# Pattern Languages for Architecture Visualization

YOUNGSU SON, NHN NEXT
JIWON KIM, SAMSUNG ELECTRONICS
SUNGWOO YU, NHN NEXT
HYUNCHUL YANG, UNIVERSITY OF SEOUL
HYESEONG OH, SEJONG UNIVERSITY
SEUNGSU JEONG, SEJONG UNIVERSITY
JEUNGWON AN, CHUNGANG UNIVERSITY

## Abstract

The architecture for Software system keeps growing besides on developing and should correspond to change to accept customer's various requirements. To achieve high quality software system, you should evaluate evolving software system's architecture continuously at any time. But only with previous partial basic method, it is too hard to identify architecture. So in this paper, we introduce various methods to visualize software system's architecture as a pattern language.

## Introduction

The architecture of software system has the possibility that can go different way from initial intent while evolving with various stakeholders. This can make potentially unintended architectural problems appear or the architecture has a chance to lose extensibility and flexibility. To prevent this situation, it is important to review architecture frequently, and to improve hazardous point.

The method of identifying software system's architecture to know surgery point is generally reviewing UML diagram like class, sequence diagram or analyzing already implemented source code. But these methods have several disadvantages. With class diagrams each evaluation scope is various but normally too big and high level at an actual implementation view. It's very difficult to identify architecture exactly, because it cannot provide proper and sufficient detail information. On the contrary, with source code analyzing, each scope is too small and it requires very wide searching range, at last very many time is consumed. So we need to find other fast method.

In this paper we will introduce several visualization methods to help us identifying software system's architecture more easily and quick. First, introduce each visualization method for implementing, then shows each methods' advantage and disadvantage. Our primary target reader of these patterns is who want to implement architecture visualization, but the contents about trade-off of each pattern can help architect people who don't know architecture visualization to understand.

## 1. Pattern Languages for Architecture visualization

### Context

In the middle-to-large size software systems needs to incorporate various customer requirements and to prepare extendable software architecture to bear continuous changes. But these days' software system development environments go to open-styles, it stimulates many people into gathering to cooperate and to keep developing. So the system configuration may go unwanted way from initial direction, and software system that has a latent problem implied by architectural unbalance from original target objective can grow weaker than before.

Therefore, you, the architect in a software system development team should identify the growing architecture of

current developing software system quick and precisely, because it's their timeless responsibility to find latent architectural problems and to make it as explicit issues then to determine its way. But this identifying effort still stroll about mere level of reviewing class diagram and analyzing source code.

**Problem**

- The time-cost limitation of code analyzing to identify software system architecture

The architect can choose the simplest way to identify software system architecture is just analyze all source codes. It can give them the most details of software system, but analyzing time is huge. So, it's not proper to identify in time. Additionally, it's difficult to capture system-wide big picture, and it's easy to lean against architect's experience, they cannot catch architectural problem with balanced view, then false negative possibility can go up.

- The limitation of precise analyzing class diagram to identify software system architecture

The other simple method to identify software system architecture is to analyze class diagram. This method shows higher abstracted view of software system architecture, it's good to capture big picture in time. But its view can go too high-level abstraction, and normally just get object relation information like inheritance and composition. It may not help to find actual detail architectural problem of software system.

**Force**

- Should help software system architecture keep extendable.
- Should inform software system architecture's latent architectural problem before actual problem occur.
- Should provide intuitive and explicit analyzing report related with software system architecture.
- Should analyze software system architecture in time, as soon as possible.

*Note: To describe the implementation method for architecture visualization using Object-Oriented concept, we will use Java Programming Language itself as example.*

**Solution**

In the problem section, two referred methods to identify architecture have a gap, from too wide and far scope to narrow and near scope. If you stand at a proper level then see software system, you can find some point that's unseen in previous methods. To stand at a proper level, the method of measuring various Metrics of software system and analyzing dependency of each component (domain) is required at the first. This metrics provides the quantity of information about software system architecture and several sign of latent problem as a form of numerical result. It's big help to understand dependency of software system architecture. It makes analyzing architecture easier and finding architectural problem simple like cross reference. But only with these Metrics and dependency information of software system's domains, you cannot get intuitive and watchable analyzing result, and it's hard for you to analyze complete in fast time.

Therefore, not just providing Metrics and dependency information as numerical texts, you should provide visualized information using various charts to make analyst understand easier. Using these architecture visualization methods requires a more initial cost like preparing visualization environment than source code analysis and class diagram analysis. But it gives faster analysis of software system architecture, well help to find latent architectural problem, to prevent problem from occurring, then to

maintain more extendable architecture easily. Finally, the architects can response to customer's requirement changes.

**Implementation**

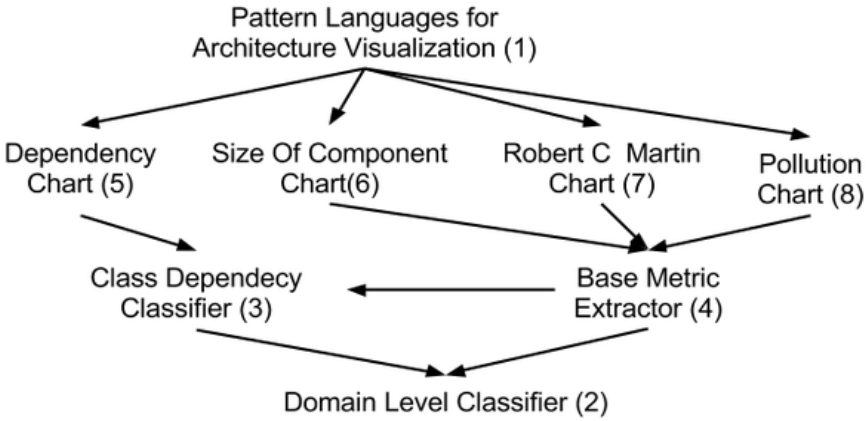
The steps to implement architecture visualization pattern is described next.

1st step: classify software system's domains based on *Domain Level Classifier Pattern*

2nd step: classify object-oriented dependency between classified domains based on *Class Relationship Classifier Pattern*

3rd step: Extract Metrics information related with software system's quality using *Base Metric Extractor pattern* with classified domains and object-oriented dependency information

4th step: Present architecture visualization with *Dependency Chart pattern, Size of Component Chart Pattern, Robert C Martin Chart Pattern, Pollution Chart Pattern* using refined Metric information and dependency information.




<Figure 1 – Relations of patterns constructing Architecture Visualization Pattern Language>

**Consequences**

- It's easier to maintain software architecture extendable by understand through visualization.
- It can decrease maintaining cost of software system's quality because it inform latent software system's architectural problem before it occur.
- software system visualization helps identifying software system architecture intuitive
- software system visualization helps identifying software system architecture faster in time

**Side-effects**

- It's required to configure additional environment more than software system development environment for architecture visualization
- It's required to learn additional knowledge about Metrics of software system quality

**Knows Uses**

- CodePro AnalytiX: Google™ providing CodePro AnaytiX also can provide architecture visualization to analyze domain dependency as a form of Chart and provide Metric information about basic software system's size. This can visualize dependency but has lack of Metric visualization.
- Structure 101: this can visualize various Metric of software quality and domain dependency, but still lack of Metric visualization.
- STAN4J: this can visualize various Metric of software quality and domain (component) dependency as a graph form, and also visual Metric as a different method, it can help you understand software system architecture easy.

**Related Patterns**

- *Dependency Chart Pattern*: a pattern related with dependency analysis by dependency Chart generation
- *Size of Component Chart Pattern*: a pattern related with system size analysis by generating Chart of system system's component.
- *Pollution Chart Pattern*: a pattern related with a method of measuring software system's purity.
- *Robert C Martin Chart Pattern*: a pattern related with abstraction level of software system's component by generating Chart

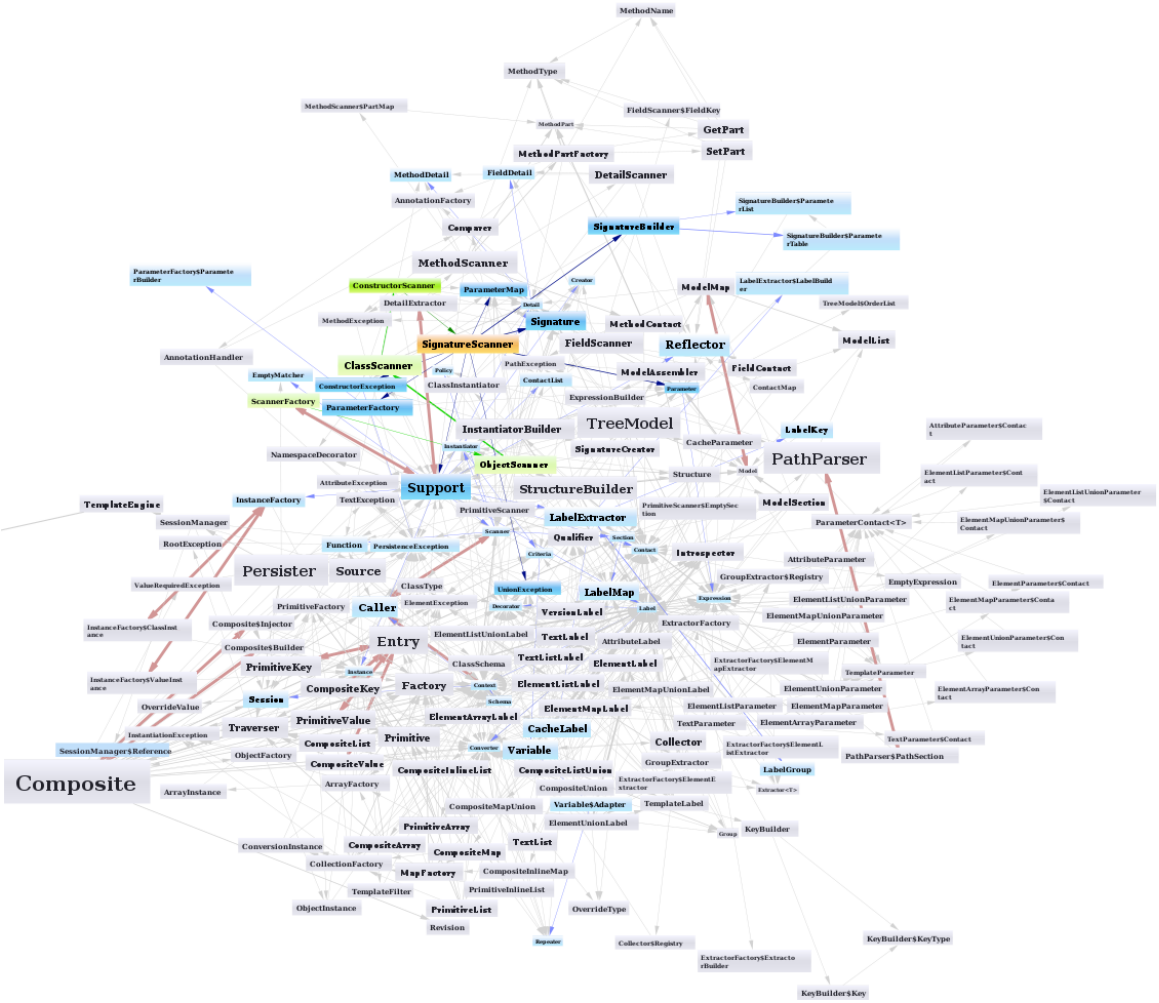
## 2. Domain Level Classifier Pattern

### Context

You want to analyze source code written by object-oriented programming language for visualizing software system's architecture. The source code is regarded to be written under MECE (Mutually Exclusive and Collectively Exhaustive) policy. MECE mean each item's role and feature is not overlapped, and it composes whole completely. For example, in a source code every field and method has no overlapped role and composes of class, and every class has no overlapped role and compose package.

### Problem

When analyzing software system's architecture, if you analyze codes that's written object-oriented language only with fine-grain level like method and field. Then its result is too complex for the reader that can be the customer. The figure 2 is its example.




<Figure 2 – Just fine-grain Level analysis result of source code>

But if you go up to coarse level, the information will become hidden and maybe too little. So you need to separate source code with focusing each reader into proper level domains.

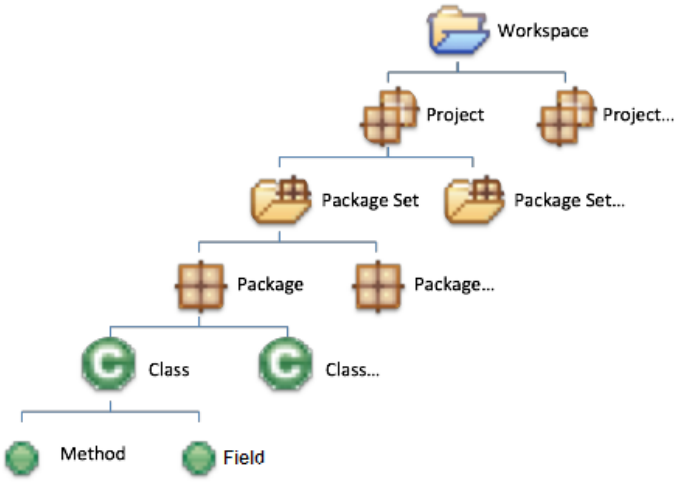
**Force**

- Should separate software system into proper domain levels to make a reader understand easily.
- Should provide information with proper domain levels as many as possible.

**Solution**

Make a source code as a form of standardization into proper domain levels that should be recognizable to a reader. In this time, the criterion should be easily understandable unit for everyone. The source code of software system is already written following MECE principle with separate units; it's criteria for a domain is included in source code. These criteria will be used as a unit that also will be understandable for a reader. For example, if you use JAVA object-oriented language, you can use 6 level domains: Method, Class, Package, Package Set, Project, Workspace, and these domains can be presented as a Tree structure.

- Method & Field: Components of Class. Field is a variable in a Class; Method is a function in a class.
- Class: This is a basic level for dependency analysis. Methods and Fields compose a class. A class is a one meaningful task unit following SRP(Single Responsibility Principle)
- Package: In a package, related Classes are associated and interacted for complex composited requirements.
- Package Set: related Packages gathers for a small sub-system.
- Project: a part of Workspace. Each Project takes a role of whole software system. This is able to be compiled
- Workspace: a software system that's complete software.




<Figure 3 – Domain levels of software source code>

With these domain levels you can analyze source code for each level and steps, a reader can see the

analysis result at a proper level easier and have intuitive understanding

**Implementation**

With JAVA language, you can implement this pattern as following steps.

- Basic steps

1st step: Convert source code into Abstract Syntax Tree (AST) to split a meaningful unit.
2nd step: Extract Class, Method domains with Top-down approach
3rd step: In an opposite, Extract Project, Workspace domains with Bottom-up approach.

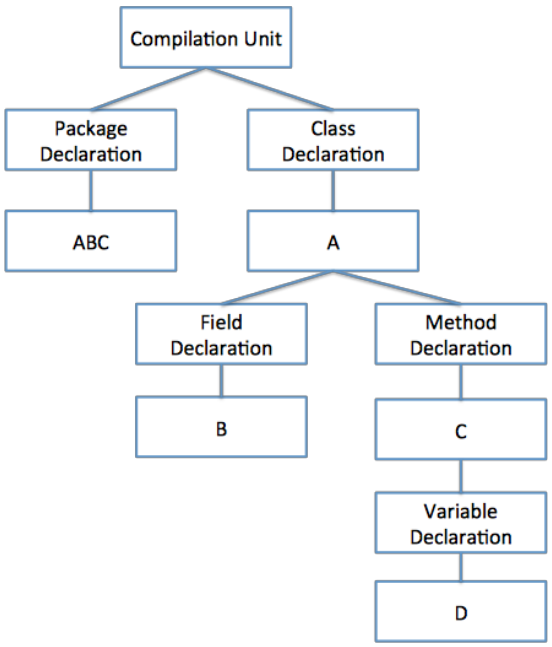
- Example of Source Code

```
package ABC;
class A{
    int B;
    void C(void){
            int D;
    }
}
```

- Detail Step Description

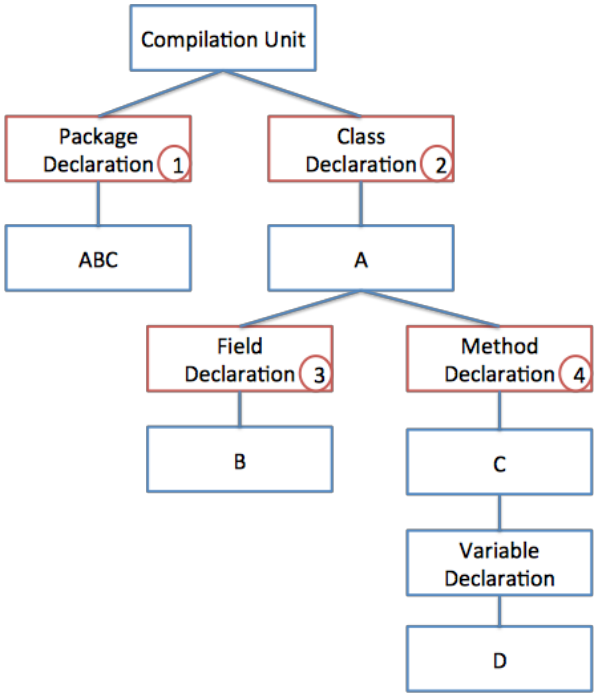
1st step: Convert each source code file into Abstract Syntax Tree (AST) to split a meaningful unit.

AST has nodes which are split with meaningful unit. You can get the AST of example source code. See figure 4



<Figure 4 – Converted Abstract Syntax Tree from Source code>

2nd step: Extract Class, Method domains with Top-down approach.
You made source code structure to AST; the class will be placed in upper node. While traversing tree, you can get package domain first, then get class, and method domains.




<Figure 5 – Classifying domains, and traversing tree with Top-Down approach>

3rd step: Extract Project, Workspace domains with Bottom-up approach.
After every one AST traversing, you can get the parent domains from the information of package, sub-system that holds class domains. Because each AST is made for one Source code file, you can just aggregate and normalize each sub-information.

**Consequences**

- It's able to analyze object-oriented language into each level domains at step by step
- It's able for a reader to understand architecture intuitive by separating each unit domains already known.

**Known Use**

- Structure 101: this can visualize domain dependency using "Domain Level Classifier Pattern" and show architecture separating each domain level to make people understand easily at once.

- Eclipse IDE: famous IDE also famous JAVA programming development IDE, in the Package Explorer, it can visualize each domain level using “Domain Level Classifier Pattern” for a developer

**Related Pattern**

- Base Metric Extractor Pattern: for each domain level, the unit and type of Domain level varies.
- Class Dependency Classifier Pattern: Define all dependency through domains

### 3. Class Dependency Classifier Pattern

#### Context

Each domains constructed by Domain Level Classifier Pattern has explicit boundary and mutually exclusive. But if actually there is no dependency between domains, it cannot be a software system, the domain dependency is natural and this composes one sub-system. The dependency is a kind of bridge between domains that have exclusive boundary

#### Problem

- Need to find exact dependency between domains to identify architecture

Domain Level Classifier Pattern just defines and classifies domains. So the dependency information between each domains is not included, it's hard to analyze exact architecture.

#### Forces

- Should define all type of dependency through all domains
- Should identify exact levels and names of depending two domain

#### Solution

Search and define all dependency through all domains. The dependency should include all inheritances and compositions occurring in a software system exactly. With defining dependency among Class domains and its sub-domains, you can search all dependency because all upper domains is based on lower domains. And for more exact searching dependency, you should know the Source and Target between domains, and whose level of domain to define dependency exactly. Generally, the dependency of software system can be classified as basic classification criterion below.

- Dependency is defined between Source and Target domains
- Classify domain based on various relationship like inheritance and reference used in a object-oriented language
- Upper domain should be constructed based on dependency to lower domain

#### Implementation

Software system's dependency is very different for each target object-oriented language. In this pattern, using JAVA language, you can implement this pattern with following steps

- Basic steps

1st step: Implement Domain Level Classifier Pattern
2nd step: Define all dependency through Class domain and all sub-domains
3rd step: Extract defined dependency at 2nd step from Source code and Target Code
4th step: Write extracted dependency into template defined as table

- Details Step Description

1st step: Implement Domain Level Classifier Pattern

This pattern is defining dependency between domains. So at the first, all domains should be defined already to find each dependency exactly.

2nd step: Define all dependency through Class domain and all sub-domains

The definition of dependency can be any types on various languages, in the JAVA of Object-oriented language; you can find the definition of dependency like following table.

| Source | Dependency Kind | Target | Description |
|---|---|---|---|
| Class | Extends | Class | The source type extends/implements the target type |
| Class | Contains | Class | The source type contains a nested class whose type is the target type |
| Class | Contains | Field | The source type contains the target field |
| Method | Returns | Class | The source method declares a return type based on the target type |
| Method | Has param | Class | The source method declares a parameter based on the target type |
| Method | Throws | Class | The source method declares the target type in its throws clause |
| Method | Calls | Method | The source method code invokes the target method |
| Method | Accesses | Field | The source method code accesses the target field |
| Field | is of type | Class | The source field's type is based on the target type |
| Any | References | Class | Any relation not<br>covered by one of the others, e.g.<br><br>- generic type parameters/bounds<br>-parameter/return types of called methods<br>- local variable types<br>- types declared in catch blocks<br>- types used in an instanceof operator |

<Figure 6 – The dependency types of Java>

3rd step: Extract defined dependency at 2nd step from Source code and Target Code

3 - 1: the case of already compiled target code exists.

There is easy implementation method with already compiled target code. All domain names are already exactly described in target code, so you can know the dependency between domains easily. But you should wait for implementing done that code is compiled successfully, this method has disadvantage that it's depends on IDE or useful only for a kind of libraries like JAR.

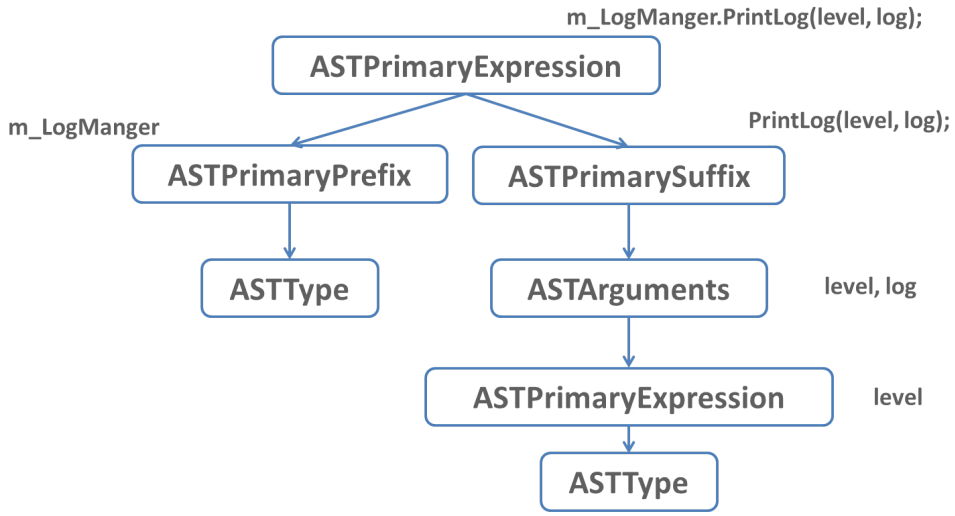
3 - 2: the case of only source code exists.

This is the method of searching dependency from source code. This method requires only source code, there is no restriction related with IDE. But it's hard to extract dependency from source code than compiled target code. Moreover, if it is linked with external library, the result may not be perfect.

When implementing pattern, with just parsing code it's difficult to get exact dependency, because source code can have a redundant name or recursive call. So it's more convenient and right way to convert code to Abstract Syntax Tree (AST) for searching dependency of domains. AST is a tree form of each parsed item with syntax separator. The following figure is a example of AST for Java source code.

```
public void PrintLog(Level level,String log){
    m_LogManager.PrintLog(level, log);
}
```

<Figure 7 – The example of Source code>




<Figure 8 - AST>

With converting source code to AST above, you can search which method is called in "void PrintLog()" with finding m_LogManager type, and the searching speed will be better. For more speed, with excluding branches that's not be analyzed, it'll be much faster than analyzing all source code.

4th step: Write extracted dependency into template defined as table

Write extracted dependency in previous step as "Source – Relationship – Target" form. It can be various methods, one is writing "Edge" between domains, or making "List", but each Source and Target's domain should be distinguished exactly when writing dependency information

**Consequences**

- This is a basic data to visualize dependency
- It's able to define all dependency through domains
- It's able to search exactly the dependency between domains.

**Related Patterns**

- *Domain Level Classifier Pattern*: This pattern is to search dependency between each domain; Domains should be classified using Domain Level Classifier Pattern.
- *Base Metric Extractor Pattern*: Extract Metrics based on specific elements in dependency of domains
- *Dependency Graph Pattern*: Visualize dependency through all domains using dependency between basic domains

## 4. Base Metric Extractor Pattern

### Context

To analyze software system's architecture, just using dependency is not sufficient because it can make analyzing scope narrow, so prevent us from approaching at a various view. And the Chart only with dependency just provides the chance of finding architectural problem like cross-reference, there is no way to check the problem related with the size of source code and complexity.

### Problem

- There is no other analyzing method except for dependency analysis

There is restriction possible to search architectural problem only with dependency analysis. So you need another assist to fill up. Another method also should be very intuitive and explicit and fast in-time to identify architecture.

### Force

- Should need another method to get more information than dependency
- Should check software system's problem intuitive and explicit
- Should analyze software system fast in-time.

### Solution

The software Metric can solve this problem that's index of software system's quality. This index is objective and scientific quantifying numbers from software's feature and characteristics in software lifetime based on various software's measuring technology. As more detail description, It's a kind of process about problem analyzing process Defining various problems, Collecting basic related data, Analyzing problems with objective and scientific numerical methods, Searching the relationships of various problems, Representing these information into software indexes and Finding solutions from indexes.

Among these indexes there are 4 types of software Metric that can help to improve software quality. First, "*Count Metric*" represents counts of package, class, method, field, and code line simply. Second, "*Complexity Metric*" represents circular complexity, heaviness, tangling in a code. Third, "*Robert C. Martin Metric*" represents dependency between codes. Finally, "*Chidamber & Kemerer Metic*" represents inheritance between codes in number.

### Consequences

- It's possible to analyze software system's code quantitatively.
- It's possible to analyze objectively because it's a numerical Metric of software system's quality.
- It's possible to search software system's problem intuitively.
- It's possible to analyze software system at a various view.

### Side-effects

- It may not apply real development status just with formative analysis

**Known Use**

- STAN4J: an Eclipse IDE Plugin, using "*Base Metric Extractor*" pattern provides quantitative numbers of software system. And it visualize architecture using it's numbers
- Restructure 101: an architecture visualization and refactoring tool, using "*Base Metric Extractor*" pattern calculate quantitative numbers, to visualize with Metric index where problems are and need modification.

**Related Patterns**

- *Pollution Chart Pattern*: Visualize pollution degree with criteria defined from Base Metric.
- *Size of Component Chart Pattern*: Visualize sizes of ELOC in Base Metric.
- *Robert C. Martin Chart Pattern*: Visualize abstraction degree using Robert C. Martin indexes in Base Metric.

### 4.1 Count Metric

"Count Metric" can represent how many codes are included in each domain, and how many lines are. This can provide whole size of software; also give a chance to know development progress. Metric is limited for each domain level or its information is changed. See following table.

| Count Metric | Domain | | | | | |
|---|---|---|---|---|---|---|
| Number of Libraries | ✓ | | | | | |
| Number of Packages | | ✓ | | | | |
| Number of Top Level Classes (Units) | | | ✓ | ✓ | | |
| Number of Member Classes | | | | | ✓ | |
| Number of Methods | | | | | ✓ | |
| Number of Fields | | | | | ✓ | |
| Average Number of Top Level Classes per Package (Units / Package) | | ✓ | | | | |
| Average Number of Member classes per Class (Methods / Class) | | | ✓ | ✓ | | |
| Average Number of Methods per Class (Methods / Class) | | | ✓ | ✓ | | |
| Average Number of Fields per Class (Fields / Class) | | | ✓ | ✓ | | |
| Estimated Lines of Code (ELOC) | | | ✓ | ✓ | ✓ | ✓ |
| Estimated Lines of Code per Top Level Class (ELOC / Unit) | | | ✓ | ✓ | | |

<Table 1 - Count Metric>

### 4.2 Complexity Metric

"Complexity Metric" is the most important index to visualize architecture. "Cyclometic Complexity Metric" [Gill+91] is one of them, and represent how many branches is included in code. More branches means more exception generations, so it's complex code. "Fat Metric" means numbers of dependency between child domains in a parent domain. More dependency in child domains means fatter parent domain. "Tangled Metric" is an index that can figure out reverse-reference. In one domain, if two nodes have reference to each other, it's called as Tangled state, changes in one side can affect other side, and again back. So if it's detected, the architecture is not good.

| Complexity Metrics | Domain | | | | | |
|---|---|---|---|---|---|---|
| Cyclomatic Complexity (CC) | | | | | | ✓ |
| Average Cyclomatic Complexity | | | ✓ | ✓ | | |
| Fat | | | ✓ | ✓ | ✓ | |
| Fat for Library Dependencies (Fat – Libraries) | ✓ | | | | | |
| Fat for Flat Package Dependencies (Fat – Packages) | | ✓ | | | | |
| Fat for Top Level Class Dependencies (Fat – Units) | | ✓ | | | | |
| Tangled | | | ✓ | | | |
| Tangled for Library Dependencies (Tangled – Libraries) | ✓ | | | | | |
| Tangled for Flat Package Dependencies (Tangled – Packages) | | ✓ | | | | |
| Average Component Dependency between Libraries (ACD – Library) | ✓ | | | | | |
| Average Component Dependency between Packages (ACD – Package) | | ✓ | | | | |
| Average Component Dependency between Units(ACD – Unit) | | ✓ | | ✓ | | |

<Table 2 - Complexity Metric>

### 4.3 Robert C. Martin Metric

In a package domain, this Metric gives chance to check the package is well abstracted or not easily. “Afferent Coupling” shows the domain has how many references. “Efferent Coupling” shows the domain is how many referenced. “Abstractness” shows amount of abstract. With calculating these numbers you can draw Robert C. Martin Chart [Martin 00] to see which package need to be abstracted in one picture.

| Robert C. Martin Metrics | Domain | | | | | |
|---|---|---|---|---|---|---|
| Afferent Couplings (Ca) | | | |  |  | |
| Efferent Couplings (Ce) | | | |  |  | |
| Abstractness (A) | | | |  | | |
| Instability (I) | | | |  | | |
| Distance (D) | | | |  | | |
| Average Distance | |  | | | | |
| Average Absolute Distance | |  | | | | |

<Table 3 - Robert C. Martin Metric>

### 4.4 Chidamber & Kemerer Metric

This index is related with only class. This focuses on the number of parent class, the number of child class, the dependency of field and method in the class. You can know the pollution amount of class using this Metric.
[Hitz+96]

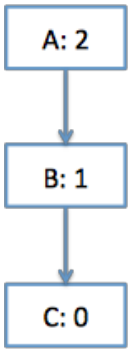
| Chidamber & Kemerer Metrics | Domain | | | | | |
|---|---|---|---|---|---|---|
| Weighted Methods per Class (WMC) | | | | |  | |
| Depth of Inheritance Tree (DIT) | | | | |  | |
| Number of Children (NOC) | | | | |  | |
| Coupling between Objects (CBO) | | | | |  | |
| Response for a Class (RFC) | | | | |  | |
| Lock of Cohesion in Methods (LCOM) | | | | |  | |
| Average Weighted Methods per Class | |  | |  | | |
| Average Depth of Inheritance Tree | |  | |  | | |
| Average Number of Children | |  | |  | | |
| Average Coupling between Objects | |  | |  | | |
| Average Response for a Class | |  | |  | | |
| Average Lock of Cohesion in Methods | |  | |  | | |

<Table 4 - **Chidamber & Kemerer Metric**>

**Implementation**

Before running "*Base Metric Extractor*" pattern, "*Domain Level Classifier*" pattern should be done. Separating object-oriented language into domains, then getting indexes of code matched for each domain can represent more intuitive software architecture visualization. In addition to this, "Class Dependency Classifier" pattern also should be done to visualize dependency between domains in numbers.

To implement "*Base Metric Extractor*" pattern, you should get four type of Metric described above. In this section, we will describe the hardest achievable index, ACD (Average Component Dependency)[Jungmayr 02]. ACD is an index that represents all dependency through all classes. For example, three classes, Class A, B, C, If Class A calls Class B, then Class B calls Class C, Class A has dependency to Class B and C, Class B has dependency to Class C. ACD represent average of these dependency count.



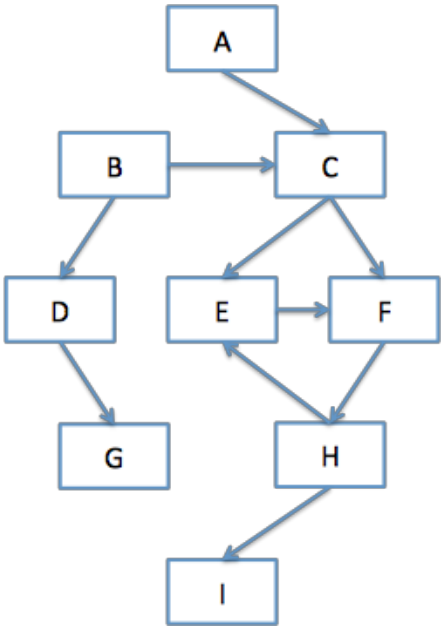

<Figure 9 – Simple Example of ACD>

- Basic Procedures

1st step: Represent dependency between classes as a graph. For easier calculation of ACD, represent Class as Node, Dependency as Edge of graph.
2nd step: Convert cyclic nodes into one semantic node in the graph.
3rd step: Search Leaf node in the tree graph.
4th step: the CD value of new leaf node can be calculated using the sum of CD value of leaf node that's already excluded because of other leaf node's call.
5th step: unpack packaged one semantic node of cyclic nodes; apply same CD value to all nodes.
6th step: Calculate Average of CD for all nodes.

- Detail procedure description

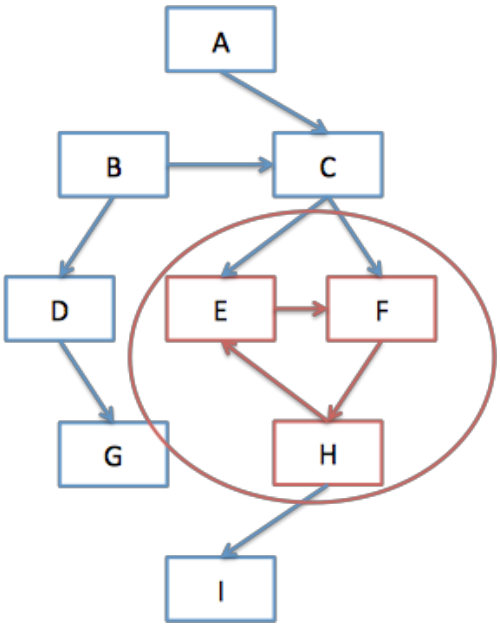
1st step: Represent dependency between classes as a graph. For easier calculation of ACD, represent Class as Node, Dependency as Edge of graph.




<Figure 10 – Dependency of classes represented as a graph >

2nd step: Convert cyclic nodes into one semantic node in the graph.

Set aside cyclic nodes from initial searching by grouping into one semantic node, you can remove redundant searching and improve searching speed. As a result of this step, the graph becomes tree form.

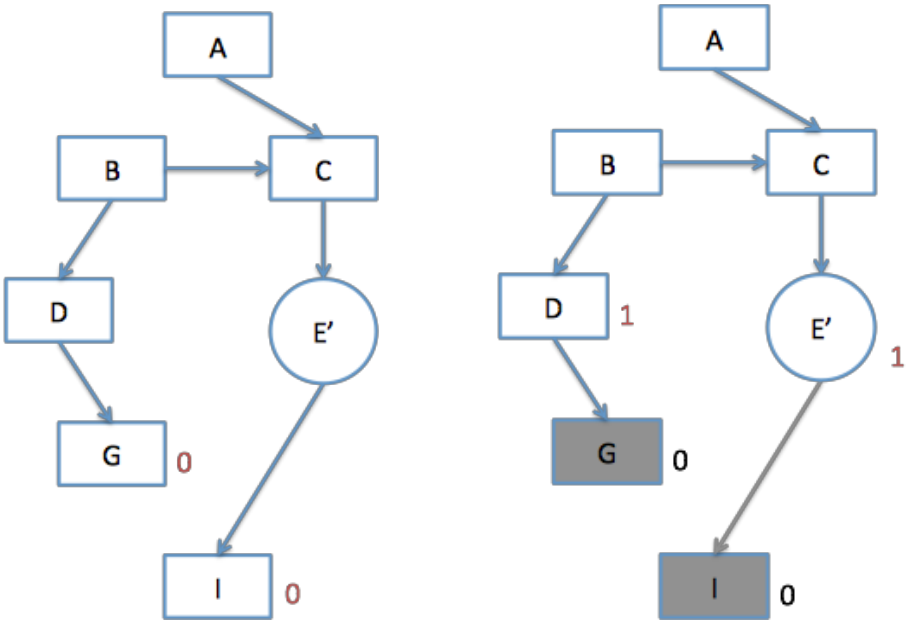



<Figure 11- Grouping cyclic nodes>

3rd step: Search Leaf node in the tree graph.

Leaf node has no other edge can go other nodes; you can get CD of leaf node as a zero. Then detach leaf

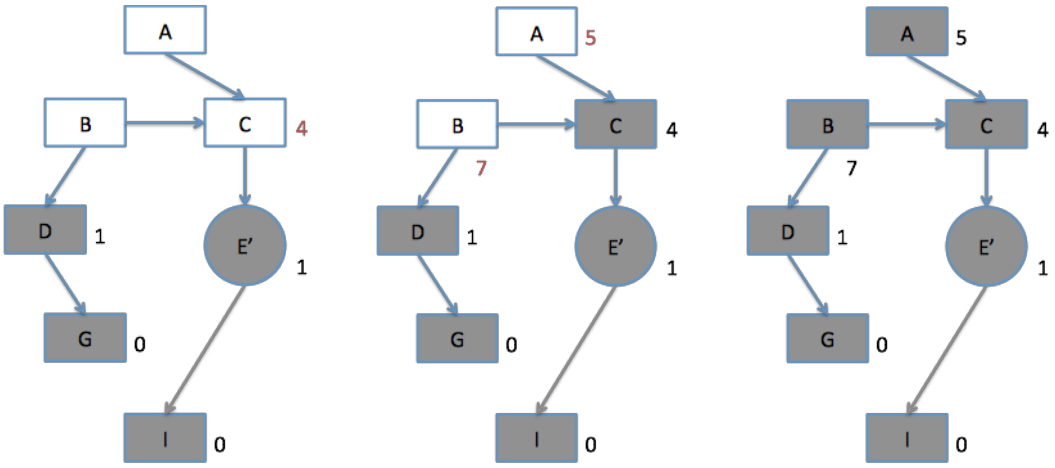
node from graph to make new leaf node.




<Figure 12 – Detach Leaf node>

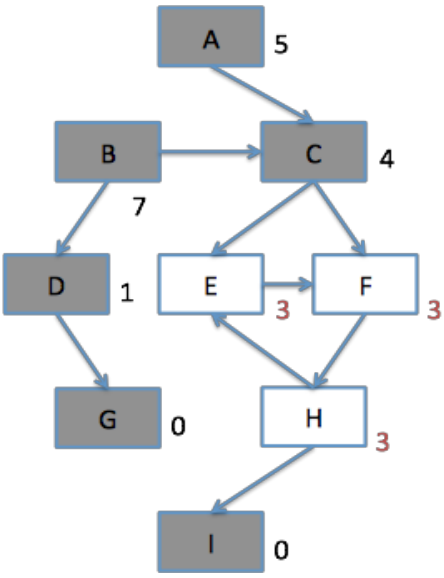
4th step: the CD value of new leaf node can be calculated using the sum of CD value of leaf node that's already excluded because of other leaf node's call.

While this step, if parent calls child node has a packaged semantic node in step 2, you should add the node number in a semantic node into CD. Then again detach leaf node to make new leaf node. Repeat this step until all nodes are detached.



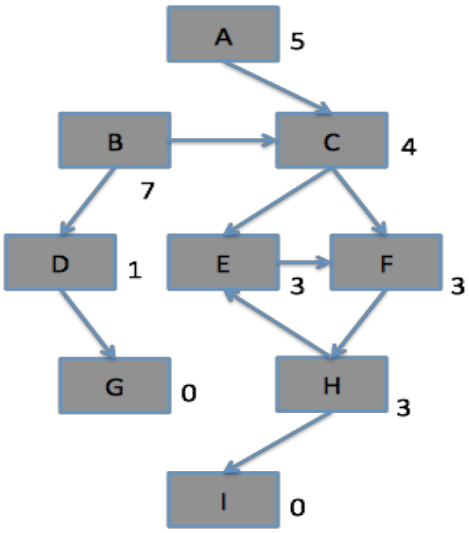

<Figure 13- Repetitive procedure of calculation of CD for each node>

5th step: unpack packaged one semantic node of cyclic nodes; apply same CD value to all nodes.




<Figure 14 – Recover cyclic nodes, and update cyclic node's CD value>

6th step: Calculate Average of CD for all nodes.




(5 + 4 + 7 + 1 + 3 + 3 + 0 + 3 + 0) / 8 = 3.25

<Figure 15- Calculate Average of CD>

**5. Dependency Chart Pattern**

**Context**

Dependencies (inheritance, references, etc.) of the software system belonging to a variety of domains are important in the analysis of architecture drawn to help the whole picture of the software system architecture serious problems. For this reason, the analysis had introduced a class diagram or in the source code analysis to identify the dependencies between these domains with a lot of emphasis on analysis. Among other things, mostly through the analysis of a class diagram determine the architecture of these dependencies.

As you look too wide through the analysis of a class diagram, too little information that can be obtained by the analysis. Also as you look too deeply through the analysis of the source code, it takes long time for a lot of analysis and to draw the big picture for the overall architecture, difficult to understand the architecture is irrelevant.

**Problem**

- Class diagrams and code analysis methods to discover the architectural issues that require a lot of time

Class diagrams and code to analyze the dependencies between the domains does not contain a variety of architectural issues such as cross-reference that is inherent in software systems is very difficult to find.

- The software architecture of the system is difficult to estimate using the class diagrams and code analysis methods

These analytical methods, it is difficult to understand the overall flow and architecture of a software system. This dependence appears only just in the case of a class diagram part to its dependence on the incidence or does not provide, because it does not provide any basic information at the code level, the analyst should identify them.

**Force**

- The software architecture of the system should be able to intuitively analyze problems.
- Class diagram does not provide more detailed information should be provided.
- The summary information should be provided rather than the code-level of analysis.
- The overall architecture of software system dependence should be identified.

Dependencies in a software system using visualization give analysts more information. Basic inheritance and the level of abstraction of class diagrams are included by the visualization of the dependence information by default. In addition to being able to see the overall flow of the current software systems, the software systems have internal problem such as the granularity or a cross-reference architecture issues can be figure out very easily.

| | Class diagram level | Code level | Dependence level |
|---|---|---|---|
| Dependence that can be analyzed | Classes in the domain of inheritance and dependency approach | All of the dependencies that exist within the software system | All of the dependencies that exist within the software system |
| Target to be analyzed | Class diagram | Actual code | Dependencies in a software system |
| Architectural information that can be achieved with the intuitive | Approximate dependencies and configuration information | | The dependence of the actual software system and configuration information |
| Architectural problems that you can see the intuitive | Level of abstraction | | Cross-reference, Abstraction Level |

<Table 5 - Comparison of analytical methods>

**Consequences**

- The software is able to analyze the system within an acceptable range.
- The software architecture of the system will be able to identify in a faster time.
- The software architecture of the system issues easier to understand.

**Side-effects**

- The visualization of software systems analysis for more complex dependence becomes to need a lot of time.
- The non-dependence such as the complexity, granularity of information, and CC cannot be analyzed; there is a limit to the analysis.

**Knows Uses**

- CodePro AnalytiX: Google provides the static analyzer as AnalytiX placing the circle in the form of dependencies between the domains of the software system to help your analysis.
- Structure 101: The Structure 101 architecture visualization tool helps your analysis through DSM in the representation of the dependencies between the domains of the software system.
- STAN4J: The STAN4J architecture visualization tool helps your analysis through Digraph in the representation of the dependencies between the domains of the software system

**Related Patterns**

- *Dependency Structure Matrix Pattern* : One of the sub-patterns to visualize the pattern of dependency

- *Dependency Composition Digraph Pattern* : One of the sub-patterns to visualize the pattern of dependency
- *Class Dependency Classifier Pattern*: The basic data for all of the dependencies between each domain.

### 5.1 Dependency Structure Matrix (DSM) Pattern

**Context**

Domains (packages, classes, methods, etc.) that exist in the software system, and there are multiple dependencies between these elements. Because they have very large amounts in even single software system should be able to give this summary.

In addition, the people that you want to analyze a software system based on this software to identify the configuration of the entire system quickly and easily as possible, so you have to figure out the problem and techniques to visualize this information are required.

**Problem**

- Dependency Chart patterns can be visualized appropriately.

Base on the Dependency Chart patterns, you need a way to be people that you want to analyze software systems, the domain of dependence of a variety of internal software systems faster and more easily, and intuitively understandable.

**Force**

- The dependence of the domain of the software system should be able to identify easily.
- The architectural problem of the software system should be able to figure out easily.
- The overall architecture of the software system should be able to understand easily.

**Solution**

The most basic way to visualize the dependencies exist the Dependency Structure Matrix (Design Structure Matrix, DSM). (Neeraj05). The DSM is a way to visualize the dependencies in the form of a matrix showing. How to represent a graph in graph theory, by default, and adjacency matrix visualization is done in the same way.

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| Element A | X | | | **3** | | |
| Element B | | X | | | | |
| Element C | 2 | | X | | | |
| Element D | 2 | 5 | | X | | |
| Element E | 4 | 3 | 5 | | X | **2** |
| Element F | 2 | 1 | 2 | 8 | 9 | X |

<Table 6 – Example of DSM>

As shown in the table, DSM to make it easier to identify the dependencies of a software system in the form of the matrix shows the dependence of the whole that you want to analyze.

**Implementation**

There are many kinds of methods to implement the DSM. The one of them describes how to visualize the DSM at the level of a particular domain.

- the basic procedure

Step 1: Select the basic tool for the visualization
Step 2: Extract the dependencies of the project to visualize based on the Class Relationship Classifier pattern.
Step 3: change the extracted dependence information to the point of domain level criteria.
Step 4: Make the DSM visualization based on changed dependency information.

- Example Source Code (Analyzed)

As an example to analyze code are as follows:

```
package PackageA;
class ElementA{
        ElementB b;
        ElementB getElementB(){return b;}
}
package PackageB;
class ElementB{

        ElementC c;
        ElementC getElementC(){return c;}
}
package PackageC;
class ElementC{
        ElementA getElementA(){return new ElementA();}
}
```

- The procedure described

Step 1: Select the basic tool for the visualization.
Select the tool you want to develop for architecture visualization tools to implement graphics or charts. If using pure web development could be the SVG or HTML5 Canvas. The developers want their primary tool for implementing visualization elements can be selected autonomously.

Step 2: Extract the dependencies of the project to visualize based on the Class Relationship Classifier pattern.

Based the Class Relationship Classifier pattern previously described, extract all of the 10 types of dependencies that exist within the software system to visualize.

| Source | Relationship | Target |
|---|---|---|
| PackageA.ElementA | Contain | PackageB.ElementB |
| PackageA.ElementA.b | Is of Type | PackageB.ElementB |
| PackageA.ElementA.getElementB() | returns | PackageB.ElementB |
| PackageB.ElementB | Contain | PackageC.ElementC |
| PackageB.ElementB.c | Is of Type | PackageC.ElementC |
| PackageB.ElementB.getElementC() | returns | PackageC.ElementC |
| PackageC.ElementC.getElementA | Contain | PackageA.ElementA |
| PackageC.ElementC.getElementA() | returns | PackageA.ElementA |

<Table 7 – Example of extracted dependency from test code >

Step 3: change the extracted dependence information to the point of domain level criteria.
Adjust to point of the criteria domain to analyze the domain-level dependence information extracted from the starting element and the elements of arrival. The simply for example, to visualize the criteria "package" unit, Draw the DSM changed to "PackageC" which is the criteria package unit of information belonging to the sub-unit "method" of information rather than "Package" unit such as “PackageC.ElementC. getElementA"

Step 4: Make the DSM visualization based on changed dependency information.
Visualize the actual DSM based on dependency information changing the point of domain. Fill the number of dependencies in the changed dependencies is the number of cases within same all of start element and arrival element.

| | 1 | 2 | 3 |
|---|---|---|---|
| PackageA: 1 | X | | 2 |
| PackageB: 2 | 3 | X | |
| PackageC: 3 | | 3 | X |

<Table 8 – Example of Complete DSM>

**Consequences**

- It is easy to analyze the architecture through the dependence of the software system.
- It is easy to find the architecture issues in the internal of software system.

**Side-effects**

- As the software of the system grows, it is difficult to analyze.
- It is not possible to identify the exact kind of dependencies.
- It is difficult to consider the overall architecture of a software system.

**Knows Uses**

- The architecture visualization tool Structure 101 shows the each package and the dependencies between the classes represented by DSM to help in the analysis of the user.

**Related Patterns**

- It is the pattern of definition, extraction about dependence on the type of software system.
- It is the pattern to solve the drawbacks of having a DSM.

### 5.2 Dependency Composition Digraph Pattern

**Context**

If you implement the visualization of Dependency Graph Pattern based on DSM Pattern, architecture make it easier to identify compared to the analysis of source code or class diagrams and it is relatively easy to find the software system architecture issues. However, as the size of the target to be analyzed is huge, the DSM Pattern is difficult to analyze the dependence shown in the table as well as it is difficult to identify the exact kind. Plus, as visualize the data based on a matrix, analysts is difficult to determine intuitionally an overall architecture for a software system, there is a drawback.

**Problem**

- The software increases, the size of the system is difficult to analyze issues

The larger the size of the software system using DSM pattern matrix increases proportionally. It has a problem which is difficult because of the matrix analysis, therefore getting a lot of elements of the domain that you want to analyze a lot more to look bigger too.

- Due to the complexity of matrix structure, it is difficult to understand the overall architecture.

DSM Pattern represents in the form to fill in the numbers at the dependence of the matrix elements. Although this method is convenient to identify the dependency of one element, it is a very tough way to determine the dependence of the entire software system.

- Problem for identify the exact the dependence of the software system

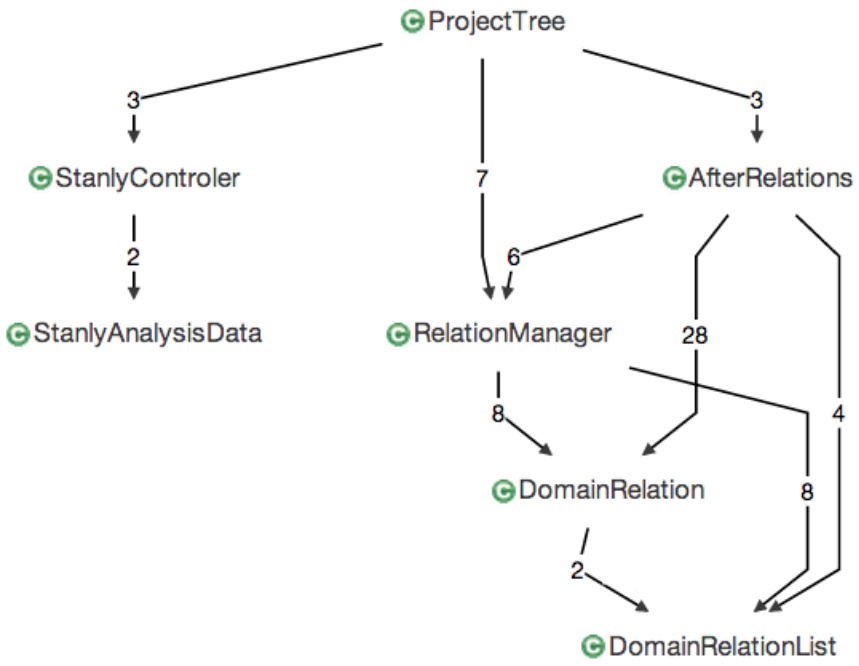
Since DSM pattern simply indicates the number, it is able to estimate the degree of dependence. But no way can you know what type of dependencies in the dependency.

**Force**

- Should be easy to identify the dependency of the elements (packages, classes, methods, and etc.)
- The software architecture of the system should be easy to find a problem.
- The software architecture of the system, to grasp an intuitive and should be explicit.

**Solution**

These shortcomings by utilizing visualization in the form of a directed graph with weights, a way of showing the dependence of the software system can be improved. Basically, considering that it is derived from the adjacency matrix, one of the ways to display a graph of the graph theory, the DSM when you draw a graph, it can improve the drawbacks associated with the matrix visualization.
And when you want to visualize a directed graph with weights, depending on the cross-reference each vertex provides a hierarchical representation, if the dependence of each element can see at a glance.




<Figure 16 – Example of Layered directional graph>

**Implementation**

The pattern can be implemented in the following procedure.

- The basic procedure

Step 1: Select the basic tool for the visualization

Step 2: Generate DSM based on the DSM pattern. And then create a directed graph with weights based on the results.

Step 3: Create a hierarchical layout that you can draw a directed graph generated.

Step 4: Draw the graph Based on the layout.

- The procedure described

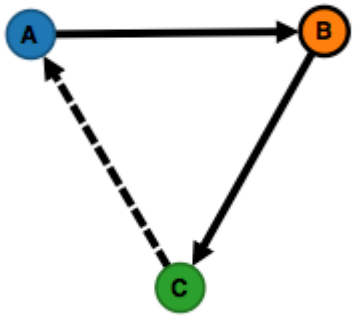
Step 1: Select the basic tool for the visualization.

Select the tool you want to develop for architecture visualization tools to implement graphics or charts. If using pure web development could be the SVG or HTML5 Canvas. The developers want their primary tool for implementing visualization elements can be selected autonomously.

Step 2: Generate DSM based on the DSM pattern. And then create a directed graph with weights based on the results.

Priority, Generate DSM base on DSM Pattern because this pattern is to create a graph with the resulting DSM, Then draw the directed graph with weights using expressed as an adjacency matrix representing the DSM.

| | 1 | 2 | 3 |
|---|---|---|---|
| PackageA: 1 | X | | 2 |
| PackageB: 2 | 3 | X | |
| PackageC: 3 | | 3 | X |




<Figure 17: Converting DSM into graph >

Step 3: Create a hierarchical layout that you can draw a directed graph generated.
On the basis of the information about the directed graph generated, create a hierarchical layout. In order to make typically a directed graph need to calculate the hierarchical layout using Sugiyama algorithm ([Bastert+01]). However, Sugiyama algorithm must be calculated based on some elements of the task greedy algorithm due to the NP-Hard problem. Therefore calculated the hierarchical layout, it should be noted that they do not have the optimal solution.

Step 4: Draw the graph Based on the layout.
Draw a directed graph based on created layout using the selected visualization tool. To draw the graph Edge, configure the typical dependencies (inheritance, references, etc.) using the symbols that are defined in the class diagram.

<Figure 18: Complete directional graph >

**Consequences**

- It is easy to find a software system architecture issues.
- As form a hierarchy, it is easy to identify the dependence of the domain.
- It is intuitively grasp of typical dependence represented by of the shape of the edge using symbol of the corresponding class diagram.
- It is easy to understand the overall architecture of a software system.
- It is enough to analyze even though the size of the software system grow.

**Side-effects**

- It is time-consuming when calculating the layout for visualization.
- The implementing visualization is difficult because it is not a standardized Chart type.

**Knows Uses**

- STAN4J: The visualization tools STAN4J in composition view, there are representation of the dependence of the software system, elements (packages, classes, etc.) of the software system in the form of a directed graph.

**Related Patterns**

- *Class Relationship Classifier pattern*: It is the pattern of definition, extraction about dependence on the type of software system.

**6. Size of Component Chart Pattern**

**Context**

If software systems are going to be huge, you may have a problem with architecture and the will be difficult to manage since a tiny object has been distributed function. Therefore, to maintain an appropriate level to the size of the components of software system is a very important factor in managing the architecture.

**Problem**

- The absence of a way to determine the size of the software system

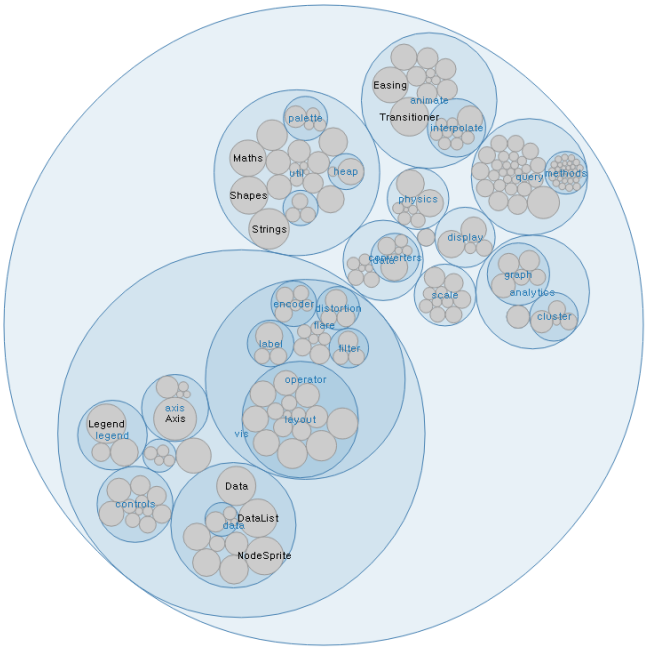
In order to maintain a stable architecture of a software system, you should be able to see the size of the domain level to keep the Level of Scale which is one of the most basic concepts of the architecture. If package at the package level, If class at the class level, you should be able to see the size and the visualization techniques are needed.

**Forces**

- The size of the domain should be able to identify at one view.
- Domain-level size should be able to understand.

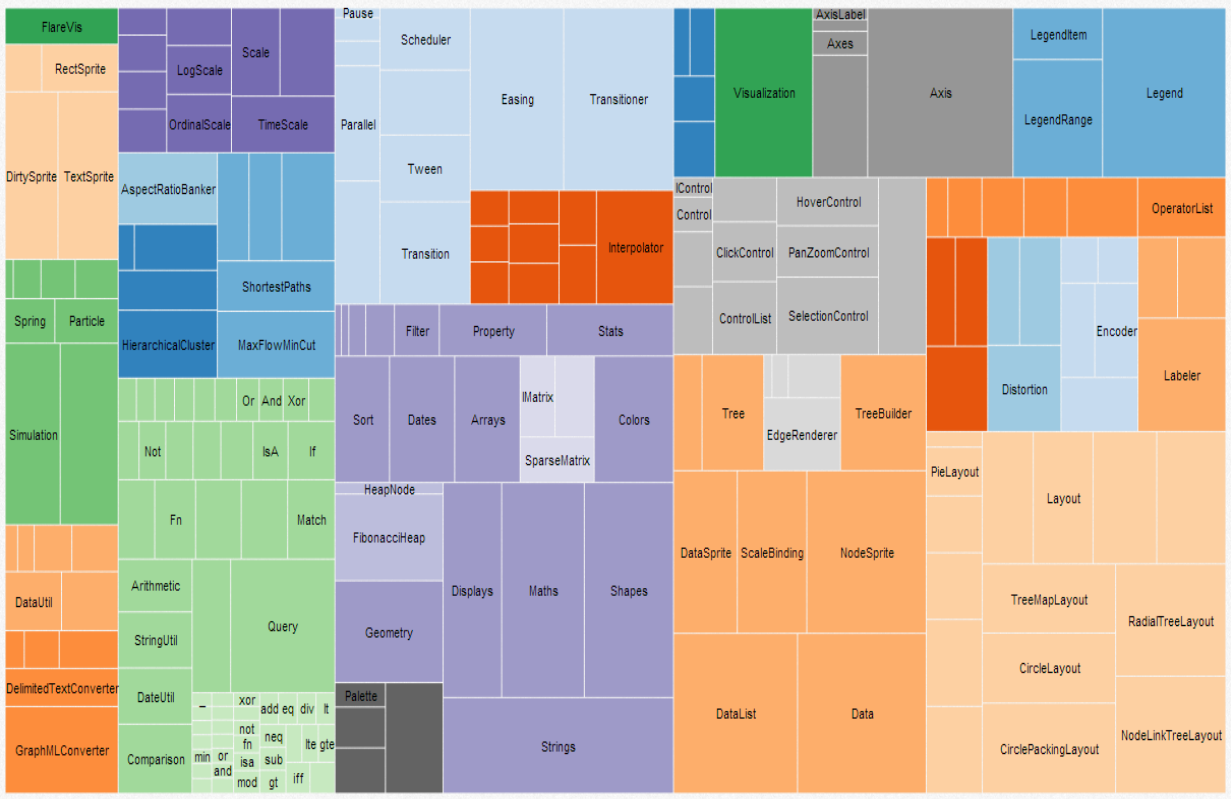
**Solution**

The criteria of all the domain size is the number of code lines of each one. The size of each domain is the sum of the number of code lines of sub-domain. And grouped by domain level can be seen as a unit. If represented divided on the level of domain, the size can be seen at a glance.




<Figure 19 - Size of Component as a circle style>

As you can see from circles grouped by domain, the diagram can represent size of domain level as well as the size of sub-domain in detail is able to be shown though zoom in or zoom out.




<Figure 20 - Size of Component as a Box style>

As above pictures are grouped by color, you can see all the domain size.
The meaning of each color represented one of the sub-systems are grouped at one view, which is about the size of it can be identified. Since zoom in and zoom out can be, the size of sub-domain of each grouped domain is shown in detail.

**Implementation**

Size of Component pattern can be implemented in the following procedure.

- The basic procedure

Step 1: Select the basic tool for the visualization
Step 2: Calculate a metric based on the Base Metric Extractor pattern
Step 3: Set the size of domain based on calculation metric
Step 4: Set the colors properly on the basis of the color assigned to the domain
Step 5: Make a visualization based on the information specified.

- The procedure described

Step 1: Select the basic tool for the visualization.

Select the tool you want to develop for architecture visualization tools to implement graphics or charts. If using pure web development could be the SVG or HTML5 Canvas. The developers want their primary tool for implementing visualization elements can be selected autonomously.

Step 2: Calculate a metric based on the Base Metric Extractor pattern

As described above based on Base Metric Extractor pattern, precede the basic operations for the visualization of the Size of Component composed with Count Metric includes ELOC represents estimated the size of each domain and other Metrics.

Step 3: Set the size of domain based on calculation metric

Set the size of domain based on one of calculation metric through Base Metric Extractor Pattern. The ELOC has given the size of each domain on the basis of Count Metric.

Step 4: Set the colors properly on the basis of the color assigned to the domain

Finally, set the color of each domain. Although color can be given by several criteria, colors representatively give the following two ways can be taken.

- Give a simple color for grouping

Assign colors to categorize groups between domains. Between each group can be given a different color, so be careful to keep in mind when you want to give colors except that the reference does not exist.

- Give a color for demonstrating the safety of domain

Give color based on selecting element to show the stability of domain in the various Metric extracted by Base Metric Extractor pattern. The color usually assigned as green is a safe, yellow is a warning, or red is a risk.

Step 5: Make a visualization based on the information specified.

Make a visualization based on the size and color information of domain previously obtained. Size of Component can be visualized using a variety of visualization methods of Treemap, Sunburst, partition layout or etc.

**Consequences**

- As you can see the size of all the domains at a glance, you can easily identify domain size can be a problem.
- Since you can determine the size of domain level, what domain very easy to identified that does not have the appropriate size.
- The size of the code simply is not enough information.

**Known Use**

- STAN4J : Express the size of each object by using the Size of Component

**Related Pattern**

- Base Metric Extractor Pattern: Make a visualization using Counter Metric
- Pollution Pattern: By the color, this allow you to easily see the level of pollution in each domain

**7. Robert .C Martin Chart Pattern**

**Context**

When you establish an architecture and work, it is very important how to be abstracted for each subsystem. If you have not abstractions that need to be abstracted as a sequence of simple object, you cannot absorb changes; it is a non-flexible and non-scalable architecture. Vice versa, if you have abstraction that does not need to be abstracted, it is difficult to respond quickly to the changing requirements.
By default, if you use introduced earlier Dependency Chart pattern, although you can understand all the dependencies of the domain belonging to the software system, it is challenge to see this abstraction problem at a glance.

**Problem**

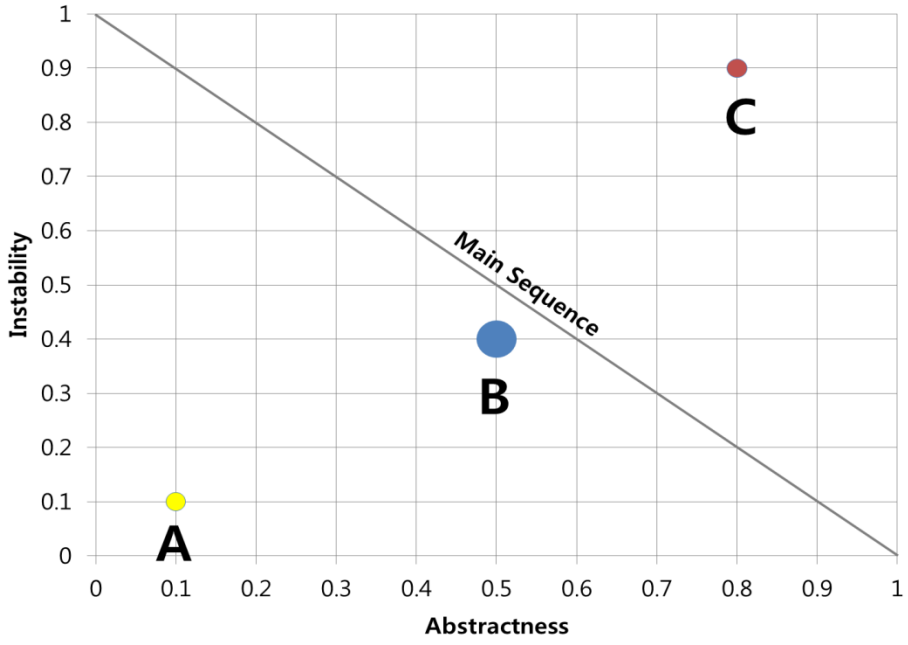
- Difficult to determine whether abstraction of a software system has been properly

Need to show as visualization at one view what the proper package is whether abstraction of the domain at the package level or a package is not properly.

**Forces**

- The specific criteria for the level of abstraction are needed.
- Should show the level of abstraction at a glance.

**Solution**




<Figure 21 - Distance Chart>

Robert C. Martin [Martin 00 Page 26] proposed a solution as above chart. First, the chart displayed domain level represents a higher Package level. It will create a Class Domain gathered a small subsystem is the smallest unit that can build in package domain level because you do not know the level of abstraction of the sub-level domain Class at all.
Instability indicators can see in a domain that how much you can afford to change. In other words, how much referred in other places. The more the value is close to 0, the more referred by the other Package domain. It means you will have less ability to change the package domain. Abstractness represents what including Interface, Abstract Class inside the package domain. Also you can see that the more the value is close to zero, the less abstraction.

<Figure 22 - Afferent Coupling, Efferent Coupling>

In order to obtain the Instability, you should aware of the Efferent Coupling and Afferent Coupling gained from the existing metric pattern. As shown in figure, Ca indicates the Package referred how many times by other packages in other places. While Ce represents the Package refer how many times other packages. Instability has been described as an alterable ability. As the below formula, Afferent Coupling in other worlds Ca values increases close to 0, you can see the properties of the Instability through Ca indicates how many package domain reference in the other domain.

$$\boldsymbol{Instability} = \frac{Ce}{Ca + Ce}$$

<Instability formula >

Abstractness can represent how abstraction has been the formula is as follows:

$$\boldsymbol{Abstractness} = \frac{AbstractClasses}{AbstractClasses + ConcreteClasses}$$

<Abstractness formula >

As you can see above, the ratio of how many of the abstract class or interface in the package domain. The values of Instability and Abstractness know the appropriate abstraction level standard Main Sequence Chart is marked on the diagonal. Main Sequence is the point at which the value of Abstractness and Instability are close to 1. As explained in detail, the substantially referred means you need to increase the level of abstraction, while less referred means you need to decrease the level of abstraction

$$\boldsymbol{Main\ Sequence} = Abstractness + Instability = 1$$

<Main Sequence formula >

Eventually, the Main Sequence represents Abstractness and Instability the sum of the values 1. When package domain is close to line, it can be determine that has made the appropriate abstraction
Finally, the value of Distance represents how far from Main Sequence. The higher Distance, it means the far away from Main sequence. So, Distance value is high means that you need to adjust the level of abstraction of domain.

$$\boldsymbol{Distance} = \frac{|Abstractness + Instability - 1|}{\sqrt{2}}$$

<Distance formula >

For example, in the above picture, A should be abstracted by a lot of reference because it is very low level of abstraction determined by far from the line. B can be seen that proper abstraction through close to line. Lastly, C reference is not a big problem since C is less referred by external.
Therefore, as you make visualization for the level of abstraction of each package domain,
You can determine immediately whether the level of abstraction is low or high at one view.

**Implementation**

In order to implement this pattern, the following process.

- The basic procedure

Step 1: Select the basic tool for the visualization.
Step 2: Calculate a Metric based on Base Metric Extractor pattern.
Step 3: Make a chart with the X-axis and Y-axis.
Step 4: Set the range of chart from 0 to 1.
Step 5: Draw a guide line passing through from (0, 1) to (1, 0).
Step 6: Draw a proper size in the appropriate position on each domain Metric

- The procedure described

Step 1: Select the basic tool for the visualization.
Select the tool you want to develop for architecture visualization tools to implement graphics or charts. If using pure web development could be the SVG or HTML5 Canvas. The developers want their primary tool for implementing visualization elements can be selected autonomously.
Step 2: Calculate a metric based on the Base Metric Extractor pattern
As explained earlier based on Base Metric Extractor pattern, precede the basic operations for the visualization through extracting various Metric introduced by Robert C Martin.
Step 3: Make a chart with the X-axis and Y-axis.
Draw Chart with X-axis and Y-axis to visualize. Abstractness assign to X-axis, Instability assign to Y axis.
Step 4: Set the range of chart from 0 to 1.
Set the values of 0 to 1 to the range value of chart's X-axis and Y-axis since the range of values of Abstractness and Instability are assigned to the X-axis and Y-axis. So, the values of 0 to 1 adjust the X-axis and Y-axis range.

Step 5: Draw a guide line passing through from (0, 1) to (1, 0).

Draw a straight line passing through (1, 0) (0, 1) in the above chart. The meaning of this straight line is the Main Sequence is 1 point, i.e., the appropriate abstraction can say is the point. Can properly say that the domain of abstraction closer to the straight line corresponding

Step 6: Draw a proper size in the appropriate position on each domain Metric

Draw a circle around a fixed position based on the value of Abstractness, Instability of the each package domain. To inform the size of the package domain through radius, determine the radius of circle based on the ELOC levels.

**Consequences**

- You can determine which package domain should be an abstraction at a glance.
- You can see which package domain is too much abstracted at one view.

**Side-effects**

The criteria for determining the level of abstraction is Chart which decided by simply reference. Due to considering only the reference count without architecture or external circumstance, it simply is not enough indicators to determine the level of abstraction at all.

**Known Use**

STAN4J: The architecture visualization tool STAN4J provides a Distance Chart to help architecture understanding based on indicators of Robert C Martin

**Related Pattern**

*Base Metric Extractor Pattern*: Robert C. Metric It is implemented through the values extracted by Martin Metric.

**8. Pollution Chart Pattern**

**Aliases**
Problem Extractor

**Context**

What identify problems during software development will help to improve quality and reduce costs. So, it becomes important to identify the problems in advance. Though the software is huge, and increasingly, it is difficult to identify every problem. Therefore, this extracted metric through the Base Metric Extractor pattern exactly be a certain level is difficult to determine. If you see only the numerical problems simply, it is hard to recognize and difficult to identify problems at a glance.

**Problem**

- Need to identify which level the problem is based on the Base Metric

To measure the quality of code at the huge software system, you need a base point showing a problem in the Base Metric.

- Need to visualize the pollution as figures for people to perceive easier.

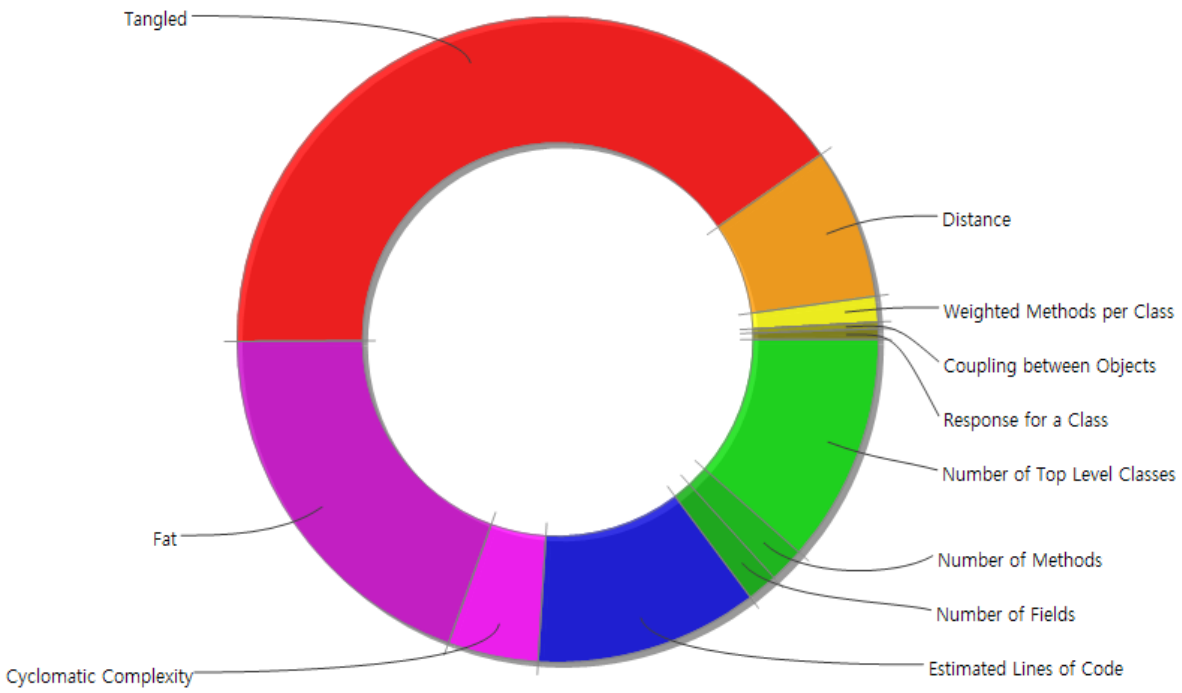
It's able to tell right away what the problem occurs exactly on which point and what happened for people. It leads to recognizing there is a need to visualize and to identify at a glance.

**Forces**

- Base Metric criteria can be a problem.
- Recognize at a glance the problem is, and should be easy to identify.

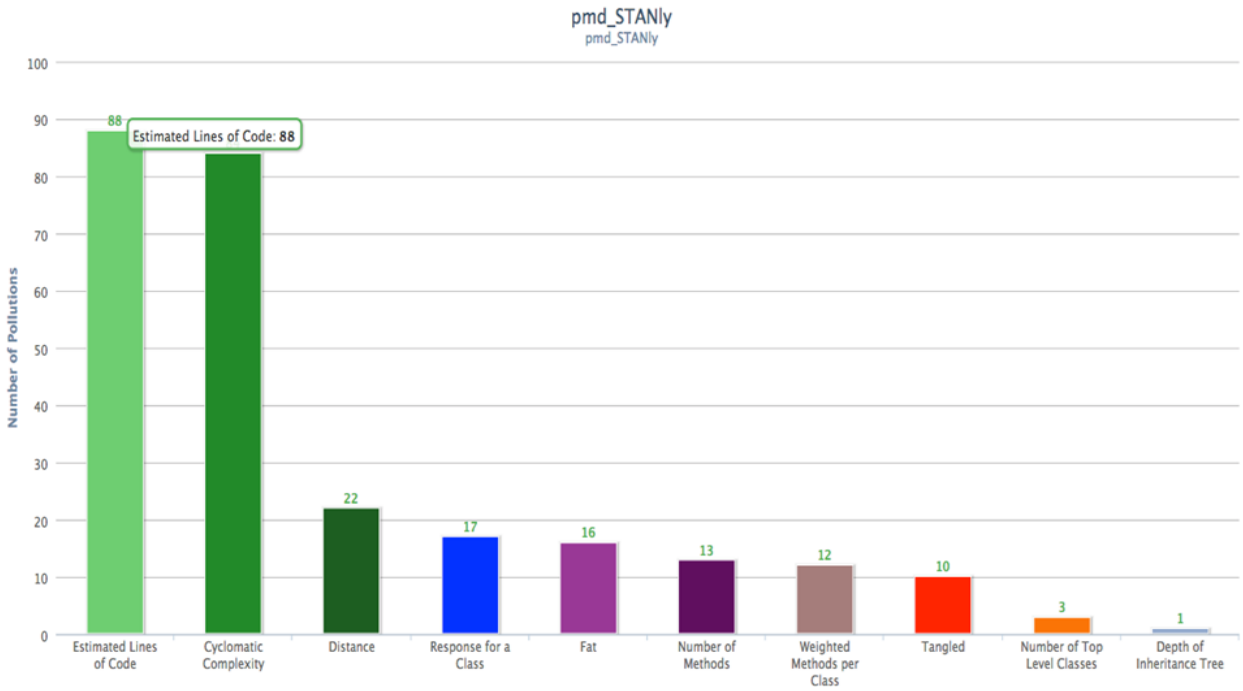
**Solution**

Extract problems in accordance with the Base Metric values obtained through the Base Metric Extractor patterns. In order to extract the problem, the base point is needed for each Metric. If carried over or away from the base point, the problem can be determined. By the characteristics of each project and the inclination of the base metric indicators, it can be changed. Pollution can be determined based on the empirical evidence, the criteria can be set.
In the Base Metric, a criteria lower or higher than the setting, it is determined that the problem everything you need to visualize the collected. Visualization methods, there are a number of ways. The visualization techniques will be different depending on whether you focus on where.



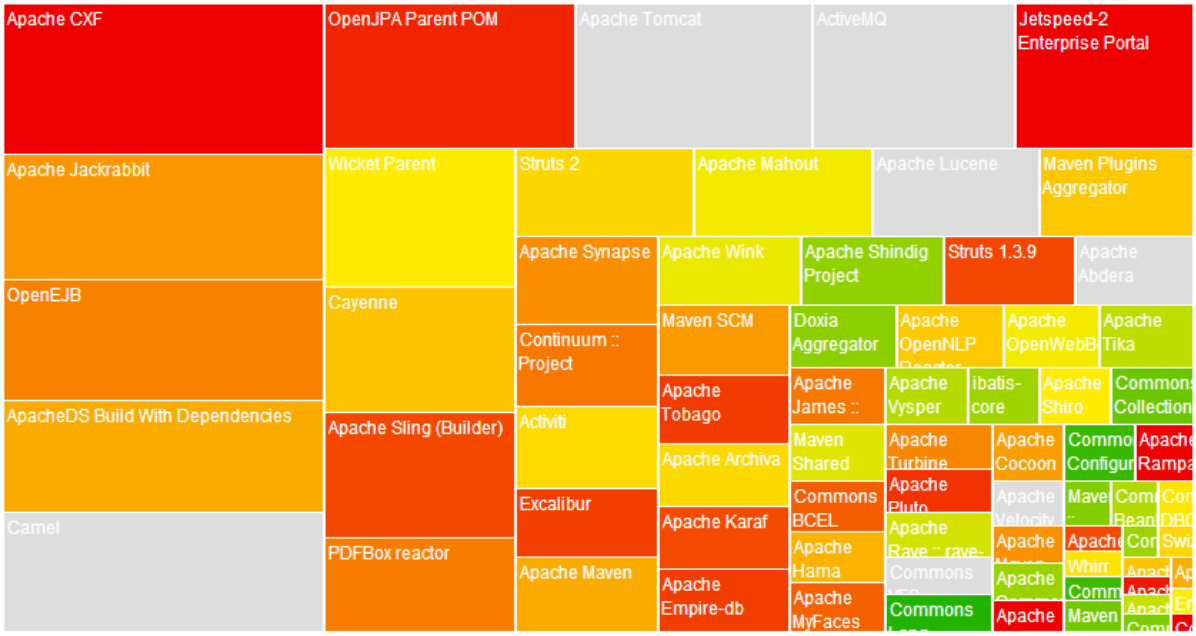

<Figure 23 – Representation of Pollution in the STAN4J >

As a circle chart is represented as a ratio, proportion of problems can be seen at a glance. While, there are disadvantages, exactly how much is unknown whether different levels because they are not represented as exact numbers. As above, in order to be expressed as a ratio, determine the weights depending on the type of problem.




<Figure 24 – Representation of Pollution in the STANly >

As a bar chart is represented pollution, you can see the exact number how the problem is. then identify worst pollution. Unfortunately, there are drawbacks, it is hard to understand that the most important issue not appearing out any problem whether you need to modify the first.




<Figure 25 - Size of Component Chart of the Sonar>

To visualize Pollution, If you apply with the Size of Component pattern has the advantage of being able to identify where greater problem at a glance. As of above chart, the red color indicates the problem.

**Implementation**

In order to implement this pattern, the following process.

- The basic procedure

Step 1: Select the basic tool for the visualization.

Step 2: Set the criteria of Base Metric.

Step 3: Calculate the Metric based on Base Metric Extractor pattern.

Step 4: Check the Metric whether the criteria over

Step 5: Make a visualization of issues exceed the criteria.

- The procedure described

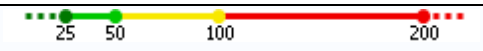
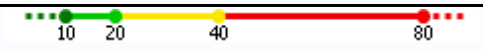
Step 1: Select the basic tool for the visualization.

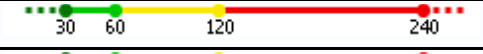
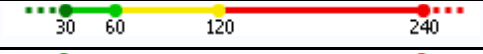
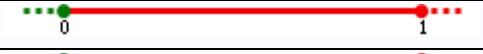
Select the tool you want to develop for architecture visualization tools to implement graphics or charts. If using pure web development could be the SVG or HTML5 Canvas. The developers want their primary tool for implementing visualization elements can be selected autonomously.

Step 2: Set the criteria of Base Metric.

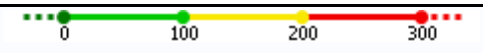
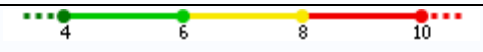
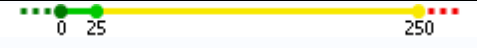
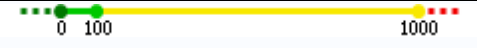
Define the criteria for each of the figures in the Base Metric. Should display the problem is going to exceed a certain level.

| Count Metrics | Rating |
|---|---|
| Number of Top Level Classes (Units) | 20 40 60 80 |

| | |
|---|---|
| Number of Methods |   |
| Number of Fields |   |
| Estimated Lines of Code (ELOC) |   |
| Estimated Lines of Code (ELOC) |   |

| **Complexity Metrics** | **Rating** |
|---|---|
| Cyclomatic Complexity (CC) |   |
| Fat |   |
| Fat |   |
| Fat |   |
| Tangled |   |
| Tangled for Library Dependencies (Tangled - Libraries) |   |
| Tangled for Flat Package Dependencies (Tangled - Packages) |   |
| Average Component Dependency between Libraries (ACD - Library) |   |
| Average Component Dependency between Packages (ACD - Package) |   |

| **Robert C. Martin Metrics** | **Rating** |
|---|---|
| Distance (D) |   |
| Average Absolute Distance |   |

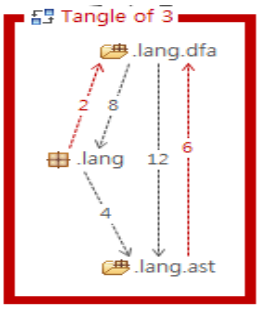
| **Chidamber & Kemerer Metrics** | **Rating** |
|---|---|
| Weighted Methods per Class (WMC) |   |
| Depth of Inheritance Tree (DIT) |   |
| Average Depth of Inheritance Tree |   |
| Coupling between Objects (CBO) |   |
| Response for a Class (RFC) |   |

<Figure 26 – Criteria of Pollution in the STAN4J >

As above table, STAN4J find of pollution criteria. The tendency is different for each project, and the areas of different criteria, depending on whether the emphasis on Metric can be changed. Set by the empirical evidence does not matter.

- Examples of selection criteria

- Tangled - Tangled means that how tightly coupled to each other. As shown under, this is an indicator that occurs when you call the top layer or sub-layer cross-references between each other. There is a need to be very careful because it can spread all the changes in tied between each other when changes occur in one place. So if you found a Tangled this can be seen as a more serious problem than the other problems. If you found Tangled the value being close to zero based on criteria, you should alert that there is a problem.




<Figure 27 Example of Tangled items in the STAN4J >

- The ELOC the indicators to know how many lines of code for each domain level.
- If you find too many code-lines on the inside of domain, you and others can be tough to see the code which has the potential to violate the Single Responsibility Principle (SRP) one of the five principles of object-oriented design. People can recognize at a glance the number of lines for each level of domain to determine the criteria is appropriate. In the method domain level, the appropriate number of lines can be displayed on one screen. You set the unit of the number of method domains can recognize at one time. If it can be exceeded, you need to inform that there is a problem.

Step 3: Calculate a metric based on the Base Metric Extractor pattern

As described above based on Base Metric Extractor pattern, to make sure the problem, obtain which is used Count Metric, Complexity Metric, Robert C Martin, Chidamber and Kemerer metric. And prepare for problems of a software system.

Step 4: Check the Metric whether the criteria over

Based on the criteria set forth in Step 2, the various Metric of software system exceed the criteria, check whether the problem occurs

Step 5: Make a visualization using collecting the issues which have exceed criteria.

When you make visualization, as discussed above, you can visualize the problems utilizing a circle chart, bar chart, and Size of Component.

**Consequences**

- If you set the criteria, you can extract problems automatically.
- If you change the criteria, you can adjust the level of problem which can be seen.
- It is difficult to determine the exact criteria.
- You can identify problems at a glance.

**Known Use**

- Stan4J pollution representation: The Stan4j was able to identify any pollution to show as circle chart form at the each domain level.
- Structure101 pollution representation: Select only certain elements more important than others, which can be a problem was displayed as circle chart form.

**Related Pattern**

- Domain Level Classifier Pattern: It can find easily pollution divided by the area of each domain.
- Base Metric Extractor Pattern:  It is extracted pollution based on the metric figures.
- Size of Component pattern: It is associated with each pollution can be visualize at once is expressed.

**Reference**

**Figures**

[Figure 2] The screen of dependency analysis of JArchitect
[Figure 6] The table of dependency definition in the STAN4J
[Figure 16] The screen of dependency analysis of STANly
[Figure 17] Directional graph for near matrix drawn using D3
[Figure 18] Layered directional graph drawn in the STAN4J
[Figure 19] Example of Zoomable Pack Layout in the D3
[Figure 20] Example of Zoomable TreeMap in the D3
[Figure 23] Representation of Pollution in the STAN4J
[Figure 24] Representation of Pollution in the STANly
[Figure 25] Size of Component Chart of the Sonar
[Figure 26] Criteria of Pollution in the STAN4J
[Figure 27] Example of Tangled items in the STAN4J

**Table**

[Table1,2,3,4] the tables of Metric types defined in the STAN4J

**Web sites**

JArchitect(http://www.jarchitect.com/)
STAN4J(http://stan4j.com/)
D3(http://d3js.org/)
Sonar(http://www.sonarsource.org/)

**Papers**

[Bastert+01] Bastert, O. and Matuszewski, C, Layered drawings of digraphs, In Kaufmann and Wagner (2001)
[Martin 00] Robert C. Martin “Design Principles and Design Patterns” (2000)
[Neeraj 05] Neeraj Sangal, “Using dependency models to manage complex software architecture” (2005)
[Gill+91] Geoffrey K. Gill, Chris F. Kemerer “Cyclomatic complexity density and software maintenance productivity”(1991)
[Jungmayr 02] Stefan Jungmayr “Testability Measurement and Software Dependencies”(2002)
[Hitz+96] Martin Hitz, Behzad Montazeri “Chidamber and Kemerer's metrics suite: a measurement theory perspective”(1996)

Appendix A: Pattern Map of each participated patterns