哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

10 Construction for Change
面向可维护性的构造技术

刘铭

June 7, 2021

Outline

- **Software Maintenance and Evolution**
- **Metrics of Maintainability**
- **Modular Design and Modularity Principles**
- **OO Design Principles: SOLID**
- **Grammar-based construction**
  - Grammar and Parser
  - Regular Expression (regexp)

本次课面向另一个质量指标：可维护性——软件发生变化时，是否可以以很小的代价适应变化？

（1）什么是软件维护；
（2）可维护性如何度量；
（3）实现高可维护性的设计原则
（4）基于语法的构造技术

Reading

- 软件工程--实践者的研究方法：第23章
- Object-Oriented Software Construction：第3章

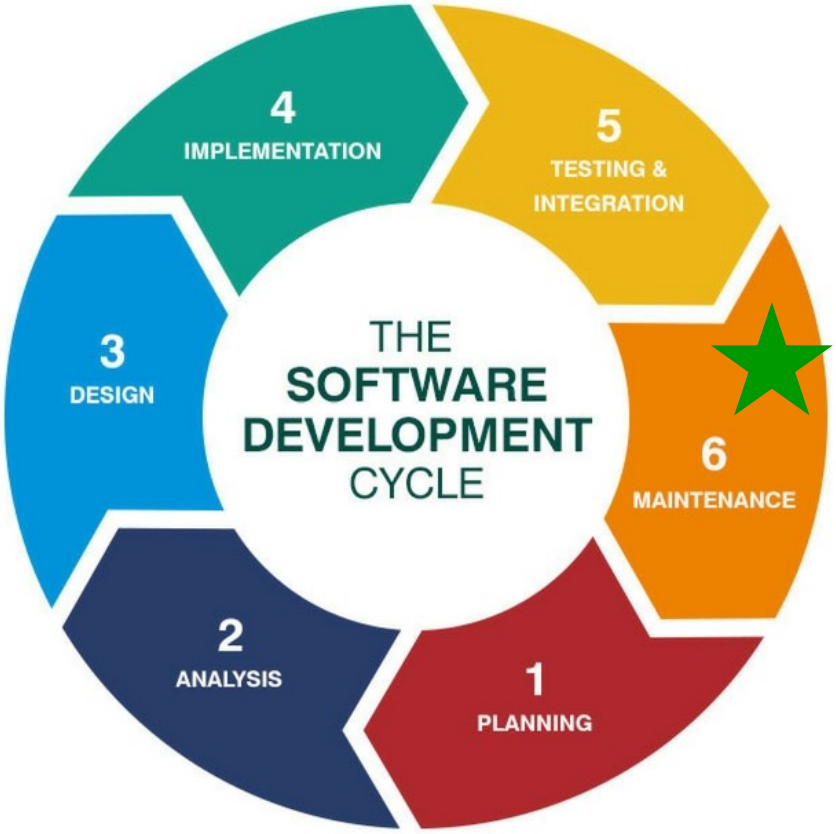
- MIT 6.031：17、18
- Java编程思想：第13.6节

1 Software Maintenance and Evolution

What is Software Maintenance?

- **Software maintenance** in software engineering is the modification of a software product after delivery to correct faults, to improve performance or other attributes. 软件维护：修复错误、改善性能

- In "ISO/IEC 14764:2006 Software Engineering — Software Life Cycle Processes — Maintenance"



Operation & Maintenance Engineer 运维工程师

- **Maintenance is one of the most difficult aspects of software production because maintenance incorporates aspects of all other phases 运维是软件开发中最困难的工作之一**
- **A fault is reported from users and is to be handed by a maintenance engineer. 处理来自用户报告的故障/问题**
- **A maintenance engineer must have superb debugging skills**
  - The fault could lie anywhere within the product, and the original cause of the fault might lie in the by now non-existent specifications or design documents (bug/issue localization).
  - Superb diagnostic skills, testing skills and documentation skills are required (testing, fix, and documenting changes).

Operation & Maintenance Engineer 运维工程师

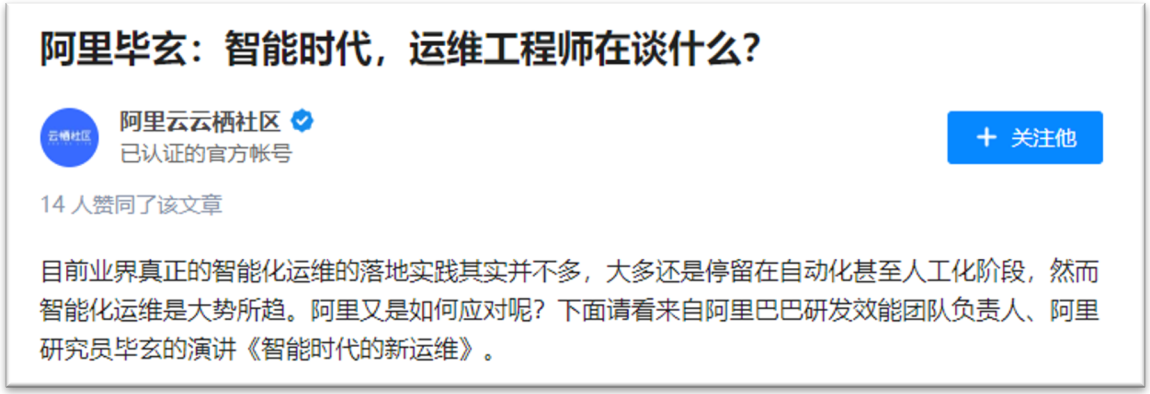
拿什么拯救你:苦逼的IT运维工程师!

2014-12-26 17:03 it168网站 原创 作者: 皮丽华 编辑: 皮丽华 5条评论

【IT168 评论】目前国内的IT运维还处于救火队员的初级阶段，除了脏活就是累活，天天疲于奔命。什么网络中断、应用卡顿、响应速度慢，服务器宕机啊，各种突发故障都可能让业务成交失败，而查找系统运行的日志又特别费时费力，挖故障如同大海捞针啊，这着实让运维人员急得团团转。作为运维工程师的您，有木有遇到过很苦逼的经历?ChinaUnix社区的网友呼声一片，各自谈起了自己的伤心往事。 (http://bbs.chinaunix.net/thread-4162292-1-1.html)

Operation & Maintenance Engineer 运维工程师

- **阿里运维工程师职责：**
  - 负责系统稳定性工作；
  - 生产系统部署、上线；
  - 维护生产系统网络安全、稳定、可靠；
  - 维护生产系统数据备份；



- **岗位要求：**
  - 深入理解运维体系结构，精于容量规划、架构设计、性能优化；
  - 熟悉服务管理、单元部署、自动扩容等运维系统建设，对成本控制和效能提升有深刻的理解和实践
  - 熟悉故障、监控、限流、降级、预案、扩容工作原理；
  - 熟悉java虚拟机，对java应用的部署及系统优化有一定的经验；
  - 熟悉自动化发布工具、熟悉docker技术优先；
  - …

大多数运维任务并非需要涉及代码修改。

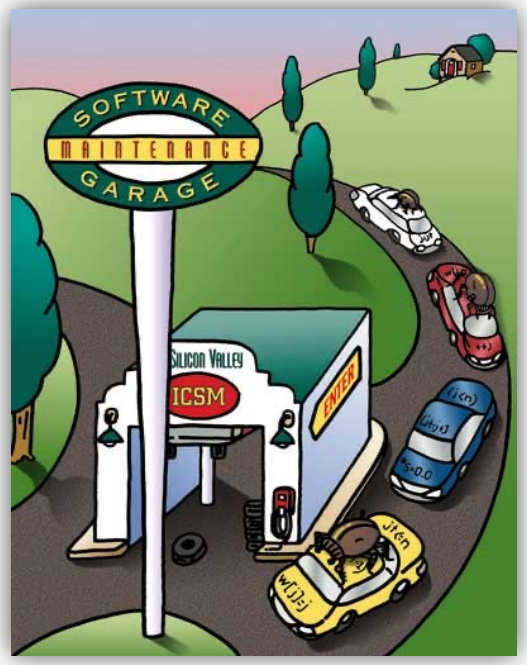
但是有时候不得不回到代码层面…

After fixing the code

- **More Steps:**
  - Test that the modification works correctly: use specially constructed test cases 测试所做的修改
  - Check for regression faults: use stored test data, and add specially test cases to stored test data for future regression testing 回归测试
  - Document all changes 记录变化
- **How to minimize regression faults 除了修复问题，修改中不能引入新的故障**
  - Consult the detailed documentation and make use of constructed test cases.
- **What usually happens: no enough documentation / test cases 最大的问题：修改后没有足够的文档记录和测试**
  - The operation engineer has to deduce from the source code itself all the information needed to avoid introducing a regression fault.

Types of software maintenance

- **Corrective maintenance** **25%** **纠错性**
  - Reactive modification of a software product performed after delivery to correct discovered problems;
- **Adaptive maintenance** **21%** **适应性**
  - Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment;
- **Perfective maintenance** **50%** **完善性**
  - Enhancement of a software product after delivery to improve performance or maintainability;
- **Preventive maintenance** **4%** **预防性**
  - Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.



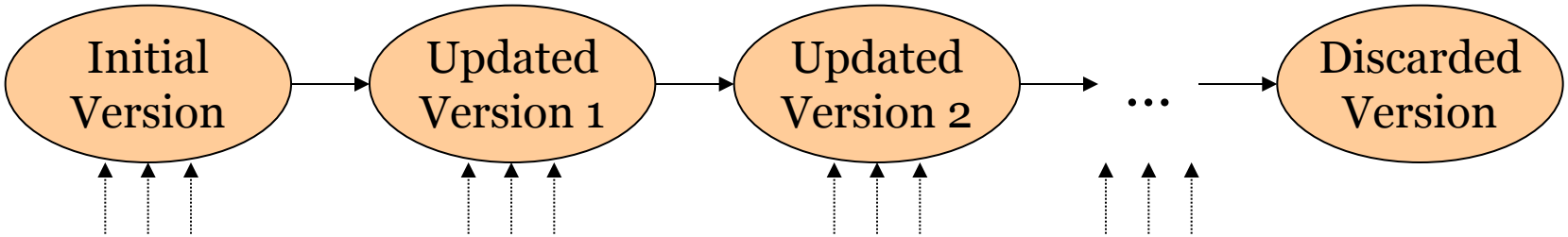
Software Evolution

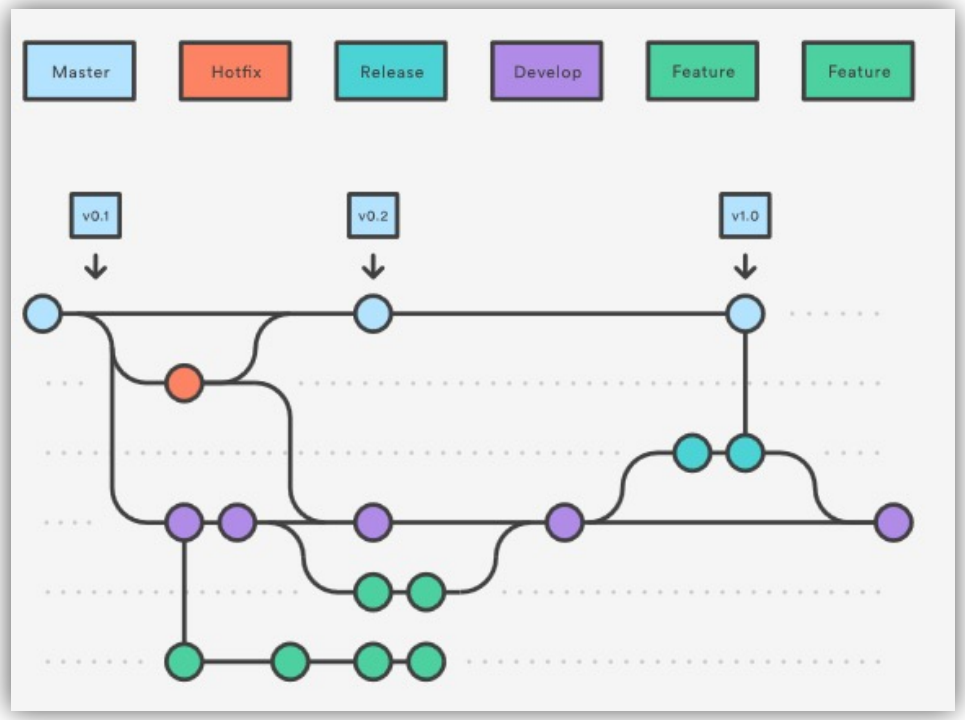
- **Software evolution is a term used in software maintenance, referring to the process of developing software initially, then repeatedly updating it for various reasons.** 软件演化：对软件进行持续的更新

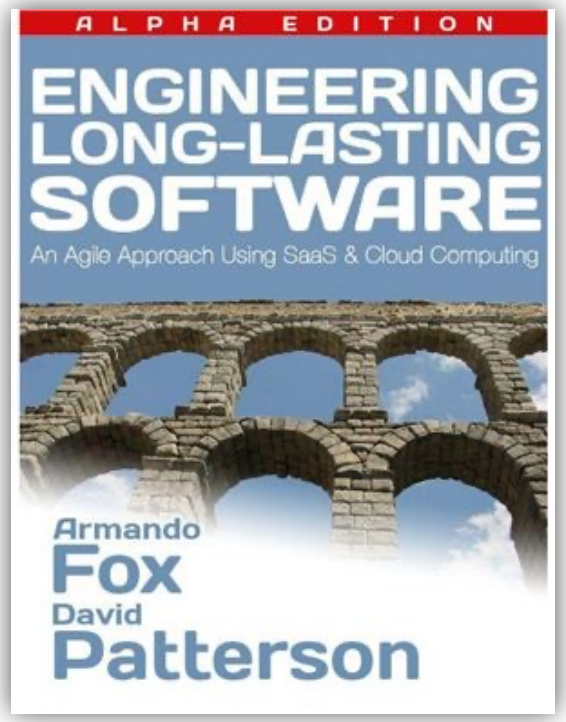
- **Over 90% of the costs of a typical system arise in the maintenance phase, and that any successful piece of software will inevitably be maintained.** 软件的大部分成本来自于维护阶段

Software Evolution

- **Multiple versions in the life of a software: From 1 to *n***

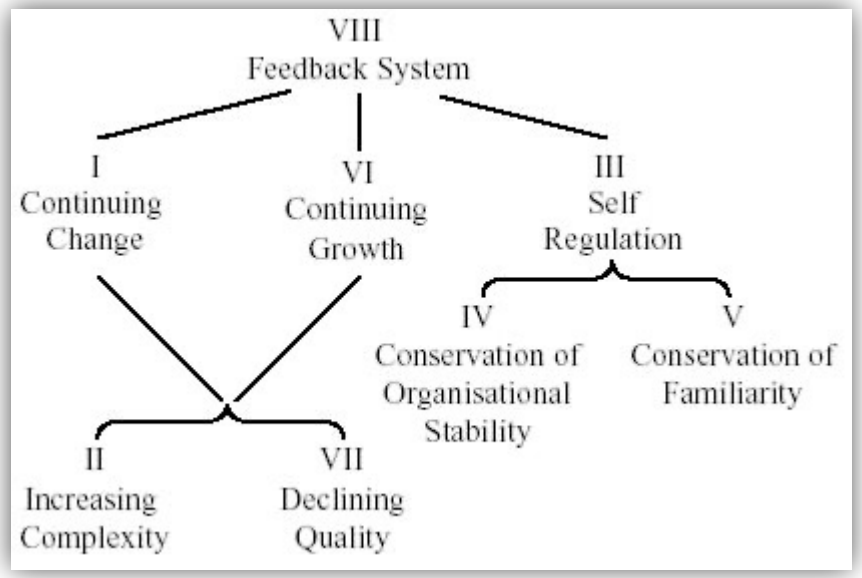






Lehman's Laws on Software Evolution

- **Feedback System**
- **Continuing Change**
- **Continuing Growth**
- **Declining Quality**
- **Increasing Complexity**
- **Self Regulation**
  - Conservation of Organizational Stability
  - Conservation of Familiarity



"变化"在软件生命周期中是不可避免的！
如何在最初的设计中充分考虑到未来的变化，避免因为频繁变化导致软件复杂度的增加和质量的下降？

Maintenance is not just the task of op engineers…

- **Maintenance is not just the task of maintenance and operation engineers, but also a potential task of software designers and developers.** 软件维护不仅仅是运维工程师的工作，而是从设计和开发阶段就开始了
- **For them, it is mandatory to consider future potential changes/extensions of the software during the design and construction phases;** 在设计与开发阶段就要考虑将来的可维护性
- **So that flexible and extensible design/constructions are comprehensively considered, in other words, "easy to change / extension".** 设计方案的"easy to change"
- **This is what's called "maintainability", "extensibility" and "flexibility" of software construction.**

Examples of maintainability-oriented construction

- **Modular design and implementation** 模块化
  - Low coupling and high cohesion
- **OO design principles** OO设计原则
  - SOLID、GRASP
- **OO design patterns** OO设计模式
  - Factory method pattern, Builder pattern
  - Bridge pattern, Proxy pattern
  - Memento pattern, State pattern
- **State-based construction (Automata-based programming)** 基于状态的构造技术
- **Table-driven construction** 表驱动的构造技术
- **Grammar-based construction** 基于语法的构造技术

2 Metrics of Maintainability

Many names of maintainability

Ready for Change
Ready for Extension

- **Maintainability** 可维护性 — "The ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment".
- **Extensibility** 可扩展性 — Software design/implementation takes future growth into consideration and is seen as a systemic measure of the ability to extend a system and the level of effort required to implement the extension.
- **Flexibility** 灵活性 — The ability of software to change easily in response to user requirements, external technical and social environments, etc.
- **Adaptability** 可适应性 — The ability of an interactive system (adaptive system) that can adapt its behavior to individual users based on information acquired about its user(s) and its environment.

Many names of maintainability

- **Manageability** 可管理性 —How efficiently and easily a software system can be monitored and maintained to keep the system performing, secure, and running smoothly.

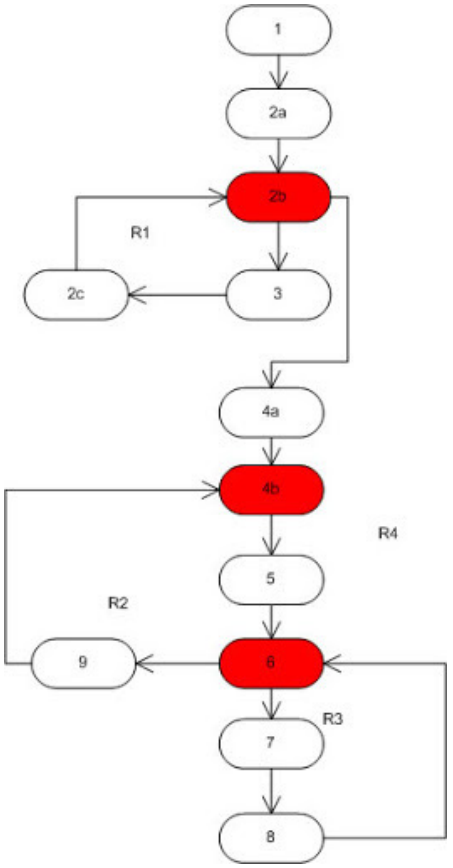
- **Supportability** 支持性—How effectively a software can be kept running after deployment, based on resources that include quality documentation, diagnostic information, and knowledgeable and available technical staff.

Questions about maintainability

- **Code review的时候经常问的关于可维护性的问题：**
  - Structural and design simplicity: how easy is it to change things? 设计结构是否足够简单？
  - Are things tightly or loosely coupled (i.e., separation of concerns)?模块之间是否松散耦合？
  - Are all elements in a package/module cohesive and their responsibilities clear and closely related? 模块内部是否高度聚合？
  - Does it have overly deep inheritance hierarchies or does it favor composition over inheritance? 是否使用了非常深的继承树，是否使用了delegation替代继承？
  - How many independent paths of execution are there in the method definitions (i.e., cycolmatic complexity)? 代码的圈复杂度是否太高？
  - How much code duplication exists? 是否存在重复代码？
  - …

Some common-used maintainability metrics

- **Cyclomatic Complexity** 圈复杂度 - Measures the structural complexity of the code.
  - It is created by calculating the number of different code paths in the flow of the program.
  - A program that has complex control flow will require more tests to achieve good code coverage and will be less maintainable.
  - CC = E-N+2, CC=P+1, CC=number of areas
- **Lines of Code** 代码行数 - Indicates the approximate number of lines in the code.
  - A very high count might indicate that a type or method is trying to do too much work and should be split up.
  - It might also indicate that the type or method might be hard to maintain.



Some common-used maintainability metrics

For a given problem, Let:

- η_1 = the number of distinct operators
- η_2 = the number of distinct operands
- N_1 = the total number of operators
- N_2 = the total number of operands

From these numbers, several measures can be calculated:

- Program vocabulary: $\eta = \eta_1 + \eta_2$
- Program length: $N = N_1 + N_2$
- Calculated program length: $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$
- Volume: $V = N \times \log_2 \eta$
- Difficulty : $D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$
- Effort: $E = D \times V$

The difficulty measure is related to the difficulty of the program to write or understand

The effort measure translates into actual coding time using the following relation,

- Time required to program: $T = \frac{E}{18}$ seconds

Halstead's delivered bugs (B) is an estimate for the number of errors in the implementation.

- Number of delivered bugs : $B = \frac{E^{\frac{2}{3}}}{3000}$ or, more recently, $B = \frac{V}{3000}$ is accepted

Halstead Volume: a composite metric based on the number of (distinct) operators and operands in source code.

Some common-used maintainability metrics

- **Maintainability Index (MI)** 可维护性指数- Calculates an index value between 0 and 100 that represents the relative ease of maintaining the code.
- **A high value means better maintainability. It is calculated based on:**
  - Halstead Volume (HV)
  - Cyclomatic Complexity (CC)
  - The average number of lines of code per module (LOC)
  - The percentage of comment lines per module (COM).

$$171 - 5.2 ln(\text{HV}) - 0.23\text{CC} - 16.2 ln(\text{LOC}) + 50.0 sin\sqrt{2.46 * \text{COM}}$$

Some common-used maintainability metrics

- **Depth of Inheritance** 继承的层次数 - Indicates the number of class definitions that extend to the root of the class hierarchy. The deeper the hierarchy the more difficult it might be to understand where particular methods and fields are defined or/and redefined.
- **Class Coupling** 类之间的耦合度 - Measures the coupling to unique classes through parameters, local variables, return types, method calls, generic or template instantiations, base classes, interface implementations, fields defined on external types, and attribute decoration.
  - Good software design dictates that types and methods should have high cohesion and low coupling.
  - High coupling indicates a design that is difficult to reuse and maintain because of its many interdependencies on other types.
- **Unit test coverage**单元测试的覆盖度 **-** indicates what part of the code base is covered by automated unit tests. (to be studied in Chapter 7)

Many other maintainability metrics

- 请自行查阅资料加以理解

Traditional metrics	Language specific coding violations (Fortran)	Code smells	Other maintainability metrics
• LOC • Cyclomatic complexity • Halstead complexity measures • Maintainability Index • Unit test coverage	• Use of old FORTRAN 77 standard practices, when better, modern ones are available in e.g. Fortran 2008	• Duplicated Code • Long Method • Large Class • Long Parameter List • Divergent Change • Shotgun Surgery • Feature Envy • Data Clumps • ...	• Defect density • Active files

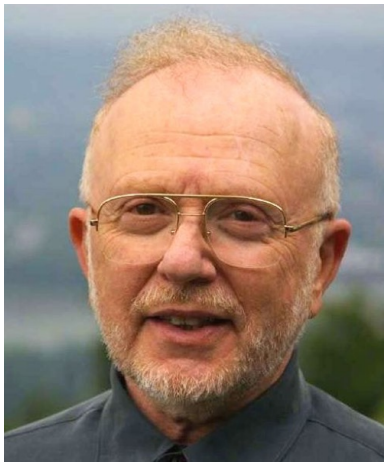
3 Modular Design and Modularity Principles

Modular programming 模块化编程

- **Modular programming is** a design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.
- **High-level decomposition of the code of an entire program into pieces in both Structured Programming and OOP.**

Niklaus Wirth (1934-)
Turing Award 1984

Edager Dijkstra (1930-2002)
Turing Award 1972



David L. Parnas (1941-)

Modular programming

- **The goal of design is to partition the system into modules and assign responsibility among the components in a way that:**
  - High cohesion within modules 高内聚
  - Loose coupling between modules 低耦合
- **Modularity reduces the total complexity a programmer has to deal with at any one time assuming:**
  - Functions are assigned to modules in away that groups similar functions together (Separation of concerns) 分离关注点
  - There are small, simple, well-defined interfaces between modules (Information hiding) 信息隐藏
- **The principles of cohesion and coupling are probably the most important design principles for evaluating the maintainability of a design.**

(1) Five Criteria for Evaluating Modularity

Five Criteria for Evaluating Modularity

- **Decomposability (可分解性)**
  - Are larger components decomposed into smaller components?
- **Composability (可组合性)**
  - Are larger components composed from smaller components?
- **Understandability (可理解性)**
  - Are components separately understandable?
- **Continuity (可持续性)** ——发生变化时受影响范围最小
  - Do small changes to the specification affect a localized and limited number of components?
- **Protection (出现异常之后的保护)** ——出现异常后受影响范围最小
  - Are the effects of run-time abnormalities confined to a small number of related components?

(2) Five Rules of Modularity Design

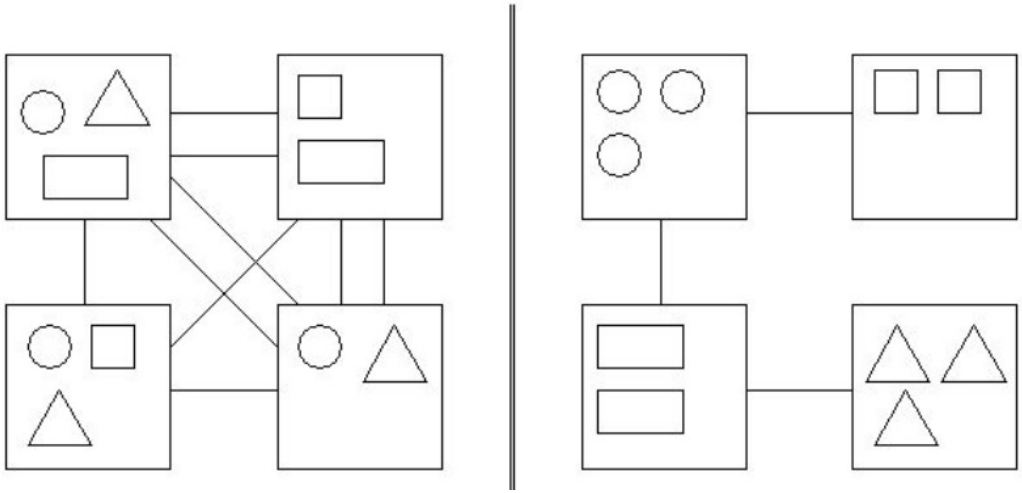
Five Rules of Modularity Design

- Direct Mapping (直接映射)
- Few Interfaces (尽可能少的接口)
- Small Interfaces (尽可能小的接口)
- Explicit Interfaces (显式接口)
- Information Hiding (信息隐藏)

(3) Coupling and Cohesion

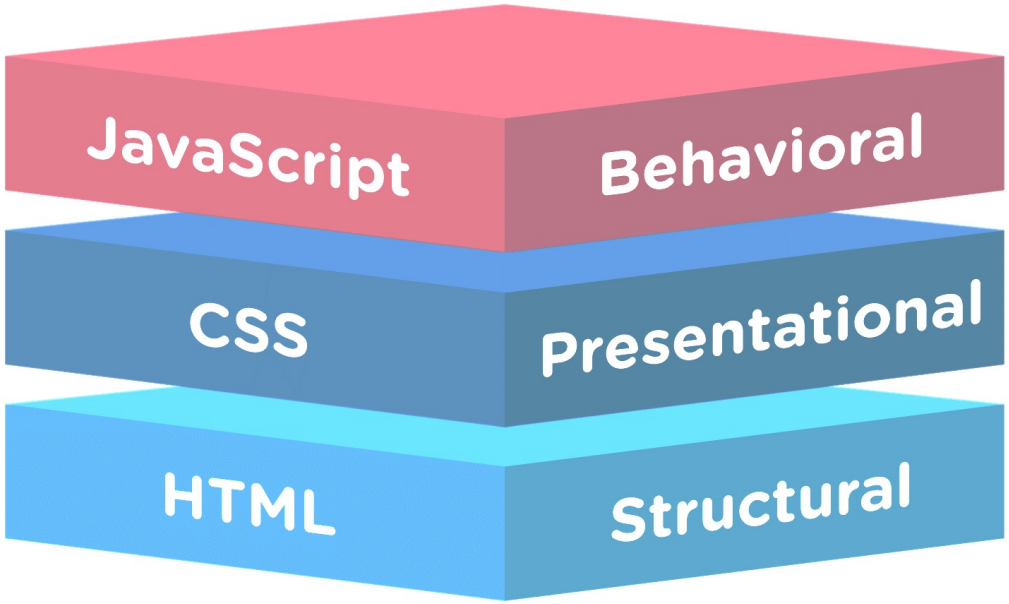
Coupling

- **Coupling is the measure of dependency between modules. A dependency exists between two modules if a change in one could require a change in the other.**
- **The degree of coupling between modules is determined by:**
  - The number of interfaces between modules (quantity), and
  - Complexity of each interface (determined by the type of communication) (quality)



Coupling between HTML, CSS and JavaScript

- **A well-designed web app modularizes around:**
  - HTML files which specify data and semantics
  - CSS rules which specify the look and formatting of HTML data
  - JavaScript which defines behavior/interactivity of page



Coupling between HTML, CSS and JavaScript

- **HTML:**

```
<!doctype html>
<html>
<head>
  <script type="text/javascript" src="base.js"></script>
  <link rel="stylesheet" href="default.css">
</head>
<body>
  <button onclick="highlight2()">Highlight</button>
  <button onclick="normal2()">Normal</button>
  <h1 id="title" class="NormalClass">CSS <--> JavaScript Coupling</h1>
</body>
</html>
```

- **CSS:**

```
.NormalClass {
  color:inherit;
  font-style:normal;
}
```

- **Output:**

Coupling between HTML, CSS and JavaScript

- **JavaScript code modifies the style attribute of HTML element.**

```
function highlight() {
    document.getElementById("title").style.color="red";
    document.getElementById("title").style.fontStyle="italic";
}

function normal() {
    document.getElementById("title").style.color="inherit";
    document.getElementById("title").style.fontStyle="normal";
}
```

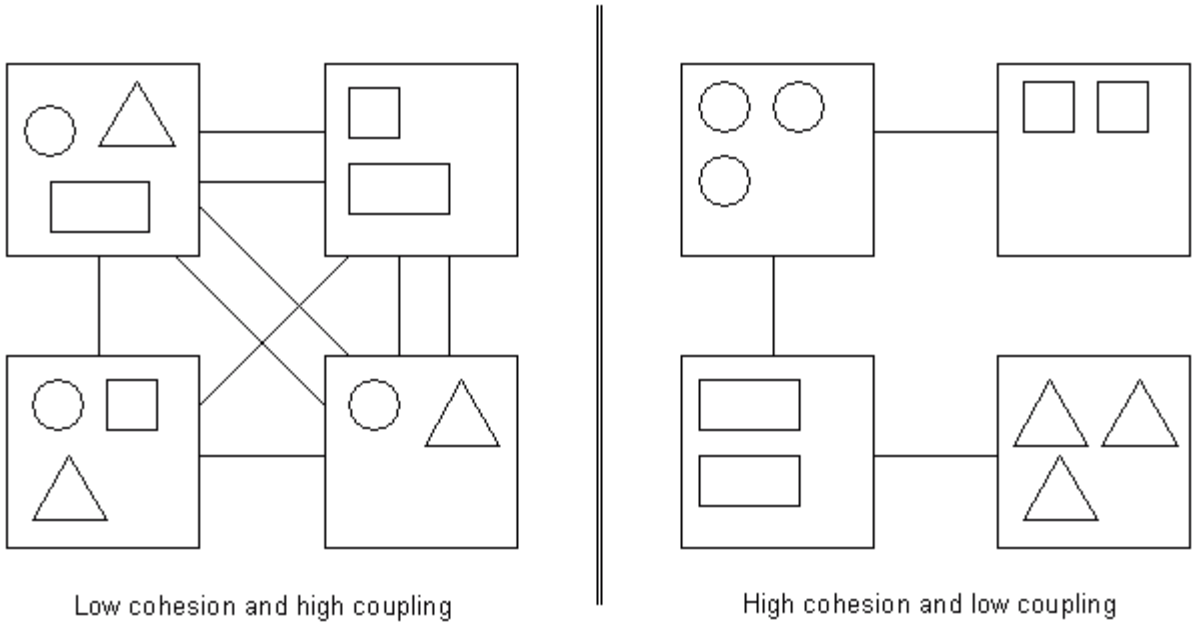
- **Or, JavaScript code modifies the class attribute of HTML element.**

```
function highlight() {
    document.getElementById("title").className = "HighlightClass";
}

function normal() {
    document.getElementById("title").className = "NormalClass";
}
```

Cohesion

- **Cohesion is a measure of how strongly related the functions or responsibilities of a module are.**
- **A module has high cohesion if all of its elements are working towards the same goal.**

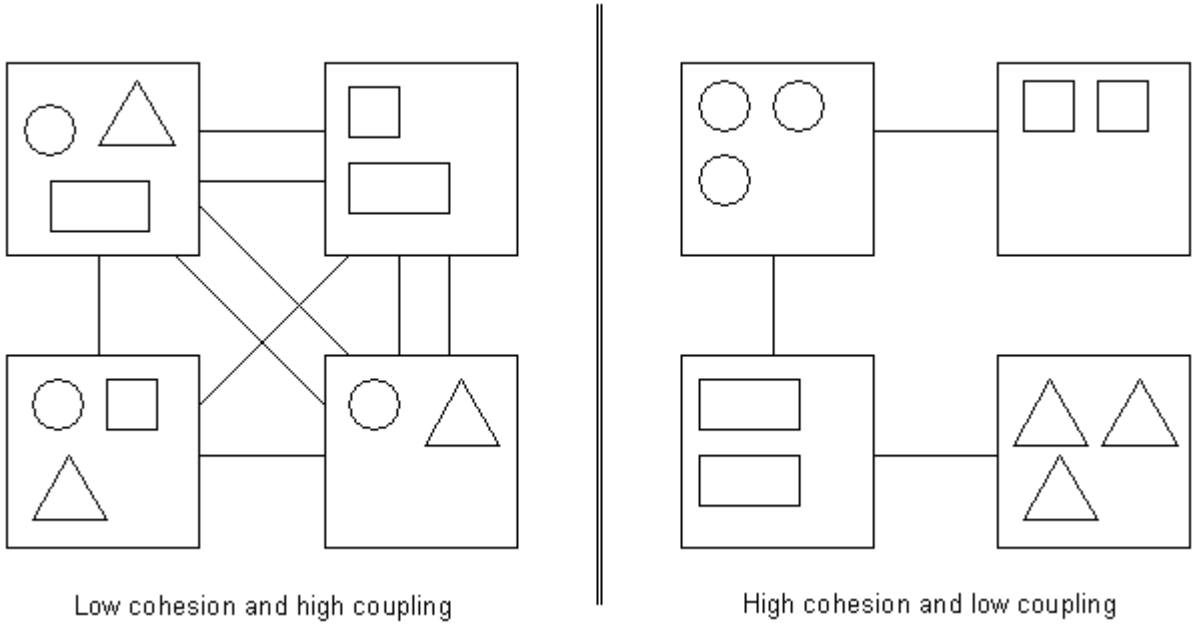


Low cohesion and high coupling

High cohesion and low coupling

Cohesion and Coupling

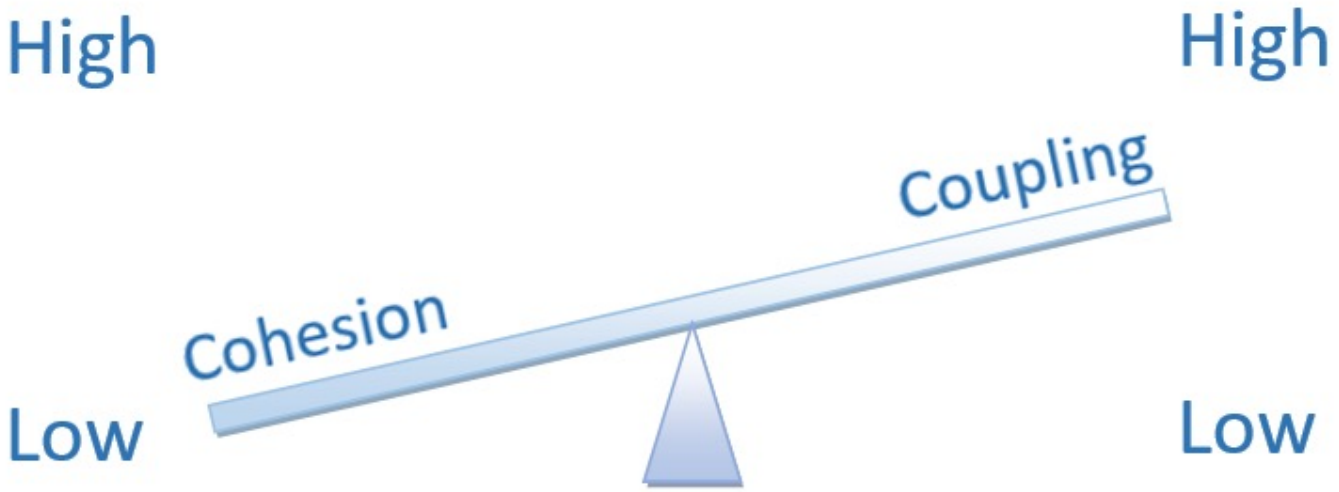
- **The best designs have high cohesion (also called strong cohesion) within a module and low coupling (also called weak coupling) between modules.**



Low cohesion and high coupling

High cohesion and low coupling

Coupling and Cohesion are with trade-off



- **When Coupling is high, cohesion tends to be low and vise versa.**

4 OO Design Principles: SOLID

SOLID: 5 classes design principles

- **(SRP) The Single Responsibility Principle** **单一责任原则**
- **(OCP) The Open-Closed Principle** **开放-封闭原则**
- **(LSP) The Liskov Substitution Principle** **Liskov替换原则**
- **(DIP) The Dependency Inversion Principle** **依赖转置原则**
- **(ISP) The Interface Segregation Principle** **接口聚合原则**

(1) Single Responsibility Principle (SRP)

单一责任原则

Single Responsibility Principle

- **"There should never be more than one reason for a class to change", i.e., a class should concentrate on doing one thing and one thing only. 不应该有多于1个原因让你的ADT发生变化，否则就拆分开**

Just because you can, doesn't mean you should

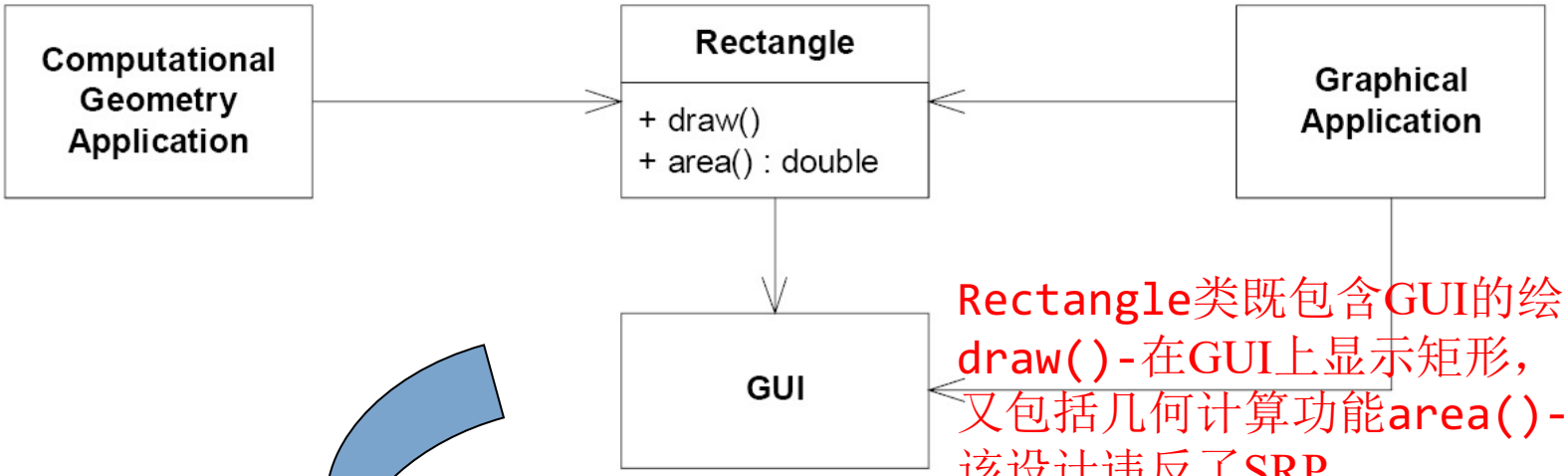
(SRP) The Single Responsibility Principle

- **Responsibility: "a reason for change." (责任：变化的原因)**
- **SRP:**
  - There should never be more than one reason for a class to change. (不应有多于1个的原因使得一个类发生变化)
  - One class, one responsibility. (一个类，一个责任)

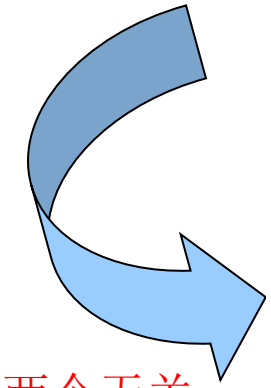
- 如果一个类包含了多个责任，那么将引起不良后果：
  - 引入额外的包，占据资源
  - 导致频繁的重新配置、部署等

- **The SRP is one of the simplest of the principle, and one of the hardest to get right. (最简单的原则，却是最难做好的原则)**

SRP的一个反例

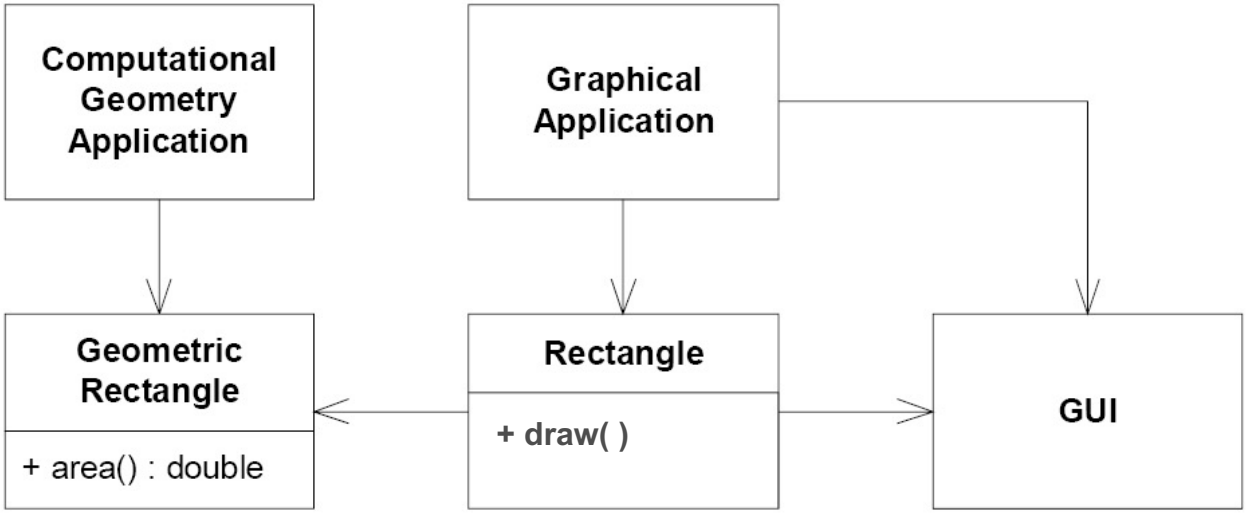


Rectangle类既包含GUI的绘图功能draw()-在GUI上显示矩形，
又包括几何计算功能area()-计算面积。
该设计违反了SRP。



通过分解，将两个无关的责任分离开来，
分别放置在两个类中：

- Geometric Rectangle类负责计算面积，
- Rectangle类负责在GUI上绘图。



Single Responsibility Principle

- **Two resposibilities**
  - Connection Management
  - Data Communication

```
interface Modem {
 public void dial(String pno);
 public void hangup();

 public void send(char c);
 public char recv();
}
```

```
interface DataChannel {
 public void send(char c);
 public char recv();
}
```

```
interface Connection {
 public void dial(String phn);
 public char hangup();
}
```

(2) Open/Closed Principle (OCP)

（面向变化的）开放/封闭原则

(OCP) The Open-Closed Principle

- **Classes should be open for extension (对扩展性的开放)**
  - This means that the behavior of the module can be extended. That we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications. (模块的行为应是可扩展的，从而该模块可表现出新的行为以满足需求的变化)

- **But closed for modification. (对修改的封闭)**
  - The source code of such a module is inviolate. No one is allowed to make source code changes to it. (但模块自身的代码是不应被修改的)
  - The normal way to extend the behavior of a module is to make changes to that module. (扩展模块行为的一般途径是修改模块的内部实现)
  - A module that cannot be changed is normally thought to have a fixed behavior. (如果一个模块不能被修改，那么它通常被认为是具有固定的行为)

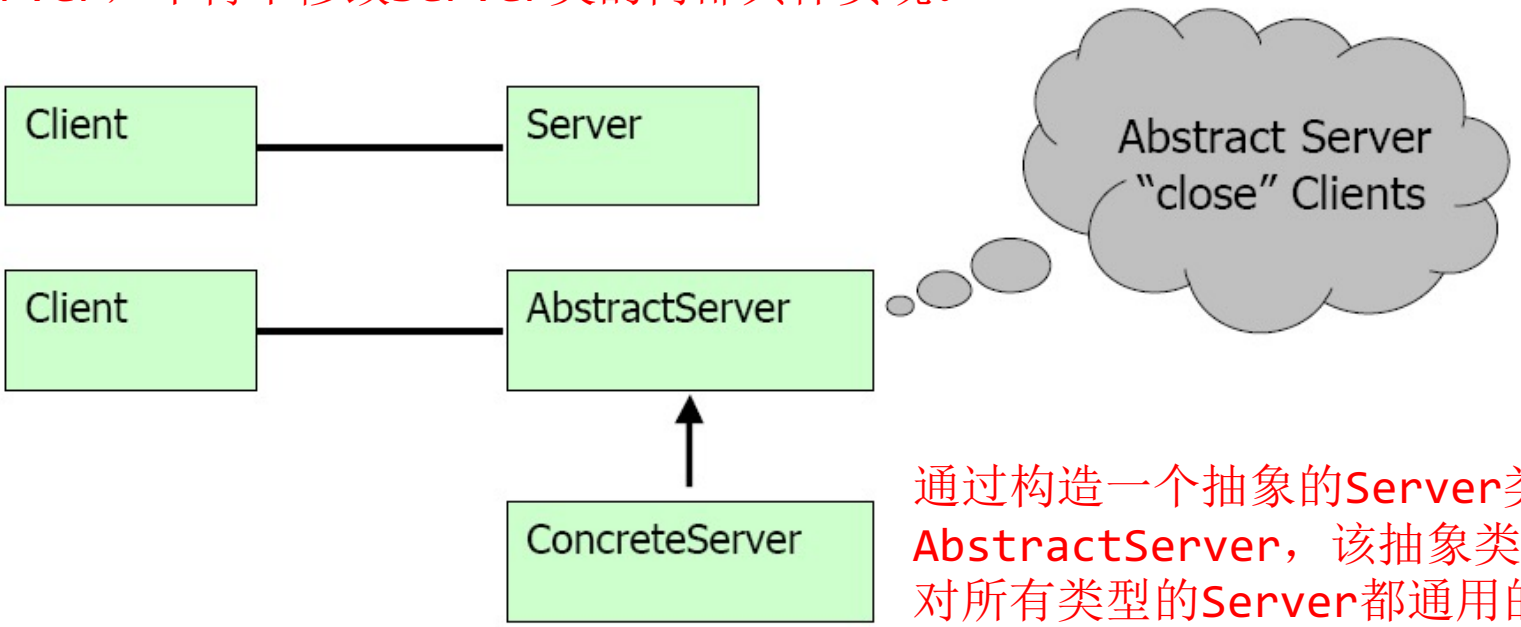
Open Closed Principle

- **Key: abstraction (关键的解决方案：抽象技术)**

- **"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification“, i.e., change a class' behavior using inheritance and composition/delegation**

OCP的一个反例

如果有多种类型的Server，那么针对每一种新出现的Server，不得不修改Server类的内部具体实现。



通过构造一个抽象的Server类：AbstractServer，该抽象类中包含针对所有类型的Server都通用的代码，从而实现了对修改的封闭；
当出现新的Server类型时，只需从该抽象类中派生出具体的子类ConcreteServer即可，从而支持了对扩展的开放。

Open Closed Principle

```
// Open-Close Principle - Bad example
 class GraphicEditor {

 public void drawShape(Shape s) {
    if (s.m_type==1)
        drawRectangle(s);
    else if (s.m_type==2)
        drawCircle(s);
    }
    public void drawCircle(Circle r)
        {....}
    public void drawRectangle(Rectangle r)
        {....}
 }
```

```
class Shape {
    int m_type;
 }

 class Rectangle extends Shape {
    Rectangle() {
        super.m_type=1;
    }
 }

 class Circle extends Shape {
    Circle() {
        super.m_type=2;
    }
 }
```

一大堆复杂的
if-else
/switch-case
结构，维护起来
非常麻烦

Open Closed Principle - Improved

```
// Open-Close Principle - Good example


class GraphicEditor {
   public void drawShape(Shape s) {
     s.draw();
   }
}

class Shape {
   abstract void draw();
}

class Rectangle extends Shape {
   public void draw() {
   // draw the rectangle
   }
}
```

(3) Liskov Substitution Principle (LSP)

Liskov替换原则

(LSP) The Liskov Substitution Principle

- **"Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it", i.e., subclasses should behave nicely when used in place of their base class**

- **LSP: Subtypes must be substitutable for their base types. (子类型必须能够替换其基类型)**

- **Derived Classes must be usable through the base class interface without the need for the client to know the difference. (派生类必须能够通过其基类的接口使用，客户端无需了解二者之间的差异)**

→ Already discussed in Section 5-2 Reusability

(4) Interface Segregation Principle (ISP)

接口隔离原则

Interface Segregation Principle

- **"Clients should not be forced to depend upon interfaces that they do not use", i.e., keep interfaces small. 不能强迫客户端依赖于它们不需要的接口：只提供必需的接口**

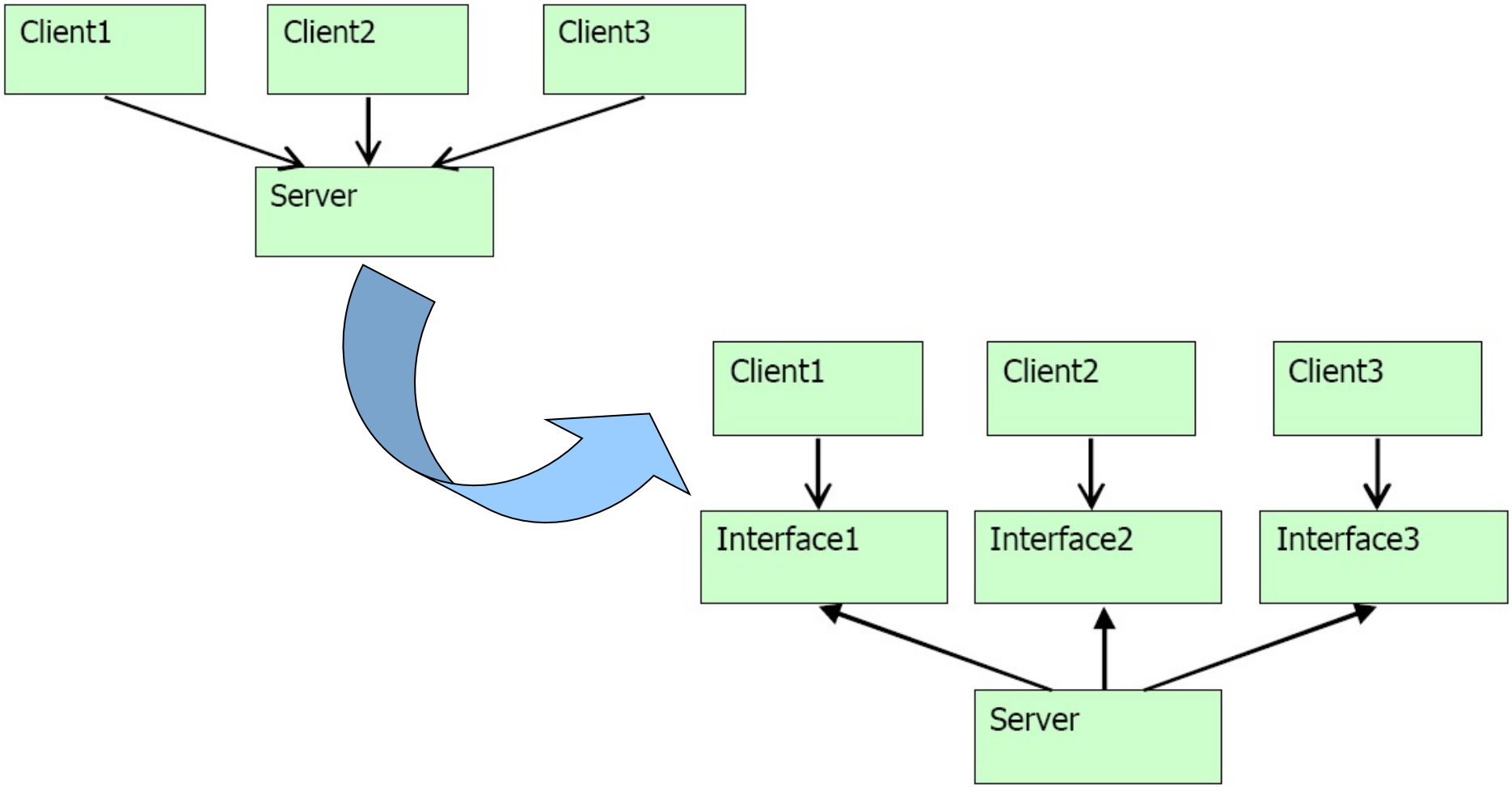
- **Don't force classes so implement methods they can't (Swing/Java)**
- **Don't pollute interfaces with a lot of methods**
- **Avoid 'fat' interfaces**

(ISP) The Interface Segregation

- **Clients should not be forced to depend upon methods that they do not use. (客户端不应依赖于它们不需要的方法)**
- **Interfaces belong to clients, not to hierarchies.**

- **This principle deals with the disadvantages of "fat" interfaces. ("胖" 接口具有很多缺点)**
- **Classes that have "fat" interfaces are classes whose interfaces are not cohesive. (不够聚合)**
  - The interfaces of the class can be broken up into groups of member functions. (胖接口可分解为多个小的接口)
  - Each group serves a different set of clients (不同的接口向不同的客户端提供服务).
  - Thus some clients use one group of member functions, and other clients use the other groups. (客户端只访问自己所需要的端口)

(ISP) The Interface Segregation Principle



Interface Segregation Principle

```
//bad example (polluted interface)
interface Worker {
    void work();
    void eat();
}
ManWorker implements Worker {
    void work() {…};
    void eat() {…};
}
RobotWorker implements Worker {
    void work() {…};
    void eat() {//Not Appliciable
                for a RobotWorker};
}
```

Solution: split into two

```
interface Workable {
   public void work();
}

interface Feedable{
   public void eat();
}
```

Interface Segregation Principle

```
interface Workable {
    public void work();
}

interface Feedable{
    public void eat();
}

ManWorker implements Workable, Feedable {

    void work() {…};

    void eat() {…};

}

RobotWorker implements Workable {

    void work() {…};

}
```

(5) Dependency Inversion Principle (DIP)

依赖转置原则

(DIP) The Dependency Inversion Principle

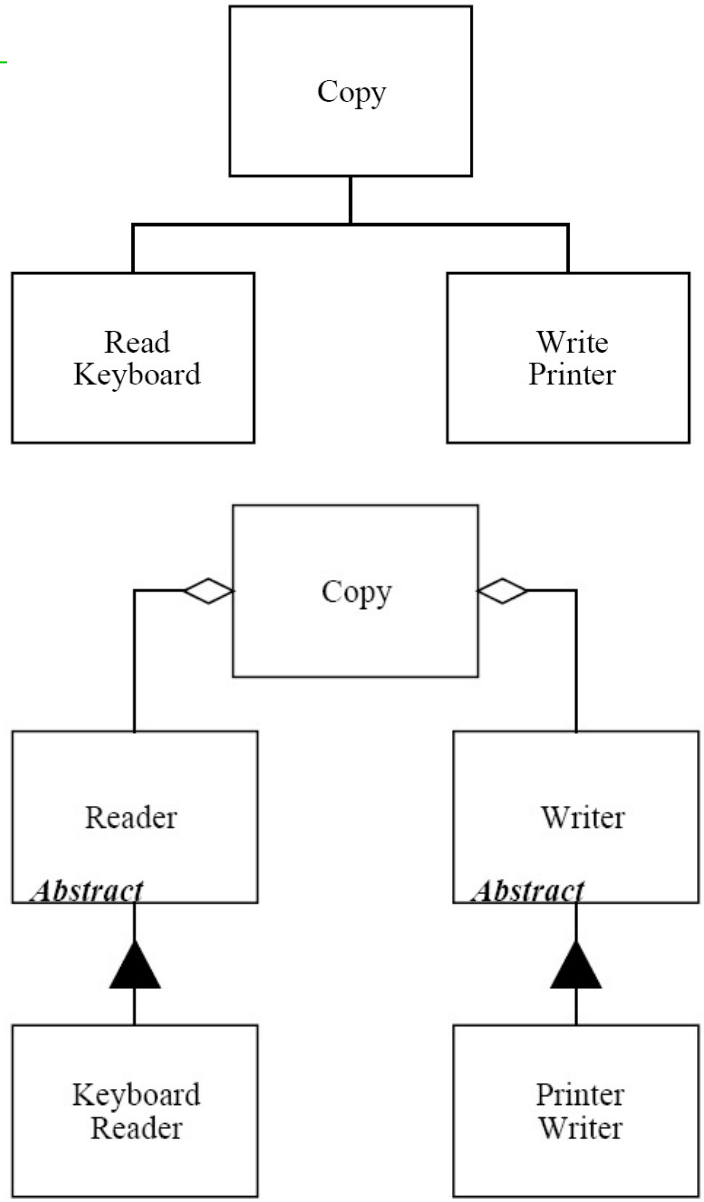
- **High level modules should not depend upon low level modules. Both should depend upon abstractions.**
  - Abstractions should not depend upon details (抽象的模块不应依赖于具体的模块)
  - Details should depend upon abstractions (具体应依赖于抽象)

- **Lots of interfaces and abstractions should be used!**

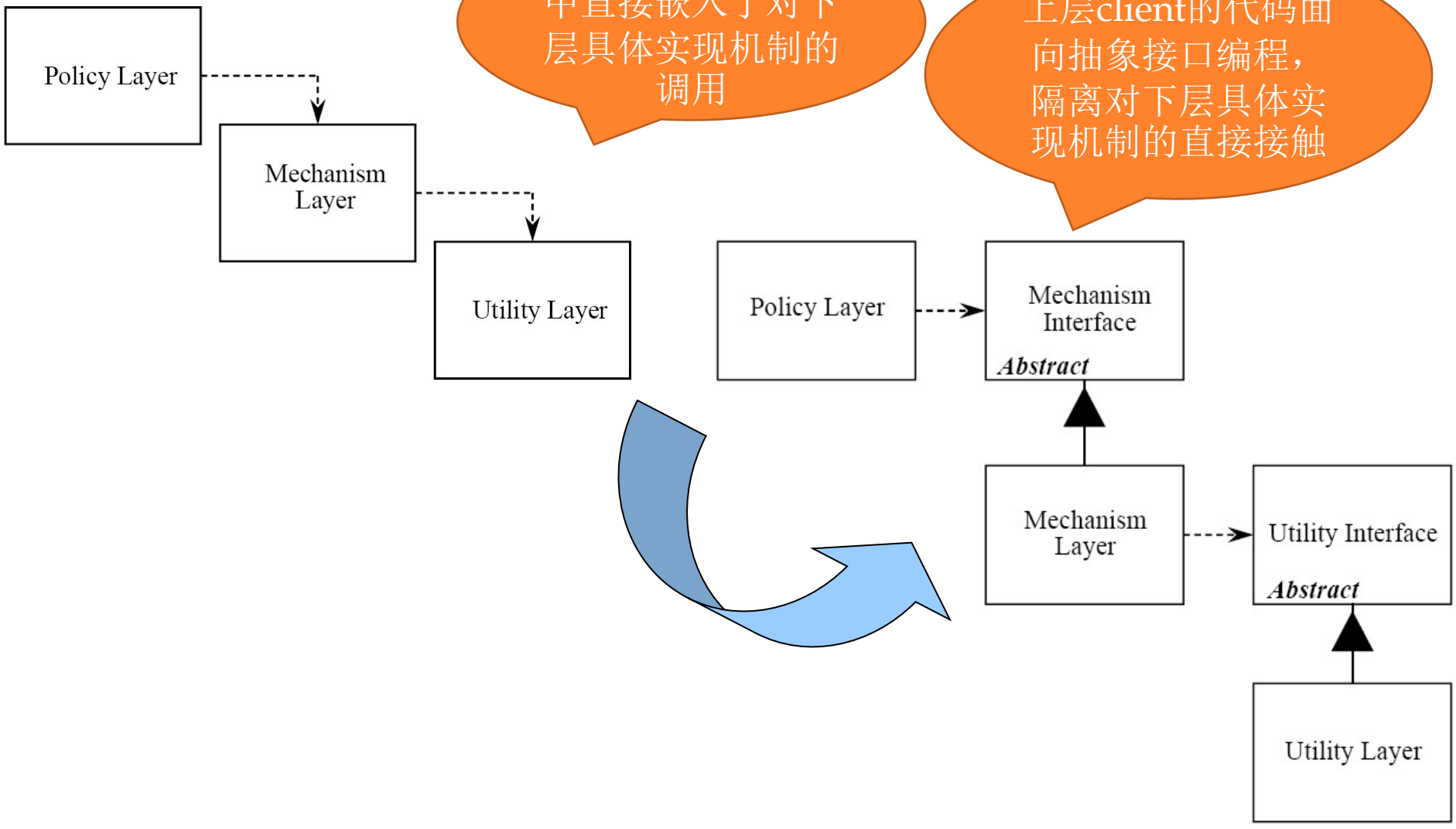
Example: the "Copy" program

```
void Copy(OutputStream dev) {
   int c;
   while ((c = ReadKeyboard()) != EOF)
        if (dev == printer)
           writeToPrinter(c);
        else
           writeToDisk(c);
}
```

```
interface Reader {
  public int read();
}
interface Writer {
  public int write(c);
}
class Copy {
  void Copy(Reader r, Writer w) {
     int c;
     while (c=r.read() != EOF)
        w.write(c);
  }
}
```



DIP: another example



Dependency Inversion Principle

```
//DIP - bad example
public class EmployeeService {
        private EmployeeFinder emFinder; //concrete class, not abstract.
        //Can access a SQL DB for instance
        public Employee findEmployee(…) {
                emFinder.findEmployee(…)
        }
}
//DIP - fixed
public class EmployeeService {
        private IEmployeeFinder emFinder
        //depends on an abstraction, no an implementation

        public Employee findEmployee(…) {
           emFinder.findEmployee(…)
        }
}
```

- **Now its possible to change the finder to be a** `XmEmployeeFinder`, `DBEmployeeFinder`, `FlatFileEmployeeFinder`, `MockEmployeeFinder`….

5 Grammar-based construction

语法驱动的构造

String/Stream based I/O

- **Some program modules take input or produce output in the form of a sequence of bytes or a sequence of characters, which is called a *string* when it's simply stored in memory, or a *stream* when it flows into or out of a module.** 有一类应用，从外部读取文本数据，在应用中做进一步处理。
- **Concretely, a sequence of bytes or characters might be:**
  - A file on disk, in which case the specification is called the *file format* 输入文件有特定格式，程序需读取文件并从中抽取正确的内容
  - Messages sent over a network, in which case the specification is a *wire protocol* 从网络上传输过来的消息，遵循特定的协议
  - A command typed by the user on the console, in which case the specification is a *command line interface* 用户在命令行输入的指令，遵循特定的格式
  - A string stored in memory 内存中存储的字符串，也有格式需要

String/Stream based I/O

```
22  UsageLog ::= <2019-01-02,15:30:00,Wechat,1>
23  UsageLog ::= <2019-01-03,11:00:00,Weibo,10>
24  UsageLog ::= <2019-01-03,09:00:00,BaiduMap,400>
25
26
27  App ::= <Wechat,Tencent,13.2,"The most popular social networking App in China","Social network">
28  App ::= <QQ,Tencent,29.2,"The second popular social networking App in China","Social network">
29  App ::= <Weibo,Sina,v0.2.3.4,"The third popular social networking App in China","Social network">
30  App ::= <Didi,Didi,ver03.32,"The most popular car sharing App in China","Travel">
31  App ::= <Eleme,Eleme,20V0.03,"The most popular online food ordering App in China","Food">
32  App ::= <BaiduMap,Baidu,2.9000000.20v03,"The most popular map App in China","Travel">
33
34  Period ::= Day
35
36
37  Relation ::= <Wechat,QQ>
38  Relation ::= <Wechat,Eleme>
39  Relation ::= <Didi,BaiduMap>
```

```
1   CentralUser ::= <TommyWong,30,M>
2
3   SocialTie ::= <TommyWong, LisaWong, 0.98>
4   SocialTie ::= <TommyWong, TomWong, 0.2>
5   SocialTie ::= <TomWong, FrankLee, 0.71>
6   SocialTie ::= <FrankLee, DavidChen, 0.02>
7   SocialTie ::= <TommyWong, DavidChen, 0.342>
8   SocialTie ::= <JackMa, PonyMa, 0.999>
9
10  Friend ::= <LisaWong, 25, F>
11  Friend ::= <TomWong, 61, M>
12  Friend ::= <FrankLee, 42, M>
13  Friend ::= <DavidChen, 55, M>
14  Friend ::= <JackMa, 58, M>
15  Friend ::= <PonyMa, 47, M>
```

```
C:\>curl --HEAD http://cms.hit.edu.cn
HTTP/1.1 200 OK
Date: Fri, 20 Apr 2019 14:00:00 GMT
Server: Apache/2.4.33 (Unix) mod_jk/1.2.43
Accept-Ranges: bytes
Vary: Accept-Encoding
Connection: close
Content-Type: text/html
```

The notion of a grammar

- **For these kinds of sequences, the notion of a grammar is a good choice for design:**
  - It can not only help to distinguish between legal and illegal sequences, but also to parse a sequence into a data structure a program can work with. 使用grammar判断字符串是否合法，并解析成程序里使用的数据结构
  - The data structure produced from a grammar will often be a recursive data type. 通常是递归的数据结构
- **Regular expression** 正则表达式
  - It is a widely-used tool for many string-processing tasks that need to disassemble a string, extract information from it, or transform it.
- **A parser generator is a kind of tool that translate a grammar automatically into a parser for that grammar.** 根据语法，开发一个它的解析器，用于后续的解析

(1) Constituents of a Grammar

Terminals: Literal Strings in a Grammar

- To describe a string of symbols, whether they are bytes, characters, or some other kind of symbol drawn from a fixed set, we use **a compact representation called a grammar.**
- **A grammar defines a set of strings.** 用语法定义一个“字符串”
  - For example, the grammar for URLs will specify the set of strings that are legal URLs in the HTTP protocol.
- **The literal strings in a grammar are called terminals** 终止节点、叶节点
  - They’re called terminals because they are the leaves of a parse tree that represents the structure of the string. 语法解析树的叶子节点
  - They don’t have any children, and can’t be expanded any further. 无法再往下扩展
  - We generally write terminals in quotes, like `‘http’` or `‘:’`. 通常表示为字符串

Nonterminals and Productions in a Grammer

- **A grammar is described by a set of productions 产生式节点, where each production defines a nonterminal 非终止节点**
  - A nonterminal is like a variable that stands for a set of strings, and the production as the definition of that variable in terms of other variables (nonterminals), operators, and constants (terminals). 遵循特定规则，利用操作符、终止节点和其他非终止节点，构造新的字符串
  - Nonterminals are internal nodes of the tree representing a string.
- **A production in a grammar has the form**
  - nonterminal ::= expression of terminals, nonterminals, and operators
- **One of the nonterminals of the grammar is designated as the root.**
  - The set of strings that the grammar recognizes are the ones that match the root nonterminal.
  - This nonterminal is often called root or start. 根节点

(2) Operators in a Grammar

Three Basic Grammar Operators

- **The three most important operators in a production expression are:**
  - Concatenation 连接, represented not by a symbol, but just a space:

    `x ::= y z` x matches y followed by z
  - Repetition 重复, represented by *:

    `x ::= y*` x matches zero or more y
  - Union, also called alternation 选择, represented by |:

    `x ::= y | z` x matches either y or z

Grouping operators using parentheses

- **By convention, the postfix operators `*`, `?`, and `+` have highest precedence, which means they are applied first.**
- **Concatenation is applied next.**
- **Alternation `|` has lowest precedence, which means it is applied last.**

- **Parentheses can be used to override precedence:**
  - `x ::= (y z | a b)*` `x` matches zero or more `yz` or `ab` pairs
  - `m ::= a (b|c) d` `m` matches `a`, followed by either `b` or `c`, followed by `d`

A small example

- `url ::= 'http://mit.edu/'`
  - Use these operators to generalize our url grammar to match some other hostnames, such as `http://stanford.edu/ and http://google.com/.`
- `url ::= 'http://' hostname '/'`
  - The `url` nonterminal matches strings that start with the literal string `http://`, followed by a match to the `hostname` nonterminal, followed by the literal string `/`.
- `hostname ::= 'mit.edu' | 'stanford.edu' | 'google.com'`
  - A `hostname` can match one of the three literal strings, `mit.edu` or `stanford.edu` or `google.com`.
- So this grammar represents the set of three strings.

A small example

- `hostname ::= 'mit.edu' | 'stanford.edu' | 'google.com'`
- To make it represent more URLs, we allow any lowercase word in place of `mit, stanford, google, com` and `edu`:

```
url ::= 'http://' hostname '/'
hostname ::= word '.' word
word ::= ('a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i'
          | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q'
          | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z')*
```

  - The new word rule matches a string of zero or more lowercase letters, so the overall grammar can now match `http://alibaba.com/` and `http://zyxw.edu/` as well.
  - Unfortunately word can also match an empty string, so this `url` grammar also matches `http://./`, which is not a legal URL.

A small example

```
word ::= ('a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i'
          | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q'
          | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z')
        ('a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i'
          | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q'
          | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z')*
```

Too complicated!

More grammar operators

- **Additional operators are just syntactic sugar (i.e., they're equivalent to combinations of the big three operators):**
  - Optional (0 or 1 occurrence), represented by `?`:

    `x ::= y?` an `x` is a `y` or is the empty string
  - 1 or more occurrences: represented by `+`:

    `x ::= y+` an `x` is one or more `y` (equivalent to `x ::= y y*`)
  - A character class [...], representing the length-1 strings containing any of the characters listed in the square brackets:

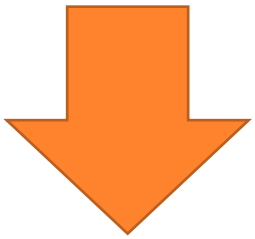
    `x ::= [a-c]` is equivalent to `x ::= 'a' | 'b' | 'c'`

    `x ::= [aeiou]` is equivalent to `x ::= 'a' | 'e' | 'i' | 'o' | 'u'`
  - An inverted character class [^...], representing the length-1 strings containing any character not listed in the brackets:

    `x ::= [^a-c]` is equivalent to `x ::= 'd' | 'e' | 'f' | ...`
    (all other characters)

Go back to the example

```
url ::= 'http://' hostname '/'
hostname ::= word '.' word
word ::= ('a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i'
        | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q'
        | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z')
      ('a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i'
        | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q'
        | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z')*
```



```
url ::= 'http://' hostname '/'
hostname ::= word '.' word
word ::= [a-z]+
```

(3) Recursion in grammars

Recursion in grammars

- **Hostnames can have more than two components, and there can be an optional port number:**

```
http://didit.csail.mit.edu:4949/
```

- **To handle this kind of string, the grammar is now:**

```
url ::= 'http://' hostname (':' port)? '/'
hostname ::= word '.' hostname | word '.' word
port ::= [0-9]+
word ::= [a-z]+
```

- `hostname` **is now defined recursively in terms of itself.**
- **Using the repetition operator, we could also write hostname without recursion, like this:**

```
hostname ::= (word '.')+ word
```

Exercise

- **Consider this grammar:**

```
S ::= (B C)* T

B ::= M+ | P B P

C ::= B | E+
```

- **What are the nonterminals in this grammar?**
- **What are the terminals in this grammar?**
- **Which productions are recursive?**

Exercise

- **Which strings match the `root` nonterminal of this grammar?**

```
root ::= 'a'+ 'b'* 'c'?
```

- **Strings**

```
aabcc
bbbc
aaaaaaaa
abc
abab
aac
```

Exercise

- **Which strings match the root nonterminal of this grammar?**

```
root    ::= integer ('-' integer)+
integer ::= [0-9]+
```

- **Strings**

```
617
617-253
617-253-1000
---
integer-integer-integer
5--5
3-6-293-1
```

Exercise

- **Which strings match the `root` nonterminal of this grammar?**

```
root    ::= (A B)+
A       ::= [Aa]
B       ::= [Bb]
```

- **Strings**

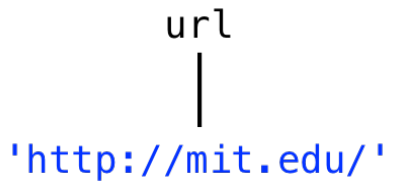
```
aaaBBB
abababab
aBAbabAB
AbAbAbA
```

(4) Parse Trees

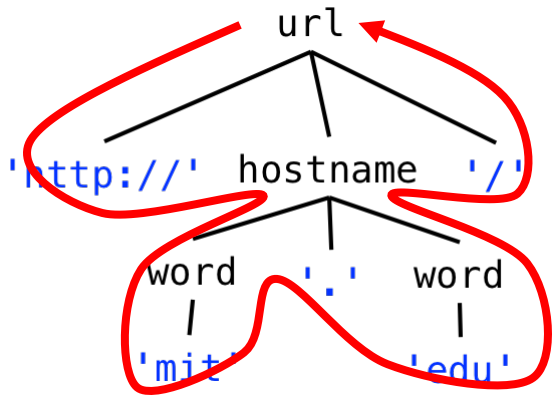
Parse Tree

- **Matching a grammar against a string can generate a *parse tree* that shows how parts of the string correspond to parts of the grammar.**
  - The leaves of the parse tree are labeled with terminals, representing the parts of the string that have been parsed.
  - They don't have any children, and can't be expanded any further.
  - If we concatenate the leaves together, we get back the original string.

```
url ::= 'http://mit.edu/'
```



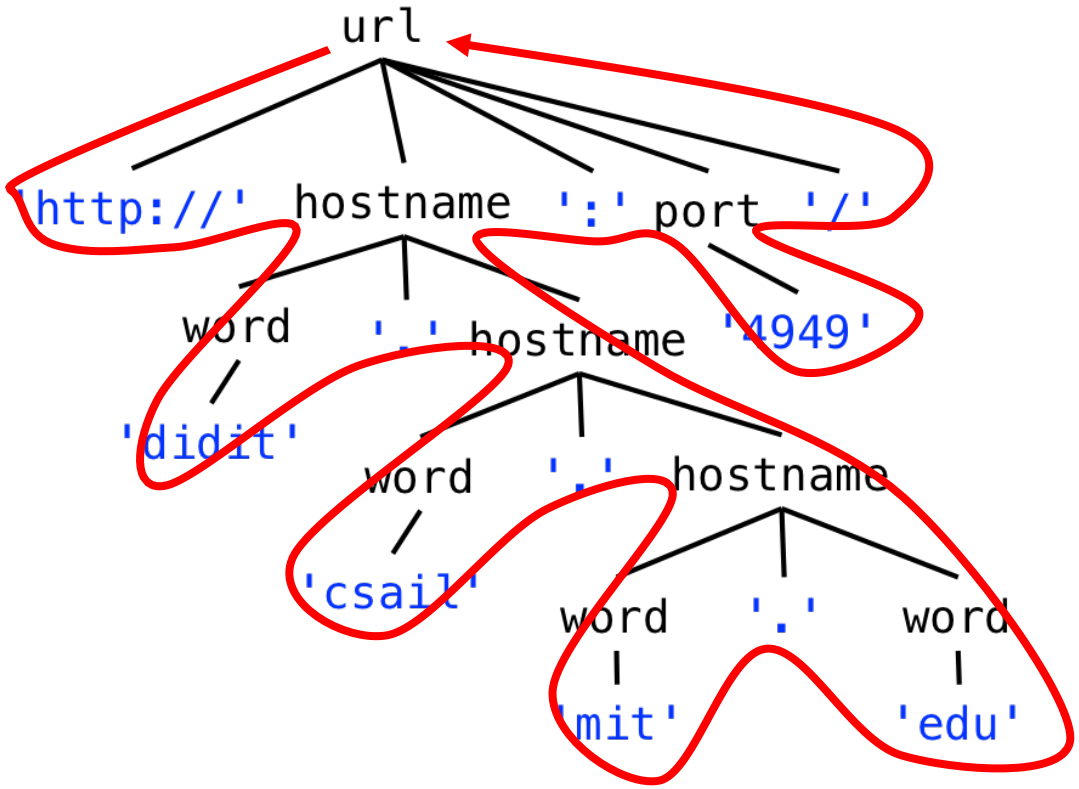

```
url ::= 'http://' hostname '/'
hostname ::= word '.' word
word ::= [a-z]+
```



Parse Tree

```
url ::= 'http://' hostname (':' port)? '/'
hostname ::= word '.' hostname | word '.' word
port ::= [0-9]+
word ::= [a-z]+
```

http://didit.csail.mit.edu:4949/



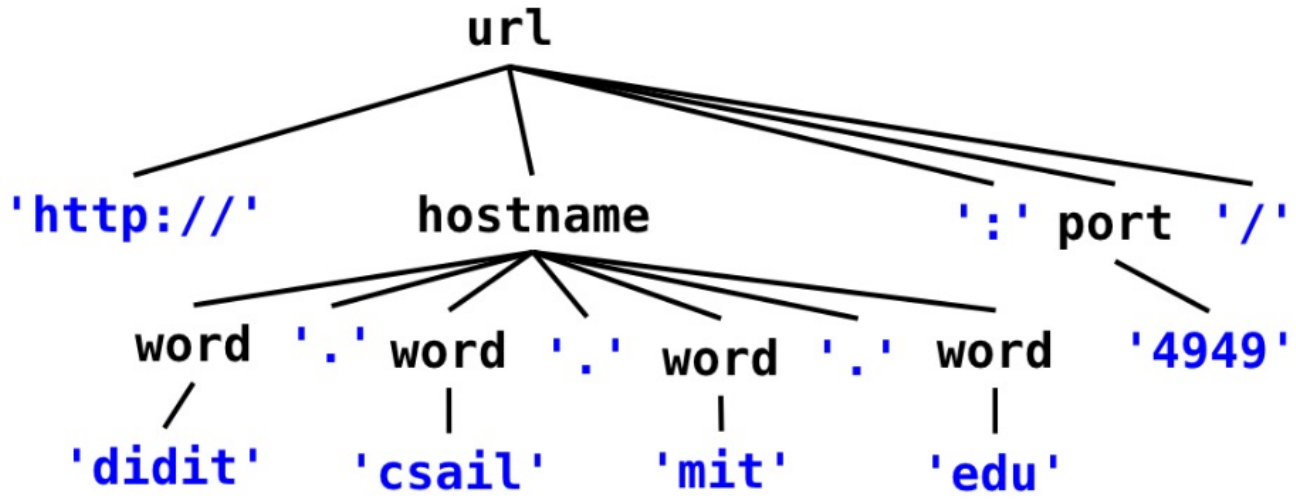
Parse Tree

- **If the same string was matched against this grammar with a non-recursive `hostname` rule:**

```
url ::= 'http://' hostname (':' port)? '/'
hostname ::= (word '.')+ word
port ::= [0-9]+
word ::= [a-z]+
```

`http://didit.csail.mit.edu:4949/`

- **What does its parse tree looks like?**



More generalizations…

- **There are more things we should do to go farther:**
  - Generalizing `http` to support the additional protocols that URLs can have, such as `ftp`, `https`, …
  - Generalizing the `/` at the end to a slash-separated path, such as `http://didit.csail.mit.edu:4949/homework/lab1/`
  - Allowing hostnames with the full set of legal characters instead of just `a-z` such as `http://ou812.com/`

- **Can you do these?**

```
url ::= protocol '://' hostname (':' port)? '/' (word '/')*
protocol ::= 'ftp' | 'http' | 'https'
hostname ::= (word '.')+ word
port ::= [0-9]+
word ::= [a-z 0-9]+
```

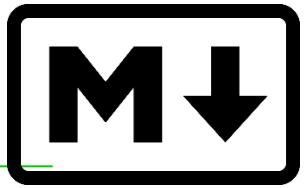
Exercise

- **We want the URL grammar to also match strings of the form:**
  - `https://websis.mit.edu/`
  - `ftp://ftp.athena.mit.edu/`
- **but not strings of the form:**
  - `ptth://web.mit.edu/`
  - `mailto:bitdiddle@mit.edu`
- **So we change the grammar to:**
- **What could you put in place of `TODO` to match the desirable URLs but not the undesirable ones?**
  - `word`
  - `'ftp' | 'http' | 'https'`
  - `('http' 's'?) | 'ftp'`
  - `('f' | 'ht') 'tp' 's'?`

```
url ::= protocol '://' hostname (':'
        port)? '/'
protocol ::= TODO
hostname ::= word '.' hostname |
             word '.' word
port ::= [0-9]+
word ::= [a-z]+
```

(5) Markdown and HTML

Markdown and HTML



- **Markup languages:** represents typographic style in text.

  https://daringfireball.net/projects/markdown/syntax

- **Markdown example for italics**

```
This is _italic_.
```

- **HTML example for italics**

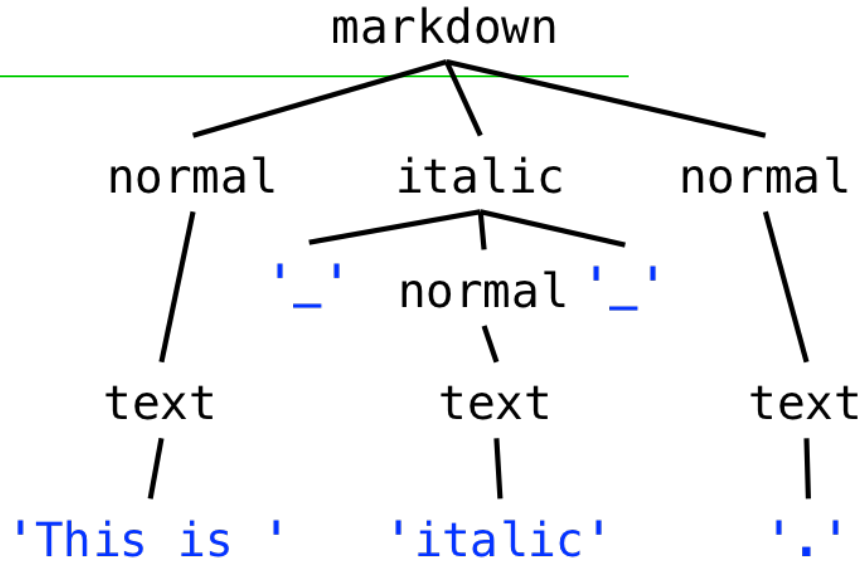
```
Here is an <i>italic</i> word.
```

- **For simplicity, we assume the plain text between the formatting delimiters is not allowed to use any formatting punctuation, like _ or <>.**

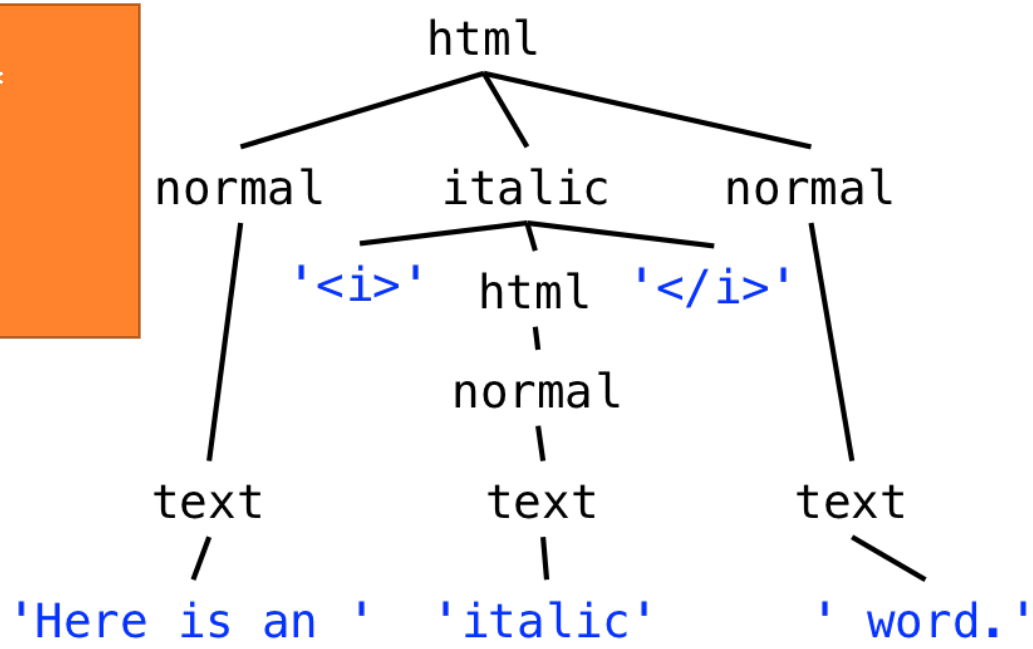
- **Can you write down their grammars?**

Markdown and HTML

```
markdown ::=  ( normal | italic ) *
italic ::= '_' normal '_'
normal ::= text
text ::= [^_]*
```




```
html ::=  ( normal | italic ) *
italic ::= '<i>' html '</i>'
normal ::= text
text ::= [^<>]*
```



Markdown and HTML

```
markdown ::=  ( normal | italic ) *
italic ::= '_' normal '_'
normal ::= text
text ::= [^_]*
```

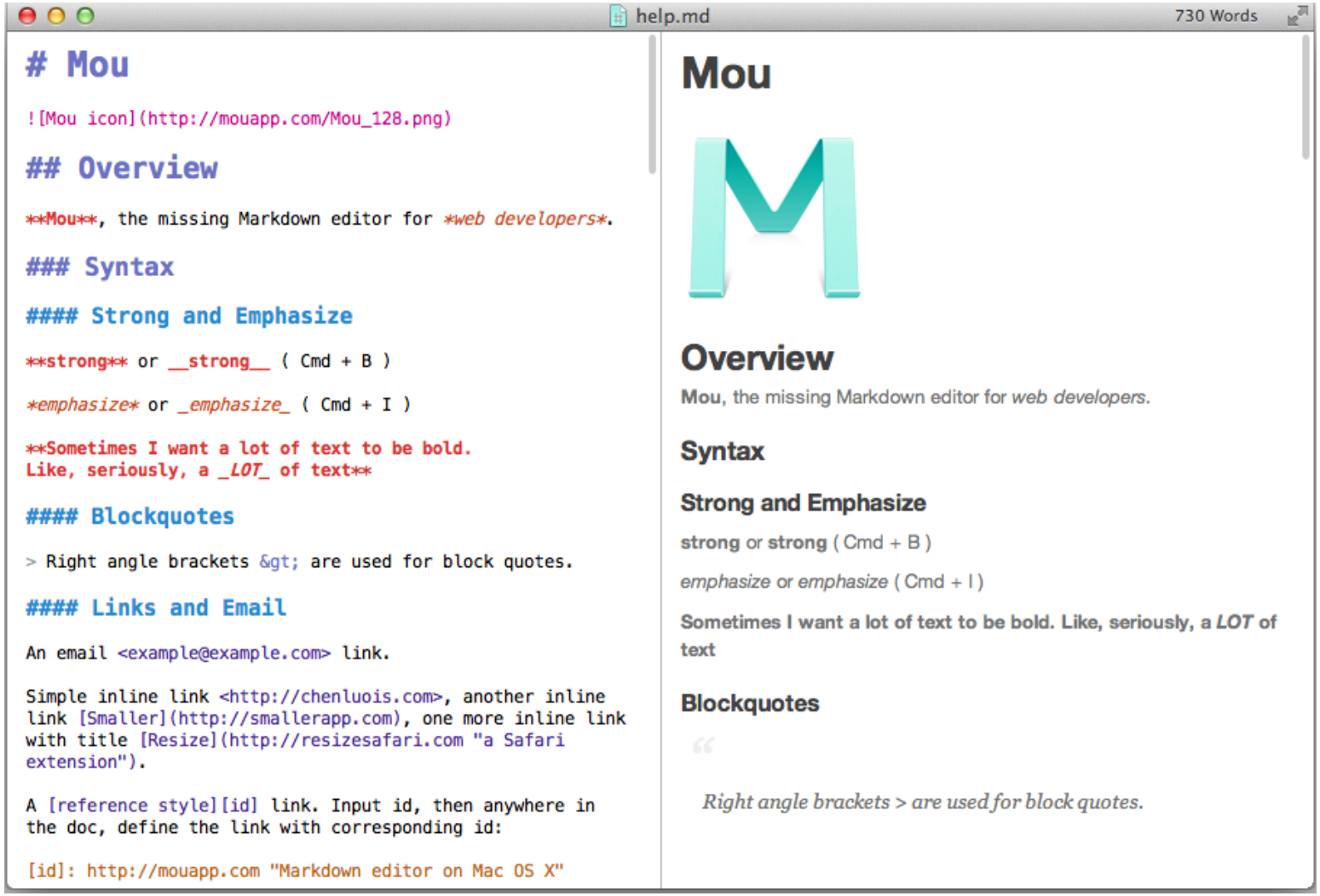
markdown:
`a_b_c_d_e`

html:
`a<i>b<i>c</i>d</i>e`

```
html ::=  ( normal | italic ) *
italic ::= '<i>' html '</i>'
normal ::= text
text ::= [^<>]*
```

If you match the specified grammar against it, which letters are inside matches to the italic nonterminal?

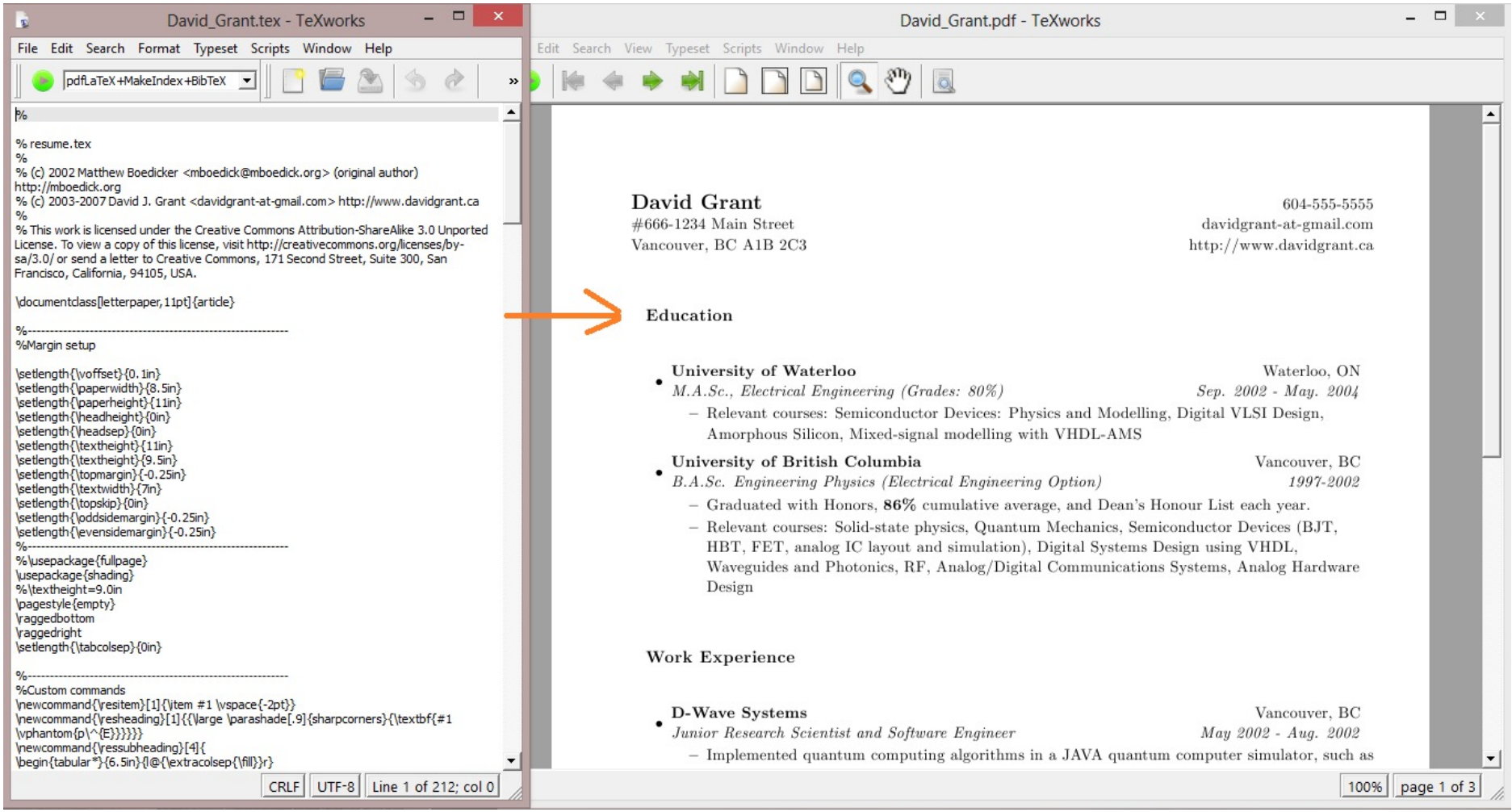
Documenting like programming: Markdown, LaTeX



Donald E. Knuth（高德纳）

- **Donald E. Knuth (1938-)**
- **Stanford University**
- **1974年图灵奖获得者**
  史上最年轻的图灵奖获得者
- 被誉为现代计算机科学的鼻祖
- 算法分析之父，为理论计算机科学的发展做出重要贡献
- 《计算机程序设计艺术》(The Art of Computer Programming)，计算机科学理论与技术的经典巨著，其作用与地位可与《几何原本》相比。
- **TeX的发明者**
- “计算机老顽童”

创造TEX和METAFONT



创造TEX和METAFONT

- 除了对排版美的追求，TEX使人们像编程一样写作文。

- TEX的版本号不是自然数列，也不是年份，而是从3开始，不断逼近圆周率(目前最新版本是3.14159265)，意思是说：这个东西趋近完美，不可能再有什么大的改进了

- 设立了一个奖项：谁发现TEX的一个错误，就付他2.56美元，第二个错误5.12美元，第三个10.24美元……以此类推

TEX

Developer(s)	Donald Knuth
Initial release	1978; 41 years ago
Stable release	3.14159265 / January 2014; 5 years ago
Repository	www.tug.org/svn/texlive/
Written in	WEB/Pascal
Operating system	Cross-platform
Type	Typesetting
License	Permissive free software
Website	tug.org

(6) Regular Grammars and Regular Expressions

Regular grammar

- **A regular grammar** has a special property: by substituting every nonterminal (except the root one) with its righthand side, you can reduce it down to a single production for the root, with only terminals and operators on the right-hand side. 正则语法：简化之后可以表达为一个产生式而不包含任何非终止节点
- Which of them are regular grammars?

```
url ::= 'http://' hostname (':' port)? '/'
hostname ::= word '.' hostname | word '.' word
port ::= [0-9]+
word ::= [a-z]+
```

```
markdown ::=  ( normal | italic ) *
italic ::= '_' normal '_'
normal ::= text
text ::= [^_]*
```

```
html ::=  ( normal | italic ) *
italic ::= '<i>' html '</i>'
normal ::= text
text ::= [^<>]*
```

Regular grammar

```
url ::= 'http://' hostname (':' port)? '/'
hostname ::= word '.' hostname | word '.' word
port ::= [0-9]+
word ::= [a-z]+
```

```
url ::= 'http://' ([a-z]+ '.')+ [a-z]+ (':' [0-9]+)? '/'
```

Regular!

```
markdown ::=  ( normal | italic ) *
italic ::= '_' normal '_'
normal ::= text
text ::= [^_]*
```

```
markdown ::= ([^_]* | '_' [^_]* '_' )*
```

Regular!

```
html ::=  ( normal | italic ) *
italic ::= '<i>' html '</i>'
normal ::= text
text ::= [^<>]*
```

```
html ::=  ( [^<>]* | '<i>' html '</i>' )*
```

Not regular!

Regular Expressions (*regex*)

- **The reduced expression of terminals and operators can be written in an even more compact form, called a regular expression. 正则表达式**
- **A regular expression does away with the quotes around the terminals, and the spaces between terminals and operators, so that it consists just of terminal characters, parentheses for grouping, and operator characters. 去除引号和空格，从而表达更简洁(更难懂)**

```
markdown ::= ([^_]* | '_' [^_]* '_' )*
```

```
markdown ::= ([^_]*|_[^_]*_)*
```

- **Regular expressions are also called *regex* for short.**
  - A regex is far less readable than the original grammar, because it lacks the nonterminal names that documented the meaning of each subexpression.
  - But a regex is fast to implement, and there are libraries in many programming languages that support regular expressions.

Some special operators in regex

- `.` **any single character**
- `\d` **any digit, same as** `[0-9]`
- `\s` **any whitespace character, including space, tab, newline**
- `\w` **any word character, including underscore, same as** `[a-zA-Z_0-9]`
- `\., \(, \), \*, \+, ...`

escapes an operator or special character so that it matches literally

An example

- **Original:**

```
'http://' ([a-z]+ '.')+ [a-z]+ (':' [0-9]+)? '/'
```

- **Compact:**

```
http://([a-z]+.)+[a-z]+(:[0-9]+)?/
```

- **With escape:**

```
http://([a-z]+\.)+[a-z]+(:[0-9]+)?/
```

Exercise

- **Consider the following regular expression:**

```
[A-G]+(♭|♯)?
```

- **Which of the following strings match the regular expression?**
  - A♭
  - C♯
  - ABK♭
  - A♭B
  - GFE

Context-Free Grammars

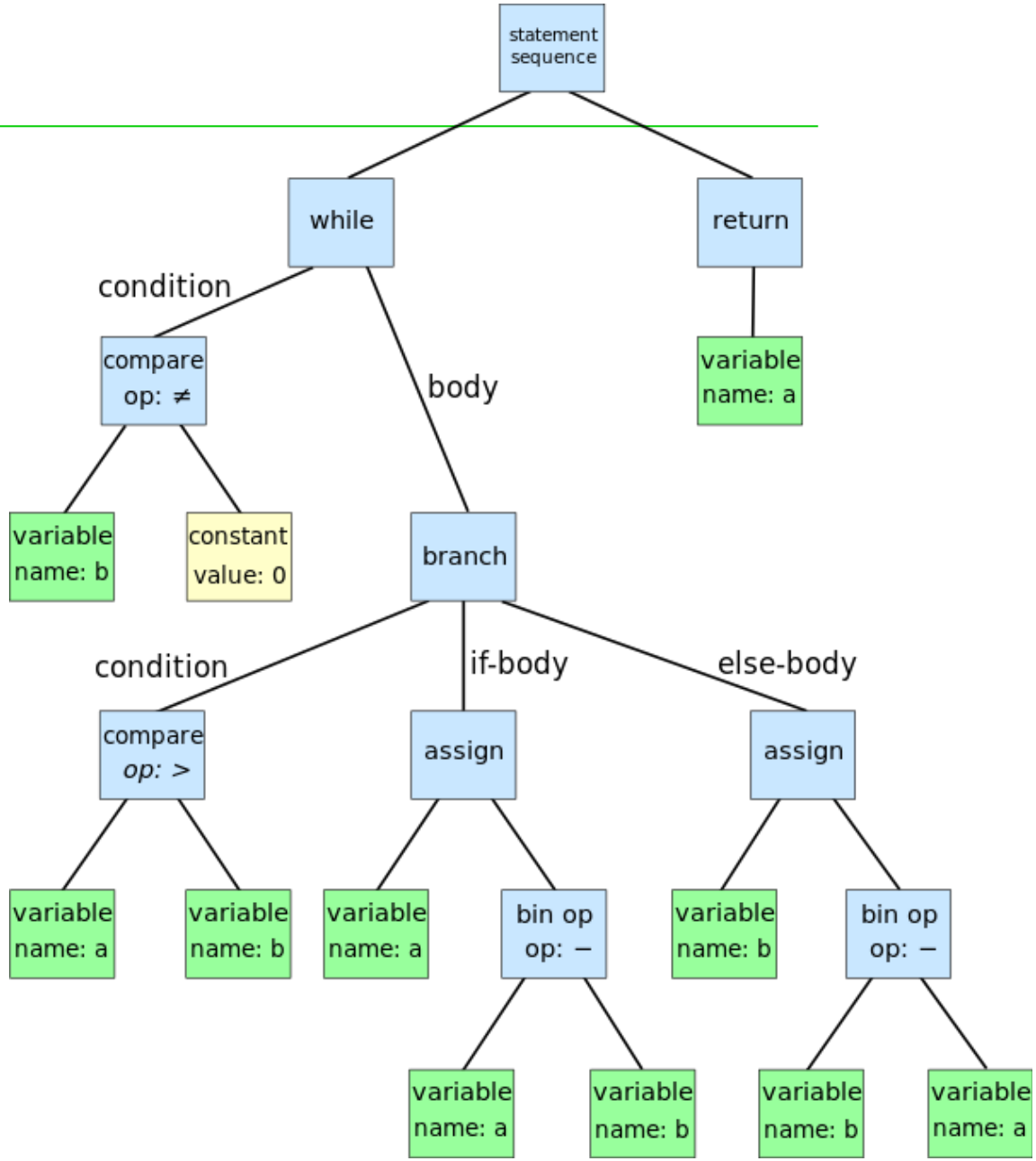
- In general, a language that can be expressed with a system of grammars is called context-free.
  - Not all context-free languages are also regular; that is, some grammars can't be reduced to single nonrecursive productions.
  - The HTML grammar is context-free but not regular.
- The grammars for most programming languages are also context-free.
- In general, any language with nested structure (like nesting parentheses or braces) is context-free but not regular.

Java grammar

```
statement ::=
  '{' statement* '}'
| 'if' '(' expression ')' statement ('else' statement)?
| 'for' '(' forinit? ';' expression? ';' forupdate? ')' statement
| 'while' '(' expression ')' statement
| 'do' statement 'while' '(' expression ')' ';'
| 'try' '{' statement* '}' ( catches | catches? 'finally' '{' statement* '}' )
| 'switch' '(' expression ')' '{' switchgroups '}'
| 'synchronized' '(' expression ')' '{' statement* '}'
| 'return' expression? ';'
| 'throw' expression ';'
| 'break' identifier? ';'
| 'continue' identifier? ';'
| expression ';'
| identifier ':' statement
| ';'
```

Java AST

```
while (b != 0) {
    if (a > b)
        a = a - b;
    else
        b = b - a;
}
return a;
```



(7) * Parsers

Parser 将输入文本转为parse tree

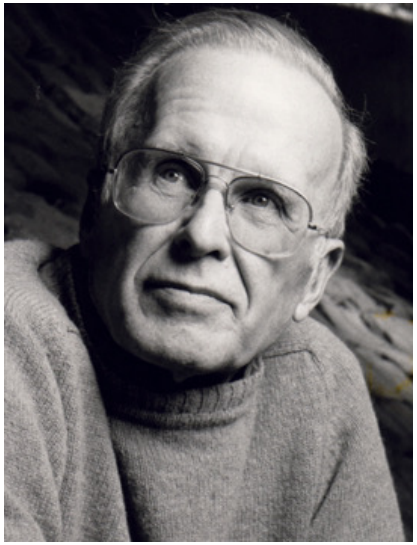
- **A parser takes a sequence of characters and tries to match the sequence against the grammar. parser：输入一段文本，与特定的语法规则建立匹配，输出结果**
- **The parser typically produces a parse tree, which shows how grammar productions are expanded into a sentence that matches the character sequence. parser：将文本转化为parse tree**
  - The root of the parse tree is the starting nonterminal of the grammar.
  - Each node of the parse tree expands into one production of the grammar.
- **The final step of parsing is to do something useful with this parse tree. 利用产生的parse tree，进行下一步的处理**
- **A recursive abstract data type that represents a language expression is called an *abstract syntax tree* (AST).**

Parser Generator 根据语法定义生成parser

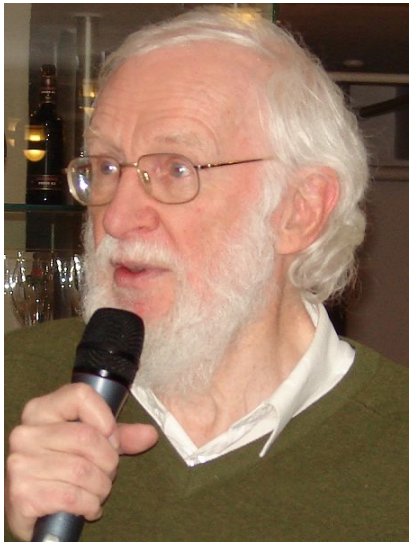
- **A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. Parser generator是一个工具，根据语法规则生成一个parser程序**
  - Read http://web.mit.edu/6.031/www/sp17/classes/18-parsers
    This is not the mandatory contents of this course.
- **More broadly:**
  - A parser generator is a programming tool that creates a parser, interpreter, or compiler from some form of formal description of a language.
  - The input may be a text file containing the grammar written in BNF or EBNF that defines the syntax of a programming language. 输入是遵循BNF或EBNF格式的文本文件
  - The output is some source code of the parser for the grammar. 输出为parser的源代码

Backus Normal Form (BNF) 巴克斯范式

- **1959年6月，Backus Normal Form (BNF)首次提出，以递归形式描述语言的各种成分，凡遵守其规则的程序就可保证语法上的正确性。**
  - 经过Peter Naur的改进与完善以及Niklaus Wirth的扩充，形成了EBNF（Extended BNF），也就是目前使用的BNF。
  - 经Donald Knuth 的建议，BNF中的N变成了Naur (Backus-Naur Form)。



John Backus (1924-2007)
1977年图灵奖得主

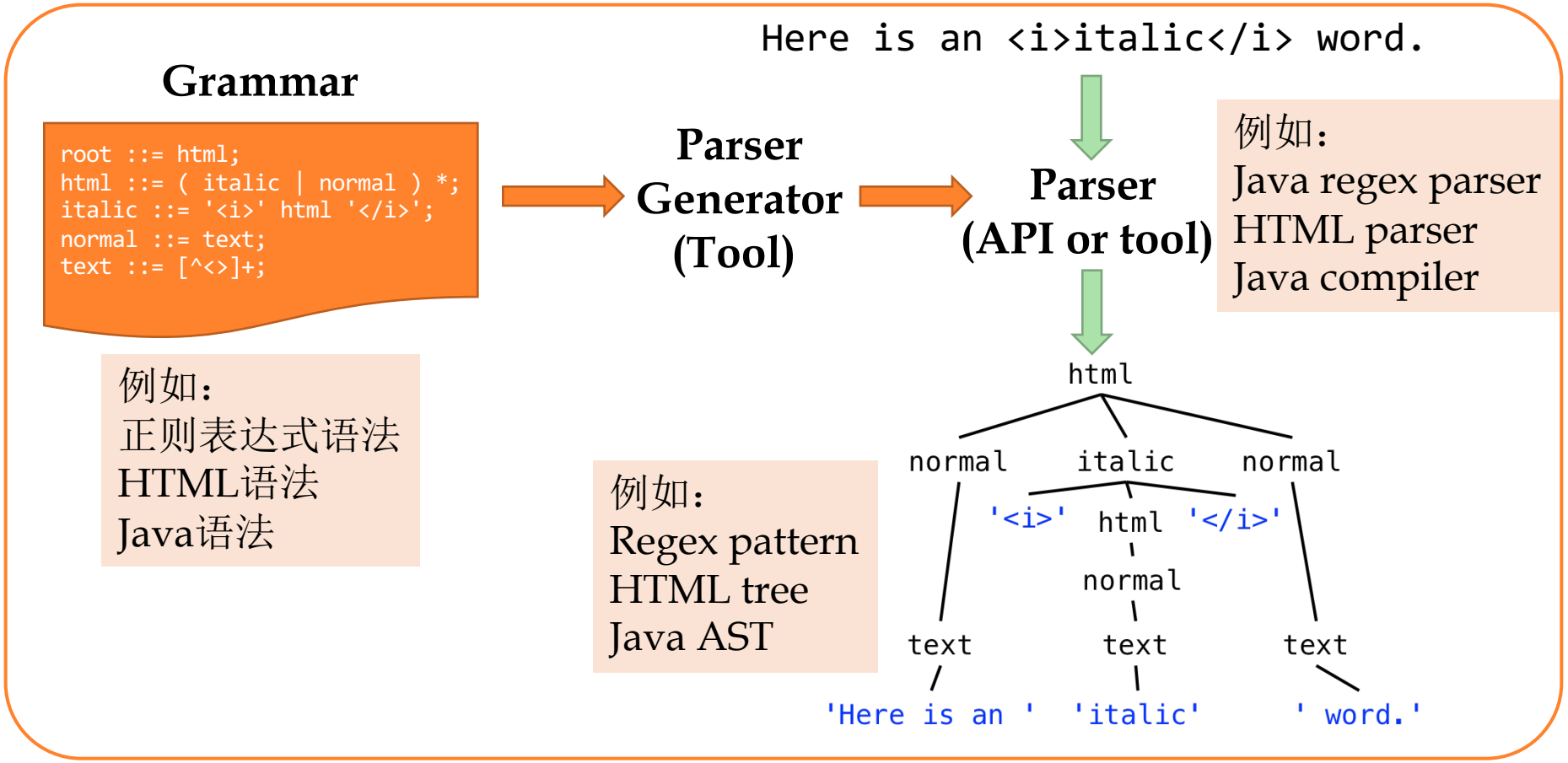


Peter Naur (1928-2016)
2005年图灵奖得主

Niklaus Wirth (1934-)
1984年图灵奖得主

Grammar, Parser Generator, and Parser

- **Grammar定义语法规则（BNF格式的文本），Parser generator根据语法规则产生一个parser，用户利用parser来解析文本，看其是否符合语法定义并对其做各种处理（例如转成parse tree）**



In your future study…

基础：形式语言
任务：设计一种语言

编译原理课程：
为某个语言设计编译器

Grammar

```
root ::= html;
html ::= ( italic | normal ) *;
italic ::= '<i>' html '</i>';
normal ::= text;
text ::= [^<>]+;
```

例如：
正则表达式语法
HTML语法
Java语法

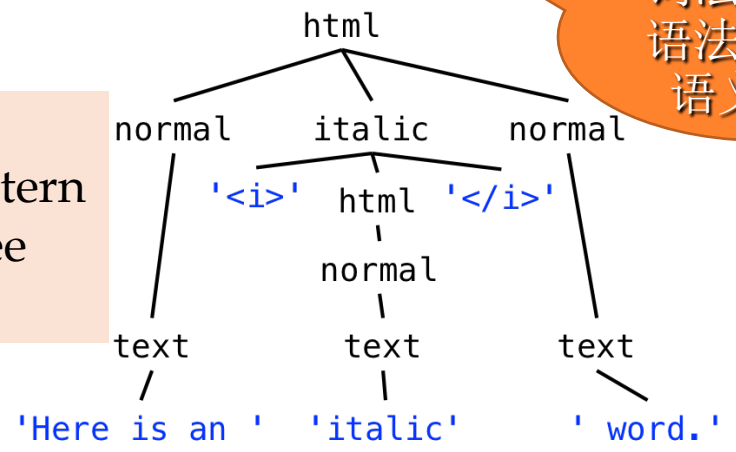
Parser Generator (Tool)

Parser (API or tool)

```
Here is an <i>italic</i> word.
```

例如：
Java regex parser
HTML parser
Java compiler

词法分析、
语法分析、
语义分析



例如：
Regex pattern
HTML tree
Java AST

(8) Using regular expressions in Java

在本课程里，只需要能够熟练掌握正则表达式regex这种“基本语法”，并熟练使用JDK提供的 regex parser进行数据处理即可

Using regular expressions in Java

- **Regexes are widely used in programming.**
- **In Java, you can use regexes for manipulating strings (see `String.split, String.matches, java.util.regex.Pattern`).**
- **They're built-in as a first-class feature of modern scripting languages like Python, Ruby, and JavaScript, and you can use them in many text editors for find and replace.**
- **Regular expressions are your friend!**

java.util.regex for Regex processing

- **The java.util.regex package primarily consists of three classes:**
  - A Pattern object is a compiled representation of a regular expression. The Pattern class provides no public constructors. To create a pattern, you must first invoke one of its public static compile methods, which will then return a Pattern object. These methods accept a regular expression as the first argument. Pattern是对regex正则表达式进行编译之后得到的结果
  - A Matcher object is the engine that interprets the pattern and performs match operations against an input string. Like the Pattern class, Matcher defines no public constructors. You obtain a Matcher object by invoking the matcher method on a Pattern object. Matcher：利用Pattern对输入字符串进行解析
  - A PatternSyntaxException object is an unchecked exception that indicates a syntax error in a regular expression pattern.

`java.util.regex` for Regex processing

Package java.util.regex

Classes for matching character sequences against patterns specified by regular expressions.

See: Description

Interface Summary

Interface	Description
MatchResult	The result of a match operation.

Class Summary

Class	Description
Matcher	An engine that performs match operations on a character sequence by interpreting a Pattern.
Pattern	A compiled representation of a regular expression.

Exception Summary

Exception	Description
PatternSyntaxException	Unchecked exception thrown to indicate a syntax error in a regular-expression pattern.

Using regular expressions in Java

- **Replace all runs of spaces with a single space:**

```
String singleSpacedString = string.replaceAll(" +", " ");
```

- **Match a URL:**

```
Pattern regex = Pattern.compile("http://([a-z]+\\.)+[a-z]+(:[0-9]+)?/");
Matcher m = regex.matcher(string);
if (m.matches()) {
    // then string is a url
}
```

- **Extract part of an HTML tag:**

```
Pattern regex = Pattern.compile("<a href=\"([^\"]*)\">");
Matcher m = regex.matcher(string);
if (m.matches()) {
    String url = m.group(1);
    // Matcher.group(n) returns the nth parenthesized part of the regex
}
```

An example

- **Write the shortest regex you can to remove single-word, lowercase-letter-only HTML tags from a string:**

```
String input = "The <b>Good</b>, the <i>Bad</i>, and the
               <strong>Ugly</strong>";
String regex = "TODO";
String output = input.replaceAll(regex, "");
```

- **If the desired output is** `"The Good, the Bad, and the Ugly"`**, what is shortest regex you can put in place of TODO?**

```
</?[a-z]+>
```

Character Classes

Construct	Description
`[abc]`	`a`, `b`, or `c` (simple class)
`[^abc]`	Any character except `a`, `b`, or `c` (negation)
`[a-zA-Z]`	`a` through `z`, or `A` through `Z`, inclusive (range)
`[a-d[m-p]]`	`a` through `d`, or `m` through `p`: `[a-dm-p]` (union)
`[a-z&&[def]]`	`d`, `e`, or `f` (intersection)
`[a-z&&[^bc]]`	`a` through `z`, except for `b` and `c`: `[ad-z]` (subtraction)
`[a-z&&[^m-p]]`	`a` through `z`, and not `m` through `p`: `[a-lq-z]` (subtraction)

Metacharacters: `<([{\^-=$!|]})?*+.>`

Two ways to force a metacharacter to be treated as an ordinary character:

- Precede the metacharacter with a backslash `\`
- Enclose it within `\Q` (which starts the quote) and `\E` (which ends it).

Predefined Character Classes

Construct	Description
`.`	Any character (may or may not match line terminators)
`\d`	A digit: `[0-9]`
`\D`	A non-digit: `[^0-9]`
`\s`	A whitespace character: `[ \t\n\x0B\f\r]`
`\S`	A non-whitespace character: `[^\s]`
`\w`	A word character: `[a-zA-Z_0-9]`
`\W`	A non-word character: `[^\w]`

Quantifiers

Greedy	Reluctant	Possessive	Meaning
`X?`	`X??`	`X?+`	X, once or not at all
`X*`	`X*?`	`X*+`	X, zero or more times
`X+`	`X+?`	`X++`	X, one or more times
`X{n}`	`X{n}?`	`X{n}+`	X, exactly n times
`X{n,}`	`X{n,}?`	`X{n,}+`	X, at least n times
`X{n,m}`	`X{n,m}?`	`X{n,m}+`	X, at least n but not more than m times

Boundary Matchers

Boundary Construct	Description
^	The beginning of a line
$	The end of a line
\b	A word boundary
\B	A non-word boundary
\A	The beginning of the input
\G	The end of the previous match
\Z	The end of the input but for the final terminator, if any
\z	The end of the input

Pattern method equivalents in `java.lang.String`

- `public boolean matches(String regex)`: **Tells whether or not this string matches the given regular expression.**
  `Pattern.matches(regex, str).`

- `public String[] split(String regex, int limit)`: **Splits this string around matches of the given regular expression.**
  `Pattern.compile(regex).split(str, n)`

- `public String[] split(String regex)`: **Splits this string around matches of the given regular expression.**

- `public String replace(CharSequence target,CharSequence replacement)`

Learn by yourself

- https://docs.oracle.com/javase/tutorial/essential/regex/index.html
- https://en.wikipedia.org/wiki/Regular_expression

Summary

- **Safe from bugs**
  - Grammars and regular expressions are declarative specifications for strings and streams, which can be used directly by libraries and tools.
  - These specifications are often simpler, more direct, and less likely to be buggy than parsing code written by hand.
- **Easy to understand**
  - A grammar captures the shape of a sequence in a form that is easier to understand than hand-written parsing code.
  - Regular expressions, alas, are often not easy to understand, because they are a one-line reduced form of what might have been a more understandable regular grammar.
- **Ready for change**
  - A grammar can be easily edited, but regular expressions, unfortunately, are much harder to change, because a complex regular expression is cryptic and hard to understand.

Summary

Summary

- **Software Maintenance and Evolution**
- **Metrics of Maintainability**
- **Modular Design and Modularity Principles**
- **OO Design Principles: SOLID**

- **Grammer-based Construction**
- Machine-processed textual languages are ubiquitous in computer science.
- Grammars are the most popular formalism for describing such languages
- Regular expressions are an important subclass of grammars that can be expressed without recursion.

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

The end

June 7, 2021