哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

1 Views and Quality Objectives of Software Construction

软件构造的多维度视图和质量目标

Wang Zhongjie
rainy@hit.edu.cn

March 2, 2024

Objective of this lecture

- **To understand the constituents of a software system in three orthogonal dimensions**
  从三个维度看软件系统的构成
- **To know what models are used to describe the morphology and states of a software system**
  用什么样的模型/视图描述软件系统
- **To treat software construction as the transformations between different views**
  将“软件构造”看作“不同视图之间的转换”

先要搞清楚“软件构造的对象是什么”、“如何刻画”，
然后再关注“如何构造”

Objective of this lecture

- **To know quality factors to be cared in software construction**
  软件构造过程中应考虑的重要质量指标
- **To understand the consequences if quality objectives cannot be achieved**
  如果达不到期望的质量目标，会有什么后果
- **To know what construction techniques are to be studied for each quality factor in this course**
  有哪些面向质量指标的软件构造技术

除了要搞清楚“要构造的结果是什么”
还要理解清楚“构造的结果和过程如何才算好”

Outline

- **Multi-dimensional software views**
  - By phases: build- and run-time views 按阶段划分：构造时/运行时视图
  - By dynamics: moment and period views 按动态性划分：时刻/阶段视图
  - By levels: code and component views
    按构造对象的层次划分：代码/构件视图
  - Elements, relations, and models of each view
- **Software construction: transformation between views**
  - ∅ ⇒ Code
  - Code ⇒ Component
  - Build-time ⇒ Run-time
  - Moment ⇒ Period

Outline

- **Quality properties of software systems**
  - External vs. internal quality factors
  - Important external quality factors 外部质量指标、内部质量指标
  - Tradeoff between quality factors 质量指标之间的折中
- **Five key quality objectives of software construction 软件构造的五个关键质量目标**
  - **Easy to understand**: elegant and beautiful code / understandability
  - **Ready for change**: maintainability and adaptability
  - **Cheap for develop**: design for/with reuse: reusability
  - **Safe from bugs**: robustness
  - **Efficient to run**: performance
- **Summary**

Reading

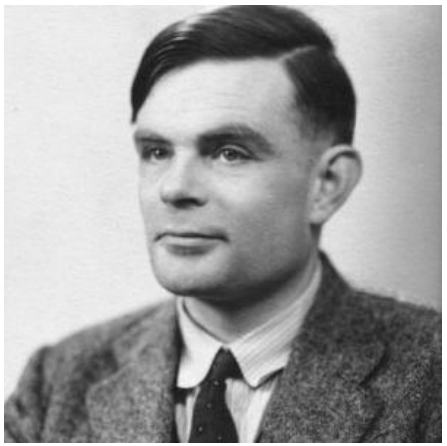
- **MIT 6.031：Getting started, readings 02**
- **CMU 17-214：Aug 29**
- **代码大全：第1-4章**
- **Object-Oriented Software Construction：第1章**
- **代码大全：第20章**
- **软件工程--实践者的研究方法：第14章**

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

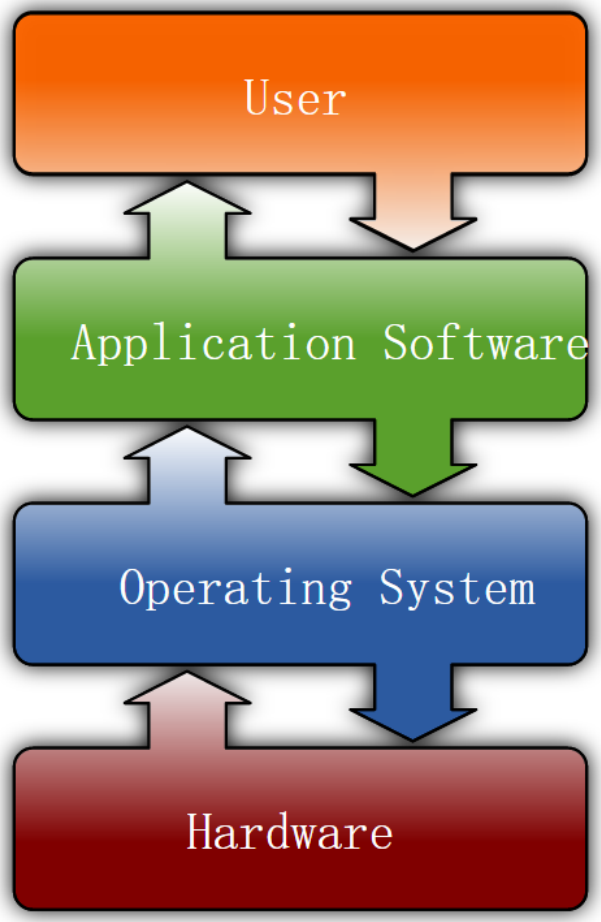
1 Multi-dimensional software views

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**

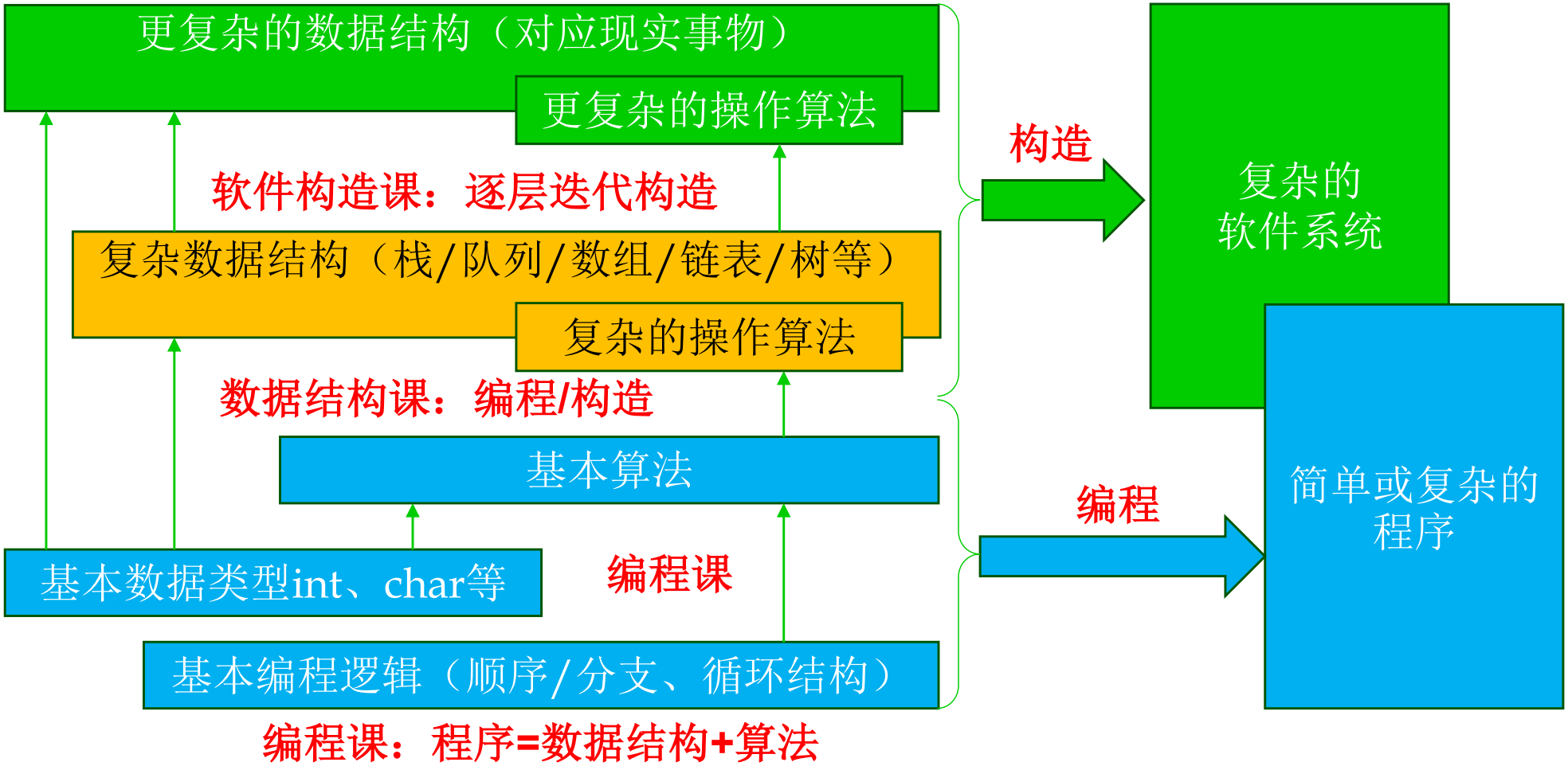
Donald E. Knuth (1938-)
Turing Award 1974

Edager Dijkstra (1930-2002)
Turing Award 1972

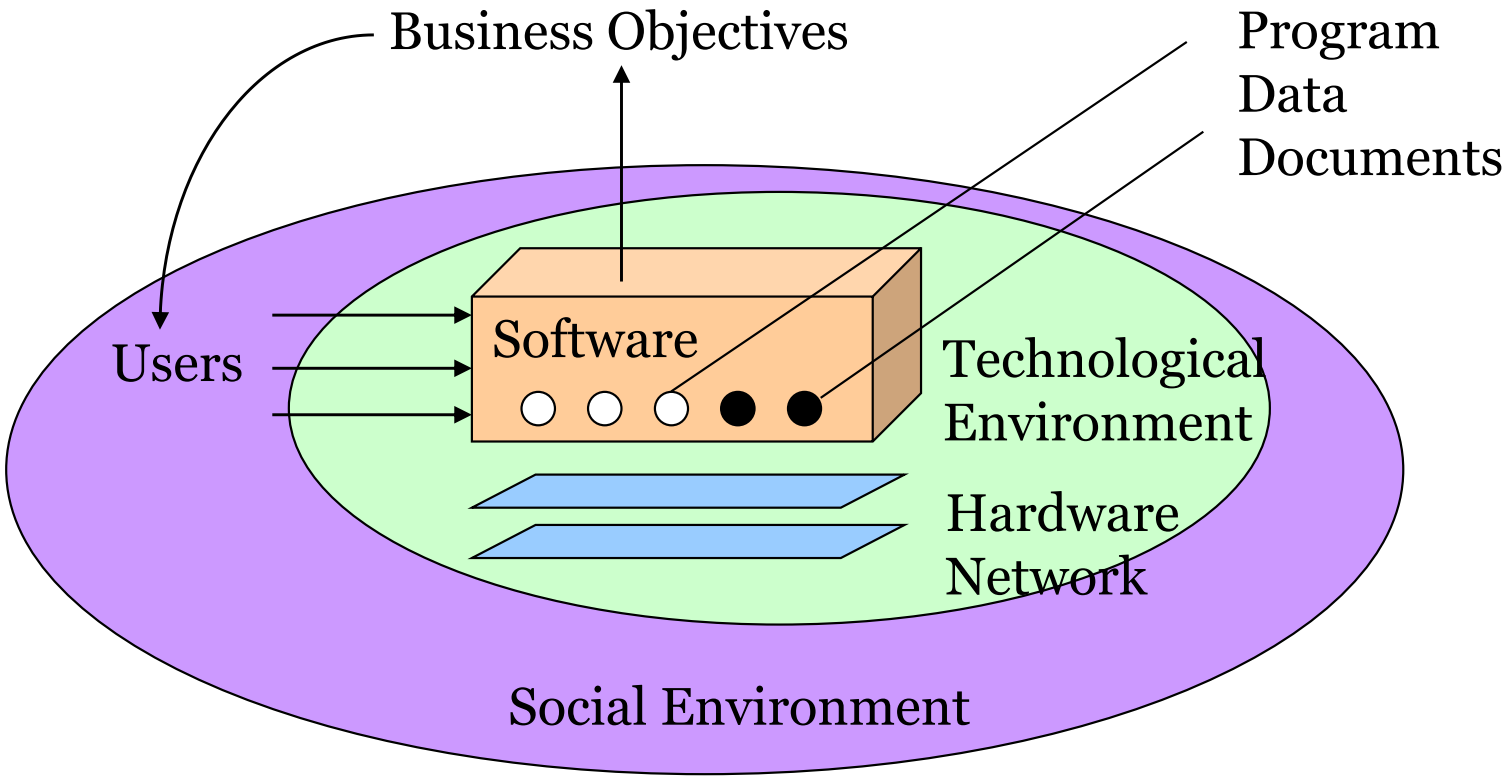
Niklaus Wirth (1934-)
Turing Award 1984

回忆：编程 vs 软件构造

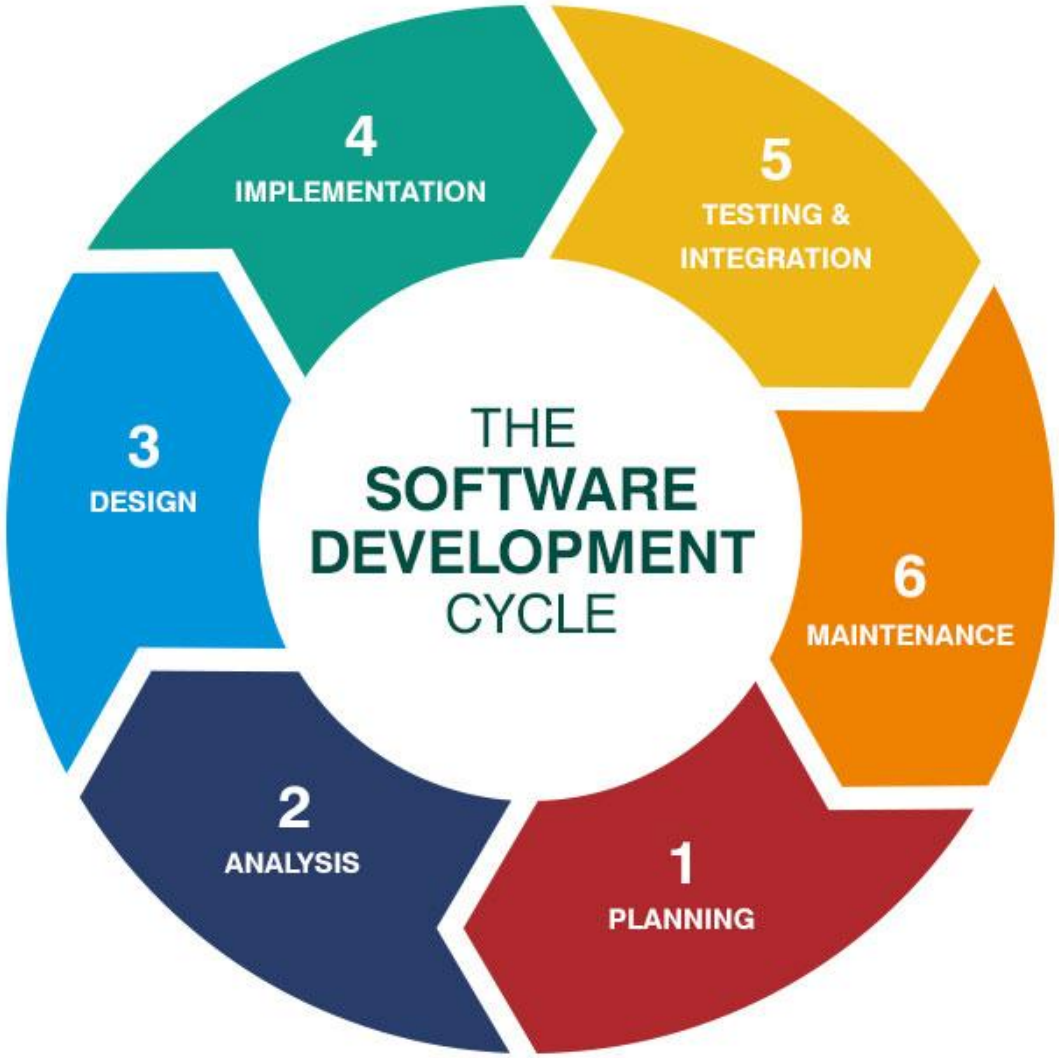
软件构造课：软件系统=构造复杂数据结构+对复杂数据结构的组合使用



Constituents of a software system: one more step



Constituents of a software system: two more steps



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><td>Build-time</td><td>Source code, AST, Interface-Class-Attribute-Method (Class Diagram)</td><td>Package, File, Static Linking, Library, Test Case, Build Script (Component Diagram)</td><td>Code Churn</td><td>Configuration Item, Version</td></tr>
<tr><td rowspan="2">Run-time</td><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 stack trace, Concurrent multi-threads</td><td>Event log, Multi-processes, Distributed processes</td></tr>
<tr><td colspan="2">Procedure Call Graph, Message Graph (Sequence Diagram)</td></tr>
</table>

(1) Build-time Views

Build-time views of a software system

- **Build-time (构造阶段): 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 代码的逻辑组织
  - **Component-level view**: architecture ---- how source code are physically organized by files, directories, packages, libraries, and the dependencies among them 代码的物理组织

  - **Moment view**: what do source code and component look like in a specific time 特定时刻的软件形态
  - **Period view**: how do they evolve/change along with time 软件形态随时间的变化

(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:**
  - 词汇层面：Lexical-oriented source code
  - 语法层面：Syntax-oriented program structure: e.g., Abstract Syntax Tree (AST)
  - 语义层面：Semantics-oriented program structure: e.g., Class Diagram

	Moment		Period	
	Code	Component	Code	Component
Build-time				
Run-time				

Lexical-based semi-structured source code

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

```
01 // First, log in
02 LoginResult loginResult=null;
03 SoapBindingStub sfdc=null;
04 sfdc = (SoapBindingStub) new SforceServiceLocator().getSoap();
05 // login
06 loginResult = sfdc.login("username","password");
07
08 // The set up some security related items
09 // Reset the SOAP endpoint to the returned server URL
10 sfdc._setProperty(SoapBindingStub.ENDPOINT_ADDRESS_PROPERTY,loginResult.getServerUrl());
11 // Create a new session header object
12 // add the session ID returned from the login
13 SessionHeader sh=new SessionHeader();
14 sh.setSessionId(loginResult.getSessionId());
15 sfdc.setHeader(new SforceServiceLocator().getServiceName().getNamespaceURI(),
16   "SessionHeader",sh);
17
18 // now that we're logged in, make some calls
19 GetUserInfoResult userInfo = sfdc.getUs
20
21 // create a new account object lo
22 Account account = new Account()
23 account.setAccountNumber("002D
24 account.setName("My New Accou
25 account.setBillingCity("Glasg
26
27
28 SObject[] sObjects = new SObjec
29 sObjects[0] = account;
30
31 // persist the object
32 SaveResult[] saveResults = sfdc.create(sObject
```

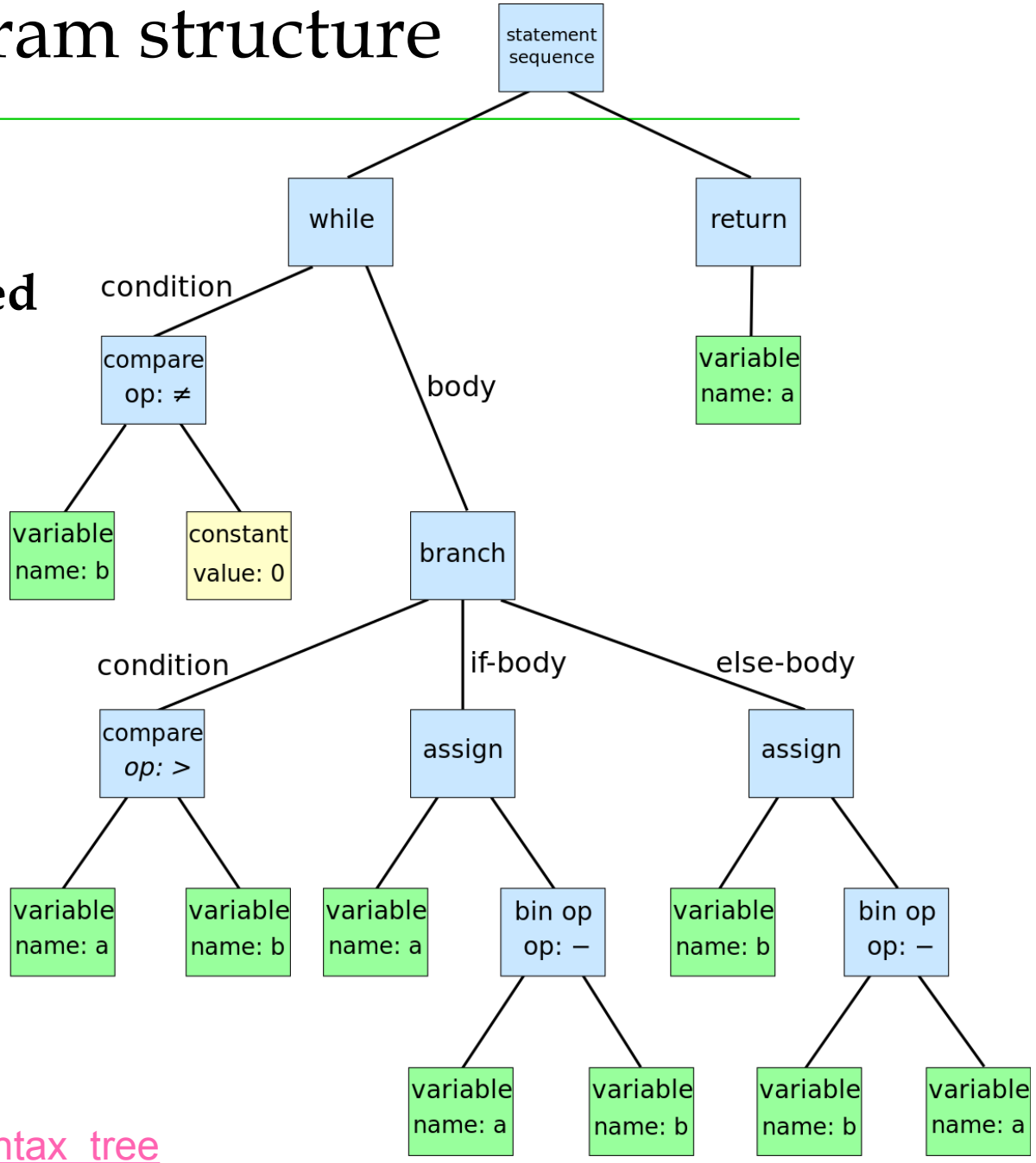
半结构化：近乎自然语言的风格+遵循特定的编程语法

前者：方便程序员
后者：方便编译器

Syntax-oriented program structure

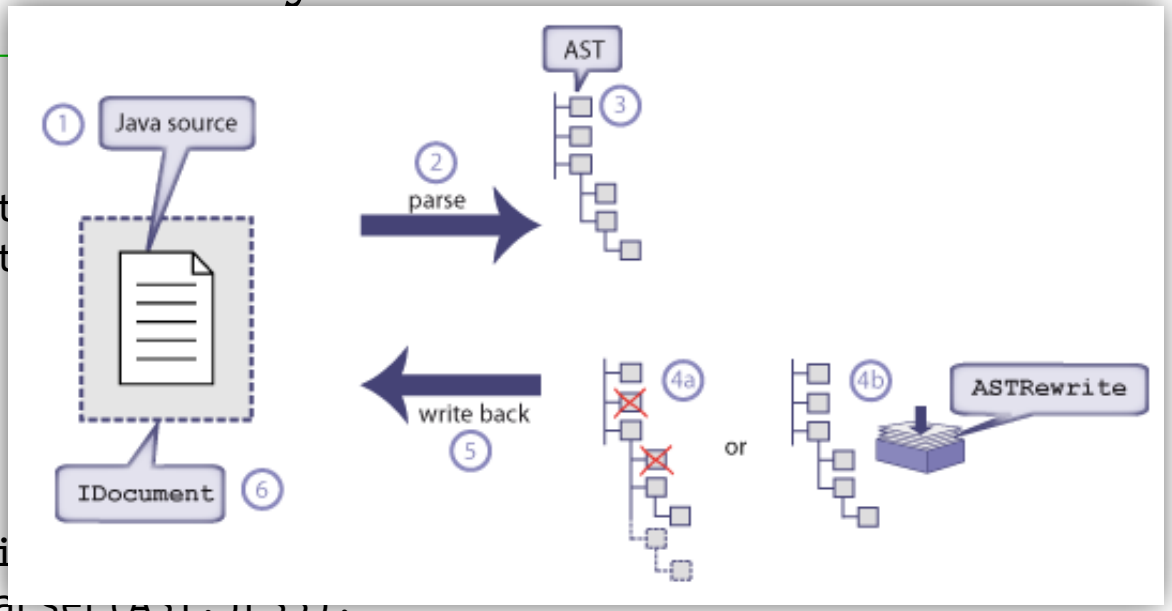
- **Abstract Syntax Tree (AST)**
- **To represent semi-structured source code as a structured tree.**

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



https://en.wikipedia.org/wiki/Abstract_syntax_tree
http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html

Processing Java Source Files by AST




```
import org.eclipse.jdt.core.dom.*;
import org.eclipse.jface.text.Document
import org.eclipse.text.edits.TextEdit

public class AST {

 void processJavaFileByAST(){

    Document doc = new Document(javaFi
    ASTParser parser = ASTParser.newPa
    parser.setResolveBindings(true);
    parser.setSource(doc.get().toCharArray());
    CompilationUnit cu = (CompilationUnit) parser.createAST(null);
    cu.recordModifications();

    AST ast = cu.getAST();
    ImportDeclaration id = ast.newImportDec
    id.setName(ast.newName(new String[] {"
    cu.imports().add(id); // add import dec
    TextEdit edits = cu.rewrite(doc, null);
 }
}
```

AST：彻底结构化，将源代码变为一棵树，对树做各种操作==对源代码的修改

http://www.vogella.com/tutorials/EclipseJDT/article.html

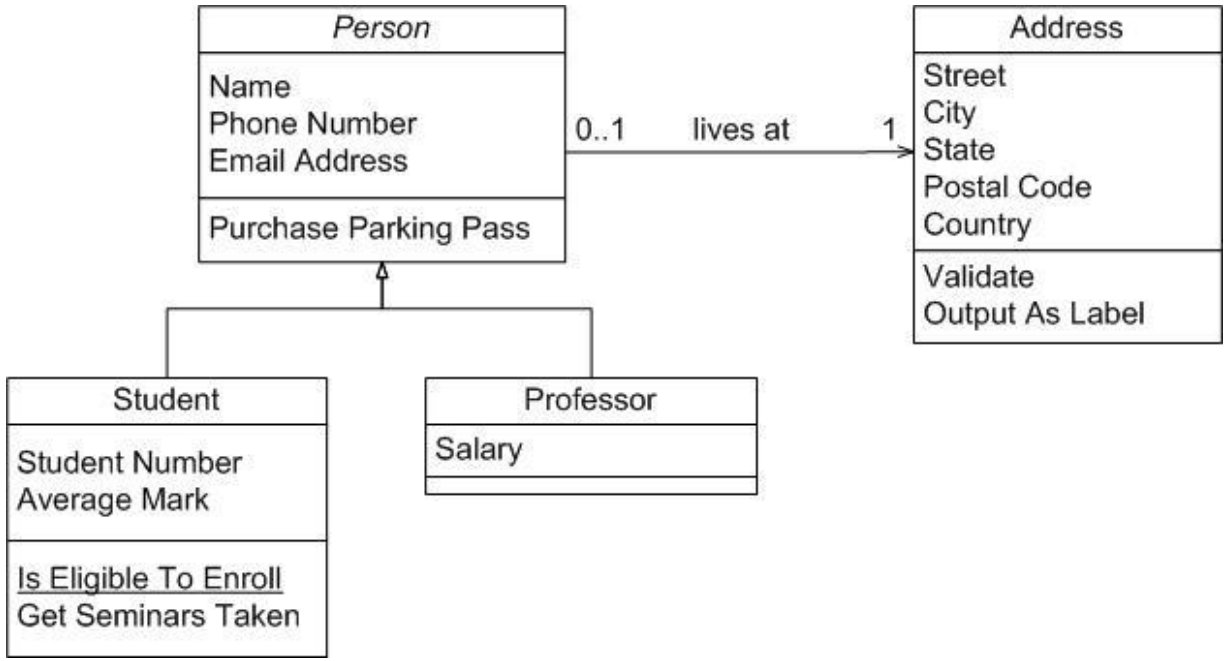
Semantics-oriented program structure

语义：源代码具体想实现什么目标？

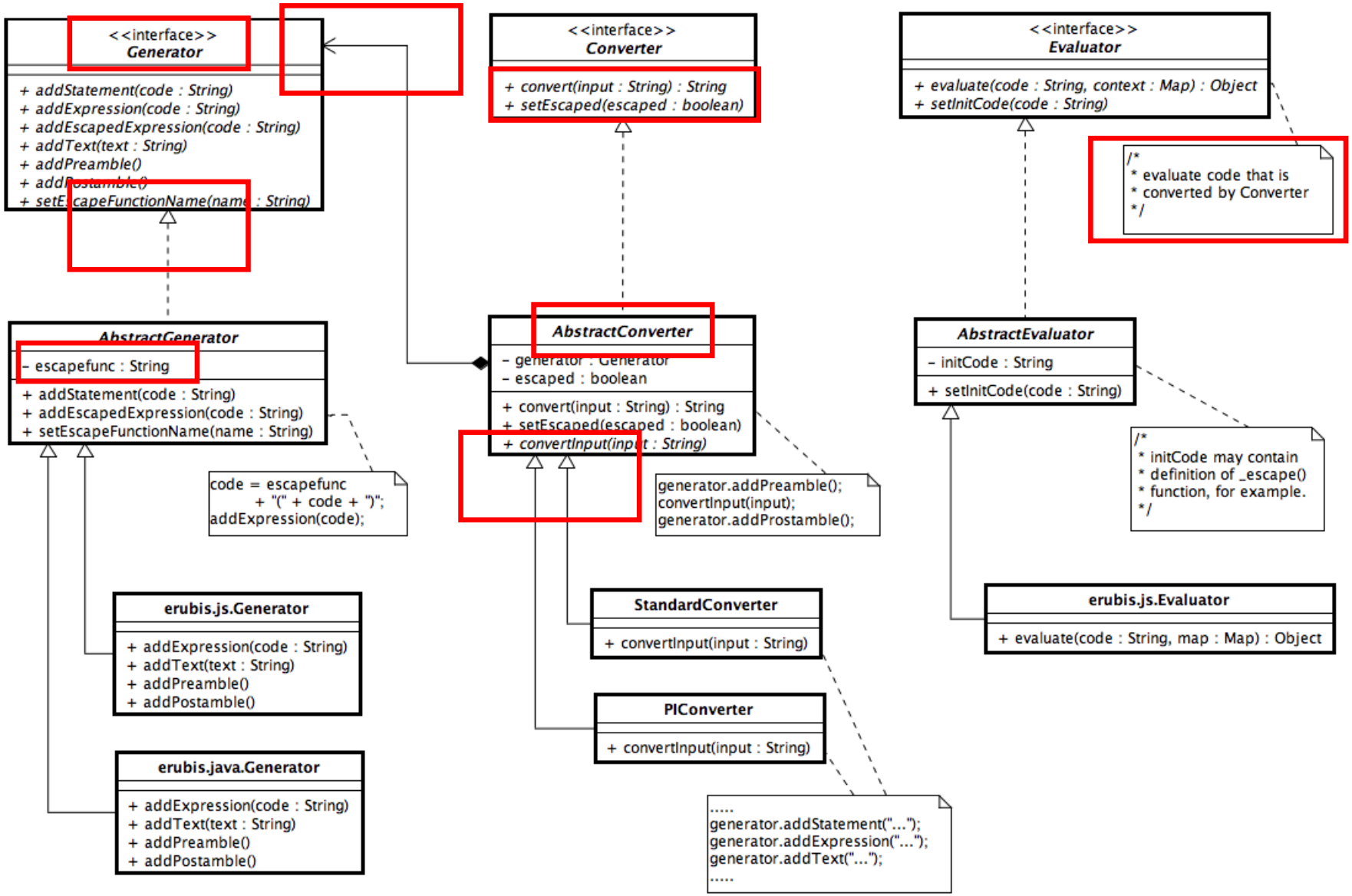
源代码---现实世界

- **E.g., using Class Diagram (UML) to describe interfaces, classes, attributes, methods, and relationships among them.**
- **Graphics-based or formally defined.** 通常是图形化或形式化的
- **Modeled in design phase, and transformed into source code.**
- **It is the result of Object-Oriented Analysis and Design in terms of user requirements.**

用于表达“需求”和“设计”思想，再转化成code



Semantics-oriented program structure



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

- **Views describing "changes" along with time.**
- **Code churn 代码变化: 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

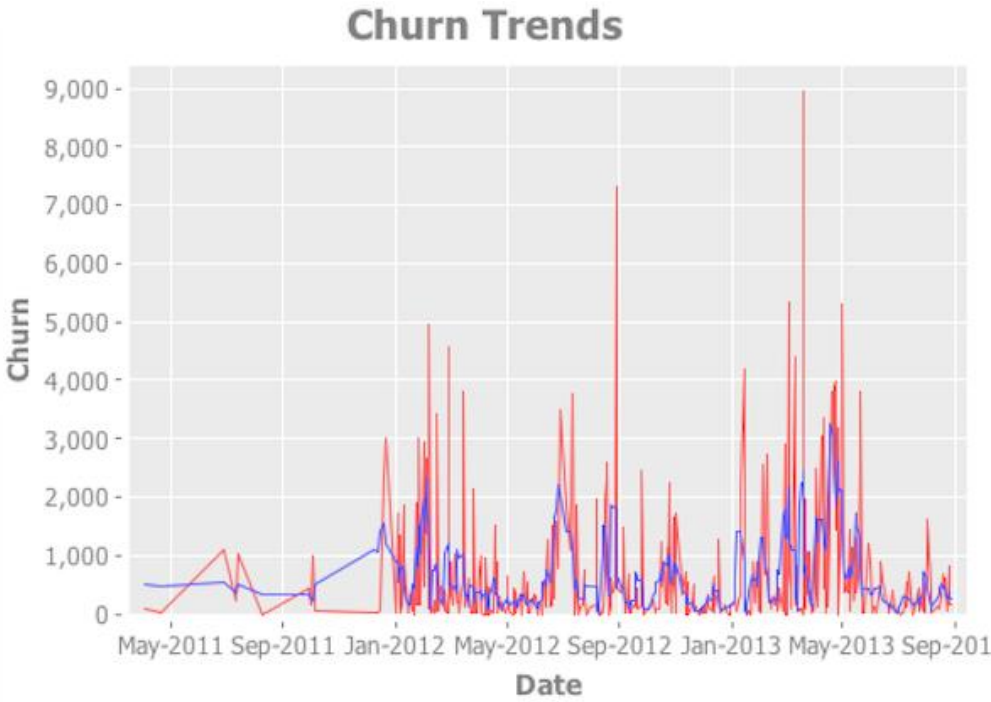
Code Before

```
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);
```

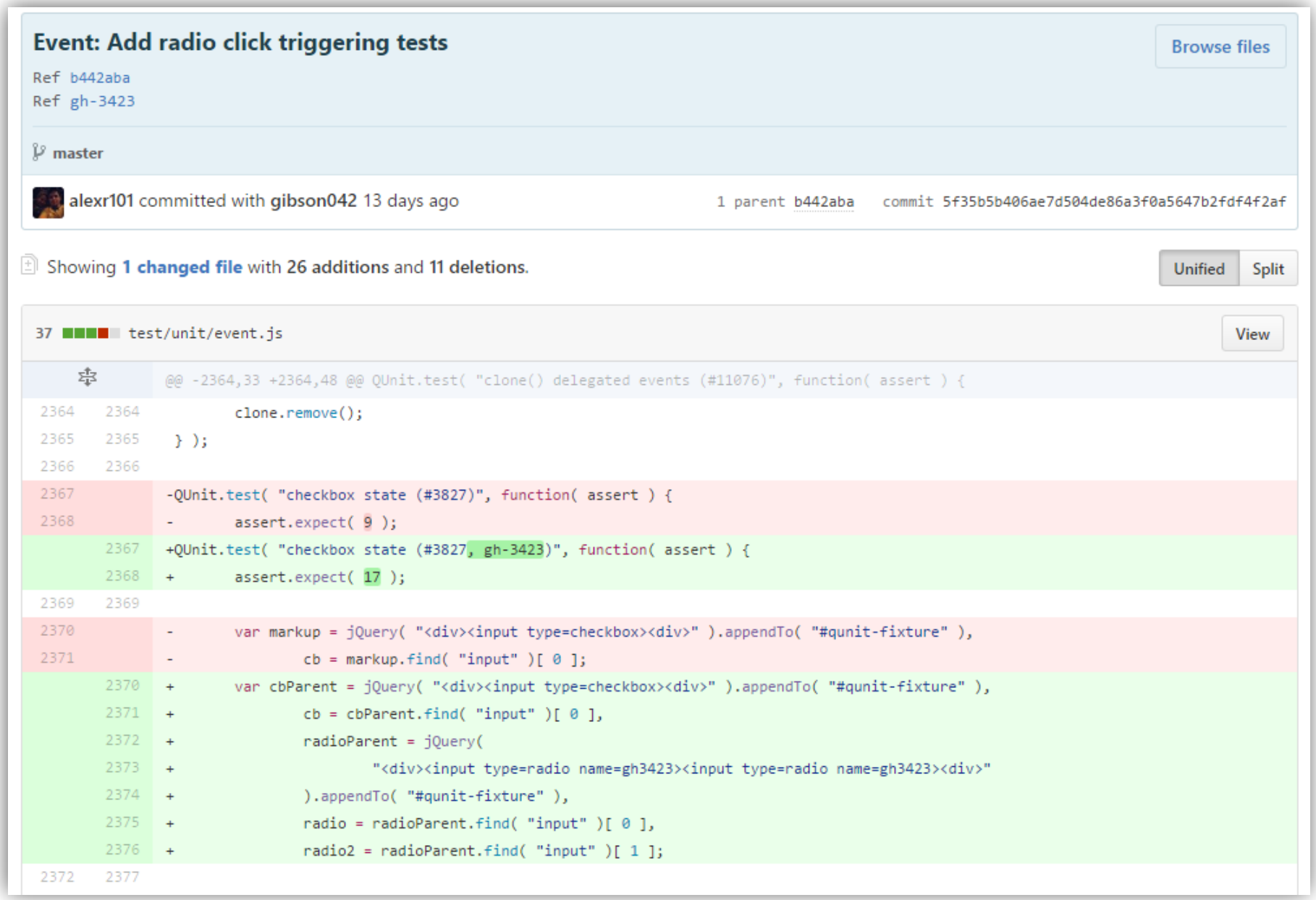
one line deleted



	Moment		Period	
	Code	Component	Code	Component
Build-time				
Run-time				

Code churn

Code churn is defined as lines added, modified or deleted to a file from one version to another.



Event: Add radio click triggering tests

Browse files

Ref b442aba
Ref gh-3423

master

alexr101 committed with gibson042 13 days ago — 1 parent b442aba — commit 5f35b5b406ae7d504de86a3f0a5647b2fdf4f2af

Showing 1 changed file with 26 additions and 11 deletions.

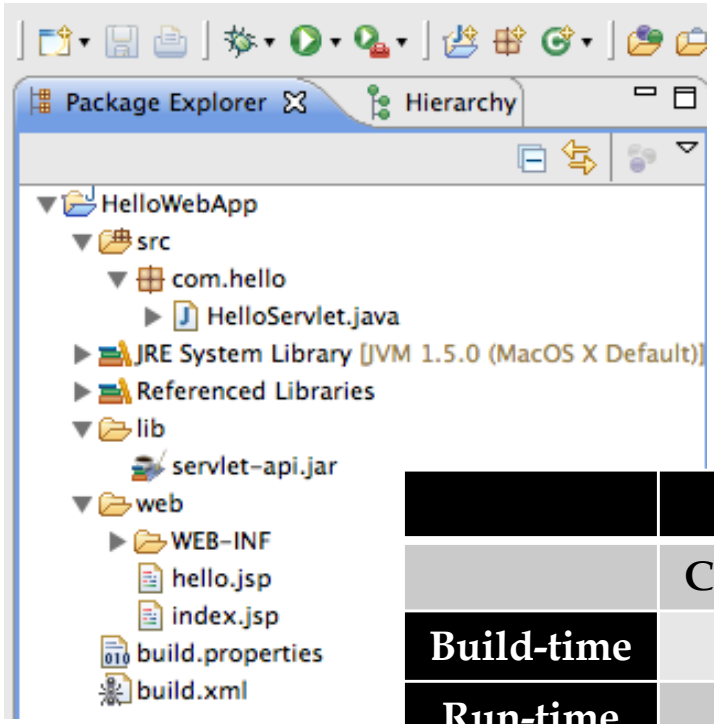
Unified | Split

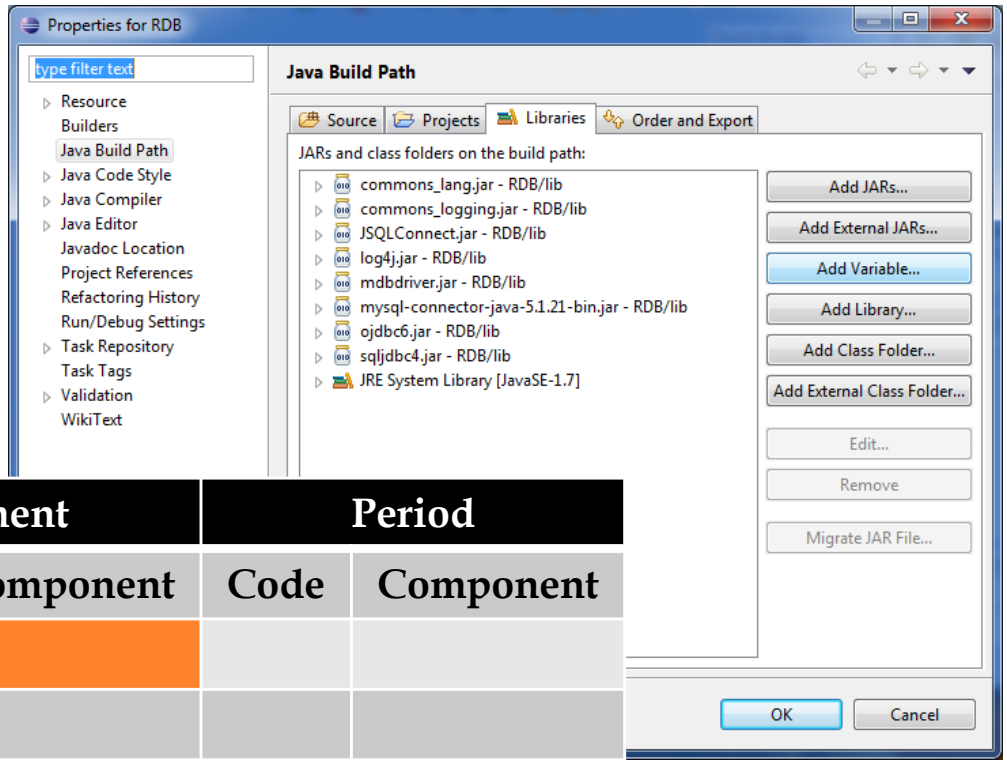
37 test/unit/event.js — View

```
@@ -2364,33 +2364,48 @@ QUnit.test( "clone() delegated events (#11076)", function( assert ) {
2364 2364         clone.remove();
2365 2365  } );
2366 2366
2367      -QUnit.test( "checkbox state (#3827)", function( assert ) {
2368      -       assert.expect( 9 );
     2367 +QUnit.test( "checkbox state (#3827, gh-3423)", function( assert ) {
     2368 +       assert.expect( 17 );
2369 2369
2370      -       var markup = jQuery( "<div><input type=checkbox><div>" ).appendTo( "#qunit-fixture" ),
2371      -               cb = markup.find( "input" )[ 0 ];
     2370 +       var cbParent = jQuery( "<div><input type=checkbox><div>" ).appendTo( "#qunit-fixture" ),
     2371 +               cb = cbParent.find( "input" )[ 0 ],
     2372 +               radioParent = jQuery(
     2373 +                       "<div><input type=radio name=gh3423><input type=radio name=gh3423><div>"
     2374 +               ).appendTo( "#qunit-fixture" ),
     2375 +               radio = radioParent.find( "input" )[ 0 ],
     2376 +               radio2 = radioParent.find( "input" )[ 1 ];
2372 2377
```

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

- **Source code are physically organized into files which further are organized by directories;**
- **Files are encapsulated into packages and, logically, components and sub-systems.**
- **Reusable modules are in the form of libraries.**





	Moment		Period	
	Code	Component	Code	Component
Build-time				
Run-time				

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. 开发者像使用编程语言指令一样使用库中的功能

```
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; 操作系统提供的库
  - From language SDK; 编程语言提供的库
  - From third-party sources; 第三方公司提供的库
  - Developers can also publish their own libraries. 你自己积累的库

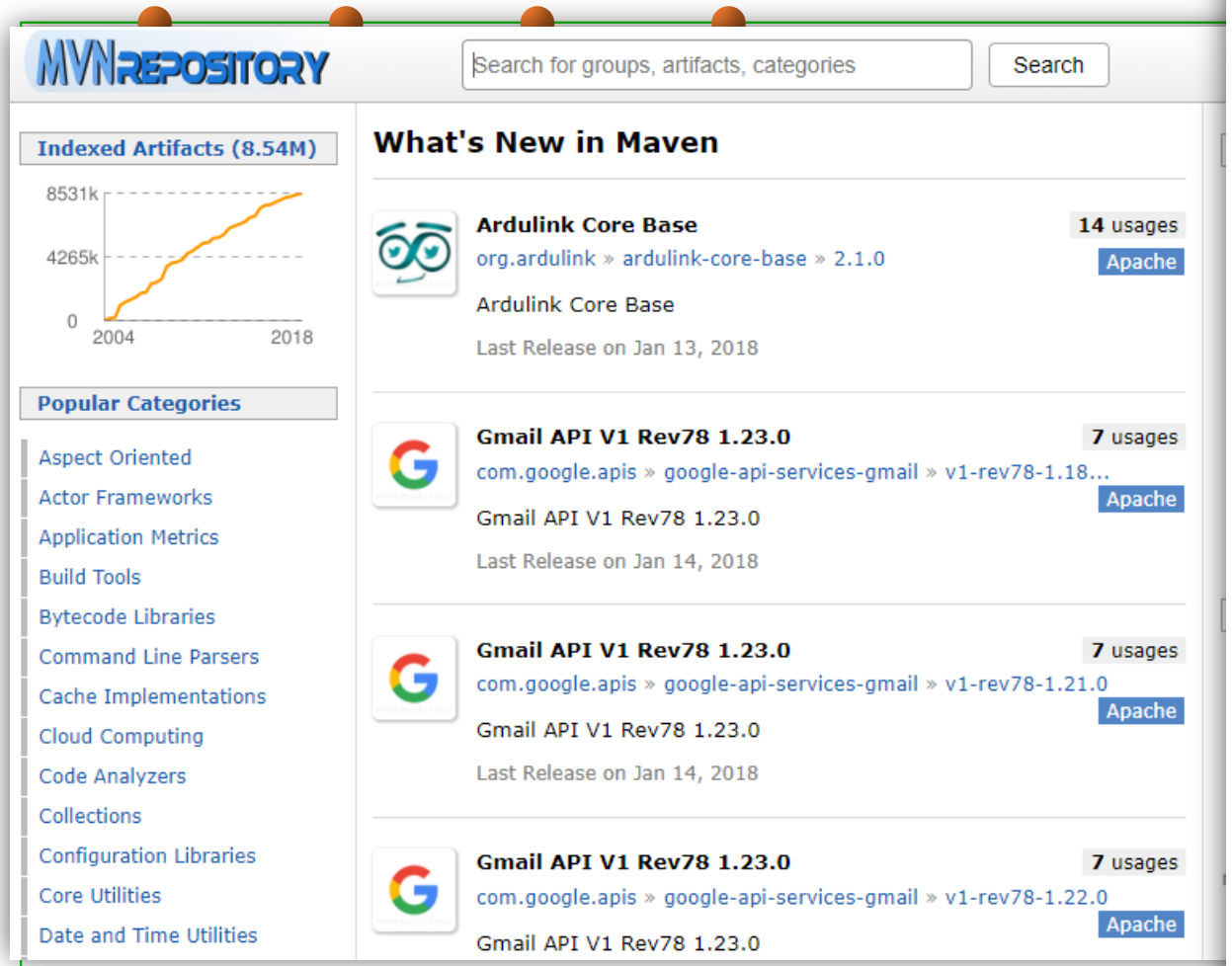
何谓“更高级”的编程语言

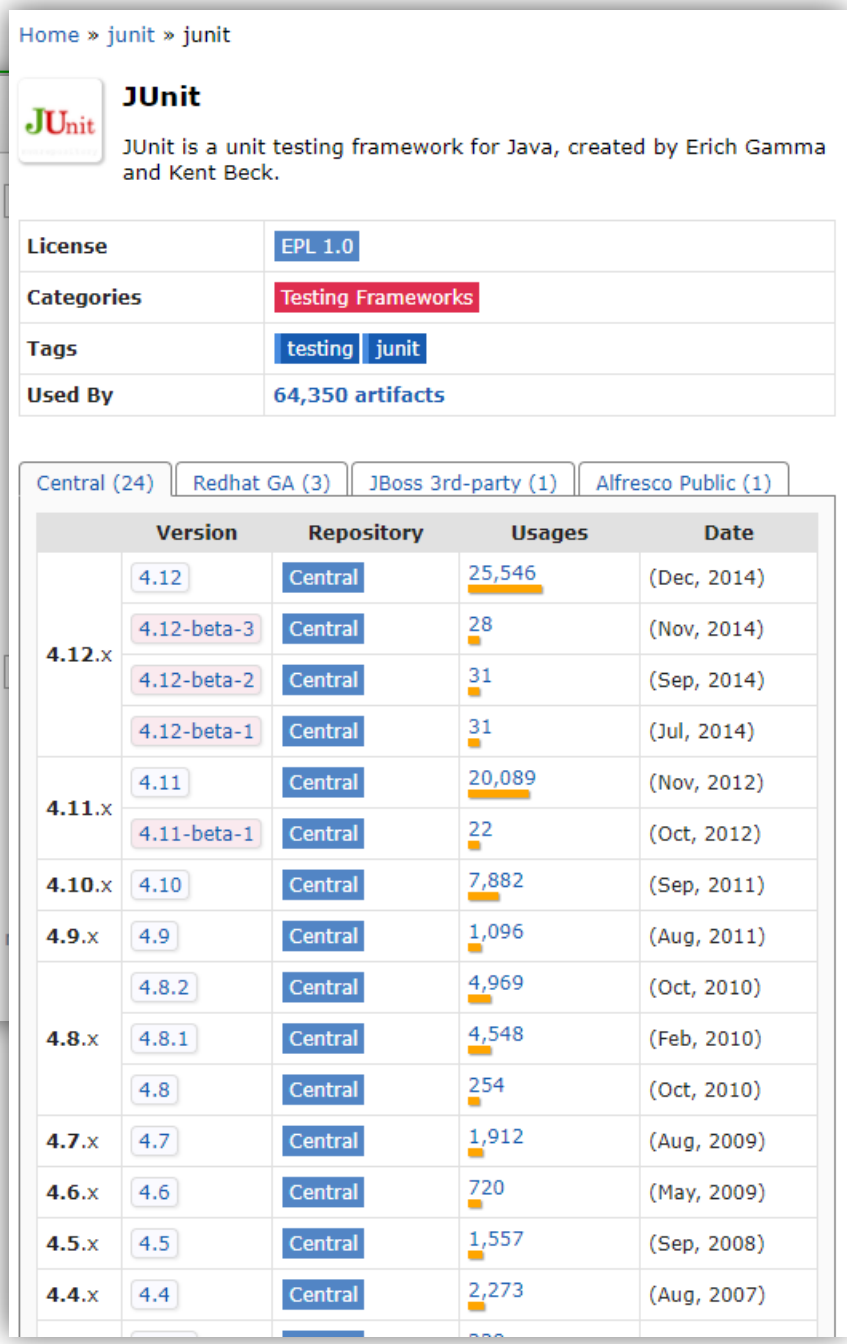
- **一个编程语言比其他编程语言“更高级”，意即它提供了基本数据类型（int、char等）和基本编程结构（顺序、分支、循环）之外，还提供更多复杂的、更贴近实际应用需求的数据结构及作用于这些复杂数据结构上的算法。**

- **换句话说：所提供的预设的库更丰富。**
  - 库：提供了一系列可供开发者直接使用的复杂数据结构和复杂算法。

- **对比一下C和C++、Java、Python**

Maven Repository






```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

Linking with a library

- **When a program is edited, built and installed, a list of libraries to search must be provided. 编程时和build时，需告诉IDE和JVM在哪里寻找某些库**
- **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.**

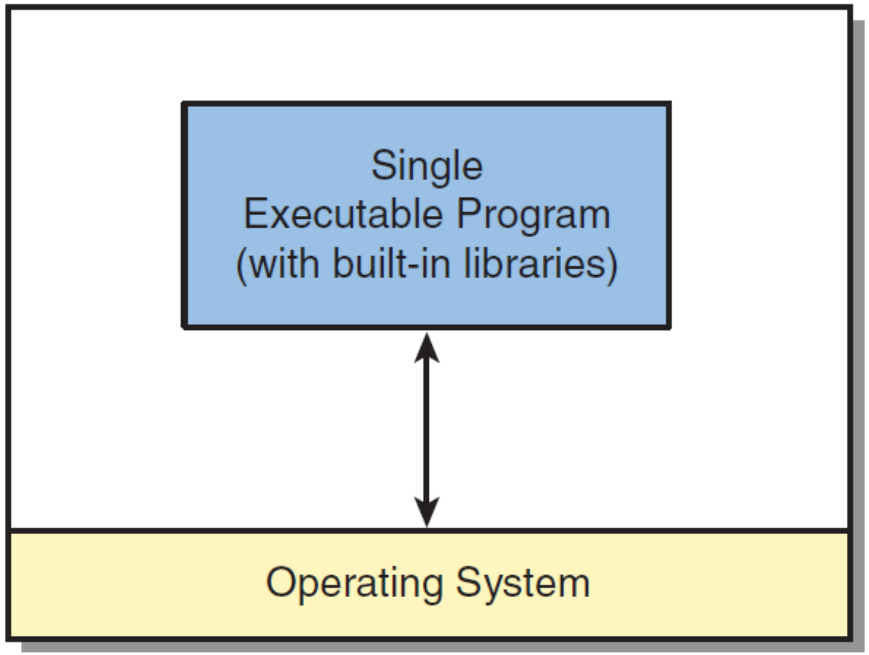
```
javac -classpath  ./lib/*.jar
```

- **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 (静态链接)
  - Dynamic linking (动态链接)

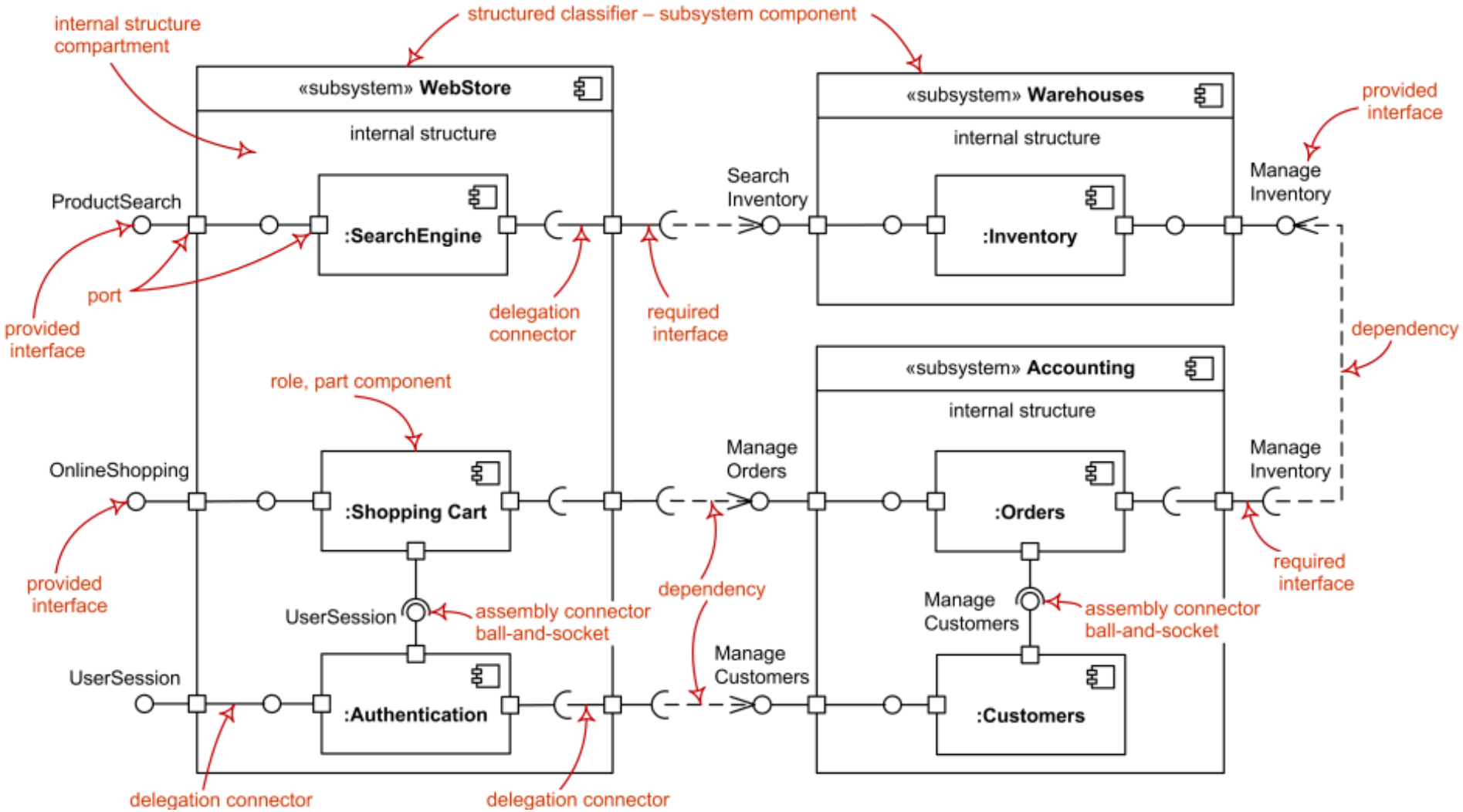
Static linking

- **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**
  - 库被拷贝进入代码形成整体，执行的时候无需提供库文件
  - 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**
  - 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.

静态链接发生在构造阶段

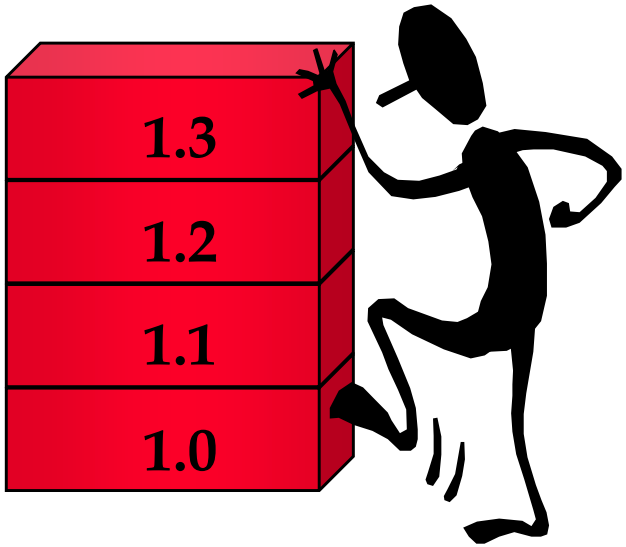


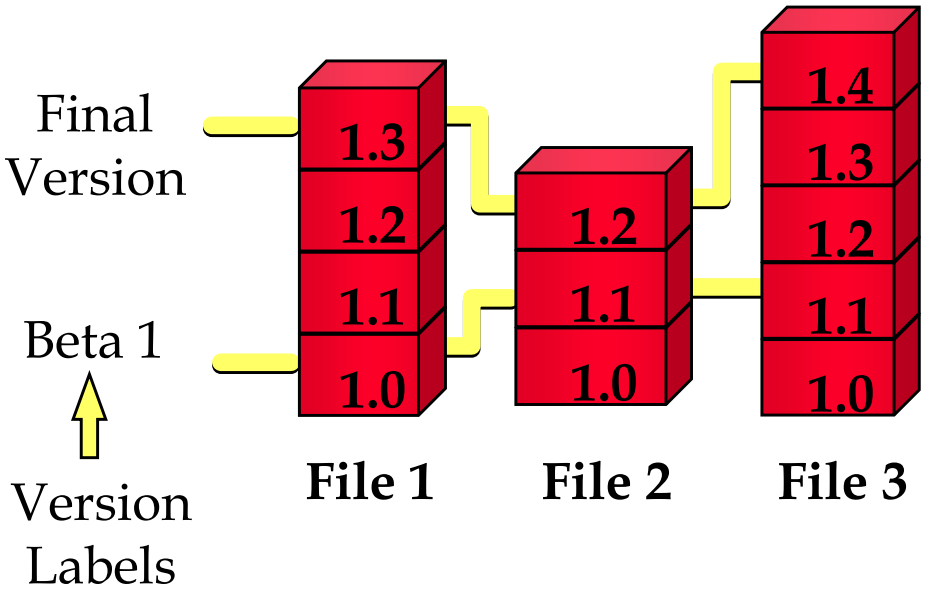
Component diagram in UML



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

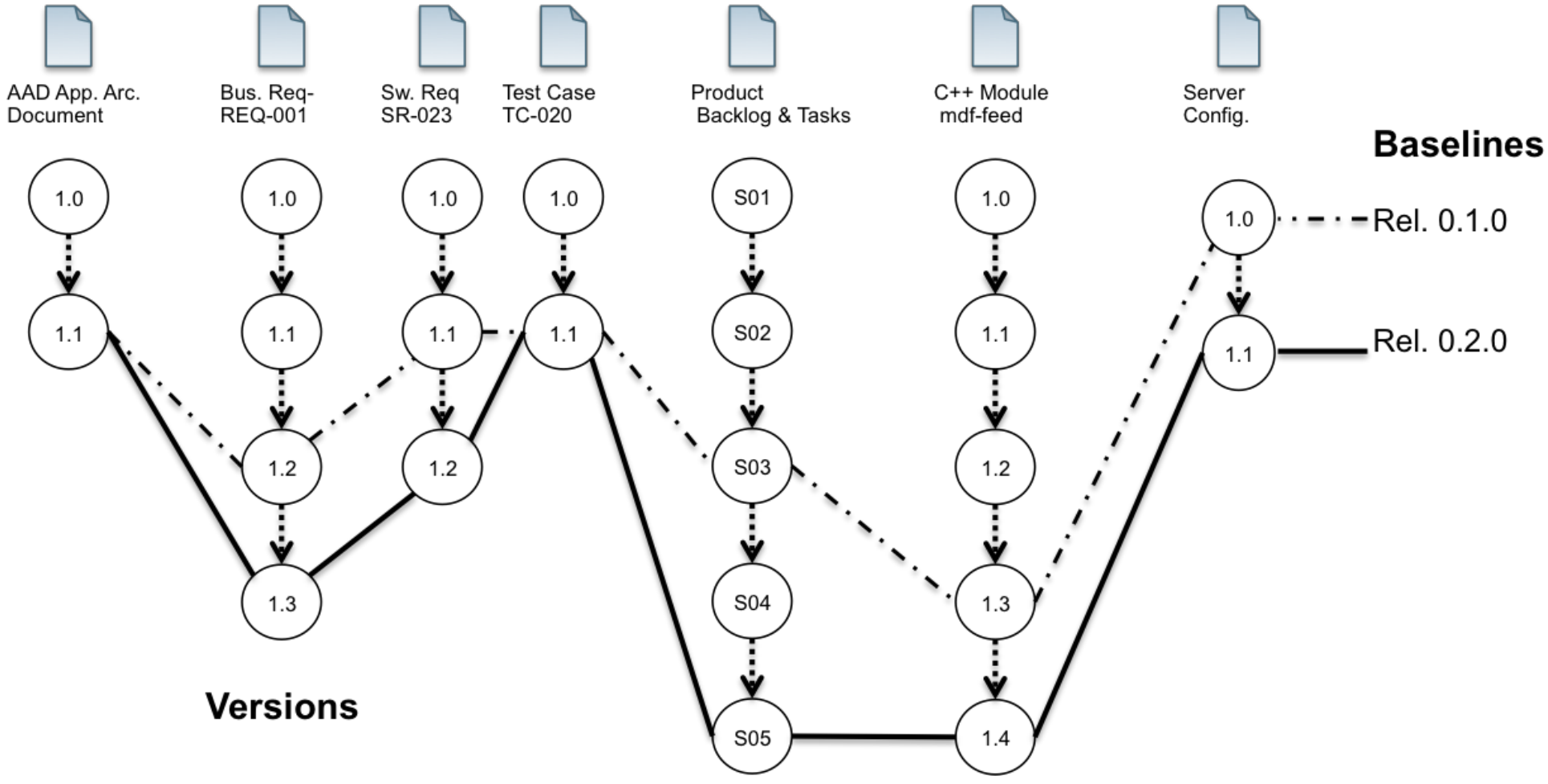
- **How do all files/packages/components/libraries change in a software system along with time? 各项软件实体随时间如何变化？**
- **Software Configuration Item (SCI，配置项)**
- **Version (版本)**



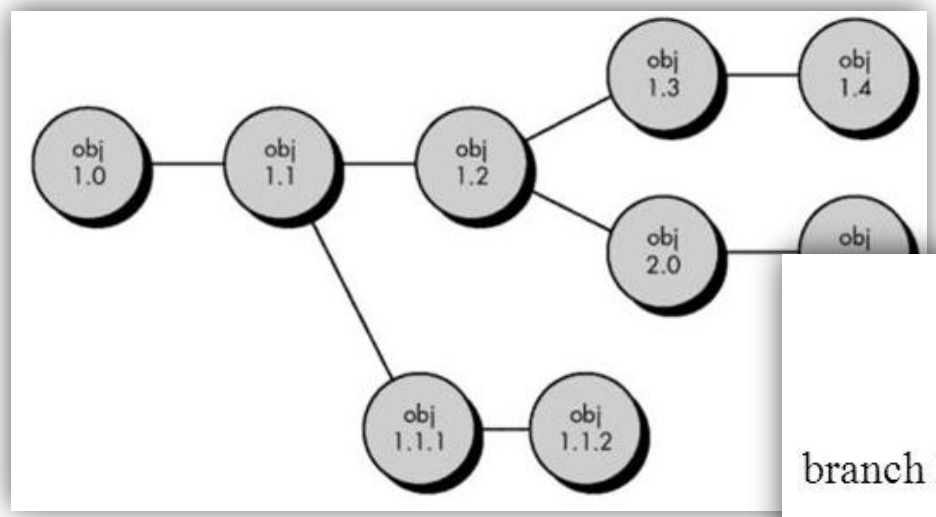


	Moment		Period	
	Code	Component	Code	Component
Build-time				
Run-time				

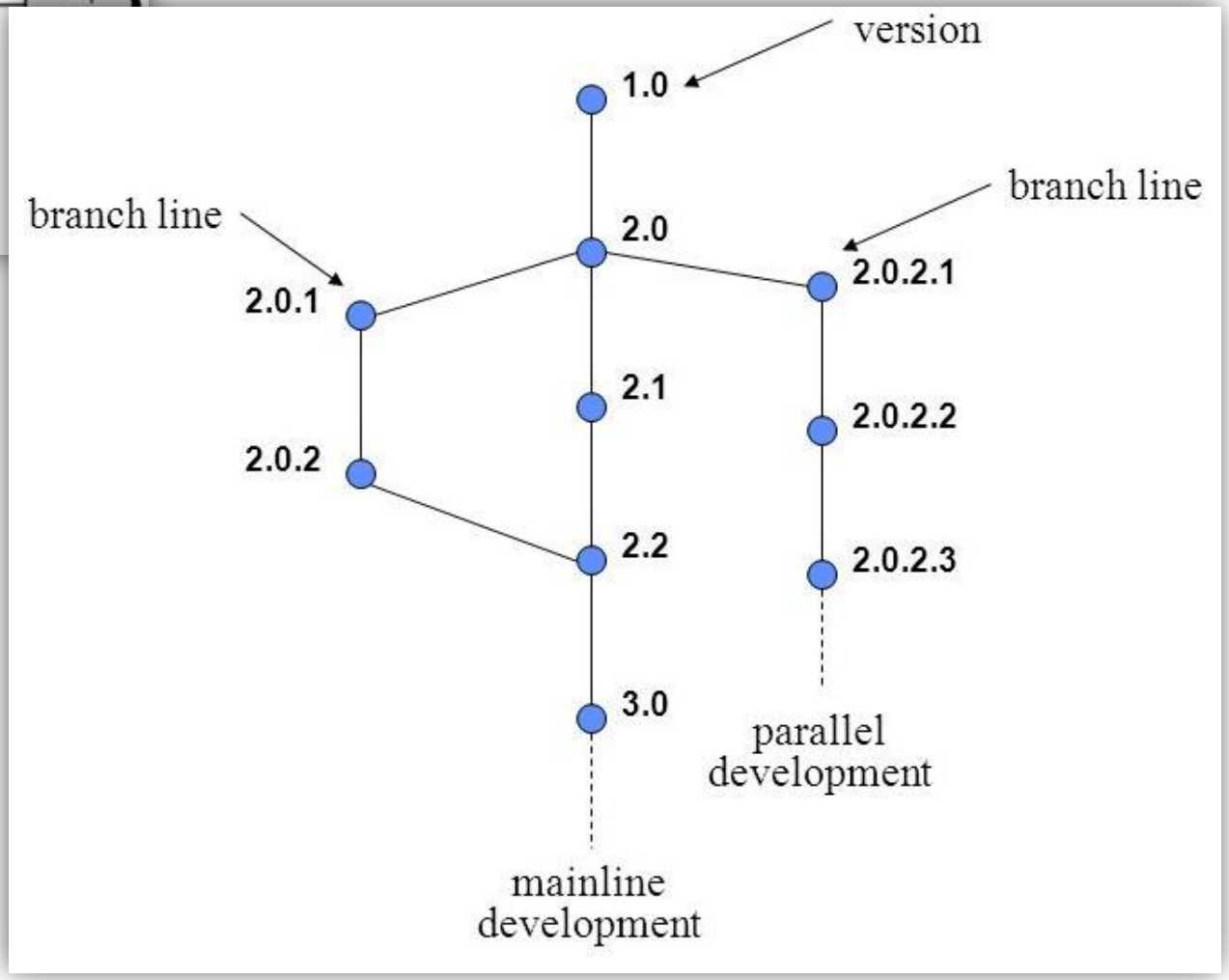
Version Control System (VCS)



Evolution Graph (of a SCI or a Software)

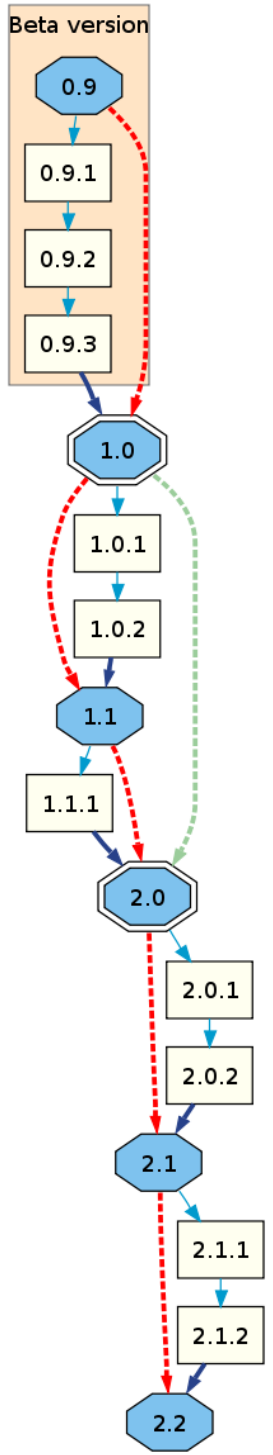


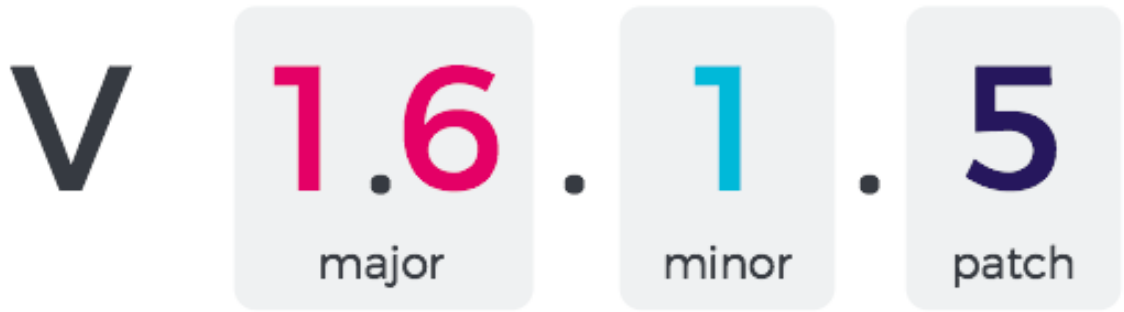
Evaluation graph describes the development process and phases of a software.



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.





(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? 运行时：程序被载入目标机器，开始执行
  - **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? 代码层面：逻辑实体在内存中如何呈现？
  - **Component-level view**: architecture ---- how are software packages deployed into physical environment (OS, network, hardware, etc) and how do they interact? 构件层面：物理实体在物理硬件环境中如何呈现？

  - **Moment view**: how do programs behave in a specific time 逻辑/物理实体在内存/硬件环境中特定时刻的形态如何？
  - **Period view**: how do they behave along with time 逻辑/物理实体在内存/硬件环境中的形态随时间如何变化？

High-level concepts of run-time software

- **Executable programs**: The sequence of machine-readable instructions that the CPU executes, along with associated data values.
  - This is the fully compiled program that's ready to be loaded into the computer's memory and executed.

- **Libraries**: Collections of commonly used object code that can be reused by different programs.
  - Most operating systems include a standard set of libraries that developers can reuse, instead of requiring each program to provide their own.
  - A library can't be directly loaded and executed on the target machine; it must first be linked with an executable program.

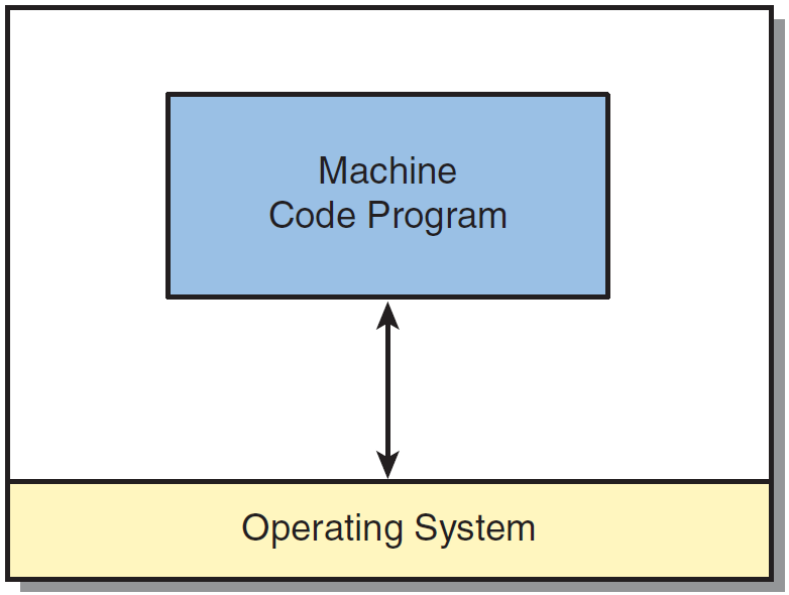
High-level concepts of run-time software

- **Configuration and data files:** These are not executable files; they provide useful data and configuration information that the program can load from disk.

- **Distributed programs:** This type of software consists of multiple executable programs that communicate with each other across a network or simply as multiple processes running on the same machine.
  - This contrasts with more traditional software that has a single monolithic program image.

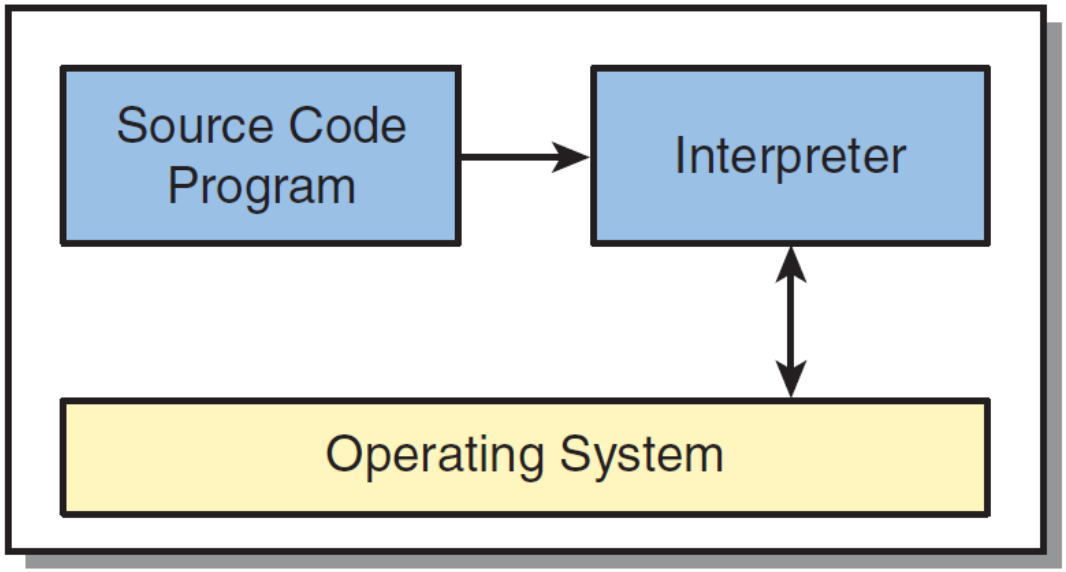
Executable Programs: Native Machine Code

- 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 (原生机器码):**
  - 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.



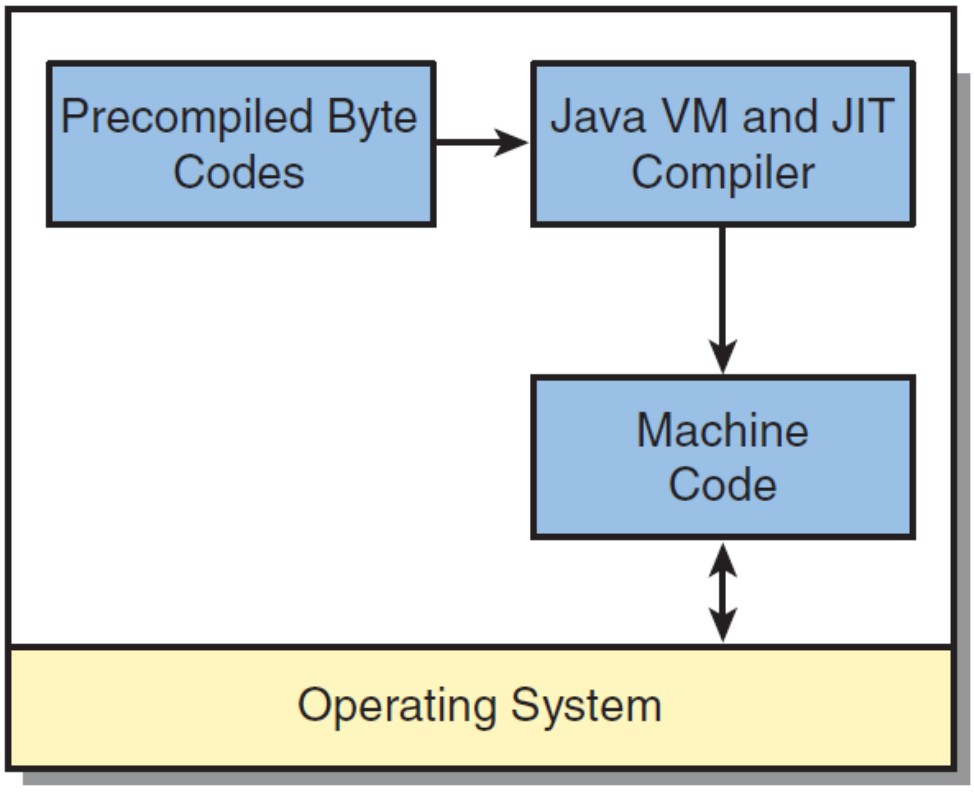
Executable Programs: Full Interpretation

- **Full Program Interpretation (程序完全解释执行):**
  - The runtime system loads the entire source code into memory and interprets it (such as BASIC, UNIX shell, etc)



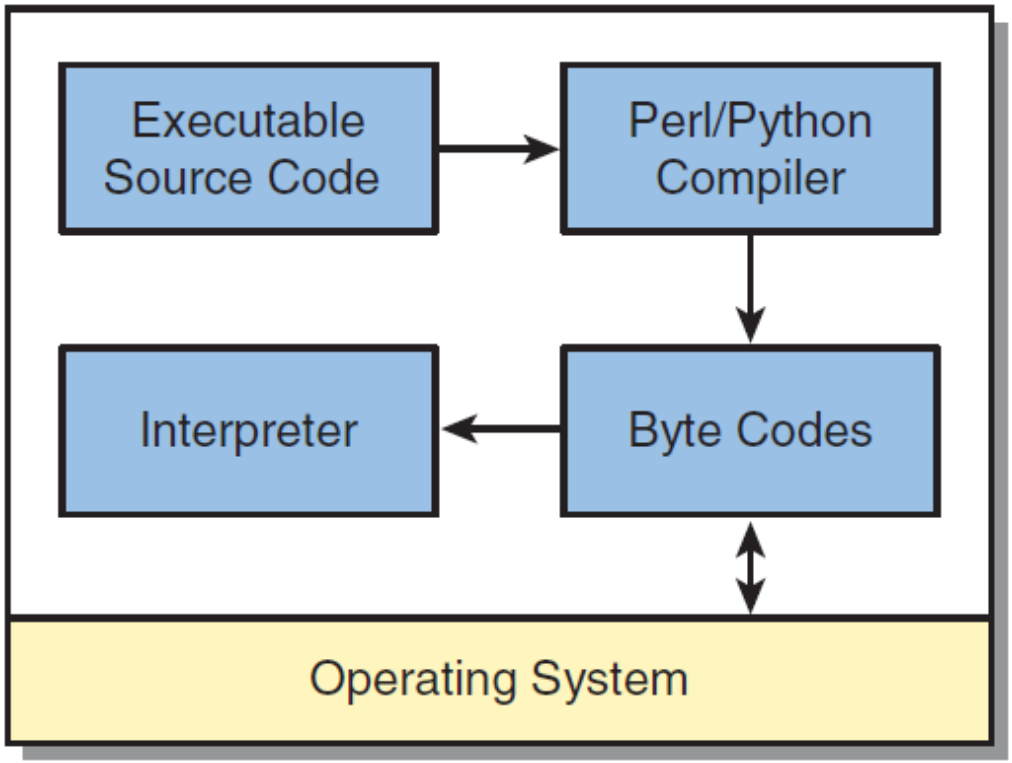
Executable Programs: Interpreted Byte Codes

- **Interpreted Byte Codes (解释型字节码):**
  - 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.**



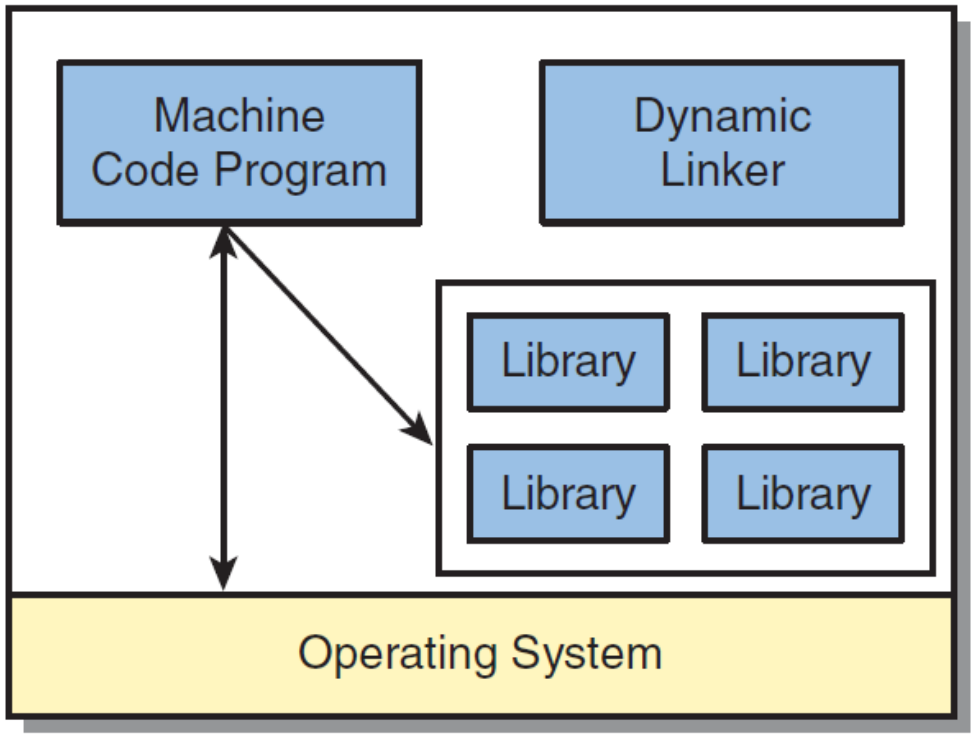
Dynamic linking

- **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. 库文件不会在**build**阶段被加入可执行软件，仅仅做出标记
- When the program starts running, th**e libraries are loaded into memory as separate entities** and then are connected with the main program. 程序运行时，根据标记装载库至内存
- 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. 发布软件时，记得将程序所依赖的所有动态库都复制给用户

你的实验，提交到GitHub的时候，切记把各种lib都一并提交上来

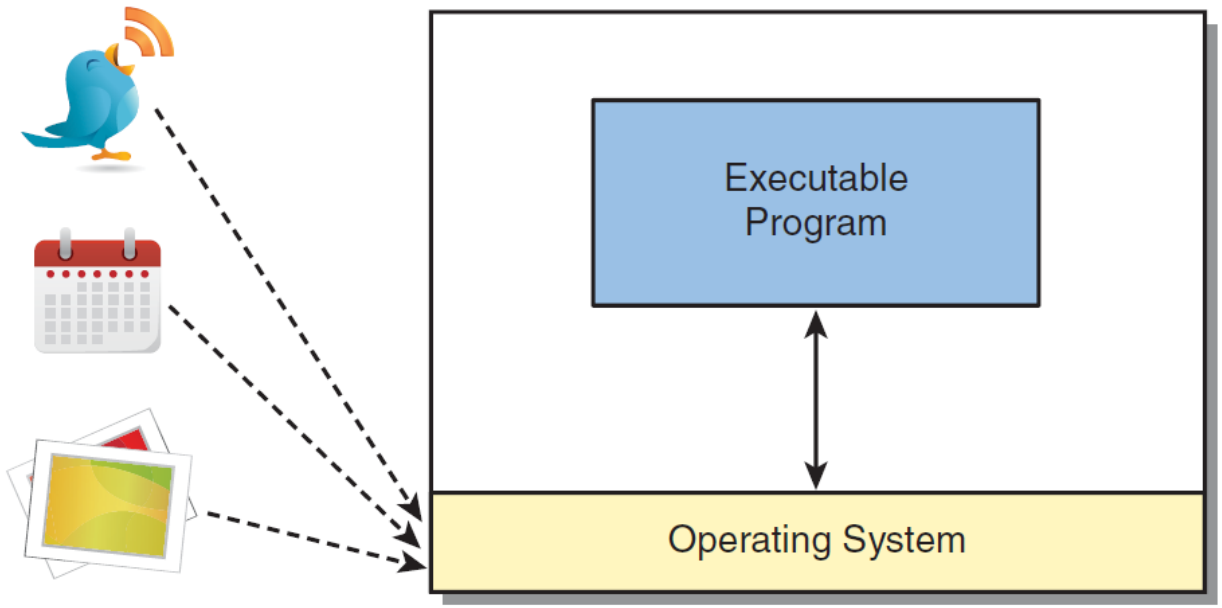
Dynamic linking

- **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



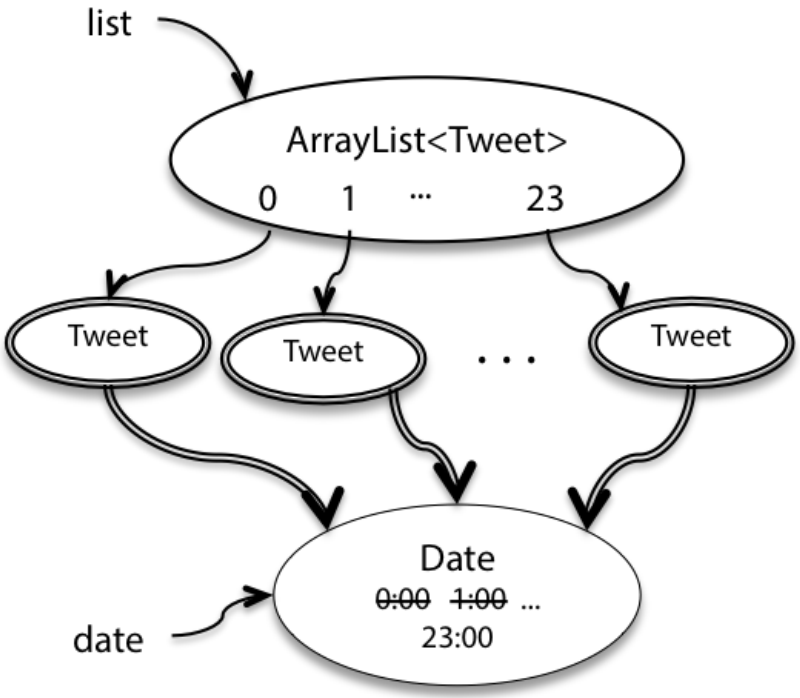
Distributed Programs

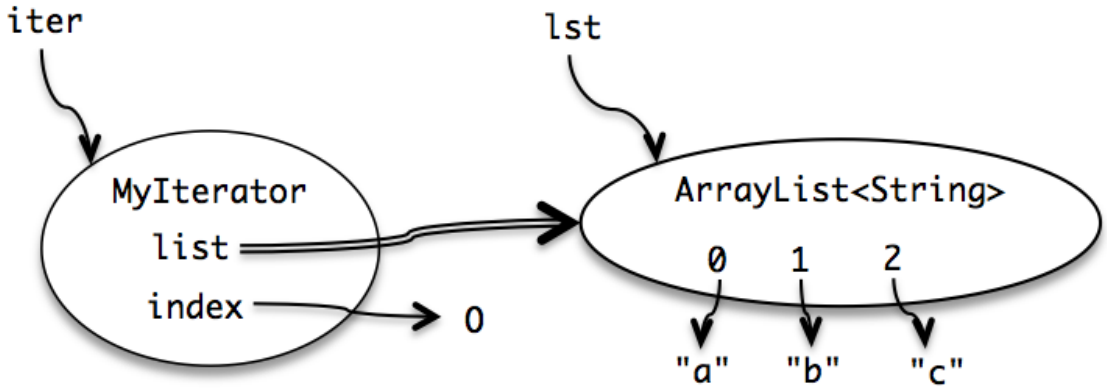
- For example, a software system might use the client/server model, with a single server program running on one computer and a large number of client programs running on many other computers.
- In this scenario, the build system could create two release packages, given that different people will be installing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

- 分布式程序的运行态：需要多个运行程序，分别编译和部署于多个计算机物理环境。

(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 of a program.**
- **代码快照图：描述程序运行时内存里变量层面的状态**

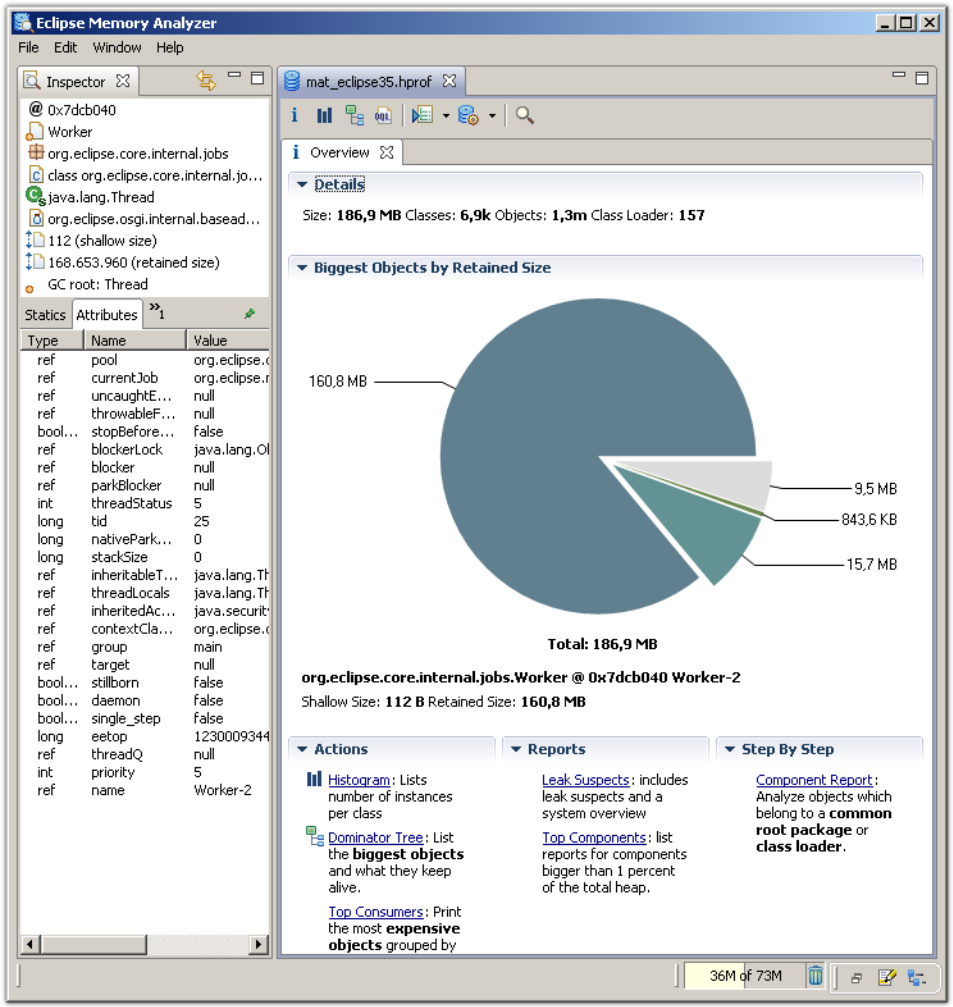




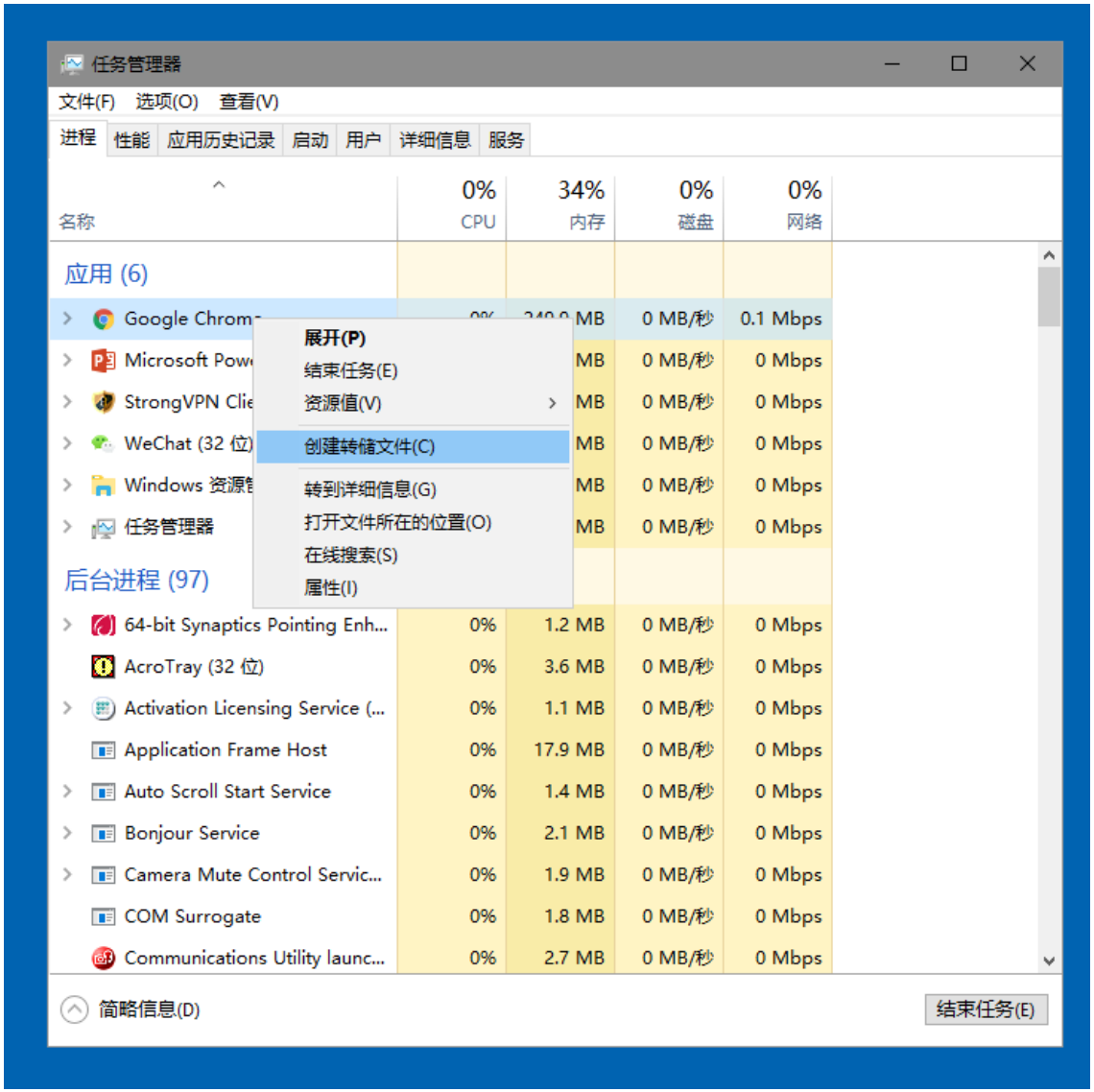
	Moment		Period	
	Code	Component	Code	Component
Build-time				
Run-time				

Memory dump (内存信息转储)

- **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.



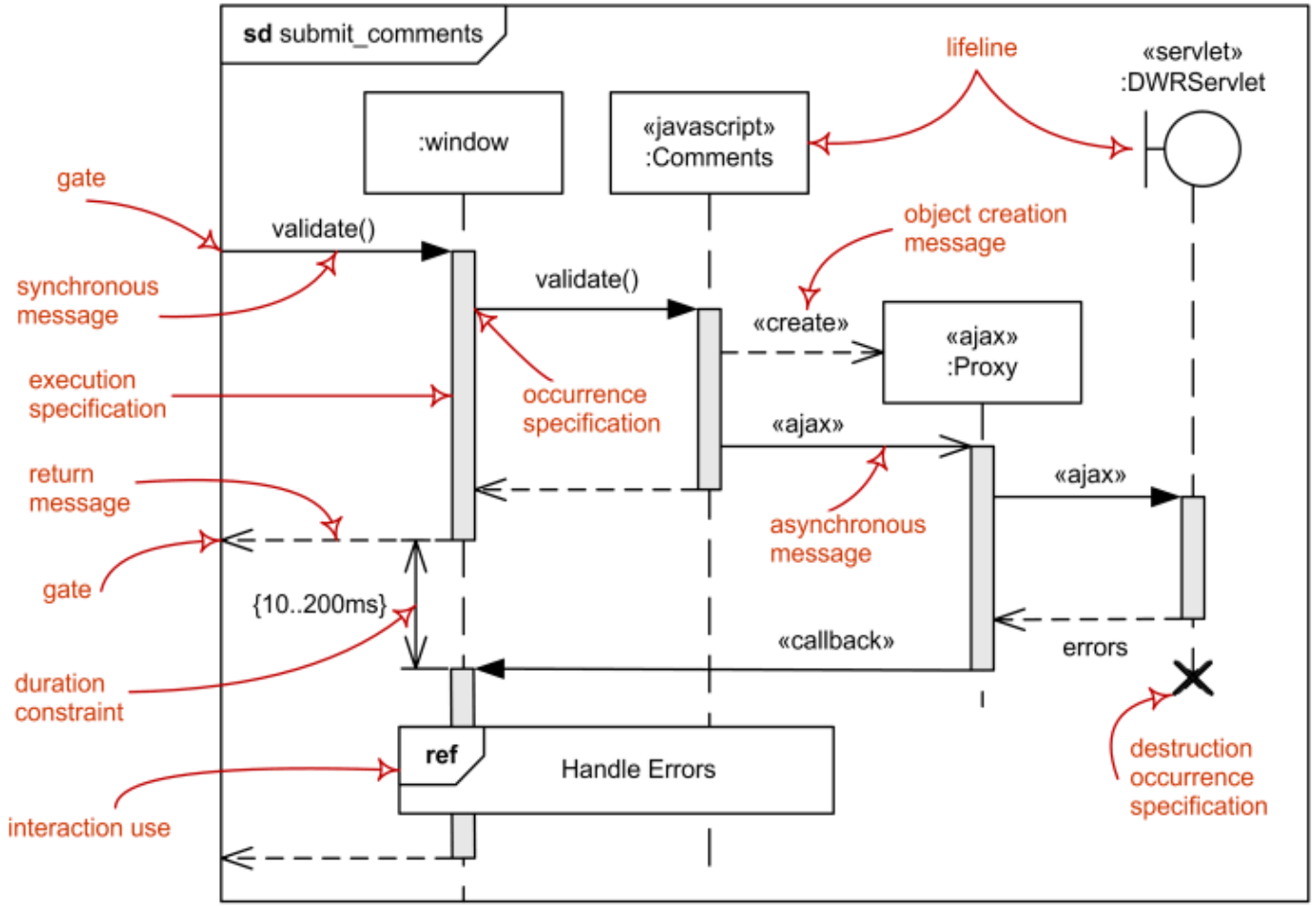
Memory dump


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 资源管
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... 0% 1.2 MB 0 MB/秒 0 Mbps
AcroTray (32 位) 0% 3.6 MB 0 MB/秒 0 Mbps
Activation Licensing Service (... 0% 1.1 MB 0 MB/秒 0 Mbps
Application Frame Host 0% 17.9 MB 0 MB/秒 0 Mbps
Auto Scroll Start Service 0% 1.4 MB 0 MB/秒 0 Mbps
Bonjour Service 0% 2.1 MB 0 MB/秒 0 Mbps
Camera Mute Control Servic... 0% 1.9 MB 0 MB/秒 0 Mbps
COM Surrogate 0% 1.8 MB 0 MB/秒 0 Mbps
Communications Utility launc... 0% 2.7 MB 0 MB/秒 0 Mbps
0 MB/秒 0.1 Mbps
0 MB/秒 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)**

	Moment		Period	
	Code	Component	Code	Component
Build-time				
Run-time				



Execution tracing 执行跟踪

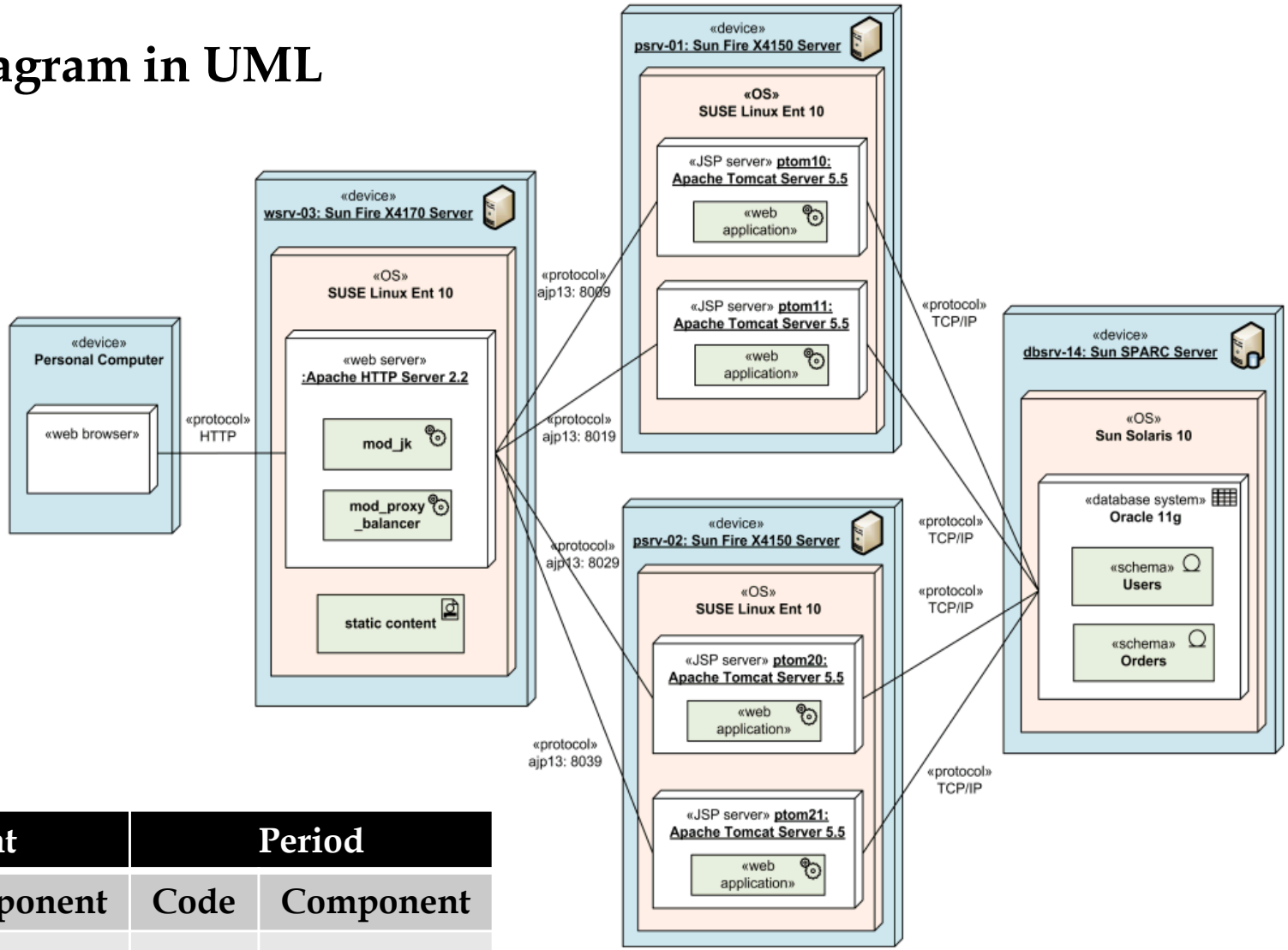
- **Tracing involves a specialized use of logging to record information about a program's execution.** 用日志方式记录程序执行的调用次序
  - This is typically used by programmers for debugging purposes, and depending on the type and detail of information contained in a trace log, by experienced administrators or technical-support personnel and by software monitoring tools to diagnose common problems with software.

```
09-26 10:59:38.056  28584-28584/
        java.lang.NullPointerExc
        at android.content.Conte
        at android.widget.Toast.
        at android.widget.Toast.
        at com.app.app.MainActiv
        at android.os.Handler.ha
        at android.os.Handler.di
        at android.os.Looper.loo
        at android.app.ActivityT
        at java.lang.reflect.Met
        at java.lang.reflect.Met
        at com.android.internal.
        at com.android.internal.
        at dalvik.system.NativeS
```

```
    0  java.lang.RuntimeException
!   1  at com.crittercism.testapp.errors.CustomError.crash(CustomError.java:57)
    2  at com.crittercism.testapp.fragments.ErrorFragment$4.onClick(ErrorFragment.java:150)
    3  at android.view.View.performClick(View.java:4438)
    4  at android.view.View$PerformClick.run(View.java:18422)
    5  at android.os.Handler.handleCallback(Handler.java:733)
    6  at android.os.Handler.dispatchMessage(Handler.java:95)
    7  at android.os.Looper.loop(Looper.java:136)
    8  at android.app.ActivityThread.main(ActivityThread.java:5001)
    9  at java.lang.reflect.Method.invokeNative(Native Method)
   10  at java.lang.reflect.Method.invoke(Method.java:515)
   11  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:785)
   12  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:601)
   13  at de.robv.android.xposed.XposedBridge.main(XposedBridge.java:132)
   14  at dalvik.system.NativeStart.main(Native Method)
   15  Caused by: java.lang.NullPointerException
   16  at com.crittercism.testapp.errors.NullPointerCustomError.performError(NullPointerCustomError.java:10)
   17  at com.crittercism.testapp.errors.CustomError.initiateError(CustomError.java:83)
```

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

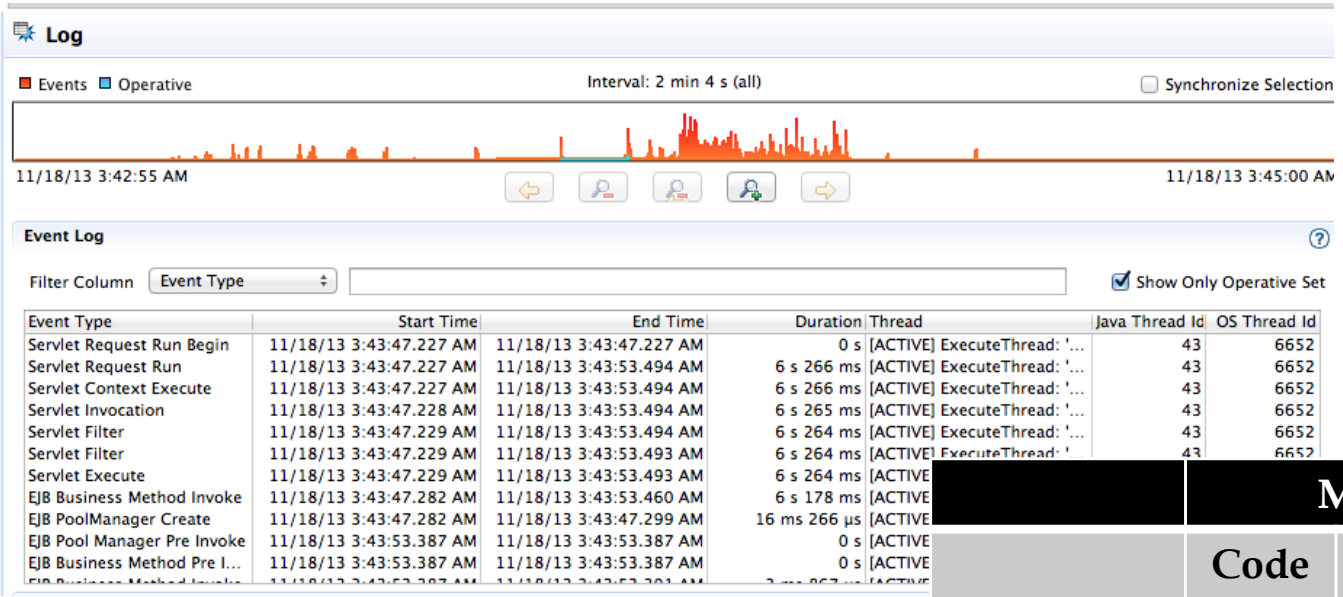
- **Deployment diagram in UML**



	Moment		Period	
	Code	Component	Code	Component
Build-time				
Run-time				

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

- **Event logging provides system administrators with information useful for diagnostics and auditing.** 事件日志：系统层面
  - 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.



	Moment		Period	
	Code	Component	Code	Component
Build-time				
Run-time				

Execution tracing and event logging

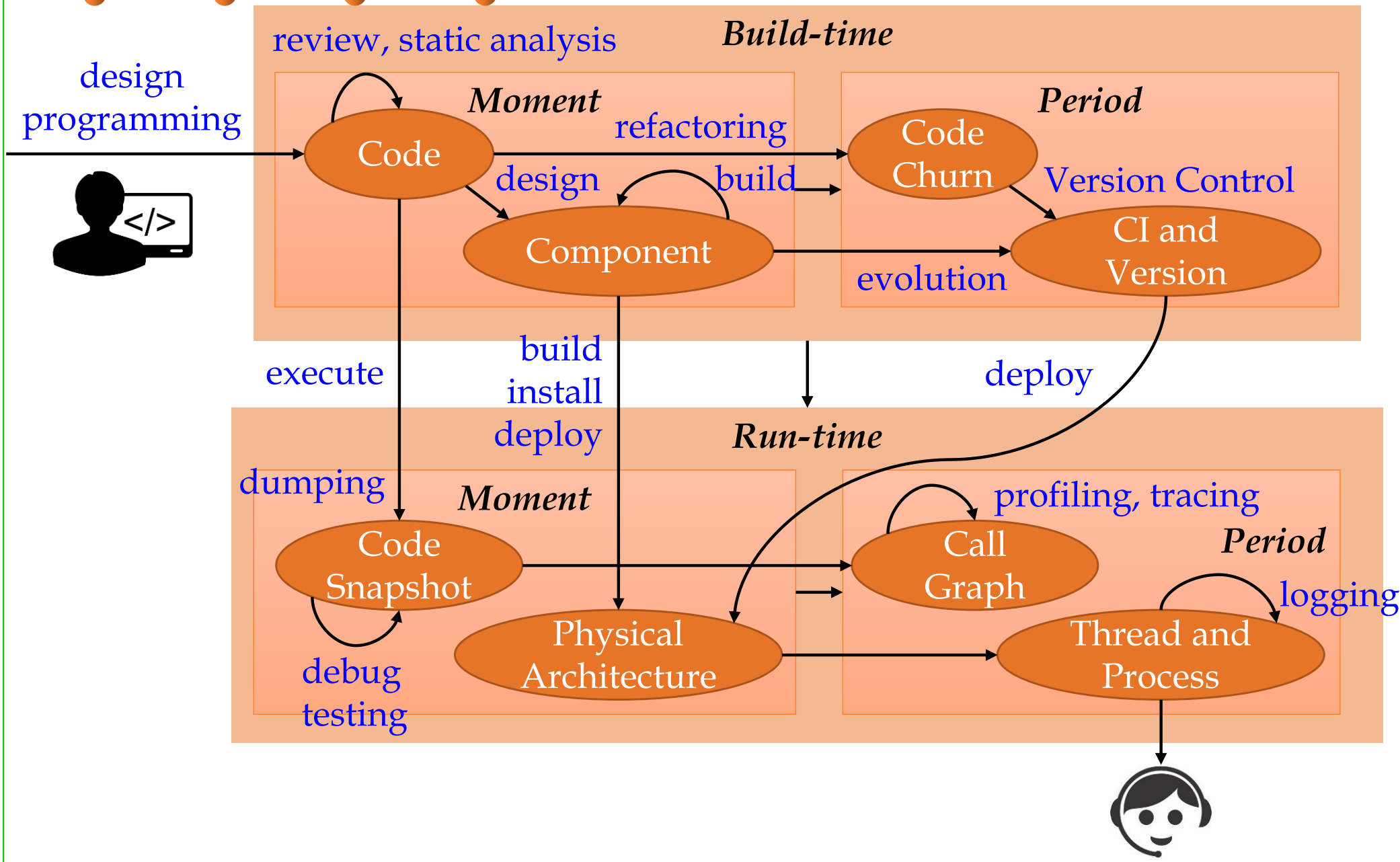
Event logging (构件/系统层面)	Execution tracing (代码层面)
Consumed primarily by system administrators	Consumed primarily by developers
Logs "high level" information (e.g. failed installation of a program)	Logs "low level" information (e.g. a thrown exception)
Must not be too "noisy" (contain many duplicate events 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

What we are focused on in this course

<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><td>Build-time</td><td>Source code, AST, Interface-Class-Attribute-Method (Class Diagram)</td><td>Package, File, Static Linking, Library, Test Case, Build Script (Component Diagram)</td><td>Code Churn</td><td>Configuration Item, Version</td></tr>
<tr><td>Run-time</td><td>Code Snapshot, Memory dump</td><td>Package, Library, Dynamic linking, Configuration, Database, Middleware, Network, Hardware (Deployment Diagram)</td><td>Execution stack trace, Concurrent multi-threads</td><td>Event log, Multi-processes, Distributed processes</td></tr>
<tr><td></td><td></td><td></td><td colspan="2">Procedure Call Graph, Message Graph (Sequence Diagram)</td></tr>
</table>

2 Software construction: Transformation between views

Software construction: transformation btw views



Types of Transformations in Software Construction

- **∅ ⇒ Code**
  - Programming / Coding (ADT/OOP)
  - Review, static analysis/checking
- **Code ⇒ Component**
  - Design (ADT/OOP; Reusability; Maintainability)
  - Build: compile, static link, package, install, etc
- **Build-time ⇒ Run-time**
  - Install / deploy
  - Debug, unit/integration testing (Robustness and Correctness)
- **Moment ⇒ Period**
  - Version control
  - Loading, dynamic linking, execution (dumping, profiling, logging)
  - Concurrent threads

3 Quality properties of software systems

External and internal quality factors

- **External quality factors: qualities such as speed or ease of use, whose presence or absence in a software product may be detected by its users.**

External quality factors affect users
外部质量因素 影响 用户

- **Other qualities applicable to a software product, such as being modular, or readable, are internal factors**, perceptible only to developers who have access to the actual software text.

Internal quality factors affect the software itself and its developers
内部质量因素 影响 软件本身和它的开发者

- **In the end, only external factors matter.**
- **But the key to achieving these external factors is in the internal ones:** for the users to enjoy the visible qualities, the designers and implementers must have applied internal techniques that will ensure the hidden qualities.

External quality results from internal quality
外部质量取决于内部质量

(1) External quality factors

External 1: Correctness

- **Correctness** is the ability of software products to perform their exact tasks, as defined by their specification. 按照预先定义的“规约”执行
- **Correctness is the prime quality** 正确性：最重要的质量指标

External 1: Correctness

- **Approaches** of ensuring correctness: **Testing and debugging**
  测试和调试：发现不正确、消除不正确 ⇒Robustness

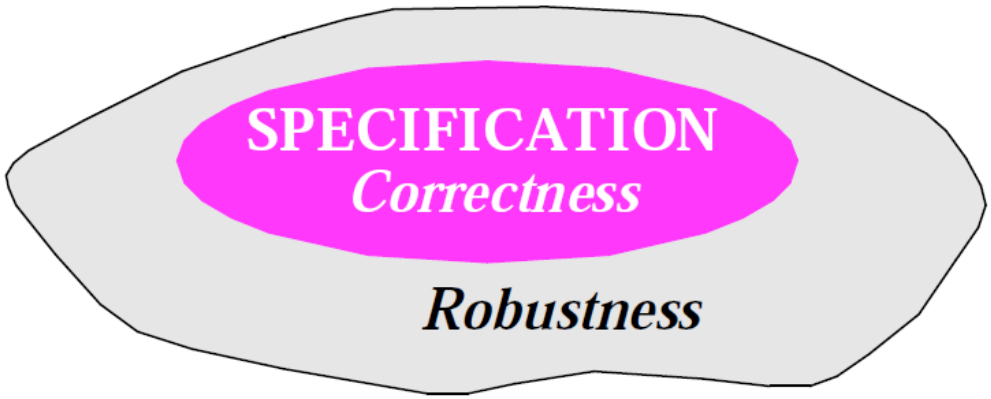
- **Defensive programming such as typing and assertions**
  防御式编程：在写程序的时候就确保正确性 meant to help build software that is correct from the start — rather than debugging it into correctness. **⇒Robustness**

- **Formal approach**: "check", "guarantee" and "ensure"
  形式化方法：通过形式化验证发现问题
  - Mathematical techniques for formal program specification and verification

  ⇒ Formal Language, Graduate courses

External 2: Robustness

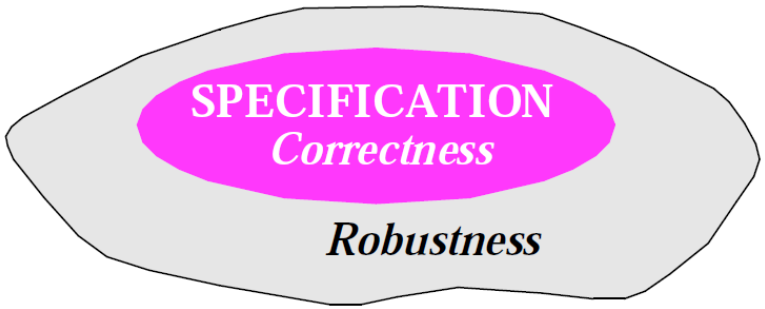
- **Robustness is the ability of software systems to react appropriately to abnormal conditions** 健壮性：针对异常情况的处理
  - Robustness complements correctness. 健壮性是对正确性的补充
  - Correctness addresses the behavior of a system in cases covered by its specification; 正确性：软件的行为要严格的符合规约中定义的行为
  - Robustness characterizes what happens outside of that specification. 健壮性：出现规约定义之外的情形的时候，软件要做出恰当的反应
- **Robustness** is to make sure that if such cases do arise, the system does not cause catastrophic events; it should produce appropriate error messages, terminate its execution cleanly, or enter a so-called "graceful degradation" mode.
- 健壮性：出现异常时不要“崩溃”



External 2: Robustness

- **Robustness is concerned with "abnormal case", which implies that the notions of normal and abnormal case are always relative to a certain specification "normal"和"abnormal"是主观而非客观**
  - An abnormal case is simply a case that is not covered by the specification 未被specification覆盖的情况即为"异常情况"
  - If you widen the specification, cases that used to be abnormal become normal — even if they correspond to events such as erroneous user input that you would prefer not to happen 所谓的"异常"，取决于spec的范畴
    - "Normal" in this sense does not mean "desirable", but simply "planned for in the design of the software".
    - Although it may seem paradoxical at first that erroneous input should be called a normal case, any other approach would have to rely on subjective criteria, and so would be useless.

⇒Exception handling



External 3: Extendibility

- **Extendibility (可扩展性)** is the ease of adapting software products to changes of specification. **对软件的规约进行修改，是否足够容易？**
- **The problem of extendibility is one of scale 规模越大，扩展起来越不容易**
  - For small programs change is usually not a difficult issue; but as software grows bigger, it becomes harder and harder to adapt.
  - A large software system often looks to its maintainers as a giant house of cards in which pulling out any one element might cause the whole edifice to collapse.
- **We need extendibility because at the basis of all software lies some human phenomenon and hence fickleness 为什么要扩展：应对变化**
  - Traditional approaches did not take enough account of change, relying instead on an ideal view of the software lifecycle where an initial analysis stage freezes the requirements, the rest of the process being devoted to designing and building a solution.

External 3: Extendibility

- **Two principles are essential for improving extendibility:**
  - ***Design simplicity***: a simple architecture will always be easier to adapt to changes than a complex one. 简约主义设计
  - ***Decentralization***: the more autonomous the modules, the higher the likelihood that a simple change will affect just one module, or a small number of modules, rather than triggering off a chain reaction of changes over the whole system. 分离主义设计

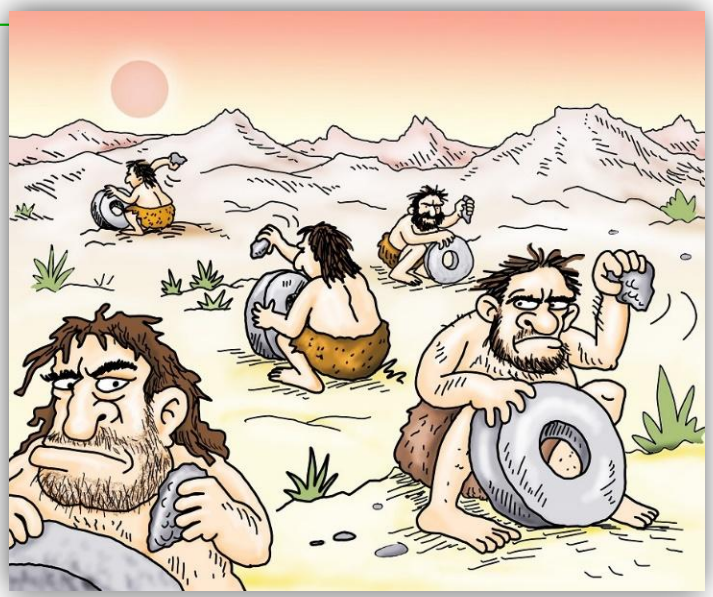
⇒ Chapter 2 (ADT and OOP)

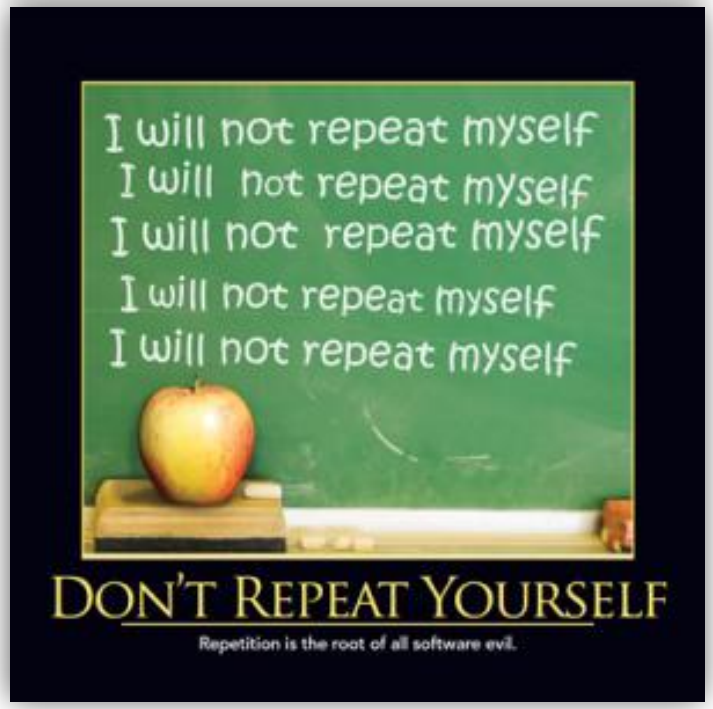
⇒ Chapter 3 (Modularity and adaptability)

External 4: Reusability

- **Reusability (可复用性)** is the ability of software elements to serve for the construction of many different applications. 一次开发，多次使用
- The need for reusability comes from the observation that software systems often follow similar patterns; it should be possible to exploit this commonality and avoid reinventing solutions to problems that have been encountered before. 发现共性
  - A reusable software element will be applicable to many different developments.
  - *Don't Repeat Yourself (DRY)*
  - *Don't Re-invent the Wheel*

⇒ Chapter 3 (Design for/with reuse)





External 5: Compatibility

- **Compatibility (兼容性)** is the ease of combining software elements with others. 不同的软件系统之间相互可容易的集成
- **Compatibility is important because we do not develop software elements in a vacuum (真空): they need to interact with each other.**
- **But they too often have trouble interacting because they make conflicting assumptions about the rest of the world.**
  - An example is the wide variety of incompatible file formats supported by many operating systems. A program can directly use another's result as input only if the file formats are compatible. (文件格式的兼容性)

External 5: Compatibility

- The key to compatibility lies **in homogeneity of design (保持设计的同构性)**, and in agreeing on standardized conventions for inter-program communication.

The key to compatibility is standardization, especially standard protocols. 标准化

- **Approaches include:**
  - **Standardized file formats**, as in the Unix system, where every text file is simply a sequence of characters.
  - **Standardized data structures**, as in Lisp systems, where all data, and programs as well, are represented by binary trees (called lists in Lisp).
  - **Standardized user interfaces**, as on various versions of Windows, OS/2 and MacOS, where all tools rely on a single paradigm for communication with the user, based on standard components such as windows, icons, etc.
- More general solutions are obtained by defining **standardized access protocols** to all important entities manipulated by the software.

External 6: Efficiency

- **Efficiency** is the ability of a software system to place as few demands as possible on hardware resources, such as processor time, space occupied in internal and external memories, bandwidth used in communication devices.
- **Efficiency does not matter much if the software is not correct** (*"do not worry how fast it is unless it is also right"*). 性能毫无意义，除非有足够的正确性
  - The concern for efficiency must be balanced with other goals such as extendibility and reusability; 对性能的关注 要与 其他质量属性进行折中
  - Extreme optimizations make the software so specialized as to be unfit for change and reuse. 过度的优化导致软件不再适应变化和复用
- Algorithms, I/O, memory management, etc.

Abstract concepts for correctness of computation vs. Concrete implementation for performance through optimization

External 6: Efficiency

过早优化是万恶之源

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

Donald Knuth

External 7: Portability (可移植性)

- **Portability** is the ease of transferring software products to various hardware and software environments. 软件可方便的在不同的技术环境之间移植

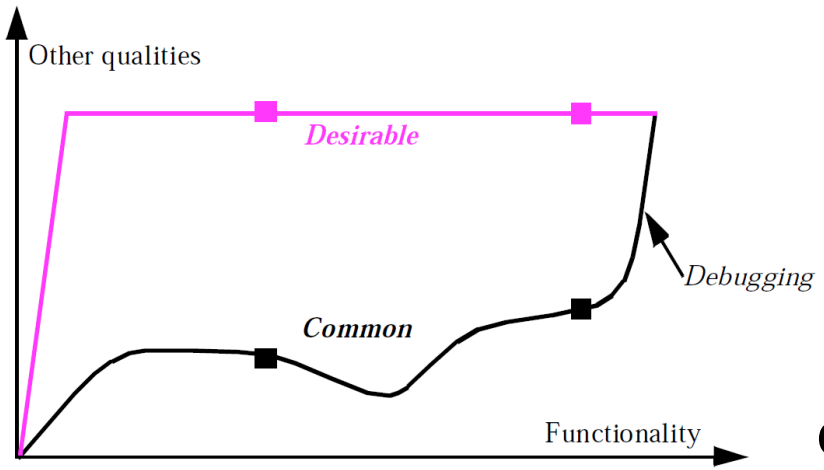
- Portability addresses variations not just of the physical hardware but more generally of the **hardware-software machine**, the one that we really program, which includes the operating system, the window system if applicable, and other fundamental tools. 硬件、操作系统

External 8: Ease of use

- **Ease of use (易用性)** is the ease with which people of various backgrounds and qualifications can learn to use software products and apply them to solve problems. (容易学、安装、操作、监控)
  - It also covers the ease of installation, operation and monitoring.
- How to provide detailed guidance and explanations to novice users, without bothering expert users who just want to get right down to business. (给用户提供详细的指南)
- **Structural simplicity**
  - A well-designed system, built according to a clear, well thought-out structure, will tend to be easier to learn and use than a messy one.
- **Know the user**
  - The argument is that a good designer must make an effort to understand the system's intended user community.

External 9: Functionality

- **Functionality is the extent of possibilities provided by a system.**
- ***Featurism* (often "*creeping featurism*")** 程序设计中一种不适宜的趋势，即软件开发者增加越来越多的功能，企图跟上竞争，其结果是程序极为复杂、不灵活、占用过多的磁盘空间
  - The easier problem is the **loss of consistency** that may result from the addition of new features, affecting its ease of use. Users are indeed known to complain that all the "bells and whistles" of a product's new version make it horrendously complex.
  - The more difficult problem is to avoid being so focused on features as to forget the other qualities (**Ignorance of overall quality**).



Osmond's curves

External 9: Functionality

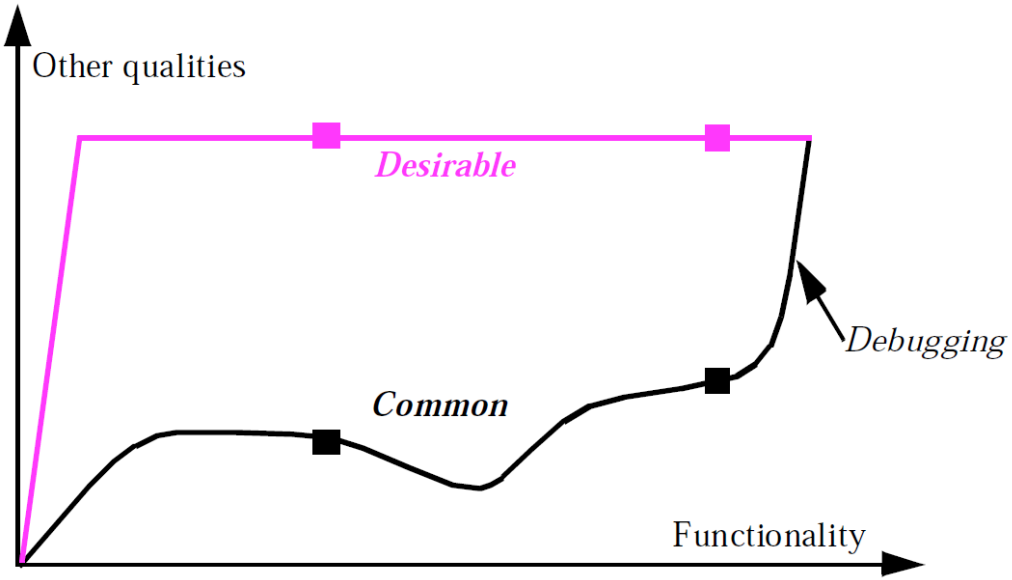
- What Osmond suggests (the color curve) is, aided by the quality-enhancing techniques of OO development, to maintain the quality level constant throughout the project for all aspects but functionality.
- You just do not compromise on reliability, extendibility and the like: you refuse to proceed with new features until you are happy with the features you have.

⇒ Chapter 2 (Agile, SCM)

Start with a small set of key features with all quality factors considered.

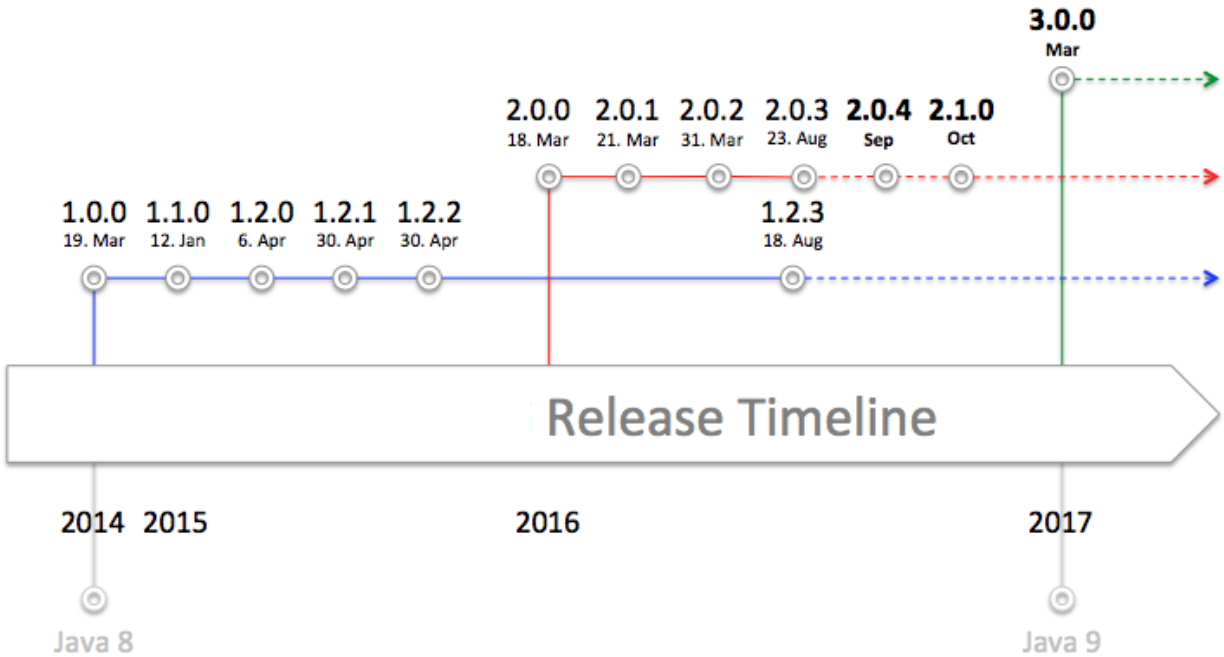
Add more features gradually during development process and guarantee the same quality as key features.

每增加一小点功能，都确保其他质量属性不受到损失



External 10: Timeliness

- **Timeliness (及时性)** is the ability of a software system to be released when or before its users want it.
- A great software product that appears too late might miss its target altogether.



External 10++: Other qualities

- **Verifiability (可验证性)** is the ease of preparing acceptance procedures, especially test data, and procedures for detecting failures and tracing them to errors during the validation and operation phases.

- **Integrity (完整性)** is the ability of software systems to protect their various components (programs, data) against unauthorized access and modification.
- **Repairability (可修复性)** is the ability to facilitate the repair of defects.
- **Economy (经济性)**, the companion of timeliness, is the ability of a system to be completed on or below its assigned budget.

(2) Internal quality factors

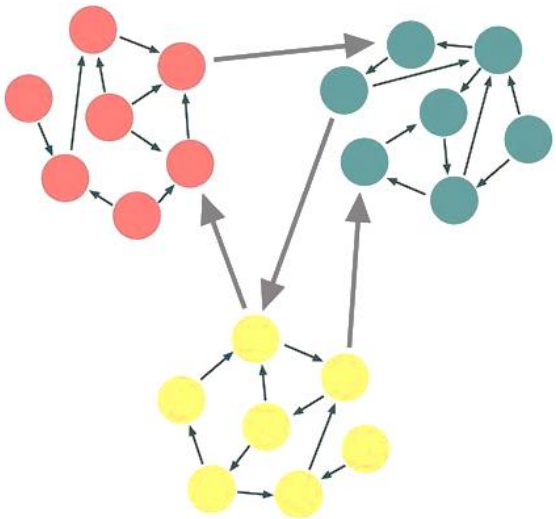
Internal quality factors

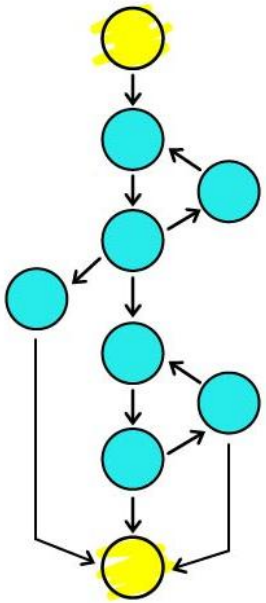
Complexity is the enemy of almost any external quality factors!

- **Source code related factors such as Lines of Code (LOC), Cyclomatic Complexity, etc**
- **Architecture-related factors such as coupling, cohesion, etc**

system	Lines of code (LOC)
Netscape	17,000,000
Space shuttle	10,000,000
Linux	1,500,000
Windows XP	40,000,000
Boeing 777	7,000,000

- **Readability**
- **Understandability**
- **Clearness**
- **Size**





Internal quality factors are usually used as partial measurement of external quality factors.

(3) Tradeoff between quality properties

Tradeoff between quality properties 折中

- How can one get *integrity* without introducing protections of various kinds, which will inevitably hamper *ease of use*?
- *Economy* often seems to fight with *functionality*.
- Optimal *efficiency* would require perfect adaptation to a particular hardware and software environment, which is the opposite of *portability*, and perfect adaptation to a specification, where *reusability* pushes towards solving problems more general than the one initially given.

Integrity vs. ease of use
Economy vs. functionality
Efficiency vs. portability
Efficiency vs. reusability
Economy vs. reusability
Timeliness vs. extendibility

- *Timeliness* pressures might tempt us to use "Rapid Application Development" techniques whose results may not enjoy much *extendibility*.

Tradeoff between quality properties

- **Developers need to make tradeoffs.**
  - Too often, developers make these tradeoffs implicitly, without taking the time to examine the issues involved and the various choices available; efficiency tends to be the dominating factor in such silent decisions.
  - A true software engineering approach implies an effort to state the criteria clearly and make the choices consciously.
  - 正确的软件开发过程中，开发者应该将不同质量因素之间如何做出折中的设计决策和标准明确的写下来
- **Necessary as tradeoffs between quality factors may be, one factor stands out from the rest: correctness.**
  - There is never any justification for compromising correctness for the sake of other concerns such as efficiency.
  - If the software does not perform its function, the rest is useless.
  - 虽然需要折中，但“正确性”绝不能与其他质量因素折中。

Key concerns of software construction

- **All the qualities discussed above are important.**
- **最重要的几个质量因素**
  But in the current state of the software industry, four stand out:
  - *Correctness* and *robustness*: reliability
    - Systematic approaches to software construction
    - Formal specification
    - Automatic checking during development process
    - Better language mechanism
    - Consistency checking tools
  - *Extendibility* and *reusability*: modularity

How OOP improves quality

- **Correctness**: encapsulation, decentralization
- **Robustness**: encapsulation, error handling
- **Extendibility**: encapsulation, information hiding
- **Reusability**: modularity, component, models, patterns
- **Compatibility**: standardized module and interface
- **Portability**: information hiding, abstraction
- **Ease of use**: GUI components, framework
- **Efficiency**: reusable components
- **Timeliness**: modeling, reuse
- **Economy**: reuse
- **Functionality**: extendibility

4 Five key quality objectives of software construction

Quality considerations of this course

- **Elegant and beautiful code ⇒ easy to understand, understandability**
- **Design for/with reuse ⇒ cheap for develop**
- **Low complexity ⇒ ready for changes, easy to extend**
- **Robustness and correctness ⇒ safe from bug, not error-prone**
- **Performance and efficiency ⇒ efficient to run**

→ Reusability

→ Maintainability

→ Robustness/Correctness

↓

Chapter 2 ADT and OOP

Understandability

	Moment		Period	
	Code-level	**Component-level**	**Code-level**	**Component-level**
Build-time	代码的可理解性（变量/子程序/语句的命名与构造标准、代码布局与风格、注释、复杂度） Code Review; Walkthrough; Static Code Analysis; ADT/函数规约	构件/项目的可理解性（包的组织、文件的组织、命名空间）	Refactoring	Version control
Run-time			Log Trace	

Reusability

	Moment		Period	
	Code-level	**Component-level**	**Code-level**	**Component-level**
Build-time	ADT/OOP; 接口与实现分离; 继承/重载/重写; 组合/代理; 多态; 子类型与泛型; OO设计模式	API design; Library; Framework (for/with reuse); Static linking		
Run-time		Dynamic linking		

Maintainability and Adaptability

	Moment		Period	
	Code-level	**Component-level**	**Code-level**	**Component-level**
Build-time	模块化设计; 聚合度/耦合度; SOLID; OO设计模式; State-based programming; Grammar-based programming	SOLID; GRASP		SCM Version Control
Run-time				

Robustness

	Moment		Period	
	Code-level	**Component-level**	**Code-level**	**Component-level**
Build-time	Error handling; Exception handling; Assertion; Defensive programming; Test-first programming		Continuous Integration; Regression Testing	
Run-time	Unit/Integration Testing; Debug; Memory Dumping		Logging Tracing	

Performance

	Moment		Period	
	Code-level	**Component-level**	**Code-level**	**Component-level**
Build-time	代码调优; Design pattern			
Run-time	空间复杂性（内存管理性能）； 时间复杂性（I/O性能）； Memory dump; Garbage Collection (GC)	分布式系统	Performance profiling, analysis and tuning	并行/多线程

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

Summary

- **Quality properties of software systems**
  - External vs. internal quality factors
  - Important external quality factors
  - Tradeoff between quality factors
- **Five key quality objectives of software construction**
  - **Easy to understand**: elegant and beautiful code / understandability
  - **Ready for change**: maintainability and adaptability
  - **Cheap for develop**: design for/with reuse: reusability
  - **Safe from bugs**: robustness
  - **Efficient to run**: performance
- **Construction techniques to be studied in this course (classified by the orientation of five key quality objectives)**

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

The end

March 2, 2024