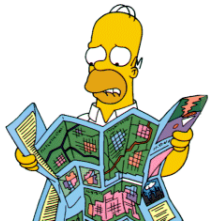
3. Software Visualization

- ***Introduction***
  - + SV in a Reengineering Context
- Static Code Visualization
  - + Examples
- Dynamic Code Visualization
  - + Examples
- Lightweight Approaches
  - + Combining Metrics and SV
- Understanding Evolution
- Conclusion



Program Visualization

- *Reduction* of complexity
- Generate *different views* on software system
- Visualization is powerful. But
  - + Can be *complex* (active research area),
    - Efficient space use, crossing edges, focus...
  - + Colors are nice but there is *no convention*
  - + *Nice* pictures do not imply *valuable* information
  - + Where to look? What is important?

A Bit of Vocabulary

- Visualization
  - + Information Visualization
- Software Visualization
  - + Algorithm Visualization
  - + Program Visualization
    - Static Code Visualization
    - Dynamic Code Visualization
- The overall goal is to reduce complexity

(Information) Visualization

- Bertin assessed three levels of questions
  - + Lower perception (one element)
  - + Medium perception (several elements)
  - + Upper perception (all elements/the complete picture)
- In Information Visualization it's all about the reduction of complexity
- Information Collection
- What to visualize
- How to visualize

Software Visualization

"Software Visualization is the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software."

Price, Baecker and Small, "Introduction to Software Visualization"

2 main fields:

- + Algorithm Visualization
- + Program Visualization

The main conceptual problem:

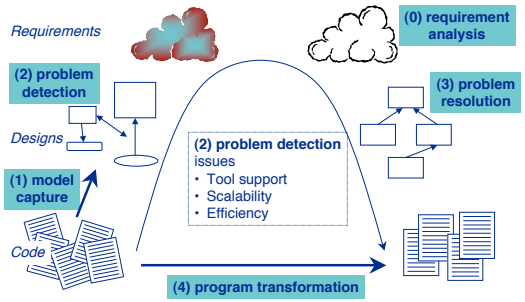
"Software is intangible, having no physical shape or size. Software visualisation tools use graphical techniques to make software visible by displaying programs, program artifacts and program behaviour."

[Thomas Ball]

In a Reengineering Context

- Work on old systems, dialects
- New tools are *not* processing *your* (C++) *dialect*
- Approaches
  - +Scalability is crucial
  - +Efficient (time/information obtained)
  - +Need a clear focus
- Solutions
  - +Minimize tools support
  - +Use existing proven tools (Rigi, CodeCrawler, Jinsight)
  - +Do it yourself but simple thing first

The Reengineering Life-cycle



Program Visualization

"Program visualization is the visualization of the actual program code or data structures in either static or dynamic form"

[Price, Baecker and Small]

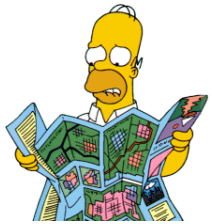
- Static code visualization
- Dynamic code visualization
- Generate different views of a system and infer knowledge based on the views
- Complex problem domain (current research area)
  - + Efficient space use, edge crossing problem, layout problem, focus, HCI issues, GUI issues, …
  - + Lack of conventions (colors, symbols, interpretation, …)

Program Visualization II

- Level of granularity?
  - + Complete systems, subsystems, modules, classes, hierarchies,...
- When to apply?
  - + First contact with a unknown system
  - + Known/unknown parts?
  - + Forward engineering?
- Methodology?

RoadMap

- Introduction
  - + SV in a Reengineering Context
- ***Static Code Visualization***
  - + Examples
- Dynamic Code Visualization
  - + Examples
- Lightweight Approaches
  - + Combining Metrics and SV
- Understanding Internals
  - + Call flow within classes
- Understanding Evolution
- Conclusion

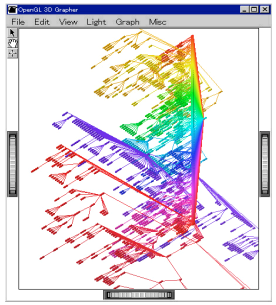


Static Code Visualization

- The visualization of information that can be extracted from the static structure of a software system
- Depends on the programming language and paradigm:
  - + Object-Oriented PL:
    - classes, methods, attributes, inheritance, …
  - + Procedural PL:
    - procedures, invocations, …
  - + …

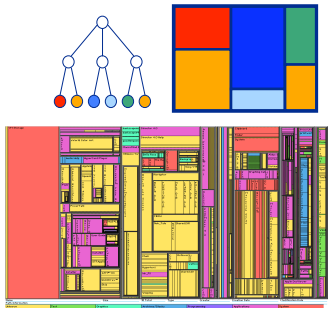
Example 1: Class Hierarchies

- Jun/OpenGL
- The Smalltalk Class Hierarchy
- Problems:
  - + *Colors are meaningless*
  - + *Visual Overload*



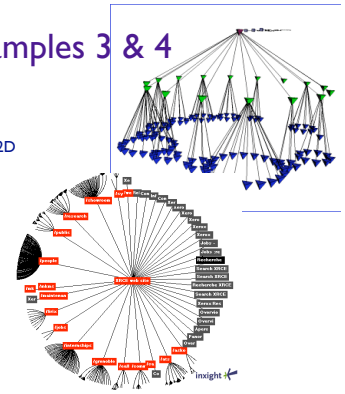
Example 2: Tree Maps

- Pros
  - + 100% screen
  - + Large data
  - + Scales well
- Cons
  - + Boundaries
  - + Cluttered display
  - + Interpretation
  - + Leaves only
- Useful for the display of HDDs



Examples 3 & 4

- Euclidean cones
  - + Pros:
    - More info than 2D
  - + Cons:
    - Lack of depth
    - Navigation
- Hyperbolic trees
  - + Pros:
    - Good focus
    - Dynamic
  - + Cons:
    - Copyright



Kind of Code Maps

- From Marcus,Feng, Maletic Software Visualization'03
- Simple
- Overview
- File-based
- One "Dot" = one line

Nesting Level

Figure 2. A 2D overview of a system containing 30 C++ source code files (approx. 4000 LOC). Each file is mapped to a container and the name of the file is shown on top of the container. Color is used to show nesting level of the line of source code.

Control Flow

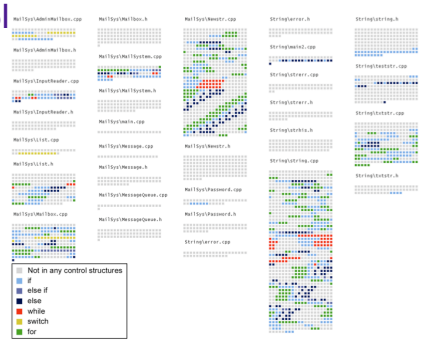


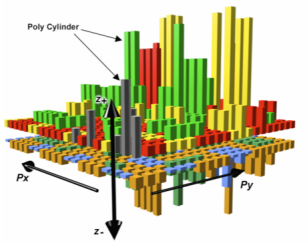
Figure 6. A 2D overview of a system containing 30 C++ source code files (approx. 4000 LOC). Each file is mapped to a container and the name of the file is shown on top of the container. Color is used to show control structures.

Evaluation

- Simple to draw
- Good overview
- Limited semantics
- Patterns difficult to identify because of line breaks

Two Cases for 3D

- Most of the time 3D is not worth but…

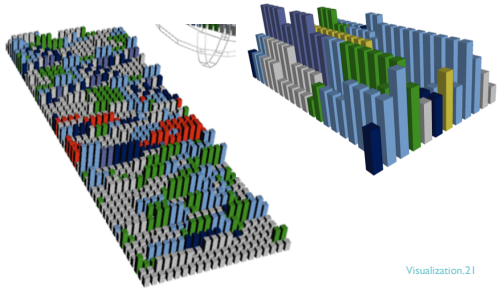


Usual Problems with 3D

- No spatial semantics (is above better than below)
- Scalability
- Extra effort
- Space localization

3D…

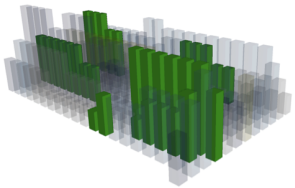
- 3D useful for quantitative information



Enabling 3D

3D Problems

- Problem: accessing hidden information



Evaluation

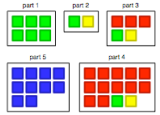
- Worth to represent quantitative information
- Spatial information is not really sexy
- Requires more work
- Requires more tooling

Class Diagram Approaches

- For example UML diagrams…
- Pros:
  - + OO Concepts
  - + Good for small parts
- Cons:
  - + Lack of scalability
  - + Require tool support
  - + Requires mapping rules to reduce noise
  - + Preconceived views

Distribution Map

How a property spread or focus on a system?



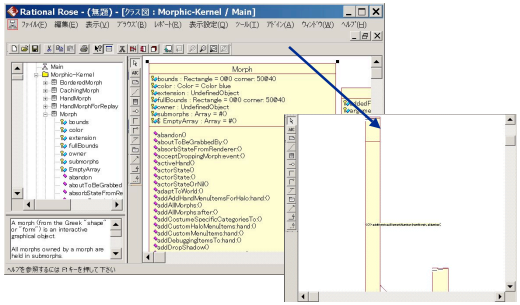
JBoss Files Owner

Jedit Symbolic Distribution

Evaluation

- Simple
- Scalable
- Work only if property hard cut the space

Class Diagram Examples

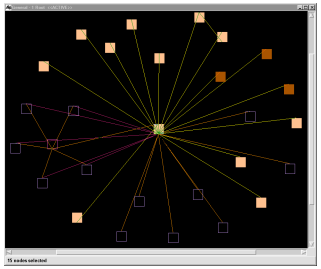


Example 5a: Rigi

- Scalability problem
- Entity-Relationship visualization
- Problems:
  - + Filtering
  - + Navigation

Example 5b: Rigi

- Entities can be grouped
- Pros:
  - + Scales well
  - + Applicable in other domains
- Cons:
  - + Not enough code semantics



Evaluation

- Pros
  - + Intuitive approaches
  - + Aesthetically pleasing results
- Cons
  - + Several approaches are orthogonal to each other
  - + Too easy to produce meaningless results
  - + Scaling up is sometimes possible, however at the expense of semantics

RoadMap

- Introduction
  - + SV in a Reengineering Context
- Static Code Visualization
  - + Examples
- ***Dynamic Code Visualization***
  - + Examples
- Lightweight Approaches
  - + Combining Metrics and SV
- Evolution
- Conclusion

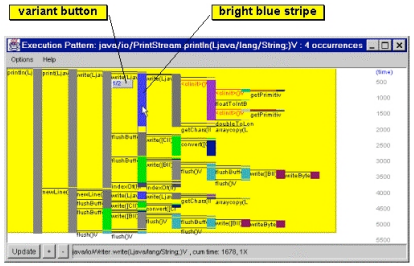
Dynamic Code Visualization

Visualization of dynamic behaviour of a software system

- + Code instrumentation
- + Trace collection
- + Trace evaluation
- + What to visualize
  - Execution trace
  - Memory consumption
  - Object interaction
  - …

Example 1: JInsight

- Visualization of execution trace



Example 2: Inter-class call matrix

- Simple
- Reproducible
- Scales well
- Excel?

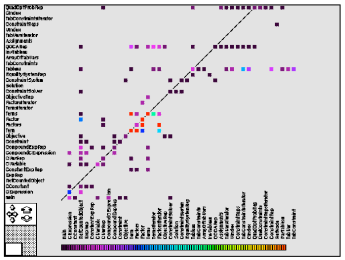
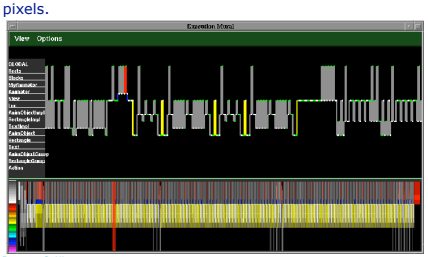


Figure 6: Inter-class call matrix

Mural View

- The algorithm takes an image of *M x N* elements and scales it into a mural of *I x J* pixels.



The Algorithm

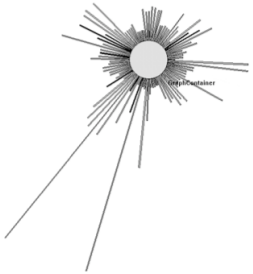
```
1) for each i,j set mural_array[i][j] to zero
2) for each element m,n of information
     a) compute x = m / M * I, y = n / N * J
     b) determine the proportion of this point that lies in each of the four
surrounding mural_array entries (totals to 1.0):
        mural_array[floor(x)][floor(y)]
        mural_array[floor(x)][ceil(y)]
        mural_array[ceil(x)][floor(y)]
        mural_array[ceil(x)][ceil(y)]
   c) add each of the proportions determined in the previous step to the
existing values of each corresponding mural_array entry
        i) update max_mural_array_value to keep track of the maximum
mural_array[][] value
3) for each i,j in the mural_array
     a) map the value mural_array[i][j] / max_mural_array_value to a grayscale
or color intensity varying scale, or to pixel size, depending on the type of
mural being created
     b) color and draw the pixel at i,j of the mural based on mapping computed
in the previous step
```


Class View

- Smith, Munro, Runtime Visualization of Object Oriented Software, Vissoft 02

A Class

- Methods/#invocation



Method Calling Counts

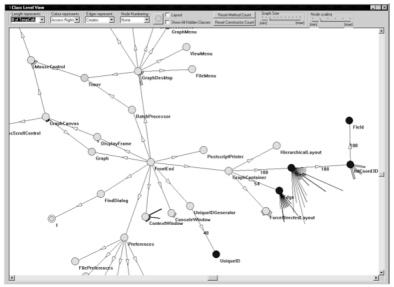


Figure 6 Class View showing method calling counts

Evaluation

- Entities as objects
- Spot fast the important methods
- For complete scenario may be difficult to reproduce
- Requires interactivity
- Layout can be a problem

Evaluation

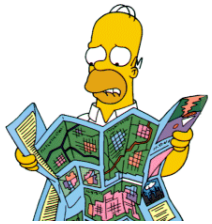
- Useful not for any kinds of data
- Handling of large amount of data

Dynamic SV: Evaluation

- Code instrumentation problem
  - + Logging, Extended VMs, Method Wrapping, C++ preprocessing is heavy
- Scalability problem
  - + Traces quickly become very big (1Mb/s)
- Completeness problem: scenario driven
- Pros:
  - + Good for fine-tuning, problem detection
- Cons:
  - + Tool support crucial
  - + Lack of abstraction without tool support

RoadMap

- Introduction
  - + SV in a Reengineering Context
- Static Code Visualization
  - + Examples
- Dynamic Code Visualization
  - + Examples
- ***Lightweight Approaches***
  - + Combining Metrics and SV
- Understanding Internals
  - + Call flow within classes
- Understanding Evolution
- Conclusion

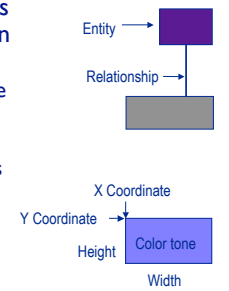


Do It Yourself Considerations

- A decent graph layout can be a hard task...
  - + Algorithmic aspects may be important
  - + Efficient space use (physical limits of a screen)
  - + Colours are nice, but... there are no conventions!
  - + Trade-off between usefulness and complexity
- Keeping a focus is hard:
  - + Beautiful graphs are not always meaningful
  - + Where should we look?
  - + What should we look for?
- Which approach be reproduced by reengineers in work context and provides useful information?

Solution: A lightweight approach

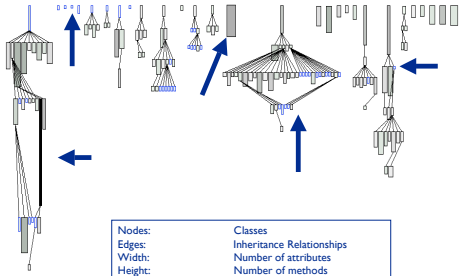
- A combination of metrics and software visualization
  - + Visualize software using colored rectangles for the entities and edges for the relationships
  - + Render up to five metrics on one node:
    - Size (1+2)
    - Color (3)
    - Position (4+5)



Combining Metrics and Visualization

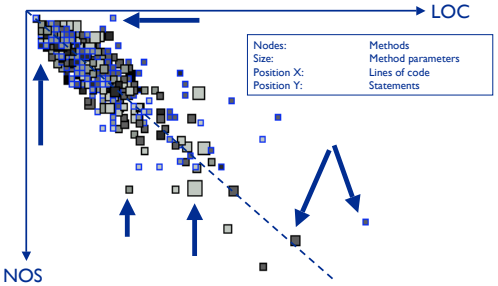
- Metrics
  - + Scale well
  - + Simple metrics ⇒ simple extraction (perl scripts)
  - + But numerical combination is meaningless
- Simple Graphs
  - + Easy to draw
  - + Scale well
  - + But not enough semantics
- CodeCrawler: www.iam.unibe.ch/~scg

System Complexity View

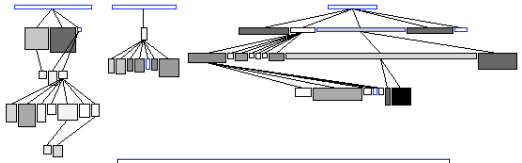


Nodes:	Classes
Edges:	Inheritance Relationships
Width:	Number of attributes
Height:	Number of methods
Color:	Number of lines of code

Method Assessment

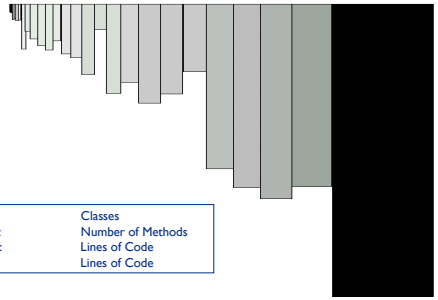


Inheritance Classification View



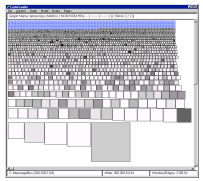
Boxes:	Classes
Edges:	Inheritance
Width:	Number of Methods Added
Height:	Number of Methods Overridden
Color:	Number of Method Extended

Data Storage Class Detection View



Boxes:	Classes
Width:	Number of Methods
Height:	Lines of Code
Color:	Lines of Code

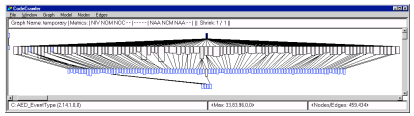
Industrial Validation



Personal experience
2-3 days to get something

Nokia (C++ 1.2 MLOC >2300 classes)
Nokia (C++/Java 120 kLOC >400 classes)
MGeniX (Smalltalk 600 kLOC >2100classes)
Bedag (COBOL 40 kLOC)
...

Used by developers + Consultants



Evaluation

- Pros
  - + Quick insights
  - + Scalable
  - + Metrics add semantics
  - + Interactivity makes the code "come nearer"
  - + Reproducible
  - + Industrial Validation is the acid test
- Cons
  - + Simple
  - + Useful in a first approach phase
  - + Code reading needed
  - + Level of granularity

RoadMap

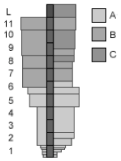
- Introduction
  - + SV in a Reengineering Context
- Static Code Visualization
  - + Examples
- Dynamic Code Visualization
  - + Examples
- Lightweight Approaches
  - + Combining Metrics and SV
- ***Understanding Evolution***
- Conclusion

Understanding Evolution

- Information is in the history!
- Overwhelming complexity
- How can we detect and understand changes?
- Solutions:
  - + Revision Towers
  - + TimeWheel, Infobug
  - + The Evolution Matrix

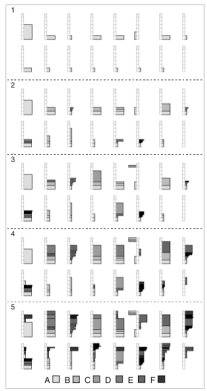
Revision Tower

- Taylor, Munro, Revision Towers, Vissoft02
- Past is at the bottom
- Middle section represents release
- Side section represents history
- Here .c files compared with .h files
- Authors are color typed
- File changed often: lot of rectangle inside a release period



Revision Tower (II)

- Simple
- Entire file
- File based
- Few information revealed

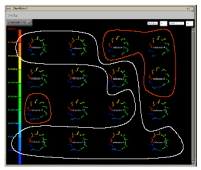


Definitions

- Glyph: A symbol, such as a stylized figure or arrow on a public sign, that imparts information nonverbally.

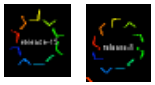
How can we represent time?

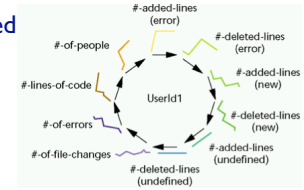
- Animation?
  - + Good for easily perceived outliers
- Time Series graph?
  - + Good for comparing trends
- Timewheel



TimeWheel (1)

- Displays trends for a number of attributes at a time
- Maintain "Objected through Gestalt principals





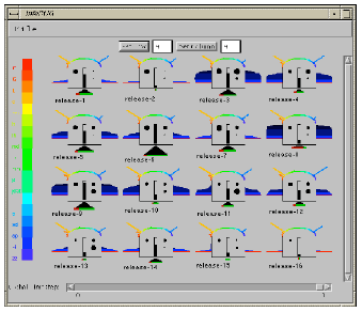
Timewheel (II)

- Multiple time series ordered in a circle
- Data attributes are color coded
- Easy recognition of two trends
  - + Increasing trend
  - + Tapering trend
- Helps to examine different trends within one object

Time Series Problems

- In row
  - + More time to spot them
  - + Less local patterns
- In circle
  - + Weakens reading order implications
  - + Rotation invariant
- Example

InfoBug

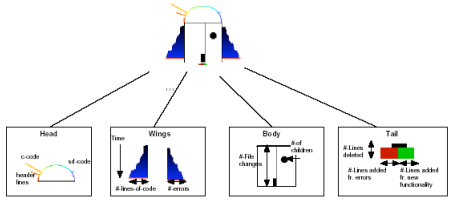


Infobug

- Look like an insect
- Show many properties while still maintaining "objectedness"
- Certain patterns preattentively pop out
- Interactive
- Represent four classes of software data
  - + Code lines, errors (wings)

4 Classes of Software Data

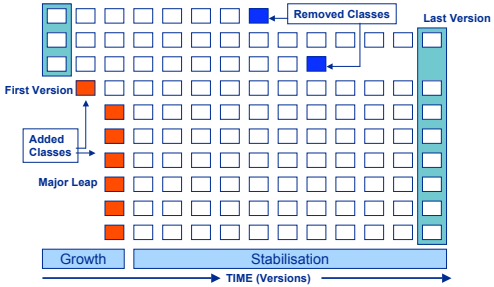
- Head (Type of code)
- Wings (Lines of codes, errors)
- Body (bar - file changes, Spots - number of subsystems)
- Tails (added, removed lines)



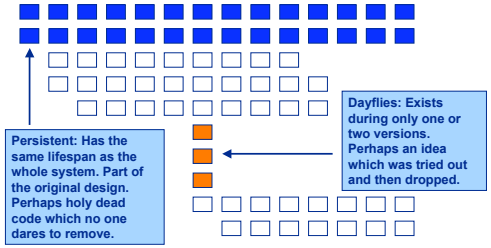
Evaluation

- Pros:
  - + Large datasets on little space
  - + Entities as objects
  - + Easy to recognise patterns
  - + Trends identification
  - + Easy to compare and analyse
  - + Interactive
- Cons:
  - + Learning (but is there something we should not learn?)
  - + Main focus on Error/Loc ratio
  - + Could include more information

The Evolution Matrix



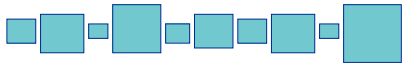
Dayfly & Persistent



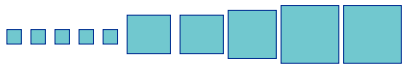
Visualizing Classes Using Metrics

- 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)
- The Classes can be categorized according to their "personal evolution" and to their "system evolution": Pulsar, Supernova,Red Giant, Stagnant,DayflyPersistent

Pulsar & Supernova

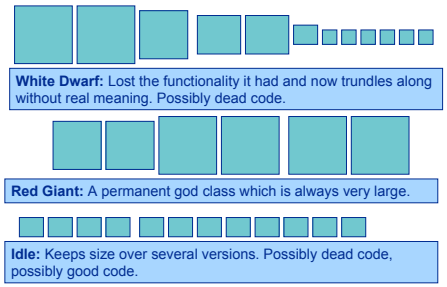


Pulsar: Repeated Modifications make it grow and shrink.
System Hotspot: Every System Version requires changes.



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

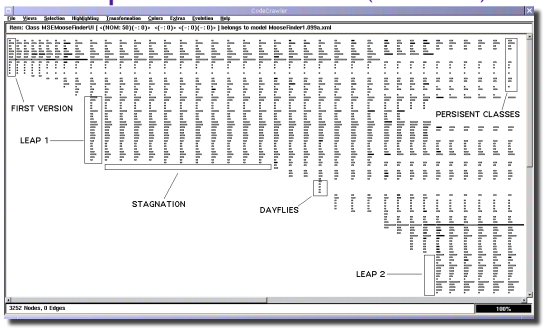


White Dwarf: Lost the functionality it had and now trundles along without real meaning. Possibly dead code.

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

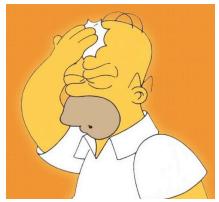
Idle: Keeps size over several versions. Possibly dead code, possibly good code.

Example: MooseFinder (38 Versions)



RoadMap

- Introduction
- SV in a Reengineering Context
- Static Code Visualization
  - + Examples
- Dynamic Code Visualization
  - + Examples
- Lightweight Approaches
  - + Combining Metrics and SV
- Understanding Evolution
- ***Conclusion***



Conclusions

- SV is very useful when used correctly
- An integrated approach is needed, just having nice pictures is not enough
- In general: only people that know what they see can react on that: SV is for expert/advanced developers
- The future of software development is coming…and SV is part of it

Lessons Learned

- Visualization is not just smoke and mirrors!
  - + Complexity reduction, abstraction
- *But* it should be adapted to
  - + your goal (first contact, deep understanding),
  - + time (2 days - a month),
  - + size (a complete system or 3 classes)
- Minimize tool support if you are not familiar
- Simple approaches give 80%, the last 20% are hard to get