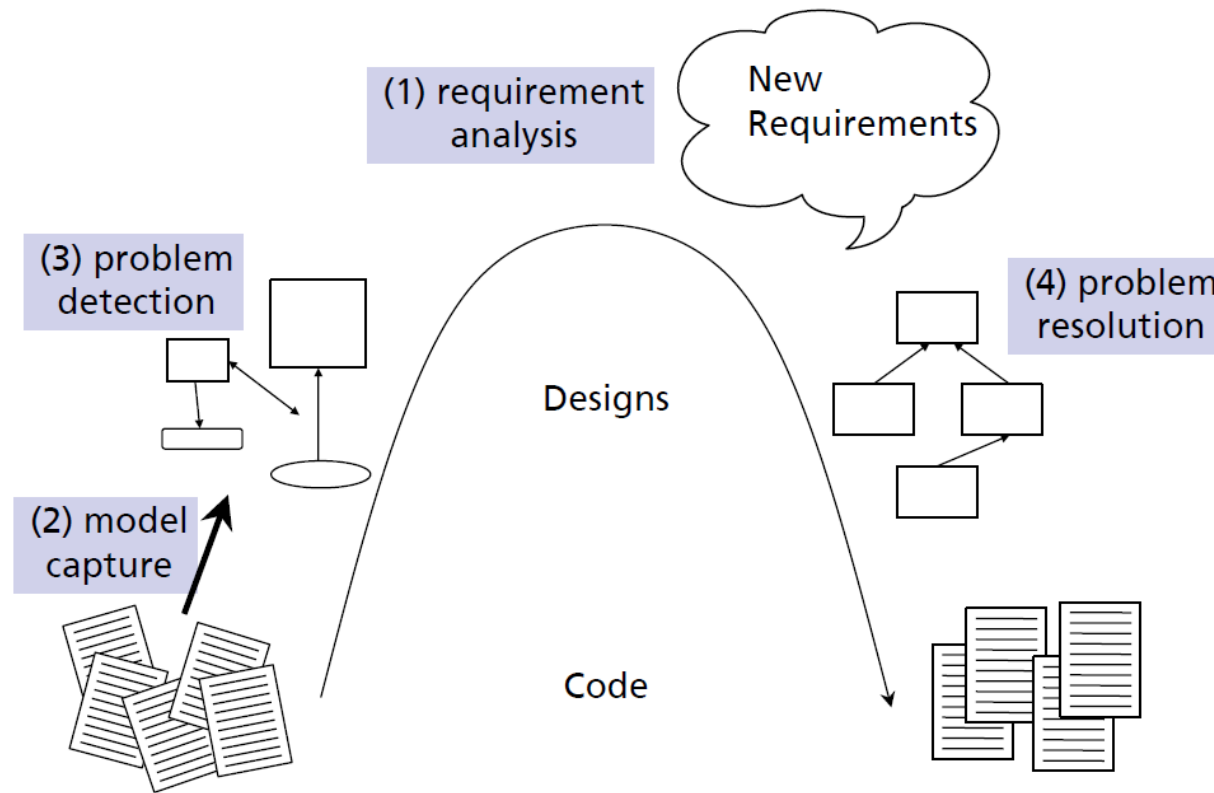




# SOFTWARE RE-ENGINEERING

# RE-ENGINEERING CYCLE






# REENGINEERING

- **Reengineering is a mix of:**
- coarsegrained,
- architectural problems,
- fine-grained,
- design problems.

# THE MOST COMMON FINE-GRAIN PROBLEMS IN OO SOFTWARE :

- Misuse of inheritance: for composition, code reuse rather than polymorphism
- Missing inheritance: duplicated code, and case statements to select behavior
- Misplaced operations: unexploited cohesion—operations outside instead of inside classes
- Violation of encapsulation: explicit type-casting, C++ “friends” .
- Class abuse: lack of cohesion—classes as namespaces

- 
- you will be preparing the code base for the reengineering activity
  - by developing exhaustive test cases for all the parts of the system that you plan to change or replace.
  - Repaired Vs replaced decision.

# REPAIRED VS REPLACED

- According to Chikofsky and Cross:

“Restructuring is the transformation from one representation form to another at the same relative abstraction level, while Preserving the system’s external behavior.”

“Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.”



# WHY PATTERNS?

- A pattern is a recurring motif
- Event
- Structure
- Design
- Document
- Practice
- Language Implementation



# SETTING DIRECTION

- What are the goals of the project?
- Find Go/No-Go decision

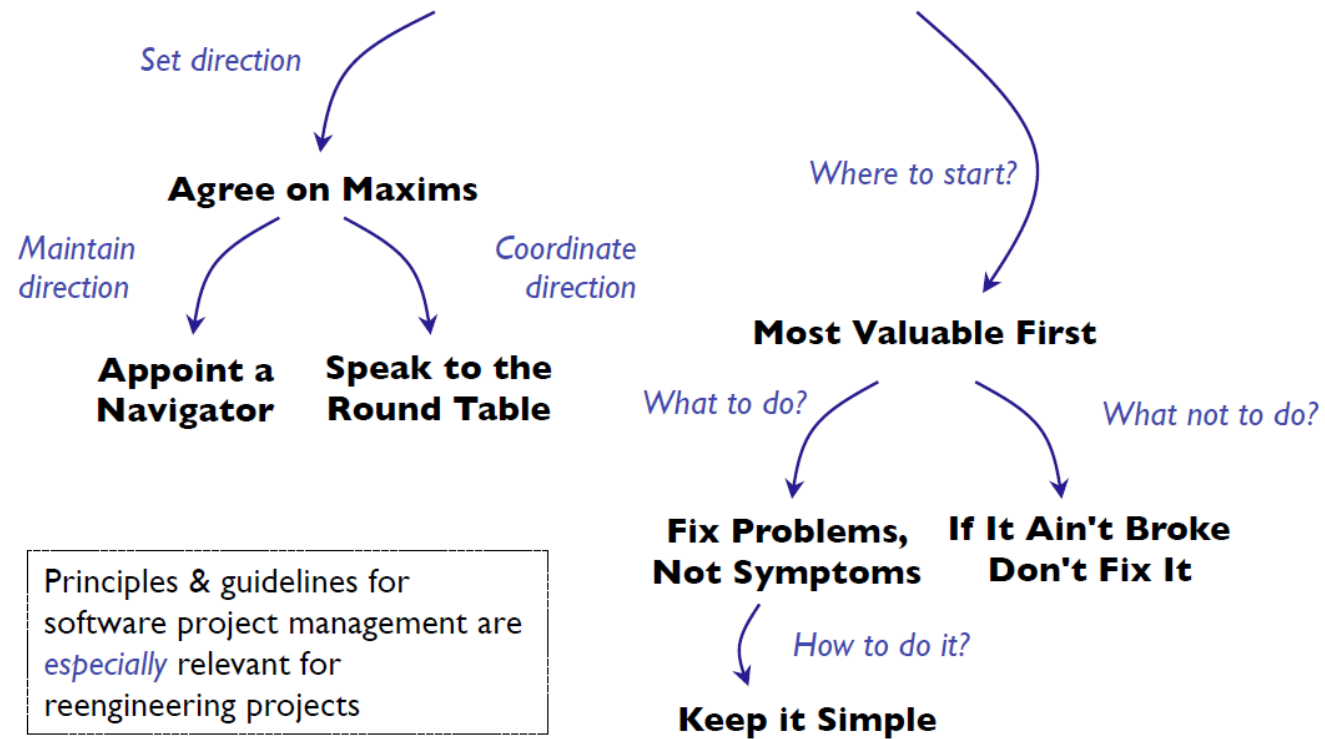




# FIRST CONTACT

- You are facing a system that is completely new to you and within hours/days you should determine:
- Whether the software is still viable
- A plan of work
- A cost-estimation

# SETTING DIRECTION PATTERNS



# MOST VALUABLE FIRST

- Problem: Which problems should you address first?



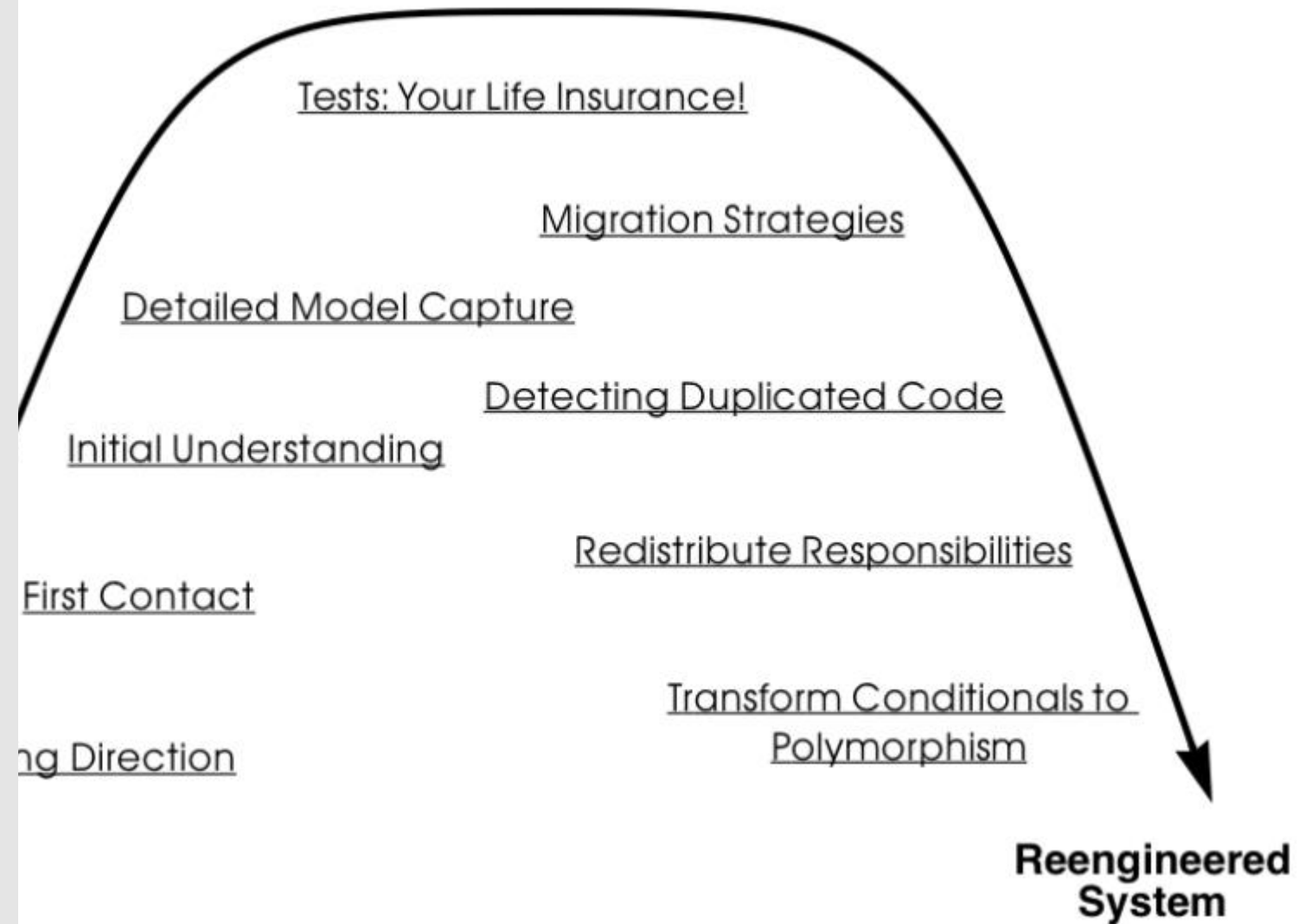


```

while (true) {
    Expression res;
    int c = StreamTokenizer.TT_EOL;
    String varName = null;

    System.out.println("Enter an expression...");
    try {
        while (true) {
            c = st.nextToken();
            if (c == StreamTokenizer.TT_EOF) {
                System.exit(1);
            } else if (c == StreamTokenizer.TT_EOL) {
                continue;
            } else if (c == StreamTokenizer.TT_WORD) {
                if (st.sval.compareTo("dump") == 0) {
                    dumpVariables(variables);
                    continue;
                } else if (st.sval.compareTo("clear") == 0) {
                    variables = new Hashtable();
                    continue;
                } else if (st.sval.compareTo("quit") == 0) {
                    System.exit(0);
                } else if (st.sval.compareTo("exit") == 0) {
                    System.exit(0);
                } else if (st.sval.compareTo("help") == 0) {
                    help();
                    continue;
                }
                varName = st.sval;
                c = st.nextToken();
            }
        }
        break;
    }
    if (c != '=') {
        throw new SyntaxError("missing initial '=' sign.");
    }
}

```



## ***If It Ain't Broke, Don't Fix It***

*Intent: Save your reengineering effort for the parts of the system that will make a difference.*

*The name is usually an action phrase.*

*The intent should capture the essence of the pattern*

### **Problem**

Which parts of a legacy system should you reengineer?

*This problem is difficult because:*

- Legacy software systems can be large and complex.
- Rewriting everything is expensive and risky.

*Yet, solving this problem is feasible because:*

- Reengineering is always driven by some concrete goals.

*The problem is phrased as a simple question. Sometimes the context is explicitly described.*

*Next we discuss the forces! They tell us why the problem is difficult and interesting. We also pinpoint the key to solving the problem.*

### **Solution**

Only fix the parts that are “broken” — that can no longer be adapted to planned changes.

*The solution sometimes includes a recipe of steps to apply the pattern.*

### **Tradeoffs**

**Pros** You don't waste your time fixing things that are not only your critical path.

*Each pattern entails some positive and negative tradeoffs.*

**Cons** Delaying repairs that do not seem critical may cost you more in the long run.

**Difficulties** It can be hard to determine what is “broken”.

*There may follow a realistic example of applying the pattern.*

### **Rationale**

There may well be parts of the legacy system that are ugly, but work well and do not pose any significant maintenance effort. If these components can be isolated and wrapped, it may never be necessary to replace them.

*We explain why the solution makes sense.*

### **Known Uses**

Alan M. Davis discusses this in his book, *201 Principles of Software Development*.

*We list some well documented instances of the pattern.*

### **Related Patterns**

Be sure to Fix Problems, Not Symptoms.

*Related patterns may suggest alternative actions. Other patterns may suggest logical followup action.*

### **What Next**

Consider starting with the Most Valuable First.



# EVOLUTION VS RE-ENGG

- Reengineering implies a single cycle of taking an existing system and generating from it a new system
- Reengineering is done to transform an existing “lesser or simpler” system into a new “better” system



# JACOBSON AND LINDSTORM EQUATION

**Reengineering = Reverse engineering + $\Delta$ + Forward engineering**


# JACOBSON EQUATION

- **Reverse engineering** is the activity of defining a more abstract, and easier to understand, representation of the system
- The core of reverse engineering is the **process of examination, not a process of change**, therefore it does not involve changing the software under examination.
- The third element “**forward engineering**,” is the traditional process of moving from **high-level abstraction and logical**, implementation-independent designs to the physical implementation of the system.
- The second element  $\Delta$  captures alteration that is change of the system.



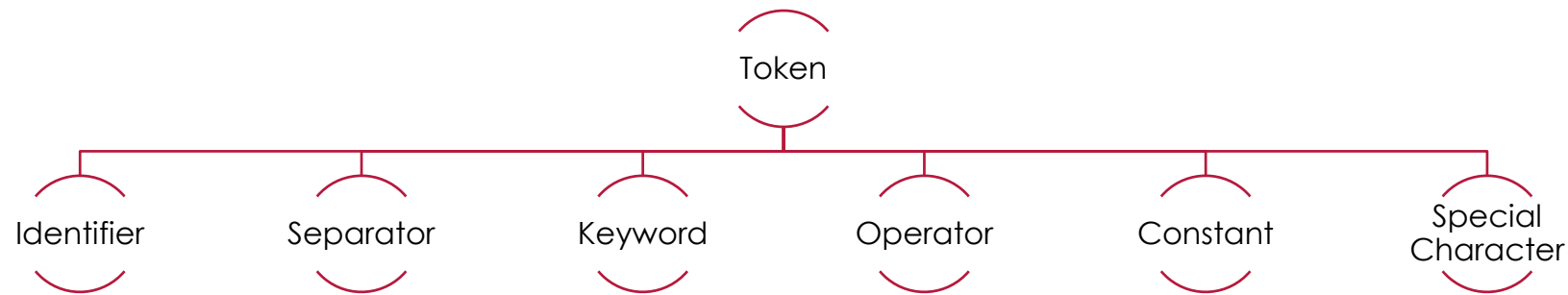
# BENEDUSI REPEATABLE PARADIGM

- **Goals.** In this phase, one analyzes the motivations for setting up the process to identify the **information needs** of the process and the **abstractions to be produced**.
- **Models.** In this phase, one analyzes the abstractions to construct **representation models** that capture the information needed for their production.
- **Tools.** In this phase, software tools are defined, acquired, enhanced, integrated, or constructed to: (i) execute the Models phase and (ii) transform the program models into the abstractions identified in the Goals phase.

- 
- In order to extract information that is not explicitly available in source code, automated analysis techniques,
  - such as lexical analysis,
  - syntactic analysis,
  - control flow analysis,
  - data flow analysis, and
  - program slicing are used to facilitate reverse engineering.

# LEXICAL ANALYSIS

- *Process of converting a sequence of characters in a source code file into a sequence of tokens*
- *processed by a compiler or interpreter*
- *followed by syntax analysis and semantic analysis.*



```
while (true) {
    Expression res;
    int c = StreamTokenizer.TT_EOL;
    String varName = null;

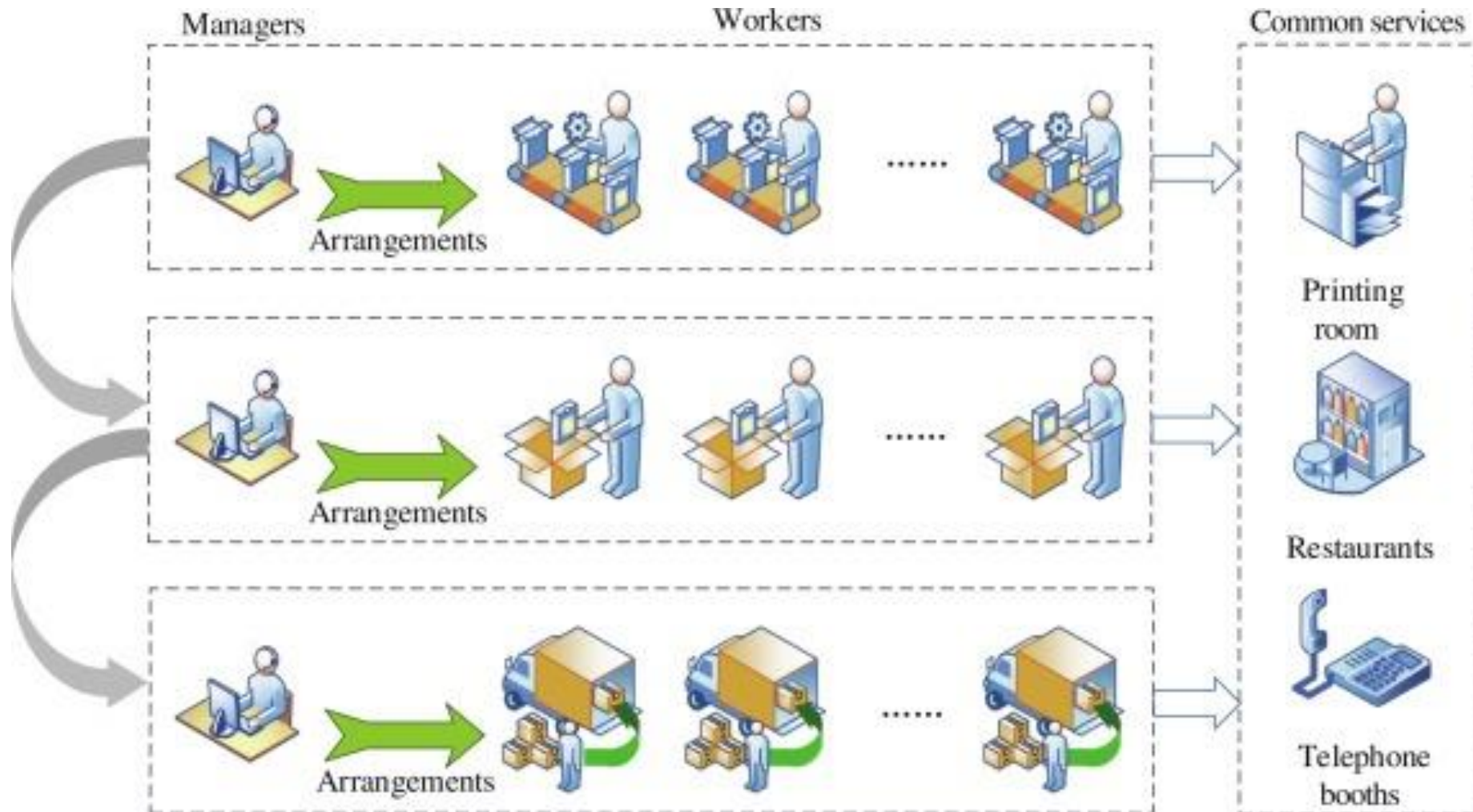
    System.out.println("Enter an expression...");
    try {
        while (true) {
            c = st.nextToken();
            if (c == StreamTokenizer.TT_EOF) {
                System.exit(1);
            } else if (c == StreamTokenizer.TT_EOL) {
                continue;
            } else if (c == StreamTokenizer.TT_WORD) {
                if (st.sval.compareTo("dump") == 0) {
                    dumpVariables(variables);
                    continue;
                } else if (st.sval.compareTo("clear") == 0) {
                    variables = new Hashtable();
                    continue;
                } else if (st.sval.compareTo("quit") == 0) {
                    System.exit(0);
                } else if (st.sval.compareTo("exit") == 0) {
                    System.exit(0);
                } else if (st.sval.compareTo("help") == 0) {
                    help();
                    continue;
                }
            }
            varName = st.sval;
            c = st.nextToken();
        }
        break;
    }
    if (c != '=') {
        throw new SyntaxError("missing initial '=' sign.");
    }
}
```

# PROGRAM SLICING

- Take a piece of code and analysis the behavior

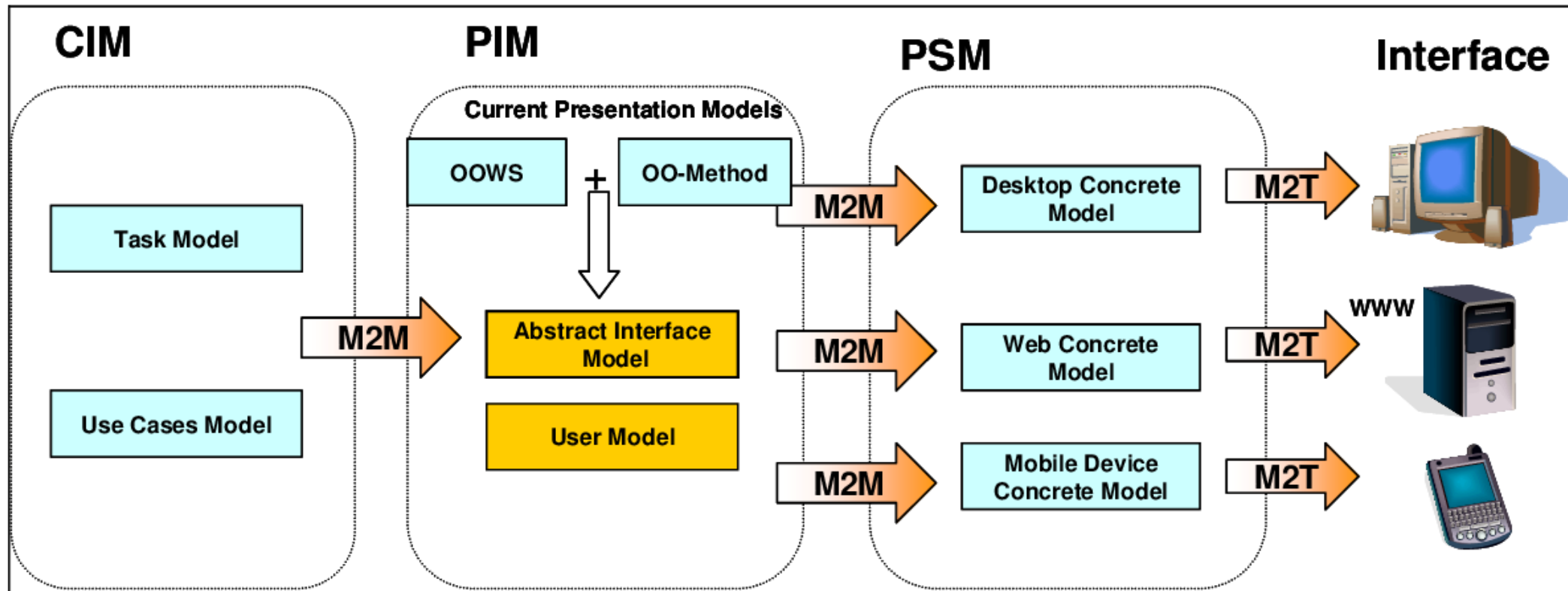
```
int z = 10;
int n;
cin >> n;
int sum = 0;
if (n > 10)
    sum = sum + n;
else
    sum = sum - n;
cout << "Hey";
```

# ABSTRACT MODEL





# LOGICAL MODEL





# LOGICAL MODEL

- Class Diagrams
- Usecase
- Sequence
- Activity etc.



# PHYSICAL MODEL

