# 3. 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
- *Detailed Model Capture*
  - + Tie Code and Questions
  - + Step through the Execution
  - + Look for the Contracts
- Conclusion



# 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***

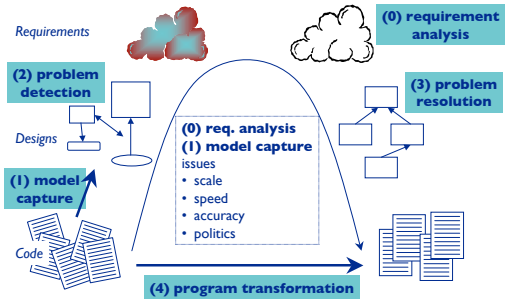
*Understanding* other people's code

(cf. 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



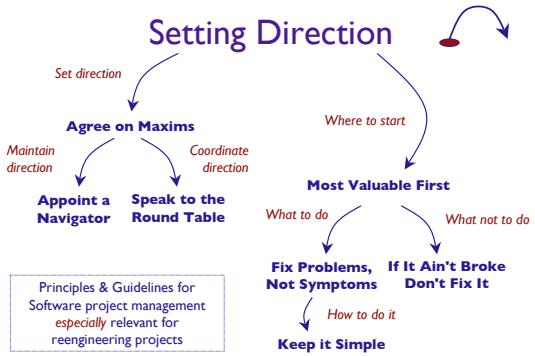



# Forces — Setting Direction

- *Conflicting interests* (technical, ergonomic, economic, political)
- Presence/absence *original developers*
- *Legacy architecture*
- *Which problems* to tackle?
  - + Interesting vs important problems?
  - + Wrap, refactor or rewrite?



# Setting Direction






# 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*
  - + Wrap, refactor or rewrite? — *Fix Problems, not Symptoms*

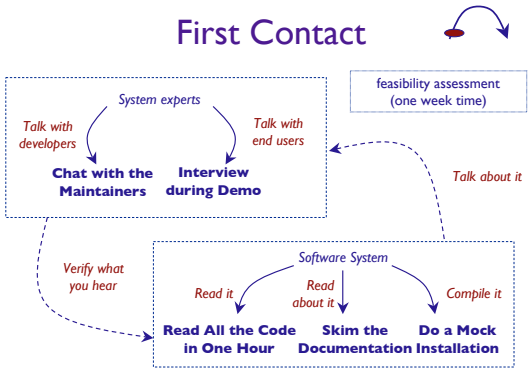


# Forces — First Contact

- 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*
- People have different agendas
  - + *Build confidence; be wary of skeptics*



# First Contact






# 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/maintenence 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 no so much technological as sociological.*

*—DeMarco and Lister, Peopleware '99*




# Read all the Code in One Hour

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

*–Woody Allen*

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!




# Interview during Demo

Problem: *What are the typical usage scenarios?*

Solution: *Ask the user!*

- ... however
  - + Which user ?
  - + Users complain
  - + What should you ask ?

- Solution: interview during demo
  - select several users
  - demo puts a user in a positive mindset
  - demo steers the interview




# First Project Plan

Use *standard templates*, including:

- project scope
  - + see "Setting Direction"
- opportunities
  - + e.g., skilled maintainers, readable source-code, documentation
- risks
  - + e.g., absent test-suites, missing libraries, …
  - + record likelihood (unlikely, possible, likely) & impact (high, moderate, low) for causing problems
- go/no-go decision
- activities
  - + fish-eye view

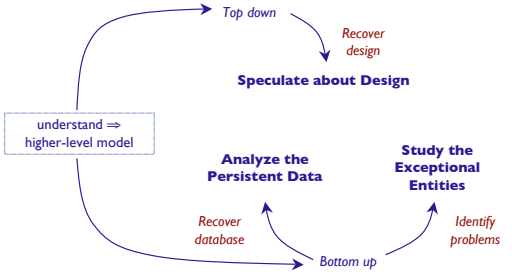



# Forces — Initial Understanding

- 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







# Analyze the Persistent Data

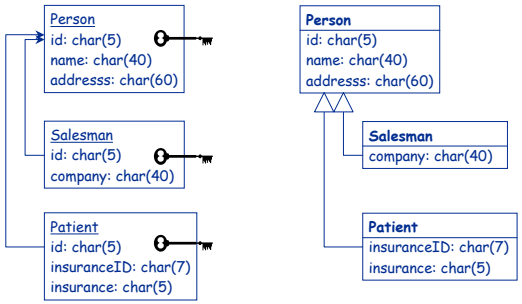
Problem: *Which objects represent valuable data?*

Solution: *Analyze the database schema*

- *Prepare Model*
  - + tables ⇒ classes; columns ⇒ attributes
  - + candidate keys (naming conventions + unique indices)
  - + foreign keys (column types + naming conventions + view declarations + join clauses)
- *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



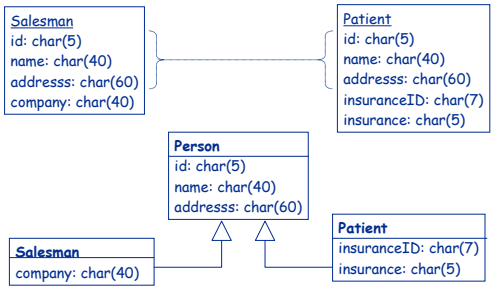

# Example: One To One



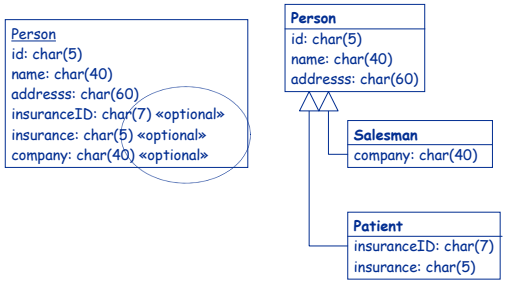




# Example: Rolled Down

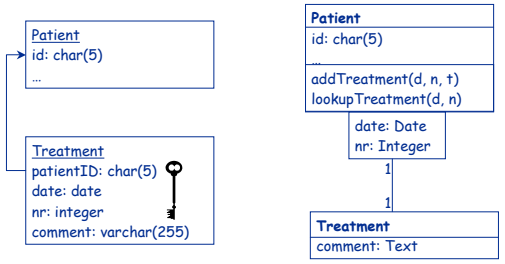


# Example: Rolled Up






# Example: Qualified Association






# Speculate about Design

Problem: *How do you recover design from code?*
Solution: *Develop hypotheses and check them*

- Develop a plausible class diagram and iteratively check and refine your design against the actual code.

*Variants:*

- Speculate about Business Objects
- Speculate about Design Patterns
- Speculate about Architecture



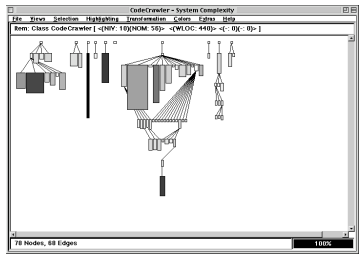
# Study the Exceptional Entities

Problem: *How can you quickly identify design problems?*
Solution: *Measure software entities and study the anomalous ones*

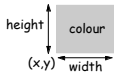
- Use simple metrics
- Visualize metrics to get an overview
- Browse the code to get insight into the anomalies



# Visualizing Metrics



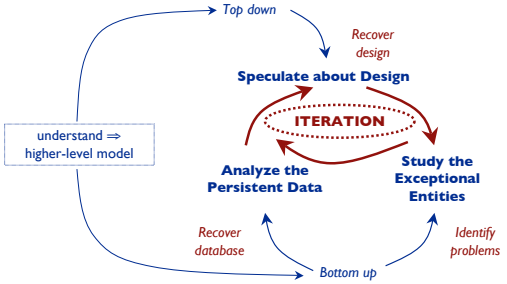
Use *simple* metrics and layout algorithms.




Visualize up to 5 metrics per node



# Initial Understanding (revisited)



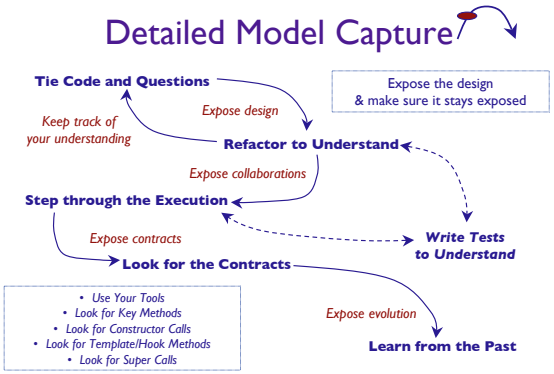



# Forces — Detailed Model Capture

- Details matter
  + *Pay attention to the details!*
- Design remains implicit
  + *Record design rationale when you discover it!*
- Design evolves
  + *Important issues are reflected in changes to the code!*
- Code only exposes static structure
  + *Study dynamic behaviour to extract detailed design*



# Detailed Model Capture






# Tie Code and Questions

Problem: *How do you keep track of your understanding?*
Solution: *Annotate the code*

- List questions, hypotheses, tasks and observations.
- Identify yourself!
- Use conventions to locate/extract annotations.
- Annotate as *comments*, or as *methods*

# Refactor to Understand

Problem: *How do you decipher cryptic code?*

Solution: *Refactor it till it makes sense*

- Goal (for now) is to *understand*, not to reengineer
- Work with a *copy* of the code
- Refactoring requires an adequate test base
  - + If this is missing, *Write Tests to Understand*
- Hints:
  - + Rename attributes to convey roles
  - + Rename methods and classes to reveal intent
  - + Remove duplicated code
  - + Replace condition branches by methods




# Step Through the Execution

Problem: *How do you uncover the run-time architecture?*

Solution: *Execute scenarios of known use cases and step through the code with a debugger*

- Difficulties
  - + OO source code exposes a *class hierarchy*, not the run-time *object collaborations*
  - + Collaborations are spread throughout the code
  - + Polymorphism may hide which classes are instantiated
- Focussed use of a debugger can expose collaborations




# Look for the Contracts

Problem: *Which contracts does a class support?*

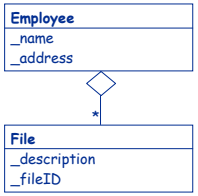
Solution: *Look for common programming idioms*

- Look for "*key methods*"
  - + Intention-revealing names
  - + Key parameter types
  - + Recurring parameter types represent temporary associations
- Look for *constructor* calls
- Look for *Template/Hook* methods
- Look for *super* calls
- *Use your tools!*




# Constructor Calls: Stored Result

```
public class Employee {
   private String _name = "";
   private String _address = "";
   public File[ ] files = { };
…
public class File {
   private String _description = "";
   private String _fileID = "";
…
```

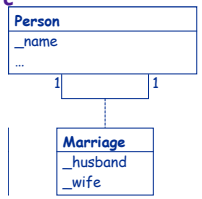



```
public void createFile (int position,
  String description, String identification) {
  files [position] = new File (description, identification);
  }
```




# Constructor Calls: "self" Argument

```
public class Person {
   private String _name = "";
…
public class Marriage {
   private Person _husband, _wife;
   public Marriage (Person husband,
            Person wife) {
            _husband = husband;
            _wife = wife;}
…
```



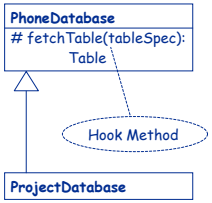

```
Person::public Marriage marryWife (Person wife) {
  return new Marriage (this, wife);
}
```




# Hook Methods

```
public class PhoneDatabase {
    …
    protected Table fetchTable
            (String tableSpec) {
    //tableSpec is a filename; parse it as
    //a tab-separated table representation
    …};

public class ProjectDatabase
            extends PhoneDataBase {
    …
    protected Table fetchTable (String tableSpec) {
    //tableSpec is a name of an SQLTable;
    //return the result of SELECT * as a table
    …};
```

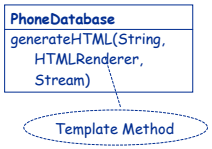






# Template / Hook Methods

```
public class PhoneDatabase {
    …
    public void generateHTML
            (String tableSpec,
            HTMLRenderer aRenderer,
            Stream outStream) {
        Table table = this.fetchTable (tableSpec);
        aRenderer.render (table, outStream);}
…};
```




```
public class HTMLRenderer {
    …
    public void render (Table table, Stream outStream) {
    //write the contents of table on the given outStream
    //using appropriate HTML tags
…}
```




# Learn from the Past

Problem: *How did the system get the way it is?*

Solution: *Compare versions to discover where code was removed*

- *Removed* functionality is a sign of design evolution
- Use or develop appropriate *tools*
- Look for signs of:
  - + *Unstable design* — repeated growth and refactoring
  - + *Mature design* — growth, refactoring and stability




# Conclusion

- *Setting Direction* + *First Contact*
  ⇒ First Project Plan
- *Initial Understanding* + *Detailed Model Capture*
  - + Plan the work … and Work the plan
  - + Frequent and Short Iterations
- Issues
  - + scale
  - + speed vs. accuracy
  - + politics