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**

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.”

[Thomas Ball]

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**

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.”

[Thomas Ball]

2 main fields:
- Algorithm Visualization
- Program Visualization

The main conceptual problem:

“Software is intangible, having no physical shape or size. Software visualization 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

- **Requirement analysis**
- **Problem detection**
- **Problem resolution**
- **Program transformation**

Program Visualization

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

[Price, Baekker 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, …)

(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**
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

Example 2: Tree Maps

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

Example 1: Class Hierarchies

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

Evaluation

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

Kind of Code Maps

- From Marcus, Feng, Maletic Software Visualization’03
- Simple
- Overview
- File-based
- One “Dot” = one line

Example 3 & 4

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

Pros:

+ Cluttered display
+ Large data
+ Scales well

Cons:

+ Boundaries
+ Cluttered display
+ Interpretation
+ Leaves only

Useful for the display of HDDs
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

Evaluation

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

Enabling 3D

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
RoadMap

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

Evaluation

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

Jedit Symbolic Distribution

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

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

Class Diagram Examples

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
Example 2: Inter-class call matrix

- Simple
- Reproducible
- Scales well
- Excel?

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 of information
   a) compute m[i]/N = R, p = m[i]/R + 1
   b) compute elements that lie in each of the four surrounding mural_array elements equals to 1 if:
      mural_array[i][j] = (m[i]/N = R)
      mural_array[i][j] = (m[i]/N = R)
      mural_array[i][j] = (m[i]/N = R)
      mural_array[i][j] = (m[i]/N = R)
   c) add each of the properties determined in the previous step to the existing mural_array[i][j] value only if R > 0
   d) update the mural_array value to keep track of the maximum
      mural_array[i][j] value
3) for each i, j in the mural_array
   a) map the value mural_array[i][j] / max_mural_array into a grayscale on color intensity varying map, so to pixel size, depending on the type of mural_array created
   b) set a pixel color for 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 (~MBs)
- 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
* Lightweight Code Visualization
  + Examples
* Light Approaches
  + Combining Metrics and SV
  + Understanding Internals
  + Call flow within classes
  + Understanding Evolution
  + Conclusion

**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.iim.unibe.ch/~scg](http://www.iim.unibe.ch/~scg)

**System Complexity View**

- **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 (+2)
    - Color (3)
    - Position (+4+)

**Method Assessment**

- **LOC**
- **NOS**

**Inheritance Classification View**

**Data Storage Class Detection View**

**Industrial Validation**

- **Personal experience**
  - 2-3 days to get something
  - **Nokia** (C++ 1.2 MLOC >3200 classes)
  - **Nokia** (C++Java 1.2 MLOC >400 classes)
  - **MBeans** (Smalltalk 600 MLOC >2100 classes)
  - **React** (COCOL 40 KLOC)

- **Used by developers + Consultants**
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

How can we represent time?

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

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

TimeWheel (I)

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

Understanding Evolution

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

Definitions

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

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

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)

Dayfly & Persistent

- Exists during only one or two versions.
- Has the same lifespan as the whole system.
- Part of the original design.
- Perhaps holy dead code which no one dares to remove.

InfoBug

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

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

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 width and height the following metrics:
  + NGM (number of methods)
  + NFA (number of attributes)
- The Classes can be categorized according to their “personal evolution” and to their “system evolution”:
  + Pulsar, Supernova, Red Giant, Stagnant, Dayfly Persistent

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.

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

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