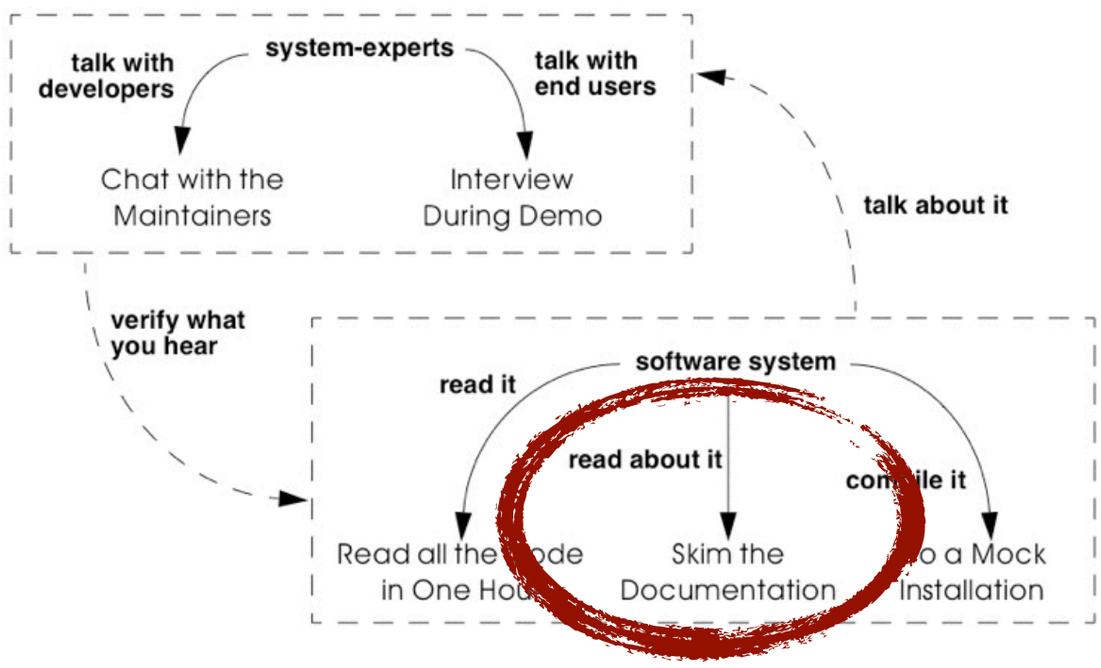
Skim the Documentation



First Contact: Skim the Documentation

- **Problem**
  - How to identify those **parts of the documentation** that might be of help?
  - Documentation, if present, is usually intended for the development team or the end users and as such not immediately relevant for reengineering purposes.
- **Solution**
  - Prepare a list summarizing those **aspects of the system** that seem interesting for your reengineering project.
  - Then, match this list against the documentation and meanwhile make a crude **assessment of how up to date** the documentation seems.
  - Finally, summarize your findings in a short report.

Assess the relevance of the documentation by reading it in a limited amount of time.



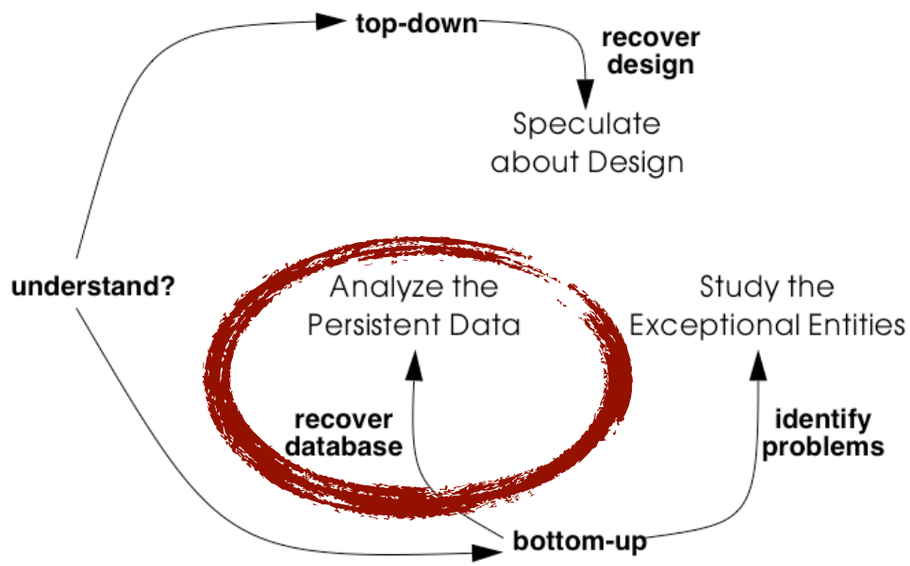
First Contact: Skim the Documentation

- **Hints**
  - A **table of contents** gives you a quick overview of the structure and the information presented.
  - **Version numbers and dates** tell you how up to date that part of the documentation is.
  - **Figures** are a a good means to communicate information.
  - **Screen-dumps**, sample print-outs, sample reports, command descriptions, reveal a lot about the functionality provided by the system.
  - **Formal specifications** (e.g., state-charts), if present, usually correspond with **crucial functionality**.
  - An **index**, if present contains the terms the author considers significant.

Initial Understanding: Analyze the Persistent Data

- **Problem**
  - Which object structures represent the **valuable data**?
- **Solution**
  - Analyze the **database schema** and filter out which structures represent valuable data.
  - **Derive a class diagram** representing those entities to **document that knowledge** for the rest of the team.
- **Note**
  - The book describes a detailed procedure on how to how perform this activity.

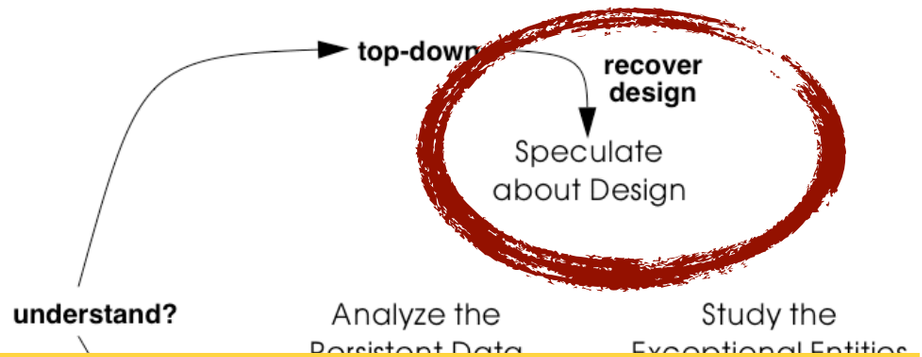
Learn about objects that are so **valuable** they must be kept inside a **database** system.



Initial Understanding: Speculate about Design

- **Problem**
  - How do you recover the way **design concepts** are represented in the source-code?
  - There are **many design concepts** and there are countless ways to represent them in the programming language used.
  - Much of the source-code won't have anything to do with the design but rather with implementation issues.
- You have a **rough understanding of the system's functionality** and you therefore have an initial idea which design issues should be addressed.
- You have **development expertise**, so you can imagine how you would design the problem yourself.
- You are **somewhat familiar with the main structure** of the source code so that you can find your way around.

- **Solution**
  - Use your **development expertise** to conceive a **hypothetical class diagram** representing the design.
  - Refine that model by **verifying** whether the names in the class diagram occur in the **source code** and by adapting the model accordingly.
  - **Repeat the process** until your class diagram stabilizes.



Progressively **refine a design** against source code by **checking hypotheses** about the design against the source code.

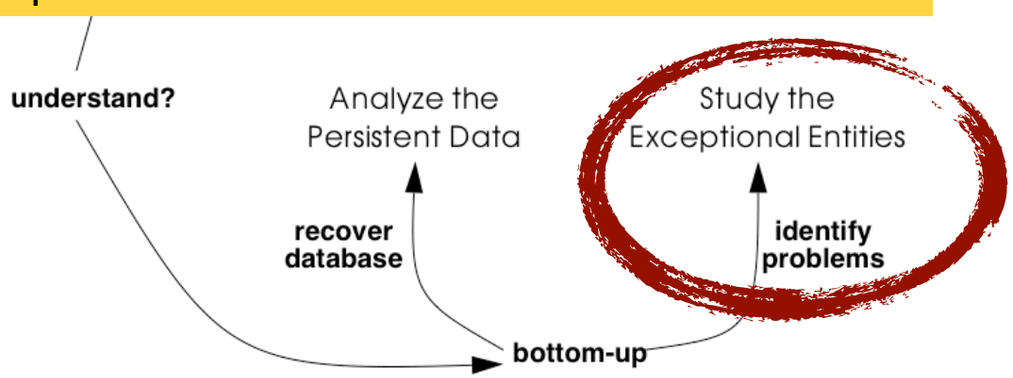
Initial Understanding: Speculate about Design

- **Pro**
  - **Scales well.** Speculating about what you'll find in the source code is a technique that scales up well. This is especially important because for large object-oriented programs (over a 100 classes) a bottom-up approach quickly becomes impractical.
- **Con**
  - **Requires expertise.** A large repertoire of **knowledge** about **idioms**, **patterns**, **algorithms**, **techniques** is necessary to recognize what you see in the source code. As such, the pattern should preferably **be applied by experts**.

Initial Understanding: Analyse the Persistent Data

- **Problem**
  - How can you **quickly identify potential design problems** in large software systems?
  - **No easy way to discern problematic from good designs**. Assessing the quality of a design must be done in the terms of the problem it tries to solve, thus can never be inferred from the design alone.
  - The system is large, thus a detailed assessment of the design quality of every piece of code is not feasible.
  - You have a **metrics tool** at your disposal, so you can quickly collect a number of measurements about the entities in the source-code.
  - You have the necessary tools to browse the source-code, so you can verify manually whether certain entities are indeed a problem.
- **Solution**
  - Measure the structural entities forming the software system (i.e., the inheritance hierarchy, the packages, the classes and the methods) and look for
  - exceptions in the quantitative data you collected. Verify manually whether these anomalies represent design problems.

Identify **potential design problems** by collecting measurements and studying the exceptional values.

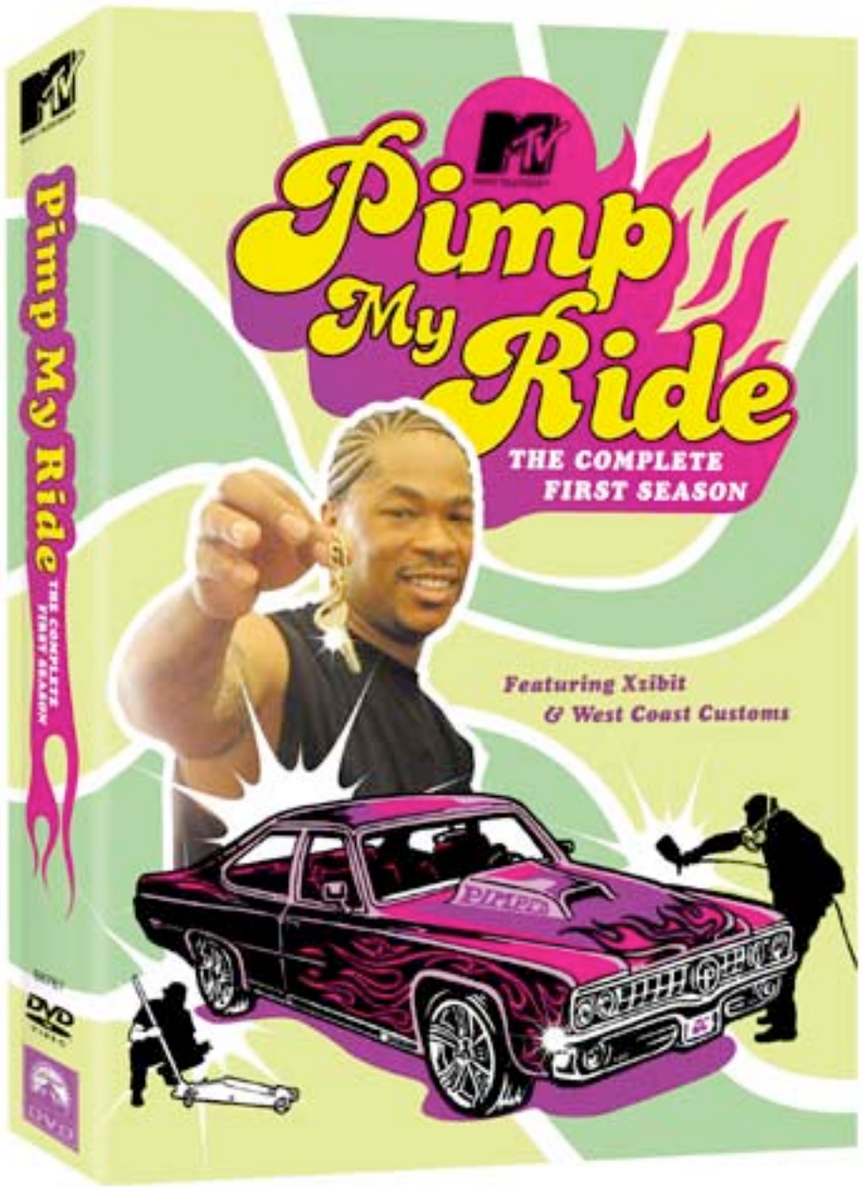


Patterns

- **Tie Code and Questions**
  - Keep the questions and answers concerning your reengineering activities synchronized with the code by storing them directly in the source files.
- **Refactor to Understand**
  - Iteratively refactor a part of a software system in order to validate and reflect your understanding of how it works.
- **Step Through the Execution**
  - Understand how objects in the system collaborate by stepping through examples in a debugger.
- **Look for the Contracts**
  - Infer the proper use of a class interface by studying the way clients currently use it.
- **Learn from the Past**
  - Obtain insights into the design by comparing subsequent versions of the system.
- **Write Tests to Understand**
  - Record your understanding of a piece of code in the form of executable
  - tests, thus setting the stage for future changes.

Reengineering Patterns

Reengineering patterns encode expertise and trade-offs in ***transforming legacy code*** *to resolve problems that have* ***emerged****.*



Reverse Engineering Patterns