Reverse Engineering

Tudor Gîrba
www.tudorgirba.com

Reverse engineering *is analyzing a subject system to:*

identify components and their relationships, and create more abstract representations.

Chikofky & Cross, 90

Elliot Chikofsky and James Cross II, “Reverse Engineering and Design Recovery: A Taxonomy,” IEEE Software, vol. 7, no. 1, January 1990, pp. 13-17.
http://dx.doi.org/10.1109/52.43044

Why reverse engineer?

Thanks to Orla Greevy for pointing out this story of extremely successful reverse engineering.

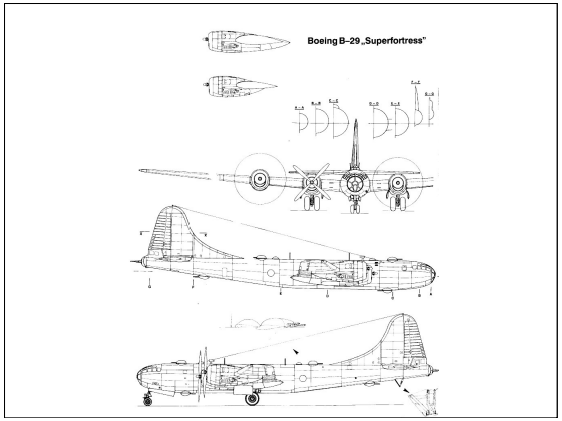
The B-29 was the main bomber of US Air forces and it provided the strategic advantage of reaching over the Pacific Ocean.

This three billion dollar project was the largest government commitment ever to a single project, including the Atomic Bomb.
http://en.wikipedia.org/wiki/B-29
http://en.wikipedia.org/wiki/Tupolev_Tu-4
http://www.rb-29.net/HTML/03RelatedStories/03.03shortstories/03.03.10contss.htm

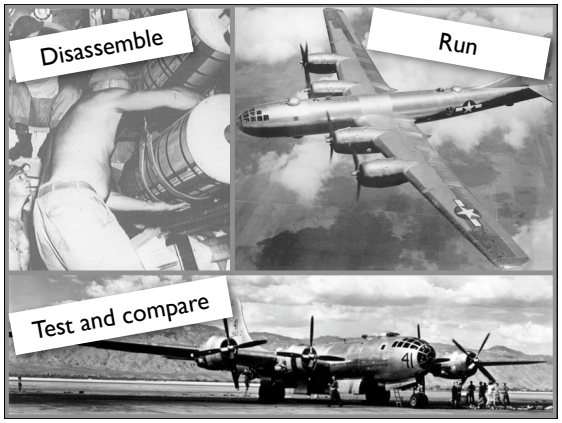
During 1944, 3 bombers had to land in Russia after bombing missions in Japan. The Russians refused to return them.

The B-29 was not a legacy system, but:
- it was tremendously valuable
- it was unknown to the Russians
- it was estimated that to build one from scratch would take about 5 years

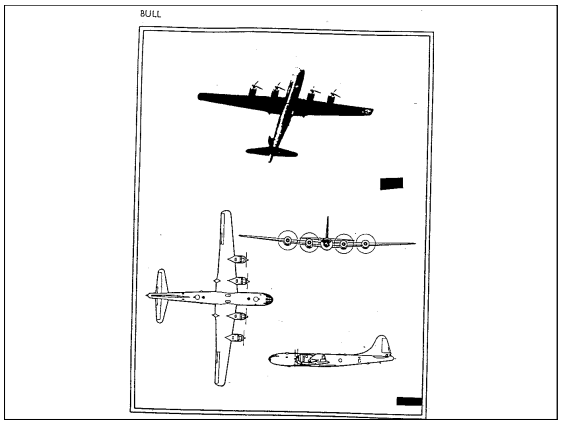
http://en.wikipedia.org/wiki/B-29
http://en.wikipedia.org/wiki/Tupolev_Tu-4



The challenge was to understand the planes well enough to be able to build a factory that would build them. This had to go beyond just the structure.



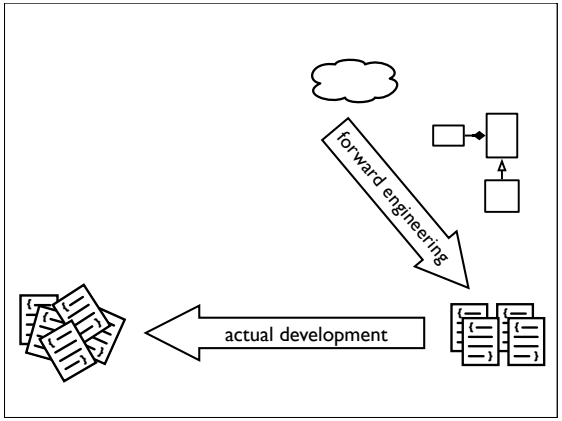
They approached the problem from several directions:
- one plane was disassembled into pieces,
- one plane was used for flying, and
- one plane was used as a comparison model and for training pilots.



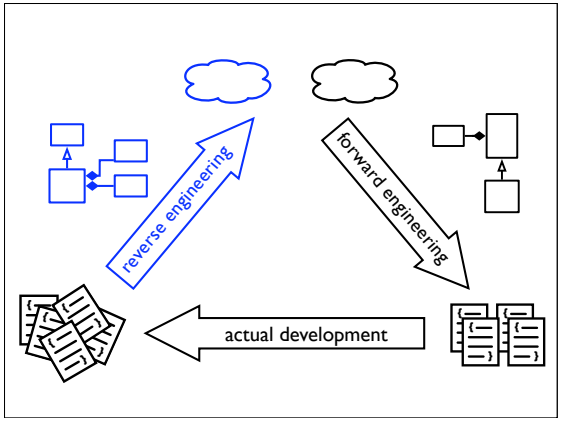
They eventually managed to build their own plans.

The Russians reverse engineered the plans in 1 year and produced the first piece 1 year later.
Tu-4 first flew on May 19, 1947. Serial production started immediately, and the type entered large scale service in 1949.
It is said that they copied even the flaws, as the engines were as unreliable as in the American version

Why reverse engineer software?



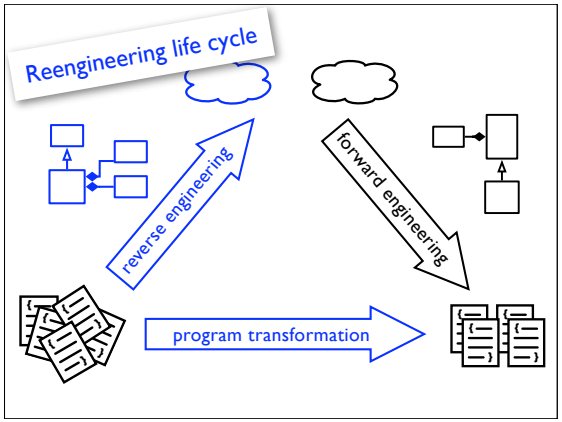
The problem is that in most projects, the actual development happens only at the code level, with only little documentation, and several years later the system is not tidy anymore.



Forward Engineering is the traditional process of moving from high- level abstractions and logical, implementation-independent designs to the physical implementation of a system.

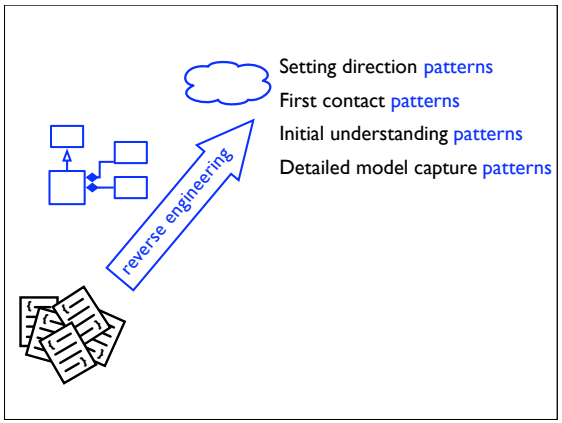
Reverse Engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction.

Elliot Chikofsky and James Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," IEEE Software, vol. 7, no. 1, January 1990, pp. 13—17.



Reengineering ... is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.

Elliot Chikofsky and James Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," IEEE Software, vol. 7, no. 1, January 1990, pp. 13—17.

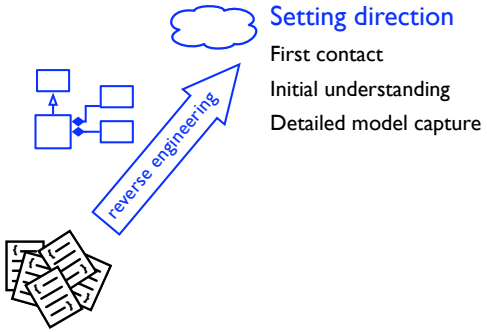


Reverse engineering is an iterative process.
Set the direction to guide your way.
Start with exploratory actions to get an impression.
Speculate and check your assumptions.
Go into details.

short intermezzo

What are patterns?

Patterns are recurrent solutions to problems that occur over and over.



Why set direction?

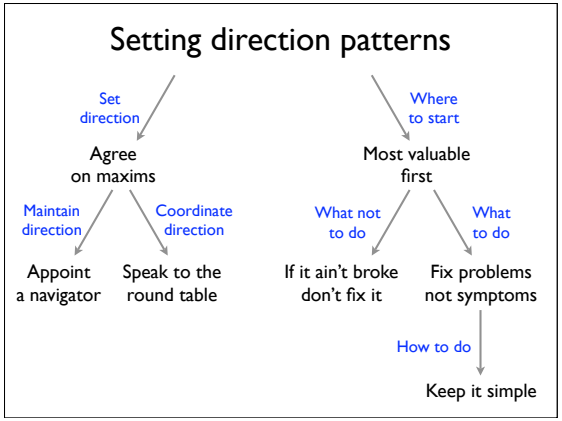
Reverse engineering is influenced by different factors:
- Conflicting interests (technical, economical, political).
- The presence or absence of original developers.
- Legacy architecture
- Interesting vs important problems

Important questions:
- Which problems to tackle?
- Wrap, refactor or rewrite?

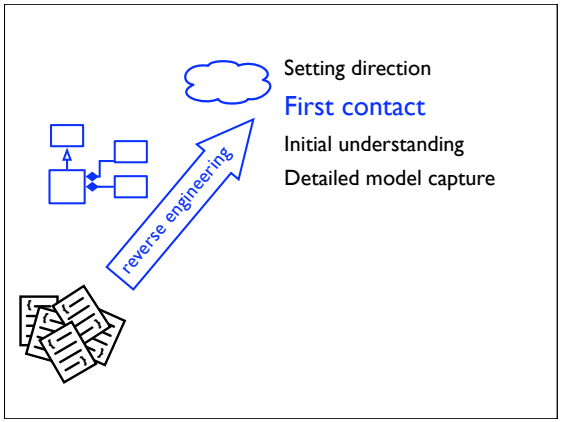
You got to be careful if you don't know where you're going, because you might not get there.

Yogi Berra

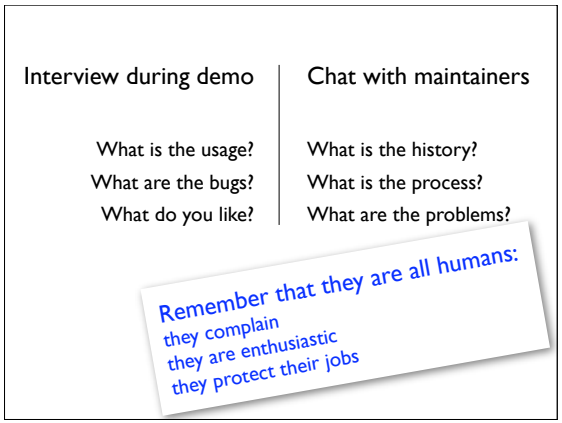
Another nice fragment :
'Would you tell me, please, which way I ought to go from here?'
'That depends a good deal on where you want to get to,' said the Cat.
'I don't much care where ... ' said Alice.
'Then it doesn't matter which way you go,' said the Cat.
'... so long as I get somewhere,' Alice added as an explanation.
~ Lewis Carroll, Alice's Adventures in Wonderland



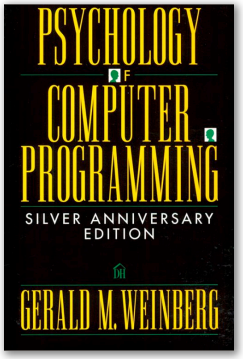
Most Valuable First
Problem: Which problems should you focus on first?
Solution: Work on aspects that are most valuable to your customer.
Maximize commitment by aiming for early results; build confidence.
Difficulties and hints:
- Which stakeholder do you listen to?
- What measurable goal to aim for?
- Consult change logs for high activity
- Play the Planning Game
- Wrap, refactor or rewrite? — Fix Problems, not Symptoms



Setting direction summary:
Given the large amount of data to be processed, it is important to not lose focus.



Other questions to ask a maintainer:
How long was your project going on?
Who worked on the project?
What was the most interesting bug you had to fix?
Why was the reengineering effort started?
How do you release?
How do you plan what to do?
How do you test?



Weinberg was among the first to point out that programming is a human activity. In one of his stories, he points out how chatting around a vending machine helped solving problems. You can read about the vending machine story here: http://www.stsc.hill.af.mil/crosstalk/2008/08/0808Cockburn.html

Read all code in one hour ?

100'000 lines of code
* 2 = 200'000 seconds
/ 3600 = 56 hours
/ 8 = 7 days

Supposing that you read one line in 2 seconds, it would take 7 working days to read 100'000 lines of code. So, what good would it make to read all code in one hout?

```
/**
 * We hang our heads in shame. There are still bugs in ArgoUML
 * and/or GEF that cause corruptions in the model.
 * Before a save takes place we repair the model in order to
 * be as certain as possible that the saved file will reload.
 * TODO: Split into small inner classes for each fix.
 *
 * @return A text that explains what is repaired.
 */
```

ArgoUML

Read all code in one hour
Problem: How to get a first impression of the code?
Solution: Scan all code in one short session.
Issues:
- limit your time, and isolate from interruptions.
- use a checklist.
- look for root and abstract classes.
- beware of misleading comments.
- log your questions and findings.

```
updateTypeAccordingToEntities

    "-- ugly code, will change once we move to CollectiveBehavior --"

    | common wantedType class |
    common := self commonMetaDescription.
    wantedType := (common name, 'Group') asSymbol.
    self metaDescription name == wantedType ifTrue: [ ^self ].
    class := AbstractGroup allSubclasses
        detect: [ :each | each asMetaDescription name == wantedType ]
        ifNone: [ ^self changeTypeToDefaultType ].
    self changeTypeTo: class.
```

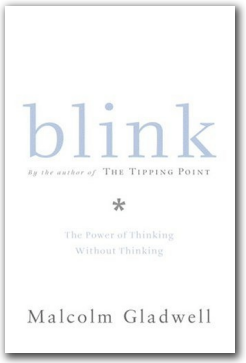
Moose

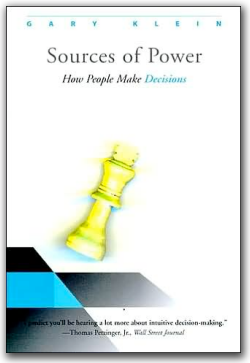
Get a first impression, but do not rely on it.
Use it for guiding your future investigations.

I took a course in speed reading and read "War and Peace" in twenty minutes.

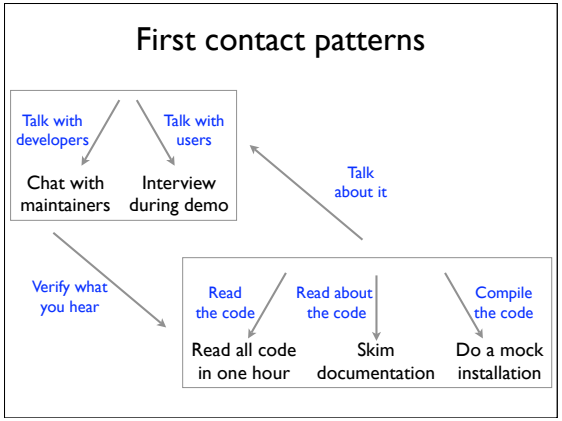
It's about Russia.

Woody Allen

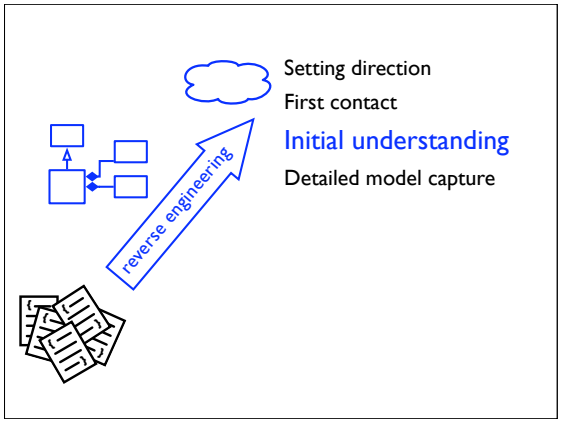




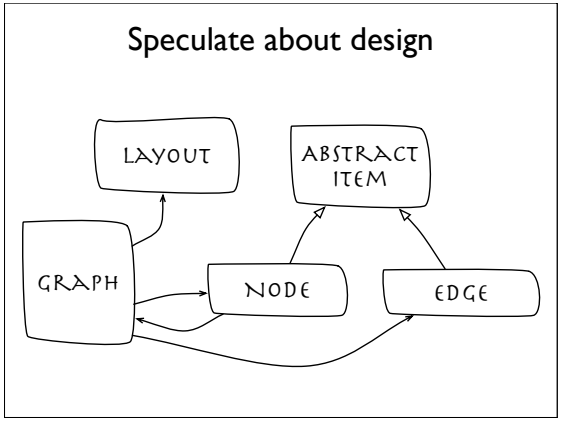
Why read all code in 1 hour? Because we have a built-in mechanism to think fast.



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.
Rule of thumb: Do not let the first contact last more than one week time.



First contact summary:
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 overview - fish-eye view

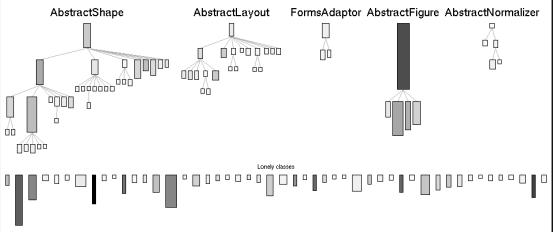


The picture shows a possible design of a graph-based visualization tool.

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.

Identify exceptional entities



The picture shows the System Complexity View of Mondrian.

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

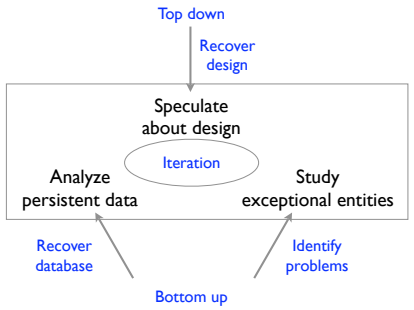
Use simple metrics.
Visualize metrics to get an overview.
Browse the code to get insight into the anomalies.

Identify exceptional entities

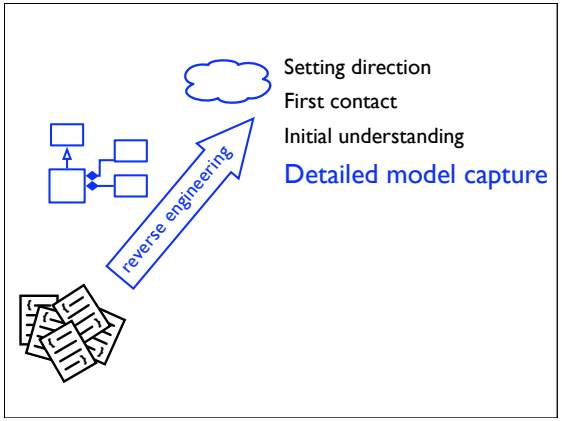
```
for i in $( ls ); do
   echo `wc -l $i` >> temp
done
sort -nr temp | head -10
```

You do not need fancy tools to get simple answers. The above program was written in 5 minutes by Jorge Ressia.

Initial understanding patterns



Knowledge must be shared.
Team need to communicate. “Use their language”
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”.



Initial understanding summary: Speculate about Design, Study Exceptional Entities.
Iterate! ... and start going into details.

Expose the design and make sure it stays exposed.

To understand: refactor

Refactor to undertand
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.

To understand: refactor
write tests

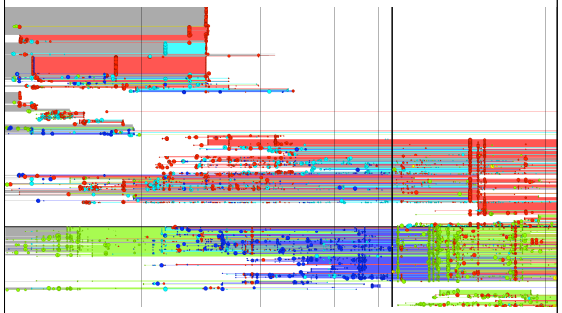
You can encode your assumptions as tests and execute them against the system.

To understand: refactor
write tests
step through execution

Step through 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.

Tests can also be used as scenario generators. If tests are missing Write Tests to Understand.
Put breakpoints in the code.
Focused use of a debugger can expose collaborations.
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

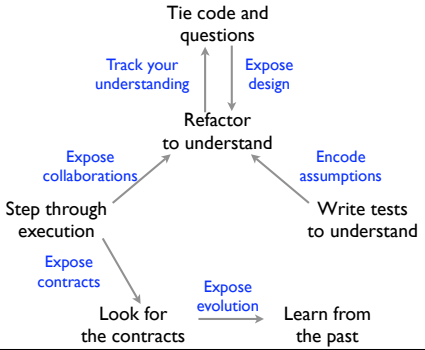
Learn from the past



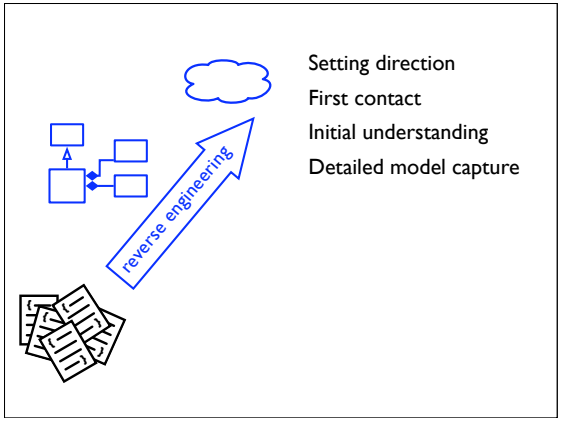
Problem: How did the system get the way it is?
Solution: Compare versions to discover where code was removed.

Removed functionality is a sign of design evolution.
Use or develop appropriate tools.
Chat with maintainers about the reasons of change.
Look for signs of:
- Unstable design — repeated growth and refactoring
- Mature design — growth, refactoring and stability

Detailed model capture patterns



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.



Setting direction
First contact
Initial understanding
Detailed model capture

Reverse engineering is creating high level views.
Plan the work and work the plan.
Iterate.
Issues:
- politics
- speed vs. accuracy
- scale

Choose your tools and use them.

Tudor Gîrba
www.tudorgirba.com