Chapter 9

Regression Testing

(回归测试)

要点

1. 回归测试的概念
2. 回归测试策略
3. 回归测试的步骤
4. 回归测试用例选择方法

1 回归测试的概念

在软件生命周期中的任何一个阶段，只要软件发生了修改和改变，就可能给该软件带来问题。

- 对问题的修改被遗漏；
- 所做的修改只修正了错误的外在表现；
- 导致软件未被修改的部分产生出新的问题。

因此，每当软件发生变化时，我们就必须重新测试现有的功能，以便确定软件修改是否达到了**预期的目的**，检查**修改是否损害了原有的正常功能**。同时，还需要**补充新的测试用例**来测试新的或被修改了的功能。为了**验证修改的正确性及其影响**，就需要进行回归测试。

回归测试策略

Version 1	Version 2
1. Develop *P*	4. Modify *P* to *P′*
2. Test *P*	5. Test *P′* for new functionality
3. Release *P*	6. Perform regression testing on *P′* to ensure that the code carried over from *P* behaves correctly
	7. Release *P′*

What tests to use?

- **Idea 1:**

- **All valid tests from the previous version and new tests created to test any added functionality. [This is the TEST-ALL approach.]**

- **What are the strengths and shortcomings of this approach?**

The test-all approach

- **The test-all approach is best when you want to be certain that the the new version works on all tests developed for the previous version and any new tests.**

- **But what if you have limited resources to run tests and have to meet a deadline? What if running all tests as well as meeting the deadline is simply not possible?**

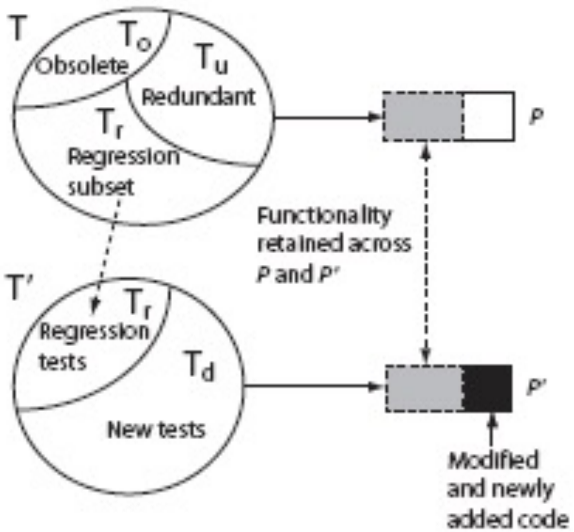
Test selection

- **Idea 2:**

- **Select a subset Tr of the original test set T such that successful execution of the modified code P' against Tr implies that all the functionality carried over from the original code P to P' is intact.**

- **Finding Tr can be done using several methods. We will discuss three of these.**

Regression Test Selection problem



- Given test set T, our goal is to determine Tr such that successful execution of P′ against Tr implies that modified or newly added code in P′ has not broken the code carried over from P.

- Note that some tests might become obsolete when P is modified to P′. Such tests are not included in the regression subset Tr.

概 念

- 测试用例库： 对于一个软件开发项目来说，项目的测试组在实施测试的过程中，会将所开发的测试用例保存到“测试用例库”中，并对其进行维护和管理 。
- 基线测试用例库 ： 当得到一个软件的基线版本时，用于基线版本测试的所有测试用例就形成了基线测试用例库

测试用例库的维护

— 随着软件的升级，软件的功能和应用接口以及软件的实现都可能发生了演变，测试用例库中的一些测试用例就可能会失去针对性和有效性，而另一些测试用例可能会变得过时，甚至完全不能运行。为了保证测试用例库中测试用例的有效性，必须对测试用例库进行维护。

- 删除过时的测试用例
- 改进不受控制的测试用例
- 删除冗余的测试用例
- 增添新的测试用例

回归测试包的选择

选择回归测试策略应该兼顾效率和有效性两个方面。常用的选择回归测试的方式包括：

- **再测试全部用例**

  再测试全部用例具有最低的遗漏回归错误的风险，但是测试成本最高。

- **基于风险选择测试**

  首先运行最重要的、关键的和可疑的测试，逐步降低风险值,直至满足回归测试要求。

- **基于操作剖面选择测试**

  测试用例是基于软件操作剖面开发的，优先选择那些针对最重要或最频繁使用功能的测试用例。

- 再测试修改的部分

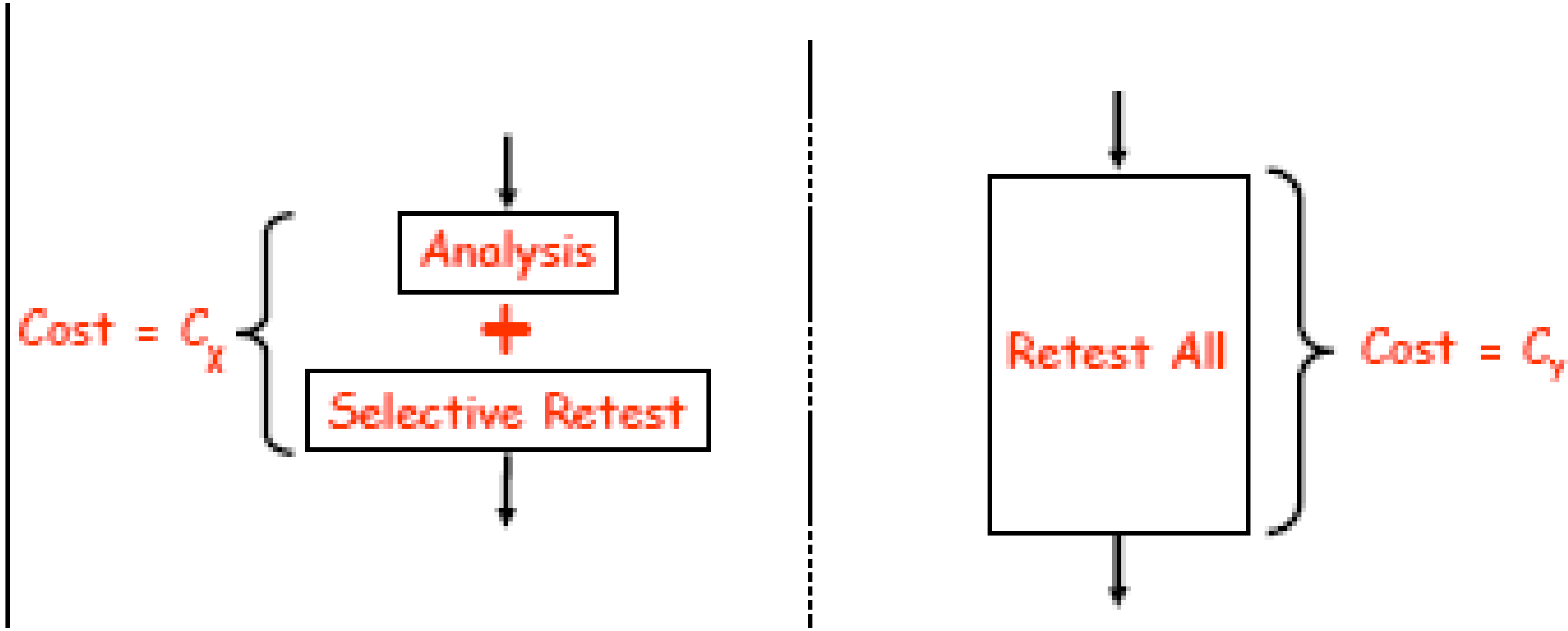
　　当测试者对修改的局部有足够的信心时，可以通过等价性分析，识别软件的修改情况并分析修改的影响，将回归测试局限于被改变的模块和它的接口上。

3 回归测试的步骤

回归测试可遵循下述基本过程进行：

1. 识别出软件中被修改的部分。
2. 从原基线测试用例库T中，排除所有不再适用的测试用例，确定那些对新的软件版本依然有效的测试用例，其结果是建立一个新的基线测试用例库T_0。
3. 依据一定的策略从T_0中选择测试用例测试被修改的软件。
4. 如果必要，生成新的测试用例集T_1，用于测试T_0无法充分测试的软件部分。
5. 用T_1执行修改后的软件。
6. 第(2)和第(3)步测试，验证修改是否破坏了现有的功能，第(4)和第(5)步测试验证修改工作本身。

回归测试的代价



We want Cx < Cy

Key is the test selection algorithm/technique

We want to maintain the same “quality of testing”

4 回归测试用例选择方法

- Test selection using execution trace and execution slice
- Test selection using test minimization
- Test selection using test prioritization

Test selection using execution trace and execution slice

Overview of a test selection method

- **Step 1**: Given P and test set T, find the execution trace of P for each test in T.

- **Step 2**: Extract test vectors from the execution traces for each node in the CFG of P

- **Step 3**: Construct syntax trees for each node in the CFGs of P and P'. This step can be executed while constructing the CFGs of P and P'.

- **Step 4**: Traverse the CFGs and determine the a subset of T appropriate for regression testing of P'.

Execution Trace [1]

- Let G=(N, E) denote the CFG of program P. N is a finite set of nodes and E a finite set of edges connecting the nodes. Suppose that nodes in N are numbered 1, 2, and so on and that Start and End are two special.

- Let Tno be the set of all valid tests for P'. Thus Tno contains only tests valid for P'. It is obtained by discarding all tests that have become obsolete for some reason.

Execution Trace [2]

➢**An execution trace of program P for some test t in Tno is the sequence of nodes in G traversed when P is executed against t. As an example, consider the following program.**

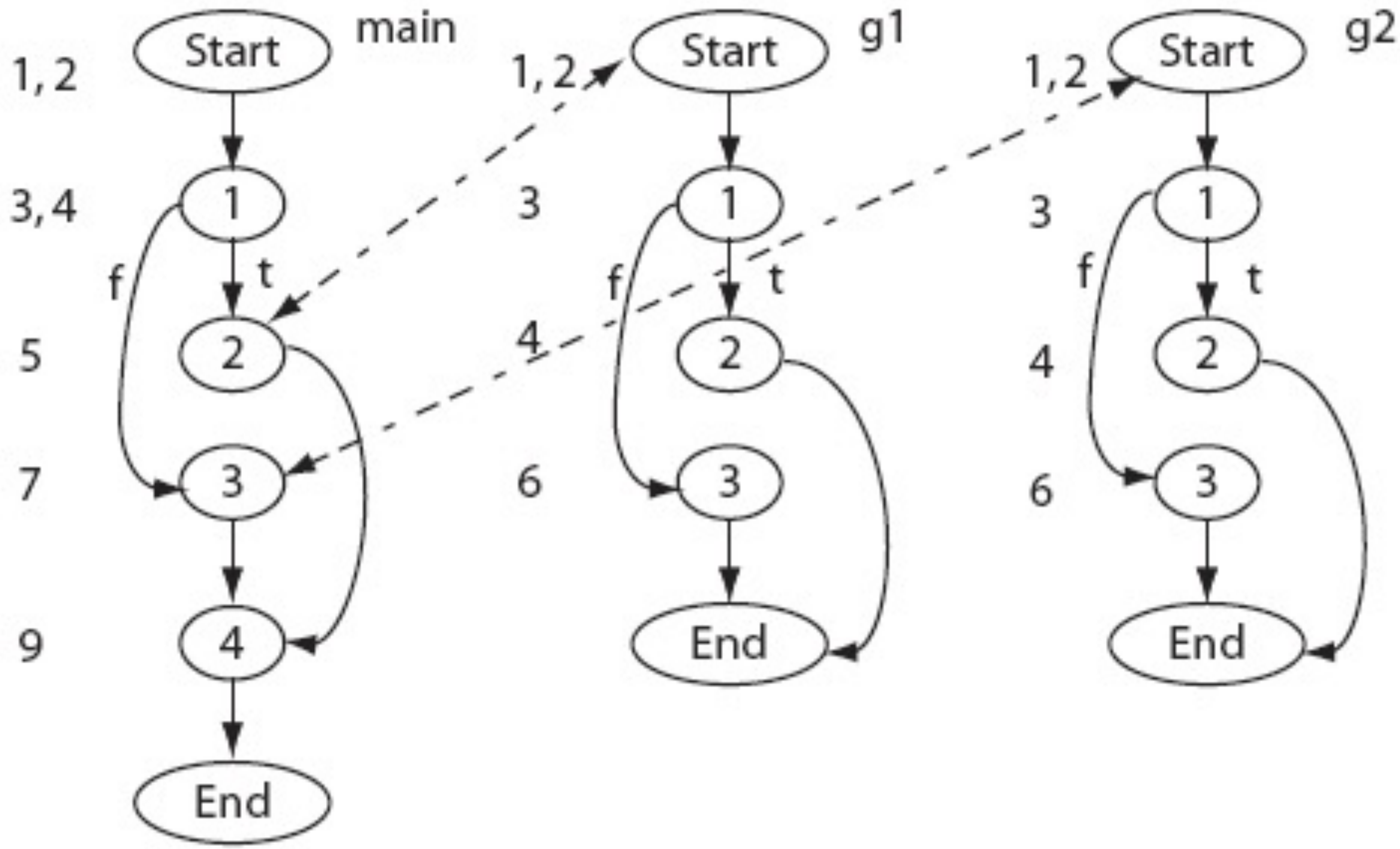
```
main(){
int x,y,p;
input (x,y);
if (x<y)
  p=g1(x,y);
else
  p=g2(x,y);
endif
output (p);
end
}
```

```
int g1(int a, b){
int a,b;
if(a+ 1==b)
  return(a*a);
else
  return(b*b);
}
```

```
int g2 (int a, b){
int a,b;
if(a==(b+1))
  return(b*b);
else
  return(a*a);
}
```

Execution Trace [3]

- Here is a CFG for our example program.



Execution Trace [4]

➢**Now consider the following set of three tests and the corresponding trace.**

$$T = \left\{ \begin{array}{l} t_1 :< x = 1, y = 3 > \\ t_2 :< x = 2, y = 1 > \\ t_3 :< x = 3, y = 4 > \end{array} \right\}$$

Test (t)	Execution trace ($trace(t)$)
t_1	main.Start, main.1, main.2, g1.Start, g1.1, g1.3, g1.End, main.2, main.4, main.End.
t_2	main.Start, main.1, main.3, g2.Start, g2.1, g2.2, g2.End, main.3, main.4, main.End.
t_3	main.Start, main.1, main.2, g1.Start, g1.1, g1.2, g1.End, main.2, main.4, main.End.

Test vector

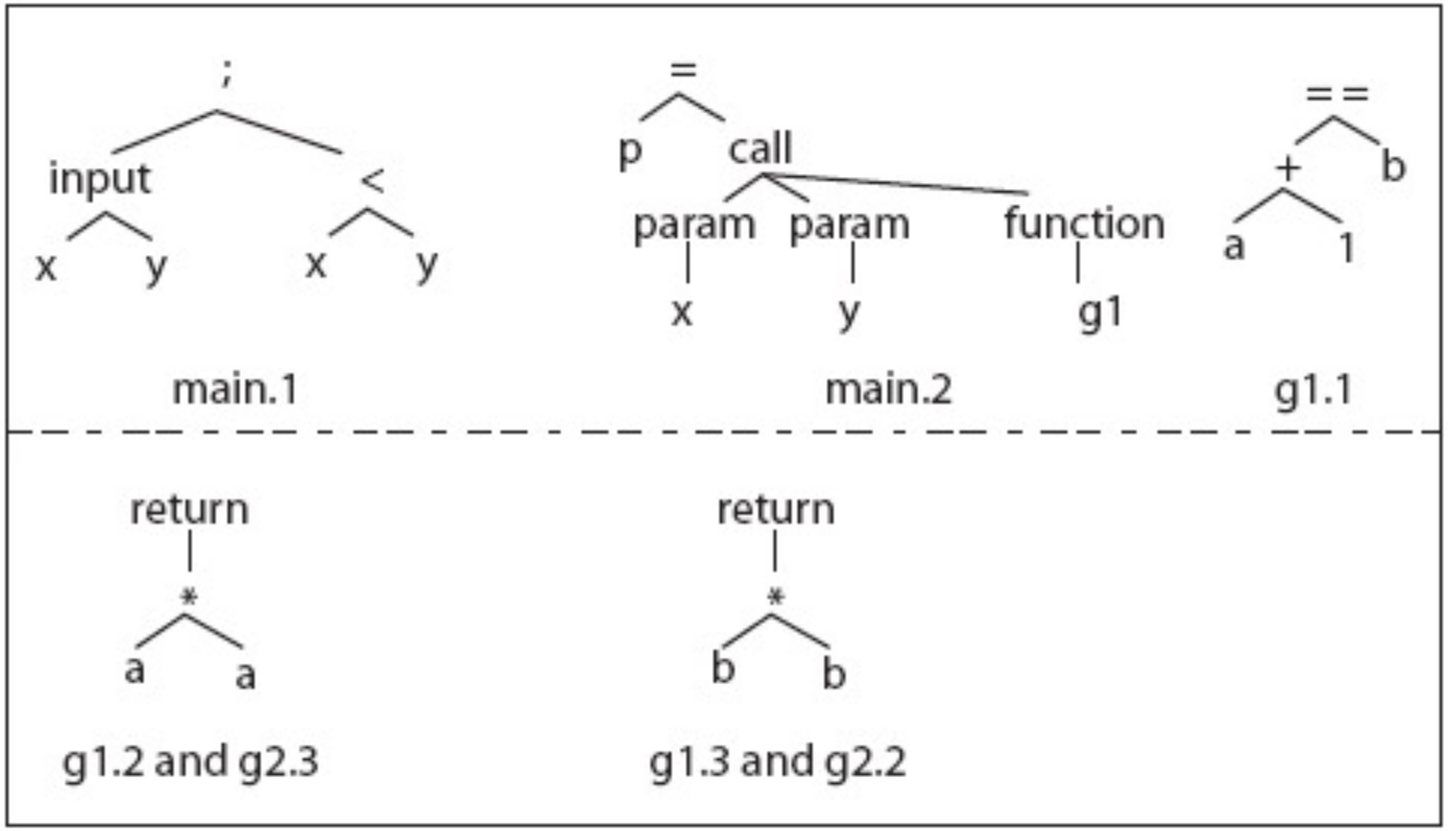
- A test vector for node n, denoted by test(n), is the set of tests that traverse node n in the CFG. For program P we obtain the following test vectors.

	Test vector (*test*(*n*)) for node *n*			
Function	1	2	3	4
main	t_1, t_2, t_3	t_1, t_3	t_2	t_1, t_2 t_3
g1	t_1, t_3	t_3	t_1	–
g2	t_2	t_2	None	–

Test (t)	Execution trace (*trace*(t))
t_1	main.Start, main.1, main.2, g1.Start, g1.1, g1.3, g1.End, main.2, main.4, main.End.
t_2	main.Start, main.1, main.3, g2.Start, g2.1, g2.2, g2.End, main.3, main.4, main.End.
t_3	main.Start, main.1, main.2, g1.Start, g1.1, g1.2, g1.End, main.2, main.4, main.End.

Syntax trees

➢A syntax tree is constructed for each node of CFG(P) and CFG(P'). Recall that each node represents a basic block. Here sample syntax trees for the example program.



Test selection [1]

➢**Given the execution traces and the CFGs for P and P′, the following three steps are executed to obtain a subset T′ of T for regression testing of P′.**

Step 1 Set $T' = \emptyset$. Unmark all nodes in G and in its child CFGs.

Step 2 Call procedure `SelectTests` (G. Start, G′.Start′), where G.Start and G′.Start′ are, respectively, the start nodes in G and G′.

Step 3 T' is the desired test set for regression testing P'.

Test selection [2]

- The basic idea underlying the SelectTests procedure is to traverse the two CFGs from their respective START nodes using a recursive descent procedure.

- The descent proceeds in parallel and the corresponding nodes are compared. If two nodes N in CFG(P) and N′ in CFG(P′) are found to be syntactically different, all tests in test (N) are added to T′.

Test selection example

- Suppose that function g1 in P is modified as follows.

```
int g1(int a, b){ ← Modified g1.
int a, b;
if(a-1==b) ← Predicate modified.
  return(a*a),
else
  return(b*b),
}
```

	Test vector ($test(n)$) for node n			
Function	1	2	3	4
main	t_1, t_2, t_3	t_1, t_3	t_2	t_1, t_2 t_3
g1	t_1, t_3	t_3	t_1	–
g2	t_2	t_2	None	–

- Try the SelectTests algorithm and check if you get T'={t1, t3}.

Issues with SelectTests

- **Think:**
- **What tests will be selected when only, say, one declarations is modified?**

Test selection using test minimization

Test minimization [1]

Test minimization is yet another method for selecting tests for regression testing.

To illustrate test minimization, suppose that P contains two functions, main and f. Now suppose that P is tested using test cases t1 and t2. During testing it was observed that t1 causes the execution of main but not of f and t2 does cause the execution of both main and f.

Test minimization [2]

Now suppose that P' is obtained from P by making some modification to f.

Which of the two test cases should be included in the regression test suite?

Obviously there is no need to execute P' against t1 as it does not cause the execution of f. Thus the regression test suite consists of only t2.

In this example we have used function coverage to minimize a test suite {t1, t2} to a obtain the regression test suite {t2}.

Test minimization [3]

Test minimization is based on the coverage of testable entities in P.

Testable entities include, for example, program statements, decisions, def-use chains, etc.

One uses the following procedure to minimize a test set based on a selected testable entity.

A procedure for test minimization

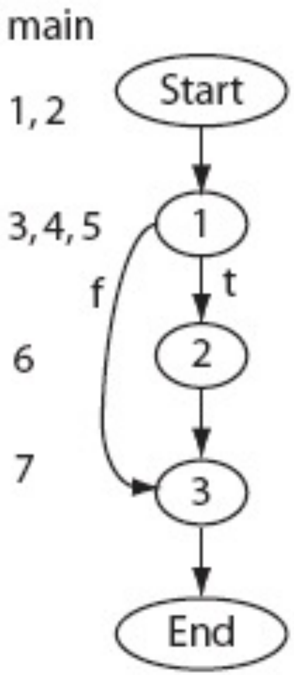
Step 1: Identify the type of testable entity to be used for test minimization. Let e1, e2, ..ek be the k testable entities of type TE present in P. In our previous example TE is function.

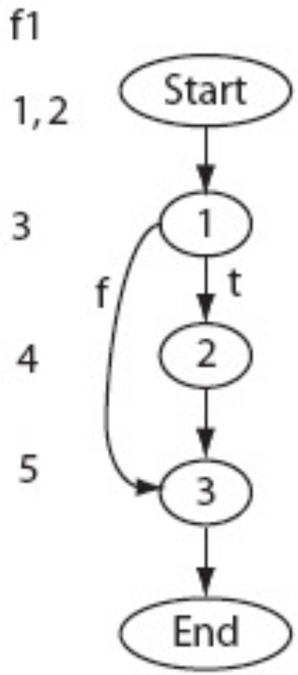
Step 2: Execute P against all elements of test set T and for each test t in T determine which of the k testable entities is covered.

Step 3: Find a minimal subset T' of T such that each testable entity is covered by at least one test in T'.

Test minimization: Example

Step 1: Let the basic block be the testable entity of interest. The basic blocks for a sample program are shown here for both main and function f1.





Step 2: Suppose the coverage of the basic blocks when executed against three tests is as follows:

- **t1: main: 1, 2, 3. f1: 1, 3**
- **t2: main: 1, 3. f1: 1, 3**
- **t3: main: 1, 3. f1: 1, 2, 3**

Step3: A minimal test set for regression testing is {t1, t3}.

Test selection using test prioritization

Test prioritization

- Note that test minimization will likely discard test cases. There is a small chance that if P′ were executed against a discarded test case it would reveal an error in the modification made.
- When very high quality software is desired, it might not be wise to discard test cases as in test minimization. In such cases one uses test prioritization.
- Tests are prioritized based on some criteria. For example, tests that cover the maximum number of a selected testable entity could be given the highest priority, the one with the next highest coverage m the next higher priority and so on.

A procedure for test prioritization

Step 1: Identify the type of testable entity to be used for test minimization. Let e1, e2, ..ek be the k testable entities of type TE present in P. In our previous example TE is function.

Step 2: Execute P against all elements of test set T and for each test t in T. For each t in T compute the number of distinct testable entities covered.

Step 3: Arrange the tests in T in the order of their respective coverage. Test with the maximum coverage gets the highest priority and so on.

Using test prioritization

Once the tests are prioritized one has the option of using all tests for regression testing or a subset.

The choice is guided by several factors such as the resources available for regression testing and the desired product quality.

In any case test are discarded only after careful consideration that does not depend only on the coverage criteria used.

Tools for regression testing

Methods for test selection described here require the use of an automated tool for all but trivial programs.

xSuds from Telcordia Technologies can be used for C programs to minimize and prioritize tests.

Many commercial tools for regression testing simply run the tests automatically; they do not use any of the algorithms described here for test selection. Instead they rely on the tester for test selection. Such tool are especially useful when all tests are to be rerun.

Tools for regression testing

TestingWhiz是一款无需编码即可使用的回归测试自动化工具，专门面向Web、移动及云应用，且提供超过290种预定义测试命令以实现测试用例的编写与编辑。

Sahi是一款开源工具，但Sahi Pro则属于面向Web应用的商用测试自动化工具。Sahi Pro能够管理大型测试套件的回归测试自动化事务。

TestComplete是一套来自Smartbear公司的平台，适用于桌面、Web以及移动测试工具。它能够实现功能与回归测试自动化，并支持由JavaScript、C++ Script、C# Script、VB Script、Python、Jscript以及DelphiScript等编写而成的测试。

Silk Test是一款由Borland推出的自动化测试工具，旨在执行功能与回归测试。它基于类似于C++的面向对象编程(简称OOP)语言，其中包含对象、类与继承等概念。

Summary

- Regression testing is an essential phase of software product development.
- In a situation where test resources are limited and deadlines are to be met, execution of all tests might not be feasible.
- In such situations one can make use of sophisticated technique for selecting a subset of all tests and hence reduce the time for regression testing.