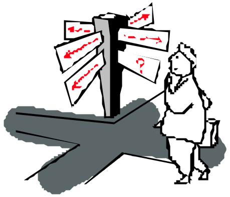
Reverse Engineering

Reverse Engineering



- What and Why
- *Setting Direction*
  - Most Valuable First
- *First Contact*
  - Chat with the Maintainers
  - Interview during Demo
- *Initial Understanding*
  - Analyze the Persistent Data
  - Study Exceptional Entities

What and Why ?

Definition

Reverse Engineering is the *process of analysing* 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.

— Chikofsky & Cross, '90

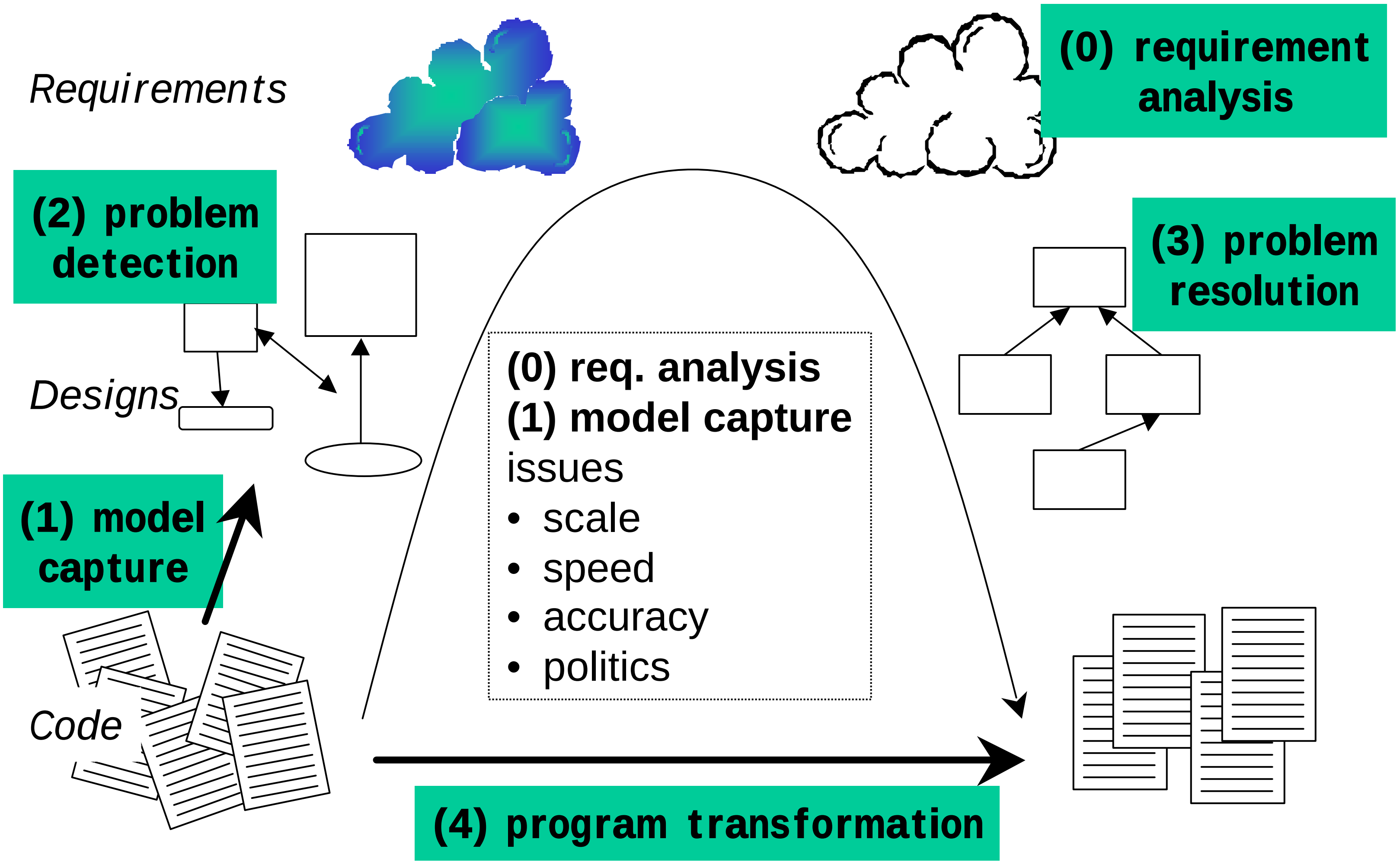
Motivation

Understanding other people's code

- newcomers in the team,
- code reviewing
- original developers left

Generating UML diagrams is NOT reverse engineering
... but it is a valuable support tool

The Reengineering Life-Cycle



Requirements

Designs

Code

(0) requirement analysis

(1) model capture

(2) problem detection

(3) problem resolution

(4) program transformation

(0) req. analysis
(1) model capture
issues
- scale
- speed
- accuracy
- politics

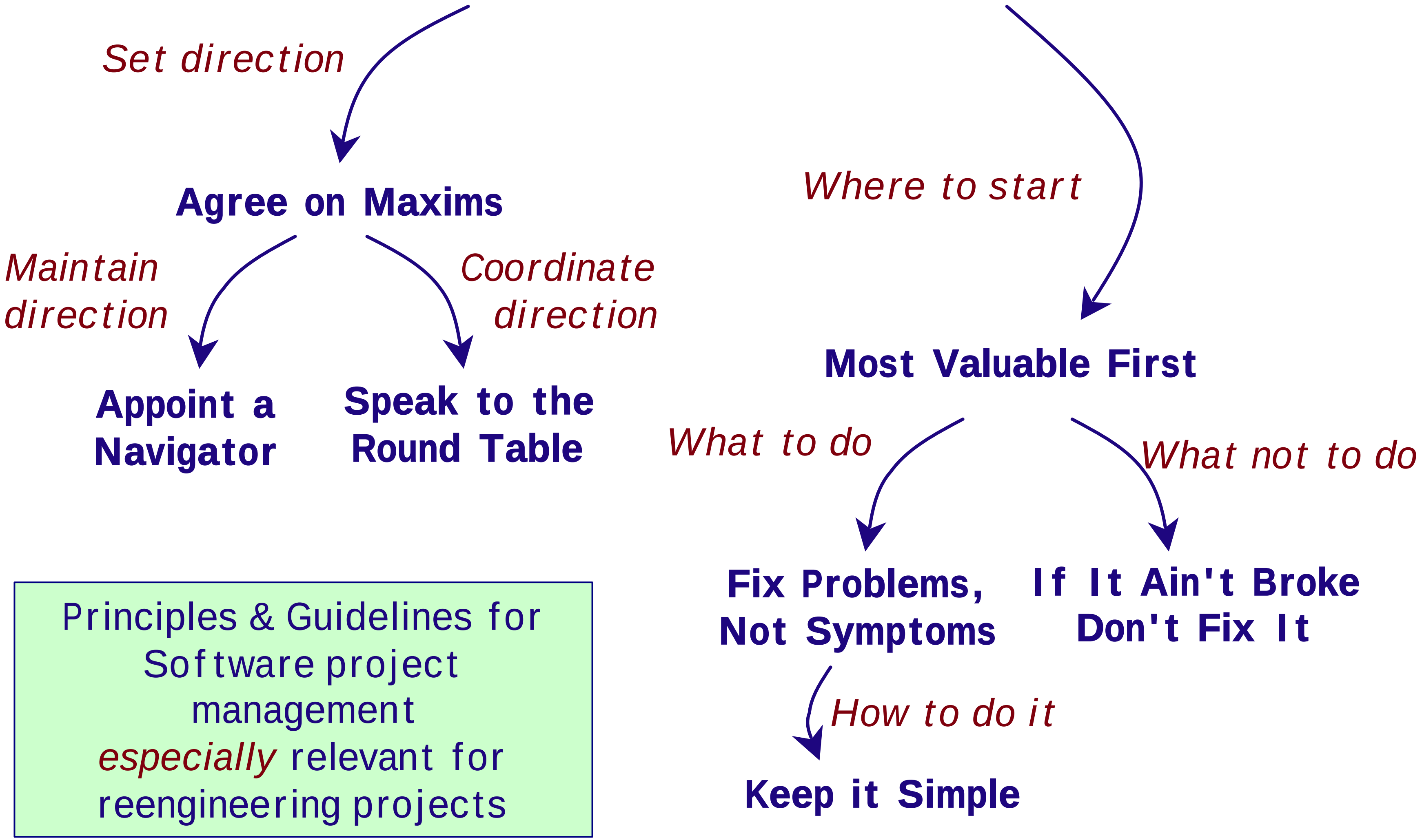
Setting Direction

- *Conflicting interests*
  - technical, ergonomic, economic, political
- Presence/absence *original developers*
- *Legacy architecture*
  - not the best

- *Which problems* to tackle?
  - Interesting vs. important problems?

- Wrap, refactor or rewrite?

Setting Direction



Set direction

Agree on Maxims

Maintain direction

Appoint a Navigator

Coordinate direction

Speak to the Round Table

Where to start

Most Valuable First

What to do

Fix Problems, Not Symptoms

How to do it

Keep it Simple

What not to do

If It Ain't Broke Don't Fix It

Principles & Guidelines for Software project management *especially* relevant for reengineering projects

Most Valuable First

Problem: Which problems should you focus on first?

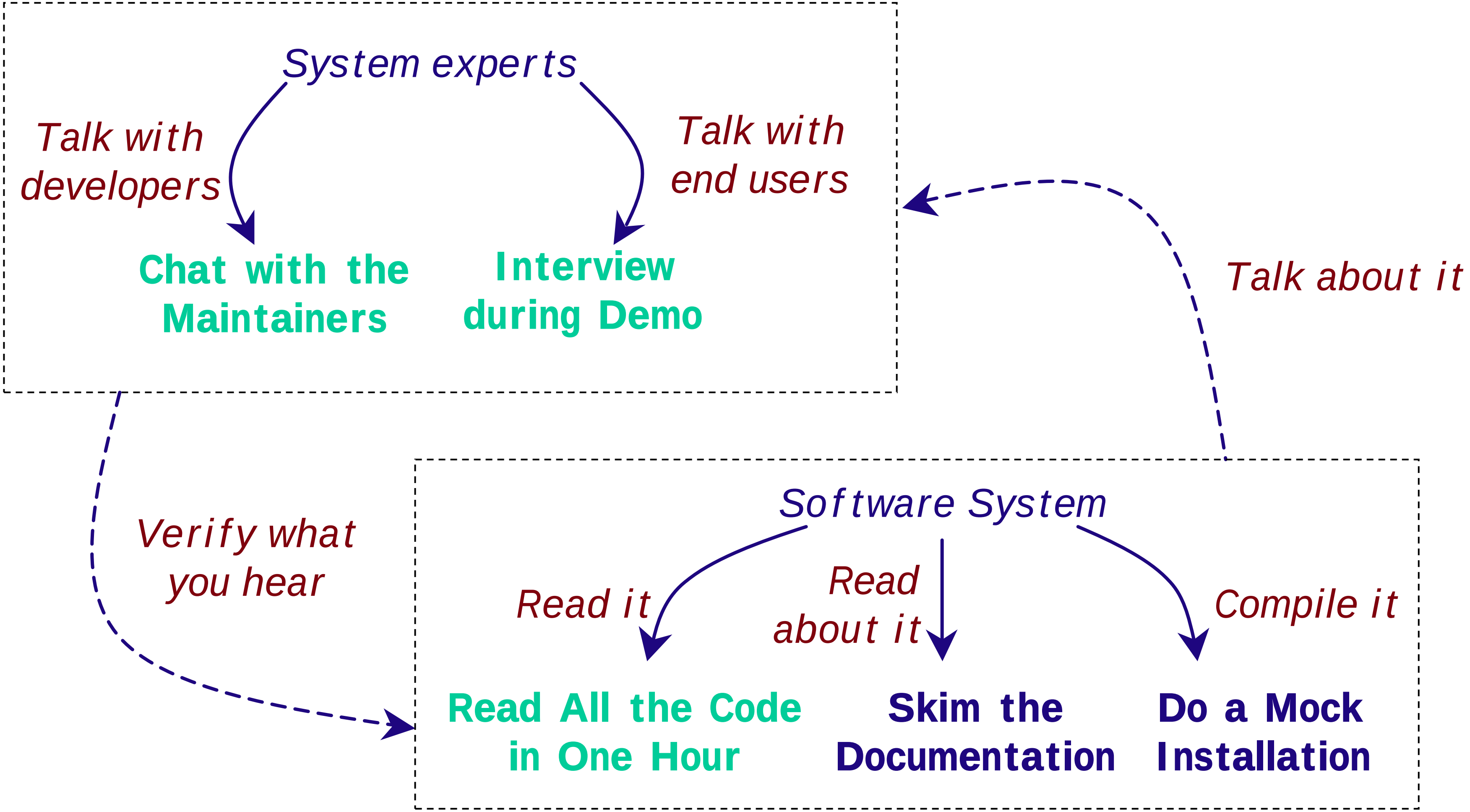
Solution: *Work on aspects that are most* ***valuable*** *to your customer*

- Maximize commitment, early results
  - build confidence
- Difficulties and hints:
  - Which *stakeholder* do you listen to?
  - What *measurable goal* to aim for?
  - Consult *change logs* for high activity
  - Play the *Planning Game*

First Contact

- Where Do I Start?
- Legacy systems are large and complex
  - *Split the system into manageable pieces*
- Time is scarce
  - *Apply lightweight techniques to assess feasibility and risks*
- First impressions are dangerous
  - *Always double-check your sources*

First Contact



System experts

Talk with developers

Chat with the Maintainers

Talk with end users

Interview during Demo

Talk about it

Verify what you hear

Software System

Read it

Read All the Code in One Hour

Read about it

Skim the Documentation

Compile it

Do a Mock Installation

Chat with the Maintainers

Problem: *What are the history and politics of the legacy system?*

Solution: *Discuss the problems with the system maintainers.*

- Documentation will mislead you (various reasons)
- Stakeholders will mislead you (various reasons)
- The maintainers know both the technical and political history

Chat with the Maintainers

Questions to ask:

- Easiest/hardest bug to fix in recent months?
- How are change requests made and evaluated?
- How did the development/maintenance team evolve during the project?
- How good is the code? The documentation?
- Why was the reengineering project started? What do you hope to gain?

> *The major problems of our work are not so much technological as sociological.*
>
> **DeMarco and Lister, Peopleware**

Read all the Code in One Hour

Problem: *How can you get a first impression of the quality of the source code?*

Solution: *Scan all the code in single, short session.*

- Use a checklist
  - code review guidelines, coding styles etc.
- Look for functional tests and unit tests
- Look for abstract classes and root classes that define domain abstractions
- Beware of comments
- Log all your questions!

> *I took a course in speed reading and read "War and Peace" in twenty minutes. It's about Russia.*
>
> Woody Allen

Read all the Code in One Hour

Pros

- Start efficiently
  - code review guidelines, coding styles etc.
- Judge sincerely
  - unbiased view of the software
- Learn the developer's vocabulary

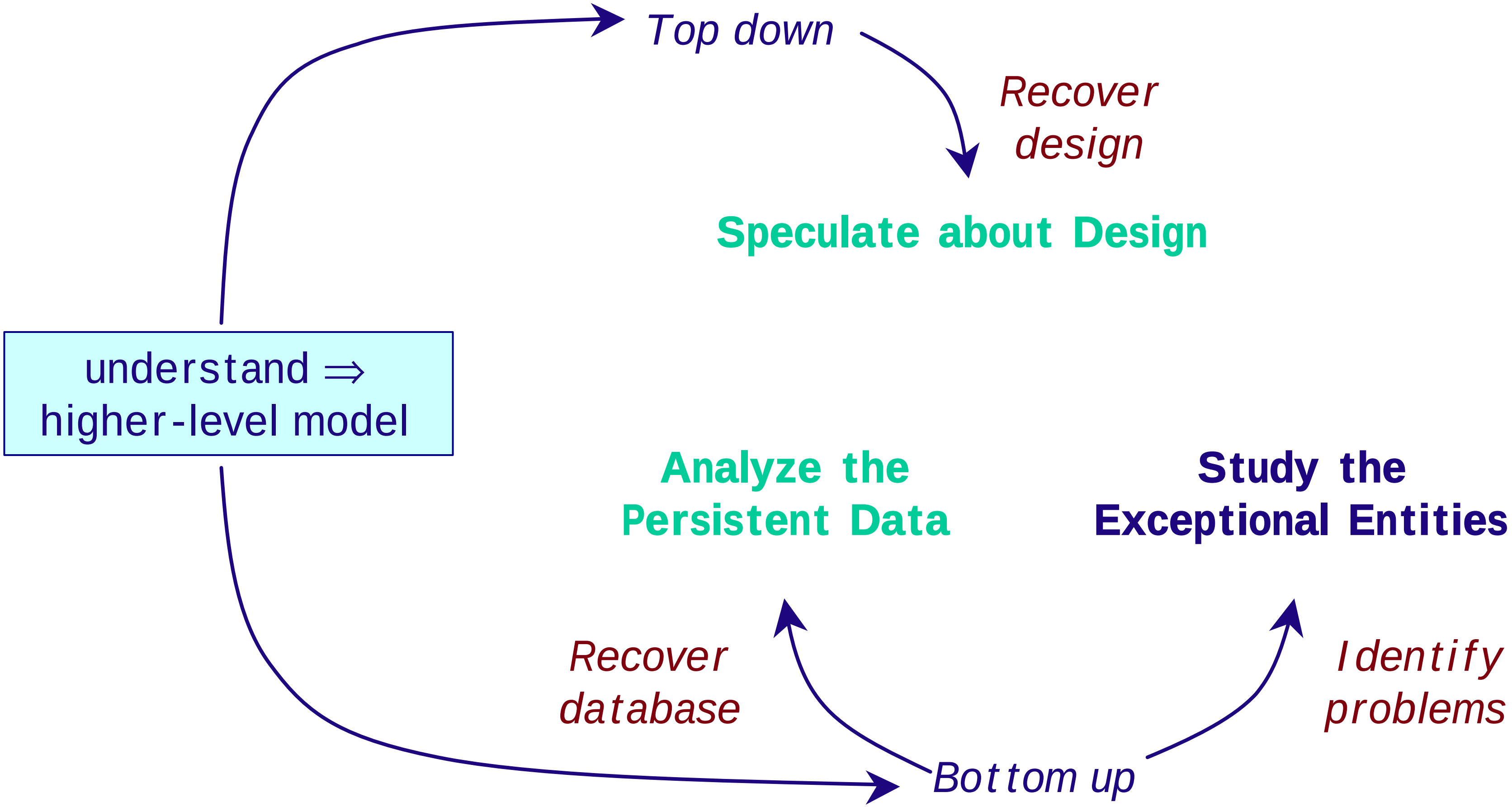
Cons

- Obtain low abstraction
- Doest no scale
- Comments may mislead you

Initial Understanding

- Data is deceptive
  - *Always double-check your sources*
- Understanding entails iteration
  - *Plan iteration and feedback loops*
- Knowledge must be shared
  - *“Put the map on the wall”*
- Teams need to communicate
  - *“Use their language”*

Initial Understanding



understand ⇒ higher-level model

Top down

Recover design

Speculate about Design

Analyze the Persistent Data

Study the Exceptional Entities

Recover database

Identify problems

Bottom up

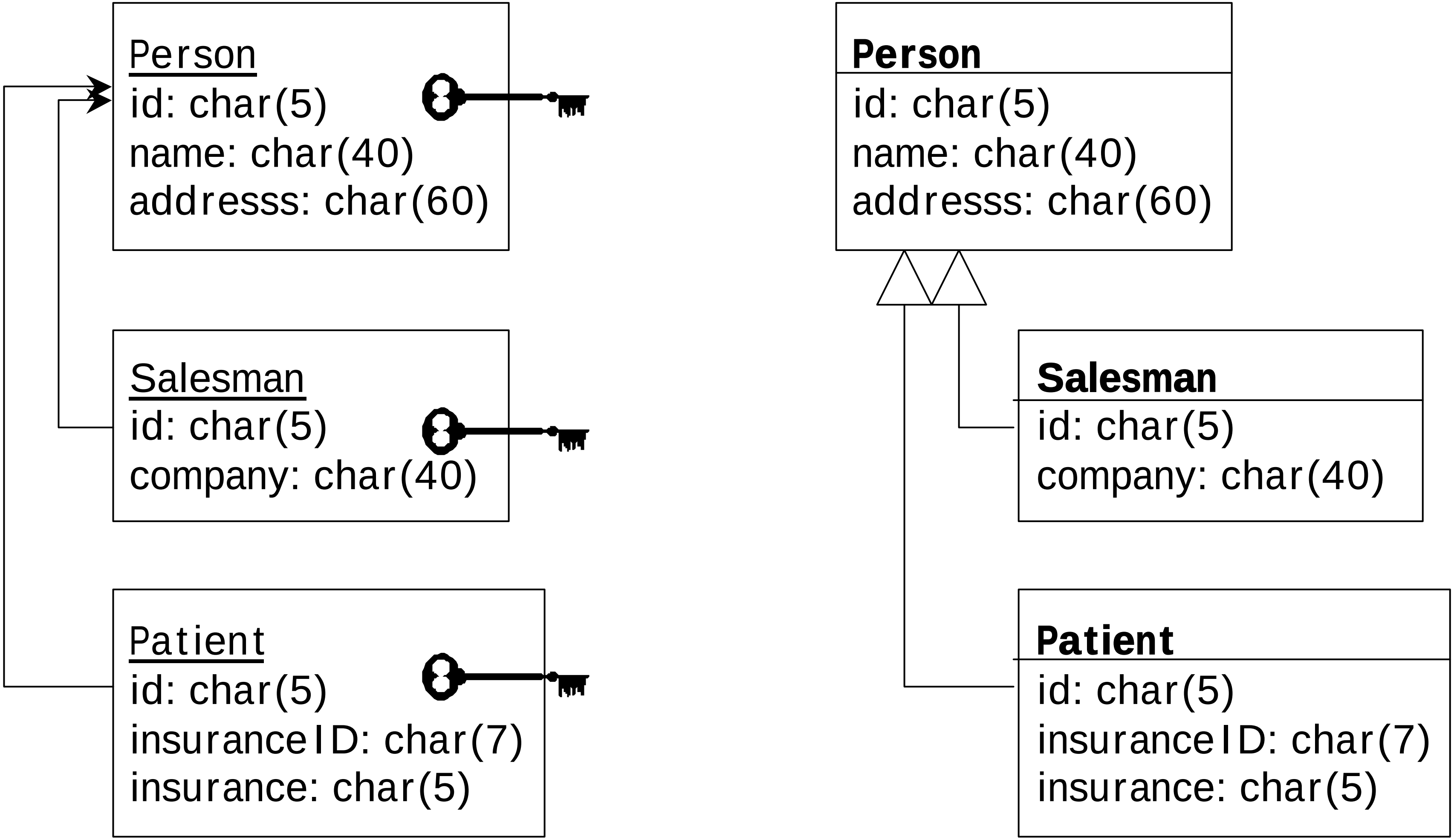
Analyze the Persistent Data

Problem: Which objects represent valuable data?

Solution: Analyze the database schema

- *Prepare Model*
  - tables ⇒ classes; columns ⇒ attributes
  - primary keys
    - naming conventions + unique indices
  - foreign keys (associations between classes)
    - be aware of *synonyms* and *homonyms*
- *Incorporate Inheritance*
  - one to one; rolled down; rolled up
- *Incorporate Associations*
  - association classes (e.g. many-to-many associations)
  - qualified associations
- *Verification*
  - Data samples + SQL statements

Example: One To One



Person
id: char(5)
name: char(40)
addresss: char(60)

Salesman
id: char(5)
company: char(40)

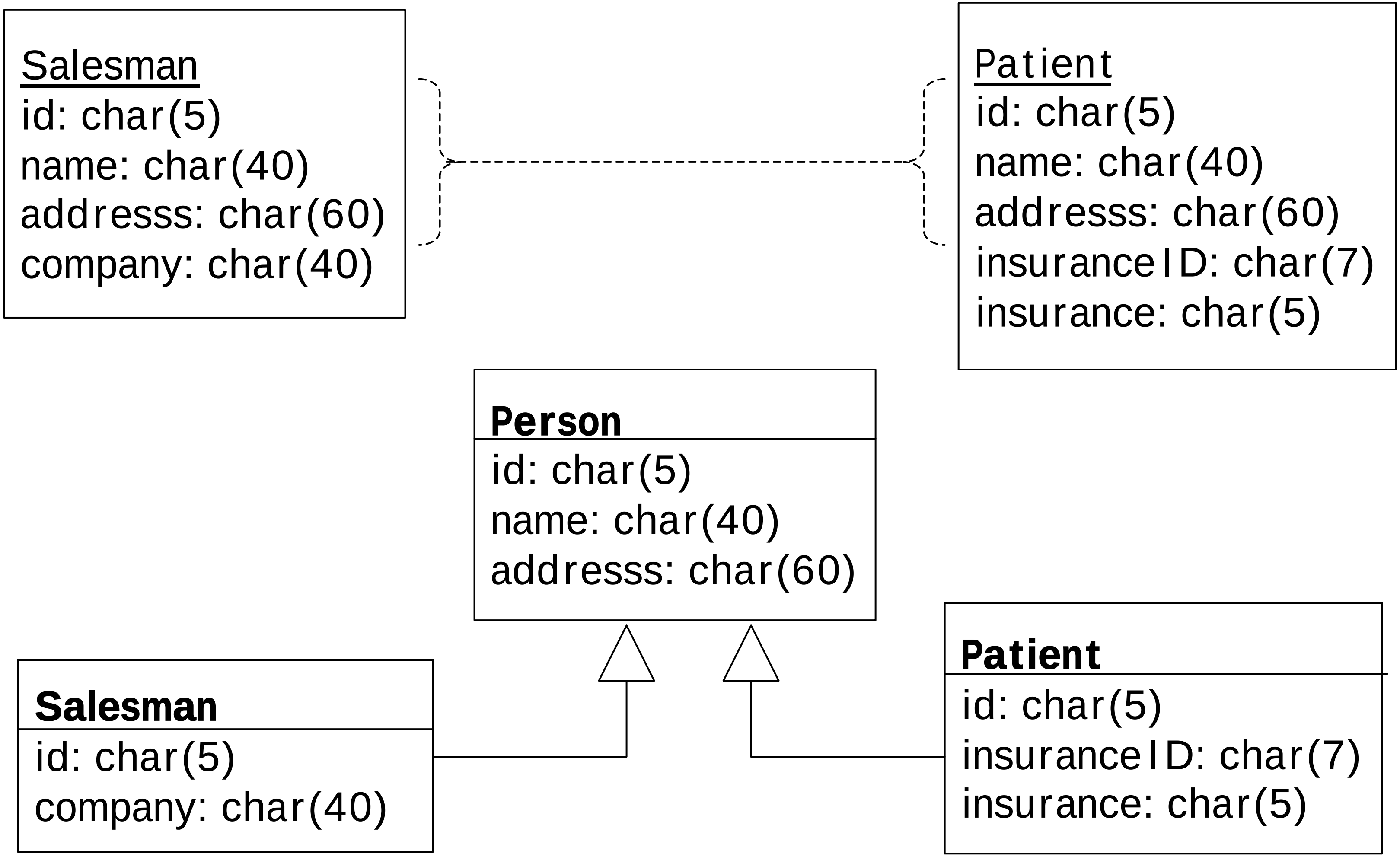
Patient
id: char(5)
insuranceID: char(7)
insurance: char(5)

Person
id: char(5)
name: char(40)
addresss: char(60)

Salesman
id: char(5)
company: char(40)

Patient
id: char(5)
insuranceID: char(7)
insurance: char(5)

Example: Rolled Down



Salesman
id: char(5)
name: char(40)
addresss: char(60)
company: char(40)

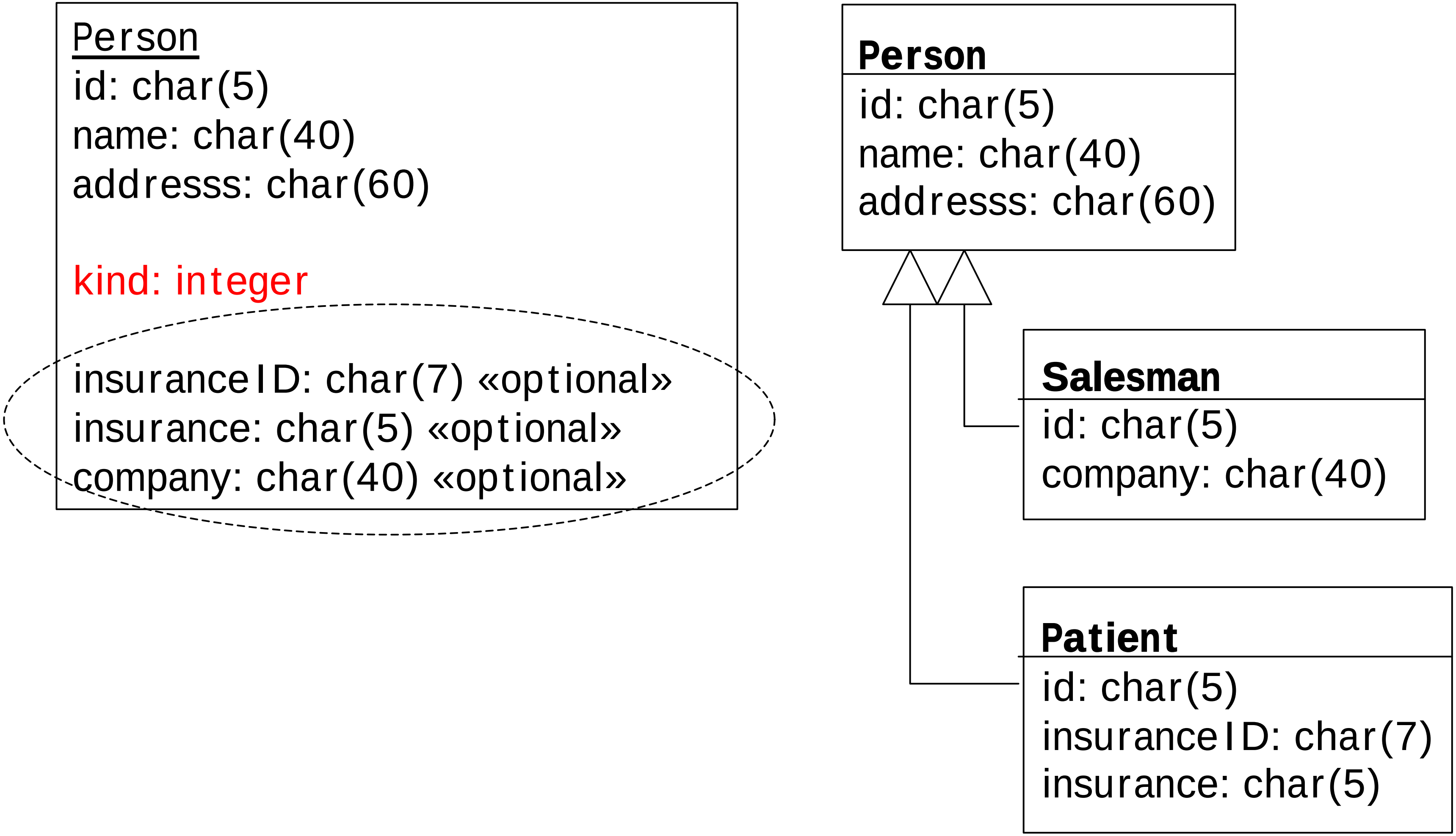
Patient
id: char(5)
name: char(40)
addresss: char(60)
insuranceID: char(7)
insurance: char(5)

Person
id: char(5)
name: char(40)
addresss: char(60)

Salesman
id: char(5)
company: char(40)

Patient
id: char(5)
insuranceID: char(7)
insurance: char(5)

Example: Rolled Up



Person
id: char(5)
name: char(40)
addresss: char(60)

kind: integer

insuranceID: char(7) «optional»
insurance: char(5) «optional»
company: char(40) «optional»

Person
id: char(5)
name: char(40)
addresss: char(60)

Salesman
id: char(5)
company: char(40)

Patient
id: char(5)
insuranceID: char(7)
insurance: char(5)

Study the Exceptional Entities

Problem: How can you quickly identify design problems?

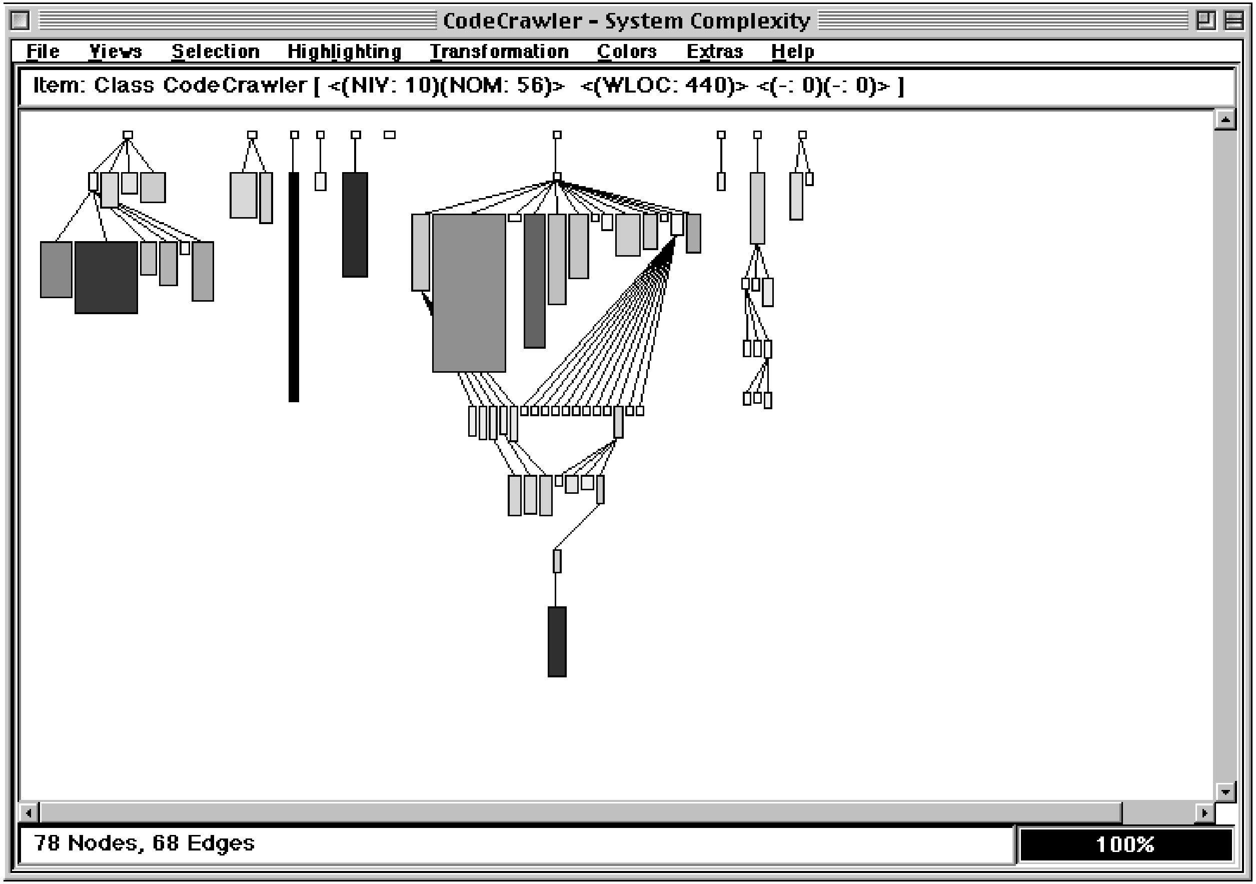
Solution: Measure software entities and study the anomalous ones

- Use simple metrics
- Visualize metrics to get an overview
- Browse the code to get insight into the anomalies

Questions

- Which tools to use?
- Which metrics to collect?
- Which thresholds to apply
- How to interpret the results?
- How to identify anomalies quickly?
- Should I trust numbers?
- What about normal entities?

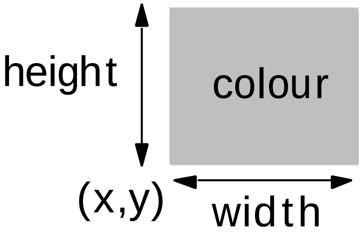
CodeCrawler: Visualizing Metrics



Item: Class CodeCrawler [<(NIV: 10)(NOM: 56)> <(WLOC: 440)> <(-: 0)(-: 0)>]

78 Nodes, 68 Edges

Use *simple* metrics and layout algorithms.



height

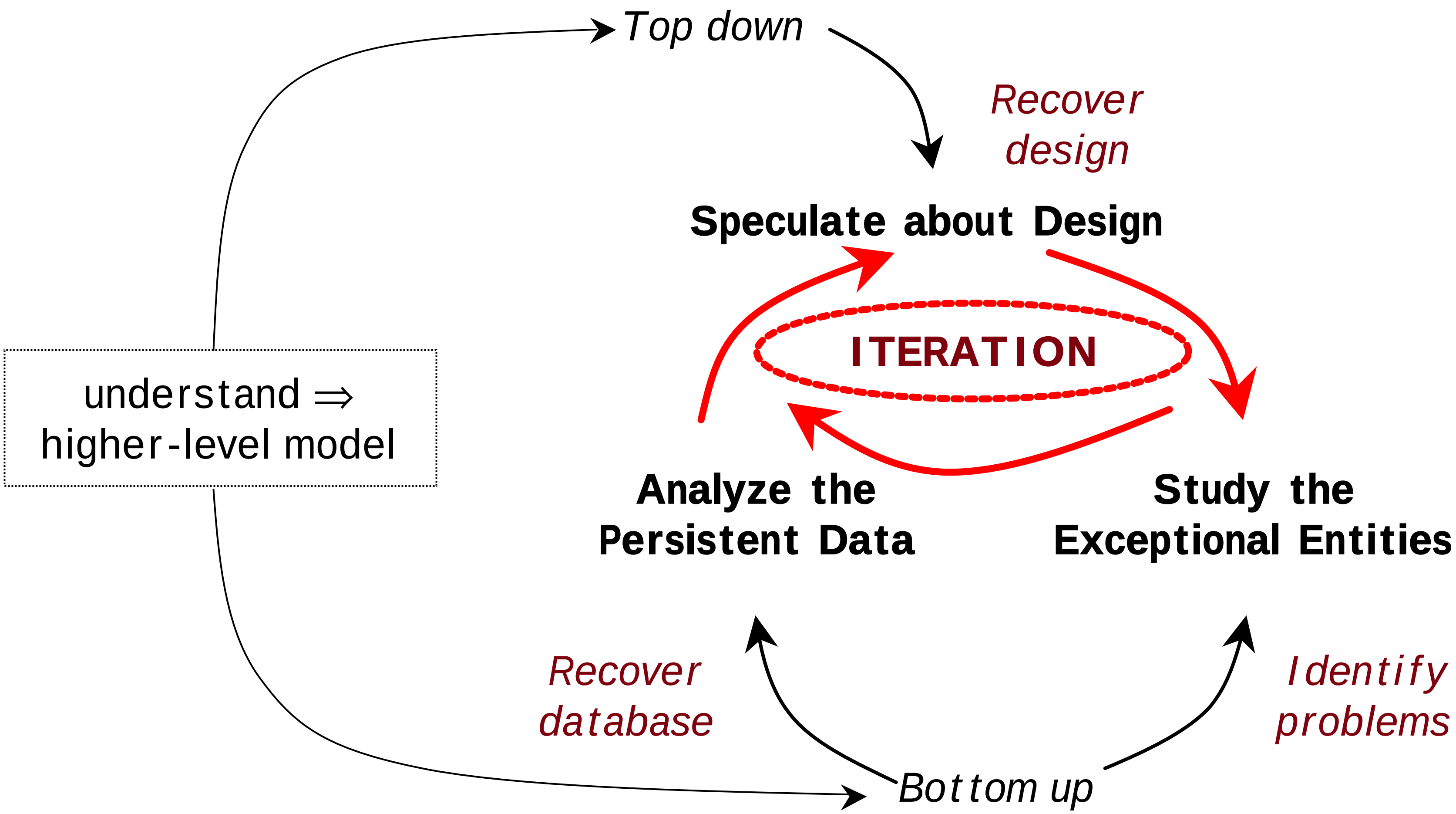
colour

(x,y)

width

Visualize up to 5 metrics per node

Initial Understanding (revisited)


Top down
Recover design
Speculate about Design
ITERATION
understand ⇒ higher-level model
Analyze the Persistent Data
Study the Exceptional Entities
Recover database
Identify problems
Bottom up