Program Comprehension

Martin Ward

Reader in Software Engineering

martin@gkc.org.uk

Software Technology Research Lab
De Montfort University

Reverse Engineering Definition

The process of analysing a subject system:

1. *to identify the system's components and their interrelationships and*
2. *create representations of the system in another form or at a higher level of abstraction*

— Chikofsky and Cross, "Reverse Engineering and Design Recovery: A Taxonomy", IEEE Software, January 1990

Program Comprehension

- **Program Comprehension** is the process of developing mental models of a software systems intended architecture, meaning, and behavior
- During maintenance and evolution, software engineers spend 60-90% of their time on program understanding
- Programmers have to become part historian, part detective, and part clairvoyant

Goals of Program Comprehension

Program Comprehension aims to recover high-level information about a system including:

Goals of Program Comprehension

Program Comprehension aims to recover high-level information about a system including:

- Its structure (components and their interrelationships)

Goals of Program Comprehension

Program Comprehension aims to recover high-level information about a system including:

- Its structure (components and their interrelationships)
- Its functionality (what operations are performed on what components)

Goals of Program Comprehension

Program Comprehension aims to recover high-level information about a system including:

- Its structure (components and their interrelationships)
- Its functionality (what operations are performed on what components)
- Its dynamic behavior (how input is transformed to output)

Goals of Program Comprehension

Program Comprehension aims to recover high-level information about a system including:

- Its structure (components and their interrelationships)
- Its functionality (what operations are performed on what components)
- Its dynamic behavior (how input is transformed to output)
- Its rationale (how was the design process and what decisions have been taken)

Goals of Program Comprehension

Program Comprehension aims to recover high-level information about a system including:

- Its structure (components and their interrelationships)
- Its functionality (what operations are performed on what components)
- Its dynamic behavior (how input is transformed to output)
- Its rationale (how was the design process and what decisions have been taken)
- Its construction, modules, documentation, and test suites

Goals of Program Comprehension

Program Comprehension aims to recover high-level information about a system including:

- Its structure (components and their interrelationships)
- Its functionality (what operations are performed on what components)
- Its dynamic behavior (how input is transformed to output)
- Its rationale (how was the design process and what decisions have been taken)
- Its construction, modules, documentation, and test suites

Program Comprehension does not change the subject system, nor create a new system. It is the process of examining and understanding the object system.

Static and Dynamic Software Models

A program can be represented by static and dynamic models:

Static and Dynamic Software Models

A program can be represented by static and dynamic models:

The **static model** represents the general relationship between several components of the software while the **dynamic model** shows the actual interplay between components during execution.

Static and Dynamic Software Models

A program can be represented by static and dynamic models:

The **static model** represents the general relationship between several components of the software while the **dynamic model** shows the actual interplay between components during execution.

Static information is extracted directly from the source code. This information can be visualised by well known diagrams like call graph, flow chart, dependency graph, class diagram, etc.

Static and Dynamic Software Models

A program can be represented by static and dynamic models:

The **static model** represents the general relationship between several components of the software while the **dynamic model** shows the actual interplay between components during execution.

Static information is extracted directly from the source code. This information can be visualised by well known diagrams like call graph, flow chart, dependency graph, class diagram, etc.

Dynamic information can be gathered by running the target software under a debugger. The visualisation here is more difficult since the amount of extracted information is huge and the important information must be isolated.

Static and Dynamic Software Models

- A static model is any collection of information which can be discovered by examining the source code of the system and related documents.

Static and Dynamic Software Models

- A static model is any collection of information which can be discovered by examining the source code of the system and related documents.
- A dynamic model is any collection of information which can be discovered by executing the software and examining input data, output data, execution traces and any other data produced by the program execution.

Static and Dynamic Software Models

- A static model is any collection of information which can be discovered by examining the source code of the system and related documents.
- A dynamic model is any collection of information which can be discovered by executing the software and examining input data, output data, execution traces and any other data produced by the program execution.
- Static models are usually valid for all possible executions of the program.

Static and Dynamic Software Models

- A static model is any collection of information which can be discovered by examining the source code of the system and related documents.
- A dynamic model is any collection of information which can be discovered by executing the software and examining input data, output data, execution traces and any other data produced by the program execution.
- Static models are usually valid for all possible executions of the program.
- Dynamic models may only be valid for the particular input, or set of inputs, which were used to generate the data.

Abstracting Software Models

Using formal methods an extracted software model can be abstracted into a formal system.

Abstracting Software Models

Using formal methods an extracted software model can be abstracted into a formal system.

Formal Methods are mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems

Abstracting Software Models

Using formal methods an extracted software model can be abstracted into a formal system.

Formal Methods are mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems

Formal Methods can produce specifications which might be used:

Abstracting Software Models

Using formal methods an extracted software model can be abstracted into a formal system.

Formal Methods are mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems

Formal Methods can produce specifications which might be used:

- To produce a precise system documentation as the basis for a conventional system development.

Abstracting Software Models

Using formal methods an extracted software model can be abstracted into a formal system.

Formal Methods are mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems

Formal Methods can produce specifications which might be used:

- To produce a precise system documentation as the basis for a conventional system development.
- To verify the correctness of the system.

Abstracting Software Models

Using formal methods an extracted software model can be abstracted into a formal system.

Formal Methods are mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems

Formal Methods can produce specifications which might be used:

- To produce a precise system documentation as the basis for a conventional system development.
- To verify the correctness of the system.
- To derive a new system through correctness preserving refinement rules. Such a system has a high degree of certainty and trustworthiness.

Issues of Program Comprehension

Some things have to be considered before carrying out program comprehension:

Issues of Program Comprehension

Some things have to be considered before carrying out program comprehension:

- The code of a legacy system is in many cases very specific and not generic

Issues of Program Comprehension

Some things have to be considered before carrying out program comprehension:

- The code of a legacy system is in many cases very specific and not generic
- The code may include errors

Issues of Program Comprehension

Some things have to be considered before carrying out program comprehension:

- The code of a legacy system is in many cases very specific and not generic
- The code may include errors
- The result of the reverse engineering process might include new errors

Issues of Program Comprehension

Some things have to be considered before carrying out program comprehension:

- The code of a legacy system is in many cases very specific and not generic
- The code may include errors
- The result of the reverse engineering process might include new errors
- The process is expensive and a good result can not be guaranteed

Issues of Program Comprehension

Some things have to be considered before carrying out program comprehension:

- The code of a legacy system is in many cases very specific and not generic
- The code may include errors
- The result of the reverse engineering process might include new errors
- The process is expensive and a good result can not be guaranteed
- There are no real standards for reverse engineering

Issues of Program Comprehension

Some things have to be considered before carrying out program comprehension:

- The code of a legacy system is in many cases very specific and not generic
- The code may include errors
- The result of the reverse engineering process might include new errors
- The process is expensive and a good result can not be guaranteed
- There are no real standards for reverse engineering

Formal methods can mitigate some of these problems

Different Kinds of Models

- A **mental model** describes the maintainer's mental representation of the program to be understood

Different Kinds of Models

- A **mental model** describes the maintainer's mental representation of the program to be understood
- The programmer's **knowledge base** is their accumulated knowledge before they attempt to understand the code

Different Kinds of Models

- A **mental model** describes the maintainer's mental representation of the program to be understood
- The programmer's **knowledge base** is their accumulated knowledge before they attempt to understand the code
- A **cognitive model** describes the processes and information structures used to form the mental model

Different Kinds of Models

- A **mental model** describes the maintainer's mental representation of the program to be understood
- The programmer's **knowledge base** is their accumulated knowledge before they attempt to understand the code
- A **cognitive model** describes the processes and information structures used to form the mental model
- The **assimilation process** continuously updates and augments the programmer's mental model.

Program Understanding Theories

- Top-down approach: start with the most abstract problem domain concepts and attempt to map them onto the source code
  - Brooks 83
- Bottom-up approach: focus on understanding the behaviour of small pieces of code and later combining this information into larger abstractions
  - Pennington 87
- Opportunistic approach: the programmer/maintainer switches between Top-down and Bottom-up during the comprehension process. The switching depends on the initial knowledge.
  - Letovsky 86

Top-down Approach

- Tries to reconstruct the mappings from the **problem domain** into the **programming domain** that were made during the development of the system:
  - Programmer creates **assumptions** or **hypotheses** based on both acquired or existing knowledge to arrive at an understanding
  - Hypotheses are checked against the source code to prove their validity
  - **Beacons** are places in the source code that prove or falsify a hypotheses
- An example of a high level hypothesis is: "This program produces invoices." This hypothesis maps the task domain (invoicing), to the programming domain (the program itself).

Bottom-up Approach

- Typically used when unfamiliar with code/application
- Look for recognizable **idioms** within the code
  - E.g. the "swap" idiom

$$t := x;\ x := y;\ y := t;$$

  - E.g. the "accumulation" pattern:

while $F(i)$ **do**
  total := total $+ A[i]$;
  $i := i + 1$ **od**;

- Combine recognized units to understand ever larger sections of code: eg recognise that the "swap" is part of a "sort" process

Opportunistic Approach

- Programmers frequently change between top-down and bottom-up approaches
- E.g. Begin with top-down, gain an overview of the functions of the program
- Then selectively apply bottom-up strategies when nearing code level
- Use the information derived from bottom-up analysis to verify the hypotheses resulting from top-down reading

From Studying Real Programmers

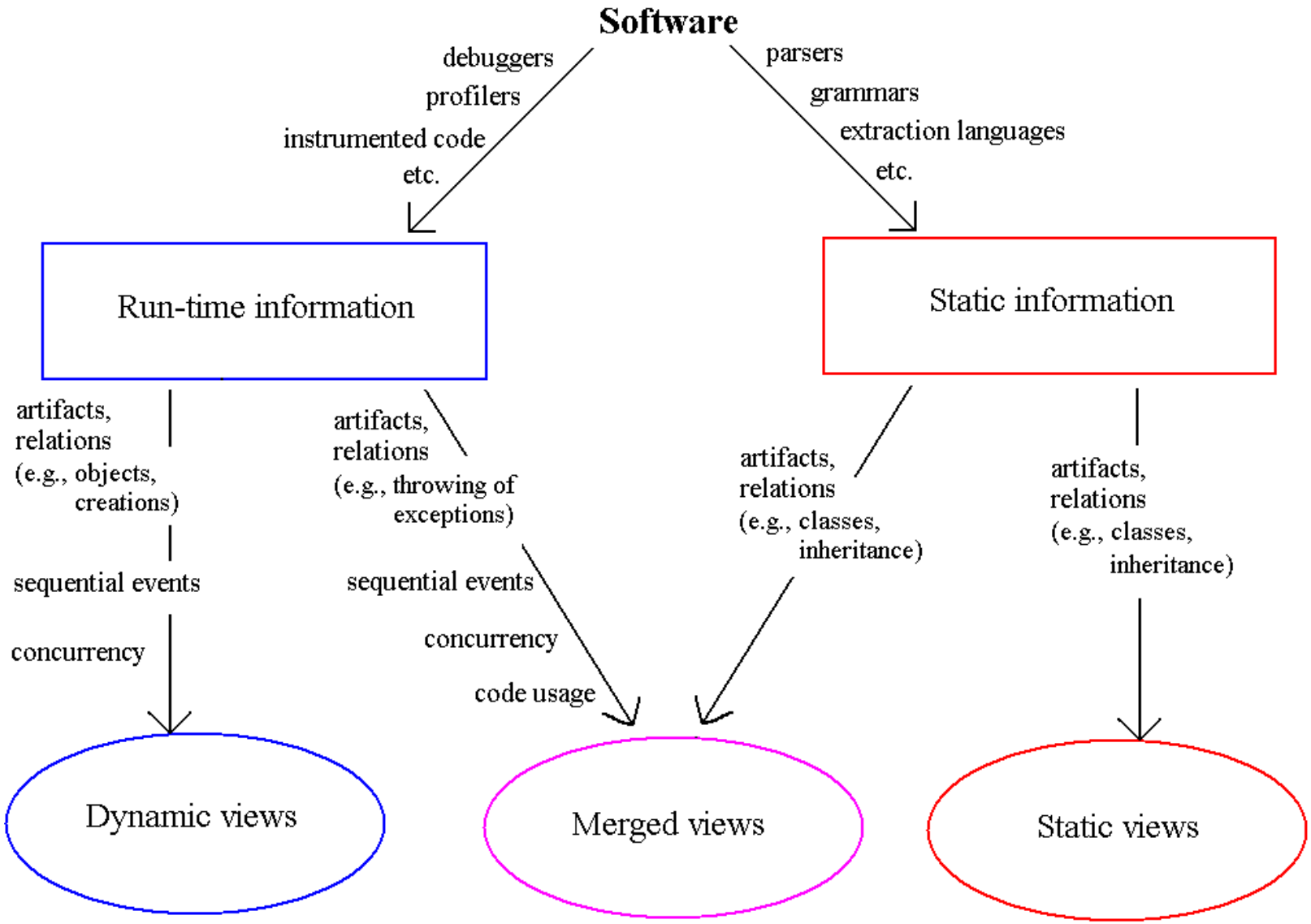
The maintenance programmer needs answer to seven basic questions [Erdos/Sneed]:

1. Where is a particular subroutine or procedure invoked?
2. What are the arguments and results of a particular function?
3. How does the flow of control reach a particular location?
4. Where is a particular variable set, used or queried?
5. Where is a particular variable declared?
6. Where is a particular data object accessed, i.e. created, read, updated, or deleted?
7. What are the inputs and outputs of a particular module?

Program Comprehension and Tools

- Source and binary code is often the only source of information for understanding programs
- *Reverse engineering* describes the extraction of high level design information from code
  - Collecting information:
    - Parsers, debuggers, profilers, event recorders
  - Abstracting information:
    - Making understandable, high level models
  - Navigating information:
    - Tools such as interactive slicers, graph displays, editors

Program Comprehension Overview



Source Code vs. Binaries

- **Source Code**
  - Better form of representation
  - Not always available
  - Result depends on the parser (the parser's view may be different from the compilers!)
- **Binaries**
  - Faster information collection (eg Java byte code)
  - Legality issues
  - Loss of information: variable names, comments, structure

Usage of Binaries

(Reverse engineering, decompilation, disassembly)

- Recovery of lost source code
- Migration of applications to a new hardware platform
- Translation of code written in obsolete languages not supported by current compiler tools
- Determination of the existence of viruses or malicious code in the program
- Recovery of someone else's source code (to determine an algorithm for example)

Static Models

- Finding out the static structure, architecture:
  - Code (using a parser)
  - Documents
  - Interviews
  - Static slicing
- Visualisation:
  - Class diagrams
  - Call graphs
  - Control flow graphs
  - Data flow graphs
  - Program dependence graphs

Dynamic Models

- Finding out the run-time behaviour of software
  - Debugger
  - Profiler
  - Source code instrumentation
  - Execution and Testing: profiling, testing and observing program behaviour
  - Dynamic slicing
- Visualisation:
  - Scenarios (sequence diagrams)
  - State diagrams
  - Hierarchical graphs

Abstracting the Static Model

- Abstracting the high-level components, eg subsystems
- Automatic abstraction:
  - Using the structure of the language
  - Using measurements
- Manual abstraction

Abstracting the Dynamic Model

- Finding behaviour patterns, repeating sequences of events
  - E.g. initialising a dialogue
- Using static abstractions
  - E.g. representing interactions between high-level software elements in sequence diagrams
- Dynamic information may be combined with the high-level static model to produce more detail

Analysing the Static Model

- Syntax, type checking, interfaces
- Control flow and data flow analysis
- Structure analysis
- Static slicing
- Size, complexity and other metrics
- Navigation

Analysing the Dynamic Model

- Dynamic slicing
- Object creation and related dependencies
- Dynamic binding, polymorphism
- Method calls
- Looking for dead code/reachability analysis
- Memory management
- Performance and related problems
- Concurrency

Program Slicing

- **Idea:** when attempting to understand a program we often need to know how variables got their values at specific points

Program Slicing

- **Idea:** when attempting to understand a program we often need to know how variables got their values at specific points
- **Informal Definition:** A program slice is a subset of a program which contains all the statements which can potentially affect the values of certain variables of interest at given positions in the program (E.g. we are interested in the value of variable x on line 232. The sliced program contains everything needed to compute x at that point)

Program Slicing

- **Idea:** when attempting to understand a program we often need to know how variables got their values at specific points
- **Informal Definition:** A program slice is a subset of a program which contains all the statements which can potentially affect the values of certain variables of interest at given positions in the program (E.g. we are interested in the value of variable x on line 232. The sliced program contains everything needed to compute x at that point)
- **More Formal Definition:** A program slice **S** is a *reduced, executable program* obtained from a program **P** by removing statements, such that **S** replicates part of the behaviour of **P** [Weiser 1984]

Program Slicing

- **Idea:** when attempting to understand a program we often need to know how variables got their values at specific points
- **Informal Definition:** A program slice is a subset of a program which contains all the statements which can potentially affect the values of certain variables of interest at given positions in the program (E.g. we are interested in the value of variable x on line 232. The sliced program contains everything needed to compute x at that point)
- **More Formal Definition:** A program slice **S** is a *reduced, executable program* obtained from a program **P** by removing statements, such that **S** replicates part of the behaviour of **P** [Weiser 1984]
- This would now be called an *executable backward static* slice

Classification of Slices

Direction	
Backward	Forward
Source of information	
Static (source code)	Dynamic (runtime)
Type of result	
Executable (correct syntax)	Closure (syntax unimportant)
Procedure calls	
Interprocedural	Intraprocedural
Syntax Preserving	
Syntactic	Semantic

Program Slicing

- Slicing allows one to find *semantically meaningful decompositions* of programs, where the decompositions consist of elements that are not textually contiguous
- Program slicing is a technique for visualising dependencies and restricting attention to just the components of a program relevant to evaluation of certain expressions.

Program Slicing

Classes of slicing techniques:

- Static slicing
- Dynamic slicing
- Amorphous slicing
- Conditioned slicing
- Semantic slicing
- Conditioned Semantic slicing etc. . .

Program Slicing Applications

- Program understanding
- Program comprehension
- Maintenance
- Testing
- Debugging
- Complexity measurement: functional cohesion
- Program integration
- Assist parallelisation
- Comparison of program versions

Slicing Example

```
sum := 0;
prod := 1;
i := 1;
while i ⩽ n do
    sum := sum + A[i];
    prod := prod * A[i];
    i := i + 1 od;
PRINT("sum = ", sum);
PRINT("prod = ", prod)
```

Slice with respect to the variable prod on the last line

Slicing Example

```
sum := 0;
prod := 1;
i := 1;
while i ⩽ n do
    sum := sum + A[i];
    prod := prod * A[i];
    i := i + 1 od;
PRINT("sum = ", sum);
PRINT("prod = ", prod)
```

Slice with respect to the variable prod on the last line

These statements can be deleted

Slicing Example

prod $:= 1;$
$i := 1;$
while $i \leqslant n$ **do**

    prod $:=$ prod $* A[i];$
    $i := i + 1$ **od**;

PRINT(“`prod = `”, prod)

Slice with respect to the variable prod on the last line

The resultant slice

Computing a Slice

Slices can be constructed by tracking **control dependencies** and **data dependencies**.

Control Dependency If the result of a condition at one place can directly affecy whether another statement will subsequently be executed or not, then there is a control dependency. To be precise: if there is one edge from the control node where there is a path to the statement, and another edge where every path to the end node misses the statement, then the statement is control dependent on the condition.

Data Dependency If there is a control flow path from an assignment to a variabe in one place to a reference in another place, with no intervening assignment to the variable, then there is a data dependency.

Slicing Example

For example:

while $p?(i)$ **do**
  if $q?(c)$
    then $x := f$;
      $c := g$ **fi**;
  $i := h(i)$ **od**

Which statements do not contribute to the final value of x?

Slicing Example

while $p?(i)$ **do**
  if $q?(c)$
    then $x := f$;
      $c := g$ **fi**;
  $i := h(i)$ **od**

Some of the control and data dependencies:

$$
\begin{aligned}
x := f &\xrightarrow{\text{ctrl}} q?(c) \\
q?(c) &\xrightarrow{\text{data}} c := g \\
x := f &\xrightarrow{\text{ctrl}} p?(i) \\
q?(i) &\xrightarrow{\text{ctrl}} p?(i) \\
p?(i) &\xrightarrow{\text{data}} i := h(i)
\end{aligned}
$$

It seems that *everything* is needed! ?

Slicing Example

Tracking all data and control dependencies will always produce a *valid* slice, but not necessarily a *minimal* slice.

What is the minimal slice?

while $p?(i)$ **do**
  if $q?(c)$
    then $x := f$;
      $c := g$ **fi**;
  $i := h(i)$ **od**

Slicing Example

Tracking all data and control dependencies will always produce a *valid* slice, but not necessarily a *minimal* slice.

What is the minimal slice?

while $p?(i)$ **do**
  if $q?(c)$
    then $x := f$;
      $\boxed{c := g}$ **fi**;
  $i := h(i)$ **od**

The assignment to c is redundant: once x has been assigned the value f, it does not matter whether it is assigned again, or how many times!

Slicing Example

A *semantic slice* gives a more concise and understandable result:

while $p?(i)$ **do**
  if $q?(c)$
    then $x := f$;
      $c := g$ **fi**;
  $i := h(i)$ **od**

Slicing Example

A *semantic slice* gives a more concise and understandable result:

while $p?(i)$ **do**
  if $q?(c)$
    then $x := f;$
      $c := g$ **fi**;
  $i := h(i)$ **od**

Becomes:

if $p?(i) \wedge q?(c)$ **then** $x := f$ **fi**

Software Architecture Recovery

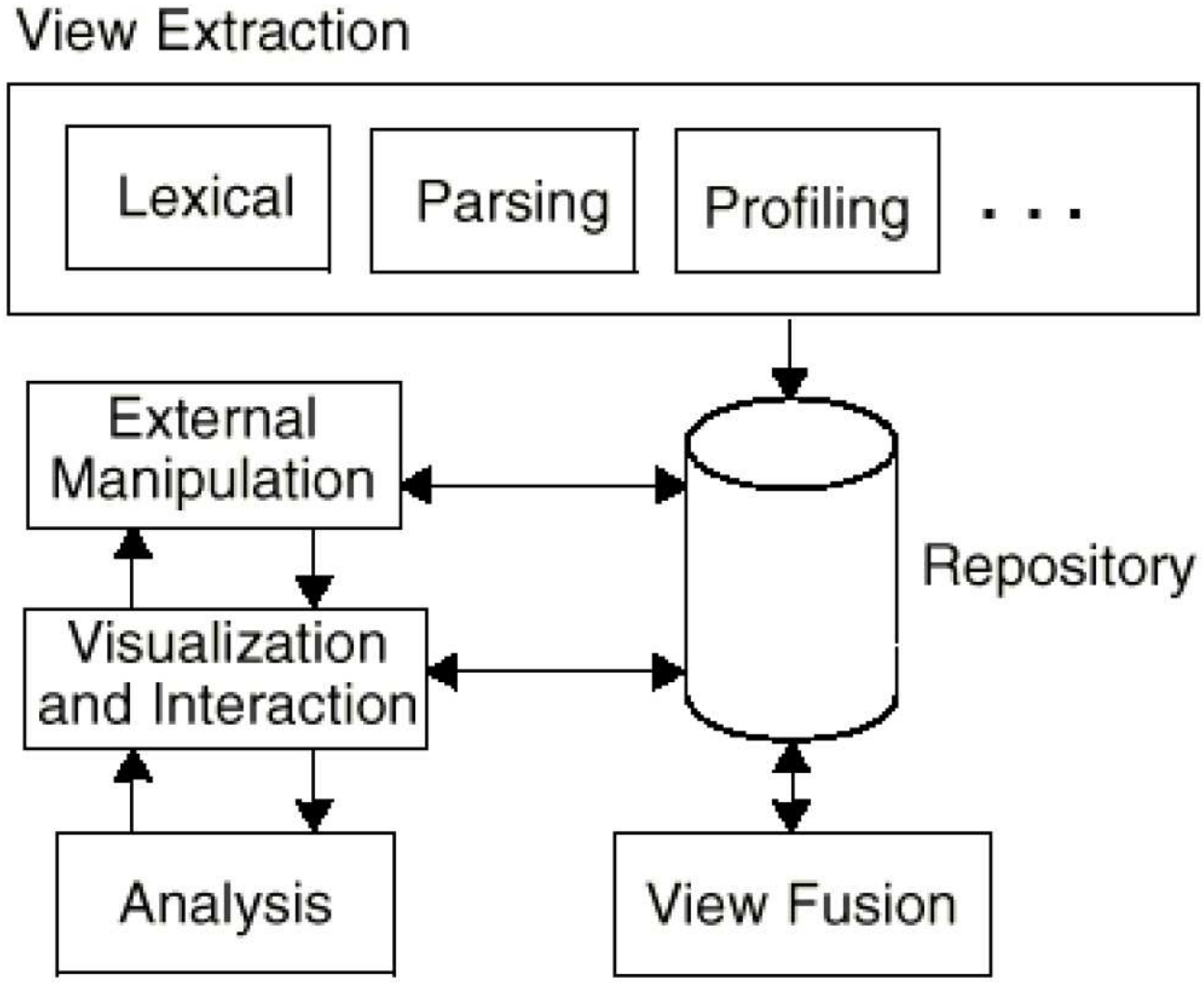
Aims at presenting existing software systems at the more abstract, architectural level. An architecture reconstruction process consists normally of 4 steps:

1. Definition of architecturally significant concepts.
2. Data gathering, in which a model of a system is built in terms of the concepts defined in step 1.
3. Abstraction, in which the model is enriched with (domain specific) abstractions that lead to a higher view of the system.
4. Presentation of the reconstructed architecture in a series of formats, such as graphs, hyperlinks, UML diagrams, and message sequence charts, taking the required architectural view (logical, process, physical, development) into account.

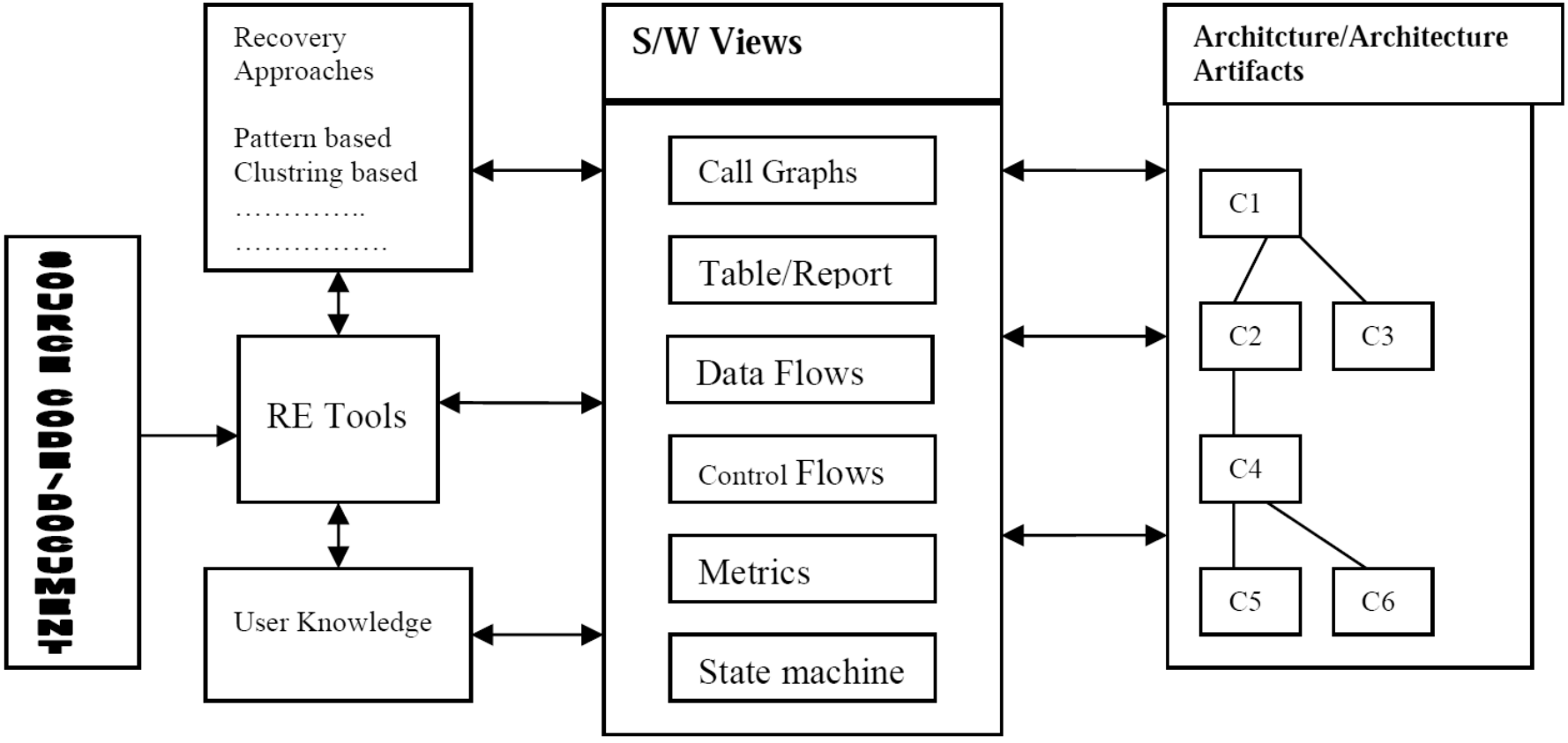
Example Dali Tool

- Semiautomatic technique
- Extracting different views (static, dynamic)
- Putting extracted views to repository
- Combining views to more complex views

Example Dali Tool



Architecture Recovery Framework



Reading by Stepwise Abstraction

This technique was developed by Mills for identifying defects in code documents. During code reading, the reader looks at critical subroutines in the program and determines their function. Once the function is determined then the function, as a behavior, can be used to describe that block of code (abstraction). The reader works through the program hierarchy in this manner assembling abstractions to describe higher level components until the function of the program is determined. This is a bottom-up strategy requiring the understanding of code, and requiring the reader to map the code to suggested problem domain activity.

Basili & Selby (1987) investigated the effectiveness and efficiency of this technique in a professional environment. Their results show that the technique detects more software faults, and has a higher fault detection rate than functional or structural testing.

Active Design Reviews

Parnas and Weiss (1985) suggested a modification of the Fagan inspection process. Reviewers are given a checklist which attempted to focus their attention on particular issues within the document being reviewed. Different reviewers were given different checklists, therefore each reviewer would concentrate on different aspects of the document. Hence, when the review team assembled members of the group brought differing perspectives which were then integrated during the course of the review.

Defect-based Reading

Defect-based reading (Porter95) was developed as a strategy for identifying defects in requirements documents. Defects were categorized. A set of questions was developed for each defect class that would help characterize the class. The questions guides the reader by providing a set of steps (called a scenario) that should be performed during reading. The reader tries to answer the questions presented by the scenario while reading.

Perspective-based Reading

Perspective-based reading (Laitenberger95) is similar to defect-based reading in that different readers are given different tasks. In perspective-based reading readers have different roles—tester, designer and user—that guide the activity. These roles have associated with them operational descriptions (called scenarios). A scenario consists of a set of questions, much like that found in defect-based reading, and activities that guides the reading of the document. Perspective-based reading has been applied to the inspection of requirements documents.

Data Reverse Engineering

Data reverse engineering concentrates on the data aspect of the system that is the organization. It is a collection of methods and tools to help an organization determine the structure, function, and meaning of its data. — Elliot Chikofsky "Data Reverse Engineering: Slaying the Legacy Dragon"

Data reverse engineering (DRE) grew up through the database community and the software engineering community.

Over the years, the research and publications in DRE by both communities has been mainly in three areas :

1. DRE translation and methodologies algorithms
2. DRE tools
3. The DRE of specific applications and experiences in DRE

Data Reverse Engineering

DRE was very prevalent during the Y2K work that was done at the end of the millennium. Currently, DRE is assisting in various areas:

- Analysis of legacy systems
- Evaluation of packages
- Test planning
- Extracting of business rules

Kathi Hogshead Davis and Peter H. Aiken, "Data Reverse Engineering: A Historical Survey" Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE), 2000

Data Reverse Engineering

- For example: relational data bases (RDBs):

$$\text{flat/hierarchical files} \Rightarrow \text{RDBs}$$

$$\text{RDBs} \Rightarrow \text{OO model}$$

Data Reverse Engineering

- **Physical Schema**:
  - Data
  - Schema catalogue
  - Code
  - Documentation
- Analyse to: **Logical Schema**:
  - Domain expert
  - Developer
  - Reengineer
- Abstract to: **Conceptual Schema**:
  - Reengineer

Data Reverse Engineering

Enables:

- Extension
- Migration
- Wrapping
- Integration
- Distribution
- . . .

Reverse Engineering of OO Software

The process of reverse engineering Object-Oriented systems has two main aspects:

- Identifying the object structure

  G. Canfora and A.Cimitile, "An Improved Algorithm for Identifying Objects in Code", Software Practice and Experience 26 (1), 25–48, January 1996

- Identifying design patterns and frameworks which form the architecture.

  Santanu Paul and Atul Prakash, A Framework for Source Code Search Using Program Patterns", IEEE Transactions on Software Engineering 20 (6), 463–474, June 1994

Reverse Engineering of OO Software

- Dynamic behavior may be hard to detect from static model (creating and deleting objects, garbage collection, dynamic binding,...)
  - **This emphasises dynamic modelling**
- Pure object languages support encapsulation (classes, packages,.. .)
  - **Helps in static reverse engineering**
  - **Increases usability of metrics**
- OO paradigm supports the use of design patterns
  - **Reusability of applications (pattern recognition)**