第 3 章　软件测试过程

学习要点：

- ❖ 单元测试。
- ❖ 集成测试。
- ❖ 回归测试。
- ❖ 确认测试。
- ❖ α 测试与 β 测试。
- ❖ 系统测试。

软件测试过程分成 4 个步骤：单元测试、集成测试、确认测试和系统测试，每一步是前一步的继续，其顺序关系如图 3-1 所示。在软件系统底层进行的测试为单元测试或模块测试，经过单元测试，底层软件缺陷被找出并修复之后，就组成在一起，对模块的组合进行集成测试，这个不断增加的测试过程继续进行，加入越来越多的软件片段，直至整个软件。

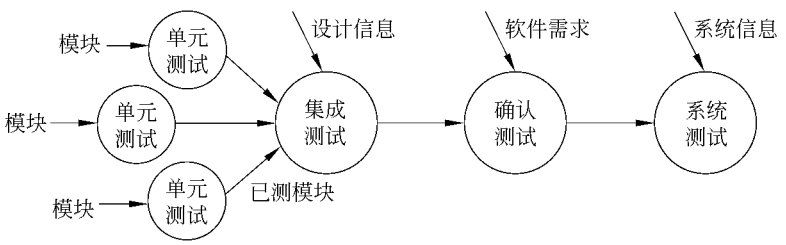

图 3-1　软件测试步骤

3.1　单 元 测 试

单元测试又称为模块测试、逻辑测试、结构测试，单元测试是对软件基本组成模块和主要的控制路径进行的测试，进而发现模块内部的错误。有关测试复杂度和错误发现是受单元测试的约束范围限定的。单元测试采用白盒测试方法，而且可以多个单元并行进行测试。

3.1.1　单元测试内容

设计单元测试的测试用例时主要考虑的问题如图 3-2 所示。

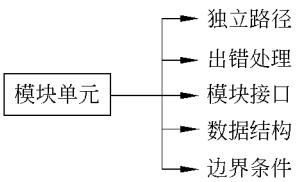

图 3-2　设计单元测试用例时要考虑的问题

1. 模块接口测试

模块接口测试检查进出程序单元的数据流是否正确。对模块接口数据流的测试必须在任何其他测试之前进行，如果不能确保数据正确地输入和输出，所有的测试都将无法进行，测试主要内容如下。

(1) 形式参数与实际参数是否一致。

(2) 模块接收输入参数与模块变元是否一致。

(3) 模块调用内部函数变元个数、属性、次序是否一致。

(4) 文件的属性是否正确。

(5) 全程变量在各模块间的定义是否一致等。

2. 数据结构测试

在模块工作过程中，必须测试其内部的数据能否在算法执行的整个过程中保持完整性，包括内部数据的内容、形式及相互关系保证不发生错误。应该说模块的局部数据结构是经常发生错误的错误源，所以应该在单元测试中注意对局部数据结构的测试，测试内容如下。

(1) 变量从来没有被使用。

(2) 变量没有初始化。

(3) 错误的类型转换。

(4) 数组越界。

(5) 非法指针。

(6) 变量或函数名称拼写错误。

3. 边界条件测试

边界条件测试可保证模块在极限的情形下能够正确执行，测试内容如下。

(1) 普通合法数据是否能正确处理。

(2) 普通非法数据是否能正确处理。

(3) 边界内最接近边界的(合法)数据是否能正确处理。

(4) 边界外最接近边界的(非法)数据是否能正确处理。

4. 独立路径测试

在单元测试中，最主要的是路径测试。在控制结构中的所有独立路径(基本路径)都要走遍，进而保证在一个模块中的所有语句都能执行至少一次。在单元测试过程中，对执行路径的选择性测试是最主要的任务。测试用例必须能够发现由于计算错误、不正确的判定或不正常的控制流而产生的错误，测试内容如下。

(1) 死代码。

(2) 错误的计算优先级。

(3) 精度错误(比较运算错误、赋值错误)。

(4) 表达式的不正确符号(>、>=;=、==、!=)。

(5) 循环变量的使用错误(错误赋值)。

5. 出错处理测试

完善的模块设计要求能预见出错的条件,并设置适当的出错处理,以便在程序出错时,能对出错程序重新调整,保证其逻辑上的正确性。这种出错处理也是模块功能的一部分,测试内容如下。

(1) 检查错误是否出现,重点检查以下时刻。

① 资源使用前后。

② 其他模块使用前后。

(2) 出现错误,是否进行了错误处理,主要有以下几种处理方式。

① 抛出错误。

② 通知用户。

③ 进行记录。

(3) 错误处理是否有效,主要有以下两种判定方法。

① 在系统干预前处理。

② 报告和记录真实而详细描述了错误。

3.1.2 单元测试规则

单元测试与编码测试属于同一步骤。在代码开发、评审和语法正确性验证之后,单元测试用例设计就已开始。因为一个单元本身不是一个单独的程序,所以必须为每个单元测试开发驱动模块或/和桩模块软件。在大多数应用中,一个驱动模块只用于接收测试数据并把数据传送给要测试的单元。桩模块的功能是替代那些属于被测试单元的被调用模块。一个桩模块或空子程序使用从属模块的接口,可以做少量的数据操作,接收被测试模块的调用和输出数据,是被测试模块要调用的模块。

驱动模块和桩模块都是额外的开销,都属于必须开发但又不能和最终软件一起提交的软件,如果驱动模块和桩模块很简单的话,那么额外开销相对来说很低。但是,许多模块使用简单的额外软件不能进行足够的单元测试。在这些情况下,完整的测试要推迟到集成测试步骤时再完成。

当一个单元被设计为高内聚时,测试被简化。当一个单元只表示一个函数时,测试用例的数量就会降低,而且错误也就更容易被预测和发现,如图 3-3 所示。

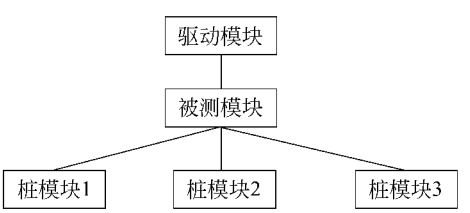

图 3-3 单元测试规则

1. 单元测试的步骤

(1) 配置测试环境，设计辅助测试模块、驱动模块和桩模块。

(2) 编写测试数据，根据单元测试要解决的问题设计测试用例。

(3) 可以进行多个单元的并行测试。

通过编译以后，首先做静态代码的复审。参加者为软件设计者、程序编写者和程序测试者，其中有软件设计能力很强的高级程序员任组长。在研究软件设计文档基础上召开审查会，分析程序逻辑与错误清单，经测试预演、人工测试、代码复审后再进入计算机代码执行活动的动态测试，再做单元测试报告。

为了提高测试效率，克服无法穷尽测试的困难，在单元测试中应采用白盒法与黑盒法相结合、静态测试与动态测试相结合、人工测试与机器测试相结合的策略。

2. 动态单元测试步骤

(1) 用边值分析方法设计测试集，测试边界易出错之处。

(2) 用等价类划分方法设计测试集，测试主要的软件错误。

(3) 结合用人工测试的错误推测法设计测试集做补充。

(4) 用逻辑覆盖设计测试集做补充。

3.1.3　单元测试的问题

1. 单元测试的重要性

(1) 覆盖了最小代码单元。

(2) 支持组包测试。

(3) 执行率是 100%。

(4) 可以随时执行，并且覆盖变动后的代码范围，以确定变动后的代码对原代码功能未作修改。

(5) 提升软件系统整体信赖度。

(6) 并不仅仅体现在测试覆盖上，还包含对可能出现问题的代码进行彻底排查。

(7) 支持先测试后编码的行为。

(8) 支持变化，因为任何变化所导致的失败情况都会被立刻反映出来。

(9) 便于后期维护，因为测试更能准确反映原代码设计人员的思维。

2. 单元测试的误区

1) 集成测试将找出所有的 Bug

由于规模大的代码集成存在较高的复杂性。如果软件单元没有进行测试，开发人员很可能花费大量的时间仅仅是为了使软件能够运行，而任何实际的测试方案都无法执行。一旦软件可以运行了，开发人员又要面对这样的问题：在考虑软件全局复杂性的前提下对每个单元进行全面的测试。这是一件非常困难的事情，甚至在创造一种单元调用的测试条件的时候，要全面地考虑单元被调用时的各种入口参数。在软件集成阶段，对单元功能进行全面测试的复杂程度远远超过独立进行的单元测试过程。

最后的结果是测试将无法达到它所应该有的全面性。一些缺陷将被遗漏，并且很多

Bug 将被忽略。

2）单元测试浪费了过多时间

编码一旦完成，很多人迫不及待地进行软件集成工作，以便及早看到运行结果。此时进行单元测试，错误地认为单元测试浪费时间，进而成为障碍。

在开发过程中，难免会有代码错误，而使得软件无法运行或者无法正常运行，修改错误成为软件开发新增的额外工期，而且当这个系统投入使用时也无法确保它能够可靠地运行。而进行完整的单元测试和编写实际的代码所花费的精力大致相同。一旦完成了单元测试，则可以进行更高效的集成工作，所以单元测试是对项目可用时间更高效的利用。

3）可以不进行单元测试

许多程序员认为自己开发的软件能直接正常运行，测试是浪费时间。而忽视了编码不是一个可以一次性通过的过程，人们所认为的正确性局限在人们的思维中，或者局限在某个时间段中，当这种正确性因为现实环境的变化被否定时，错误就产生了，所以单元测试是软件测试中必不可少的一步。

3.2 集成测试

如果每一个模块都能正常工作，但把它们集成到一起不能正常工作，这主要就是由于模块相互调用时的接口处引入许多新问题。例如，数据经过接口时可能丢失；一个模块对另一模块可能造成不应有的影响；几个子功能组合起来不能实现主功能；误差不断积累达到不可接受的程度；全局数据结构出现错误，等等。集成测试是通过测试发现与接口有关的问题来构造程序的系统化技术，其目标是利用通过了单元测试的模块，构造设计中所描述的程序结构。

非增量集成是指把所有模块全部集成起来，进行整体测试。然而，这样做的结果是测试时可能发现一大堆错误，为每个错误定位和纠正非常困难，并且在改正一个错误的同时又可能引入新的错误，新旧错误混杂，更难断定出错的原因和位置。

增量式集成是与之相反的方法，程序段段地扩展，测试的范围步步地增大，错误易于定位和纠正，界面的测试可做到完全彻底。在集成测试过程中，首先把已经通过单元测试的模块组装起来，构成一个在设计阶段所定义的程序结构，然后通过集成测试发现与接口有关的问题。

1. 集成测试的重要性

（1）尽管开发时尽力做到仔细，但是并不能保证接口不出错。也就是说虽然可以通过单元测试验证单个或者几个模块之间工作正常，但是并不能保证所有模块组合起来就能正常工作。

（2）在单元测试中用了很多桩模块，但并不能保证这些模块在实际的系统中的真实性。

（3）所有模块出错的概率不同，集成测试中可以找出那些容易使别的模块出错的模块。

（4）能尽早发现错误。

（5）报告错误后，可同时进行 Bug 消除和开发工作。

2. 集成测试的主要过程

(1) 构建的确认过程。

(2) 补丁的确认过程。

(3) 测试组提交过程。

(4) 测试用例设计过程。

(5) 测试代码编写过程。

(6) Bug 的报告过程。

(7) 每周/每两周的构建过程。

(8) 点对点的测试过程。

(9) 组内培训过程。

集成测试的方案有很多种，如自顶向下集成测试、自底向上集成测试、混合式集成测试、核心系统先行集成测试、高频集成测试和回归测试等。

3.2.1　自顶向下集成测试

自顶向下集成是构造程序结构的一种增量式方式，它从主控模块开始，按照软件的控制层次结构，以深度优先或宽度优先的策略，逐步把各个模块集成在一起。深度优先策略首先把主控制路径上的模块集成在一起，至于选择哪一条路径作为主控制路径，一般根据问题的特性确定。以图 3-4 为例，若选择了最左一条路径，首先将模块 M1、M2、M5 集成在一起，再将 M5 或 M6 集成起来，然后考虑中间和右边的路径。宽度优先策略则不然，它沿控制层次结构水平地向下移动。以图 3-4 为例，它首先把 M2、M3 和 M4 与主控模块集成在一起，再将 M5 和 M6 集成起来，然后继续。

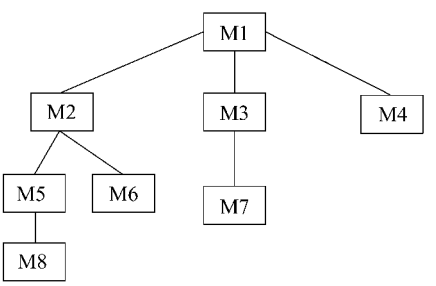

图 3-4　自顶向下集成测试

自顶向下集成测试的步骤如下。

(1) 以主控模块作为测试驱动程序，把对主控模块进行单元测试时引入的所有桩模块用从属于主控模块的模块来替代。

(2) 依据所选的深度优先或宽度优先集成策略，每次只替代一个桩模块。

(3) 每集成一个模块立即测试一遍。

(4) 每完成一次测试，接下来就有一个桩模块被一个真正的模块所替代。

(5) 为避免引入新错误，可以不断地进行回归测试，即全部或部分地重复已做过的测试。

从(2)步开始，循环执行上述步骤，直至整个程序构造完毕。

在图 3-4 中，实线表示已部分完成的结构，若采用深度优先策略，下一步将用模块 M7 替换桩模块 S7，当然 M7 本身可能又带有桩模块，随后将被对应的实际模块替代。

自顶向下集成测试的优点在于能尽早地对程序的主要控制和决策机制进行检验，因此较早地发现错误。缺点是在测试较高层模块时，低层处理采用桩模块替代，不能反映真

实情况，重要数据不能及时回送到上层模块，因此测试并不充分。解决这个问题有下述几种办法。

(1) 把某些测试推迟到用真实模块替代桩模块之后进行。

(2) 开发能模拟真实模块的桩模块。

(3) 自底向上集成模块。

方法(1)又回退为非增量式的集成方法，使错误难于定位和纠正，并且失去了在组装模块时进行一些特定测试的可能性；方法(2)要大大增加开销；方法(3)比较切实可行，下面介绍自底向上集成测试。

3.2.2 自底向上集成测试

自底向上测试是从最低层模块开始组装测试，因测试到较高层模块时，所需的下层模块功能均已具备，所以不再需要桩模块。自底向上集成测试的步骤如下。

(1) 把低层模块组织成实现某个特定子功能的模块群。

(2) 开发一个测试驱动模块，控制测试数据的输入和测试结果的输出。

(3) 对每个模块群进行测试。

(4) 删除测试使用的驱动模块，用较高层模块把模块群组织成为完成更大功能的新模块群。

从(1)步开始循环执行上述各步骤，直至整个程序构造完毕。

如图 3-5 所示，首先将最低层模块分为 3 个模块群，每个模块群引入一个驱动模块进行测试。因模块群 1、模块群 2 中的模块均属于模块 Ma，因此在驱动模块 D1、D2 去掉后，模块群 1 与模块群 2 直接与 Ma 相连。以此类推，模块群 3 属于模块 Mb，把驱动模块 D3 去掉后，Mb 与模块群 3 直接连接，最后 Ma、Mb 和 Mc 全部集成在一起进行测试。

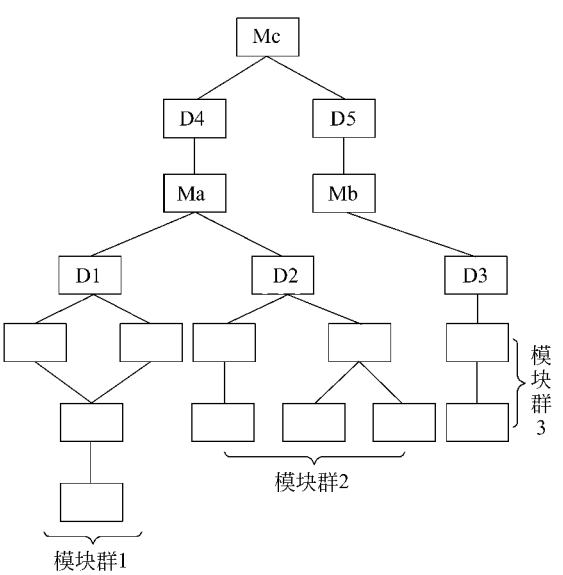

图 3-5 自底向上集成测试

自底向上集成测试的方法是最常用的测试方法。自底向上集成测试的优点是：不用桩模块，管理方便、测试人员能较好地锁定软件故障所在位置，测试用例的设计相对简单，但缺点是程序最后一个模块加入时才具有整体形象，对于某些开发模式不适用，如使用XP开发方法，它会要求测试人员在全部软件单元实现之前完成核心软件部件的集成测试。它与自顶向下集成测试方法优缺点正好相反。因此，在测试软件系统时，应根据软件的特点和工程的进度，选用适当的测试策略，有时混合使用两种策略更为有效，上层模块用自顶向下的方法，下层模块用自底向上的方法。

在集成测试中要注意关键模块，关键模块一般都具有下述特征。

(1) 对应几条需求。

(2) 具有高层控制功能。

(3) 复杂、易出错。

(4) 有特殊的性能要求。

关键模块应尽早测试，并反复进行回归测试。

3.2.3　混合式集成测试

混合式集成测试是自顶向下集成测试与自底向上集成测试的折中测试方法，结合了两者的优点。混合式集成测试把系统划分成3个层次，中间一层作为目标层。测试的时候，对目标层的上一层使用自顶向下集成策略，对目标层下一层使用自底向上的集成策略，最后测试都结合到目标层。

如图3-6所示，其中目标层为B、C、D。目标层上面一层是A，目标层下面一层是E、F、G。其具体步骤如下。

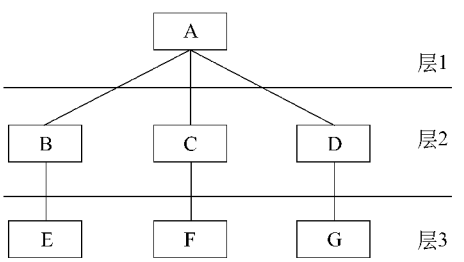

图3-6　混合式集成测试

(1) 对目标层的上一层使用自顶向下的集成策略，因此测试A，使用桩模块来代替B、C、D。

(2) 对目标层的下一层使用自底向上的集成策略，因此测试E、F、G，使用驱动模块来代替B、C、D。

(3) 把目标层的下面一层与目标层集成，因此测试(B,E)、(C,F)、(D,G)，使用驱动模块来代替A。

(4) 把三层都集成起来，因此测试(A,B,C,D,E,F,G)。

在进行混合式集成测试时，要尽量减少驱动模块和桩模块的数量。因此，在上面集成时先使用目标层与下面的一层结合来进行测试，而不是用与上面的一层结合来进行测试，因为先与上面集成时增加了下面桩模块的设计。

3.2.4　先行集成测试

先行集成测试法的思想是先对核心软件部件进行集成测试，在测试通过的基础上再

按各外围软件部件的重要程度逐个集成到核心系统中。每次加入一个外围软件部件都产生一个产品基线，直至最后形成稳定的软件产品。核心系统先行集成测试法对应的集成过程是一个逐渐趋于闭合的螺旋形曲线，代表产品逐步定型的过程。

核心系统先行集成测试的步骤如下。

(1) 对核心系统中的每个模块进行单独的、充分的测试，必要时使用驱动模块和桩模块。

(2) 对于核心系统中的所有模块一次性集合到被测系统中，解决集成中出现的各类问题。在核心系统规模相对较大的情况下，也可以按照自底向上的步骤，集成核心系统的各组成模块。

(3) 按照各外围软件部件的重要程度以及模块间的相互制约关系，拟定外围软件部件集成到核心系统中的顺序方案。方案经评审以后，即可进行外围软件部件的集成。

(4) 在外围软件部件添加到核心系统以前，外围软件部件应先完成内部的模块级集成测试。

(5) 按顺序不断加入外围软件部件，排除外围软件部件集成中出现的问题，形成最终的用户系统。

该集成测试方法对于快速软件开发很有效果，适合较复杂系统的集成测试，能保证一些重要的功能和服务的实现。缺点是采用此法的系统一般应能明确区分核心软件部件和外围软件部件，核心软件部件应具有较高的耦合度，外围软件部件内部也应具有较高的耦合度，但各外围软件部件之间应具有较低的耦合度。

3.2.5 高频集成测试

高频集成测试是指同步于软件开发过程，每隔一段时间对开发团队的现有代码进行一次集成测试。如某些自动化集成测试工具能实现每日深夜对开发团队的现有代码进行一次集成测试，然后将测试结果发到各开发人员的电子邮箱中。该集成测试方法频繁地将新代码加入到一个已经稳定的基线中，以免集成故障难以发现，同时控制可能出现的基线偏差。使用高频集成测试需要具备一定的条件：可以持续获得一个稳定的增量，并且该增量内部已被验证没有问题；大部分有意义的功能增加可以在一个相对稳定的时间间隔(如每个工作日)内获得；测试包和代码的开发工作必须是并行进行的，并且需要版本控制工具来保证始终维护的是测试脚本和代码的最新版本；必须借助于使用自动化工具来完成。高频集成测试一个显著的特点就是集成次数频繁，显然，人工的方法是不能胜任的。高频集成测试步骤如下。

(1) 选择集成测试自动化工具。如很多 Java 项目采用 JUnit＋Ant 方案来实现集成测试的自动化，也有一些商业集成测试工具可供选择。

(2) 设置版本控制工具，以确保集成测试自动化工具所获得的版本是最新版本。如使用 CVS 进行版本控制。

(3) 测试人员和开发人员负责编写对应程序代码的测试脚本。

(4) 设置自动化集成测试工具，每隔一段时间对配置管理库新添加的代码进行自动化的集成测试，并将测试报告汇报给开发人员和测试人员。

(5) 测试人员监督代码开发人员及时关闭不合格项。

按照(3)至(5)不断循环，直至形成最终软件产品。

高频集成测试能在开发过程中及时发现代码错误，能直观地看到开发团队的有效工程进度。高频集成测试中，开发维护源代码与开发维护软件测试包被赋予了同等的重要性，这对有效防止错误、及时纠正错误都很有帮助。它的缺点在于测试包有时候可能不能暴露深层次的编码错误和图形界面错误。

3.2.6　回归测试

在集成测试中，每当一个新的模块加进来的时候，软件就发生了改变。新的数据流路径被建立，新的操作可能也会出现，还有可能激活了新的控制逻辑。这些改变可能会使原来工作很正常的功能产生错误。在集成测试策略的环境中，回归测试是对某些已经进行过的测试的子集的重新执行，以保证上述改变不会有副作用。

成功测试的结果是发现错误，而错误是要被修改的，每当软件被修改的时候，软件配置的某些方面也被修改了，回归测试就是保证改动不会带来不可预料的行为或者附加的错误。

回归测试可以通过重新执行所有的测试用例的一个子集人工地进行，也可以使用自动化的捕获/回放工具来进行。

回归测试集包括下述 3 种类型的测试用例。

(1) 能够测试软件的所有功能的代表性测试用例。

(2) 专门针对可能会被修改影响的软件功能的附加测试。

(3) 注重于修改过的软件构件的测试。

在集成测试进行的过程中，回归测试可能会变得非常庞大。因此，回归测试应只包括涉及在主要的软件功能中出现的一个或多个错误类的那些测试。每进行一个修改时，就对每一个程序功能都重新执行所有的测试是不实际的，而且效率也低。

3.3　确认测试

集成测试完成以后，分散开发的模块被连接起来，构成完整的程序。各模块之间的接口存在的问题都已消除，于是测试工作进入确认测试阶段。确认测试主要由两部分组成：有效性测试和配置复审。

有效性测试主要检查软件是否达到了系统设计要求，是否满足软件需求说明书中的确认标准，若能达到，则认为开发的软件是合格的。有效性测试主要是通过一系列黑盒测试来进行的，其测试用例来自于需求分析阶段。

确认测试的另一个重要部分是配置复审，复审的目的在于保证软件配置齐全、分类有序，并且包括软件维护所必需的细节。

确认测试的步骤如下。

(1) 在模拟的环境中进行强度测试，在事先规定时期内运行所有软件功能，发现软件