System Reengineering

A comprehensive Analysis Report

System Reengineering

Several definitions of reengineering: Preparation or improvement to software, usually for increased maintainability, reusability or evolvability.

R. S. Arnold, A Roadmap Guide to Software Reengineering Technology, Software Reengineering, 1994.

The examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.

E. Chikofsky and J. Cross, Reverse Engineering and Design Recovery: A Taxonomy, IEEE Software 7(1) Jan 1990: 13-17.

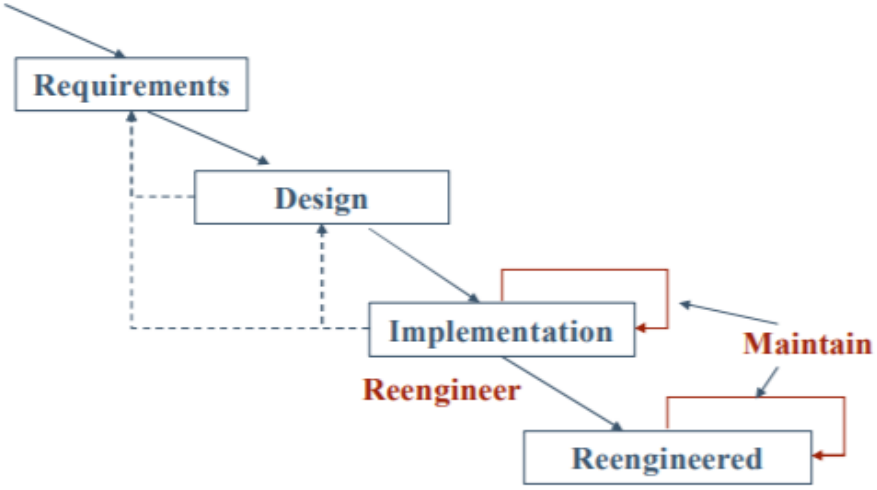
Reengineering is the systematic transformation of an existing system into a new form to realize quality improvements in operation, system capability, functionality, performance, or evolvability at a lower cost, schedule, or risk to the customer.

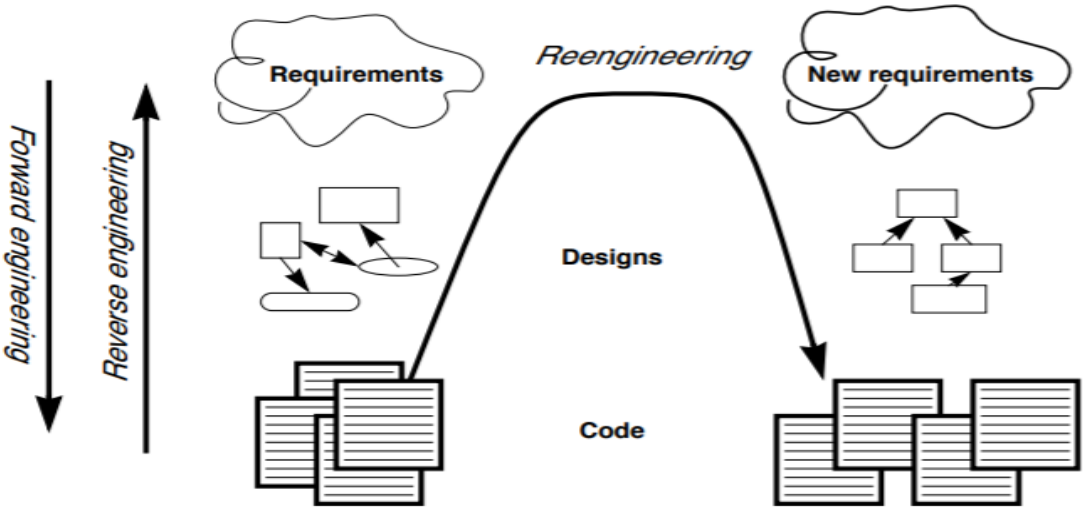
S. Tilley and D. Smith, Perspectives on Legacy System Reengineering, SEI White Paper, 1995

Reengineering patterns codify and record knowledge about modifying legacy software: they help in diagnosing problems and identifying weaknesses which may hinder further development of the system, and they aid in finding solutions which are more appropriate to the new requirements. We see reengineering patterns as stable units of expertise which can be consulted in any reengineering effort: they describe a process without proposing a complete methodology, and they suggest appropriate tools without “selling” a specific one.

Reengineering patterns entail more than code refactorings. A reengineering pattern may describe a process which starts with the detection of the symptoms and ends with the refactoring of the code to arrive at the new solution. Refactoring is only the last stage of this process, and addresses the technical issue of automatically or semi-automatically modifying the code to implement the new solution. Reengineering patterns also include other elements which are not part of refactorings: they emphasize the context of the symptoms, by taking into account the constraints

that reengineers are facing, and include a discussion of the impact of the changes that the refactored solution may introduce





1.Code Inspection:

Code inspections are good at finding defects of maintainability, reliability, and functionality. A code inspection checklist can find simple errors of omission and commission. An omission defect is when functionality is missing from the code that should have been there. A commission defect is when the code performs a function incorrectly. A maintainability defect is one that obviously negatively impacts the service's maintainability. This might be a defect in which a hard-coded value would be better stored in a file and read at runtime. A reliability defect is one that might affect the running of the system. Reliability defects are best found by code inspections. A functionality defect is one in which the behavior of the service does not meet the requirements for the service.

For the case given above, Object Oriented Reengineering Pattern would be ideal for code inspection and understanding. Java being an object-oriented language follows the same constructs and rules. The goal is to assess the state of a software system by means of a brief, but intensive code review.

For the purpose of understanding code functions and methods we should use the following two reengineering pattern clusters:

1.a.1 Detecting Duplicated Code can help you identify locations where code may have been copied and pasted, or merged from different versions of the software. Detecting Duplicated Code consists of two patterns: Compare Code Mechanically, which describes how we can detect duplicated code, and Visualize Code as Dotplots, which shows how duplicated code can be better understood by simple matrix visualization. Once you have detected and understood duplication in the system, you may decide on a variety of tactics. Various refactoring patterns, such as Extract Method, may help you to eliminate the duplication. Duplication may be a sign of

misplaced responsibilities, in which you should may decide to Move Behavior Close to Data. Complex conditional statements are also a form of duplication, and may indicate that multiple clients have to duplicate actions that should belong to the target class. The pattern cluster Transform Conditionals to Polymorphism can help you to resolve these problems.

1.a.2 Redistribute Responsibilities helps you discover and reengineer classes with too many responsibilities. This cluster deals with problems of misplaced responsibilities. The two extreme cases are data containers, classes that are nothing but glorified data structures and have almost no identifiable responsibilities, and god classes, procedural monsters that assume too many responsibilities. Although there are sometimes borderlines cases where data containers and god classes may be tolerated, particularly if they are buried in a stable part of the system which will not change, generally they are a sign of a fragile design.

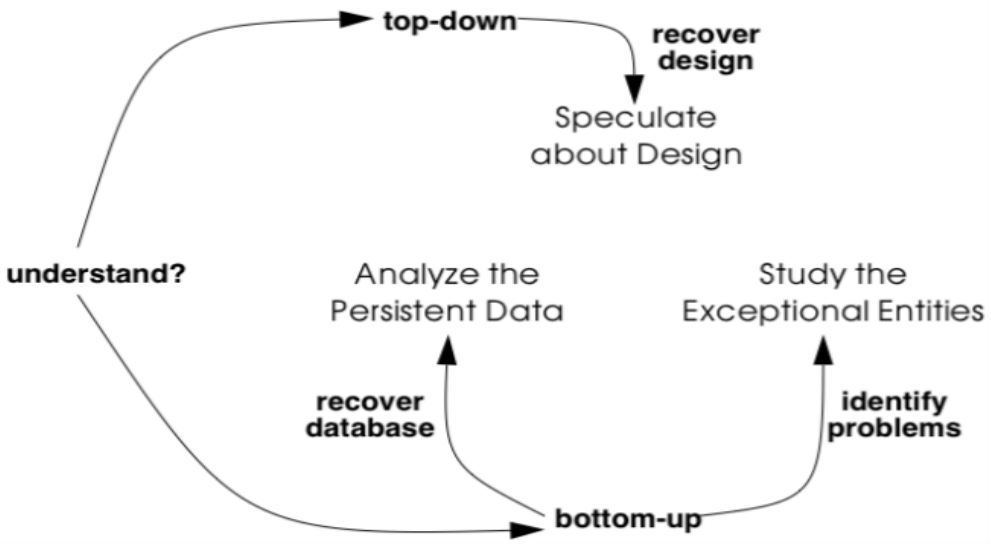
Both these patterns would prove to be ideal to understand the basic coding structure and classes with a Java based legacy system.

1.b Basic Architectural Features:

Each cluster of patterns is presented as a simple "pattern language" — a set of related patterns that may be combined to address a common set of problems. As such, each chapter will typically start with an overview and a map of the patterns in that chapter, suggesting how they may be related. In practice you are more likely to iterate between reverse engineering and reengineering tasks. The main cluster for OORP that can be used to understand the basic architectural features is:

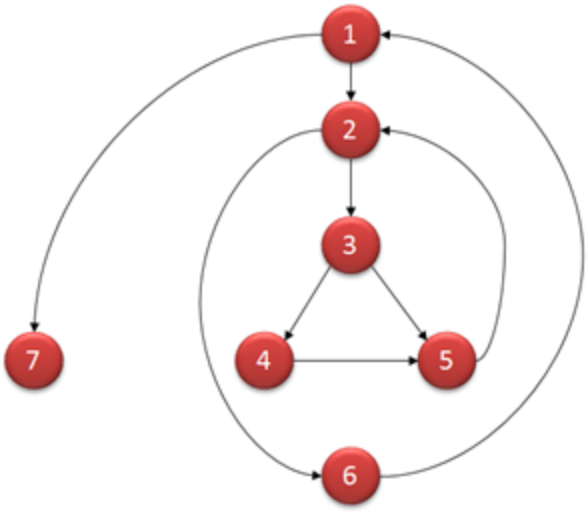
1.b.1Initial Understanding helps you to develop a first simple model of a legacy system, mainly in the form of class diagrams. When developing your initial understanding of a software

system, incorrect information is your biggest concern. Therefore, these patterns rely mainly on source-code because this is the only trustworthy information source. In principle, there are two approaches for studying source-code: one is top-down, the other is bottom-up. In practice, every reverse engineering approach must incorporate a little bit of both, still it is worthwhile to make the distinction. With the top-down approach, you start from a highlevel representation and verify it against the source-code (as for instance described in Speculate about Design). In the bottom-up approach, you start from the source-code, filter out what's relevant and cast the relevant entities into a higher-level representation. This is the approach used in Analyze the Persistent Data and Study the Exceptional Entities.



---

2. Static analysis



2.a Cyclomatic Complexity: Cyclomatic Complexity is a testing metric used for measuring the complexity of a software program. It is a quantitative measure of independent paths in the source code of a software program. Cyclomatic complexity can be calculated by using control flow graphs or with respect to functions, modules, methods or classes within a software program.

The algorithm metric to calculate Cyclomatic Complexity was developed by Thomas J. McCabe in 1976 and it is based on a control flow representation of the program. Cyclomatic complexity is computed using the control flow graph of the program: the nodes of the graph correspond to indivisible groups of commands of a program, and a directed edge connects two nodes if the second command might be executed immediately after the first command. Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program.

The cyclomatic complexity of a section of source code is the number of linearly independent paths within it—where "linearly independent" means that each path has at least one edge that is not in one of the other paths. For instance, if the source code contained no control flow statements (conditionals or decision points), the complexity would be 1, since there would be only a single path through the code. If the code had one single-condition IF statement, there would be two paths through the code: one where the IF statement evaluates to TRUE and another

one where it evaluates to FALSE, so the complexity would be 2. Two nested single-condition IFs, or one IF with two conditions, would produce a complexity of 3.

Mathematically, the cyclomatic complexity of a structured program[a] is defined with reference to the control flow graph of the program, a directed graph containing the basic blocks of the program, with an edge between two basic blocks if control may pass from the first to the second. The complexity M is then defined as

$M = E - N + 2P,$

where

E = the number of edges of the graph.

N = the number of nodes of the graph.

P = the number of connected components.

For a single program (or subroutine or method), P is always equal to 1. So a simpler formula for a single subroutine is

$M = E - N + 2$

Cyclomatic complexity may, however, be applied to several such programs or subprograms at the same time (e.g., to all of the methods in a class), and in these cases P will be equal to the number of programs in question, as each subprogram will appear as a disconnected subset of the graph.

McCabe showed that the cyclomatic complexity of any structured program with only one entry point and one exit point is equal to the number of decision points (i.e., "if" statements or conditional loops) contained in that program plus one. However, this is true only for decision points counted at the lowest, machine-level instructions. Decisions involving compound predicates like those found in high-level languages like IF cond1 AND cond2 THEN ... should be

counted in terms of predicate variables involved, i.e. in this example one should count two decision points, because at machine level it is equivalent to IF cond1 THEN IF cond2 THEN. Cyclomatic complexity may be extended to a program with multiple exit points; in this case it is equal to $\pi - s + 2$, where π is the number of decision points in the program, and s is the number of exit points.

2.b Halstead Complexity:

Halstead complexity measures are software metrics introduced by Maurice Howard Halstead in 1977 as part of his treatise on establishing an empirical science of software development. Halstead made the observation that metrics of the software should reflect the implementation or expression of algorithms in different languages, but be independent of their execution on a specific platform. These metrics are therefore computed statically from the code.

Halstead's goal was to identify measurable properties of software, and the relations between them. This is similar to the identification of measurable properties of matter (like the volume, mass, and pressure of a gas) and the relationships between them (analogous to the gas equation). Thus, his metrics are actually not just complexity metrics.

For a given problem, Let:

- n1 = the number of distinct operators
- n2= the number of distinct operands
- N1= the total number of operators
- N2= the total number of operands

The Halstead Complexity can be calculated using the following formula: D = (n1 / 2) * (N2 / n2). The difficulty measure is related to the difficulty of the program to write or understand, e.g. when doing code review

From these numbers, several measures can be calculated:

- Program vocabulary: n=n1+n2
- Program length: N=N1+N2
- Calculated estimated program length: $\hat{N} = n1\log_2 n1 + n2\log_2 n2$

- Volume: $V = Size * (\log_2 n1+n2) = N * \log_2(n)$
- Effort: $D = (n1 / 2) * (N2 / n2)$
- Intelligence Content: $I = V / D$
- Programming Time**:** $T = E / (f * S)$

---

3. Dynamic Analysis:

3.a Program Dependence Graph:

A program dependence graph (PDG) is a representation, using graph notation, that makes data dependencies and control dependencies explicit. These dependencies are used during dependence analysis in optimizing compilers to make transformations so that multiple cores are used, and parallelism is improved.

The PDG represents a program as a graph in which the nodes are statements and predicate expressions (or operators and operands) and the edges incident to a node represent both the data values on which the node's operations depend and the control conditions on which the execution of the operations depends.

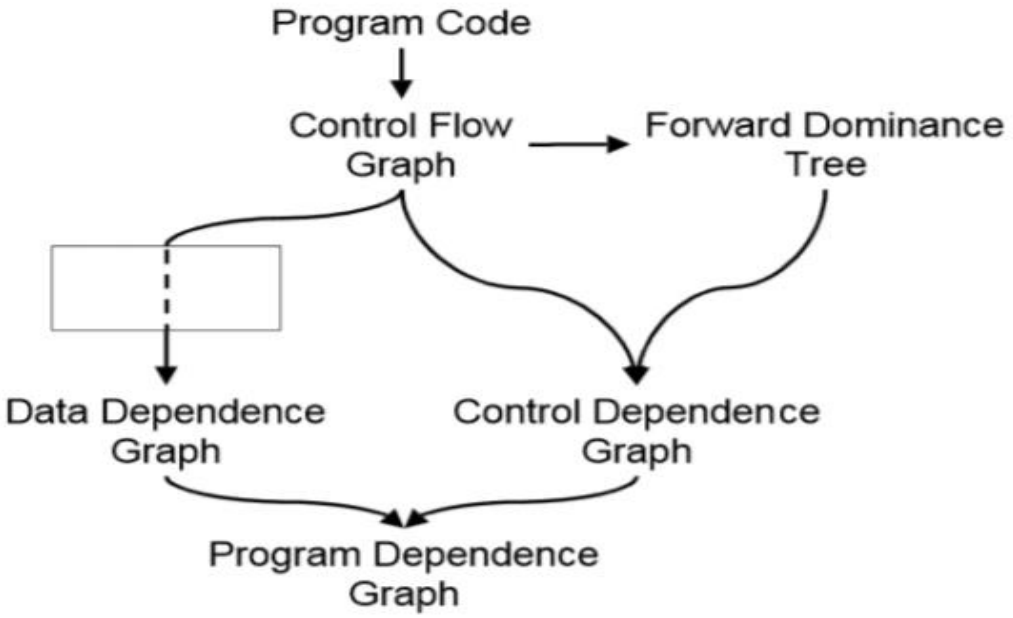
Nodes representing statements and predicates are sufficient for some transformations such as vectorization and simplify our illustrations in this paper. For almost all other optimizing transformations, nodes represent operators and operands. The set of all dependences for a program may be viewed as inducing a partial ordering on the statements and predicates in the program that must be followed to preserve the semantics of the original program. Node: statements Edge: control and data dependence edges Data Dependence + Control Dependence

- **Data Dependence**
  - S2 depends on S1
    - Since variable A, the result of S1, is read in S2

```
S1: A = B * C
S2: D = A * E + 1
```

- **Control Dependence**
  - S2 depends on predicate A
    - Since the value of A determines whether S2 is executed

```
S1: if (A) then
S2:     B = C * D
    endif
```



3.b Backward Slice:

A program slice is a subset of statements that is obtained from the original program, usually by removing zero or more statements. Slicing is often helpful in debugging and program understanding. For instance, it's usually easier to locate the source of a variable on a program slice. A backward slice is constructed from a target in the program, and all data flows in this slice end at the target. Slicing: All the statements of a program that may affect the values of some variables in a set V at some point of interest *p*.

Slicing Criterion: C = (*p* , V)

The backward slice of a program is defined with respect to a program location *l* and a variable x, called the slicing criterion, as all statements of the program that might affect the value of x at *l*, considering all possible executions of the program. Slicing was first developed to facilitate software debugging, but it has subsequently been used for performing diverse tasks such as parallelization, software testing and maintenance, program comprehension, reverse engineering, program integration and differencing, and compiler tuning. Although static slicing has been successfully used in many software engineering applications, slices may be quite imprecise in practice -" slices are bigger than expected and sometimes too big to be useful ". Two possible sources of imprecision are: inclusion of dependencies originated from infeasible paths, and merging abstract states (via join operator) along incoming edges of a control flow merge.

Backward Slicing general approach: backward traversal of program flow

- Slicing starts from point p (C = (p , V))
- Examines statements that could be executed before p (not just statements that appear before p)
- Add statements that affect value of V at p or execution to get to p

- Considers transitive dependencies.

A simple backward slicer for a node would look like this where the slicing criteria is Criterion<10, product>

1. read (n)
2. i := 1
3. sum := 0
4. product := 1
5. while i <= n do
6. sum := sum + i
7. product := product * i
8. i := i + 1
9. write (sum)
10. write (product)

---

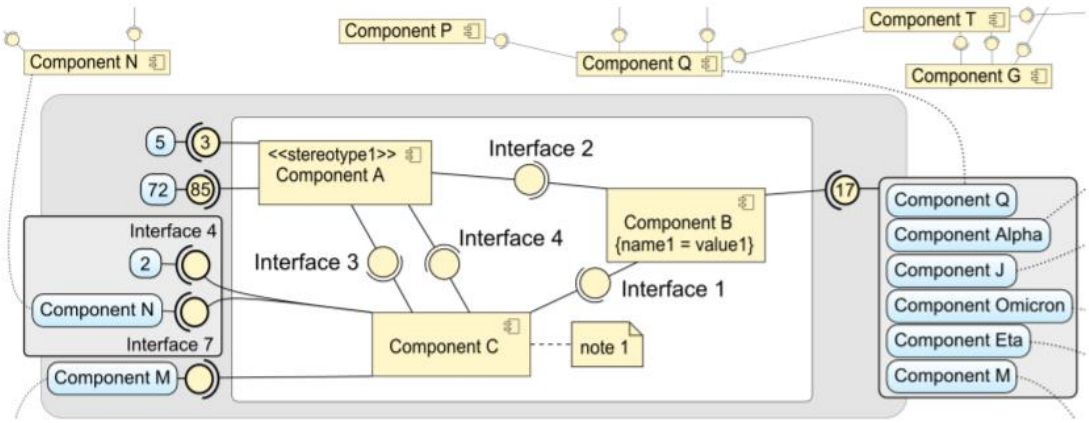
4. System visualization:

Software architects and developers have been using various forms of visualizing the structure of software applications since the advent of the discipline. In the last 20 years, the increased adoption of object-oriented programming lead first to several proposals for adequate modeling notations which were then gradually consolidated into the current standard – the Unified Modeling Language (UML). While UML is able to model both the static and dynamic aspects of many kinds of software, recent development in the field of component-based software engineering (CBSE) brings new challenges.

Visualization is very effective way in understanding software structure, behavior or evolution. When visualizing complex systems, we usually face the problem of not having enough space on the screen to visualize the whole diagram in the desired level of details. Thus, we are forced to use some technique to navigate through such a large diagram while showing only part of it on the screen.

Computer science educators and experts have traditionally used algorithm visualization (AV) software to create graphical representations of algorithms that are later used as visual aids in lectures, or as the basis for interactive labs. Typically, such visualizations are “high fidelity” in the sense that (a) they depict the target algorithm for arbitrary input, and (b) they tend to have the polished look of textbook figures. In contrast, ‘low fidelity’ visualizations illustrate the target algorithm for a few, carefully chosen input data sets, and tend to have a sketched, unpolished appearance.

In order to effectively visualize a Java based legacy system we will think of the components as the clusters. When it comes to basic understanding of an architecture and cluster classes, visualization is a known technique.



The proposed technique shows the graph (standard UML component diagram) zoomed-out to provide the appropriate overview of the complete architecture, with elements displayed without details. Besides that, it shows selected components in detail inside a viewport area plus all their relations with other components in the diagram in an interactive border area. The components here represent different clusters of classes and the interfaces basically show how they interact. UML diagrams have been known to provide a comprehensive understanding of system components. The Unified Modelling Language (UML) has become the standard language for object-oriented analysis and design. Static structure diagrams like class diagrams visualize design aspects, serve as a guideline for the implementation and can be used to document a concrete implementation

The viewport technique should enable to explore and understand the dependencies in large diagrams by showing the context of a selected diagram subset. The clustering shall reduce the visual clutter otherwise caused by large number of relations. The proxy elements should reduce the need for the disorienting pan and zoom otherwise necessary while exploring dependencies and provide user relevant information in one place. The viewport can either be

placed on a given position in the diagram (there can be more viewports in a diagram) or have a fixed position on the screen.

Jinsight is a tool that displays a Java program's behavior at execution. It displays object population, messages, garbage collection, bottlenecks for CPU time and memory, thread interactions, deadlocks, and memory leaks. Jinsight can also take repetitive execution behavior and boil it down to its essentials, eliminating redundancy and uncovering the highlights of an execution. By displaying program behavior and hot spots from several perspectives, Jinsight strengthens your ability to understand, debug, and fine-tune your program. Jinsight advances the analysis of dynamic, object-oriented (OO) programs in a number of ways:

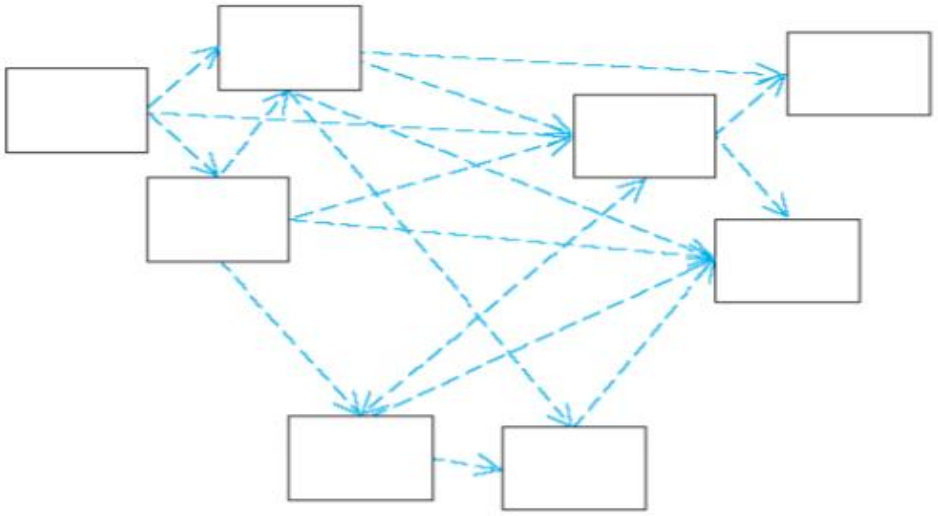
It is fully object-oriented. Most performance-tuning tools for OO languages do little more than profile methods as they do procedures in non-OO languages. Some go as far as showing the total number of objects per class. But the OO programming model is fundamentally different from the procedural model, and much of the power of OO is lost on conventional tools. Jinsight's views of program execution use metaphors that are both natural and consistent with the OO model.

4.b Defect Identification:

Coupling is a phenomenon which occurs when there are interdependencies between one module and another. It can be used as an indication for software quality. In general, the more tightly coupled (high degree of interconnections) a system is, the harder it is to understand and modify the system, because of the interdependencies. A change in one module will probably lead to changes in the other module(s) (because they often share variables or functions). A network of interdependencies makes it hard to see at a glance how a certain component works. One of the major defects that could occur in code and could be identified using the visualized class

diagrams would be two highly coupled classes or clusters with extensive interdependency to eachother.

The defect could be any of the following: 1) Content coupling: where one cluster or class repeatedly changes internal data from another component. 2) Common coupling: the use of global variables in a program 3) Control coupling: function's parameter determines what the function does. This is also referred to as polymorphism.



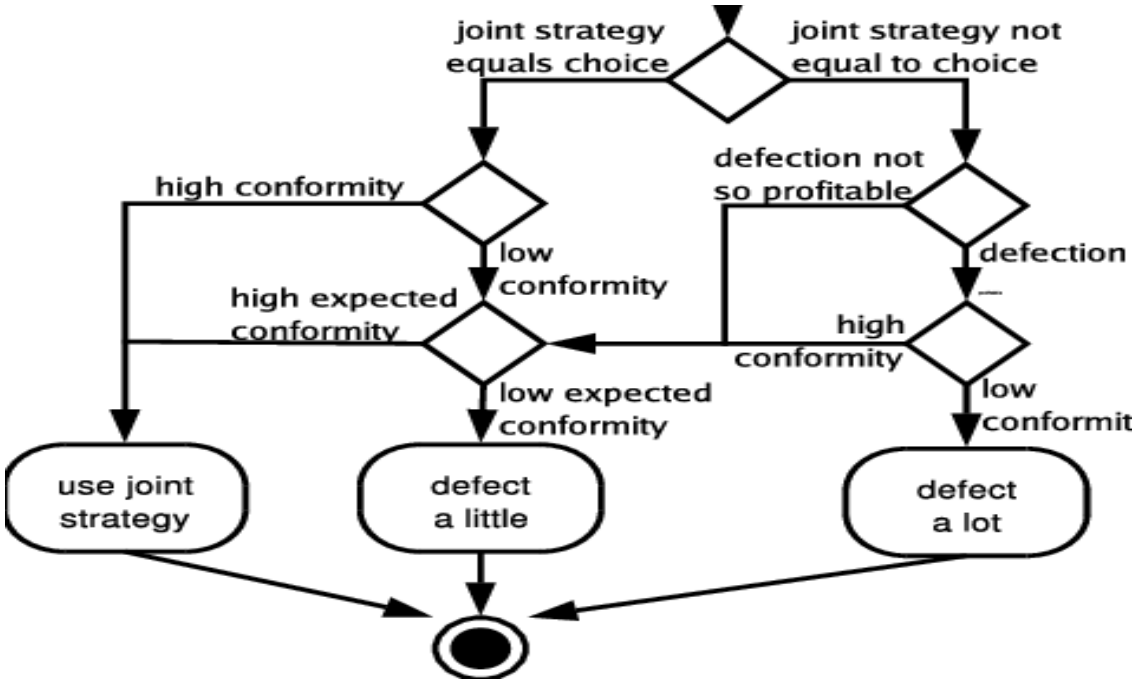
High coupling

The other defect could be the associations made to different class members belonging to different clusters. Accessing members of a class that are either private or protected could raise a probable defect and can be visualized in the UML by visibility symbols. This defect could lead to the obstruction of privacy.

Detecting the backwards dependency is trickier but important given that we find such reversals are among the most common errors. To detect these defects, we assume every use case must involve at least one actor at least indirectly. Since <> dependencies should point from the exception to the normal case, a component B may have apparently no associated actors. This

leads to the conclusion that the <> is backwards. UMLint uses the same logic to check for backwards <> dependencies, requiring each to point towards the service being provided. Complete automation of this check would require extensive domain knowledge.

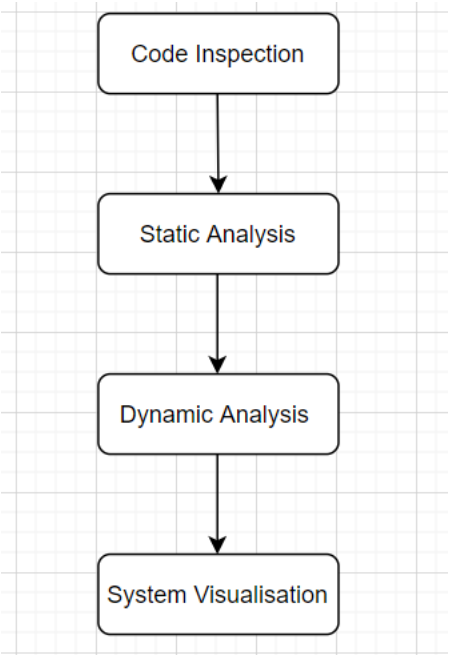
Object generalization and class specification can also be viewed in detail to look for any possible defects that may arise in the system code.



---

5. Reengineering as a Process:

5.a Workflow Diagram:



The reengineering process uses the above techniques in order to have a complete view of the system and be able to reengineer its core features. We start with inspecting the code in detail to statis analysis of the clusters to the dynamic analysis and finally we move to system visualization.

5.b Code Inspection in Reengineering:

Code inspection is the first step of reengineering a legacy system win which we take a very close and detailed loo at the code. The step involves understanding the classes grouped in each cluster and how they communicate with each other. The step also involves detecting any coding or logic defects present in the code and fixing those. Multiple techniques are available to carry out this including the ones mentioned above. The code structure, functions and core methods are highlighted and reviewed resulting in a deeper view into the legacy system and a clearer perspective on how to reengineer it.

5.c Static Analysis in Reengineering: The second step in the process is to perform a static analysis of the system where by the code is examined without being run. This is the step where we can calculate the complexity of each cluster in the system using various algorithms available such as Cyclomatic and Halstead Complexity. In general, Static Analysis addresses weaknesses in source code that might lead to vulnerabilities. Of course, this may also be achieved through manual code reviews. But using automated tools is much more effective. Static analysis is commonly used to comply with coding guidelines — such as MISRA. And it's often used for complying with industry standards — such as ISO 26262. This analysis speeds up the reengineering process, provides depth into the system and accurately point out defects.

5.d Dynamic Analysis in Reengineering: Dynamic analysis is the third step of reengineering. Dynamic program analysis is the analysis of computer software that is performed by executing programs on a real or virtual processor. For dynamic program analysis to be effective, the target program must be executed with sufficient test inputs to cover almost all possible outputs. Also, care must be taken to minimize the effect that instrumentation has on the execution (including temporal properties) of the target program. Dynamic analysis is in contrast to static program analysis. Unit tests, integration tests, system tests and acceptance tests use dynamic testing. Dynamic analysis has the following types: Code coverage, Memory error detection, Fault localization, Invariant inference, Security analysis, Concurrency errors, Program slicing and Performance analysis.

5.e System Visualization is Reengineering: The final step in reengineering process is to visualize the system in order to get a clearer idea of its core features and their interaction. Various techniques can be used to do that with the most common being UML diagrams and Software maps. The objectives of software visualization are to support the understanding of

software systems (i.e., its structure) and algorithms (e.g., by animating the behavior of sorting algorithms) as well as the analysis and exploration of software systems and their anomalies (e.g., by showing classes with high coupling) and their development and evolution. One of the strengths of software visualization is to combine and relate information of software systems that are not inherently linked, for example by projecting code changes onto software execution traces. Software visualization can be used as tool and technique to explore and analyze software system information.

References:

https://rmod.inria.fr/archives/books/OORP.pdf

https://www.geeksforgeeks.org/software-engineering-halsteads-software-metrics/

http://homepages.inf.ed.ac.uk/perdita/OOR.html#GHJV

https://www.cs.cmu.edu/~aldrich/courses/654-sp05/handouts/MSE-Reeng-05.pdf

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.697.9905&rep=rep1&type=pdf

https://tomassetti.me/recognize-patterns-in-code/

http://web.cs.iastate.edu/~weile/cs513x/5.DependencySlicing.pdf

https://arxiv.org/ftp/arxiv/papers/1112/1112.4016.pdf

http://web.cs.iastate.edu/~weile/cs513x/5.SlicingAsDataflowProblem.pdf

https://www.dagstuhl.de/Reports/01/01211.pdf

https://www.researchgate.net/publication/265272931_Dynamic_and_Static_Approachesfor_GlyphBased_Visualization_of_Software_Metrics/figures?lo=1&utm_source=google&utm_medium=organic

---