## Reverse Engineering



Reverse engineering is analyzing a subject system to:

identify components and their relationships, and

create more abstract representations.

Chikofky & Cross, 90



## Why reverse engineer?



## In 1944, 3 B-29 had to land in Russia

# Requirement: Copy everything fast!



# Approach: disassemble, run, test



# TU-4 Result: 105,000 pieces reassembled in 2 years



# Reading code...

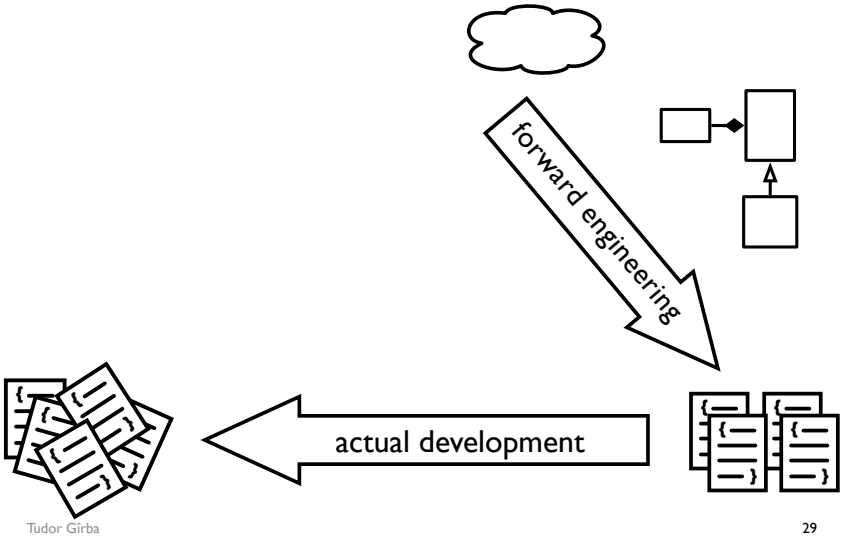
100'000 lines of code

* 2 = 200'000 seconds

/ 3600 = 56 hours

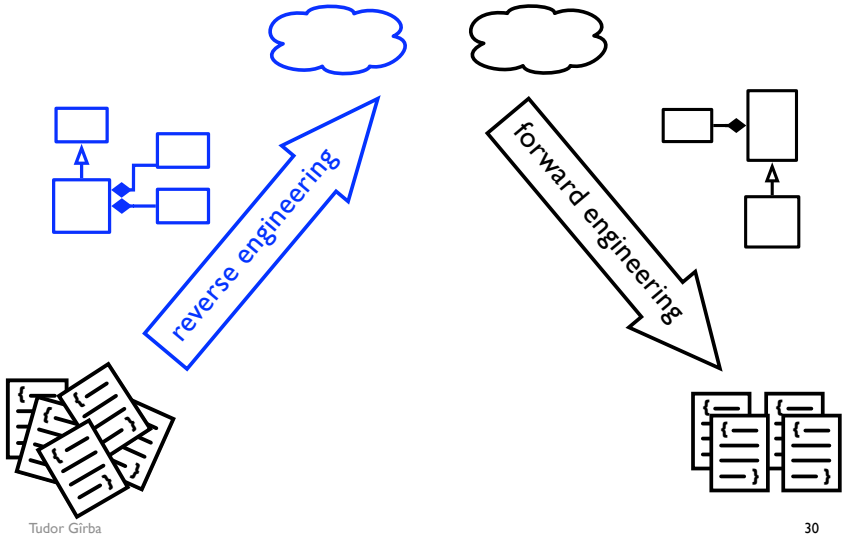
/ 8 = 7 days

# How development happens



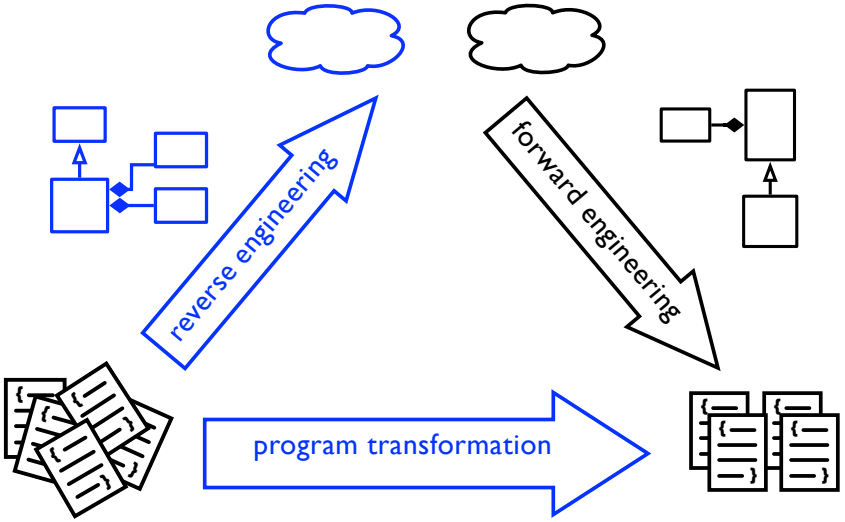



# Reengineering life cycle






# Reengineering life cycle






## 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

Synthesize higher abstractions
- identify latent abstractions in software

Facilitate reuse
- detect candidate reusable artifacts and components

**Chikofsky and Cross [in Arnold, 1993]**

## Reverse Engineering Techniques

- Redocumentation
  - pretty printers
  - diagram generators
    - e.g. Together
  - cross-reference listing generators
    - e.g. IDEA, SNiFF+, Source Navigator
- Design recovery
  - software metrics
  - browsers, visualization tools
  - static analyzers
  - dynamic (trace) analyzers



## Reverse engineering Patterns

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

- Even if design documents exist, they are typically out of sync with reality.

Example: **Refactor to Understand**



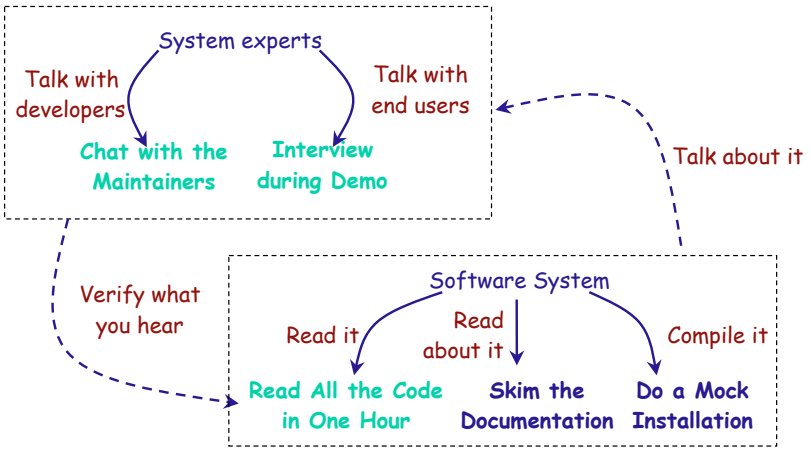
## I. First Contact: the Forces

### 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



## Read All Code in 1 Hour

- Entities that seem interesting
- Suspicious Coding Styles
- Abstract Classes and Interfaces
- Singletons
- Large Structures
- Comments

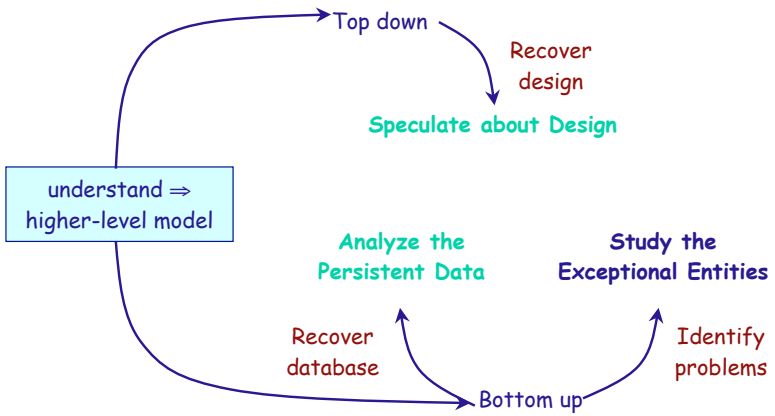


## II. Initial Understanding: the Forces

- 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



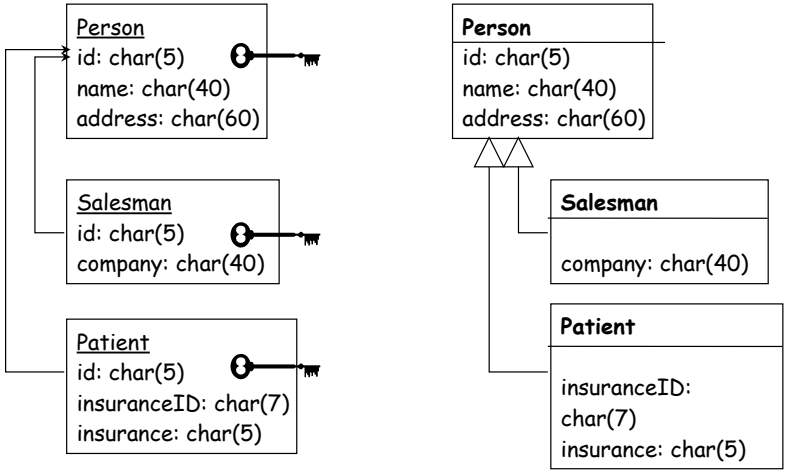



## Analyze the Persistent Data

Problem: Which objects represent valuable data?
Solution: Analyze the database schema

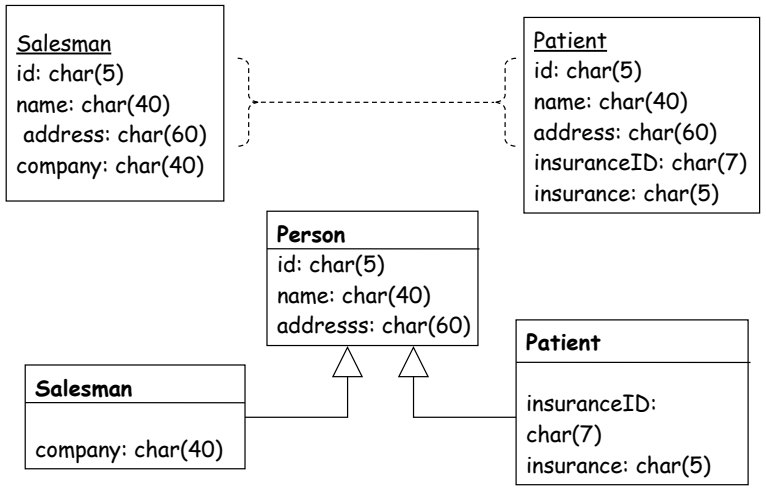
- Prepare Model
  - tables ⇒ classes; columns ⇒ attributes
  - primary keys
    - naming conventions + unique indices
  - foreign keys (associations between classes)
    - be aware of synonyms and homonyms
- Incorporate Inheritance
  - one to one; rolled down; rolled up
- Incorporate Associations
  -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

## Example: One To One



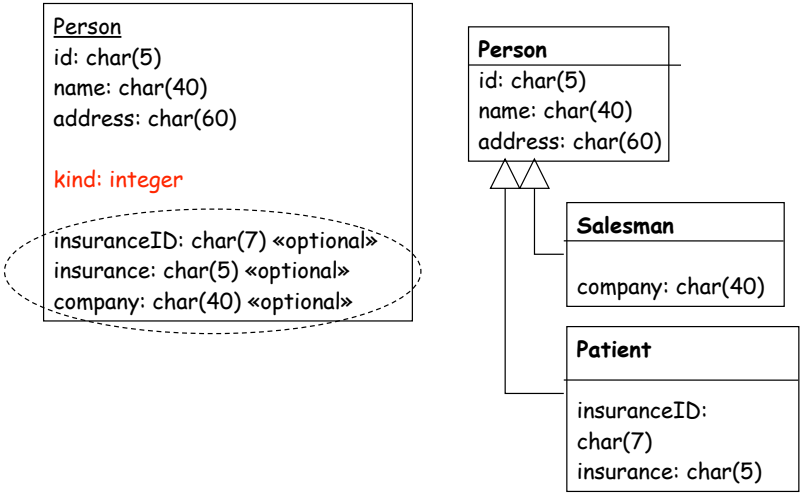



## Example: Rolled Down






## Example: Rolled Up






## Study the Exceptional Entities

Problem: How can you quickly identify key points in design?
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?



## What is a Metric?

- A precise numerical value assigned to an entity.

  Entity = product, resource, process



## Object-Oriented Product Metrics

- Size
- Structural Complexity
- Coupling
- Cohesion
- Inheritance



## Weighted Method Count

- Definition

  WMC = SUM($c_i$), $c_i$= complexity of each method $m_i$

- Interpretation
  - Time and effort for maintenance
  - The higher the WMC for a class, the higher the influence on the subclasses
  - A high WMC reduces the reuse probability for the class

## Number of Children

- Definition
  NOC = number of direct subclasses
- Interpretation
  - higher NOC, higher reuse potential
  - higher NOC, a higher probability of an improper use of inheritance
  - higher NOC, a higher impact/influence on the overall design, including testing effort



## Depth of Inheritance Tree

- Definition
  DIT = depth of a class in the inheritance graph
- Interpretation
  - the higher DIT, the lower the understandability of the class
  - the higher DIT, the more complex the class
  - the higher DIT, the higher the potential reuse from the superclasses



## Response For a Class

- Definition
  $R_i$ = set of methods called by M

  RS = {M} ∪ {$R_i$}; RFC = | RS |
- Interpretation
  - higher RFC, tests are more difficult to perform
  - measure of complexity
  - at the same time a measure of coupling with other classes



## Coupling Between Objects

- Definition
  - the number of other classes to which the measured class is coupled
- Interpretation
  - high CBO hampers reuse in another application
  - high CBO, a higher sensitivity to changes
  - high CBO, a rigorous testing

## Number of Called Classes

- Definition
  - FANOUT = SUM(FANOUT$_i$), FANOUT$_i$= classes from which each user defined method $m_i$ calls methods
- Interpretation
  - high FANOUT hampers reuse in another application
  - high FANOUT, a higher sensitivity to changes
  - high FANOUT, a rigorous testing



## Tight Class Cohesion

- Definition
  TCC = the relative number of method-pairs that access an attribute of the class
- Interpretation
  - the higher TCC, the tighter the semantic relation between the methods
  - the lower TCC, the higher the probability that a a class implements more than one functionality



Understand Code

## Overview of an OO System: Easier Said Than Done :-)

Let's play a game...

| Metric | Value |
|---|---|
| LOC | 35.000 |
| NOM | 3.600 |
| NOC | 380 |

- Want brief overview of the code of an OO system never seen before
- Want to find out how hard it will be to **understand** the code



## Now, Do We REALLY Know Something? :-)

Several questions remain unanswered...

- Is it "normal" to have...
  - ...380 classes in a system with 3.600 methods?
  - ...3.600 methods in a system with 35.000 lines of code?

➔ What means NORMAL?
  ➔ i.e. how do we compare with other projects?

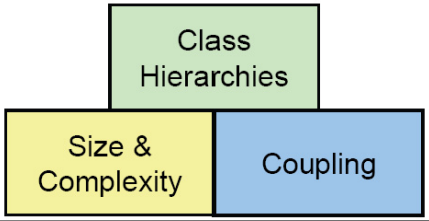
- What about the hierarchies ? What about coupling?

1. We need means of comparison. Thus, proportions are important!
2. Collect further relevant numbers; especially coupling and use of inheritance

## The Overview Pyramid [Lanza,Marinescu 2006]

- A metrics-based means to both describe and characterize the structure of an object-oriented system by quantifying its:
  - **complexity**,
  - **coupling** and
  - **usage of inheritance**

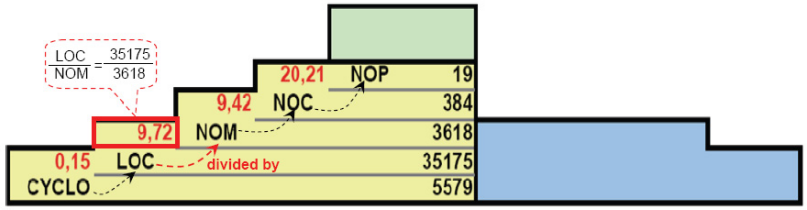
- Measuring these 3 aspects at system level provides a comprehensive ***characterization*** of an entire system








## Overview Pyramid: Size and Complexity

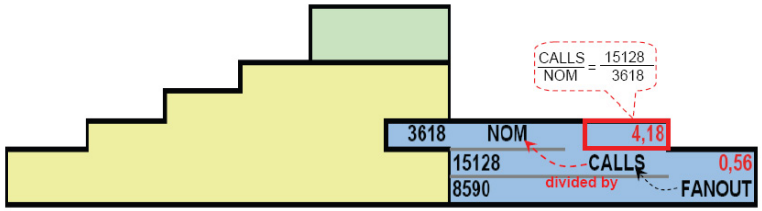



NOP = No. Of Packages
NOC = No. Of Classes
NOM = No. Of Methods
LOC = Lines Of Code
CYCLO = Cyclomatic Number (summed up over all methods)





## Overview Pyramid: Coupling



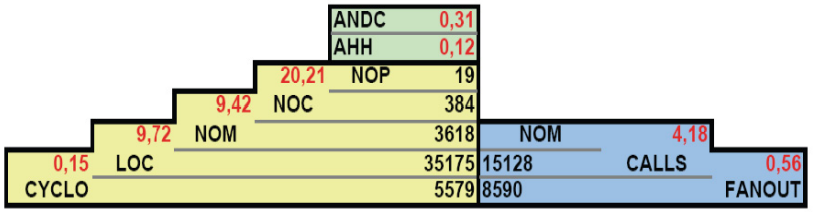

NOM = No. Of Methods
CALLS = No. Of Calls
FANOUT = No. Of Called Classes

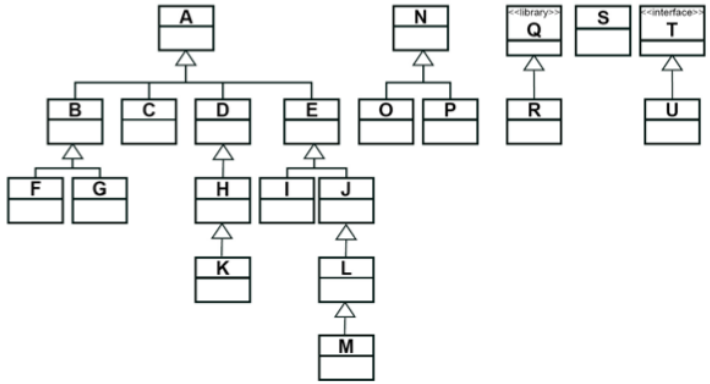




## Overview Pyramid: Inheritance

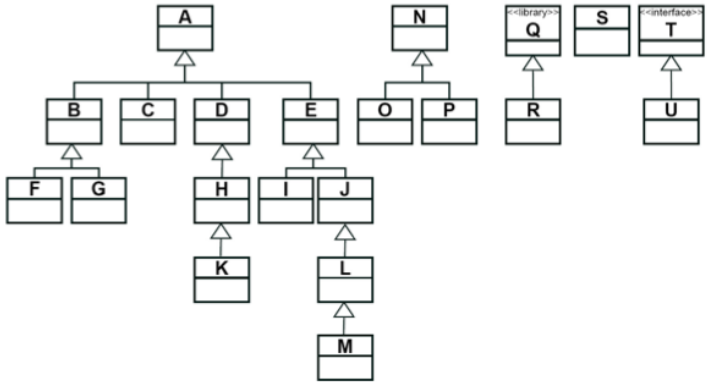
ANDC = Average Number of Derived Classes
AHH = Average Hierarchy Height






$$ANDC = \frac{11 \cdot 0 + 4 \cdot 1 + 3 \cdot 2 + 1 \cdot 4}{19} = 0.73$$



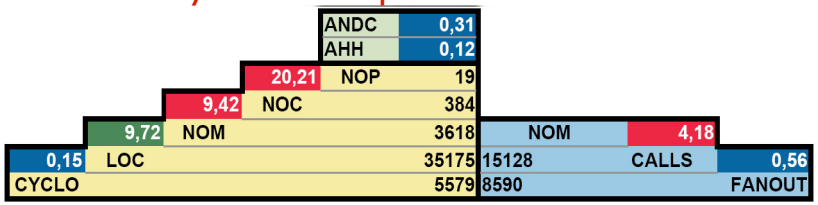



$$AHH = \frac{4 + 1 + 0 + 0 + 0}{5} = 1$$



Understand

## Overview Pyramid: Interpretation




- Interpretation based on a statistically relevant collection of data
  - collected for Java and C++
  - over 80 systems

close to AVERAGE

close to HIGH

close to LOW

| | Java | | | C++ | | |
|---|---|---|---|---|---|---|
| Metric | Low | Average | High | Low | Average | High |
| CYCLO/Line of code | 0.16 | 0.20 | 0.24 | 0.20 | 0.25 | 0.30 |
| LOC/Operation | 7 | 10 | 13 | 5 | 10 | 16 |
| NOM/Class | 4 | 7 | 10 | 4 | 9 | 15 |
| NOC /Package | 6 | 17 | 26 | 3 | 19 | 35 |
| CALLS/Operation | 2.01 | 2.62 | 3.2 | 1.17 | 1.58 | 2 |
| FANOUT /Call | 0.56 | 0.62 | 0.68 | 0.20 | 0.34 | 0.48 |
| ANDC | 0.25 | 0.41 | 0.57 | 0.19 | 0.28 | 0.37 |
| AHH | 0.09 | 0.21 | 0.32 | 0.05 | 0.13 | 0.21 |

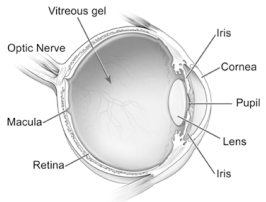


We are

# visual

beings.

70% of all external inputs come through the eyes

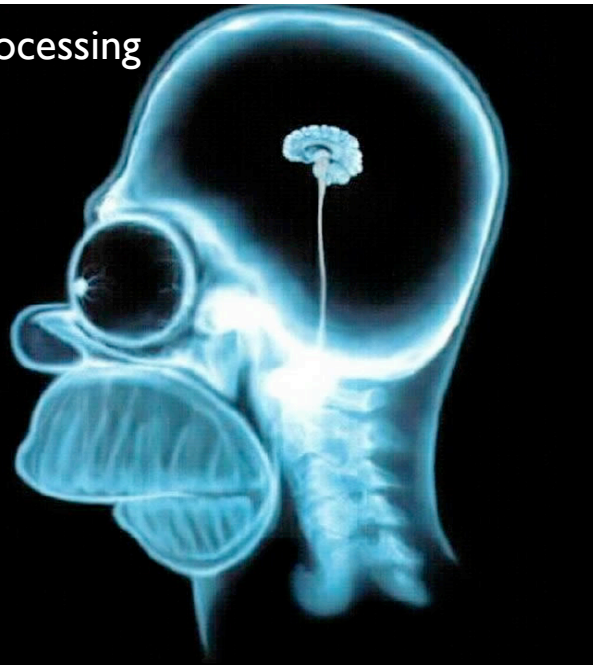



## Preattentive Processing

Iconic

Short-term

Long-term



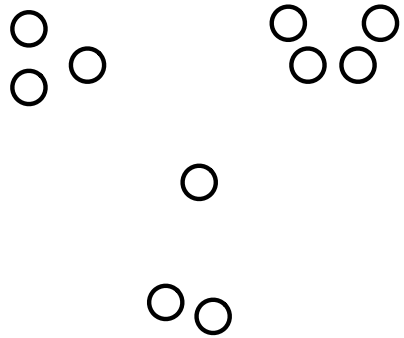
## Which numbers are larger than 0.85?

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0.103 | 0.176 | 0.387 | 0.300 | 0.829 | 0.276 | 0.179 | 0.321 | 0.192 | 0.250 |
| 0.333 | 0.384 | 0.864 | 0.587 | 0.857 | 0.698 | 0.640 | 0.621 | 0.984 | 0.316 |
| 0.421 | 0.309 | 0.654 | 0.729 | 0.228 | 0.529 | 0.832 | 0.435 | 0.699 | 0.426 |
| 1.266 | 0.750 | 0.056 | 0.936 | 0.711 | 0.749 | 0.723 | 0.201 | 0.542 | 0.819 |
| 0.225 | 0.926 | 0.643 | 0.337 | 0.721 | 0.337 | 0.682 | 0.987 | 0.232 | 0.449 |
| 0.187 | 0.586 | 0.529 | 0.340 | 0.276 | 0.835 | 0.473 | 0.445 | 1.103 | 0.720 |
| 1.153 | 0.485 | 0.560 | 0.428 | 0.628 | 0.335 | 0.456 | 0.879 | 0.699 | 0.424 |

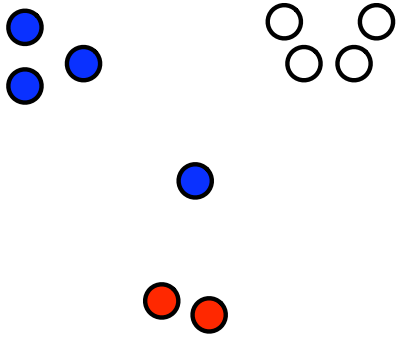
## (11) Pre-attentive features

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0.103 | 0.176 | 0.387 | 0.300 | 0.829 | 0.276 | 0.179 | 0.321 | 0.192 | 0.250 |
| 0.333 | 0.384 | **0.86** | 0.587 | **0.86** | 0.698 | 0.640 | 0.621 | **0.98** | 0.316 |
| 0.421 | 0.309 | 0.654 | 0.729 | 0.228 | 0.529 | 0.832 | 0.435 | 0.699 | 0.426 |
| **1.27** | 0.750 | 0.056 | **0.94** | 0.711 | 0.749 | 0.723 | 0.201 | 0.542 | 0.819 |
| 0.225 | **0.93** | 0.643 | 0.337 | 0.721 | 0.337 | 0.682 | **0.99** | 0.232 | 0.449 |
| 0.187 | 0.586 | 0.529 | 0.340 | 0.276 | 0.835 | 0.473 | 0.445 | **1.1** | 0.720 |
| **1.15** | 0.485 | 0.560 | 0.428 | 0.628 | 0.335 | 0.456 | **0.88** | 0.699 | 0.424 |

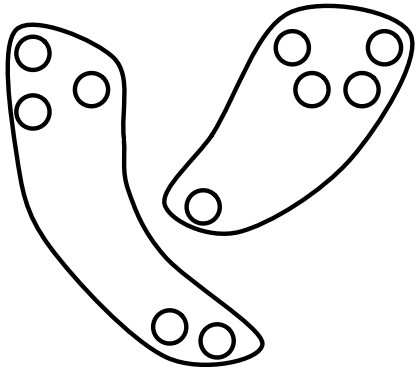
## How many groups do you see?



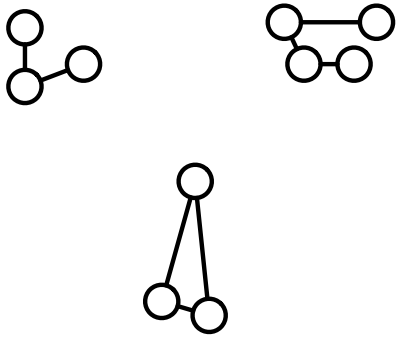
How many groups do you see?



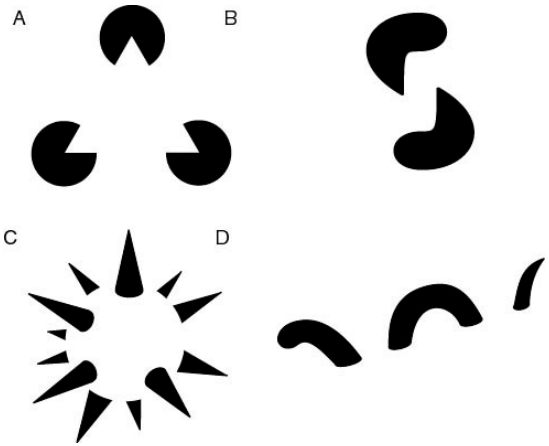
How many groups do you see?



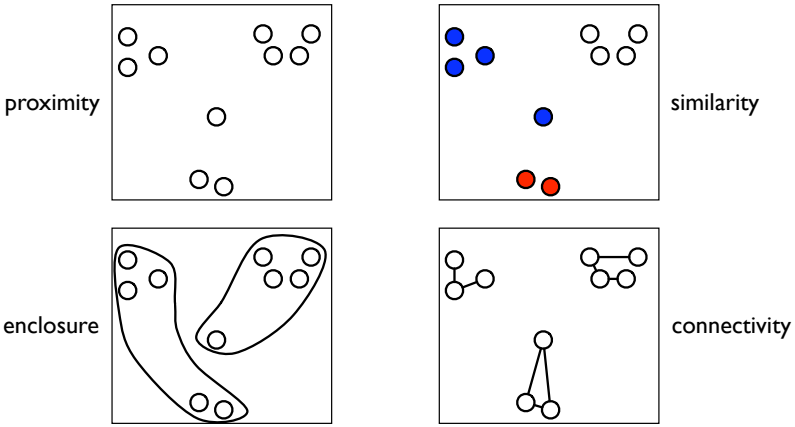
How many groups do you see?



(III) Gestalt principles



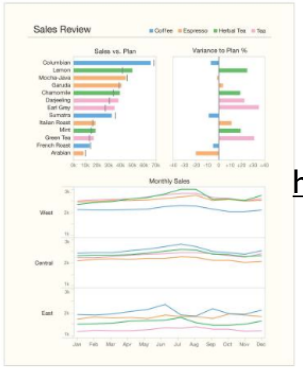
# (III) Gestalt principles




Second Edition

## Show Me the Numbers

Designing Tables and Graphs to Enlighten



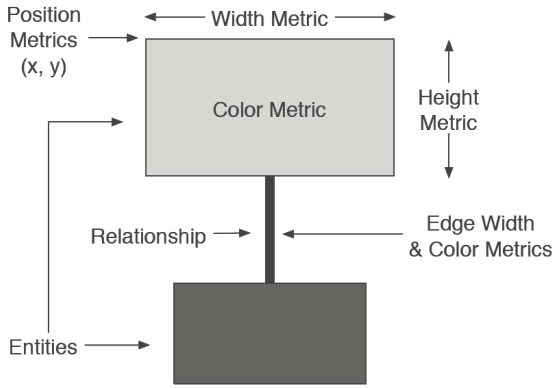
Graph Design IQ Test

http://www.perceptualedge.com/files/GraphDesignIQ.html

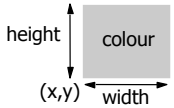
Stephen Few

## Visualizing the Whole System

**Polymetric Views**
**[Lanza,Ducasse 2003]**



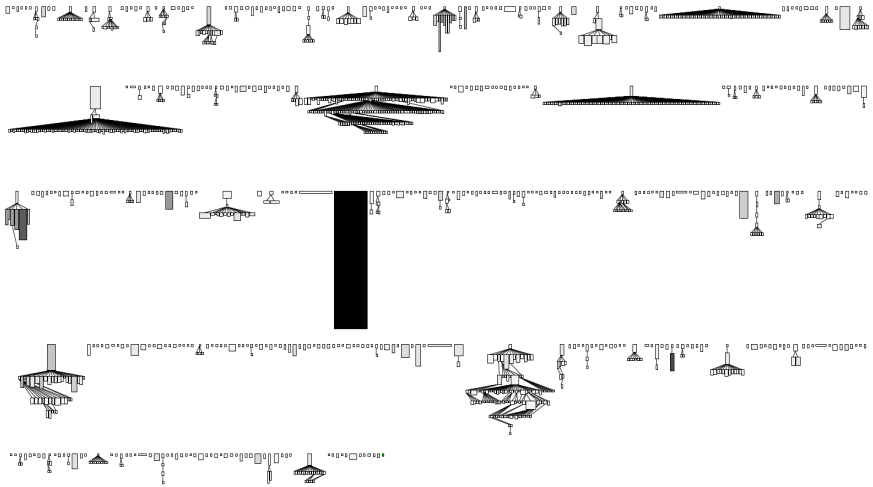

Use simple metrics and layout algorithms.




Visualize up to 5 metrics per node



## A Picture is Worth a Thousand Words...

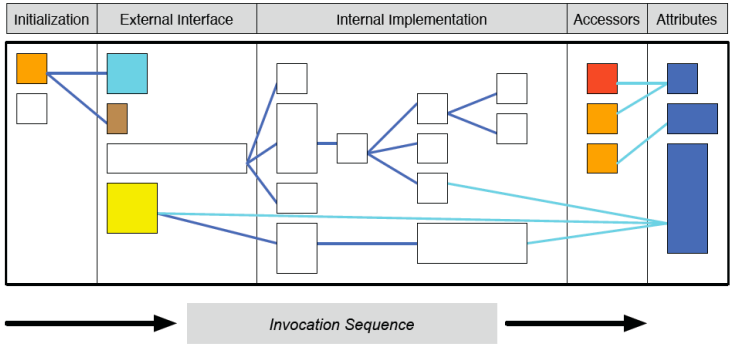


**System Complexity View of ArgoUML**


Understand

## Quickly "Reading" Classes [Lanza,Ducasse 2001]

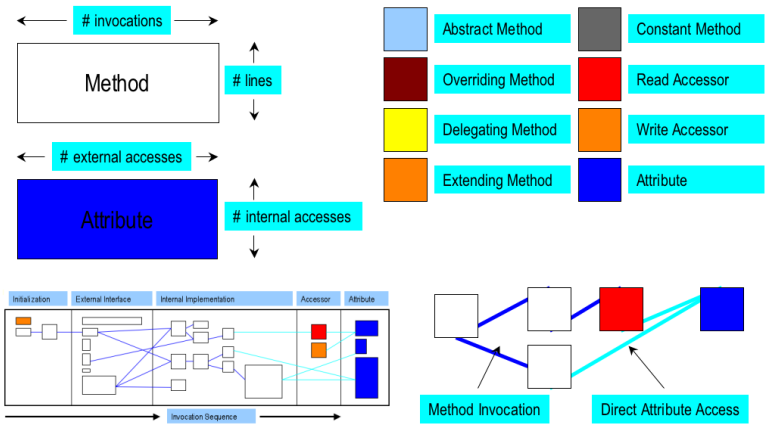
- Visualization Technique
  - serves as code inspection technique
  - reduces the amount of code that must be read






Understand

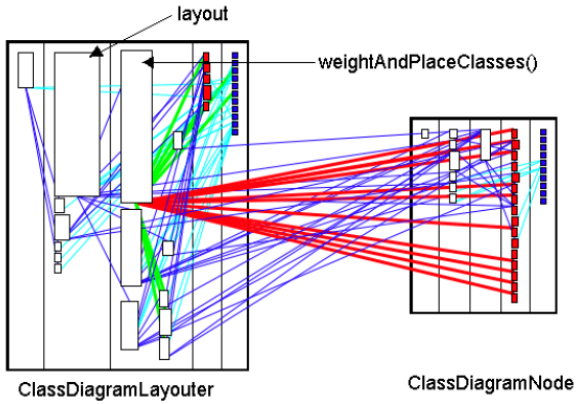
## Class Blueprint



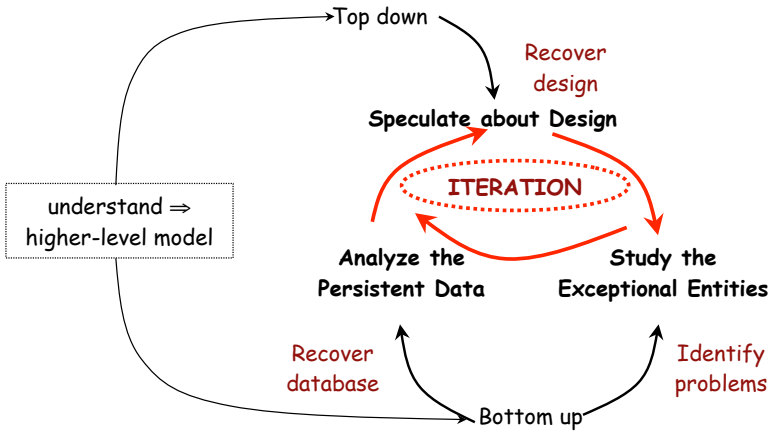



Understand

## Class Blueprint Example



## Initial Understanding (revisited)






## Detailed Model Capture

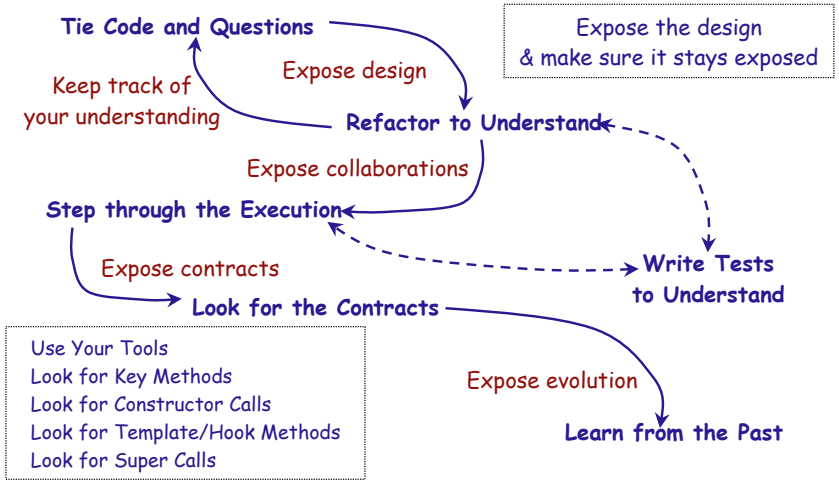


## Detailed Model Capture

- 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 behavior to extract detailed design



## Detailed Model Capture



## Learn from the Past

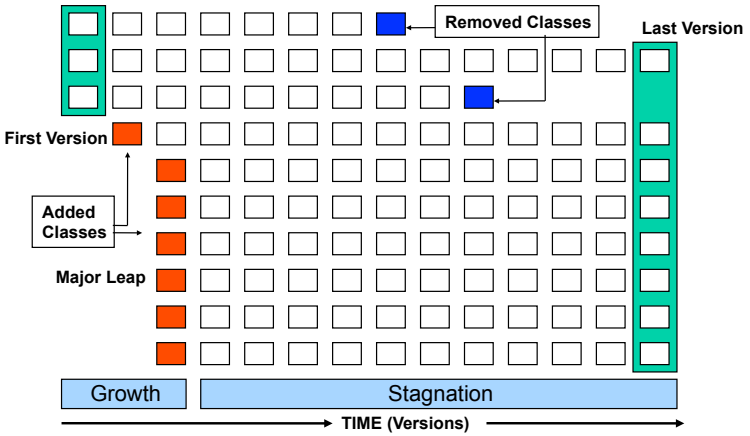
Problem: How did the system get the way it is?

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



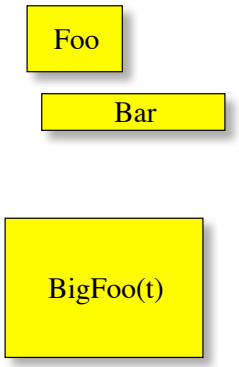
## CodEVolver: The Evolution Matrix [Lanza]





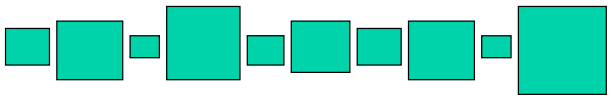

## Visualizing Classes in Evolution Using Metrics

- Object-Oriented Programming is about "state" and "behavior":
  - State is encoded using attributes
  - Behavior is encoded with methods
- We visualize classes as rectangles using for width and height the following metrics:
  - NOM (number of methods)
  - NOA (number of attributes)

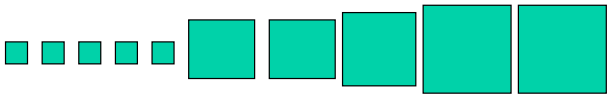





## Pulsar & Supernova

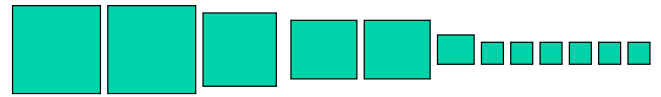


**Pulsar:** Repeated Modifications make it grow and shrink.
System Hotspot: Every System Version requires changes.

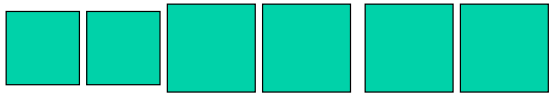


**Supernova:** Sudden increase in size. Possible Reasons:
- Massive shift of functionality towards a class.
- Data holder class for which it is easy to grow.
- *Sleeper*: Developers knew exactly what to fill in.

## White Dwarf, Red Giant, Idle



**White Dwarf:** Lost the functionality it had and now trundles along without real meaning. Possibly dead code.

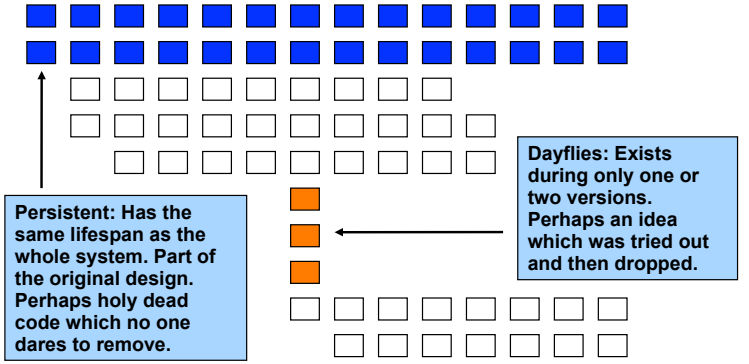


**Red Giant:** A permanent god class which is always very large.

**Idle:** Keeps size over several versions. Possibly dead code, possibly good code.



## Dayfly & Persistent






## 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!
- 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
- ...and tool support
  - automatic refactorings
- Hints:
  - Rename attributes to convey **roles**
  - Rename methods and classes to reveal **intent**
  - Remove **duplicated code**
  - Replace condition branches by methods
  - Define method bodies with same level of abstraction
- Needs tool support!

## Look for the Contracts

Problem: Which contracts does a class support?

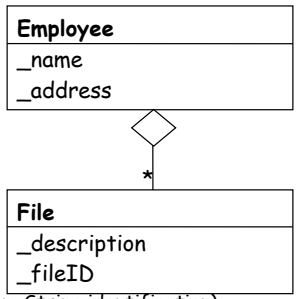
Solution: Look for common programming idioms, i.e. look for "customs" of using the interface of that class

- 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 = "";
```



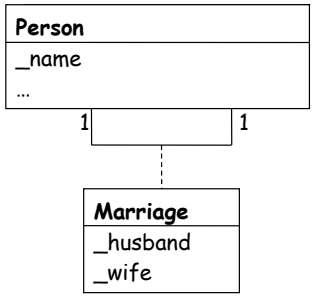

```
public void createFile (int position, String description, String identification)
{
      files [position] = new File (description, identification);
}
```

- Identify part-whole relationships (refining associations)
  - storing result of constructor in attribute ⇒ part-whole relation



## 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;}
```



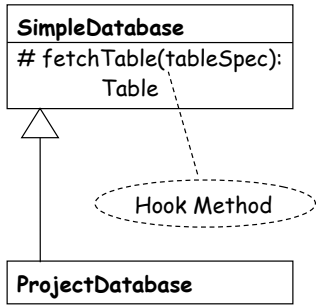

```
in class Person:
Marriage marryWife (Person wife) {
  return new Marriage (this, wife);
}
```

... acts as **PART**



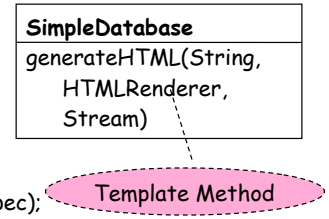
## Hook Methods

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

public class ProjectDatabase
          extends SimpleDataBase {
   ...
   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 SimpleDatabase {
    ...
    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
...}
```






## 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