哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Chapter 1: Views and Quality Objectives of Software Construction

# 1.1 Multi-Dimensional Views of Software Construction

Ming Liu

February 11, 2020

# Outline

- **Multi-dimensional software views**
  - By phases: build- and run-time views
  - By dynamics: moment and period views
  - By levels: code and component views
  - Elements, relations, and models of each view
- **Software construction: transformation between views**
  - $\varnothing \Rightarrow$ Code
  - Code $\Rightarrow$ Component
  - Build-time $\Rightarrow$ Run-time
  - Moment $\Rightarrow$ Period
- **Summary**

# Objective of this lecture

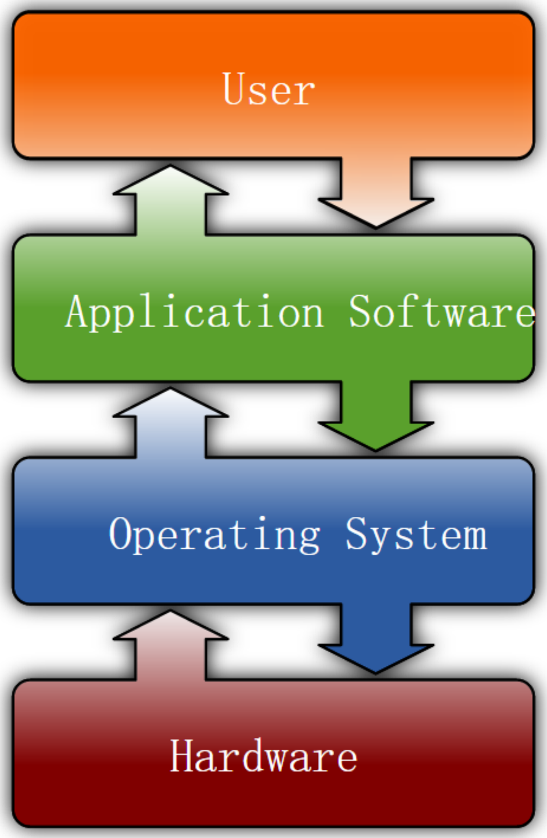
- **To understand the constituents of a software system in three orthogonal dimensions;**
- **To know what models are used to describe the morphology and states of a software system;**
- **To treat software construction as the transformations between different views;**

# 1 Multi-dimensional software views

# What is a Software?

- **The term "software" was firstly proposed by Alan Turing.**
  - System software vs. Application software
  - Desktop/web/mobile/embedded software
  - Business/personal-oriented software
  - Open source vs. proprietary software

Alan Turing (1912-1954)



# Constituents of a software system

- **Software = Program (codes)?**
- **Software = Algorithms + Data Structure?**
- **Software = Program + Data + Documents**
- **Software = Modules (Components) + Data/Control Flows**

Donald E. Knuth (1938- )
Turing Award 1974

Edager Dijkstra (1930-2002)
Turing Award 1972

Niklaus Wirth (1934-)
Turing Award 1984

# Multi-dimensional software views

<table>
<tr><th></th><th colspan="2">Moment</th><th colspan="2">Period</th></tr>
<tr><th></th><th>Code-level</th><th>Component-level</th><th>Code-level</th><th>Component-level</th></tr>
<tr><th>Build-time</th><td>Source code, AST, Interface-Class-Attribute-Method (Class Diagram)</td><td>Package, Source File, Static linking, Library, Test Case (Component Diagram)</td><td>Code Churn (代码变化)</td><td>Configuration Item, Version</td></tr>
<tr><th rowspan="2">Run-time</th><td rowspan="2">Code Snapshot, Memory dump</td><td rowspan="2">Package, Library, Dynamic linking, Configuration, Database, Middleware, Network, Hardware (Deployment Diagram)</td><td>Execution trace</td><td>Event log</td></tr>
<tr><td colspan="2">Procedure Call Graph, Message Graph (Sequence Diagram)<br><br>Parallel and multi-threads/processes<br>Distributed processes</td></tr>
</table>

# (1) Build-time Views

# Build-time views of a software system

- **Build-time: idea ⇒ requirement ⇒ design ⇒ code ⇒ installable / executable package**
  - **Code-level view**: source code ---- how source code are logically organized by basic program blocks such as functions, classes, methods, interfaces, etc, and the dependencies among them
  - **Component-level view**: architecture ---- how source code are physically organized by files, directories, packages, libraries, and the dependencies among them

    ------------------------------------------------------------
  - **Moment view**: what do source code and component look like in a specific time
  - **Period view**: how do they evolve/change along with time

# (1) Build-time, moment, and code-level view

- **How source code are logically organized by basic program blocks such as functions, classes, methods, interfaces, etc, and the dependencies among them.**
- **Three inter-related forms:**
  - Lexical-oriented source code
  - Syntax-oriented program structure: e.g., Abstract Syntax Tree (AST)
  - Semantics-oriented program structure: e.g., Class Diagram

# Lexical-based semi-structured source code

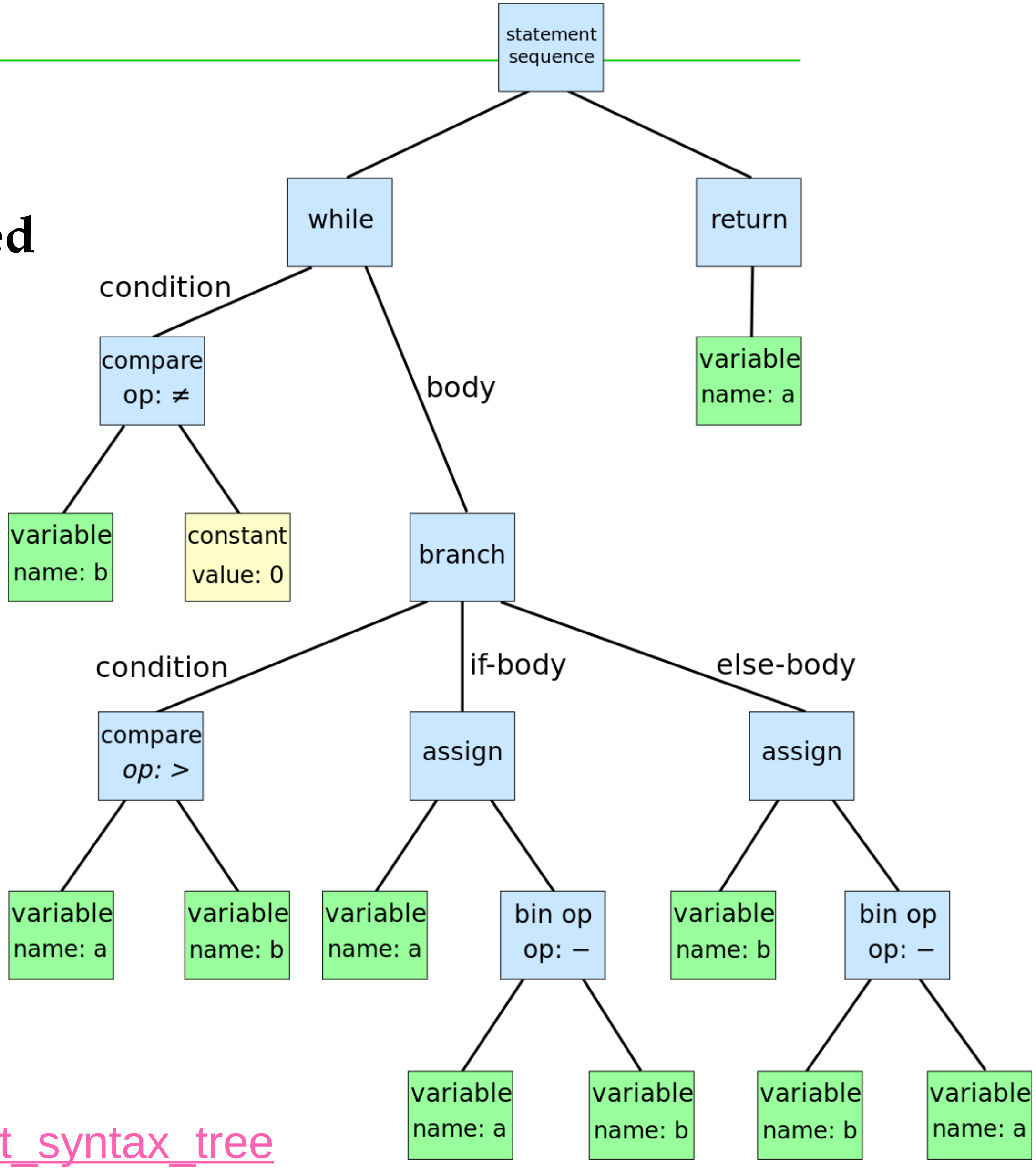
- **Source code: the most important assets in software development**

```
// First, log in
LoginResult loginResult=null;
SoapBindingStub sfdc=null;
sfdc = (SoapBindingStub) new SforceServiceLocator().getSoap();
// login
loginResult = sfdc.login("username","password");

// The set up some security related items
// Reset the SOAP endpoint to the returned server URL
sfdc._setProperty(SoapBindingStub.ENDPOINT_ADDRESS_PROPERTY,loginResult.getServerUrl());
// Create a new session header object
// add the session ID returned from the login
SessionHeader sh=new SessionHeader();
sh.setSessionId(loginResult.getSessionId());
sfdc.setHeader(new SforceServiceLocator().getServiceName().getNamespaceURI(),
  "SessionHeader",sh);

// now that we're logged in, make some calls - retrieve information about the user
GetUserInfoResult userInfo = sfdc.getUserInfo();

// create a new account object locally
Account account = new Account();
account.setAccountNumber("002DF99ELK9");
account.setName("My New Account");
account.setBillingCity("Glasgow")


SObject[] sObjects = new SObject[2];
sObjects[0] = account;

// persist the object
SaveResult[] saveResults = sfdc.create(sObjects);
```

# Syntax-oriented program structure

- **Abstract Syntax Tree (AST)**
- **To represent semi-structured source code as a structured tree.**

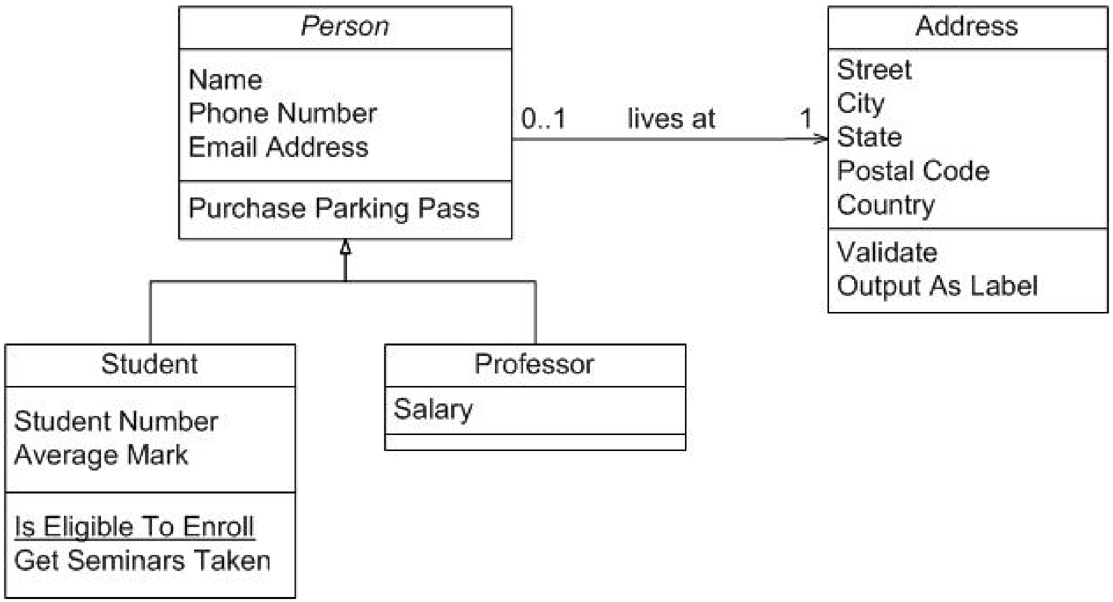
```
while (a ≠ b) {
   if (a > b)
      a = a − b;
   else
      b = b − a;
   return a;
}
```



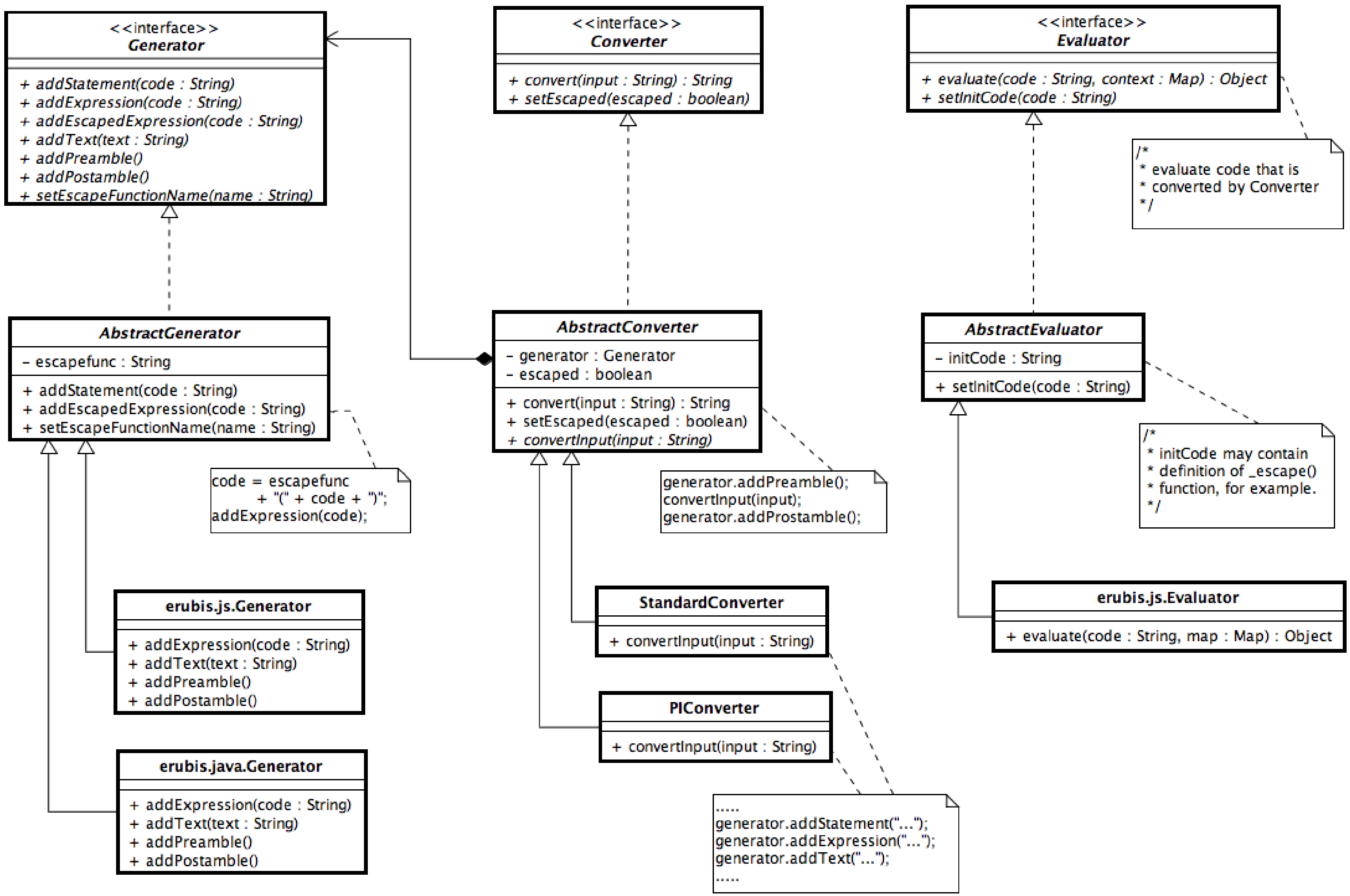

https://en.wikipedia.org/wiki/Abstract_syntax_tree

# Semantics-oriented program structure

- **E.g., using Class Diagram (UML) to describe the interfaces, classes, attributes, methods, and relationships among them.**
- **Graphics-based or formally defined.**
- **Usually modeled in design phase, and transformed into source code.**
- **It is the result of Object-Oriented Analysis and Design in terms of user requirements.**



# Semantics-oriented program structure



# (2) Build-time, period, and code-level view

- **Views describing "changes" along with time.**
- **Code churn: Lines added, modified or deleted to a file from one version to another.**

**Code Before**

```
int i = n;
while (i−−)
   printf (" %d", i);
```

**Code After**

```
//print n integers iff n≥ 0
int i = n;
while (−−i > 0)
   printf (" %d", i);
```

two lines added

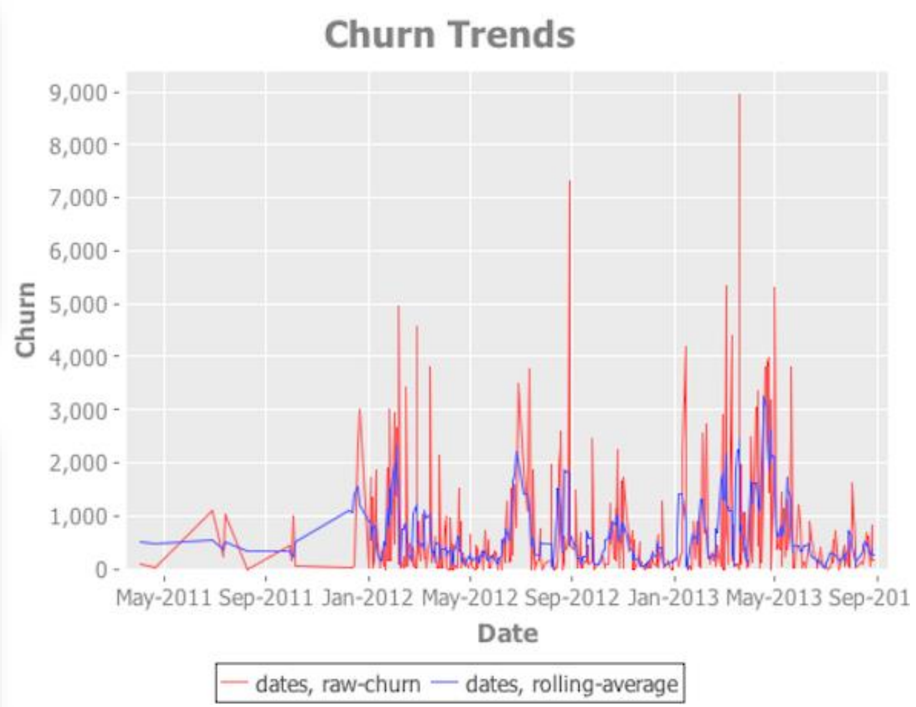
**Code Before**

```
int i = n;
while (i−−)
   printf (" %d", i);
```

**Code After**

```
//print n integers iff n≥ 0
int i = n;
while (−−i > 0)
   printf (" %d", i);
```

one line deleted

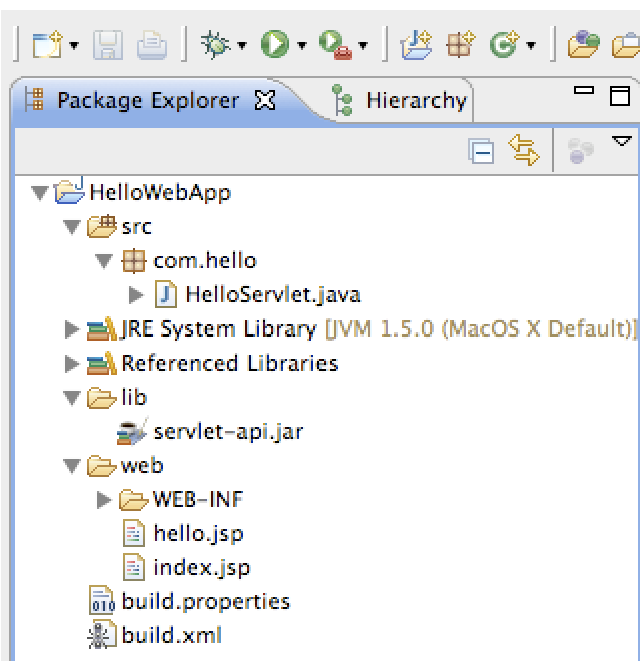


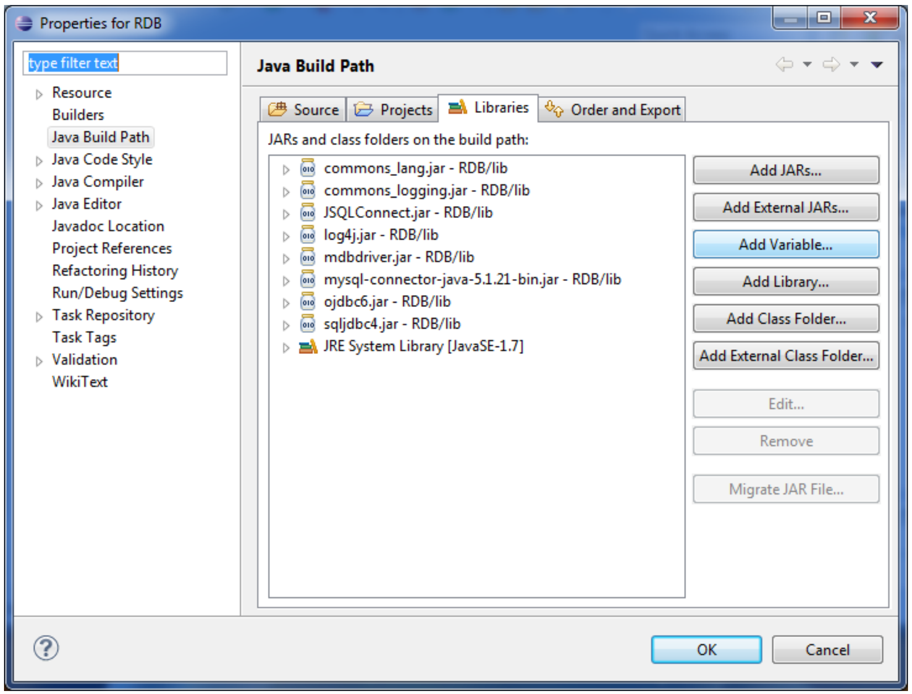
# (3) Build-time, moment, and component-level view

- **Source code are physically organized into files which further are organized by directories;**

Static linking

- **Files are encapsulated into packages and, logically, components and sub-systems.**
- **Reusable modules are in the form of libraries.**



# Library

- **Libraries** are stored in disk files of their own, collect a set of code functions that can be reused across a variety of programs.
  - Developers aren't always building a single executable program file, but join custom-developed software and prebuilt libraries into a single program.
- **In build-time**, a library function can be viewed as an extension to the standard language and is used in the same way as functions written by the developers.

```
System.out.println("Hello World");
```

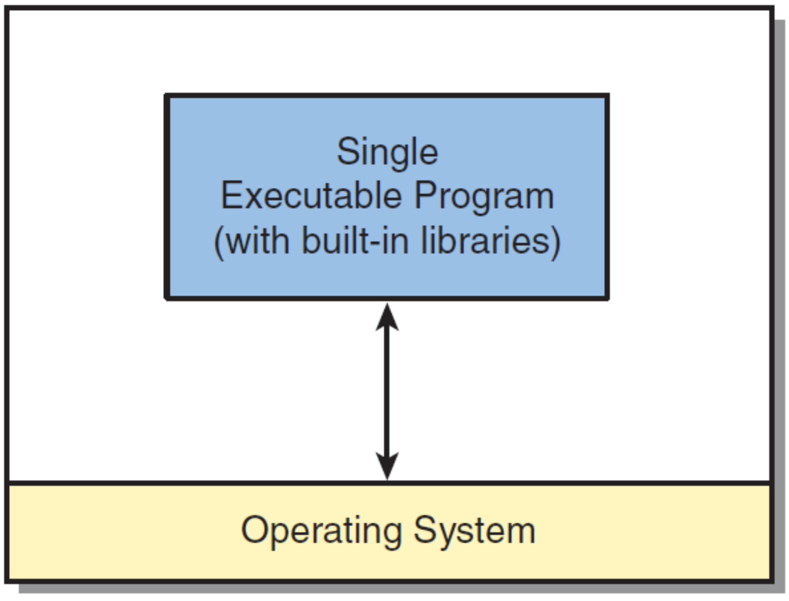
- **Sources of libraries:**
  - From OS pre-installed set of libraries for operations such as file and network I/O, GUI, mathematics, database assess;
  - From language SDK;
  - From third-party sources, such as downloading them from the Internet.
  - Developers can also publish their own libraries.

# Linking with a library

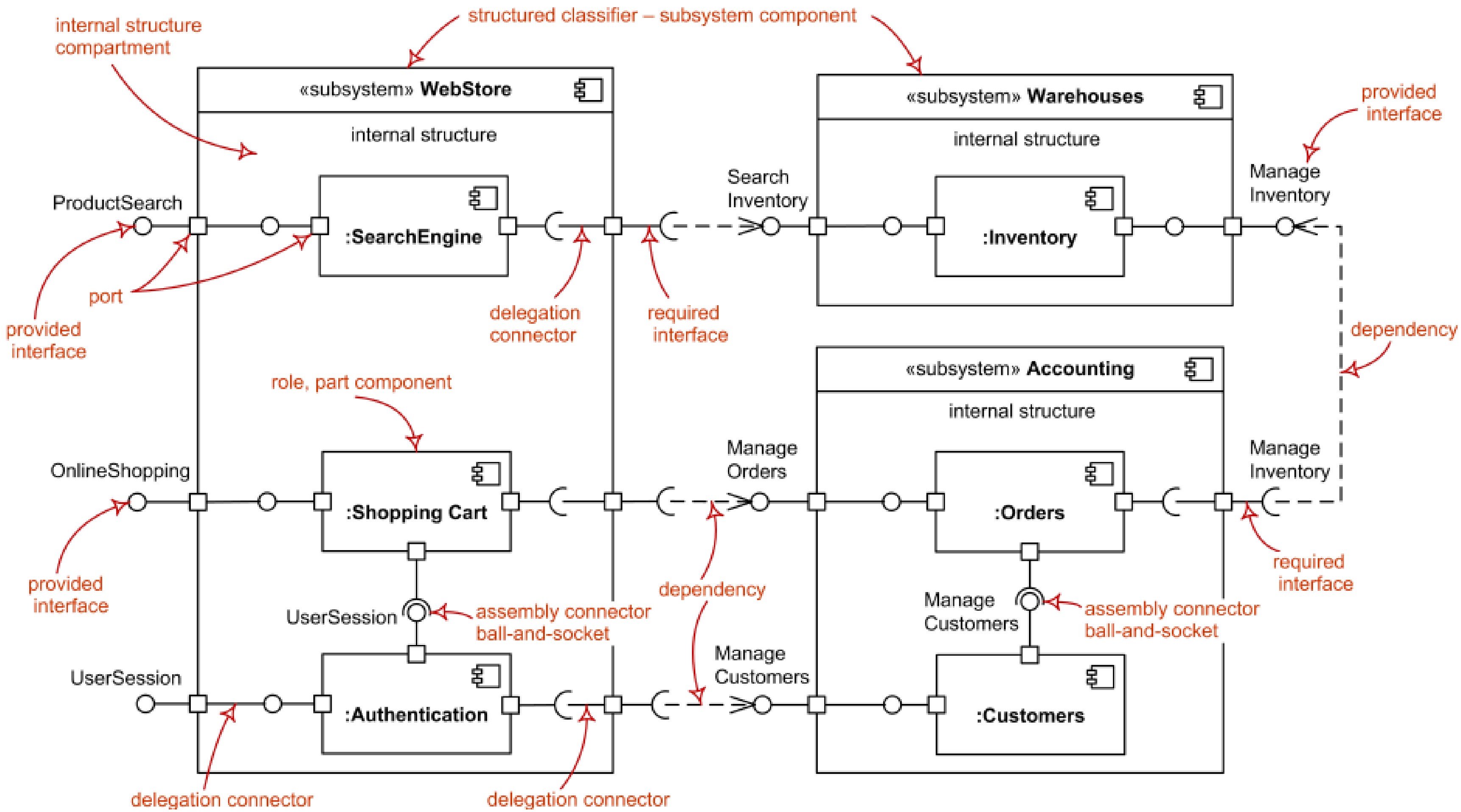
- **When a program is edited, built and installed, a list of libraries to search must be provided.**
- **If a function is referenced in the source code but the developer didn't explicitly write it, the list of libraries is searched to locate the required function.**
- **When the function is found, the appropriate object file is copied into the executable program**.
- **Two different approaches of integrating a library into an executable program:**
  - Static linking
  - Dynamic linking

# Static linking

- **In static linking, a library is a collection of individual object files.**
- **During the build process**, when the linker tool determines that a function is required, **it extracts the appropriate object file from the library and copies it into the executable program.**
  – The library's object file appears identical to any of the object files the developer created on his or her own.
- **Static linking happens in build time ---** The act of linking a library with the developer's own software happens during the build process.
  – **End up with a single executable program** to be loaded onto the target machine.
  – After the final executable program has been created, it's impossible to separate the program from its libraries.

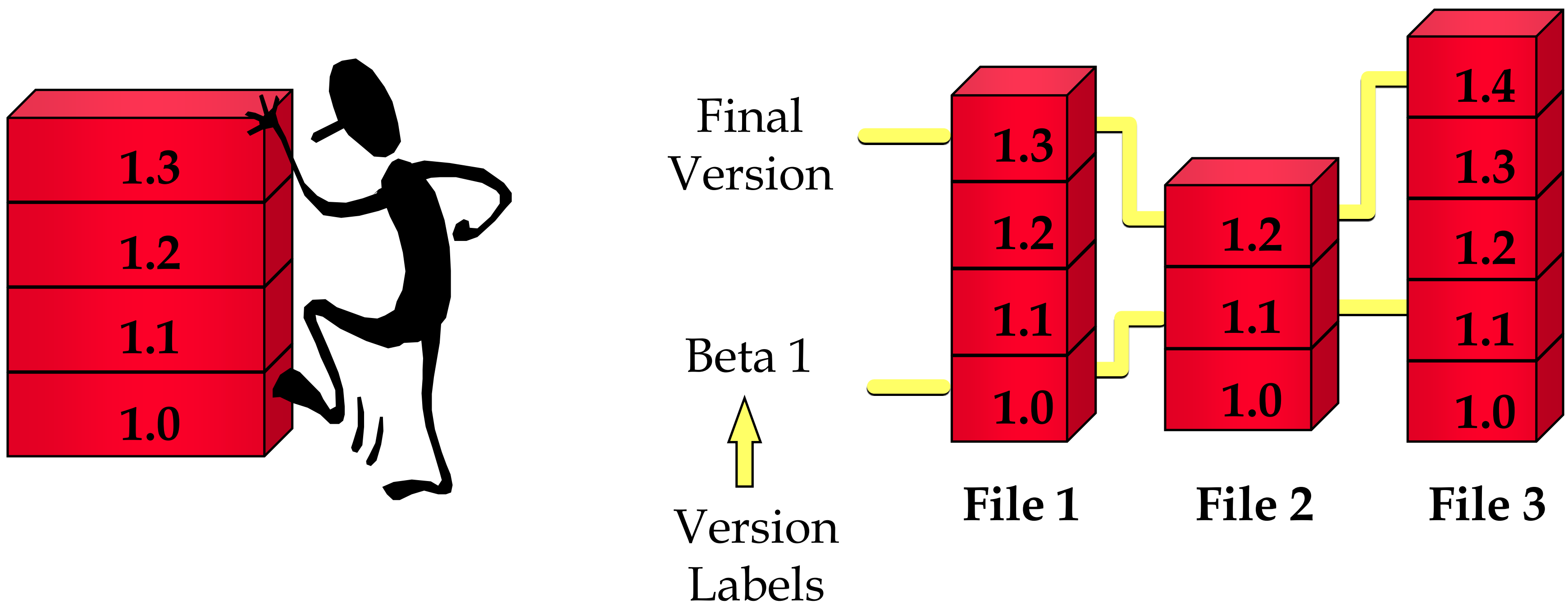


# Component diagram in UML

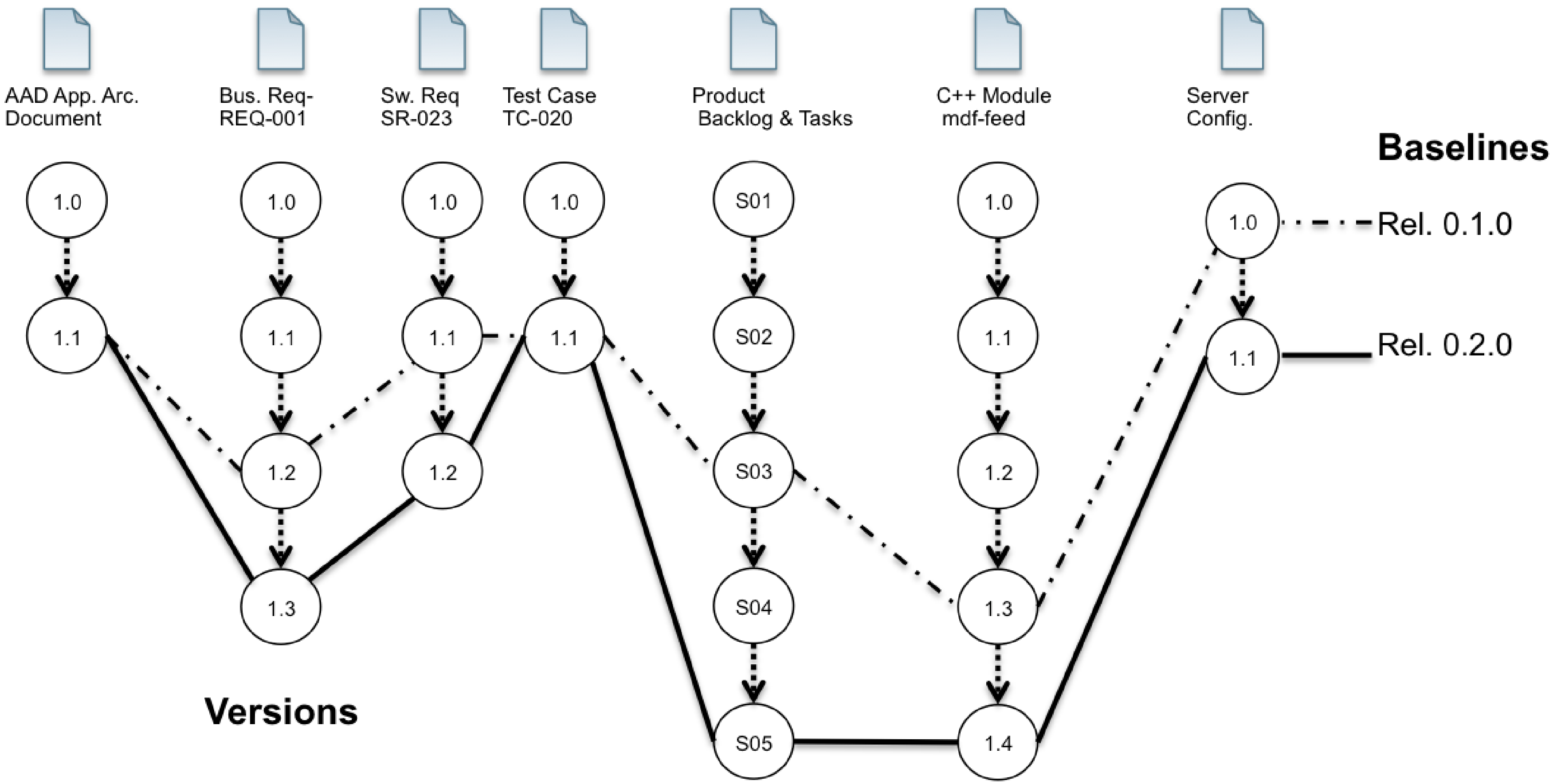


# (4) Build-time, period, and component-level view

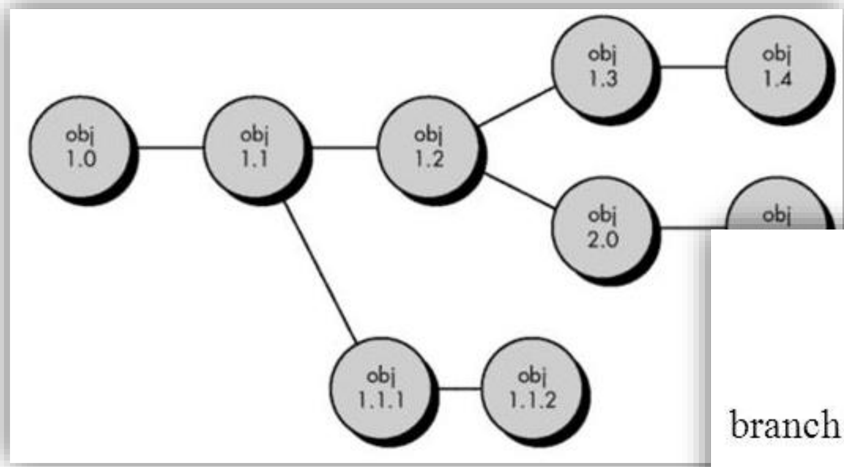
- **How do all files/packages/components/libraries change in a software system along with time?**
- **Software Configuration Item (SCI)**
- **Version**

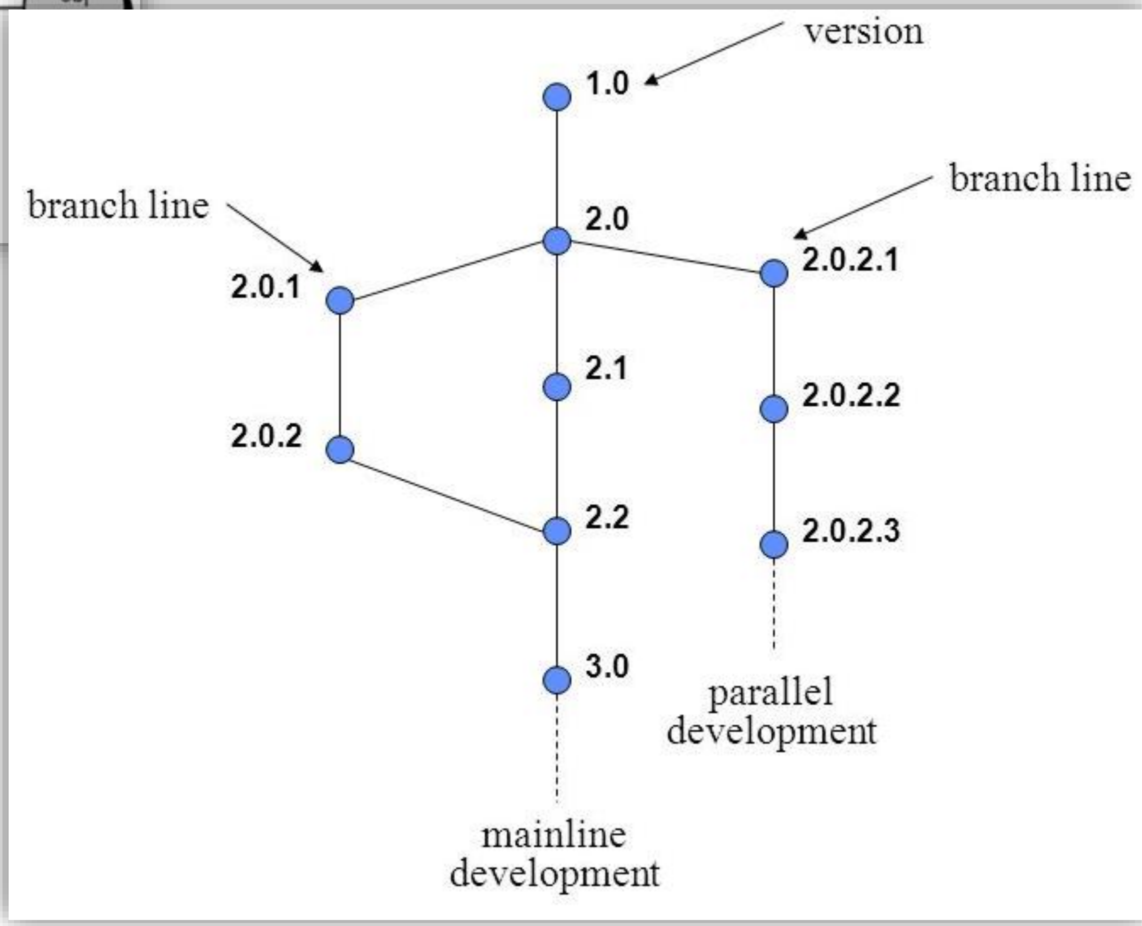


# Version Control System (VCS)



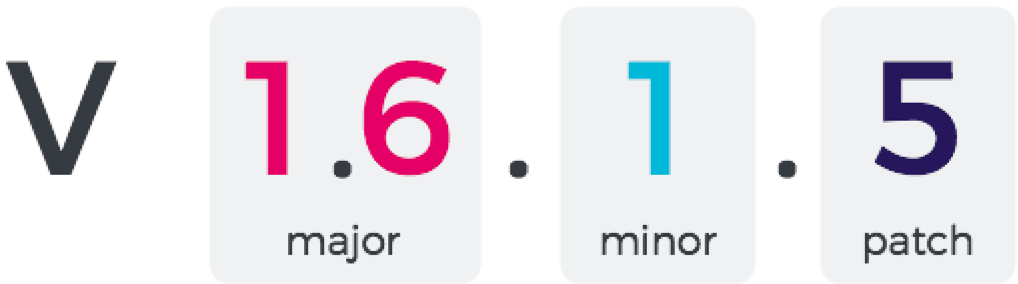
# Evolution Graph (of a SCI or a Software)

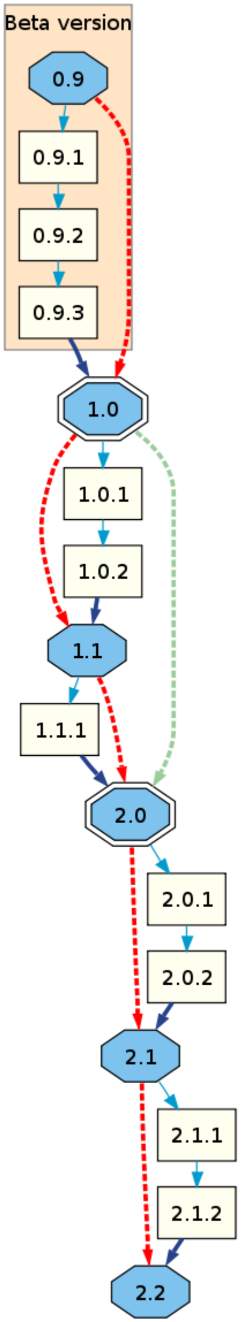


# Versioning

- **Software versioning is the process of assigning either unique version names or unique version numbers to unique states of computer software.**
  - Within a given version number category (major, minor), these numbers are generally assigned in increasing order and correspond to new developments in the software.
  - At a fine-grained level, revision control is often used for keeping track of incrementally different versions of electronic information, whether or not this information is computer software.



# Software Evolution

- **Software evolution is a term used in software maintenance, referring to the process of developing software initially, then repeatedly updating it for various reasons.**
  - Over 90% of the costs of a typical system arise in the maintenance phase, and that any successful piece of software will inevitably be maintained.


Frederick Brooks (1931-)
Turing Award 1999

# (2) Runtime Views

# Runtime views of a software system

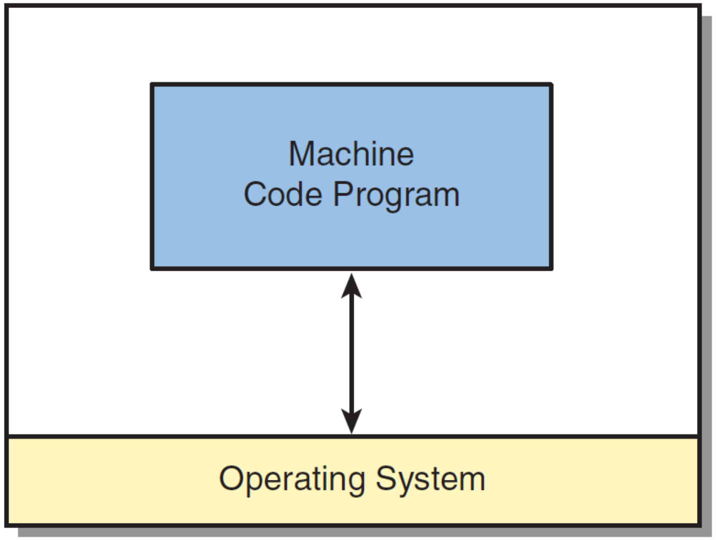
- **Runtime:** what does a program look like when it runs inside the target machine, and what are all the disk files that the target machine needs to load into memory?
  - **Code-level view**: source code ---- what do the in-memory states of an executable program look like and how do program units (objects, functions, etc) interact with each other?
  - **Component-level view**: architecture ---- how are software packages deployed into physical environment (OS, network, hardware, etc) and how do they interact?

------------------------------------------------------------

  - **Moment view**: how do programs behave in a specific time
  - **Period view**: how do they behave along with time

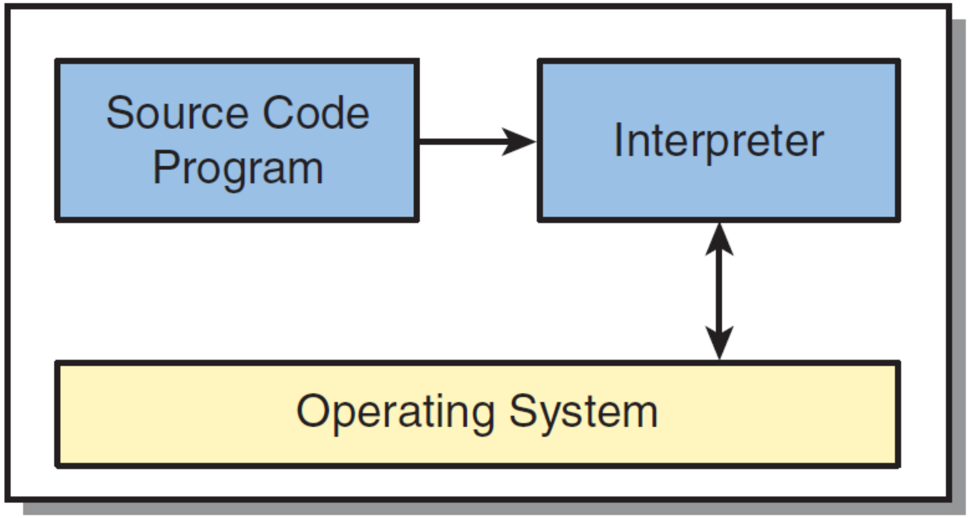
# Executable Programs: Native Machine Code

- A program is loaded into memory first, and several mechanisms exist for executing the software, depending on how much compilation took place before the program was loaded and how much OS supports the program requires.
- **Native Machine Code:**
  - Fully converted executable program into the CPU's native machine code.
  - The CPU simply "jumps" to the program's starting location, and all the execution is performed purely using the CPU's hardware.
  - While it's executing, the program optionally makes calls into the operating system to access files and other system resources.
  - This is the fastest way to execute code, because the program full accesses to the CPU's features.



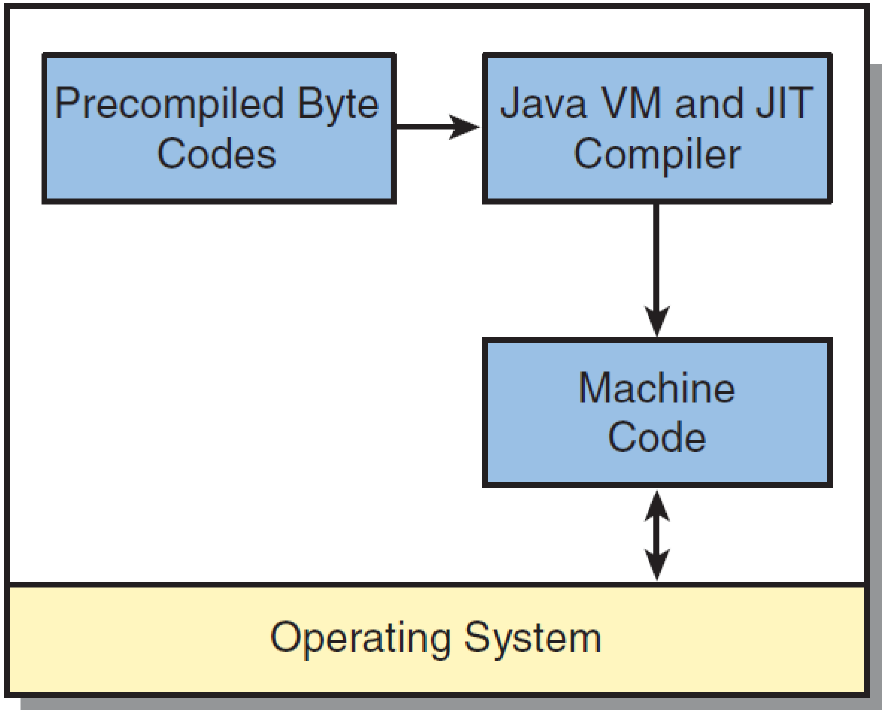
# Executable Programs: Full Interpretation

- **Full Program Interpretation:** the runtime system loads the entire source code into memory and interprets it (such as BASIC, UNIX shell, etc)



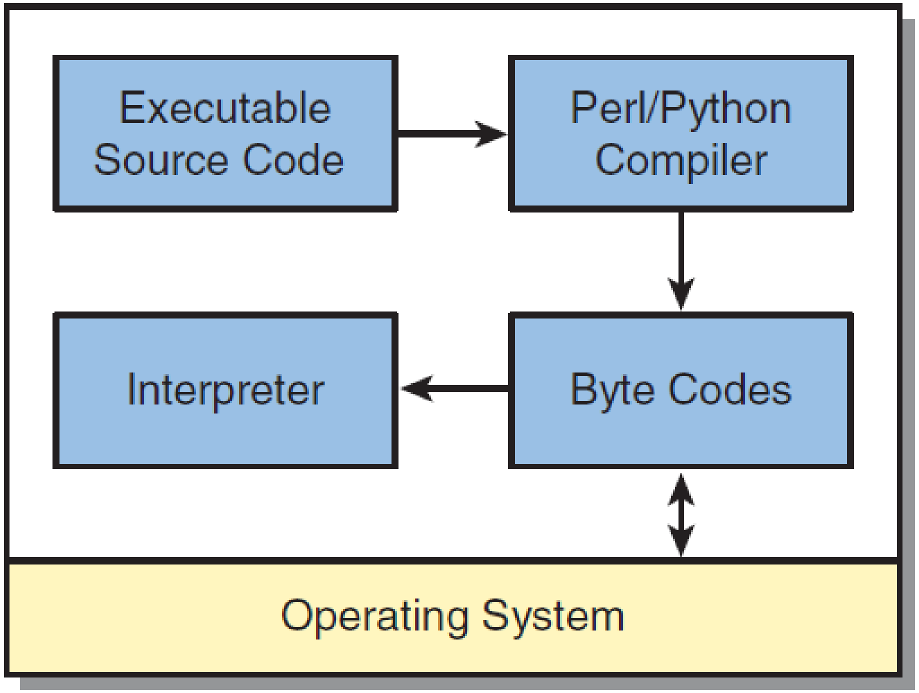
# Executable Programs: Interpreted Byte Codes

- **Interpreted Byte Codes:**
  - Byte codes are similar to native machine code, except that the CPU doesn't directly understand them.
  - It first translates them into native machine code or interprets them as the program executes.
  - A byte code environment therefore requires that an additional interpreter or compiler be loaded alongside the program.
- **Java Virtual Machine (JVM)**



# Executable Programs: Interpreted Byte Codes

- **Perl or Python:** they are interpreted rather than compiled, but use byte codes at runtime.
- **The simple act of executing the Perl or Python script automatically triggers the generation of byte codes.**

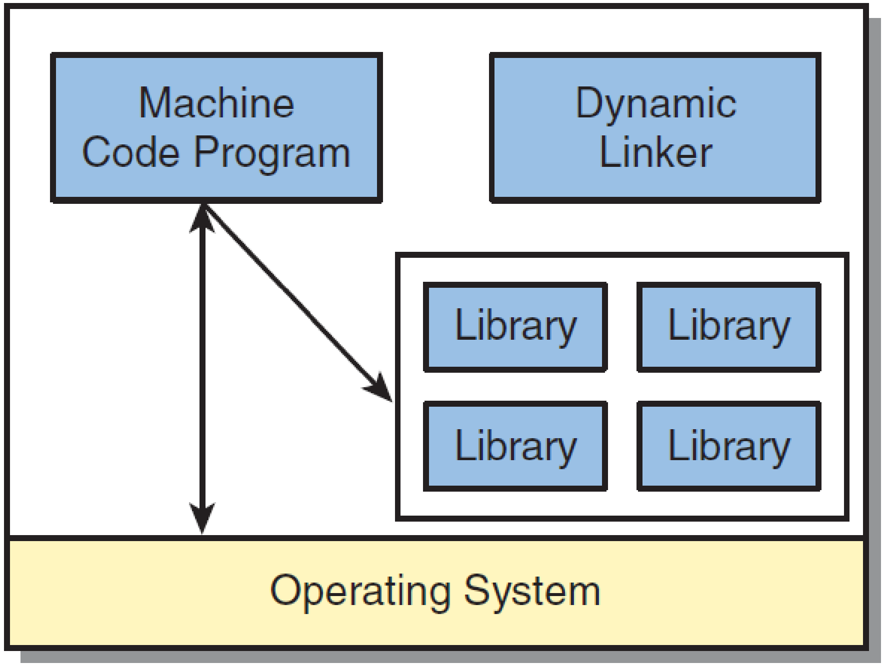


# Dynamic linking

- **Dynamic linking** method doesn't copy the object file into the executable image; instead, **it notes which libraries are required to successfully execute the program.**
- When the program starts running, th**e libraries are loaded into memory as separate entities** and then are connected with the main program.
- A dynamic library is a disk file that is constructed by joining object files. **The library is then collected into the release package and installed on the target machine**. Only then can it be loaded into the machine's memory.

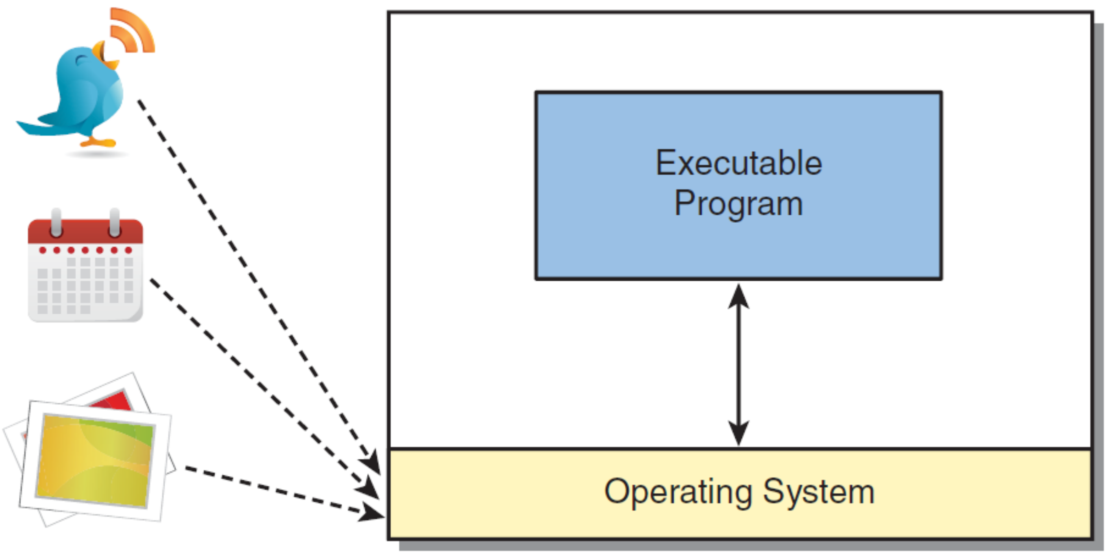
# Dynamic linking

- **Advantages:**
  - It's possible to upgrade to a newer version of a library (adding features or fixing bugs), without needing to re-create the executable program.
  - Many operating systems can optimize their memory usage by loading only a single copy of the library into memory, yet sharing it with other programs that require that same library.



# Configuration and Data Files

- **Any program of significant size uses external data sources, such as a file on a disk.**
- **Your program makes calls into the operating system to request that data be read into memory.**
  - A bitmap graphic image displayed onscreen
  - A sound stored as a digitized wave form
  - A configuration file that customizes the behavior of a program
  - A set of documents containing online help text
  - A database containing names and addresses

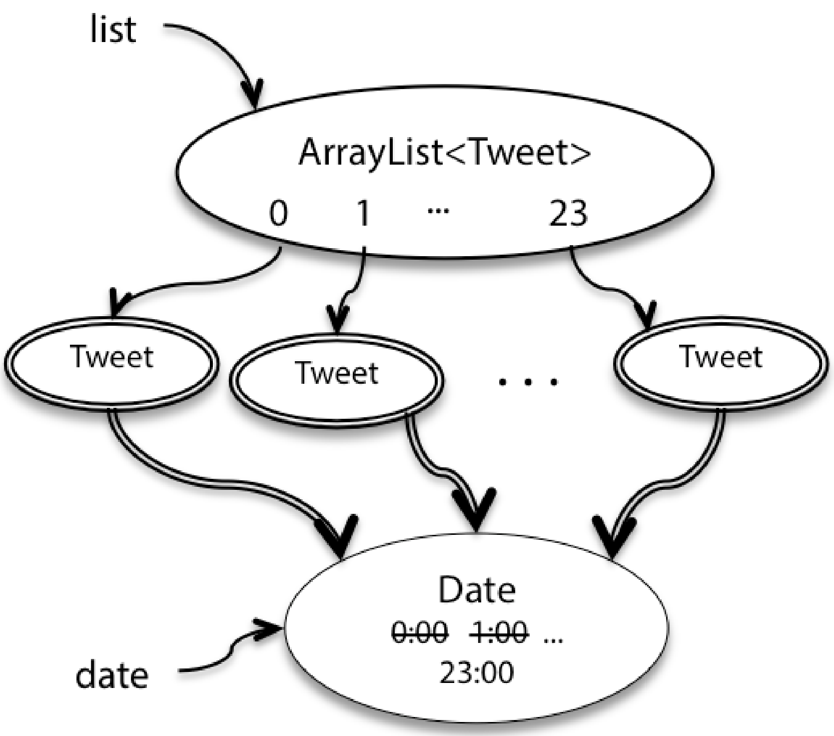


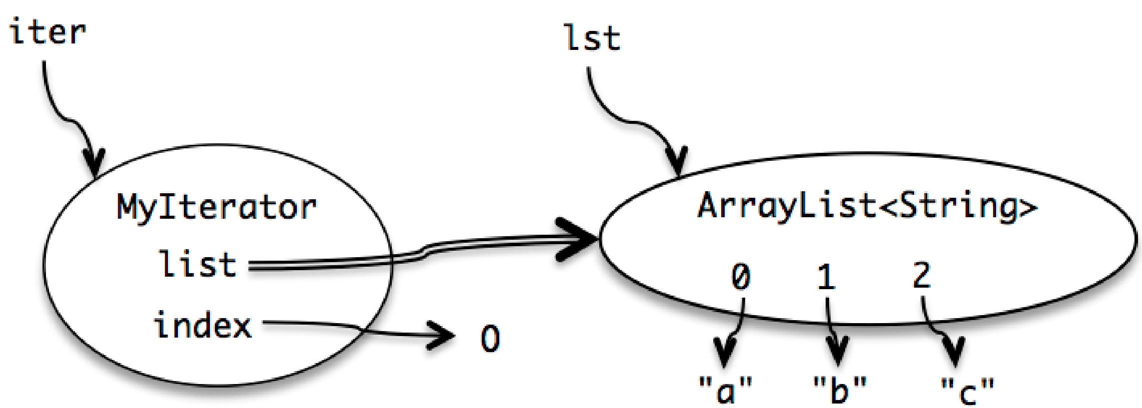
# Distributed Programs

- For example, a software system might use the client/server model, with a single server program running on one computer and a large number of client programs running on many other computers.
- In this scenario, the build system could create two release packages, given that different people will be installing the server program versus the client program.
- Alternatively, the same release package could be used to install the two separate programs.

# (5) Run-time, moment, and code-level view

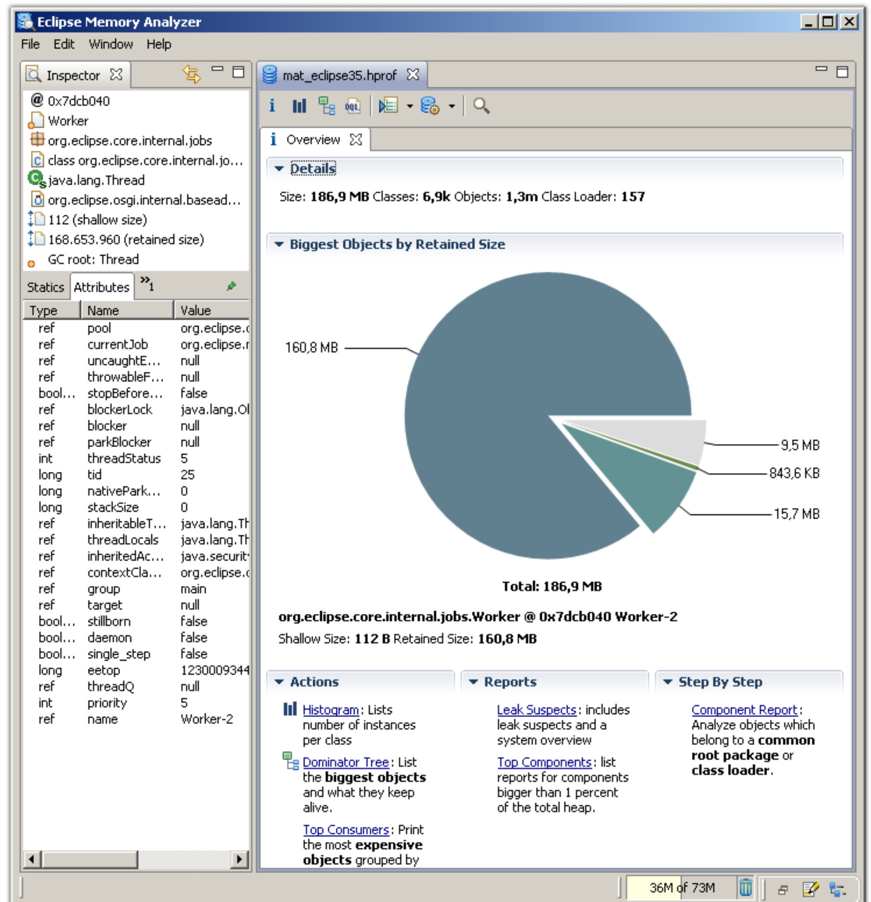
- **Snapshot diagram: focusing on variable-level execution states in the memory of a target computer.**
- **Fine-grained states.**



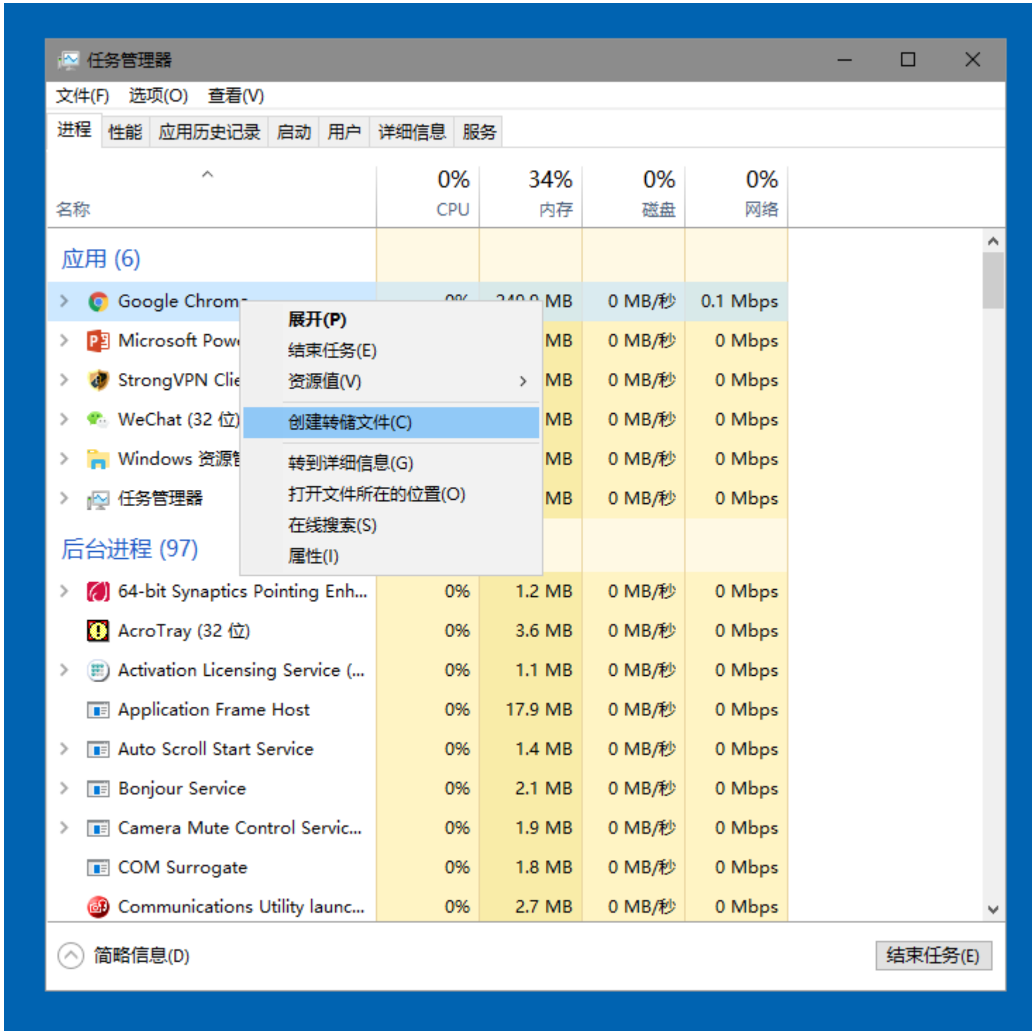



# Memory dump

- **Memory dump: a file on hard disk containing a copy of the contents of a process's memory, produced when a process is aborted by certain kinds of internal error or signal.**
  - Debuggers can load the dump file and display the information it contains about the state of the running program.
  - Information includes the contents of registers, the call stack and all other program data (counters, variables, switches, flags, etc).
  - It is taken in order to analyze the status of the program, and the programmer looks into the memory buffers to see which data items were being worked on at the time of failure.



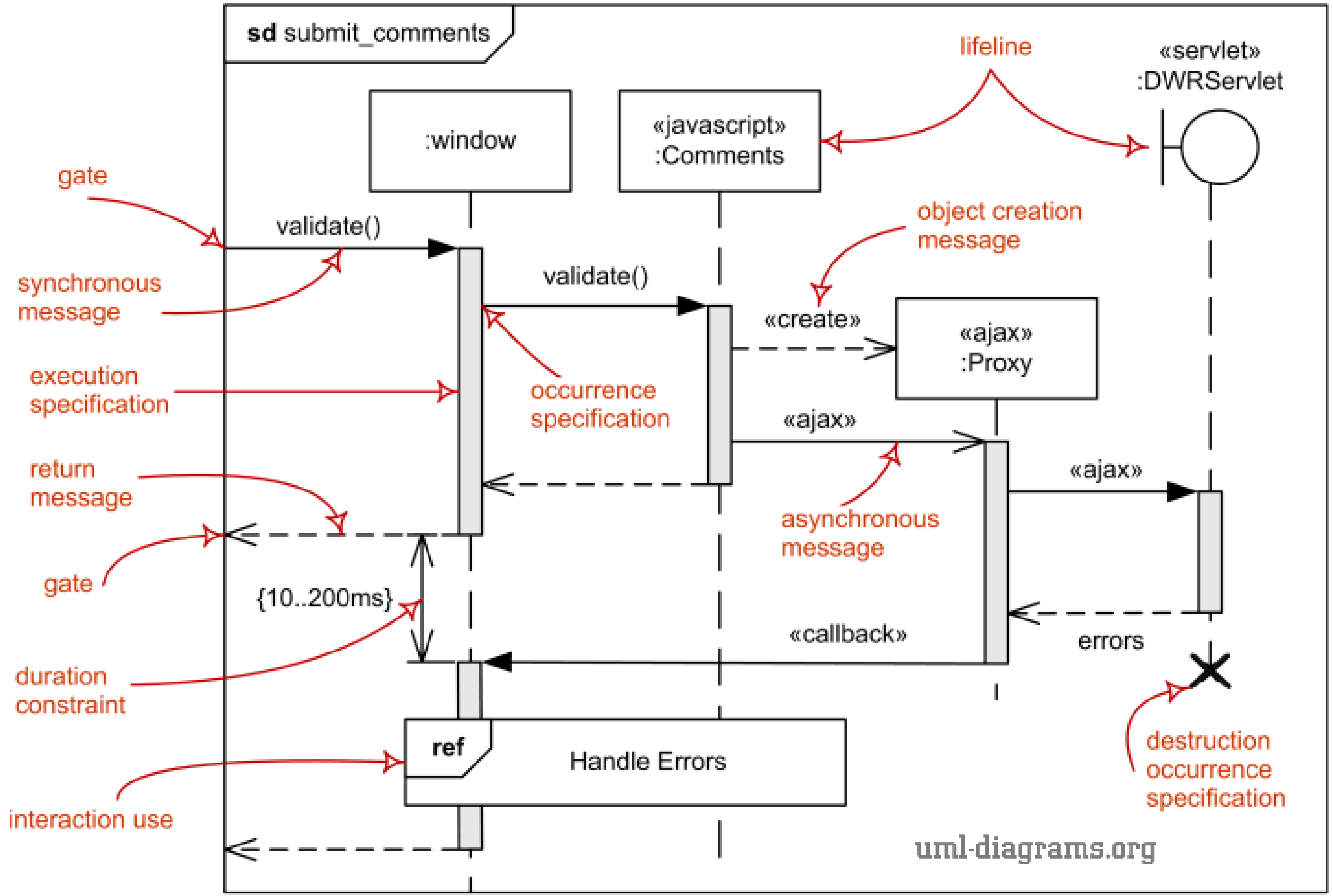
# Memory dump



任务管理器
文件(F) 选项(O) 查看(V)
进程 性能 应用历史记录 启动 用户 详细信息 服务
名称
0% CPU
34% 内存
0% 磁盘
0% 网络
应用 (6)
Google Chrom
Microsoft Pow
StrongVPN Clie
WeChat (32 位)
Windows 资源
任务管理器
展开(P)
结束任务(E)
资源值(V)
创建转储文件(C)
转到详细信息(G)
打开文件所在的位置(O)
在线搜索(S)
属性(I)
后台进程 (97)
64-bit Synaptics Pointing Enh...
AcroTray (32 位)
Activation Licensing Service (...
Application Frame Host
Auto Scroll Start Service
Bonjour Service
Camera Mute Control Servic...
COM Surrogate
Communications Utility launc...
0%
1.2 MB
3.6 MB
1.1 MB
17.9 MB
1.4 MB
2.1 MB
1.9 MB
1.8 MB
2.7 MB
0 MB/秒
0.1 Mbps
0 Mbps
简略信息(D)
结束任务(E)

# (6) Run-time, period and code-level view

- **Sequence diagram in UML: interactions among program units (objects)**



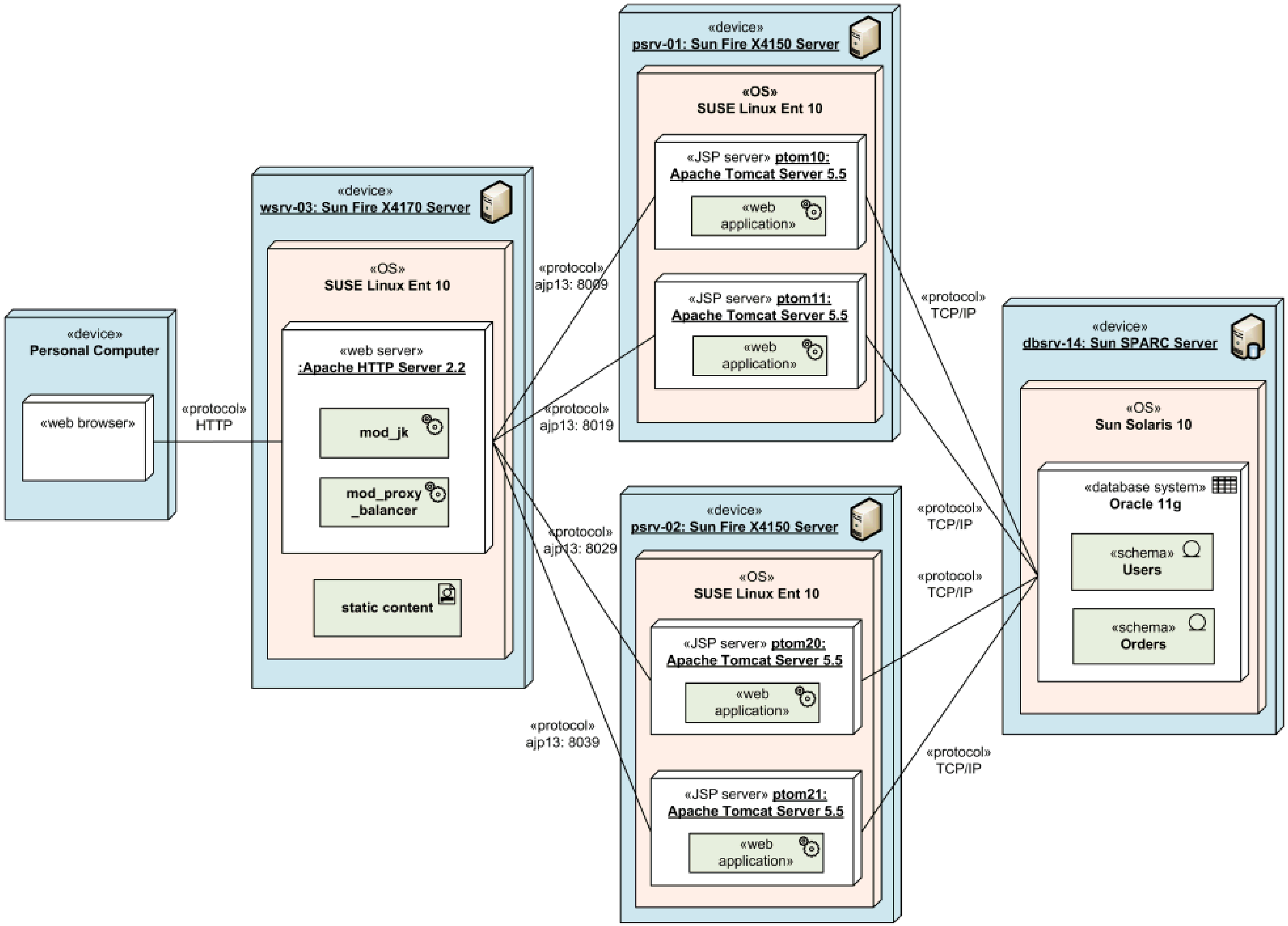
# Execution tracing

- **Tracing involves a specialized use of logging to record information about a program's execution.**
  - This information is typically used by programmers for debugging purposes, and additionally, depending on the type and detail of information contained in a trace log, by experienced system administrators or technical-support personnel and by software monitoring tools to diagnose common problems with software.

```
09-26 10:59:38.056  28584-28584/? E/AndroidRuntime:  FATAL EXCEPTION: main
        java.lang.NullPointerException
        at android.content.ContextWrapper.getResources(ContextWrapper.java:81)
        at android.widget.Toast.<init>(Toast.java:92)
        at android.widget.Toast.makeText(Toast.java:233)
        at com.app.app.MainActivity$3.run(MainActivity.java:132)
        at android.os.Handler.handleCallback(Handler.java:605)
        at android.os.Handler.dispatchMessage(Handler.java:92)
        at android.os.Looper.loop(Looper.java:137)
        at android.app.ActivityThread.main(ActivityThread.java:4428)
        at java.lang.reflect.Method.invokeNative(Native Method)
        at java.lang.reflect.Method.invoke(Method.java:511)
        at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:787)
        at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:554)
        at dalvik.system.NativeStart.main(Native Method)
```

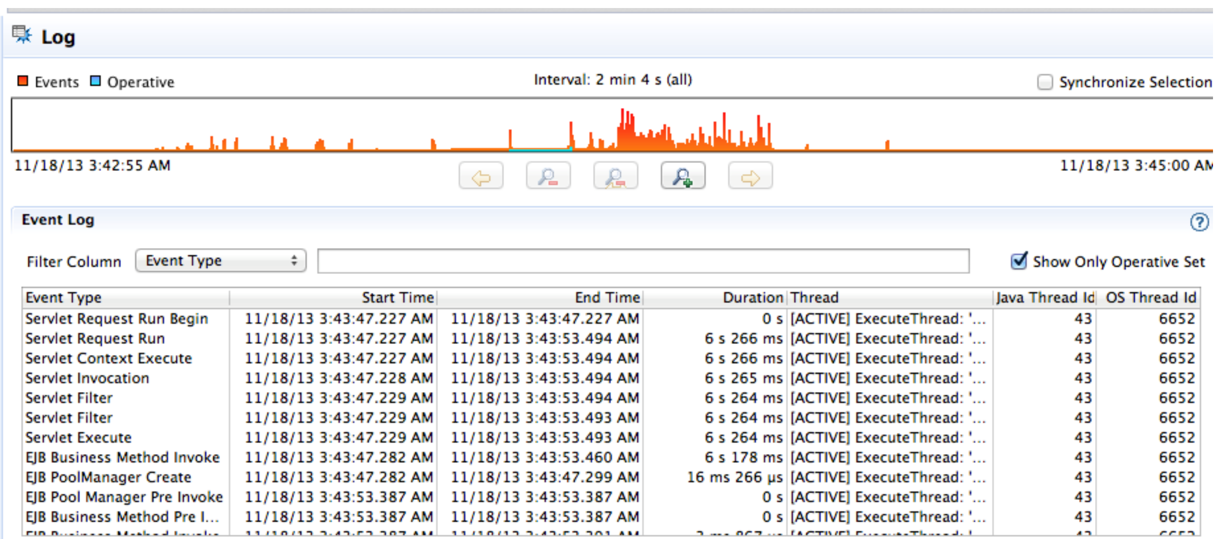
# (7) Run-time, moment, and component-level view

- **Deployment diagram in UML**



# (8) Run-time, period, and component-level view

- **Event logging provides system administrators with information useful for diagnostics and auditing.**
  - The different classes of events that will be logged, as well as what details will appear in the event messages, are considered in development cycle.
- **Each class of event to be assigned a unique "code“ to format and output a human-readable message.**
  - This facilitates localization and allows system administrators to more easily obtain information on problems that occur.

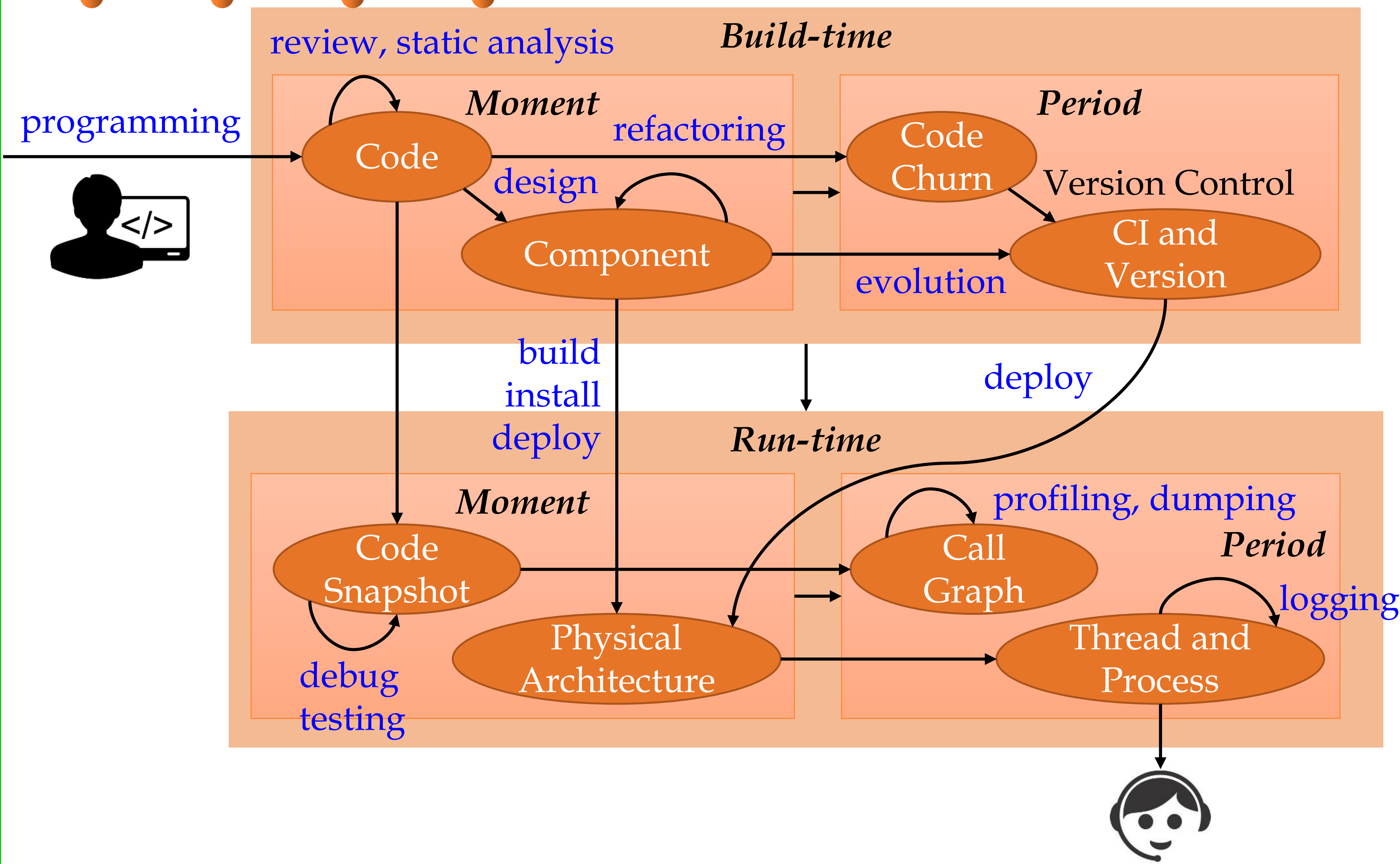


# Execution tracing and event logging

| Event logging | Software tracing |
|---|---|
| Consumed primarily by system administrators | Consumed primarily by developers |
| Logs "high level" information (e.g. failed installation of a program) | Logs "low level" information (e.g. a thrown exception) |
| Must not be too "noisy" (contain many duplicate events or information not helpful to its intended audience) | Can be noisy |
| A standards-based output format is often desirable, sometimes even required | Few limitations on output format |
| Event log messages are often localized | Localization is rarely a concern |
| Addition of new types of events, as well as new event messages, need not be agile | Addition of new tracing messages *must* be agile |

# 2 Software construction: transformation between views

# Software construction: transformation btw views



# Types of Transformations in Software Construction

- **∅ ⇒ Code**
  - Programming / Coding (Chapter 3 ADT/OOP)
  - Review, static analysis/checking (Chapter 4 Understandability)
- **Code ⇒ Component**
  - Design (Chapter 3 ADT/OOP; Chapter 5 Reusability; Chapter 6 Maintainability)
  - Build: compile, static link, package, install, clean (Chapter 2 Construction process)
- **Build-time ⇒ Run-time**
  - Install / deploy (Course in the 3rd year)
  - Debug, unit/integration testing (Chapter 7 Robustness)
- **Moment ⇒ Period**
  - Refactoring (Chapter 9 Refactoring)
  - Version control (Chapter 2 SCM)
  - Loading, dynamic linking, interpreting, execution (dumping, profiling, logging) (Chapter 8 Performance)

# Summary

# Summary of this lecture

- **Three dimensions of describing a software system:**
  - By phases: build- and run-time views
  - By dynamics: moment and period views
  - By levels: code and component views
- **Elements, relations, and models of each view**
- **Software construction: transformation between views**
  - $\varnothing \Rightarrow$ Code
  - Code $\Rightarrow$ Component
  - Build-time $\Rightarrow$ Run-time
  - Moment $\Rightarrow$ Period

# The end

February 11, 2020