SOFTWARE TESTING

苏临之

sulinzhi029@nwu.edu.cn

Cyclomatic Complexity

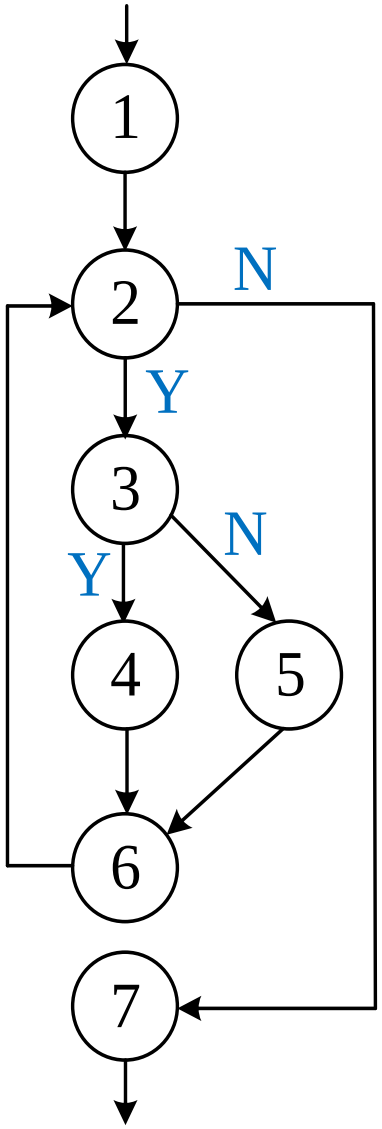
- Three methods for calculating the cyclomatic complexity:
  - Two Formulae

$$V = E - N + 2P \qquad V = E_0 - N$$

  - Number of Plane Region
  - Number of Binary Predicate Nodes plus 1

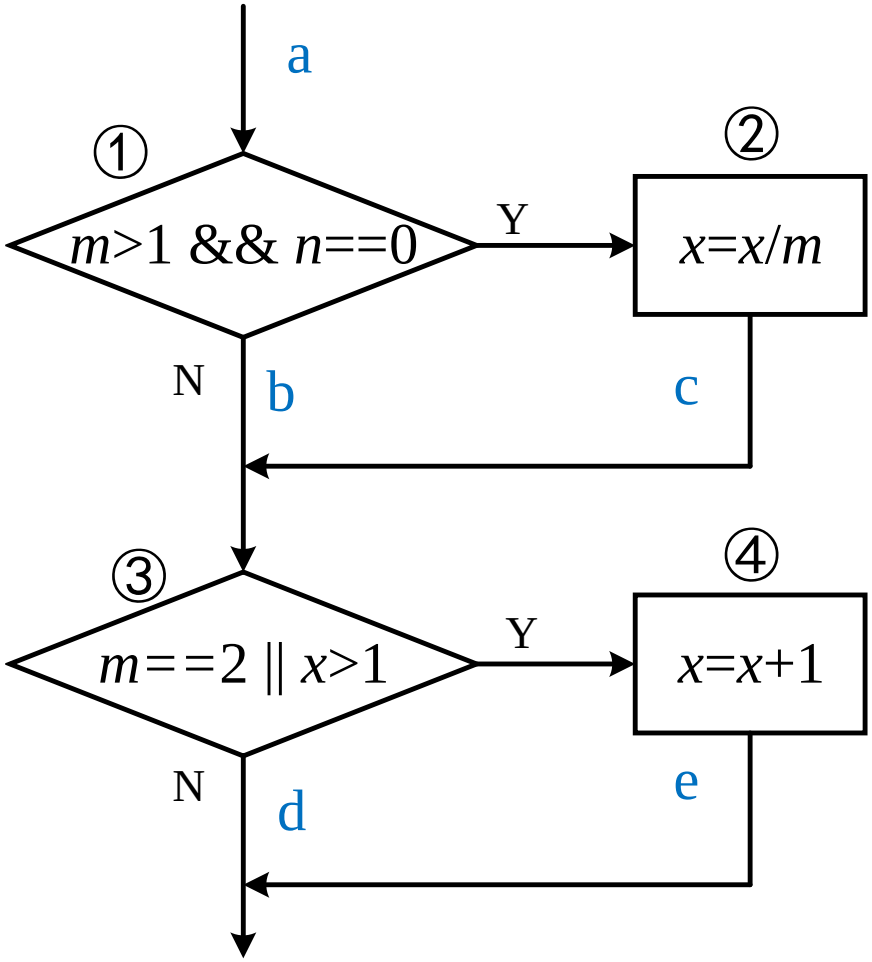
Basic Path Set

- In a loop, the predicate node should start from N, shipping the loop; then enter the loop by choosing Y.
- In an IF-ELSE structure, the paths can be counted in parallel.



Basic Path Set

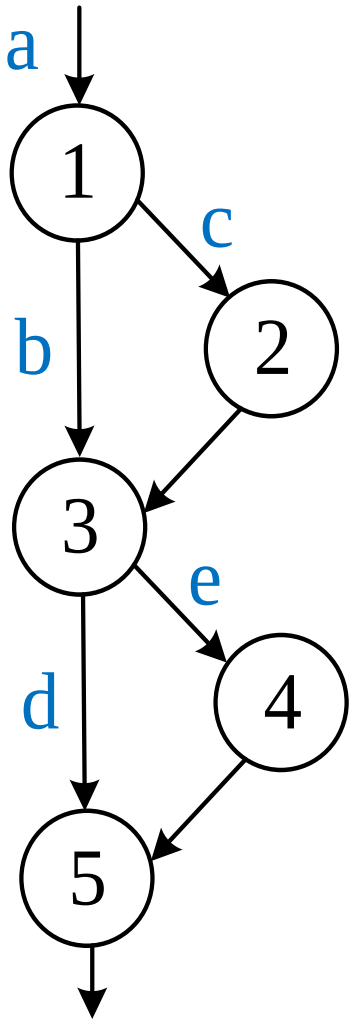
- What about this one?



Basic Path Set

- What about this one?
- Here *V*=3, so the basic paths are:

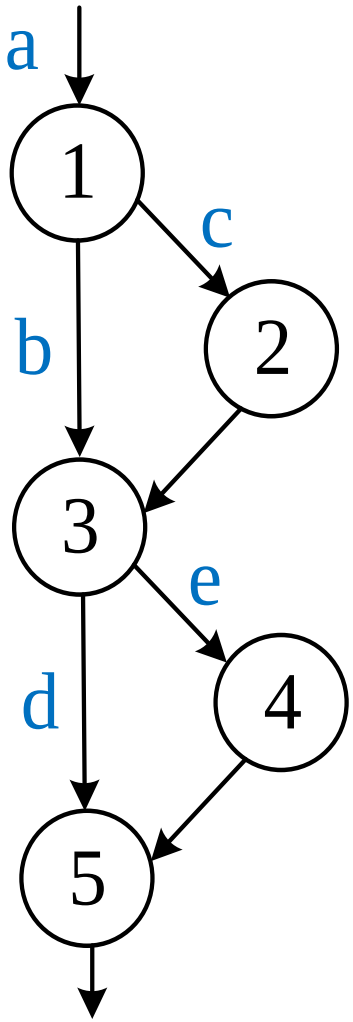
P1: 1-3-5
P2: 1-2-3-5
P3: 1-3-4-5



Basic Path Set

- Why not these two?

P1: 1-3-5
P2: 1-2-3-4-5

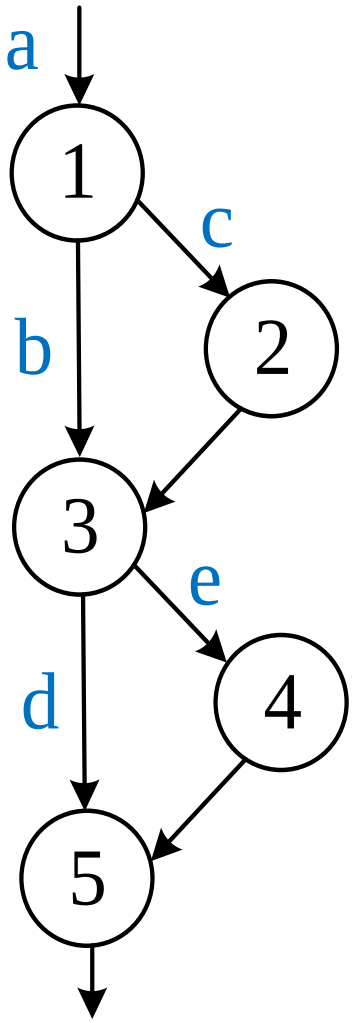


Basic Path Set

- Why not these two?

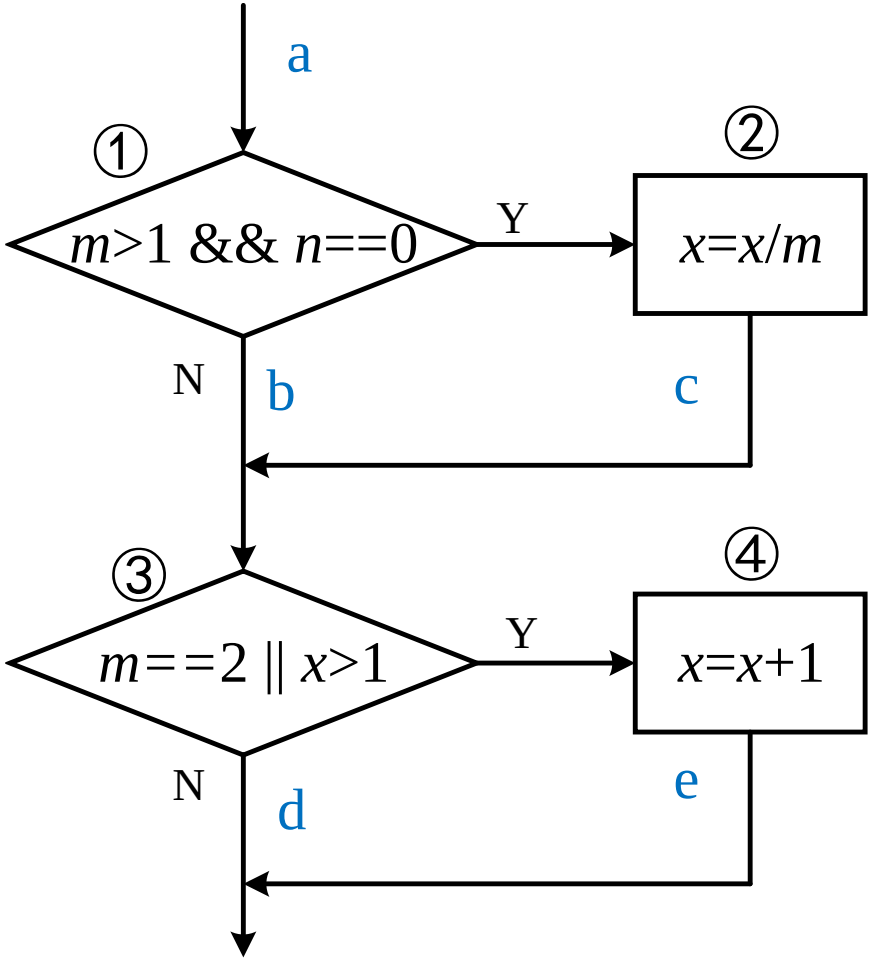
P1: 1-3-5
P2: 1-2-3-4-5

- For the IF-ELSE structures in series, P1 serves as a reference, and each of the other paths should serve as one with only one different factor (node or nodes).

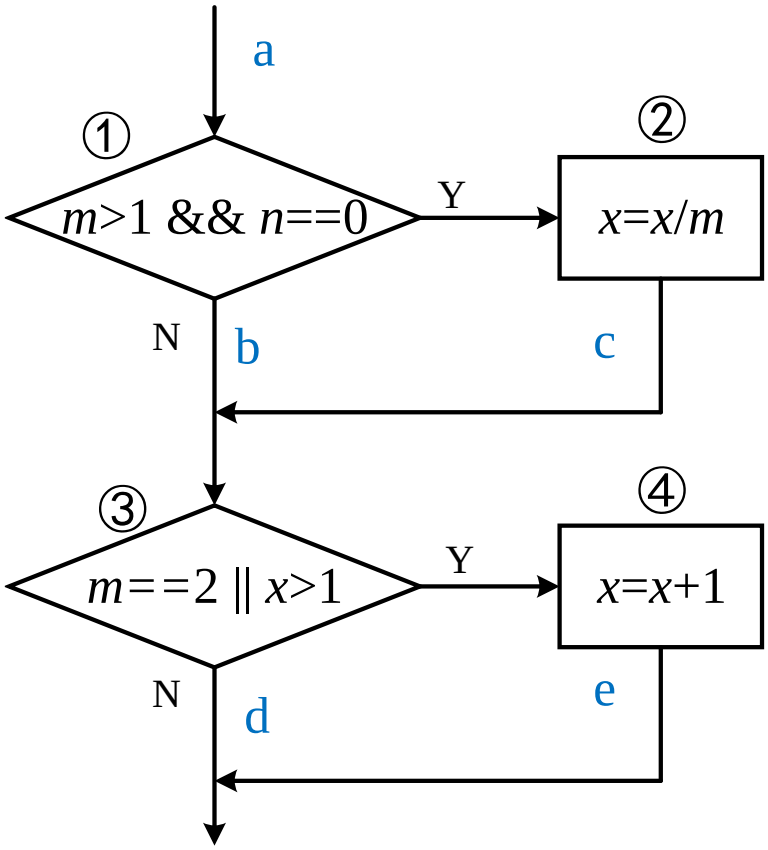


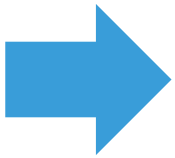
Path Coverage for Conditions

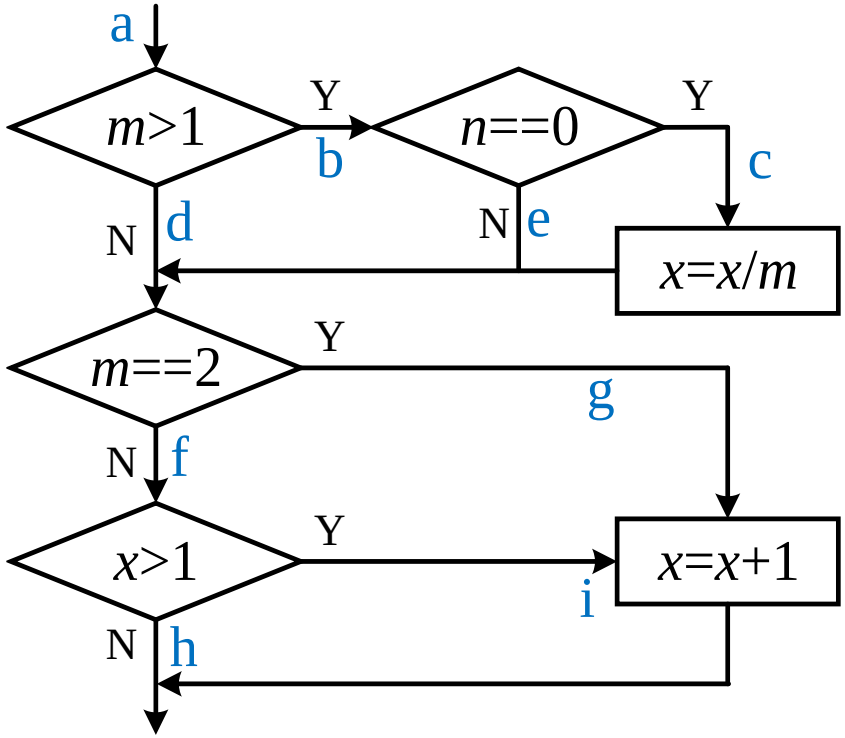
- Basic path coverage is not able to detect some minor condition error. In the IF statement ①, two conditions are involved, but in the flow graph, it is represented as only one node. If a complete path coverage is undertaken, it should be modified first.



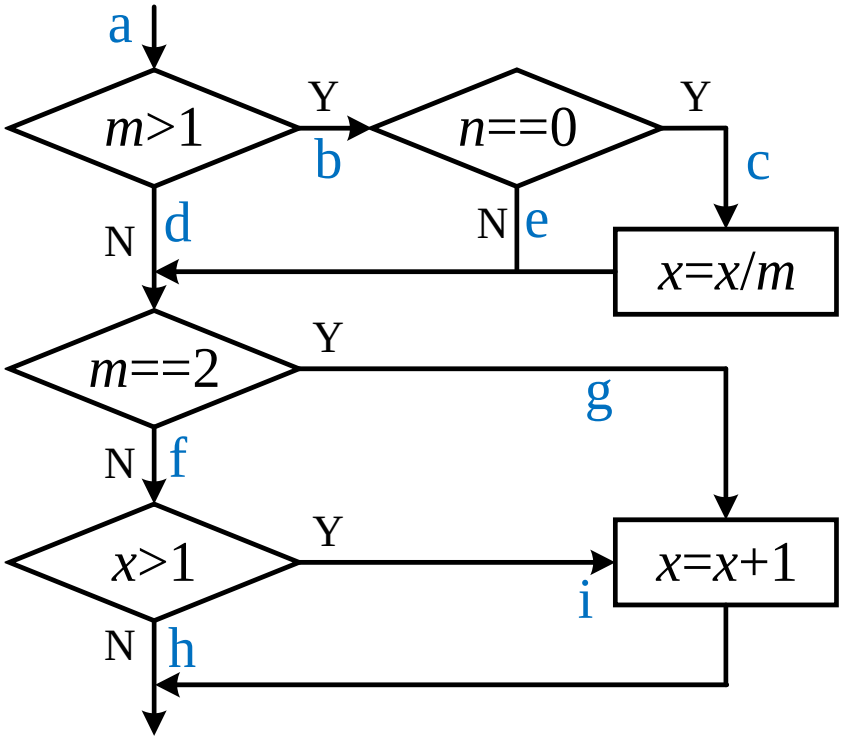
Path Coverage for Conditions

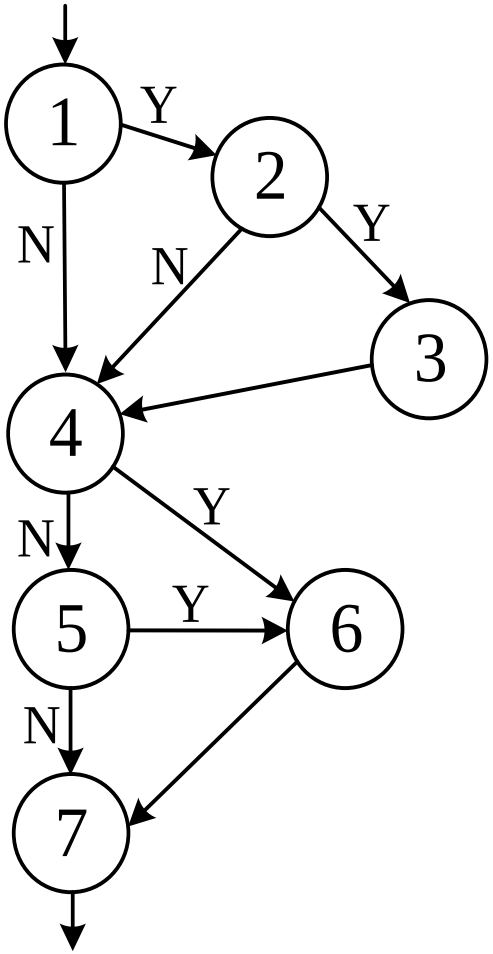






Path Coverage for Conditions

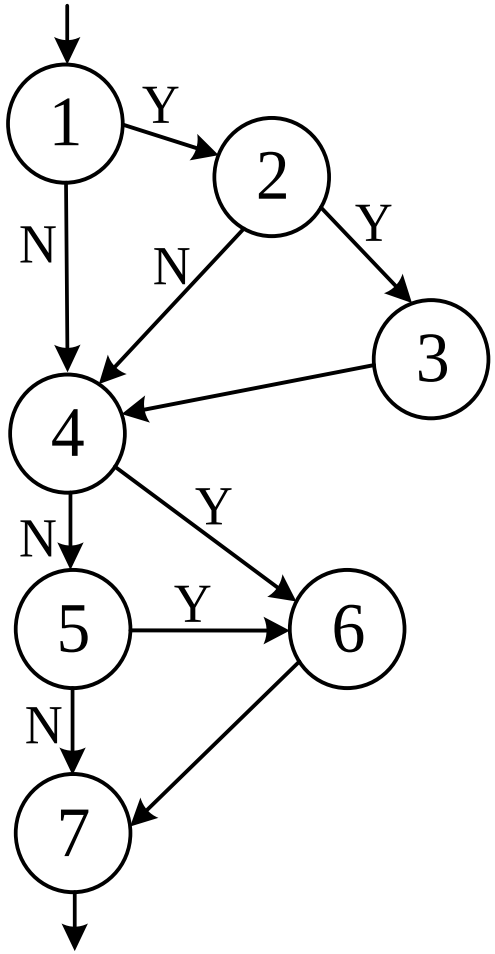




Path Coverage for Conditions

- Then we find a set of basic paths for this modified one.
- Here *V*=5, so the basic paths are:

P1: 1-4-5-7
P2: 1-2-4-5-7
P3: 1-2-3-4-5-7
P4: 1-4-6-7
P5: 1-4-5-6-7

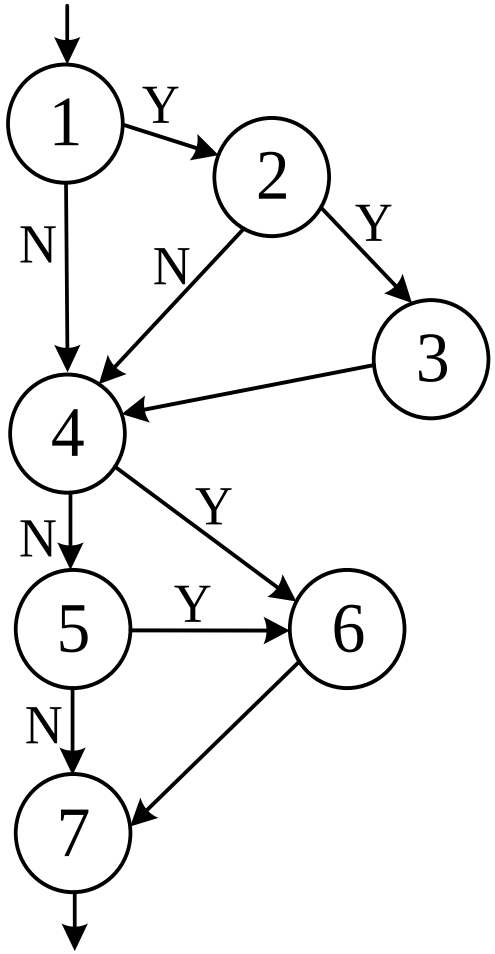


Path Coverage for Conditions

- Then we find a set of basic paths for this modified one.
- Here *V*=5, so the basic paths are:

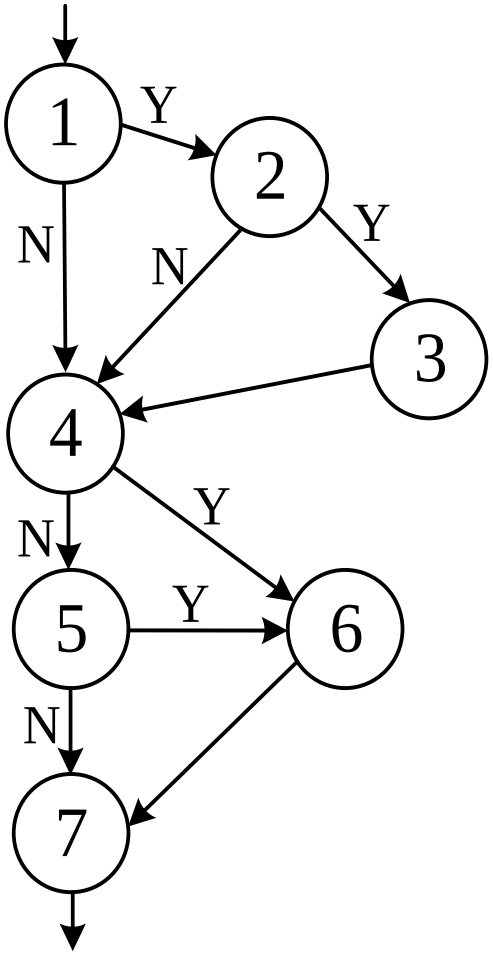
P1: 1-4-5-7
P2: 1-2-4-5-7
P3: 1-2-3-4-5-7
P4: 1-4-6-7
P5: 1-4-5-6-7

- The paths with 1-4 are impossible with no proper test cases, so they should be replaced. (Infeasible paths)



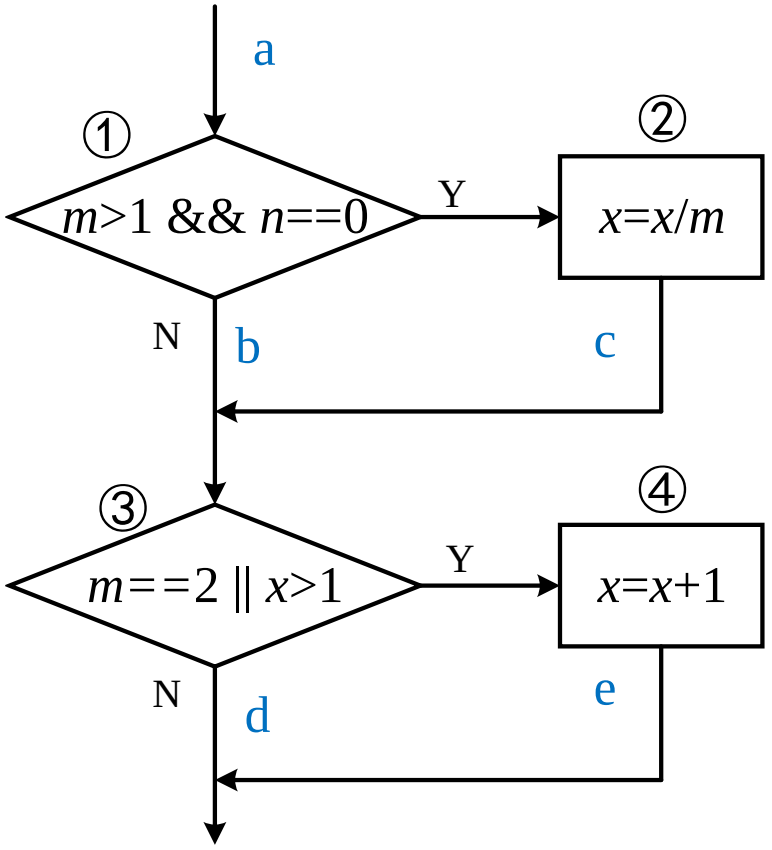
Path Coverage for Conditions

P1: 1-2-4-5-6-7
P2: 1-2-4-5-7
P3: 1-2-3-4-5-7
P4: 1-2-4-6-7
P5: 1-2-3-5-6-7

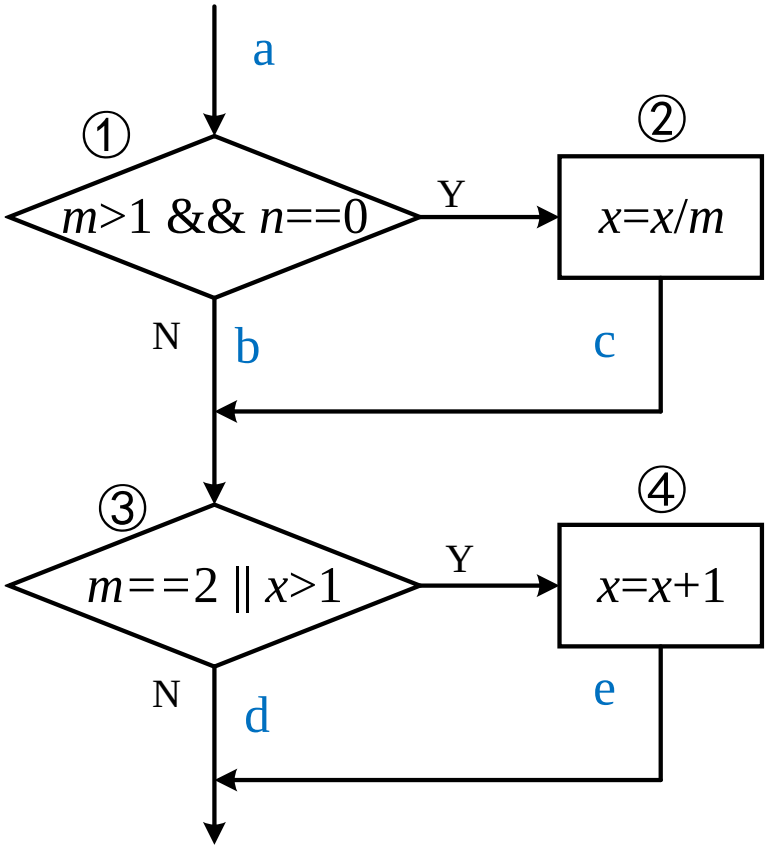


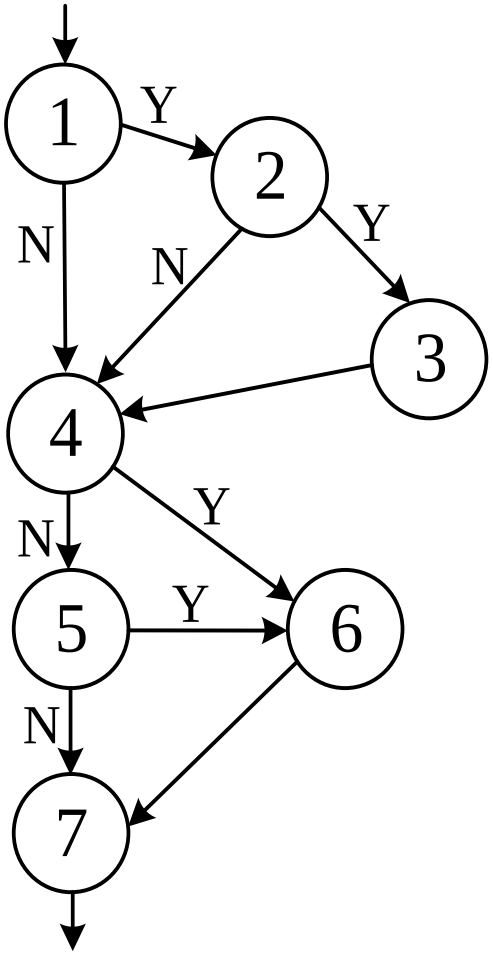
Complete Path Coverage

- Sometimes, a complete path coverage is needed. In this case, all the combinations of paths should be considered.



Complete Path Coverage





Complete Path Coverage

- Two IF-ELSE structures are in series, each containing three sub-paths. So we have nine paths here.

P1: 1-4-5-7

P2: 1-2-4-5-7

P3: 1-2-3-4-5-7

P4: 1-4-6-7

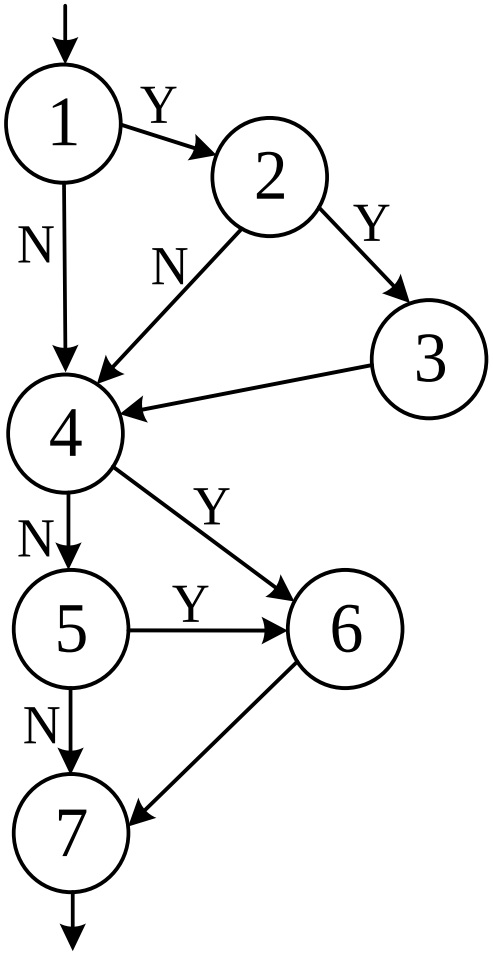
P5: 1-2-4-6-7

P6: 1-2-3-4-6-7

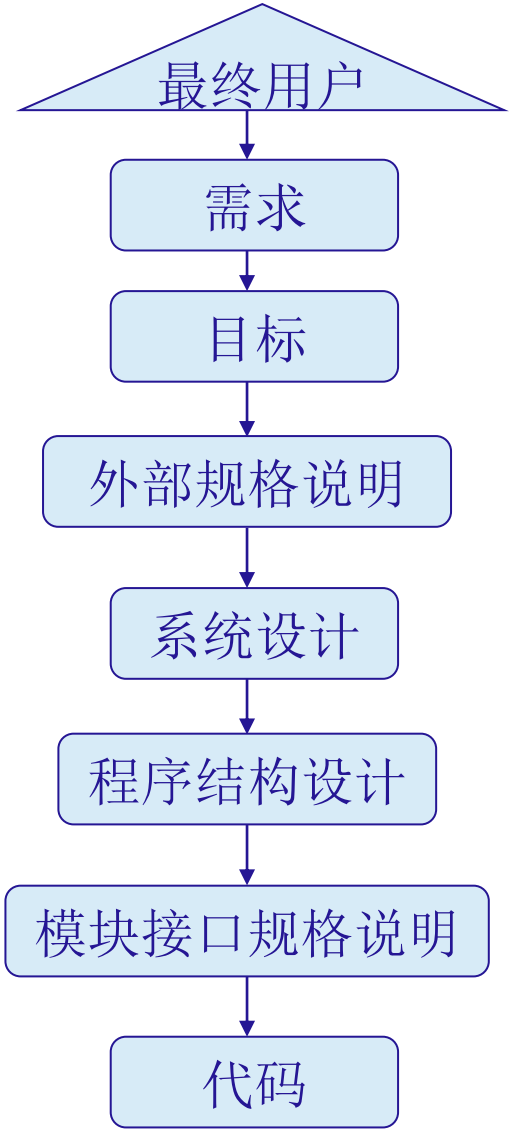
P7: 1-4-5-6-7

P8: 1-2-4-5-6-7

P9: 1-2-3-4-5-6-7

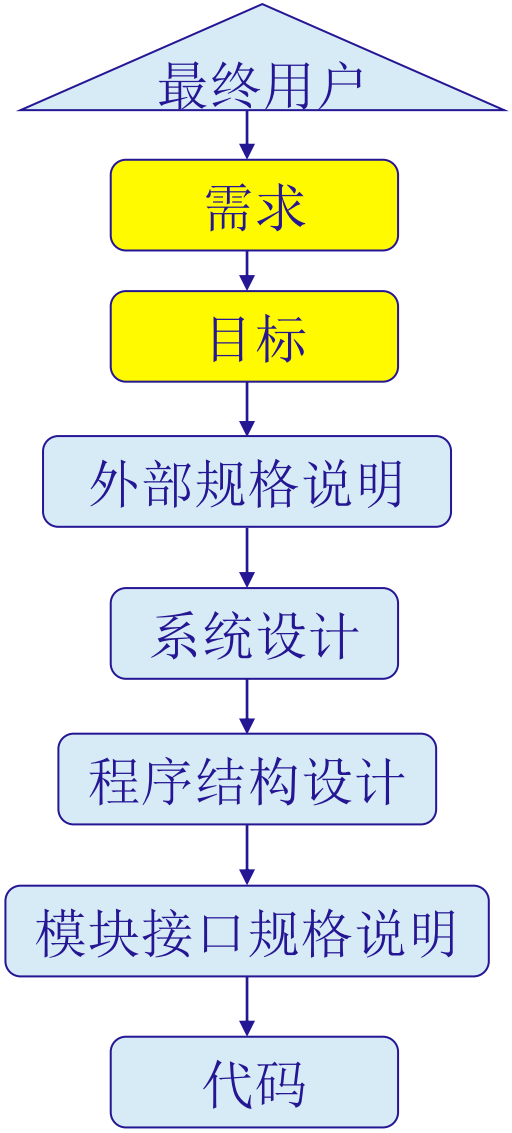


Development of Software



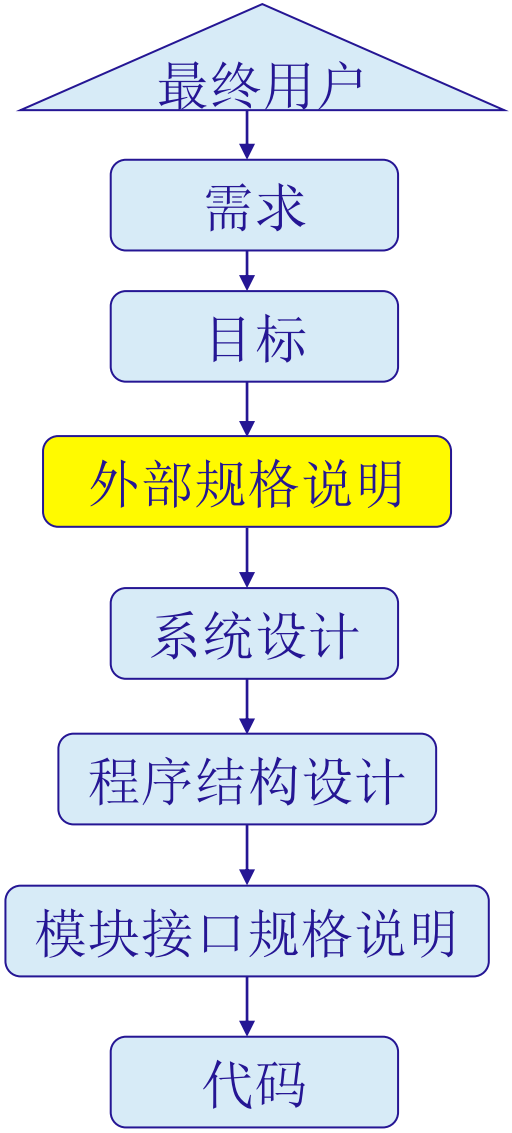
Development of Software

- These two consists the materials directly from the users. They are of great value although not so precise.
- In them, some basic goals and detailed goals (ability, capacity, usability and so on) can be obtained.



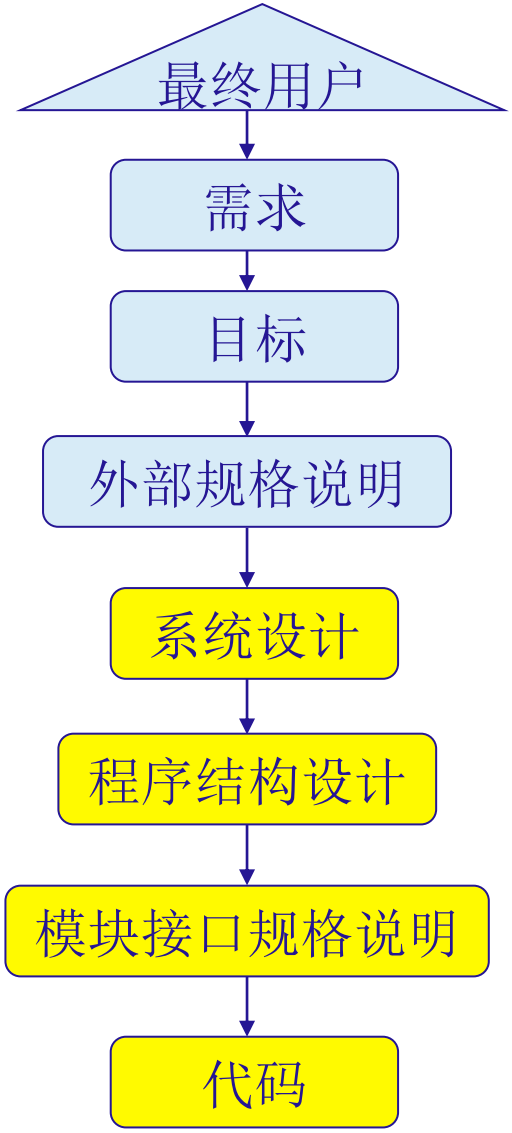
Development of Software

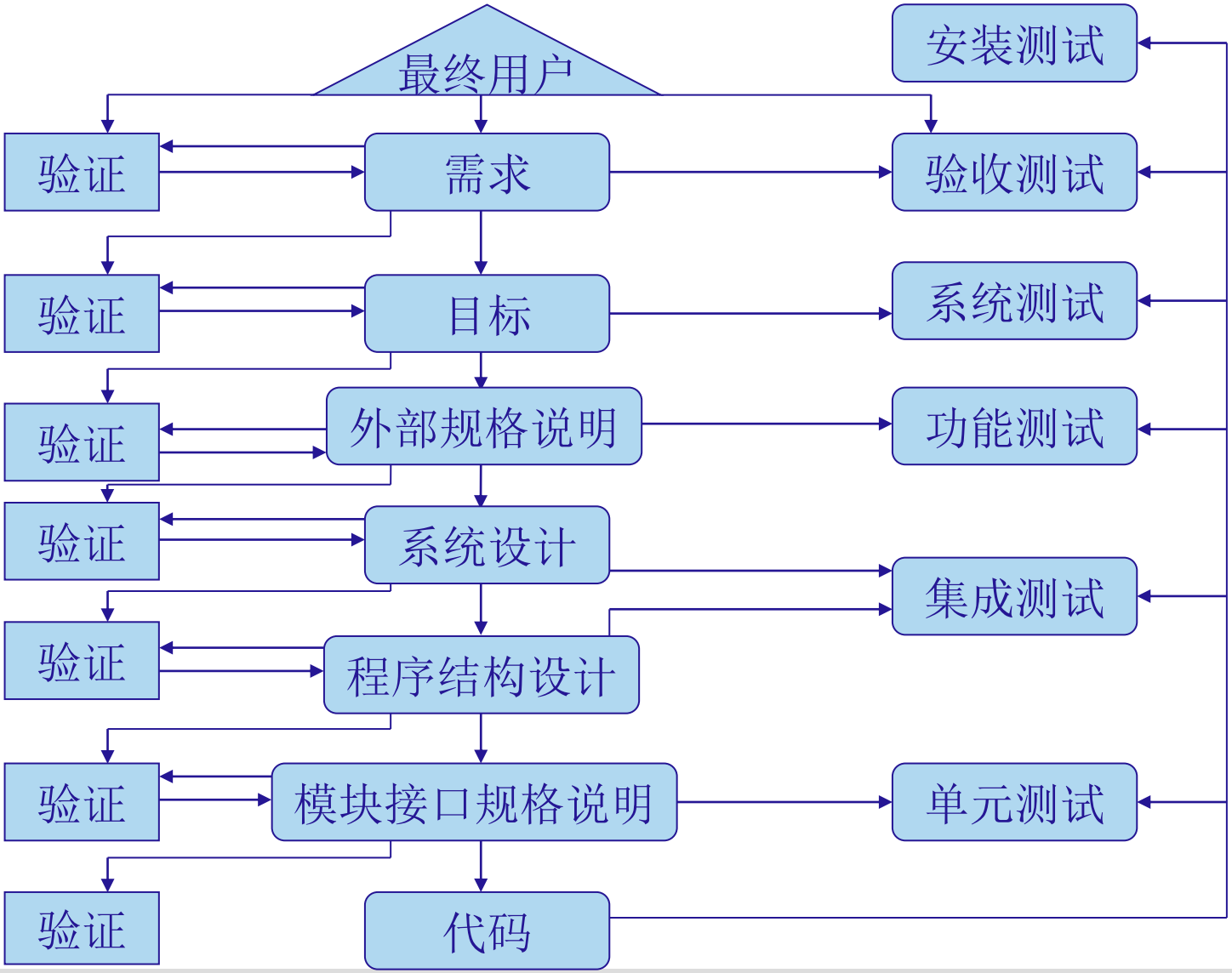
- It is the detailed description summarized from the users, providing the input from the following steps for design with clear standards and criteria.



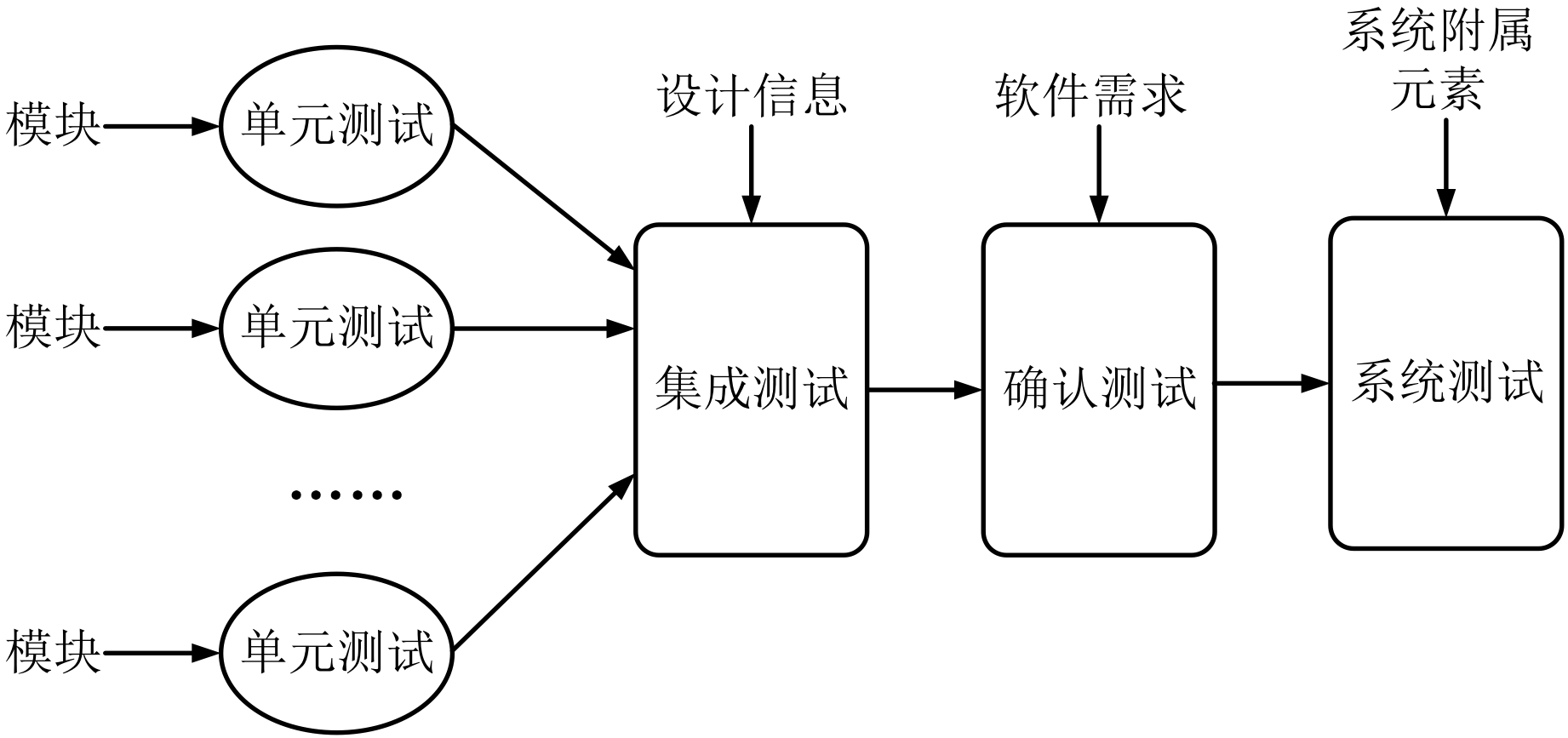
Development of Software

- From the system design to code, the developers make it clearer and clearer how to build up a program.

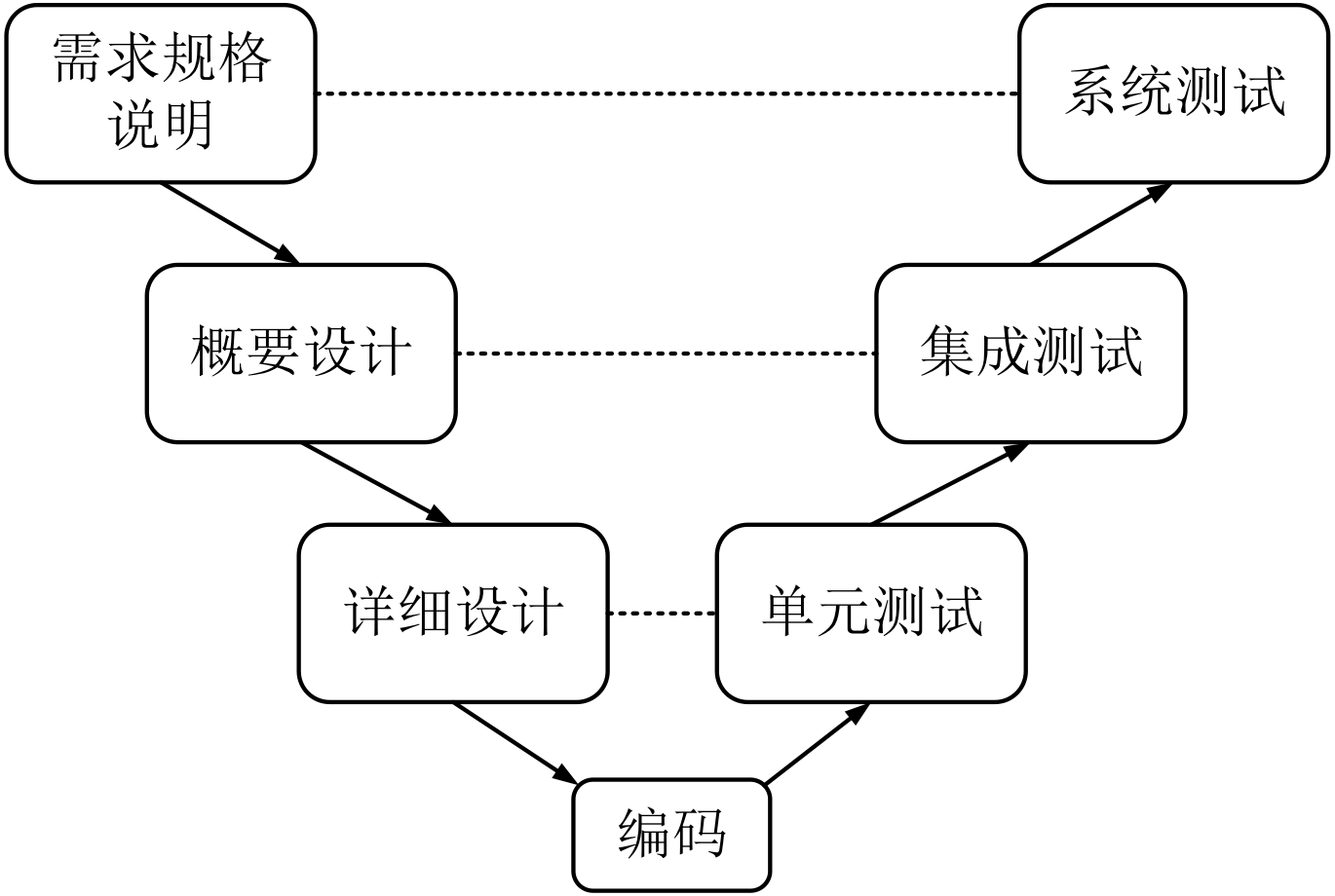



最终用户
安装测试
验证
需求
验收测试
验证
目标
系统测试
验证
外部规格说明
功能测试
验证
系统设计
集成测试
验证
程序结构设计
验证
模块接口规格说明
单元测试
验证
代码

Basic Procedure



Basic Model



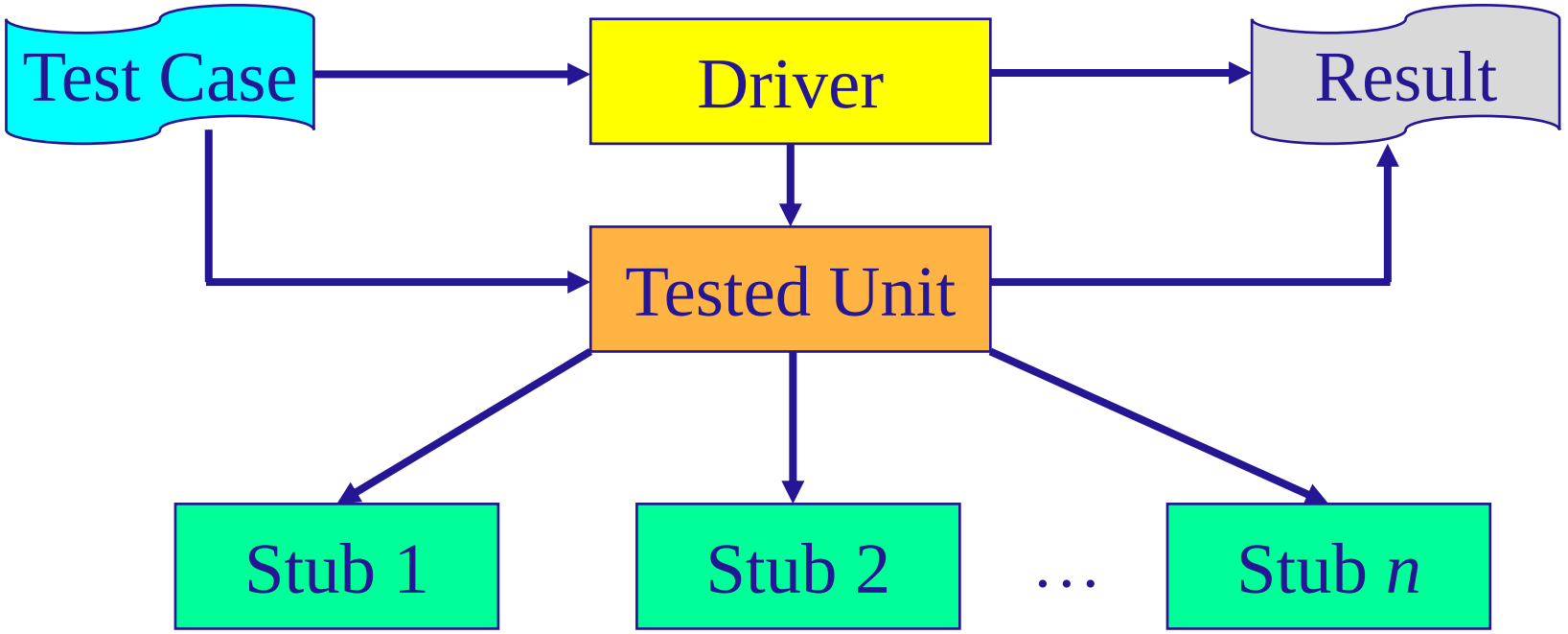
Unit Testing

- Unit testing (单元测试) is also called the module testing, which alleviates the difficulty in the testing process. One can also test several units in parallel.
- The unit testing is generally a white-box testing, which means that it is essential to know the basic code in the module, including the input and output as well as its function.

Problems Concerning Unit Testing

- 模块接口：检查进出单元的数据流是否正确
- 局部数据结构：测试内部数据是否完整？例如不正确的类型说明，错误的初始化等。
- 路径测试：发现由于不正确的判定或不正常的控制流而产生的错误，例如不正确的逻辑操作或优先级，不适当地修改循环变量。
- 边界条件：测试边界处程序是否正确工作？
- 出错处理：测试出错处理措施是否有效？例如提供的错误信息不足，难以找到错误原因等。

Basic Model

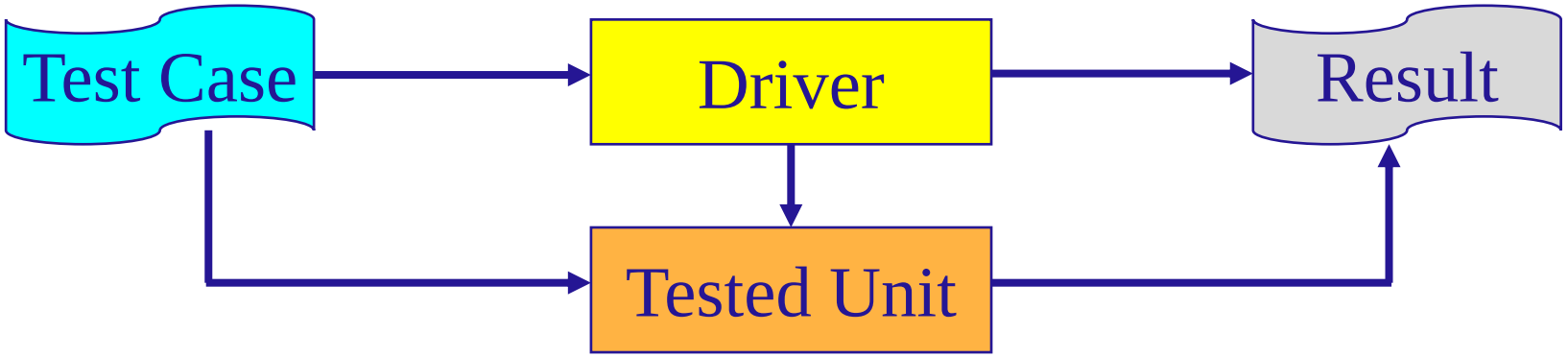


Driver Module

- The driver module (驱动模块) acts as the higher part of the unit. It sends data to the unit and then shows the corresponding result after operation. It can simulate, for example, the performance of a user or call a function.

Driver Module

- The driver module (驱动模块) acts as the higher part of the unit. It sends data to the unit and then shows the corresponding result after operation. It can simulate, for example, the performance of a user or call a function.

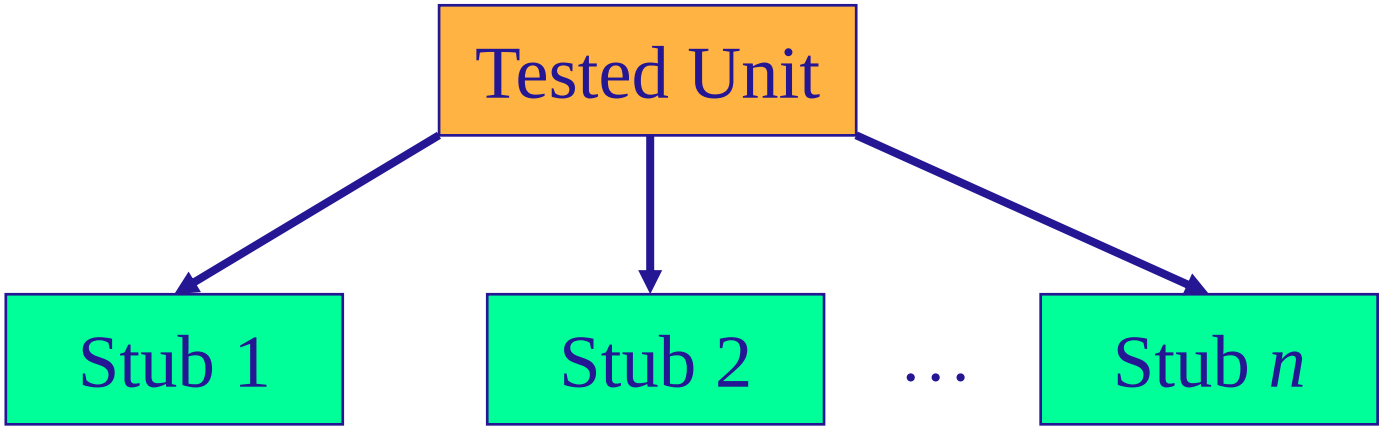


Stub Module

- The stub module (桩模块) acts as the lower sub-routine of the unit, and it actually does not comprise the software itself. By simulating the module called by the tested unit, the stub module itself does not operate but just return a static value.

Stub Module

- The stub module (桩模块) acts as the lower sub-routine of the unit, and it actually does not comprise the software itself. By simulating the module called by the tested unit, the stub module itself does not operate but just return a static value.



Example 1

- 例如要测试右侧程序，必须给它一个运行的环境，即编写一个上级程序来调用f这个函数，并在给定输入的情况下打印出f的输出结果。这个运行的环境模块就是驱动模块。

```
int f(int x)
{
    int y;
    if (x%2!=0)
       y=3*x+1;
    else
        y=sgn(x)*x/2;
}
```

Example 1

- 但在测试时，sgn这个函数也需要有返回值，否则就无法测试。因此需要编写一个同名子程序来模拟其调用过程，它只用来返回静态值而不执行sgn本身。这就是桩模块。

```
int f(int x)
{
    int y;
    if (x%2!=0)
       y=3*x+1;
    else
        y=sgn(x)*x/2;
}
```

Example 2

- 这个计算阶乘求和的程序单元，要测试的时候需要让其有一个含有确定测试用例作为实参的运行环境。

```
function y=fa_sum(n)
y=0;
for i=1:n
    x=fa(i);
    y=y+x;
end
```

Example 2

- 这个计算阶乘求和的程序单元，要测试的时候需要让其有一个含有确定测试用例作为实参的运行环境。
- 如输入测试用例是5，则编写以下驱动模块：

```
function y=fa_sum(n)
y=0;
for i=1:n
    x=fa(i);
    y=y+x;
end
```

```
clc;clear all;close all;
tic;
n=5;
y=fa_sum(n);
sprintf('%d', y);
toc;
```

Driver

Example 2

- 但fa这个函数也是一个模块，测试时候需要涉及。因此创建一个同名桩模块来模拟其运行，只返回静态值而没有其他的操作。

```
function y=fa_sum(n)
y=0;
for i=1:n
    x=fa(i);
    y=y+x;
end
```

Example 2

- 但fa这个函数也是一个模块，测试时候需要涉及。因此创建一个同名桩模块来模拟其运行，只返回静态值而没有其他的操作。

```
function y=fa_sum(n)
y=0;
for i=1:n
    x=fa(i);
    y=y+x;
end
```

```
function x=fa(i)
if i==1
    x=1;
elseif i==2
    x=2;
……
end
```

Stub

Testing Methods

- Non-incremental Testing (非增量测试): Also called the Big-bang integration. The units are tested individually and then integrated together for the integration testing.
- Incremental Testing (增量测试): The tested modules are arranged into the structure of the program in advance. In this way the module testing also includes the integration testing, which will not serve as an individual part.
  - Top-Down Testing (自顶向下的测试)
  - Bottom-Up Testing (自底向上的测试)

Comparison

Incremental Testing	Non-incremental Testing
工作量小：使用前面测试过的模块来取代非增量测试中所需要的驱动模块或桩模块	工作量较大：要设计驱动模块和桩模块
可以较早发现模块中与不匹配接口、不正确假设等编程错误	到了测试过程的最后阶段，模块之间才能“互相看到”
容易进行调试，新出现的错误往往与最近添加的模块有关	直到整个程序组装之后，模块之间接口相关的错误才会浮现，难以定位

Comparison

Incremental Testing	Non-incremental Testing
测试可以进行地更彻底，每个模块经受了更多的检验	使用驱动模块和桩模块而非实际模块，对被测试模块的测试只影响自身
在测试上花费的时间多，设计驱动模块和桩模块所用时间少	测试时间少，但设计驱动模块和桩模块需要大量时间
并行性差	可以同时并行测试很多模块

Incremental Testing

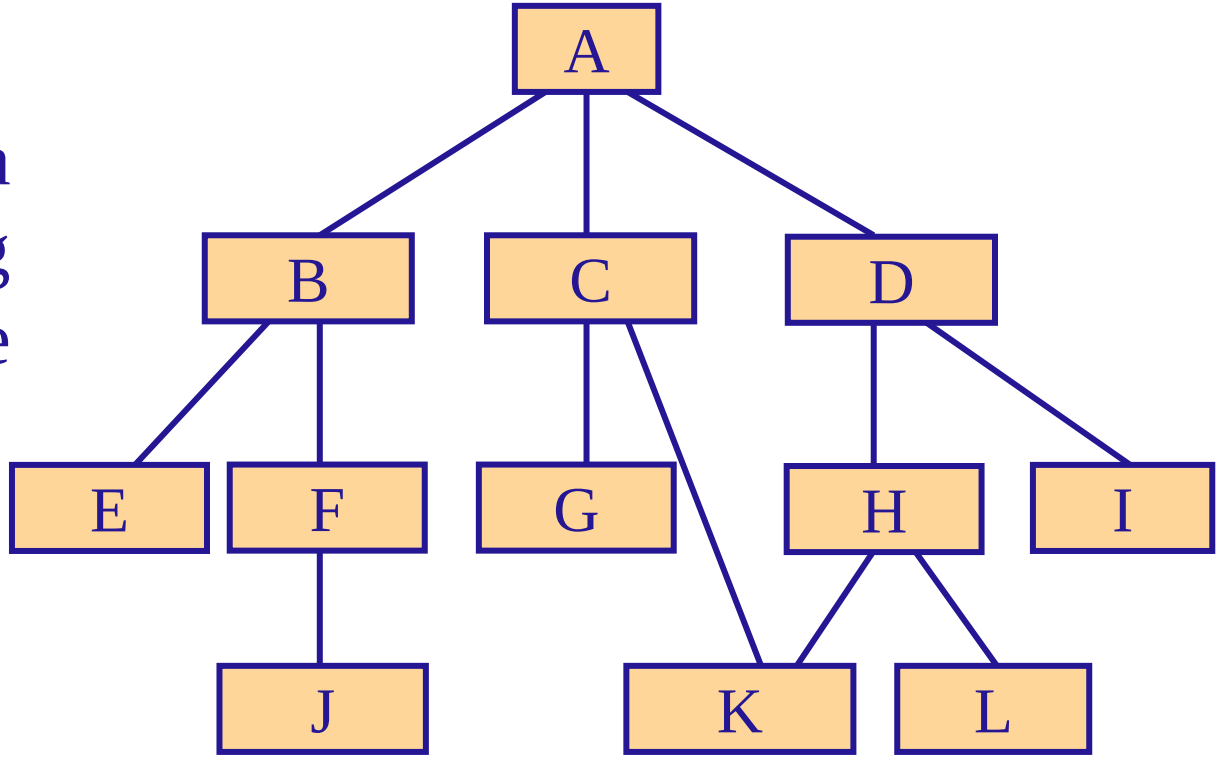
Incremental Testing

Top-Down Testing

- The top-down testing starts from the root node of the entire program tree. In the process, at least one module that calls the tested unit has been tested.
- To test the upper unit, the stub modules are established.

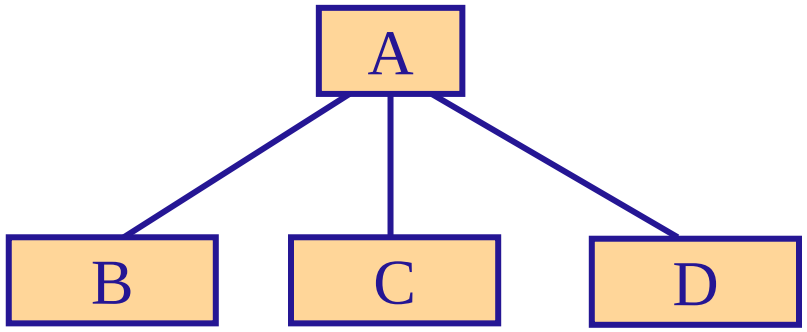
Top-Down Testing

- How to design a top-down testing strategy for the issue?



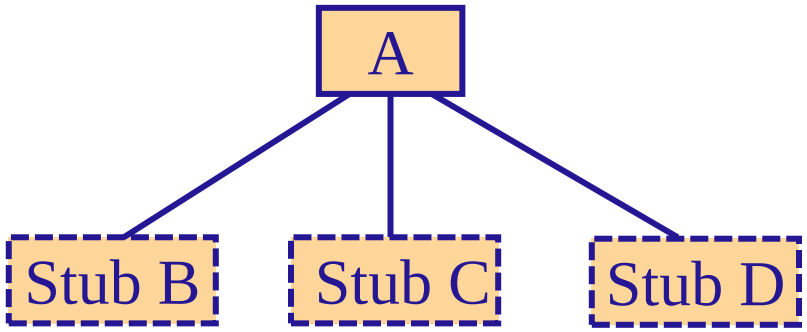
Top-Down Testing

- 测试从作为主程序的根结点A开始。BCD作为其下属子程序，三者的输出数据需作为输入数据提供给A。此时单对A测试，BCD需要虚化为桩模块，在其需要调用的时候只提供向A输入的数据而不执行BCD子程序本身。



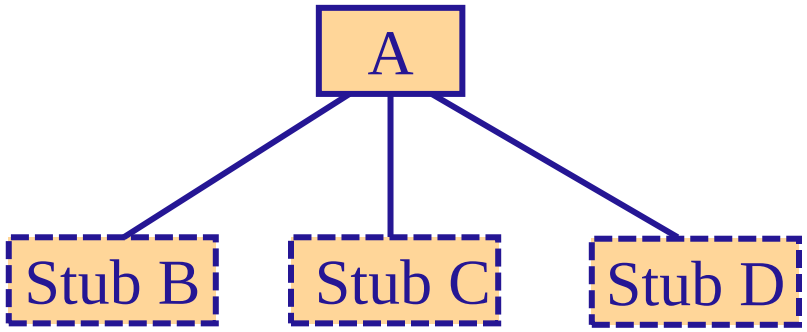
Top-Down Testing

- 测试从作为主程序的根结点A开始。BCD作为其下属子程序，三者的输出数据需作为输入数据提供给A。此时单对A测试，BCD需要虚化为桩模块，在其需要调用的时候只提供向A输入的数据而不执行BCD子程序本身。



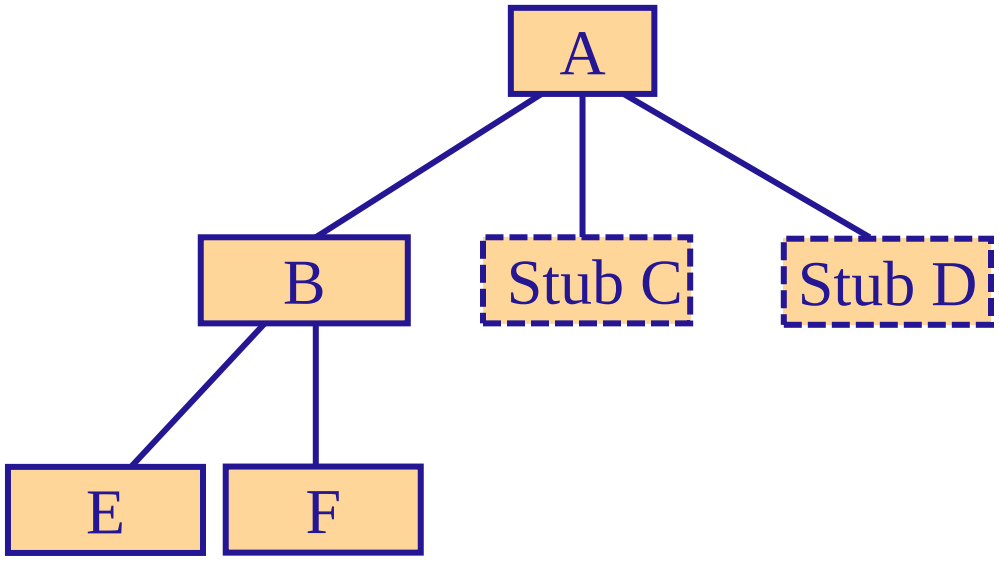
Top-Down Testing

- A的测试结束后，就进行下一级BCD的测试，这里不妨先测试B。B的直接下属模块是EF，因此把原桩模块的Stub B恢复为B，然后标注出EF。



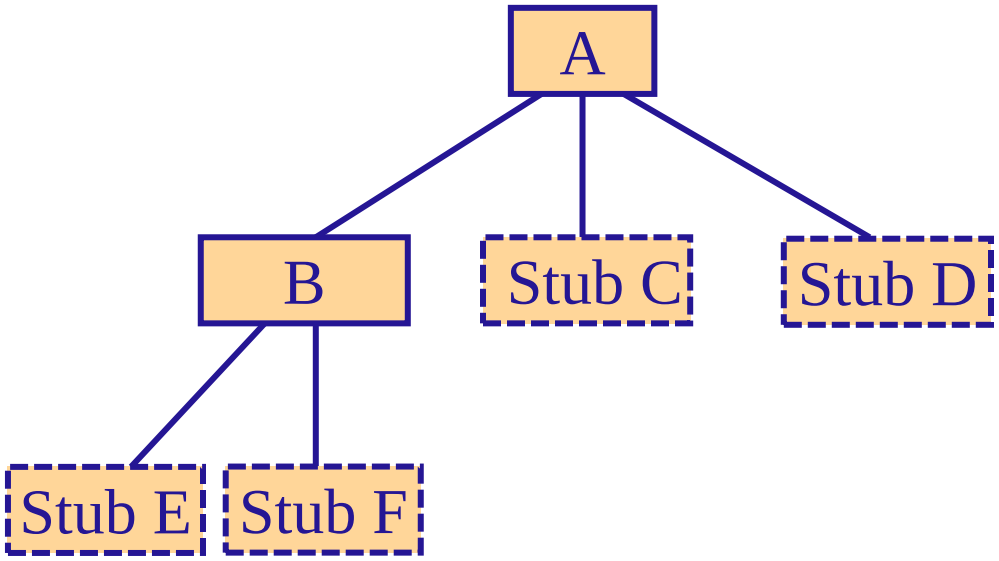
Top-Down Testing

- A的测试结束后，就进行下一级BCD的测试，这里不妨先测试B。B的直接下属模块是EF，因此把原桩模块的Stub B恢复为B，然后标注出EF。



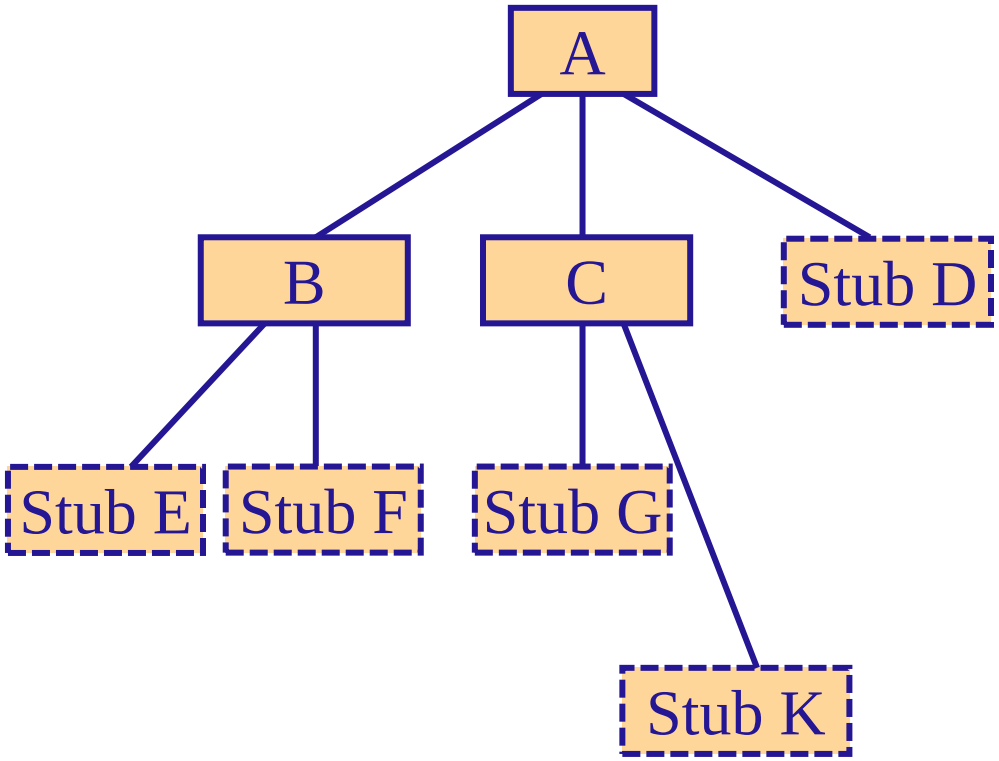
Top-Down Testing

- 同理，B的直接下属模块是EF两个模块，它们的输出数据会给B提供输入数据。若单对B测试，EF需要虚化为桩模块，在其需要调用的时候只提供向B输入的数据而不执行EF子程序本身。



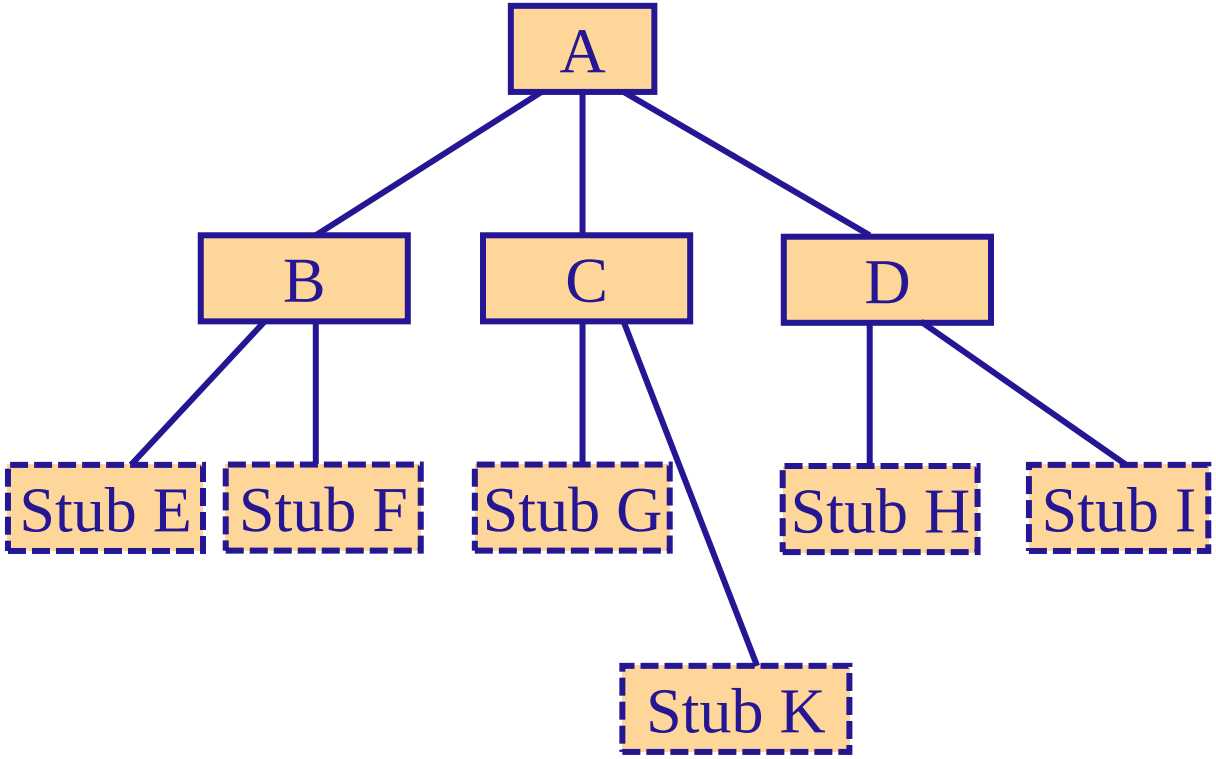
Top-Down Testing

- 依照同样的模式可以对C和D进行测试，即先寻找直接下属模块，再恢复自身在上一次测试中的桩模块为实际模块，然后把其直接下属模块作为本级的桩模块。



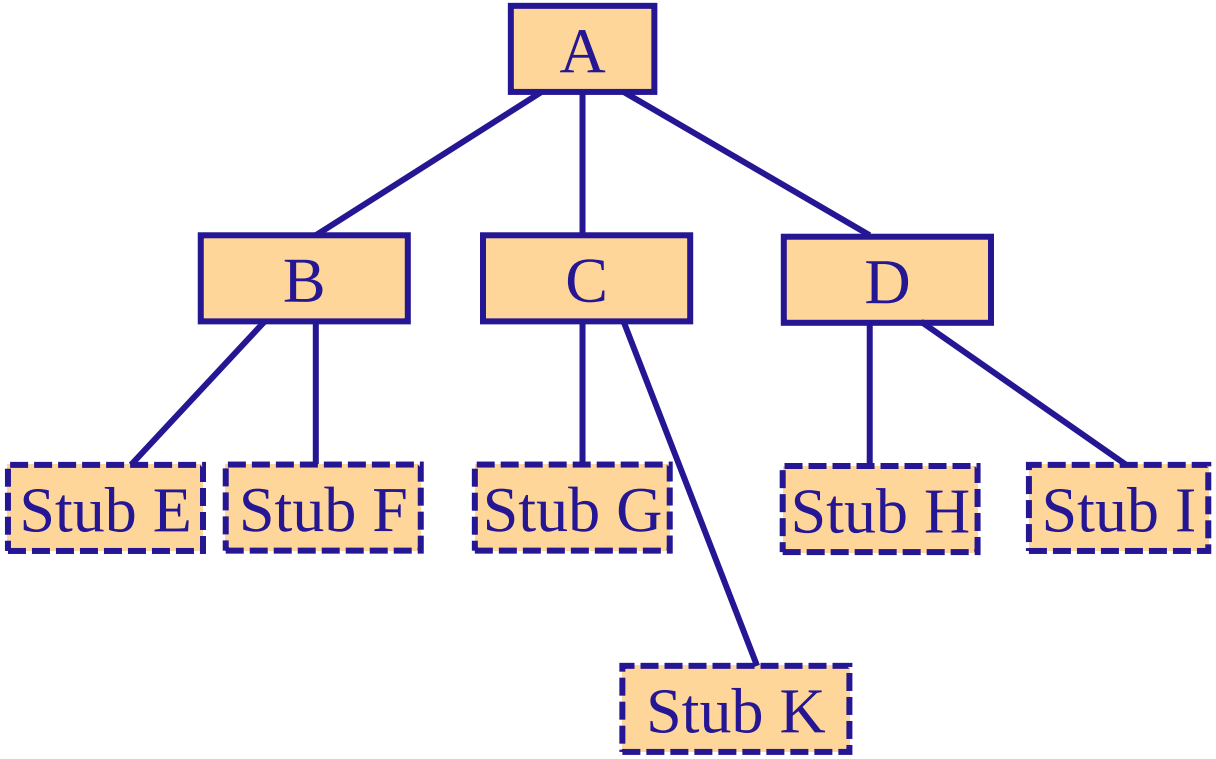
Top-Down Testing

- 依照同样的模式可以对C和D进行测试，即先寻找直接下属模块，再恢复自身在上一次测试中的桩模块为实际模块，然后把其直接下属模块作为本级的桩模块。



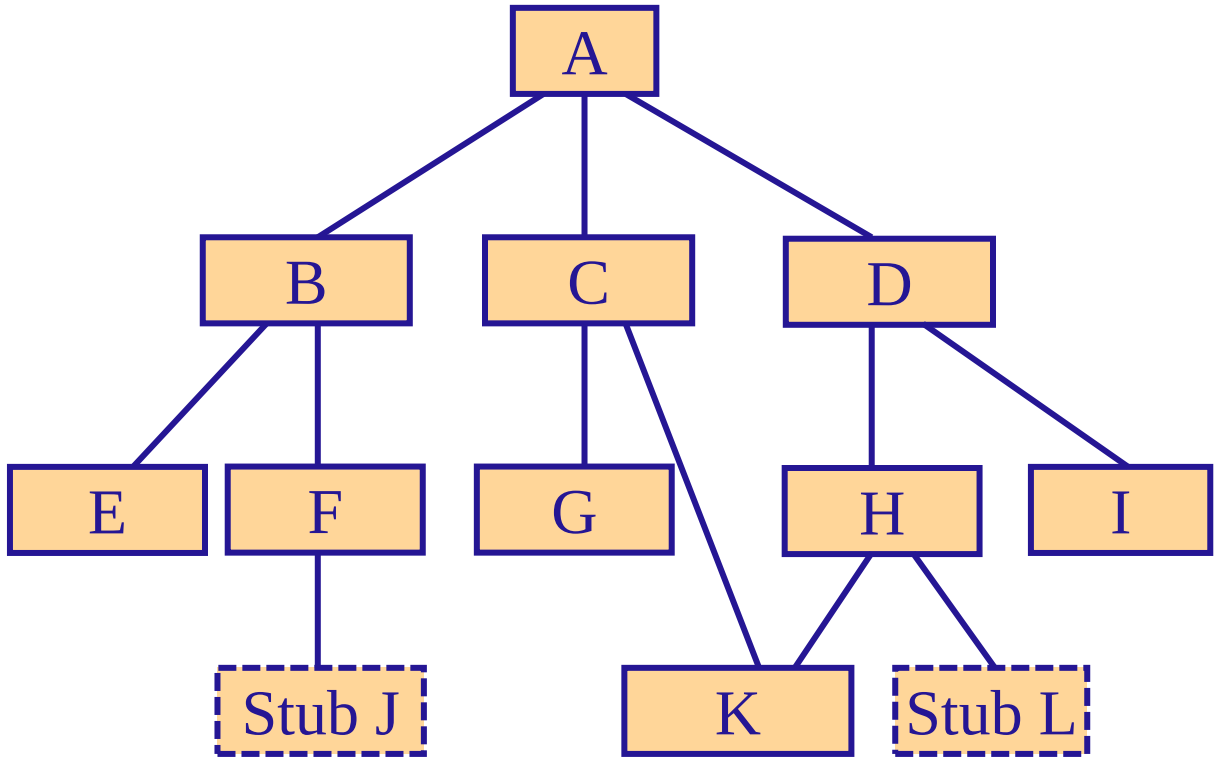
Top-Down Testing

- 然后对于EFGKHI也进行测试。如果为中间结点，则把其直接下属作为桩模块；如果为叶子结点，则程序已位于最底层，不需要建立桩模块。



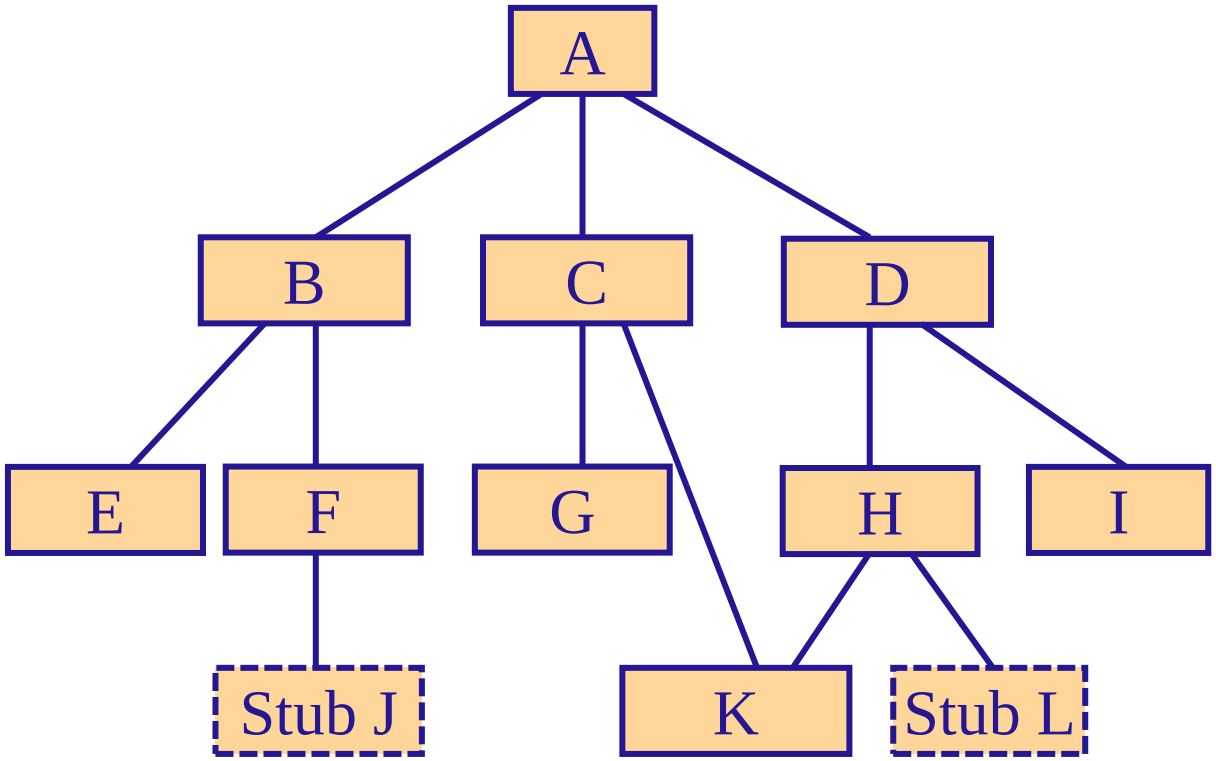
Top-Down Testing

- 然后对于EFGKHI也进行测试。如果为中间结点，则把其直接下属作为桩模块；如果为叶子结点，则程序已位于最底层，不需要建立桩模块。



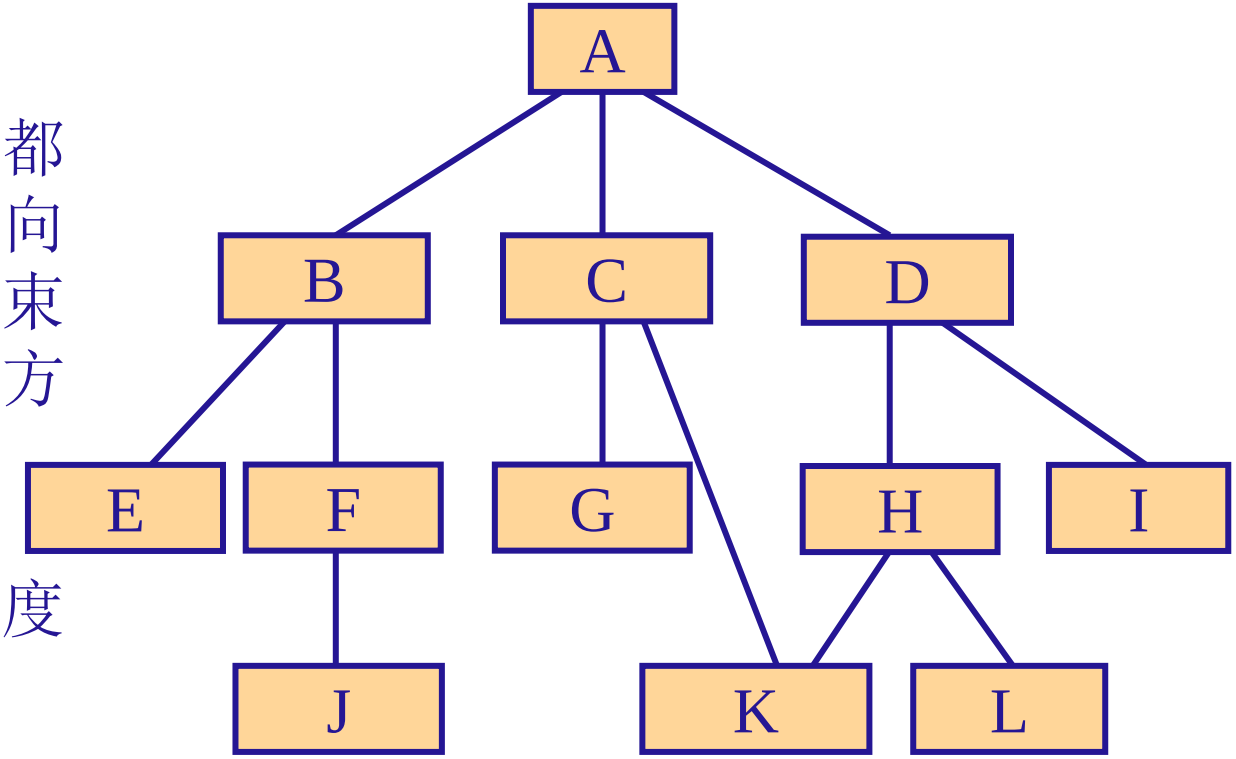
Top-Down Testing

- 这里需要注意模块K，它既是C也是H的直接下属。因此当EFGHI恢复为实际模块后，K已经测试并验证过了，能够根据程序本身提供有效的动态返回值，因此仍然是实际模块。



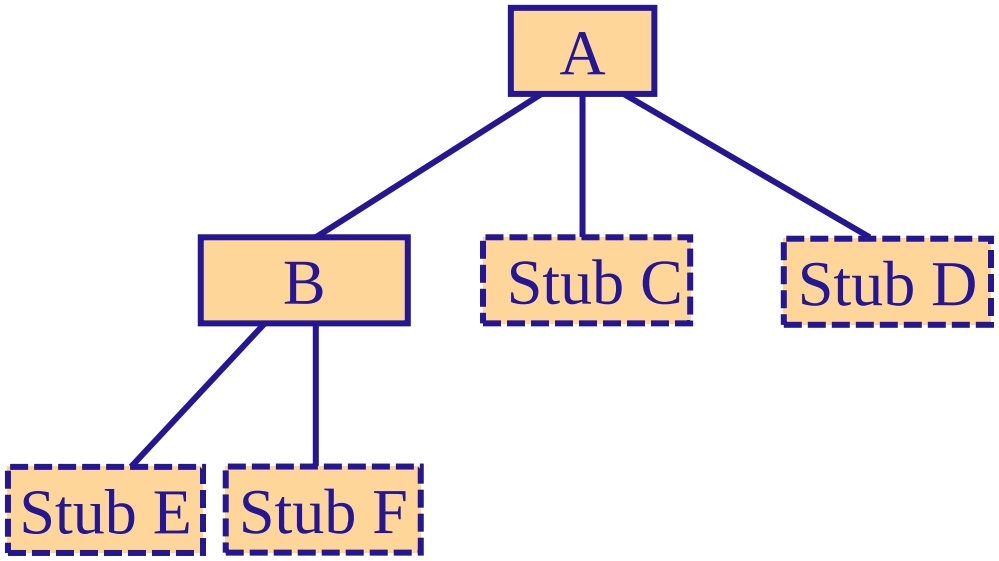
Top-Down Testing

- 当所有的叶子结点都测试结束后，自顶向下的测试也就结束了。按照刚才的方式，测试顺序是：ABCDEFGKHIJL
- 这种测试顺序是广度优先（BFS）。



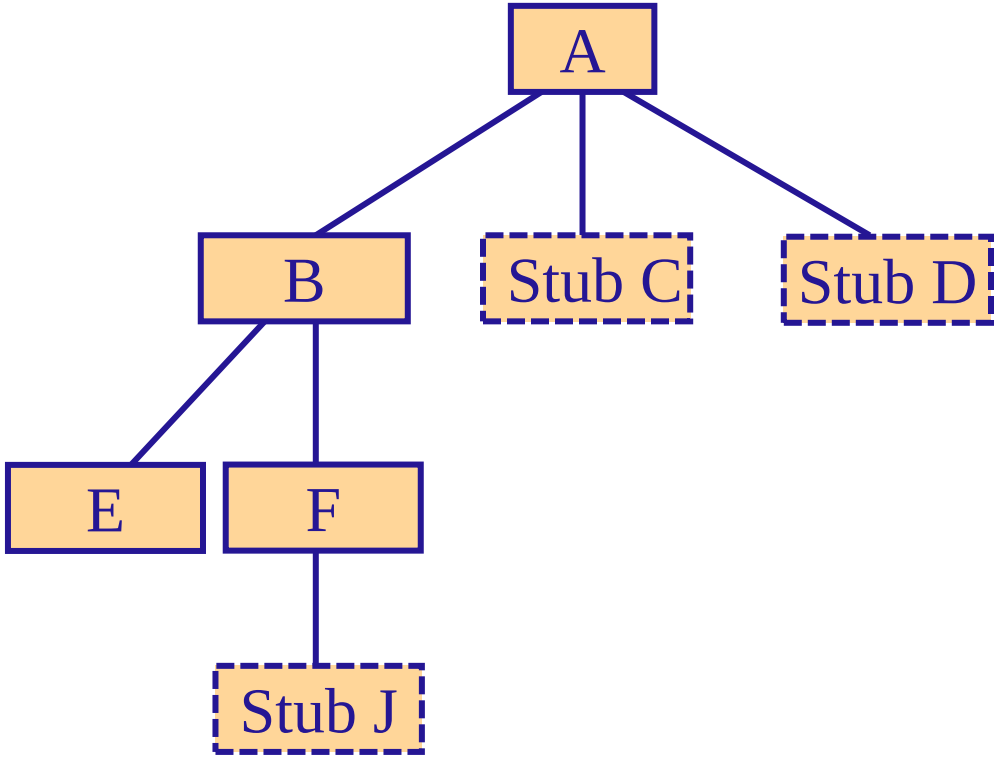
Top-Down Testing

- 当然也可以使用深度优先（DFS）。具体来看，在测试完B以后继续选择这一分支测试EF。



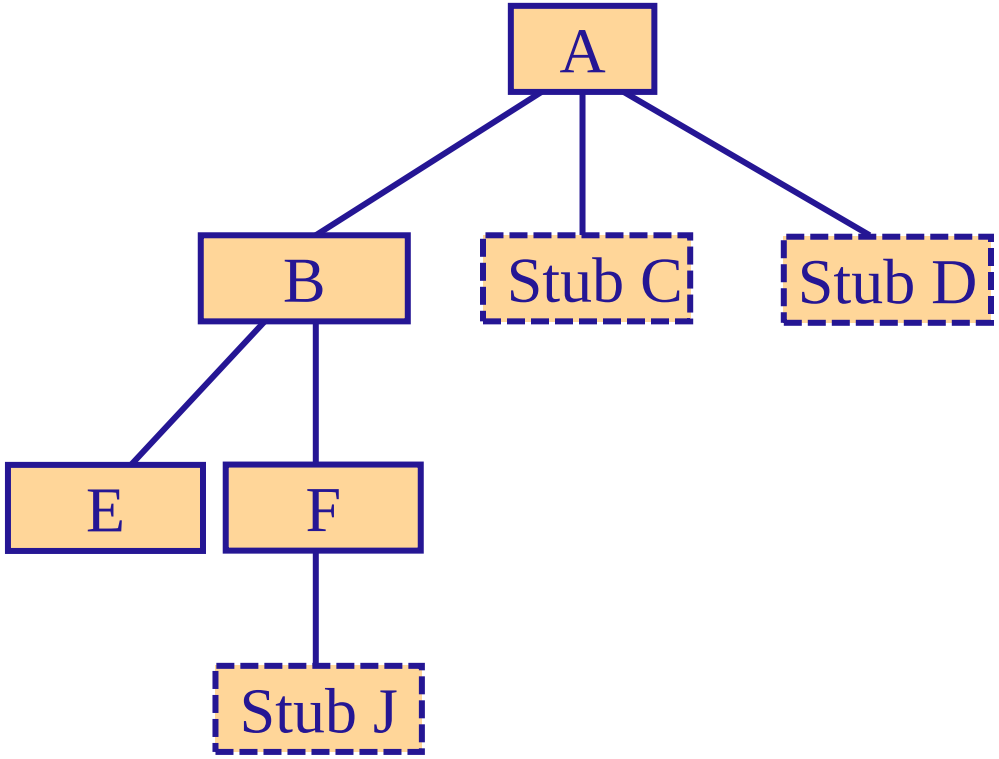
Top-Down Testing

- 当然也可以使用深度优先（DFS）。具体来看，在测试完B以后继续选择这一分支测试EF。



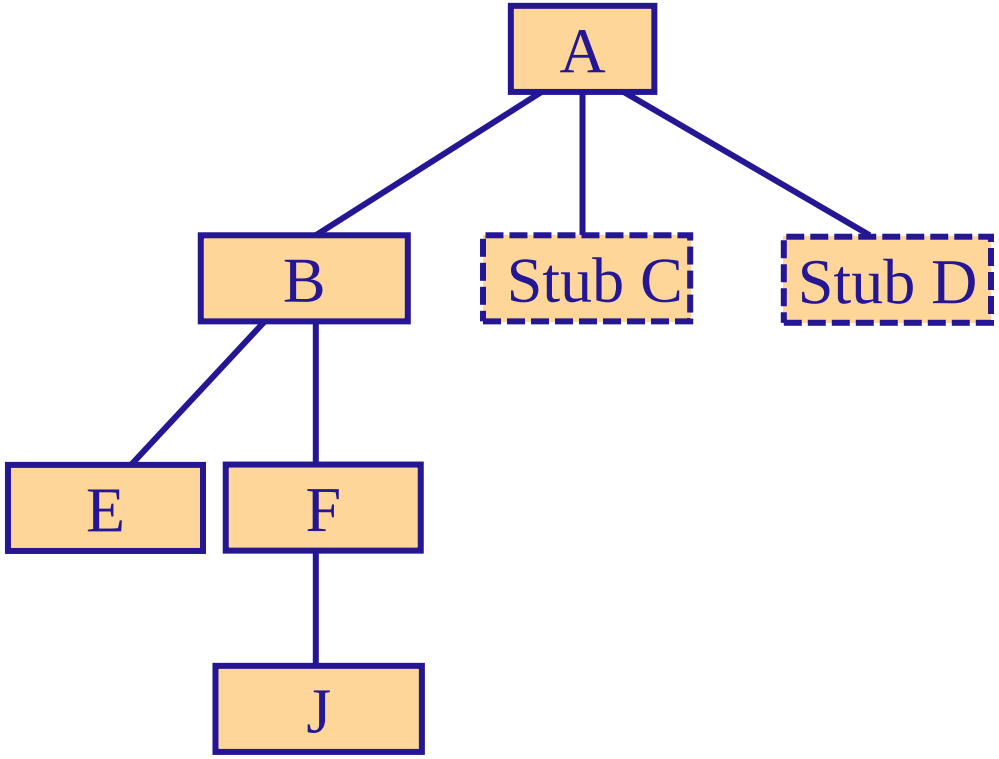
Top-Down Testing

- 测试完EF以后，E已经是叶子结点，而F有直接下属J，故测试J。

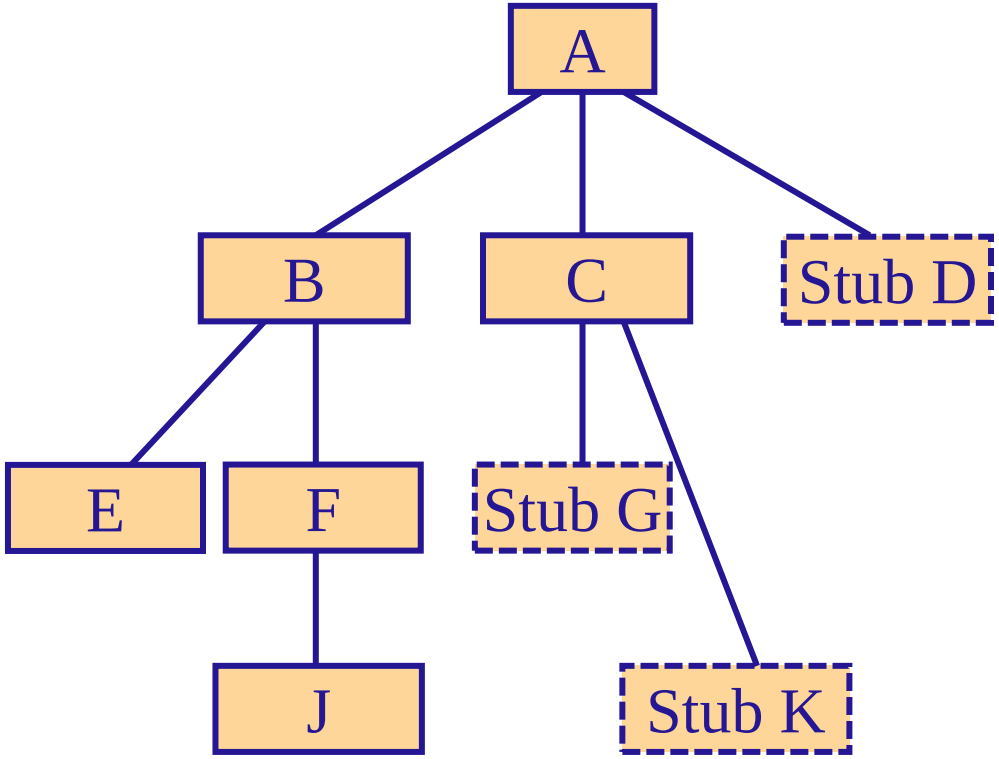


Top-Down Testing

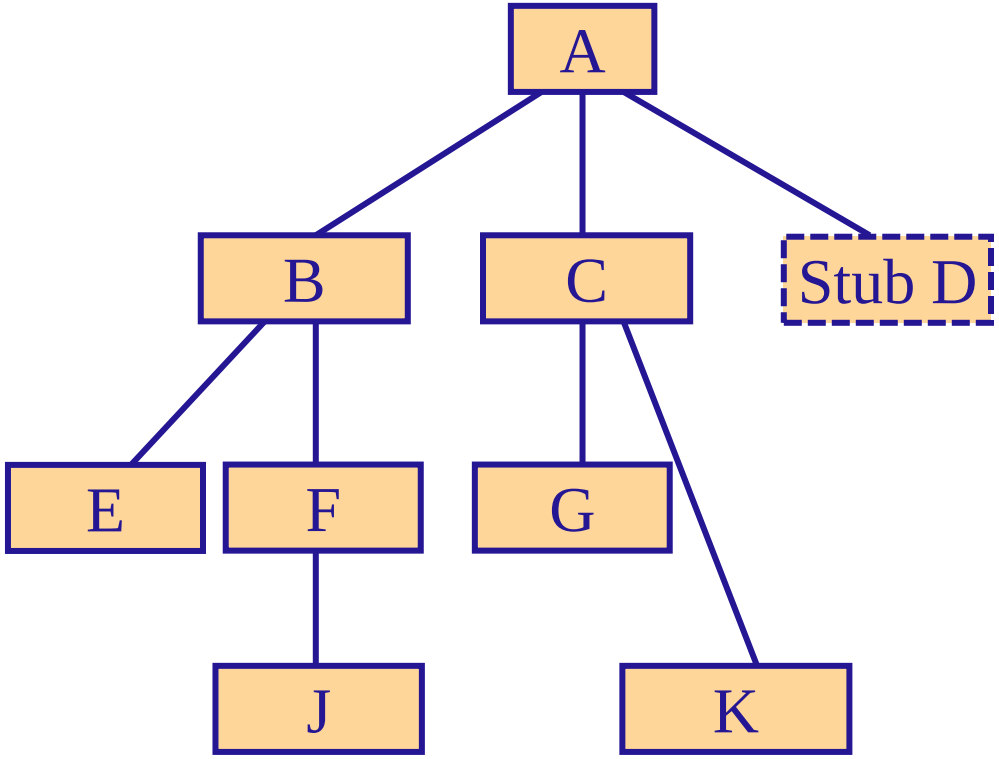
- 测试完EF以后，E已经是叶子结点，而F有直接下属J，故测试J。
- 测试结束后，再也找不到有下属的模块，故返回C继续后续测试。



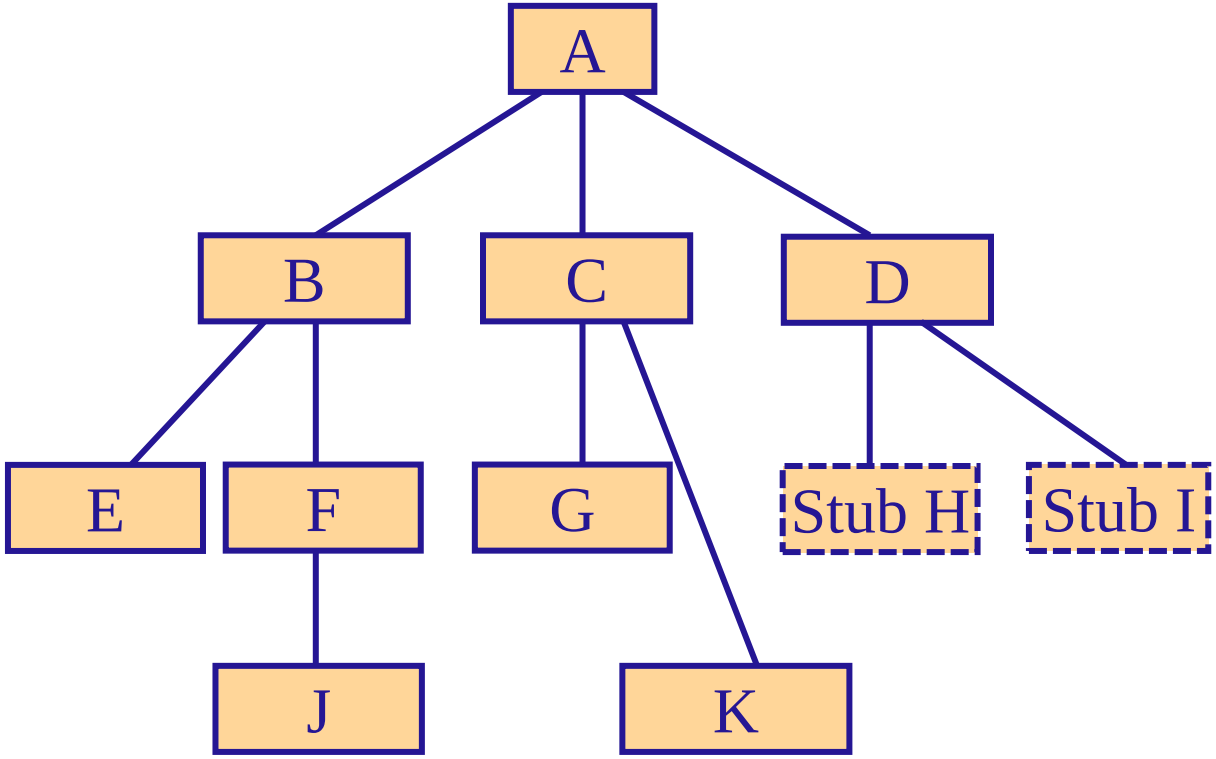
Top-Down Testing



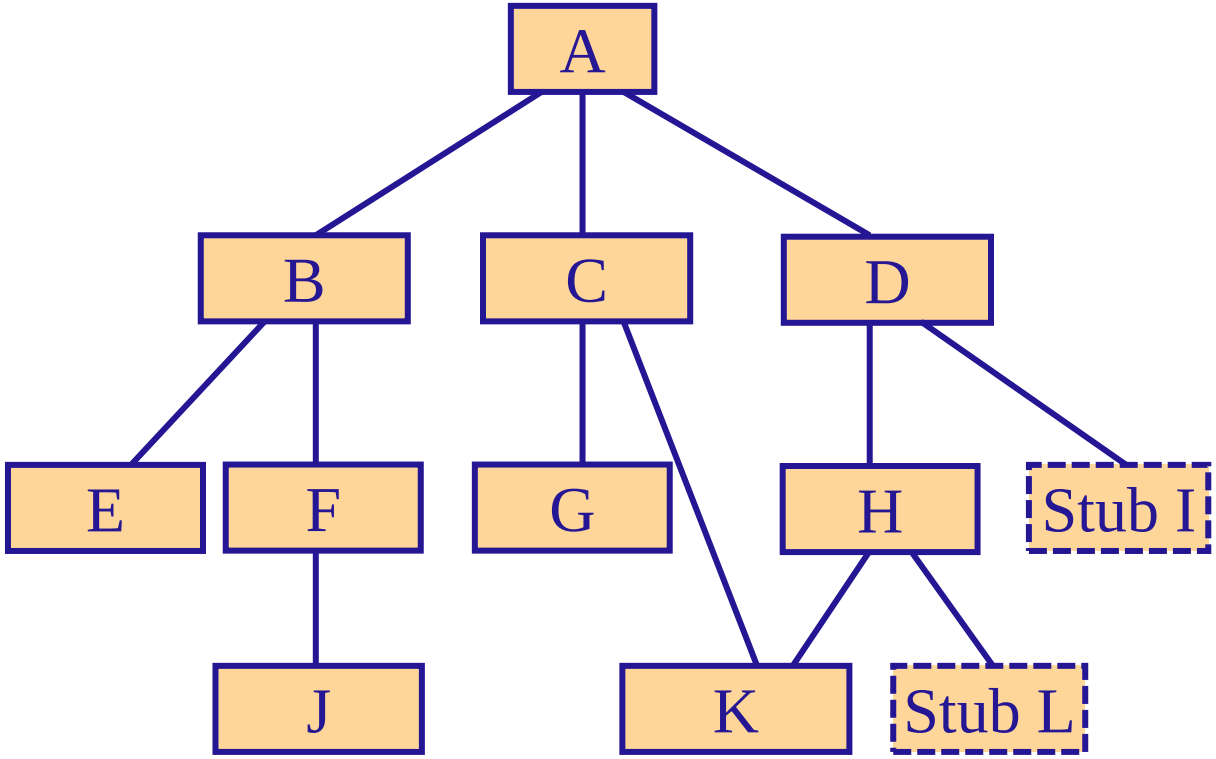
Top-Down Testing



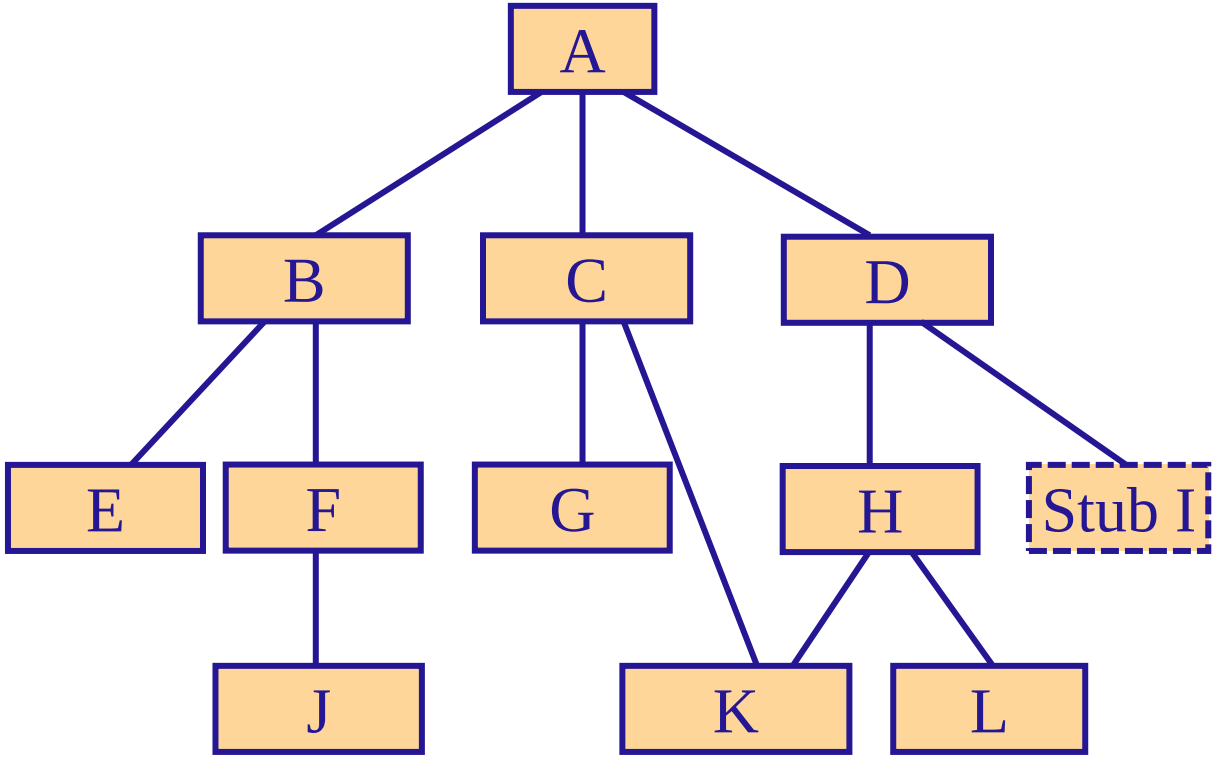
Top-Down Testing



Top-Down Testing



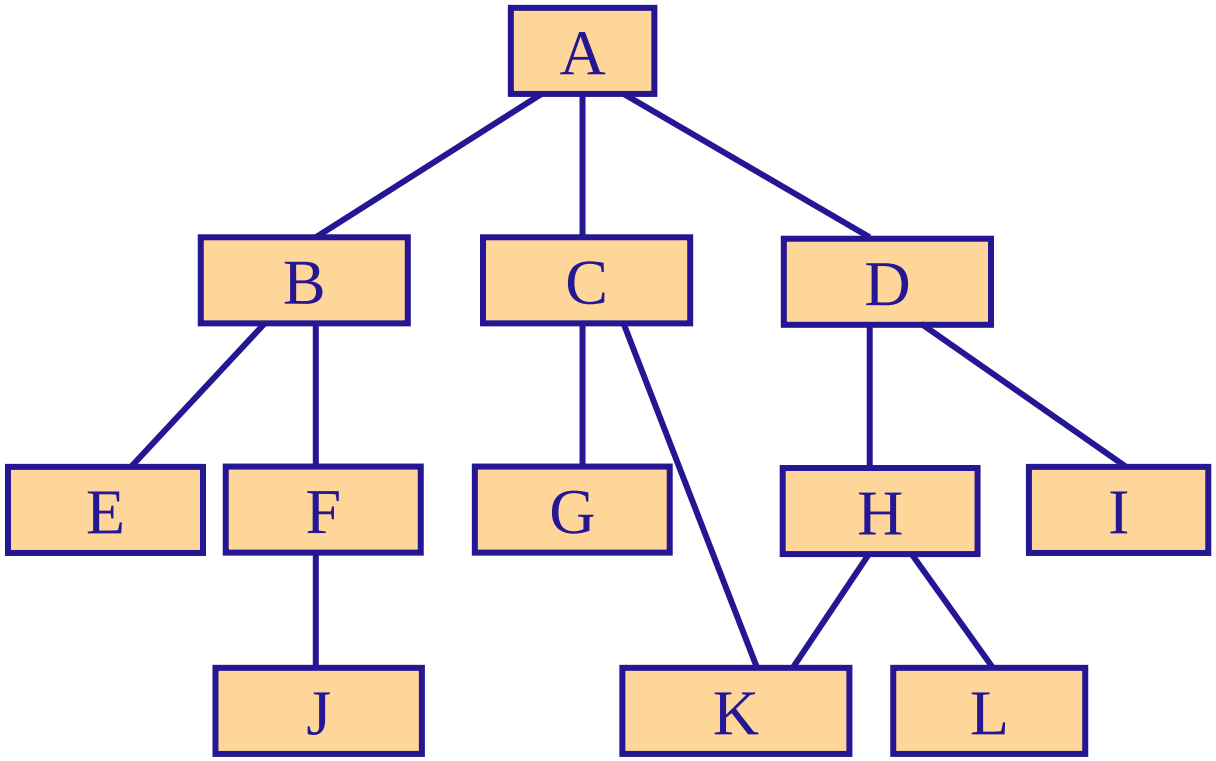
Top-Down Testing



Top-Down Testing

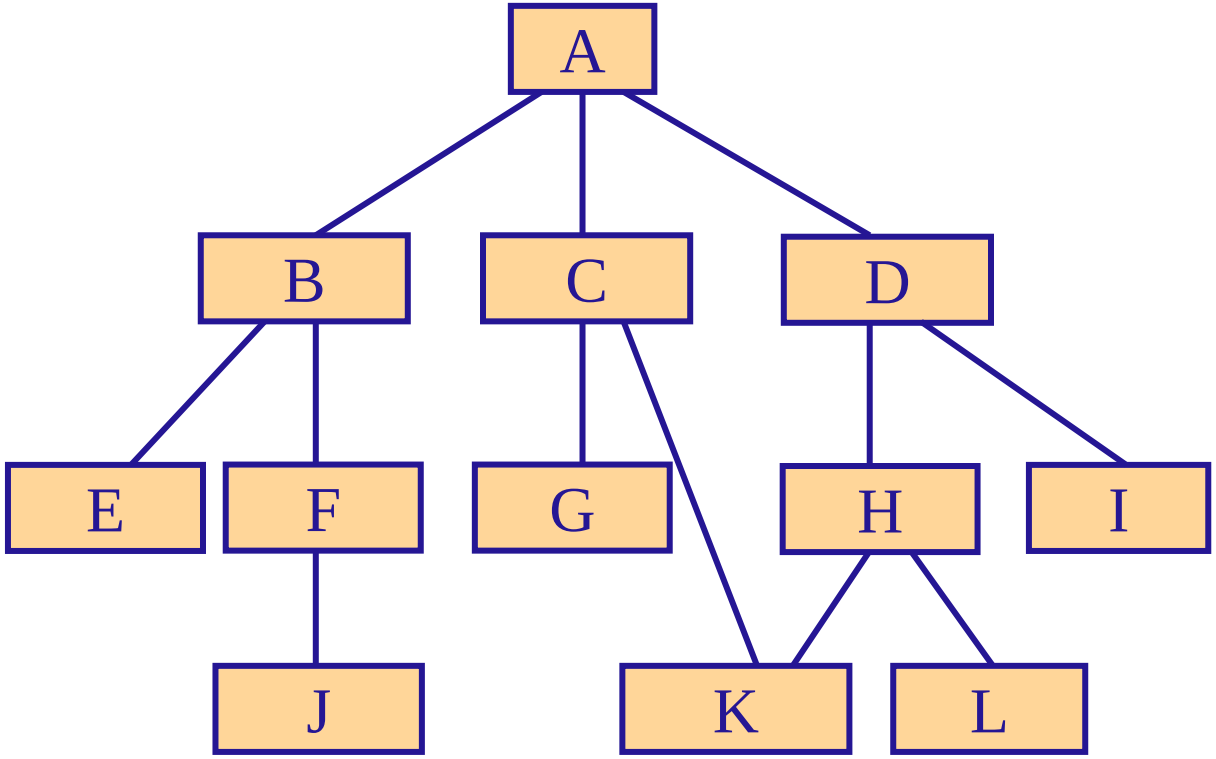
- 按照深度优先测试顺序是:

ABEFJCGKDHLI



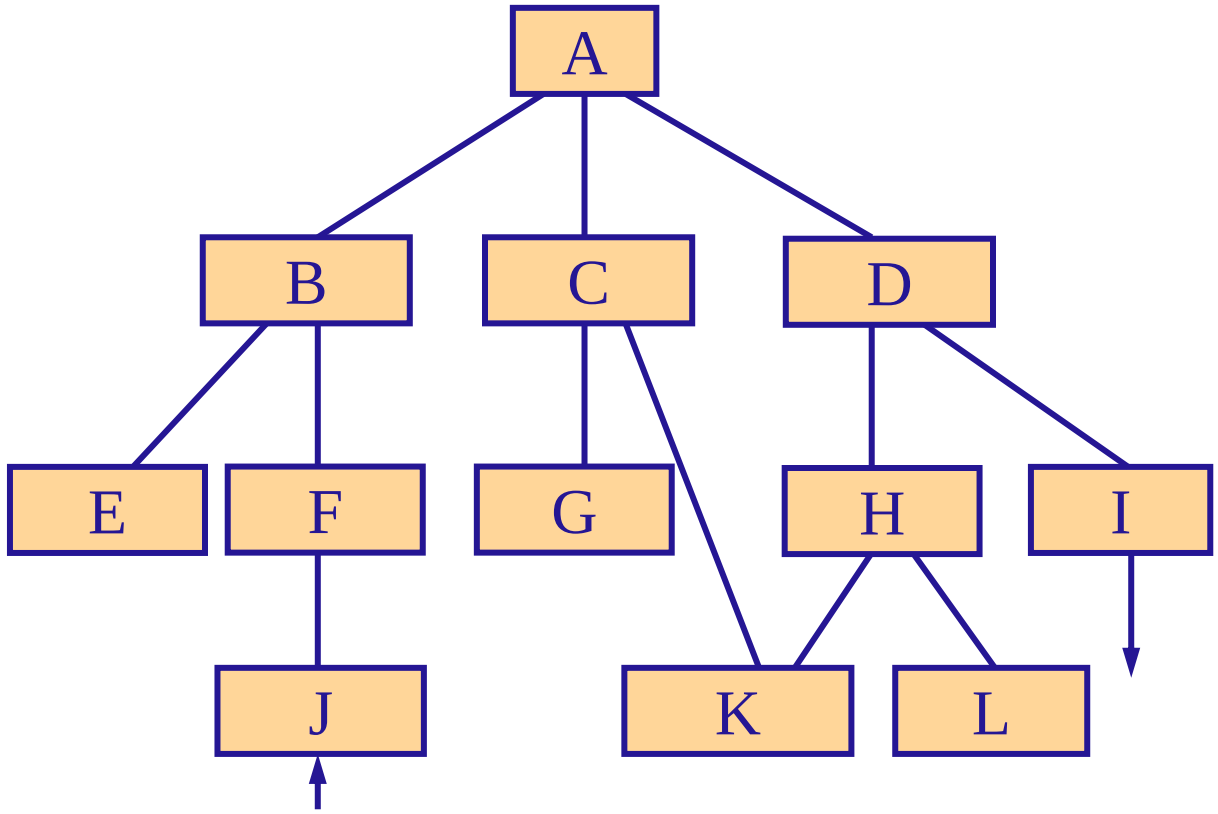
Top-Down Testing

- 需要注意的是，BFS和DFS只是两种策略，实际中也会出现两者掺杂混合情况。



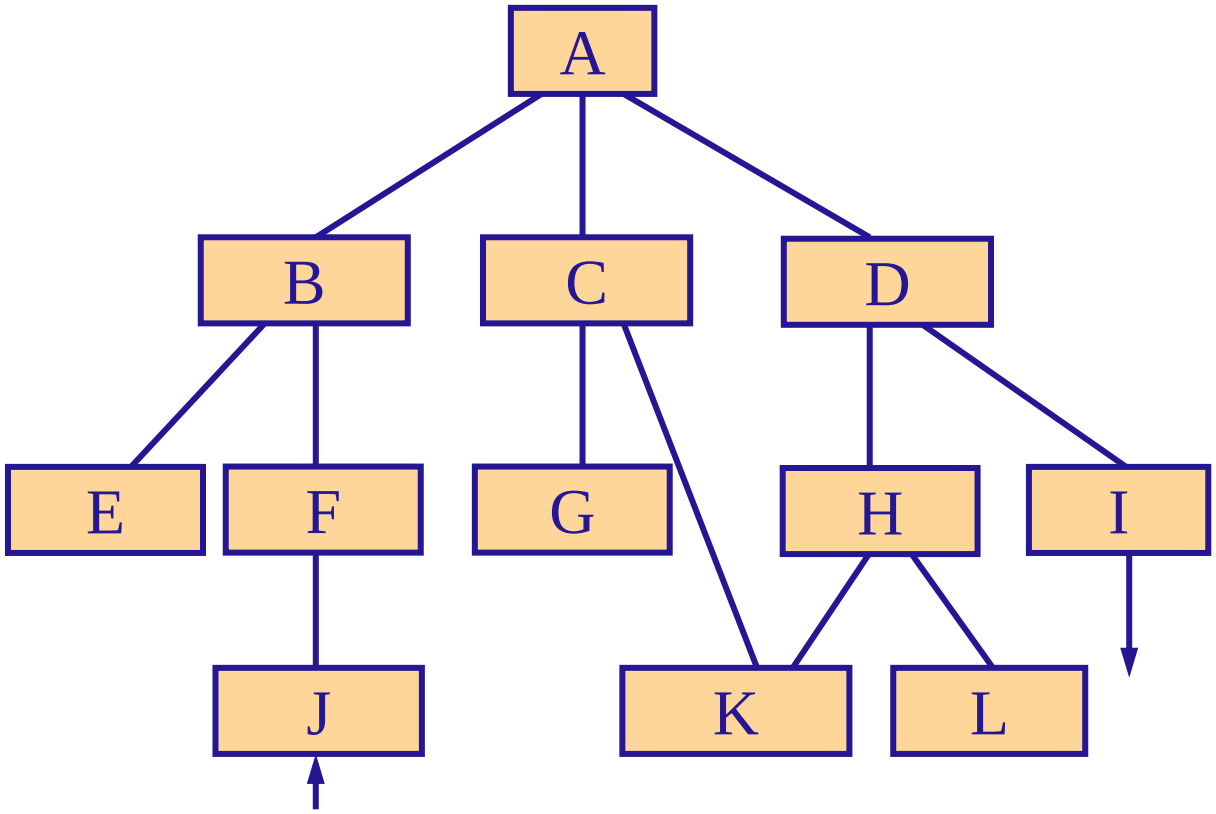
Top-Down Testing with I/O

- 图中共有12个模块A到L，模块I包含I/O的写操作，模块J包含I/O的读操作。有多种可能的测试序列时，应考虑优先测试I/O模块。这时候就不能单用BFS或DFS。



Top-Down Testing with I/O

- 如先按照BFS走，测试完F以后，J是读操作，因此下一个就按照DFS来测试J；同样，按BFS测试完D以后，I是写操作，因此按照DFS先测试I。



Incremental Testing

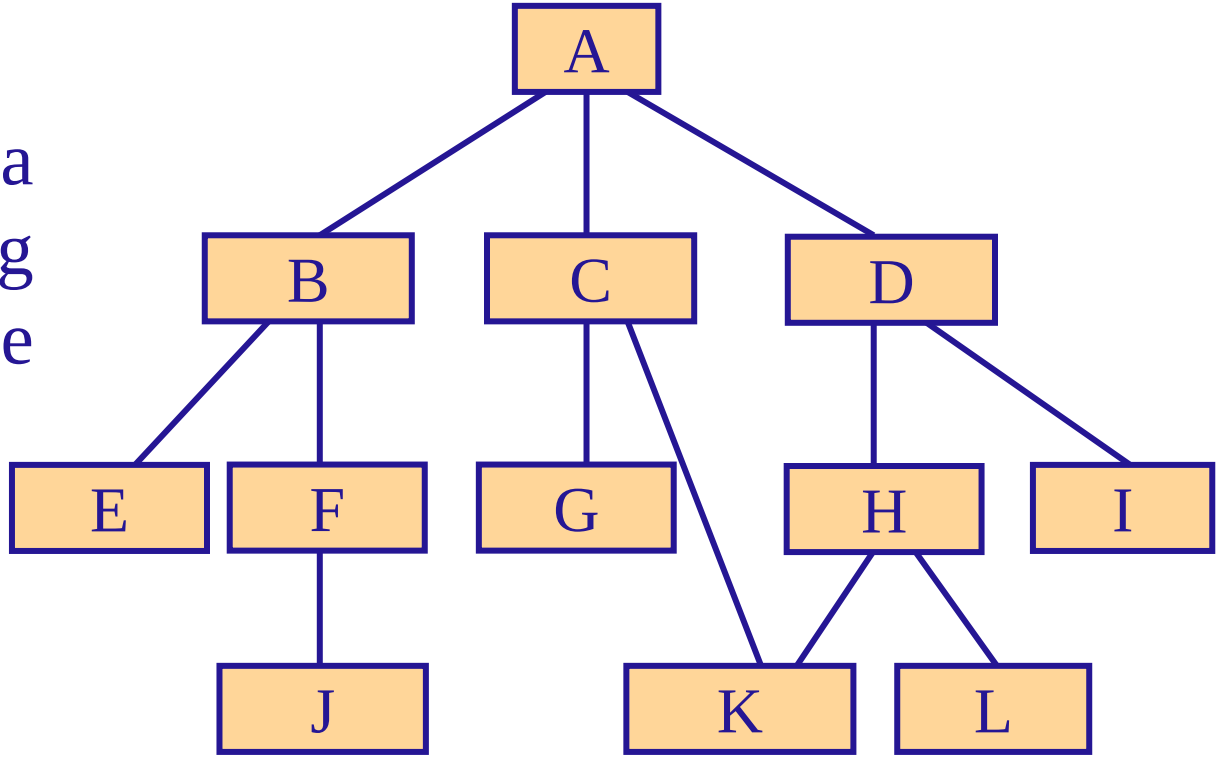
- Top-Down Testing
- **Bottom-Up Testing**

Bottom-Up Testing

- The bottom-up testing starts from the leaf nodes that are the terminal of a program. Before testing, all the modules that are called by the tested unit have been tested.
- To test the lower unit, the driver modules are established.

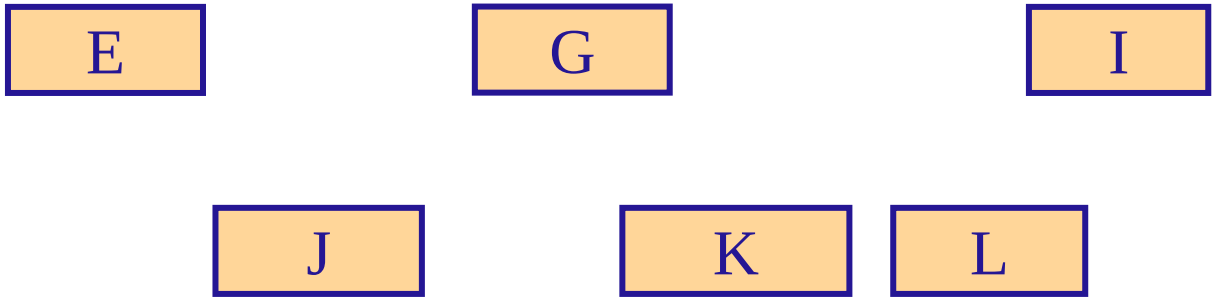
Bottom-Up Testing

- How to design a bottom-up testing strategy for the issue?



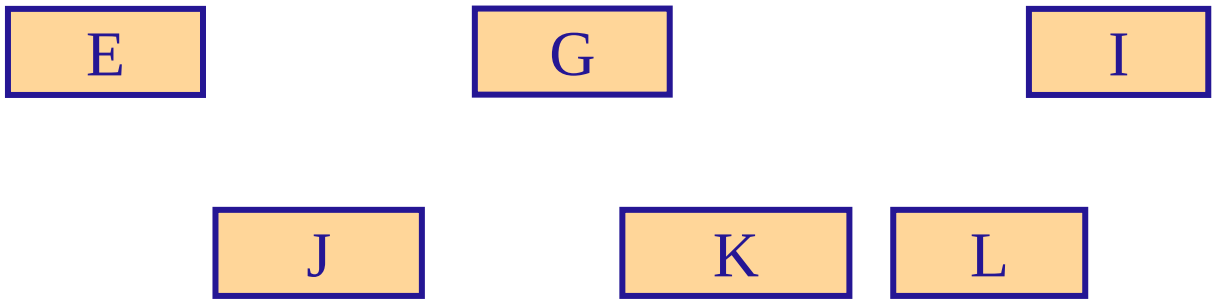
Bottom-Up Testing

- 首先找出全体叶子结点EJGKLI。



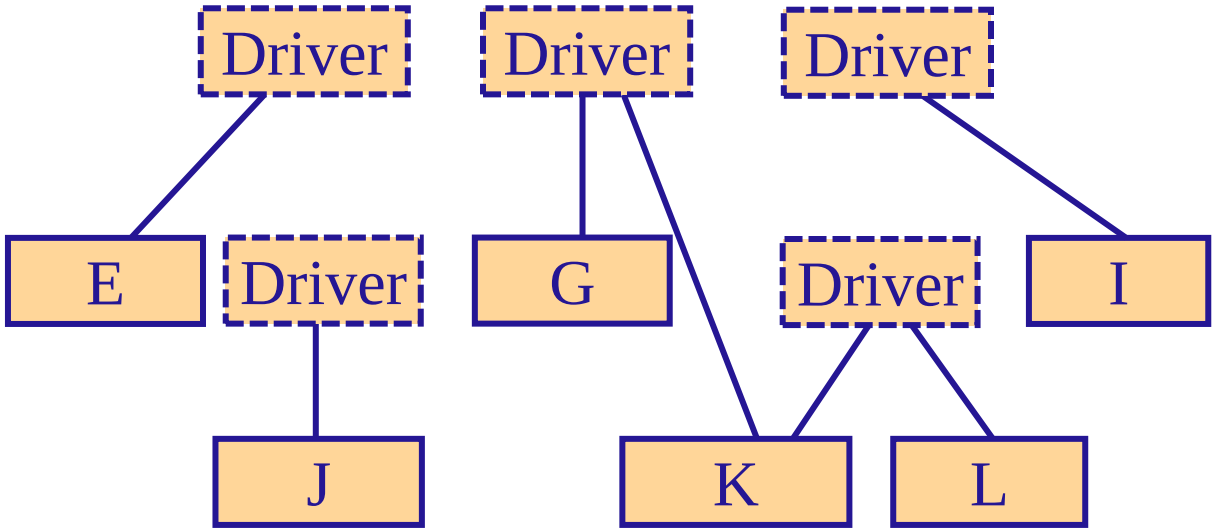
Bottom-Up Testing

- 欲测试这些叶子结点的模块，必须设置对应的驱动模块来提供测试运行的环境。有的驱动模块可以供好几个模块使用。



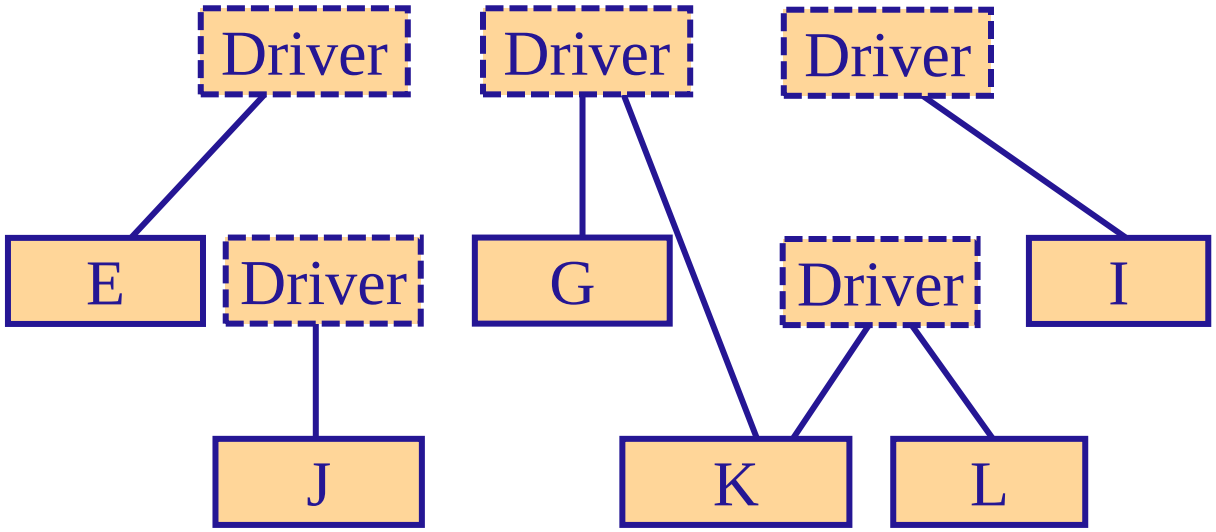
Bottom-Up Testing

- 欲测试这些叶子结点的模块，必须设置对应的驱动模块来提供测试运行的环境。有的驱动模块可以供好几个模块使用。



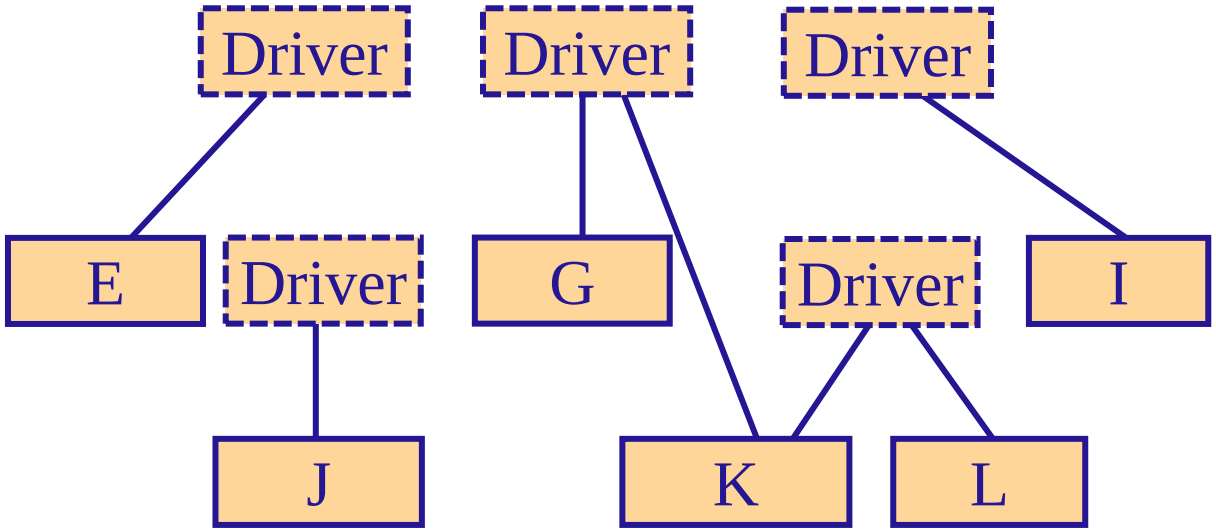
Bottom-Up Testing

- EJGKLI这些叶子结点模块，既可以全选，也可以分批每次只选择其中一部分；既可以串行测试，也可以并行测试。但必须保证测试高一级模块时所有下属结点都要测试过。



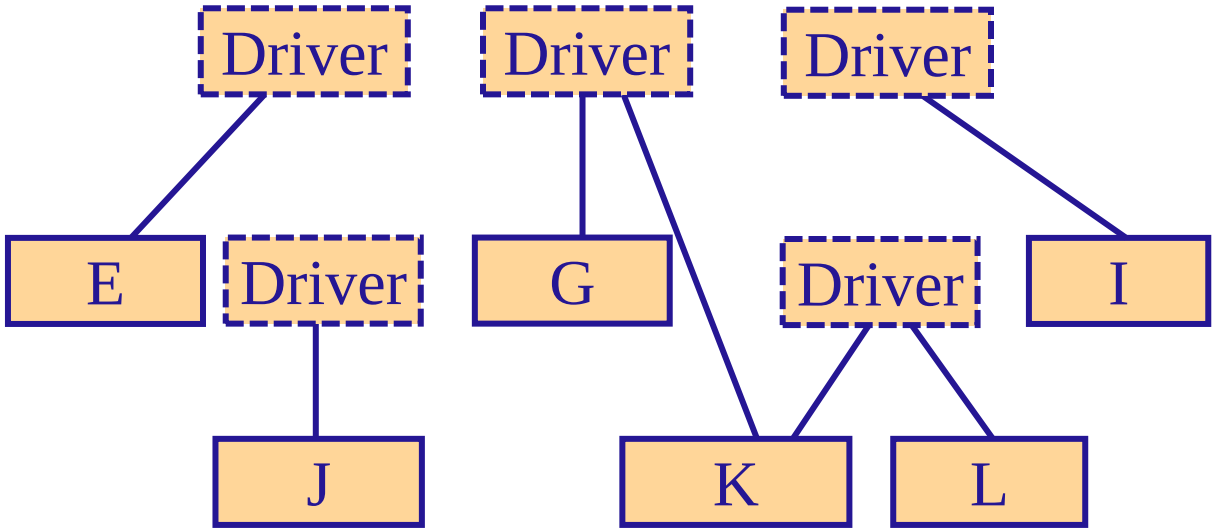
Bottom-Up Testing

- 进行高一级测试时，需要将驱动模块恢复为对应位置的实际模块。而能否恢复，就要考察其所有直接下属模块是否已全部测试过。直观来看就是驱动模块下是否已不存在Driver。



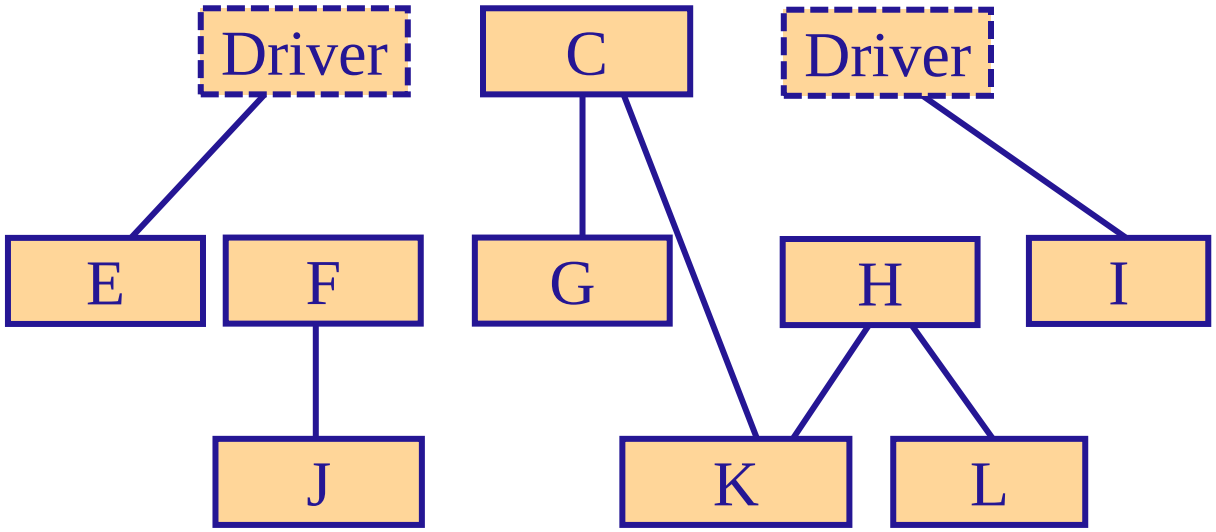
Bottom-Up Testing

- 例如FCH可以直接转为实际模块，因为其直接下属都已经被测试过；但BD不能转为实际模块，因为其下属有含有Driver的部分。



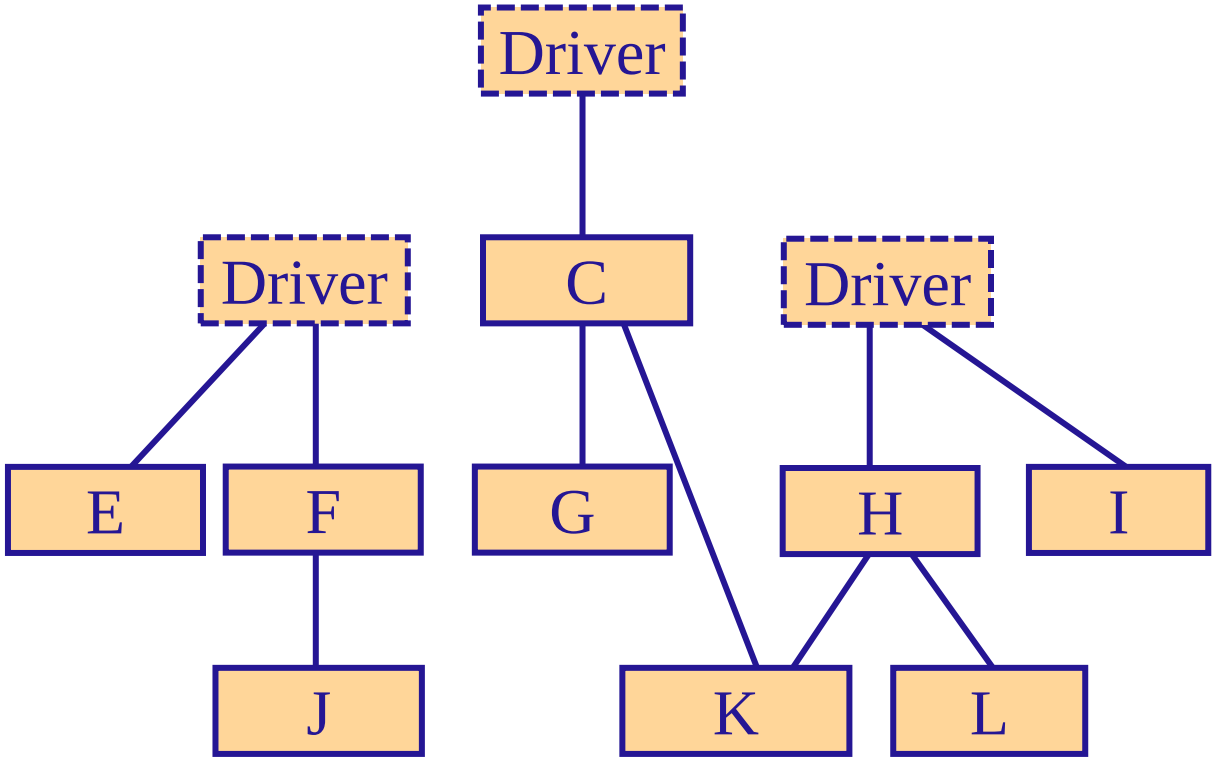
Bottom-Up Testing

- 例如FCH可以直接转为实际模块，因为其直接下属都已经被测试过；但BD不能转为实际模块，因为其下属有含有Driver的部分。



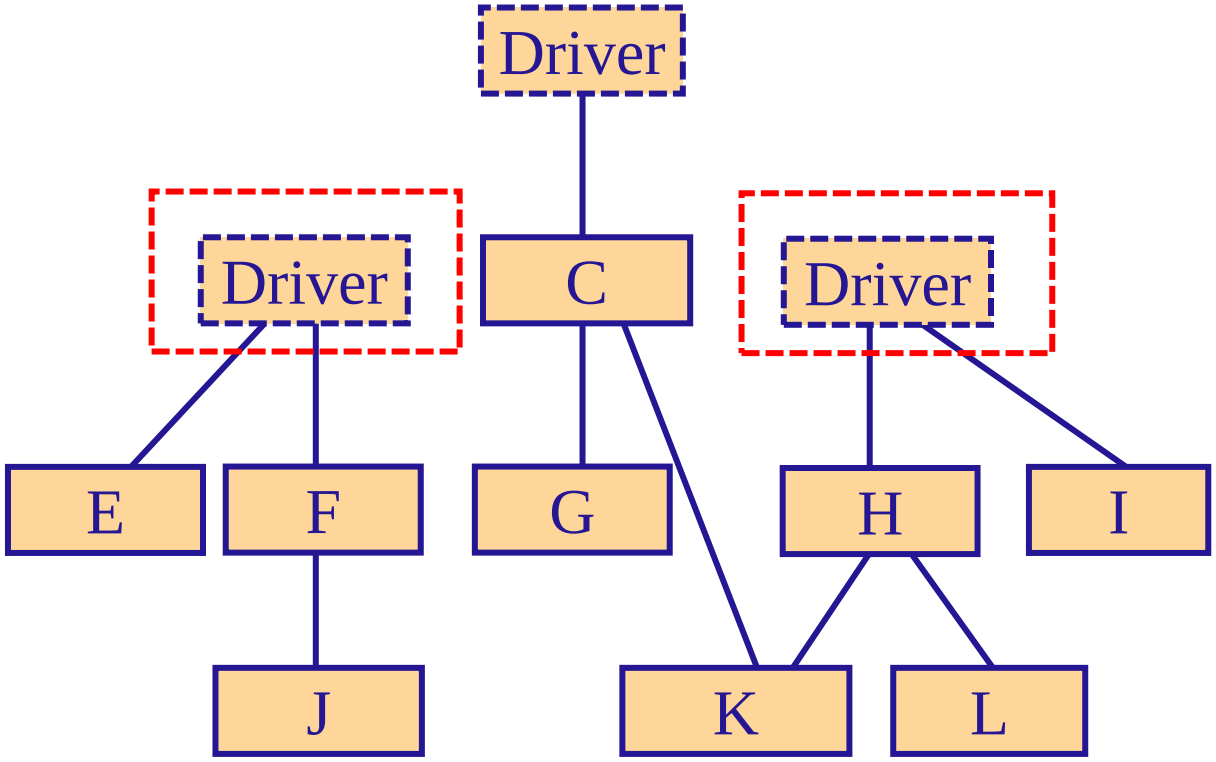
Bottom-Up Testing

- FCH恢复后就要对它们进行测试，这仍需要上一级的驱动模块作为其运行环境。



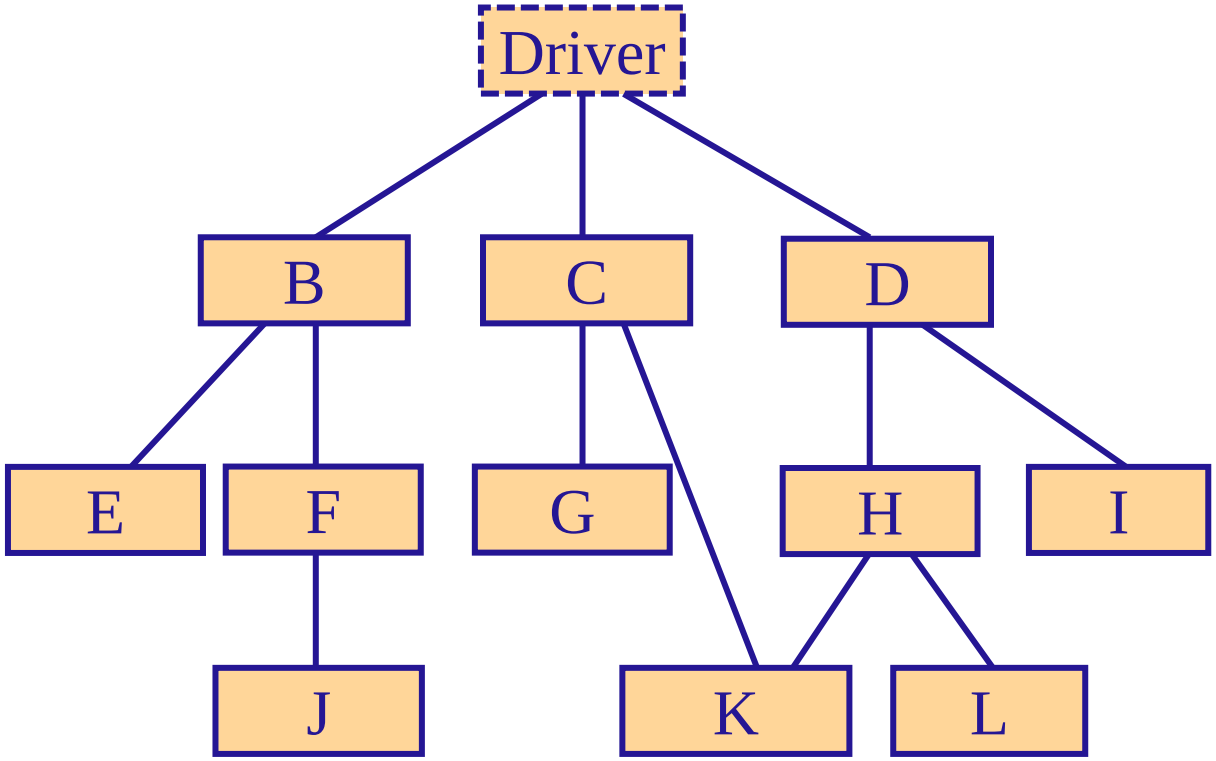
Bottom-Up Testing

- 注意FH的驱动模块与先前测试EI的两个Driver虽然在图中位置相同，内容却是不同的。因为先前的Driver只针对EI，而现在的Driver针对FH。这是Driver后面不加字母的原因。



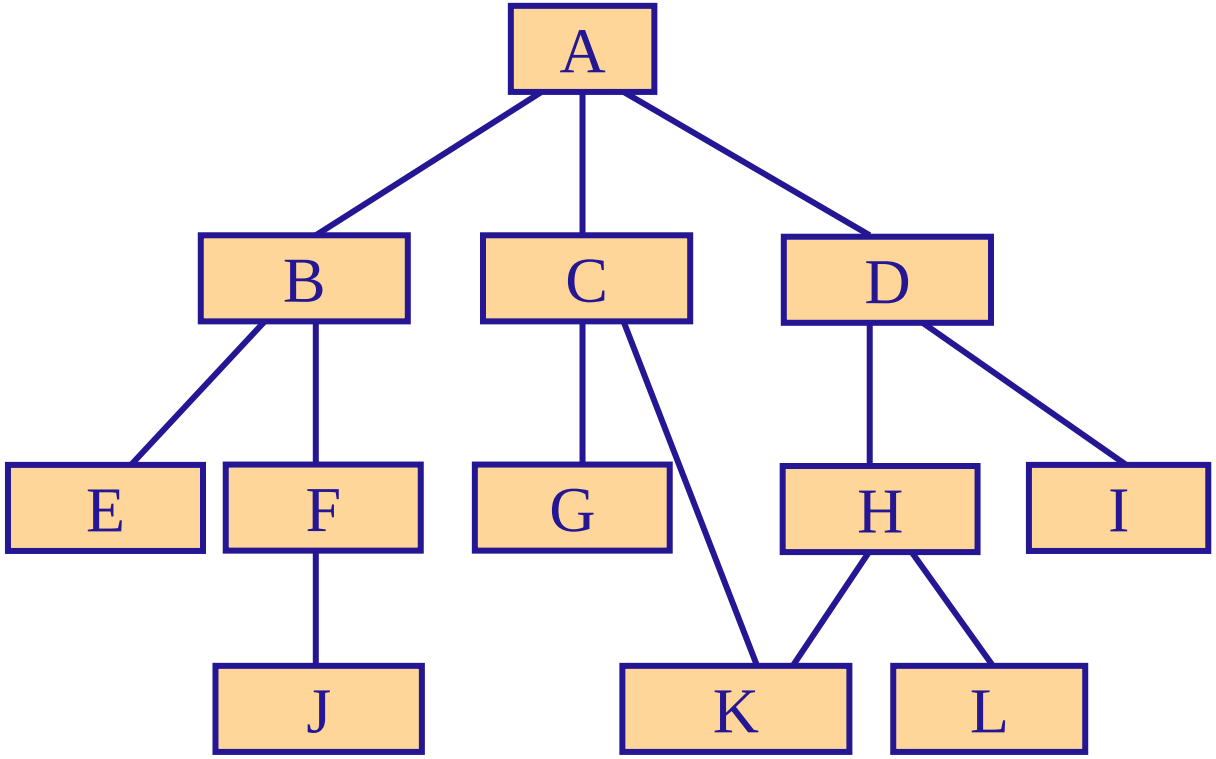
Bottom-Up Testing

- FCH测试结束之后，BD下的所有直接下属都测试过，因此予以恢复。这样继续测试BD



Bottom-Up Testing

- BD测试结束后，可以恢复A并对其进行测试。而A是根结点，不需作为驱动模块的运行程序。因此测试自底向上测试完毕。



Example 3

- Please design the up-down and bottom-up testing strategies.

```
%Main
x=3;
y1=A(x);
y2=B(x,y1);
y3=C(x,y2);
t=y1+y2*y3;
```

```
function y=A(x)
y=D(x)+E(x);
```

```
function z=B(x,y)
z=E(x)-F(y);
```

```
function z=C(x,y)
z=5;
if x>0||y>0
    z=G(x)*G(y);
end
```

```
function y=D(x)
y=x^2+3*x+5;
```

```
function y=E(x)
y=H(x)*x^3;
```

```
function y=F(x)
y=H(x)*I(x);
```

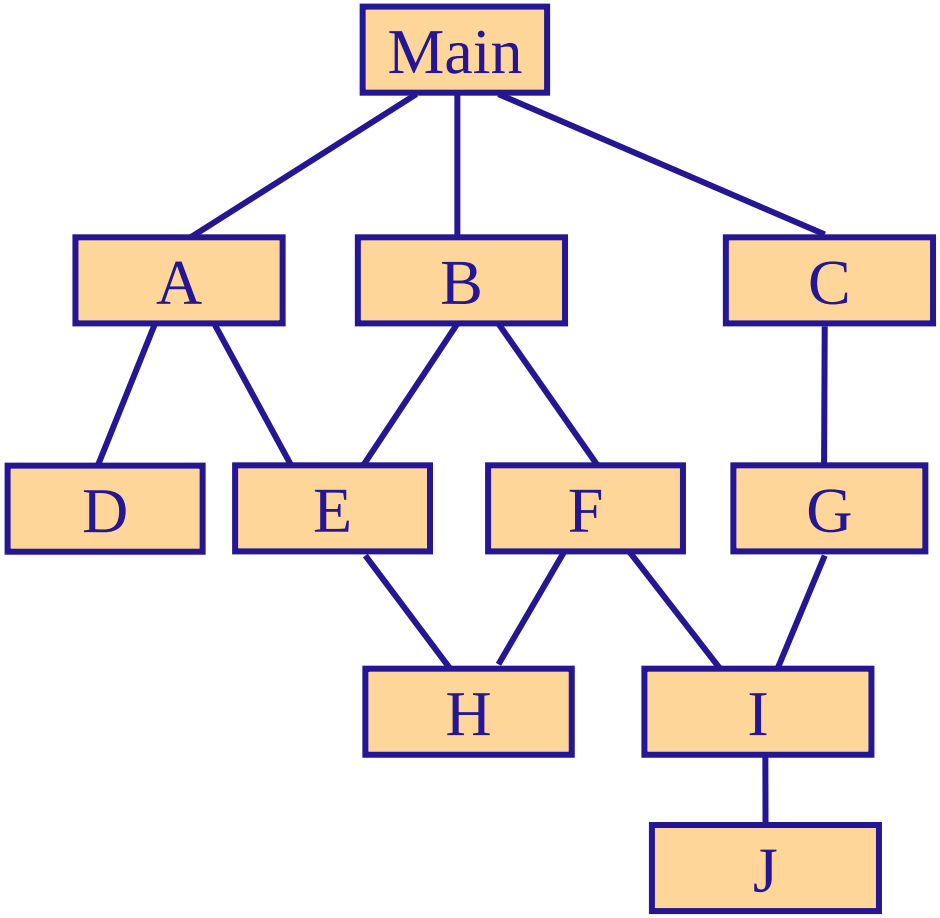
```
function y=G(x)
y=I(x)^2+5*I(x);
```

```
function y=H(x)
y=0;
if x>=0
    y=1;
end
```

```
function y=I(x)
y=0;
for i=1:ceil(x)
    y=y+1/J(i);
end
y=floor(y);
```

```
function y=J(x)
x=ceil(x);
if rem(x,2)==0
    y=sqrt(x);
else
    y=x;
end
```

Example 3



Comparison

自顶向下	优点	1、若主要缺陷出现于顶层则非常有利 2、预知框架结构，因此能提早发现主要控制问题
	缺点	1、必须开发桩模块，可能它们比最初表现更复杂 2、创建测试环境可能很难，甚至无法实现 3、观测测试输出比较困难
自底向上	优点	1、若主要的缺陷发生在程序的底层将非常有利 2、提早发现程序当中的主要算法问题 3、测试环境比较容易建立 4、观测测试输出比较容易
	缺点	1、必须开发驱动模块 2、直到最后一个模块添加进去，程序才形成整体

THANK YOU!