Find API Usage Patterns

Visualization Based API Usage Patterns Refining

Yubo Feng

Introduction

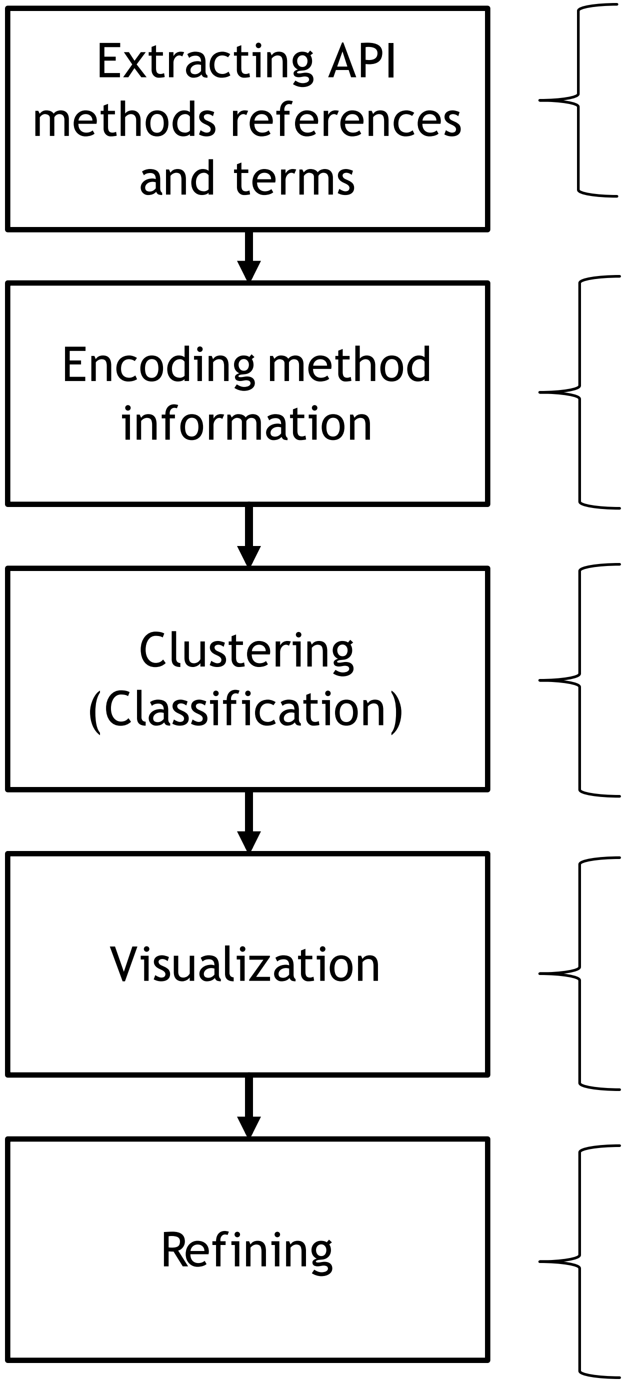
- What is API usage pattern?
  - Pattern: context, problem, solution
  - Given a lib, how to identify what problem it could solve?
  - Given a problem and a lib, how do we know if there exists some API in this lib that helps?
- Why it is necessary?
  - Read lib code and demo program is time consuming and tedious
  - Some amazing API in lib will never have chance to be used
  - dependency within single API is unknown
- Question: identify a groups of APIs in a given lib that solves bunch of problems

One possible solution and weakness

- Client program based analysis
- Basic assumption: API methods are interactive, interlocking
- Basic idea: a group of APIs will be used again and again to solve similar problems
- Analysis of the frequency and consistency of co-usage **relations** between the APIs methods within a variety of client programs of the API of interest
- Weakness: how about semantic relations?

What this paper tells us

- Capture contextual information is important
- A semi-automated approach to identify API pattern
- idea:
  - using client-program based approach and semantic analysis to find groups that may be consist some pattern
  - using some visualization methods to present these pattern and API method that is **easy to read** and interpret by human
  - using human knowledge and experience to refine results that already got



Extracting API methods references and terms

Client-Program analysis, Semantic analysis

Raw client program data analysis

Encoding method information

output two vectors: one for related methods; another for term. Prepare attributes/data set for classifier

Clustering (Classification)

Machine learning method involved, do clustering (classification)

Visualization

Paint everything we get in a “nice” form
easy for human to recognized

Refining

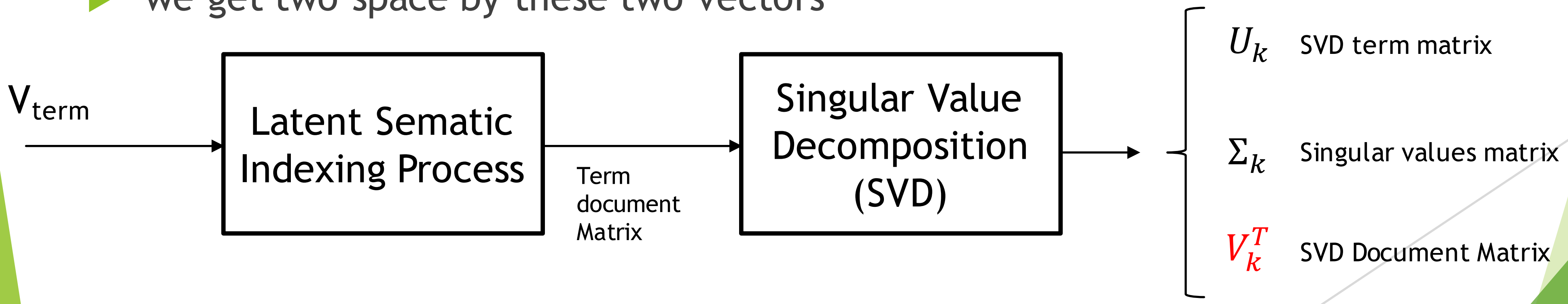
Human work involved: try to enrich pattern by other methods

Extracting API methods references and terms

- **Client-Program analysis:** statically analyze the code, extracting references within each other
- **Semantic analysis:** API method name, terms, parameters, local variables
- **Multi-level API Usage Pattern (MLUP)**
- **After the extraction, we got tables including references and term info inside**

Encoding method information

- **Assumption: for one particular domain purpose, the domain knowledge is encapsulated in the methods vocabulary**
- V_{usage} : vector for each API method, i_{th} element (0/1) indicates if this method is used in client program. (| V_{usage} | = # of client programs)
- V_{term} : vector for each API method, i_{th} element (0/1) indicates if this term is used in this API method vocabulary. (| V_{term} | = # of all lemmatized collected terms in public APIs)
- we get two space by these two vectors



V_{term}

Latent Sematic Indexing Process

Term document Matrix

Singular Value Decomposition (SVD)

U_k SVD term matrix

Σ_k Singular values matrix

V_k^T SVD Document Matrix

Cluster (classification)

- Distance metrics definition

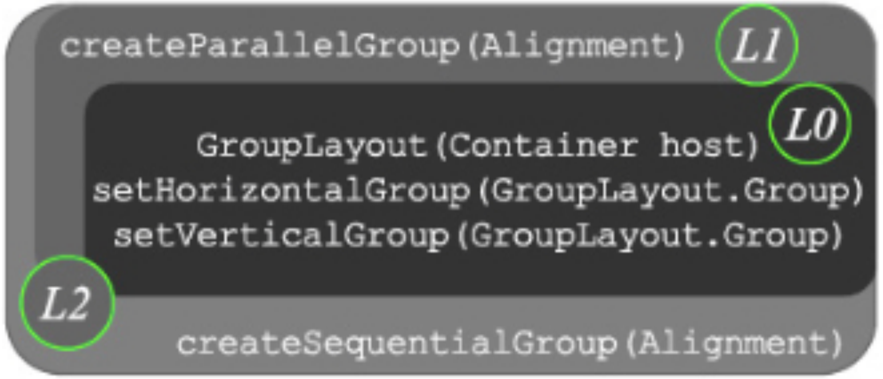
$$USim(m_i, m_j) = \frac{|Cl\_mtd(m_i) \cap Cl\_mtd(m_j)|}{|Cl\_mtd(m_i) \cup Cl\_mtd(m_j)|}$$

$$SemanticSim(m_i, m_j) = \frac{\vec{V_i} \times \vec{V_j}}{||\vec{V_i}|| \times ||\vec{V_j}||}$$

- DBSCAN cluster algorithm:
  - Two parameters control # of methods within one group
  - clusters according to V_{usage} we could get groups of APIs that close to each other (reference together always)
  - clusters according to V_k^T we could get groups of APIs that close to each other (semantic close to each other)
  - **by recursively apply this algorithm in one big group, we can get more smaller child groups**

Pattern Visualization

- From previous result, we get results actually is a hierarchic structure
  - big groups contains a lot of smaller group
- **Multi-level API Usage Pattern (MLUP)**



createParallelGroup(Alignment)

GroupLayout(Container host)
setHorizontalGroup(GroupLayout.Group)
setVerticalGroup(GroupLayout.Group)

createSequentialGroup(Alignment)

Figure 1. The cluster *L2* which represents the MLUP of class `GroupLayout`: *L0* represents the `GroupLayout`'s *core* usage pattern, then the cluster *L1*/*L2* includes *partially*/*totally* the `GroupLayout`'s *peripheral* usage pattern.

Pattern Visualization
(Naïve approach)

- tree-map visualization (according to V_{usage})

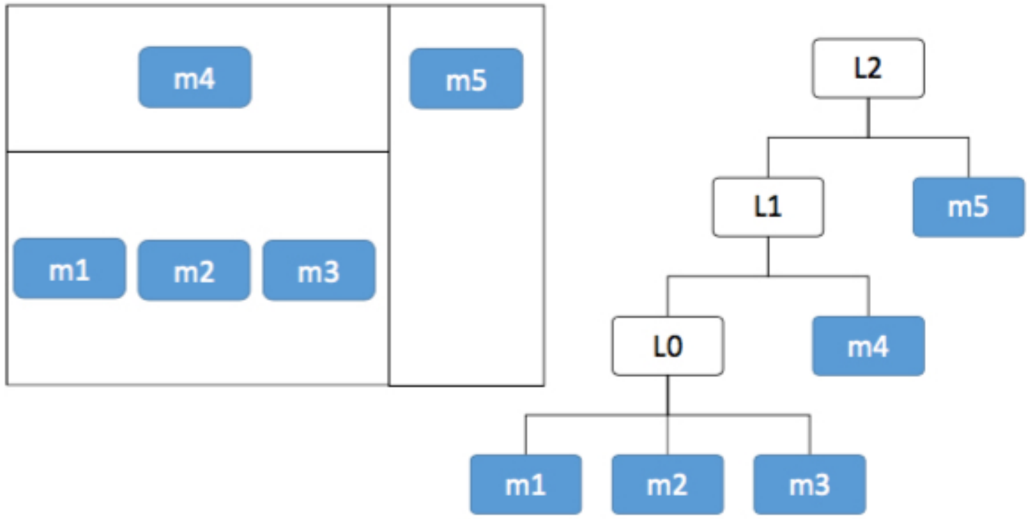


Figure 2. Treemap layout for GroupLayout pattern, m1 is the Grouplayout(Container) constructor, m2, m3, m4 and m5 are respectively the methods setHorizontalGroup(...) , setVerticalGroup (...) , createParallelGroup(...) and createSequentialGroup(...).

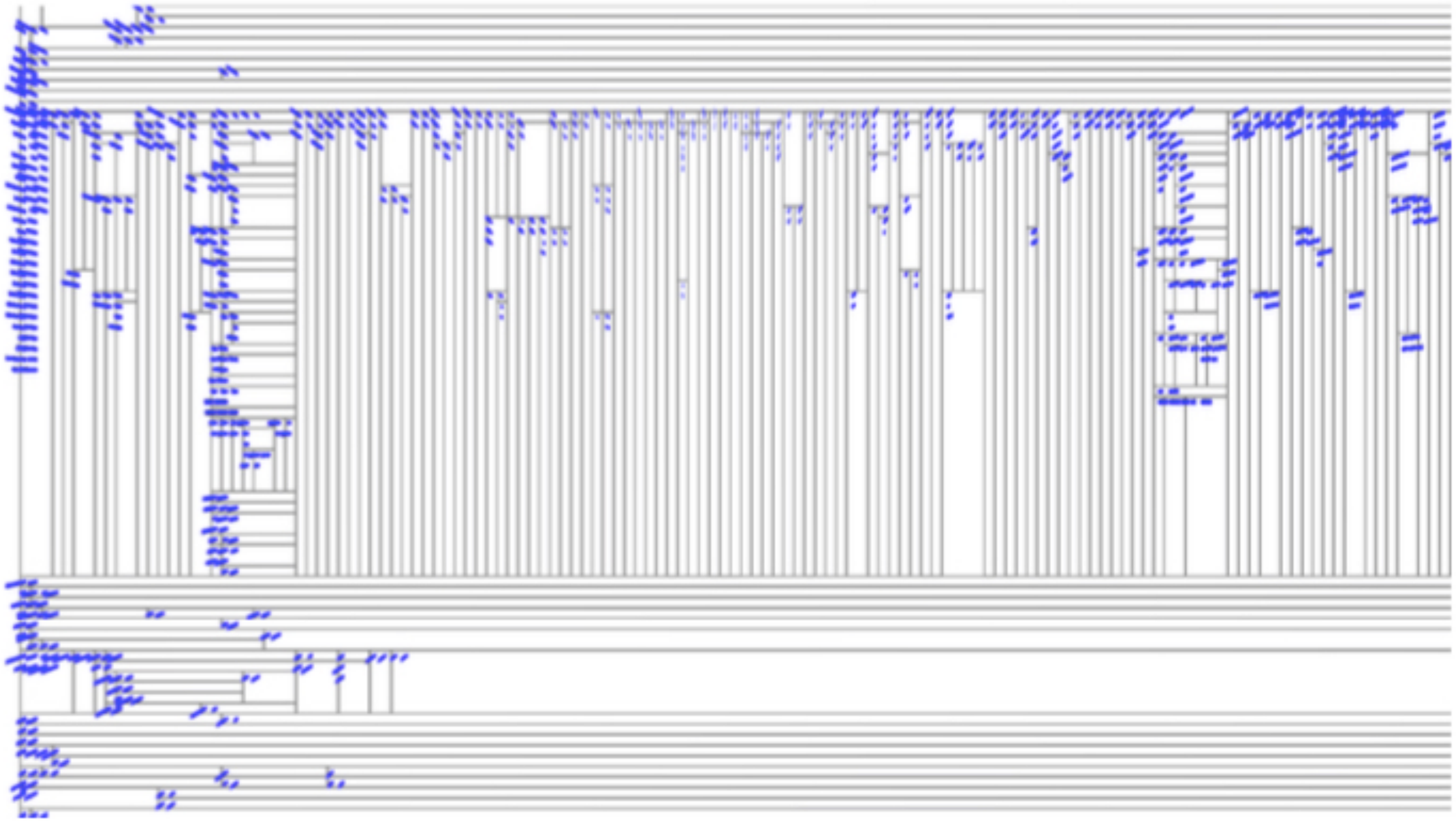


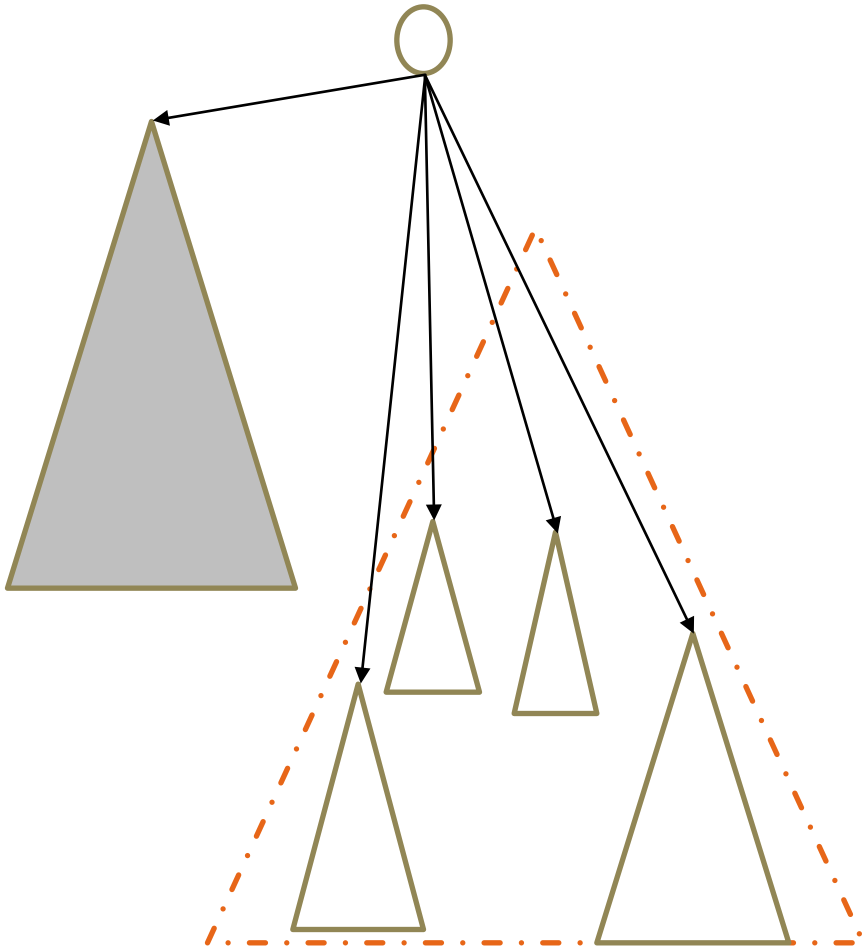
Figure 3. Standard treemap layout for the multi-level usage patterns of the Swing API.

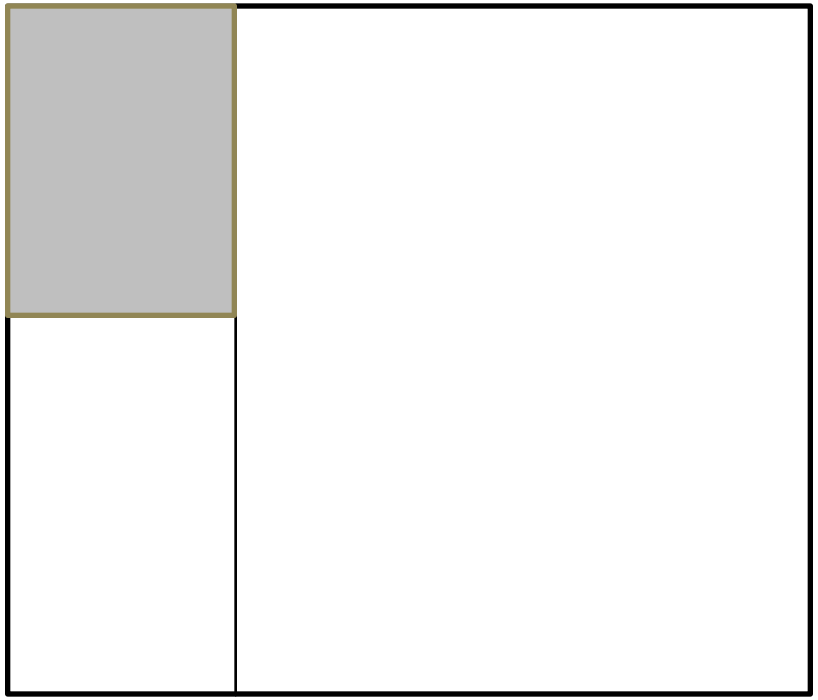
Seems feasible, however, it is not space efficiency

Pattern Visualization
(Advanced approach)

- Using bottom-up Bin-packing algorithm to pack similar APIs into one single group (according to V_{usage})
  - Bin-Packing algorithm and 2-Dim Bin-packing
  - expand board of bin, first fit always good
- (Next page show the idea)

Greedy: Largest children go first





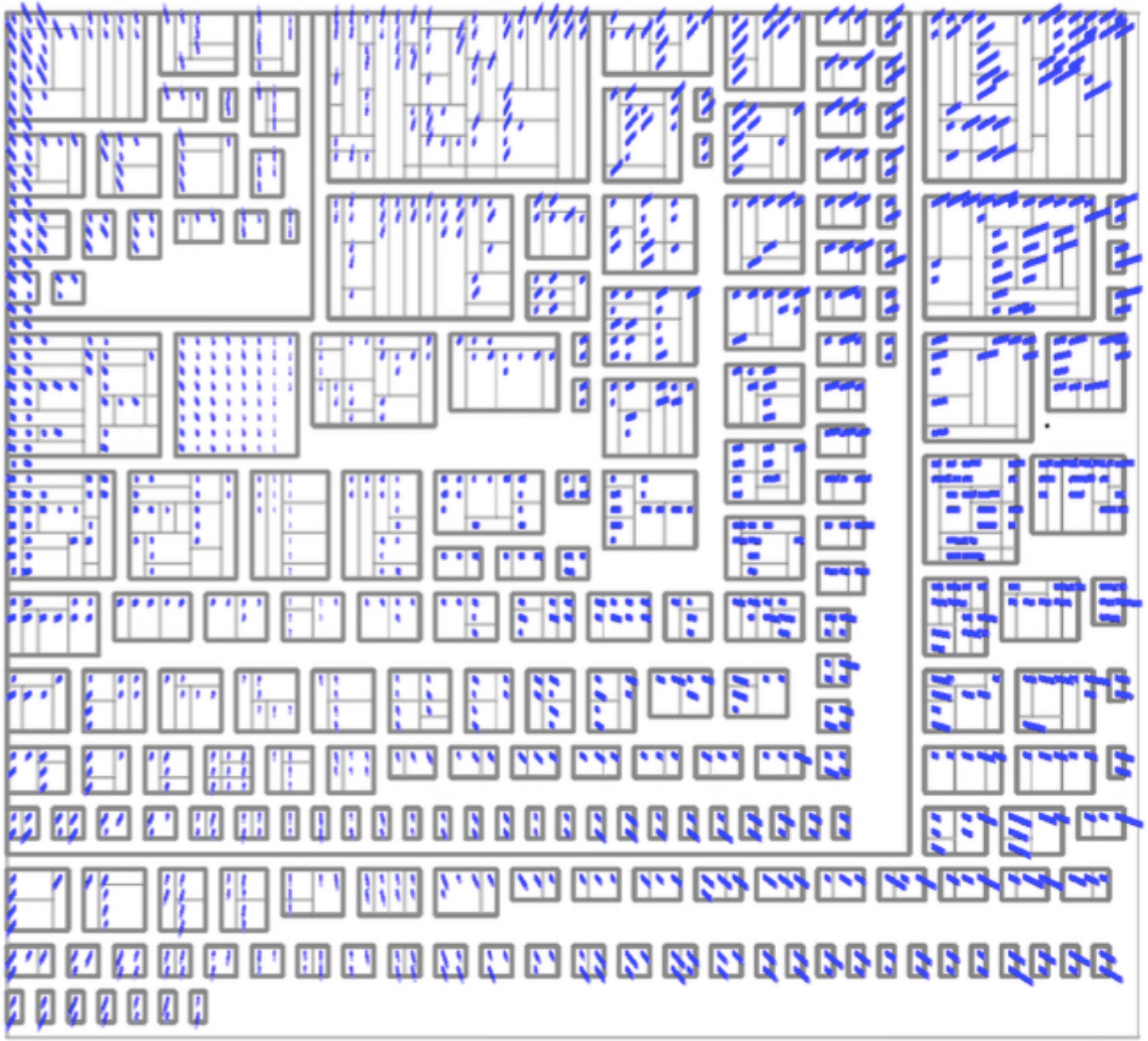


Figure 4. Combination of Bin Packing and Treemap Layout for the multi-level usage patterns of the Swing API.

Refining

- Color maps different info to this graph
  - region color indicates semantic coherence extent
  - height of box indicates population
  - box color indicates same semantic group
- Check if semantic group is the same with usage group
- Add outliers to usage pattern by check document to enrich pattern
- After these refine, we could get a usage pattern within a lib by usage consistency and semantic consistency

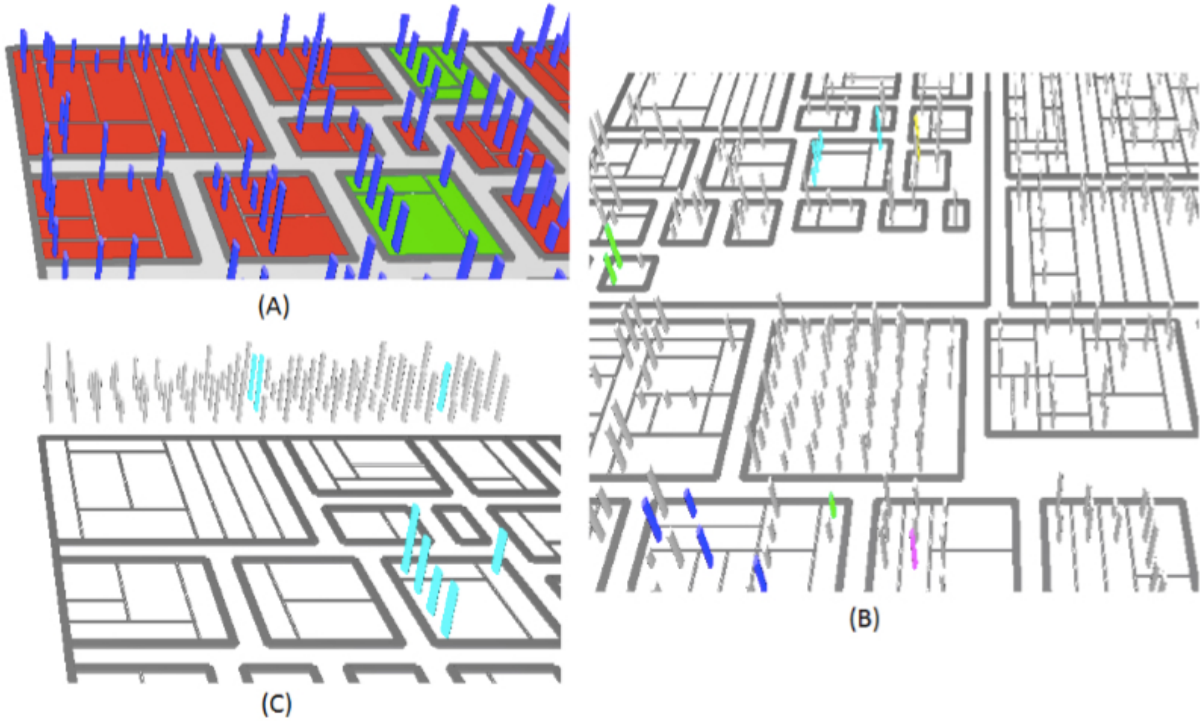


Figure 5. Usage Scenario for refining of the `GroupLayout`'s pattern of the Swing API

Thanks for watching!