测试简介

概念

目的

测试的目的在于：

- 核实对象之间的交互。
- 核实软件的所有构件是否正确集成。
- 核实所有需求是否已经正确实施。
- 确定缺陷并确保在部署软件之前将缺陷解决。

在很多组织中，软件测试占软件开发费用的 30% 到 50%。但大多数人仍然认为软件在交付之前没有进行充分的测试。这一矛盾根植于两个明显的事实。第一个，测试软件十分困难。给定程序具有无数的不同行为方式。第二个，测试通常是在没有明确的方法，不采用必须的自动化手段和工具支持的情况下进行的。由于软件的复杂性，无法实现完全测试，但采用周密的方法和最新技术水平的工具可以明显提高软件测试的生产率和有效性。

对于失败将导致人员伤亡这类“安全至上”的系统（如空中交通管制系统、导弹制导系统、或医用输送系统）来说，高质量的软件是系统成功的要素。对于典型的 MIS 系统，上述情况不是非常明显，但是消除缺陷造成的影响将需要相当昂贵的开支。

在软件生命周期的早期启动的执行良好的测试，将明显降低完成和维护软件的开支。它还可以大大降低与部署质量低劣的软件相关的责任或风险，如用户的生产率低下、数据输入和计算错误，以及令人无法接受的功能行为。现在，许多 MIS 系统是“任务至上”的，也就是说当出现失败时，公司将无法正常运转并导致大量损失。例如：银行或运输公司。测试任务至上的系统时，必须使用安全至上的系统所采用的类似严格方法。

质量的定义

质量的定义（取自 The American Heritage Dictionary of the English Language, 3rd Edition, Houghton Mifflin Co., c 1992, 1996）为：

> **Quality** (kwol'i-te) n., pl.**-ties**. Abbr. **qlty**. **1.a.**An inherent or distinguishing characteristic; a property. **b.**A personal trait, especially a character trait.**2.** Essential character; nature.**3.a.** Superiority of kind. **b.**Degree or grade of excellence.

如定义所述，质量不是单方面的概念，而是多方面的概念。要利用该定义并将它应用到软件开发中，这个定义必须改进。因此，考虑到要在 Rational Unified Process 中使用，质量被定义为：

> “由以下三点所确定的特征：
> 满足或超出认定的一组需求，并
> 使用经过认可的评测方法和标准来评估，还
> 使用认定的流程来生产。”

因此，质量达标不是简单地“满足需求”或生产出满足用户需要或期望的产品。更确切地说，质量还包含确定证明质量达标所使用的评测方法和标准，以及如何实施流程，以确保由此流程生产的产品已达到预期的质量水平（而且能够管理该流程并重复使用）。

产品质量

产品质量是正在由流程生产的产品的质量。在软件开发中，产品是许多工件的聚合关系，其中包括：

已部署的可执行代码（应用程序、系统等），这可能是最显而易见的工件，因为项目通常是由于该工件才存在的。也就是说，它是为客户（最终用户、股东、涉众等）提供价值的首要产品。

已部署的不可执行工件，包括用户手册和教程资料等工件。

未部署的可执行工件，如工件的实施集，包括已创建用于支持实施的测试脚本和开发工具。

未部署的不可执行工件，如实施计划、测试计划和各种模型。

由于很多工件都建立在其他工件的基础上，所以在某种程度上，所有工件的质量都是相关的。因此，对每个工件的质量都应该评测和评估。

1．可执行工件的产品质量（部署的和未部署的）：

可执行工件是通过其需求来描述的，并表述为用例或补充需求（如销售、性能等）。要评测并达到质量要求，必须了解这些需求并以清楚、简明和可核实（可测试）的方式陈述这些需求。对于软件来说，测试角色不会将所有需求当作测试对象（如市场渗透或销售收益）。对于那些将成为测试对象的需求来说，测试设计员必须能够指定一种方法来核实是否满足需求（正如已指定的）、没有偏离既定意图并且没有缺陷。

产品质量是通过评测以下质量维度和评测产品是否满足这些维度的要求来决定的：

可靠性：已部署的代码在执行过程中的防故障（崩溃、挂起、内存丢失等）能力。

功能：已部署的代码按既定意图执行所需的用例。

性能：在实际的操作特征（如负载、强度和长时间运行）条件下，已部署的代码以及时和可接受的方式执行和响应，并以可接受的方式继续运行。

对于每一维度，在测试的一个或多个不同阶段，应该实施和执行一种或多种测试类型。

此外，还可通过评测每一工件新版本的可执行工件中所作的变更数量和类型来评估产品质量。

2．不可执行工件的产品质量（已部署的或未部署的）：

不可执行工件的产品质量通过工件的目的、目标和结构来描述，并通过确保工件符合以下各项要求来评估：

工件内部和工件之间的一致性（语言的使用、术语或语义等）。

指南、标准和合同需求（语言的使用、术语、语义、格式或内容等）的兼容性

此外，还可以通过工件版本之间所作变更的数量和类型来评估不可执行工件的产品质量。

为了帮助评估 RUP 中工件的产品质量,我们在 RUP 中包括了以下针对大多数工件的信息类型：

工件指南和检查点：有关如何开发、评估和使用工件的信息。

模板：工件的“模型”或原型，为内容提供结构和指导。

质量纬度

正如在最佳方案：检验质量中讨论的那样，质量不是一个可以简单描述的概念。同样，当我们转而关注于鉴别质量的测试时，也无法从单一的角度来描述质量的概念或评测质量的方法。在 Rational Unified Process 中，这一问题是通过指定“质量”具备以下维度来解决的：

可靠性：软件坚固性和可靠性（防故障能力，如防止崩溃、内存丢失等能力）、资源利用率和代码完整性以及结构（语言和语法的技术兼容性）。

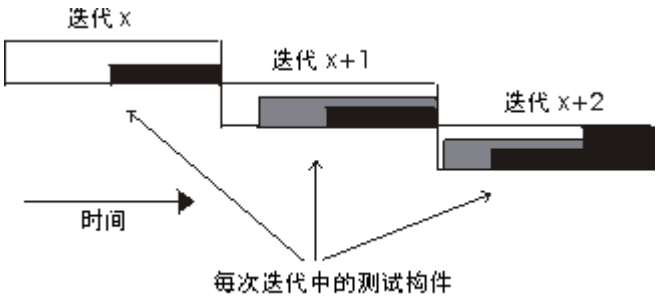
功能：按照既定意图和要求，执行指定用例的能力。

性能：测试对象的计时配置文件和操作特征。计时配置文件包括代码的执行流、数据访问、函数调用和系统调用。性能的操作特征包括与作业负载相关的特征，如响应时间、操作可靠性(MTTF)； 以及与操作限制相关的特征，如负载容量或强度。

对于各维度的元素，在测试的不同阶段，应该实施和执行一种或多种测试类型

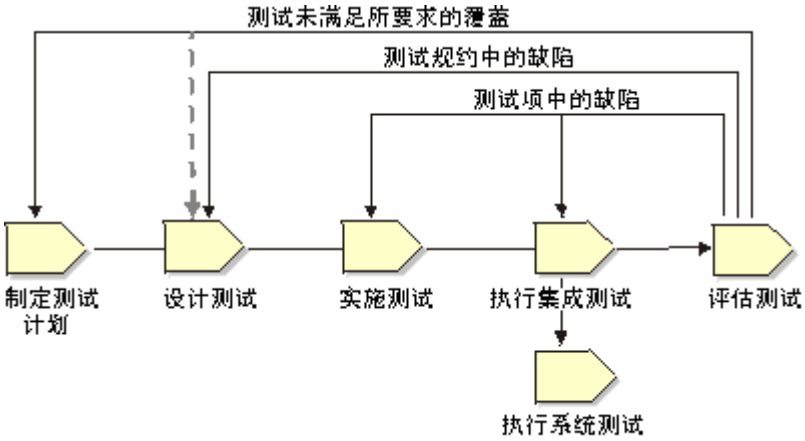
测试的生命周期

在软件开发生命周期中，软件是通过迭代来不断加以完善的。在这种环境中，对于每个作为测试目标的工作版本，测试的生命周期还都必须具有一种迭代方法。对于针对每个工作版本执行的测试，都作出了增补和改进，并累积为一个测试体，用于后续阶段的回归测试。该方法表明它将导致在整个流程中重复进行测试，就象修订软件本身一样。这里没有一成不变的软件规约，也没有一成不变的测试。



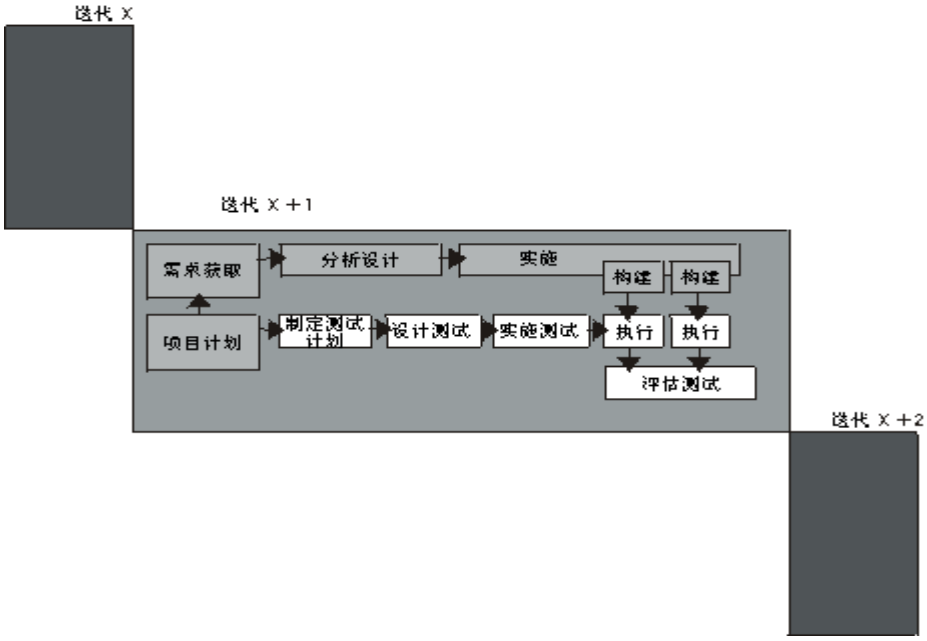
该迭代方法非常注重回归测试。迭代 X 中的大多数测试在迭代 X+1 中都用作回归测试。在迭代 X+2 中，将使用迭代 X 和迭代 X+1 中的大多数测试作为回归测试，后续迭代中采用的原则与此相同。因为相同的测试要重复多次，所以投入一些精力将测试自动化将会获益良多。此外，也有必要有效地自动执行测试，来满足完工期限的要求。

在同一张图中，观察不具有项目其余部分的测试的生命周期。图中展示了不同测试活动在非迭代视图中相互联系的方式：



测试的生命周期。

该生命周期必须与迭代方法结合起来，这意味着每个迭代都将具有遵循该模式的测试周期。



执行测试既是新测试的执行，又是使用先前测试的回归测试。

测试的生命周期是软件生命周期的一部分；它们应该同时开始。测试的设计开发过程与正在构建的应用程序一样复杂和艰巨。如果未能尽早开始，测试或者不够完善，或者会导致需要在开发时间表上附加一个长时间的测试和错误修正时间表，这将有违迭代开发的初衷。此外，测试计划和设计活动可以揭示应用程序定义中的故障和缺陷。这些问题越早得以解决，对整个时间表造成的影响就越小。评价过程中发现的问题可以在本次迭代解决，也可以留待下次迭代解决。通过核实已经实施的需求来评测迭代的完全程度，是评价的主要任务之一。迭代之间始终存在着某种“需求蠕变”，您需要意识到其存在并能够对其加以管理。

执行测试的方式取决于多种因素：您的应用领域、预算、公司策略和风险承受能力以及职员。对于测试的投资多少取决于在具体环境中评价质量和承受风险的方式。

测试的主要评测方法

简介

测试的主要评测方法包括覆盖和质量。
测试覆盖是对测试完全程度的评测，它建立在测试覆盖基础上，测试覆盖是由测试需求和测试用例的覆盖或已执行代码的覆盖表示的。
质量是对测试对象（系统或测试的应用程序）的可靠性、稳定性以及性能的评测。质量建立在对测试结果的评估和对测试过程中确定的变更请求（缺陷）的分析的基础上。

覆盖评测

覆盖指标提供了“测试的完全程度如何？”这一问题的答案。最常用的覆盖评测是基于需求的测试覆盖和基于代码的测试覆盖。简而言之，测试覆盖是就需求（基于需求的）或代码的设计/实施标准（基于代码的）而言的完全程度的任意评测，如用例的核实（基于需求的）或所有代码行的执行（基于代码的）。
系统的测试活动建立在至少一个测试覆盖策略基础上。覆盖策略陈述测试的一般目的，指导测试用例的设计。覆盖策略的陈述可以简单到只说明核实所有性能。
如果需求已经完全分类，则基于需求的覆盖策略可能足以生成测试完全程度的可计量评测。例如，如果已经确定了所有性能测试需求，则可以引用测试结果来得到评测，如已经核实了 75% 的性能测试需求。
如果应用基于代码的覆盖，则测试策略是根据测试已经执行的源代码的多少来表示的。这种测试覆盖策略类型对于安全至上的系统来说非常重要。
两种评测都可以手工得到（公式如下所示）或通过测试自动化工具计算得到。

基于需求的测试覆盖

基于需求的测试覆盖在测试生命周期中要评测多次，并在测试生命周期的里程碑处提供测试覆盖

的标识（如已计划的、已实施的、已执行的和成功的测试覆盖）。
测试覆盖通过以下公式计算：
测试覆盖 = $T^{(p,i,x,s)}$ / RfT

其中：
T 是用测试过程或测试用例表示的测试 (Test) 数（已计划的、已实施的或成功的）。
RfT 是测试需求 (Requirement for Test) 的总数。
在制定测试计划活动中，将计算测试覆盖以决定已计划的测试覆盖，其计算方法如下：
测试覆盖（已计划的） = T^{p} / RfT

其中：
T^{p} 是用测试过程或测试用例表示的已计划测试 (Test) 数。
RfT 是测试需求 (Requirement for Test) 的总数。
在实施测试活动中，由于测试过程正在实施中（按照测试脚本），在计算测试覆盖时使用以下公式：
测试覆盖（已执行的） = T^{i} / RfT

其中：
T^{x} 是用测试过程或测试用例表示的已执行的测试 (Test) 数。
RfT 是测试需求 (Requirement for Test) 的总数。
在执行测试活动中，使用两个测试覆盖评测，一个确定通过执行测试获得的测试覆盖，另一个确定成功的测试覆盖（即执行时未出现失败的测试，如没有出现缺陷或意外结果的测试）。
这些覆盖评测通过以下公式计算：
测试覆盖（已执行的） = T^{x} / RfT
其中：
T^{x} 是用测试过程或测试用例表示的已执行的测试 (Test) 数。
RfT 是测试需求 (Requirement for Test) 的总数。

成功的测试覆盖（已执行的） = T^{s} / RfT
其中：
T^{s} 是用完全成功、没有缺陷的测试过程或测试用例表示的已执行测试 (Test) 数。
RfT 是测试需求 (Requirement for Test) 的总数。

如将以上比率转换为百分数，则以下基于需求的测试覆盖的陈述成立：
x% 的测试用例（上述公式中的 $T^{(p,i,x,s)}$）已经覆盖，成功率为 y%
这一关于测试覆盖的陈述是有意义的，可以将其与已定义的成功标准进行对比。如果不符合该标准，则此陈述将成为预测剩余测试工作量的基础。

基于代码的测试覆盖

基于代码的测试覆盖评测测试过程中已经执行的代码的多少，与之相对的是要执行的剩余代码的多少。代码覆盖可以建立在控制流（语句、分支或路径）或数据流的基础上。控制流覆盖的目的是测试代码行、分支条件、代码中的路径或软件控制流的其他元素。数据流覆盖的目的是通过软件操作测试数据状态是否有效，例如，数据元素在使用之前是否已作定义。
基于代码的测试覆盖通过以下公式计算：

测试覆盖 = I^e / TIic
其中：
I^e 是用代码语句、代码分支、代码路径、数据状态判定点或数据元素名表示的已执行项目数。
TIic (Total number of Items in the code) 是代码中的项目总数。
如将该比率转换为百分数，则以下基于代码的测试覆盖的陈述成立：
x% 的测试用例（上述公式中的 I）已经覆盖，成功率为 y%
这一关于测试覆盖的陈述是有意义的，可以将其与已定义的成功标准进行对比。如果不符合该标准，则此陈述将成为预测剩余测试工作量的基础。

质量评测

测试覆盖的评估提供对测试完全程度的评测，在测试过程中已发现缺陷的评估提供了最佳的软件质量指标。因为质量是软件与需求相符程度的指标，所以在这种环境中，缺陷被标识为一种更改请求，该更改请求中的测试对象与需求不符。
缺陷评估可能建立在各种方法上，这些方法种类繁多，从简单的缺陷计数到严格的统计建模不一而足。
严格的评估假定测试过程中缺陷达到的比率或发现的比率。常用模型假定该比率符合泊松分布。则有关缺陷率的实际数据可以适用于这一模型。生成的评估将评估当前软件的可靠性，并且预测继续测试并排除缺陷时可靠性如何增长。该评估被描述为软件可靠性增长建模，这是一个活跃的研究领域。由于该类型的评估缺乏工具支持，所以应该慎重平衡成本与其增加价值。
缺陷分析就是分析缺陷在与缺陷关联关系的一个或多个参数值上的分布。缺陷分析提供了一个软件可靠性指标。
对于缺陷分析，常用的主要缺陷参数有四个：
状态：缺陷的当前状态（打开的、正在修复或关闭的等）。
优先级：必须处理和解决缺陷的相对重要性。
严重性：缺陷的相关影响。对最终用户、组织或第三方的影响等等。
起源：导致缺陷的起源故障及其位置，或排除该缺陷需要修复的构件。

可以将缺陷计数作为时间的函数来报告，即创建**缺陷趋势**图或报告；也可以将缺陷计数作为一个或多个缺陷参数的函数来报告，如作为**缺陷密度**报告中采用的严重性或状态参数的函数。这些分析类型分别为揭示软件可靠性的缺陷趋势或缺陷分布提供了判断依据。

例如，预期缺陷发现率将随着测试进度和修复进度而最终减少。可以设定一个阈值，在缺陷发现率低于该阈值时才能部署软件。也可根据执行模型中的起源报告缺陷计数，以允许检测“较差的模块”、“热点”或需要再三修复的软件部分，从而指示一些更基本的设计缺陷。
这种分析中包含的缺陷必须是已确认的缺陷。不是所有已报告的缺陷都报告实际的缺陷，这是因为某些缺陷可能是扩展请求，超出了项目的规模，或描述的是已报告的缺陷。然而，需要查看并分析一下，为什么许多报告的缺陷不是重复的缺陷就是未经确认的缺陷，这样做是有价值的。

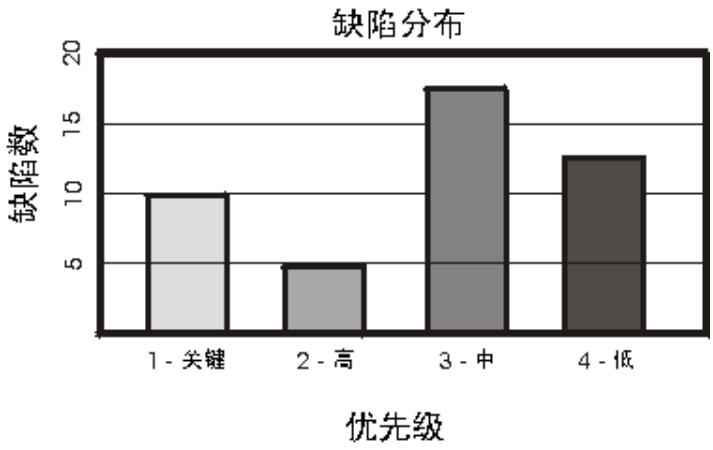
缺陷报告

Rational Unified Process 以三类形式的报告提供缺陷评估：
缺陷分布（密度）报告允许将缺陷计数作为一个或多个缺陷参数的函数来显示。

缺陷龄期报告是一种特殊类型的缺陷分布报告。缺陷龄期报告显示缺陷处于特定状态下的时间长短，如“提出的”。在龄期类别中，缺陷还可以按其他属性分类，如“拥有者”。
缺陷趋势报告按状态（新的、已打开的或关闭的）将缺陷计数作为时间的函数显示。趋势报告可以是累计的，也可以是非累计的。
测试结果和进度报告显示对测试的应用程序进行若干次迭代和测试生命周期后的测试过程执行结果。
许多此类报告对于评估软件质量具有很高的价值。一般测试标准中包括有关特定类别（如严重性级别）中打开的缺陷数的陈述。通过缺陷分布评估可以轻松地核对该标准。对测试需求进行过滤或分类，该评估可以侧重于不同的需求集。
要有效生成此类报告，一般需要工具支持。

缺陷密度报告

缺陷状态与优先级
应该给定所有缺陷的优先级，通常可行的做法是设定四种优先级中的一种：
立即解决
高优先级
正常排队
低优先级
一个成功测试的标准可以表示为缺陷在上述优先级上所应体现的分布方式。例如，对于一个成功的测试标准来说，可能不存在优先级为 1 的打开的缺陷，而且优先级为 2 的打开的缺陷要少于 5 个。例如下面的缺陷分布图：



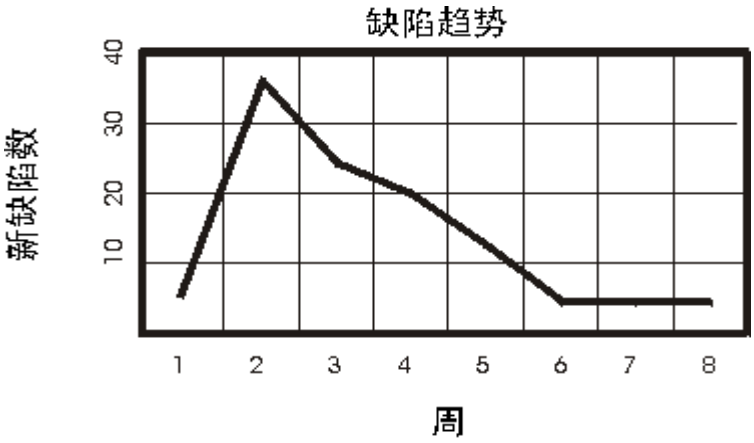
很明显该图显示的情况没有达到标准。请注意，该图需要通过过滤器才能只显示需要的打开的缺陷。
缺陷状态与严重性
缺陷严重性报告显示每种严重性级别的缺陷个数，例如致命错误、未执行主要功能、次要错误等严重性级别。
缺陷状态与在实施模型中的位置
缺陷起源报告显示缺陷在实施模型元素上的分布情况。

缺陷龄期报告

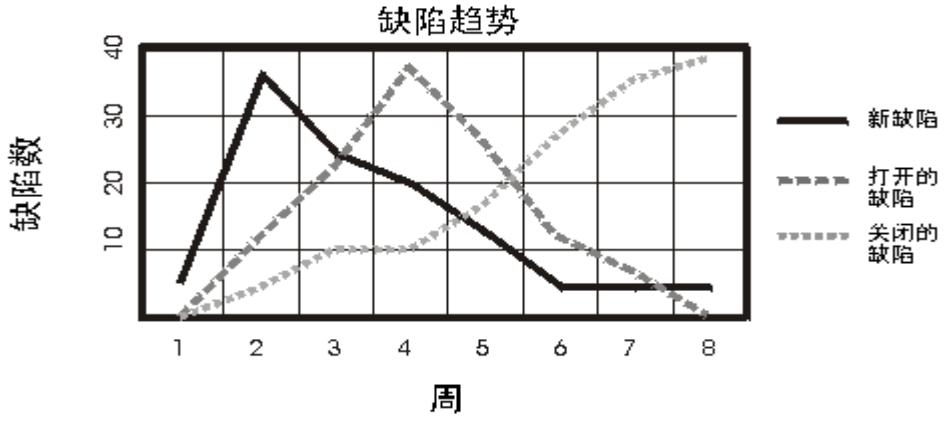
缺陷龄期分析提供了有关测试有效性和缺陷排除活动的良好反馈。例如，如果大部分龄期较长的、未解决的缺陷处于有待确认的状态，则可能表明没有充足的资源应用于再次测试工作。

缺陷趋势报告

趋势报告确定缺陷率并提供了一个出色的测试状态视图。在测试生命周期中，缺陷趋势遵循着一种比较好预测的模式。在生命周期的初期，缺陷率增长很快。在达到顶峰后，就随时间以较慢的速率下降。



要发现问题，可以根据这一趋势复审项目时间表。例如，在四个星期的生命周期中，如果缺陷率在第三个星期中仍然增长，则项目很明显没有按时间表进行。
这一简单的趋势分析假定：缺陷是立即关闭的，且在随后的工作版本中对修复进行测试，这样关闭缺陷的速率应该遵循与打开缺陷的速率相同的增减趋势。如果情况并非如此，则表明缺陷解决流程发生了问题；缺陷修复所需的资源或再次测试和确认修复所需的资源可能不足。



该报告反映的趋势显示，在项目开始时，发现和打开新缺陷的速率很快，但随着时间推移，该速率不断降低。打开的缺陷的趋势与新缺陷的趋势相似，但稍微滞后一些。关闭的缺陷的趋势随着打开的缺陷的修复和核实而不断增长。这些趋势描述的是成功的工作。

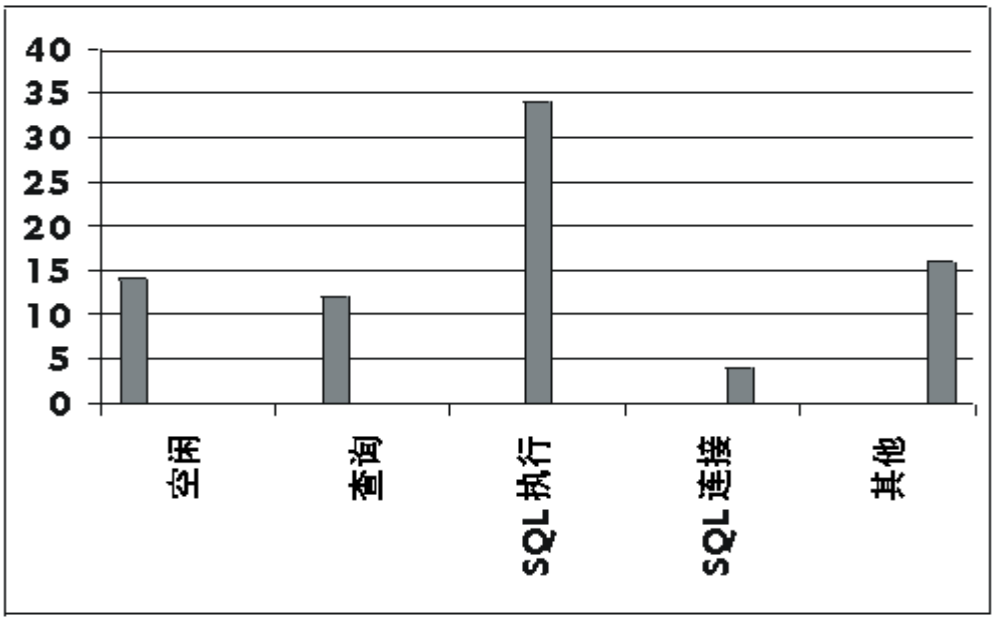
如果您的趋势与这些趋势差别显著，则表明存在问题，并可以确定可能需要附加资源以应用于开发或测试特定区域的时间。
当与测试覆盖评测结合起来时，缺陷分析可提供出色的评估，测试完成的标准也可以建立在此评估基础上。

性能评测

评估测试对象的性能行为时，可以使用多种评测，这些评测侧重于获取与行为相关的数据，如响应时间、计时配置文件、执行流、操作可靠性和限制。这些评测主要在评估测试活动中进行评估，但是也可以在执行测试活动中使用性能评测评估测试进度和状态。
主要的性能评测包括：
动态监测 - 在测试执行过程中，实时获取并显示正在执行的各测试脚本的状态。
响应时间/吞吐量 - 测试对象针对特定主角和/或用例的响应时间或吞吐量的评测。
百分位报告 - 数据已收集值的百分位评测/计算。
比较报告 - 代表不同测试执行情况的两个（或多个）数据集之间的差异或趋势。
追踪报告 -主角（测试脚本）和测试对象之间的消息/会话详细信息。

动态监测

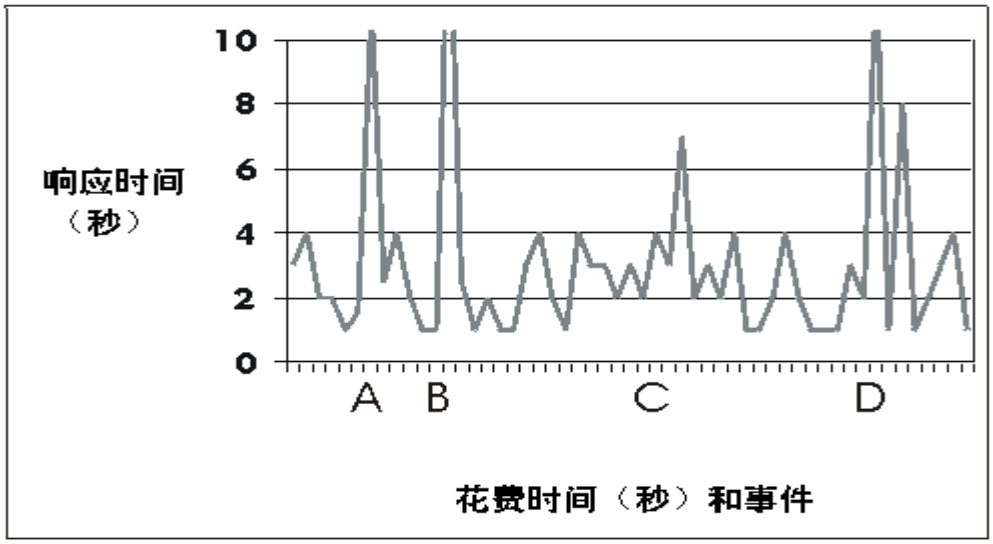
动态监测通常以柱状图或曲线图的形式提供实时显示/报告。该报告用于在测试执行过程中，通过显示当前的情况、状态以及测试脚本正在执行的进度来监测或评估性能测试执行情况。



例如，在以上柱状图中，有 80 个测试脚本正在执行相同的用例。图中显示，有 14 个测试脚本处于空闲状态，12 个处于查询状态，34 个处于 SQL 执行状态，4 个处于 SQL 连接状态，16 个处于其他状态。随着测试的进行，我们将看到各状态脚本的数量会发生变化。显示的输出将是正常执行且正在执行中的典型测试执行。但是，如果在测试执行过程中，测试脚本始终保持一种状态或没有显示任何变化，则表明测试执行发生问题或者需要实施或执行其他性能评测。

响应时间/吞吐量报告

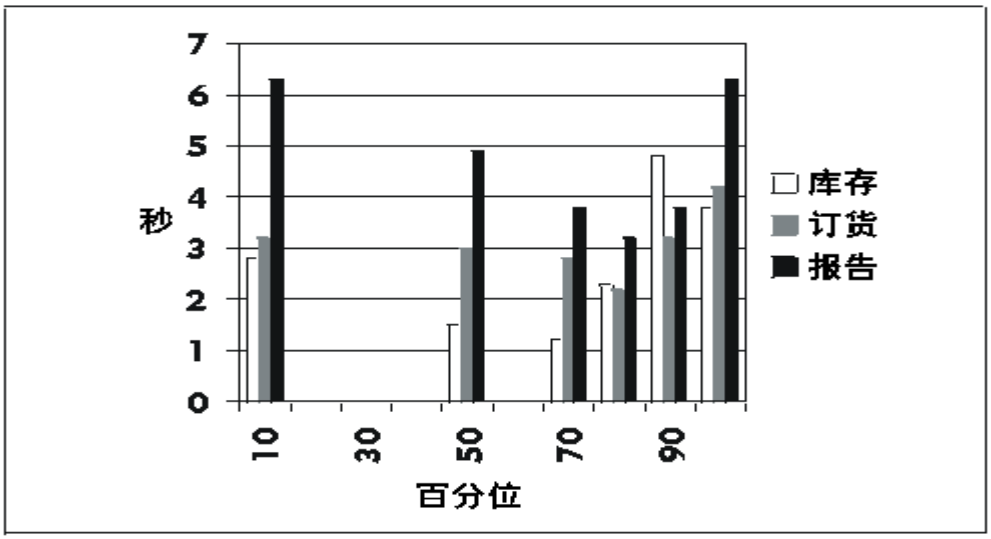
正如其名称的含义一样，响应时间/吞吐量报告评测并计算与时间和/或吞吐量（处理的事务数）相关的性能行为。这些报告通常用曲线图显示，响应时间（或事务数）在“y”轴上，而事件数在“x”轴上。



除了显示实际的性能行为外，它在计算并显示统计信息方面也很实用，如显示数据值的平均偏差和标准偏差。

百分位报告

百分位报告通过显示已收集数据类型的全体百分位值，提供了另一种性能统计计算方法。



比较报告

比较不同性能测试的结果，以评估测试执行过程之间所作的变更对性能行为的影响，这种做法是

非常必要的。比较报告应该用于显示两个数据集（分别代表不同的测试执行）之间的差异或多个测试执行之间的趋势。

追踪和配置文件报告

当性能行为可以接受时，或性能监测表明存在可能的瓶颈时（如当测试脚本保持给定状态的时间过长），追踪报告可能是最有价值的报告。追踪和配置文件报告显示低级信息。该信息包括主角与测试对象之间的消息、执行流、数据访问以及函数和系统调用。

测试策略

项目测试部分的策略描述测试活动的一般方法和目标。其中包括要进行的测试阶段（单元测试、集成测试和系统测试）以及要执行的测试类型（功能测试、性能测试、负载测试、强度测试等）。

该策略定义：

要使用的测试方法和工具。

测试完成和测试成功所采用的评价标准。例如，当成功执行 95% 的测试用例后，该标准可能允许软件进行验收测试。另一个标准是代码覆盖。在安全至上的系统中，该标准可能要求测试应该覆盖 100% 的代码。

影响资源要求或涉及进度的特殊考虑，如：

测试与外部系统之间的接口。

模拟物理损坏或安全威胁。

有些组织具有自行定义的公司测试策略。在这种情况下，需要将相应策略应用到特定的项目上。

制定测试计划活动应该侧重的最重要的维度如下：

处于什么迭代之中以及迭代的目的是什么。

正在执行什么测试阶段（单元测试、集成测试或系统测试）。可以在一次迭代中执行所有测试阶段。

现在来看一下测试活动的特征可以如何根据您所处的上述“测试维度”而变更。当然，可以查看的特征很多，如需要的资源和花费的时间，但在此处，请专注于定义测试策略的重要元素：

测试类型（功能测试、强度测试、容量测试、性能测试、可用性测试、分布测试等）。

使用的评估标准（基于代码的测试覆盖、基于需求的测试覆盖、缺陷数量、平均故障间隔时间等。）

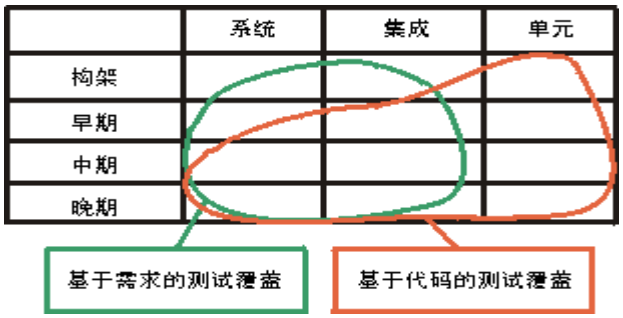
使用的测试方法（手工和自动）

测试类型在测试生命周期上没有通用的分布模式。根据迭代次数、迭代大小、项目种类的不同，可以重点执行不同的测试类型。

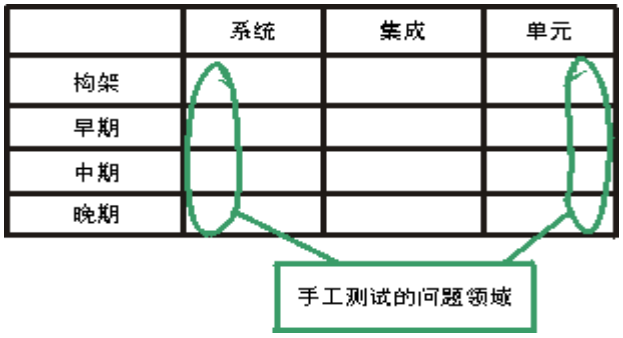
您将发现，系统测试阶段十分注重确保覆盖所有通过测试用例集表示的可测试需求。这意味着测试的完成标准将主要侧重于基于需求的测试覆盖。在集成测试和单元测试阶段，您将发现基于代码的测试覆盖是更合适的完成标准。下图显示在进行软件的新迭代时，这两类测试覆盖评测标准的使用是如何变化的。

测试计划应该定义单元测试、集成测试和系统测试的完成标准集。

可以针对各次迭代采用不同的完成标准集。



您在项目中应该考虑尽量自动执行测试，尤其对于需要多次重复执行的测试（回归测试）。但需要切记的是，创建并维护自动测试需要花费时间并占用资源。在各个项目中始终都会有一定数量的手工测试。下图说明了在什么时间和什么测试阶段可能会执行手工测试。



示例：

下表显示何时区分不同的测试类型，并提供要定义的完成标准的示例。第一个表显示的是“典型的”MIS 项目：

迭代/测试	系统测试	集成测试	单元测试
迭代 1	用于所有用例的自动性能测试。 • 所有已计划的测试都已执行。 • 所有严重性为 1 的缺陷都已经解决。 所有已计划的测试都已经重新执行，并且没有发现严重性为 1 的新缺陷。	无	非正式测试
迭代 2	用于所有新用例的自动性能和功能测试，以及作为回归测试的此前测试。 • 所有已计划的测试都已执行。 • 所有严重性为 1 和 2 的缺陷都已经解决。 • 所有已计划的测试都已经重新执行，并且没有发现严重性为 1 或 2 的新缺陷。	无	非正式测试
迭代 3	用于所有新用例的自动性能和功能测试，以及作为回归测试的此前测试。 必须有 95% 的用例通过测试。 • 所有已计划的测试都已执行。 • 所有严重性为 1、2 和 3 缺陷都已发现。	自动测试，70% 的代码覆盖。	非正式测试
迭代 4	用于所有用例的自动功能测试和负面测试，用于所有没有自动执行的部分的手工测试，以及作为回归	自动测试，80% 的代码覆盖。	非正式测试

	测试的先前测试。 必须有 100% 的用例通过测试。 • 所有已计划的测试都已执行。 • 所有严重性为 1、2 和 3 的缺陷都已经解决。 • 所有已计划的测试都已经重新执行，并且没有发现严重性为 1 或 2 的新缺陷。		

第二个表显示应用于“典型”的安全至上系统的测试类型和完成标准：

迭代 / 测试	系统测试	集成测试	单元测试
迭代 1	用于所有用例的自动性能测试，100% 的测试用例覆盖。 • 所有已计划的测试都已执行。 • 所有严重性为 1 的缺陷都已经解决。 • 所有已计划的测试都已经重新执行，并且没有发现新的缺陷。	无	无
迭代 2	用于所有用例的自动性能、功能和负面测试，100% 的测试用例覆盖。 • 所有已计划的测试都已执行。 • 所有严重性为 1 或 2 的缺陷都已经解决。 • 所有已计划的测试都已经重新执行，并且没有发现新的缺陷。	自动性能测试	非正式测试
迭代 3	用于所有用例的自动性能、功能、负面可用性和文档测试，100% 的测试用例覆盖。 • 所有已计划的测试都已执行。 • 所有严重性为 1、2 和 3 的缺陷都已经解决。 • 所有已计划的测试都已经重新执行，并且没有发现新的缺陷。	自动性能测试以及作为回归测试的先前测试	自动测试，70% 的代码覆盖
迭代 4	用于所有用例的自动性能、功能、负面可用性和文档测试，100% 的测试用例覆盖。 • 所有已计划的测试都已执行。 • 所有严重性为 1、2 和 3 的缺陷都已经解决。 • 所有已计划的测试都已经重新执行并且没有发现缺陷。	自动性能测试以及作为回归测试的先前测试	自动测试，80% 的代码覆盖

测试类型

按照测试简介中的说明，对软件进行的测试远远不止只测试测试对象的功能、接口和响应时间特征。附加测试必须注重特征/属性，如测试对象的：

完整性（防止失败的能力）

在不同平台上安装和执行的能力

同时处理多个请求的能力

...

为达到上述目的，需要实施和执行多种不同的测试，每种测试都有其具体的测试目标。每种测试都只侧重于对测试对象的一个特征或属性进行测试。

最常用的方法是按其处理的测试目标或质量维度的相似性来分组（请参见概念：质量维度），如：

质量维度	测试类型
可靠性	**完整性测试：**侧重于评估测试对象的强壮性（防止失败的能力），语言、语法的技术兼容性以及资源利用率的测试。该测试针对不同的测试对象实施和执行，包括单元和已集成单元。 **结构测试：**侧重于评估测试目标是否符合其设计和构造的测试。通常对基于 Web 的应用程序执行该测试，以确保所有链接都已连接、显示正确的内容以及没有孤立的内容。请参见概念：结构测试以获取附加信息。
功能	**配置测试：**侧重于确保测试对象在不同的硬件和/或软件配置上按预期运行的测试。该测试还可以作为系统性能测试来实施。 **功能测试：**侧重于核实测试对象按计划运行，提供需求的服务、方法或用例的测试。该测试针对不同的测试对象实施和执行，包括单元、已集成单元、应用程序和系统。 **安装测试：**侧重于确保测试对象在不同的硬件和/或软件配置上，以及在不同的条件下（磁盘空间不足或电源中断）按预期安装的测试。该测试针对不同的应用程序和系统实施和执行。 **安全测试：**侧重于确保只有预期的主角才可以访问测试对象、数据（或系统）的测试。该测试针对多种测试对象实施和执行。 **容量测试：**侧重于核实测试对象对于大量数据（输入和输出或驻留在数据库内）的处理能力的测试。容量测试包括多种测试策略，如创建返回整个数据库内容的查询；或者对查询设置很多限制，以至不返回数据；或者返回每个字段中最大数据量的数据条目。
性能 请参见概念：性能测试以获取附加信息	**基准测试：**一种性能测试，该测试将比较（新的或未知的）测试对象与已知的参照负载和系统的性能。 **竞争测试：**侧重于核实测试对象对于多个主角对相同资源（数据记录、内存等）的请求的处理是否可以接受的测试。 **负载测试：**一种性能测试，用于在测试的系统保持不变的情况下，核实和评估系统在不同负载下操作极限的可接受性。评测包括负载和响应时间的特征。如果系统结合了分布式构架或负载平衡方法，将执行特殊的测试以确保分布和负载平

衡方法能够正常工作。

性能曲线：在该测试中，将监测测试对象的计时配置文件，包括执行流、数据访问、函数和系统调用，以确定并解决性能瓶颈和低效流程。

强度测试：一种性能测试，侧重于确保系统可在遇到异常条件时按预期运行。系统面对的工作强度可以包括过大的工作量、不充足的内存、不可用的服务/硬件或过低的共享资源。

测试阶段

在软件交付周期的不同阶段，通常需要对不同类型的目标应用测试。这些阶段是从测试小的构件（单元测试）到测试整个系统（系统测试）不断向前发展的。

单元测试

单元测试在迭代的早期实施，侧重于核实软件的最小可测试元素。单元测试通常应用于实施模型中的构件，核实是否已覆盖控制流和数据流，以及构件是否可以按照预期工作。这些期望值建立在构件参与执行用例的方式的基础上，参与方式可参见该用例的序列图。实施员在单元的开发期间执行单元测试。实施工作流程对单元测试作出了详细描述。

集成测试

执行集成测试是为了确保当把实施模型中的构件集成起来执行用例时，这些构件能够正常运行。测试对象是实施模型中的一个包或一组包。要集成的包通常来自于不同的开发组织。集成测试将揭示包接口规约中不够完全或错误的地方。

系统测试

当将软件作为整体运行或实施明确定义的软件行为子集时，即可进行系统测试。这种情况下的目标是系统的整个实施模型。

验收测试

验收测试是部署软件之前的最后一个测试操作。验收测试的目的是确保软件准备就绪，并且可以供最终用户用于执行软件的既定功能和任务。请参见概念：验收测试以获取附加信息。

性能测试

性能测试是为描述测试对象与性能相关的特征并对其进行评价，而实施和执行的一类测试，如描述和评价计时配置文件、执行流、响应时间以及操作的可靠性和限制等特征。不同类型的性能测试侧重于不同的测试目标，这些性能测试的实施贯穿于整个软件开发生命周期 (Software Development Life Cycle, SDLC)。起初，在构架迭代中，性能测试侧重于确定和消除与构架有关的性能瓶颈。在构建迭代中还将实施和执行其他类型的性能测试，以调整软件和环境（优化响应时间和资源），并核实应用程序和系统是否能够处理高负载和高强度的情况，如有大量事务、客户机

和/或数据的情况。
性能测试中包含以下测试类型：

基准测试 – 比较新的或未知测试对象与已知参照标准（如现有软件或评测标准）的性能。

争用测试： - 核实测试对象对于多个主角对相同资源（数据记录、内存等）的请求的处理是否可以接受。
性能配置 - 核实在操作条件保持不变的情况下，测试对象在使用不同配置时其性能行为的可接受性。
负载测试 - 核实在保持配置不变的情况下，测试对象在不同操作条件（如不同用户数、事务数等）下性能行为的可接受性。
强度测试 - 核实测试对象性能行为在异常或极端条件（如资源减少或用户数过多）之下的可接受性。
性能评价通常是和**用户代表**一起协作并且以多级方法执行的。
性能分析的第一级涉及单一主角/用例实例的结果评价和多个测试执行的结果比较。例如，在测试对象上没有其他活动的情况下，记录单一主角执行单一用例的性能行为，并将结果与相同主角/用例的其他几个测试执行进行比较。第一级分析有助于确定可以表明系统资源中存在争用的趋势，该趋势将影响从其他性能测试结果所得出的结论的有效性。
分析的第二级检查特定主角/用例执行的摘要统计信息和实际数据值，以及测试对象的性能行为。摘要统计信息包括响应时间的标准偏差和百分位分布，这些信息显示了系统响应的变动情况，正如每个主角所见到的一样。
分析的第三级有助于理解性能问题的起因和加权值。该详细分析采用低级数据并且使用统计方法，帮助测试员从数据中得出正确的结论。详细分析为决策提供客观和定量的标准，但是它耗时较长，并且要求对统计学有基本的理解。
当性能行为差异确实存在，或是由于某些与测试数据收集相关的随机事件引起时，详细分析使用**统计加权值**的概念来帮助理解。即认为在基本级上，任何事件都具有随机性。统计测试确定是否存在无法用随机事件解释的系统差异。
有关不同性能测试报告的详细信息，请参见概念：测试的主要评测方法。

结构测试

基于 Web 的应用程序（使用 Internet 应用技术的程序）的吸引力和流行程度都在不断提高，尤其是因为组织可以通过此类应用程序利用多项与技术相关的优点，例如：
发展客户、潜在客户和业务伙伴群体，而无需分发任何软件或技术说明书。只要有浏览器并可访问“网络”(Internet / Intranet)，任何人都可以轻松地使用浏览器浏览已发布的 URL 并立即运行该应用程序。
集中式的控制和维护。借助基于 Web 的应用程序的“瘦客户机/胖服务器”模型，应用程序构件和逻辑放置在 Web 服务器上，集中并简化了控制和维护。这还使开发人员能够自动分发软件，只要将应用程序存放在服务器上，所有用户就可以立即执行它。
在为采用该技术的人员带来好处的同时，基于 Web 的应用程序也提高了对测试的要求。这些基于 Web 的应用程序的测试与其对应的非 Web 应用程序（客户机/服务器、遗留的应用程序等）相似，需要进行测试来处理应用程序的功能和性能特征。但基于 Web 的应用程序另外还需要进行侧重于应用程序结构的测试，以确保其结构良好和所有链接有效。
基于 Web 的应用程序通常使用一系列文档（既有 HTML 文本文档，又有 GIF/JPEG 图形）来构建，这些文档通过很多静态链接，以及少量活动的或由程序控制的链接连接起来。这些应用程

序还可包括“活动内容”，如表单、Java 脚本、通过插件显示的内容或 Java 应用程序。这种活动内容通常只用于输出，如用于音频输出或视频显示。它还可以用作导航助手，帮助用户遨游于应用（Web 站点）之中。基于 Web 的应用程序的这种自由形态的本质（通过其链接）是一种强大的功能，但同时也是一个巨大的弱点，因为其结构完整性很容易损坏。
实施和执行结构测试的目的是核实所有链接（静态的或活动的）都连接正确。这些测试包括：
核实是否显示每一链接的正确内容（文本、图形等）。在基于 Web 的应用程序中，目标内容是使用不同类型的链接引用的，如指向其他目标内容（在相同或不同 Web 站点中）的书签、超链接或热点链接。每个链接都要核实，以确保呈现在用户面前的是正确的目标内容。
确保没有断开的链接。断开的链接是指那些无法找到目标内容的链接。链接断开的原因有多种，包括移动、删除或重命名目标内容文件。链接还可能由于使用不正确的语法而断开，包括丢失斜杠、冒号或字母。
核实是否有孤立的内容。孤立的内容是指当前 Web 站点中没有“入站”链接的文件，即无法访问或显示的内容。 必须对孤立的内容进行仔细地调查，确定其原因 - 孤立的内容是由于本身不再需要而导致的？还是由于断开的链接而导致的？或者其内容是通过当前 Web 站点以外的链接访问的。 确定了原因之后，需要采取适当的操作（删除该内容文件、修复断开的链接或忽略孤立的内容）。

验收测试

验收测试是部署软件之前的最后一个测试操作。验收测试的目的是确保软件准备就绪，并且可以让最终用户将其用于执行软件的既定功能和任务。实施验收测试的常用策略有三种，它们分别是：
正式验收
非正式验收或 Alpha 测试
Beta 测试
您选择的策略通常建立在合同需求、组织和公司标准以及应用领域的基础上。

正式验收测试

正式验收测试是一项管理严格的过程，它通常是系统测试的延续。计划和设计这些测试的周密和详细程度不亚于系统测试。选择的测试用例应该是系统测试中所执行测试用例的子集。不要偏离所选择的测试用例方向，这一点很重要。在很多组织中，正式验收测试是完全自动执行的。
对于系统测试，活动和工件是一样的。在某些组织中，开发组织（或其独立的测试小组）与最终用户组织的代表一起执行验收测试。在其他组织中，验收测试则完全由最终用户组织执行，或者由最终用户组织选择人员组成一个客观公正的小组来执行。
这种测试形式的优点是：
要测试的功能和特性都是已知的。
测试的细节是已知的并且可以对其进行评测。
这种测试可以自动执行，支持回归测试。
可以对测试过程进行评测和监测。
可接受性标准是已知的。
缺点包括：
要求大量的资源和计划。
这些测试可能是系统测试的再次实施。

可能无法发现软件中由于主观原因造成的缺陷，这是因为您只查找预期要发现的缺陷。

非正式验收测试

在非正式验收测试中，执行测试过程的限定不象正式验收测试中那样严格。在此测试中，确定并记录要研究的功能和业务任务，但没有可以遵循的特定测试用例。测试内容由各测试员决定。这种验收测试方法不象正式验收测试那样组织有序，而且更为主观。
大多数情况下，非正式验收测试是由最终用户组织执行的。
这种测试形式的优点是：
要测试的功能和特性都是已知的。
可以对测试过程进行评测和监测。
可接受性标准是已知的。
与正式验收测试相比，可以发现更多由于主观原因造成的缺陷。
缺点包括：
要求资源、计划和管理资源。
无法控制所使用的测试用例。
最终用户可能沿用系统工作的方式，并可能无法发现缺陷。
最终用户可能专注于比较新系统与遗留系统，而不是专注于查找缺陷。
用于验收测试的资源不受项目的控制，并且可能受到压缩。

Beta 测试

在以上三种验收测试策略中，Beta 测试需要的控制是最少的。在 Beta 测试中，采用的细节多少、数据和方法完全由各测试员决定。各测试员负责创建自己的环境、选择数据，并决定要研究的功能、特性或任务。各测试员负责确定自己对于系统当前状态的接受标准。
Beta 测试由最终用户实施，通常开发（或其他非最终用户）组织对其的管理很少或不进行管理。
Beta 测试是所有验收测试策略中最主观的。
这种测试形式的优点是：
测试由最终用户实施。
大量的潜在测试资源。
提高客户对参与人员的满意程度。
与正式或非正式验收测试相比，可以发现更多由于主观原因造成的缺陷。
缺点包括：
未对所有功能和/或特性进行测试。
测试流程难以评测。
最终用户可能沿用系统工作的方式，并可能没有发现或没有报告缺陷。
最终用户可能专注于比较新系统与遗留系统，而不是专注于查找缺陷。
用于验收测试的资源不受项目的控制，并且可能受到压缩。
可接受性标准是未知的。
您需要更多辅助性资源来管理 Beta 测试员。

测试自动化和工具

测试自动化工具越来越多的出现在市场上，这些工具可以自动执行测试活动。迄今为止，现有工具中还没有哪一种工具能够自动执行所有测试活动。事实上，多数工具是一个或几个活动专用的；有些工具的功能针对性很强，只能处理一个活动的某一部分。
评估不同的测试自动化工具时，有必要了解工具类型、工具的限制以及工具能够处理和自动执行的活动情况。关于测试自动化工具分类的说明如下所示。

功能

测试工具可以按其执行的功能分类。典型的工具功能名称包括：
数据获取工具，用于获取要在测试活动中使用的数据。数据可以通过转化、析取、变换或捕捉现有数据获取，或者通过从用例或补充规约生成获取
静态评测工具，用于分析设计模型、源代码或其他固定源中包含的信息。该分析将生成有关逻辑流、数据流或质量指标的信息，如复杂性、可维护性或代码行。
动态评测工具，用于在代码的执行过程中进行分析。这些评测包括代码的运行时操作，如内存错误检测和性能。
模拟器或**驱动程序**，它们执行由于时间、费用或安全原因而无法用于测试的活动。
测试管理工具，用于辅助测试活动或工件的计划、设计、实施、执行、评估和管理。

白盒与黑盒

通常根据工具的使用方式或使用工具所需要的技术或知识，用白盒或黑盒作为测试工具的特征。
白盒工具，依赖代码、设计模型或其他来源的资料的知识来实施和执行测试。
黑盒工具，依赖用例或测试对象的功能描述。其中白盒工具知道测试对象如何处理请求，而黑盒工具依赖输入和输出条件来评估测试。

测试工具的专用性

除了以上对工具作出的不太严格的分类外，还可以按专用性对工具进行分类。
录制/回放工具，用于将数据获取与动态评测结合起来。测试数据是在记录事件期间（测试实施的过程中）获取的。在测试执行过程中，这些数据随后将用于“回放”测试脚本，而后者用于评估测试对象的执行。
质量指标工具，该工具是静态评测工具，用于执行设计模型或源代码的静态分析，以建立描述测试对象质量的参数集。这些参数可以表明可靠性、复杂性、可维护性或其他质量评测标准。
覆盖监测器工具，可通过鉴别测试过程中对于测试对象在某些维度的覆盖程度，来表示测试的完全程度。典型的覆盖类包括基于需求的（用例）、逻辑分支或节点（基于代码）、数据状态和功能点。
测试用例生成器，可自动生成测试数据。测试用例生成器使用测试对象数据输入的正式规约或设计模型/源代码，生成测试指定输入、错误输入以及限制和边界用例的测试数据。
比较器工具，可比较测试结果与参照结果，并找出区别。比较器根据其专用的特定数据格式不同而有所区别。例如，比较器可能是基于像素的（比较位图图像）或基于对象的（比较对象特征或数据）。
数据析取器，用于从现有来源为测试用例提供输入。来源包括数据库、通信系统中的数据流、报告或设计模型/源代码。

测试工件集 

测试设计员：

测试计划
工作量分析文档
测试过程
测试用例
测试脚本
测试模型
测试评估摘要

测试员：

测试结果

设计员：

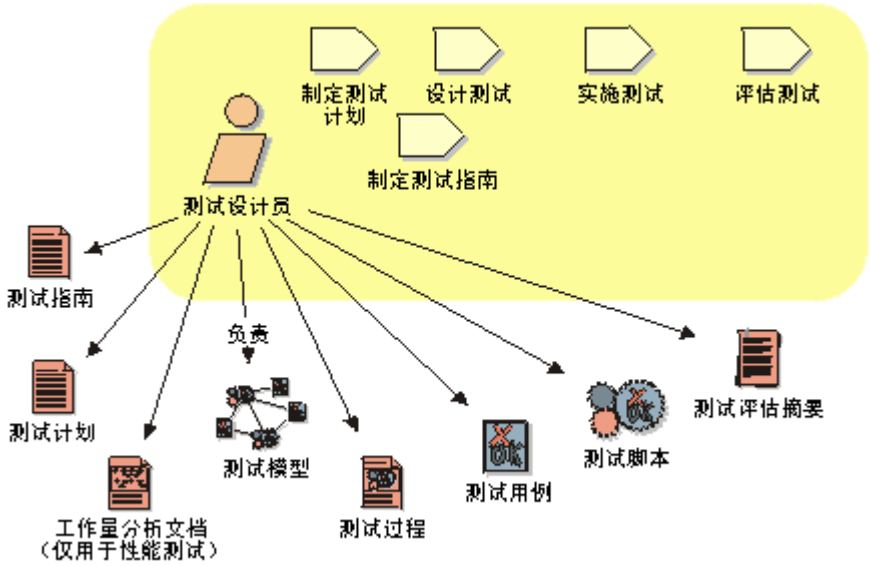
测试类
测试包

实施员：

测试构件
测试子系统

测试设计员：

测试设计员是测试中的主要角色。该角色负责对测试进行计划、设计、实施和评估，包括：
生成测试计划和测试模型
执行测试过程
评估测试范围和测试结果，以及测试的有效性
生成测试评估摘要



人员配备
测试设计员应具备的相应技能和知识包括：
了解系统或所测试的应用程序
了解测试及测试自动化工具
具备诊断和解决问题的技能
编程技能（最好具备）

制订测试计划：

目的 收集和组织测试计划信息。 创建测试计划。	
步骤 确定测试需求 评估风险 制定测试策略 确定资源 创建时间表 生成测试计划	
输入工件： 补充规约 用例模型 设计指南 设计模型 类 用例实现 软件构架文档 构件和实施子系统	**生成工件：** 已完成的测试计划

迭代计划 测试指南	
角色：测试设计员	
工作指南：复审	
工作流程明细： **核心工作流程：测试** 制定测试计划 **核心工作流程：项目管理** 管理迭代	

确定测试需求

目的 确定测试对象并指明测试范围。
工具向导： 使用 Rational TestManager™ 获取“确定测试需求”和“评估风险”的结果

确定测试需求是测试计划活动的开始。测试需求确定测试对象以及测试工作的范围和作用。测试需求还用来确定整个测试工作（如安排时间表、测试设计等）并作为测试覆盖的基础。

被确定的测试需求项必须是可核实的。即，它们必须有一个可观察、可评测的结果。无法核实的需求不是测试需求。

执行如下步骤确定测试需求：

复审所有材料。

指明测试需求。

复审所有材料。

由于测试需求可从多种来源确定，因此作为第一步，对将要开发的应用程序/系统的所有可用材料进行复审是十分重要的。最常用的测试需求来源包括现有的需求列表、用例、用例模型、用例实现、补充规约、设计需求、商业理由、最终用户访谈以及对现有系统的复审。

指明测试需求。

除确定测试需求的来源之外，还必须有某种形式的确定方法来确定将成为测试目标的需求。这将导致测试需求层次的产生。该层次可能基于现有的层次结构，也可能是新近生成的。层次结构是测试需求的逻辑分组。常用分组方法包括按照用例、商业理由、测试类型（功能、性能等）或者以上各项的组合对项目进行分组。

本步骤产生的结果是一份确定将成为测试目标的需求的报告（层次结构）。

请参见指南：测试计划以获取更多的关于确定测试需求的信息。

评估风险

目的 最大限度地提高测试效率并确定测试工作的优先级。 制定一个可接受的测试顺序。
工具向导： 使用 Rational TestManager 获取“确定测试需求”和“评估风险”的结果

要获得对风险的认识，请执行如下操作：

确定并验证风险因子

确定并验证实施概要因子
确定并验证测试优先级因子
确定并验证测试风险因子
测试工作需要平衡资源约束和风险。最重要的测试需求能够反映最高的风险。
可从以下几个方面观察风险：
效果 - 用例（需求等）失效造成的影响或后果
原因 - 确定不合需要的结果，并确定哪些用例或需求一旦失效将产生该结果
可能性 - 用例或需求失效的概率
复审每一项测试需求并确定风险因子（比如高、中或低）。有时从两个或更多方面对风险进行评估可能会得到不同的风险因子。在这种情况下，请使用最高的风险因子值。同时还应给出关于对某一给定测试需求为何选择特定风险因子的说明。
确定并验证测试的实施概要因子
大多数应用程序都有某些功能是经常使用的，而另外一些则是较少使用的。因此要对应用程序进行合理的测试，必须确保不仅测试具有最高风险的测试需求，而且还应测试经常使用的功能（因为这些功能通常是最终用户最频繁使用的）。
确定每一个测试需求的实施概要因子，并说明为什么确定特定因子值的原因。这可通过复审商业理由或者同最终用户及其经理访谈来完成。另一种方法是观察最终用户与系统的交互行为或者使用监视/记录软件来记录最终用户与系统的交互行为（供分析用）。
确定并验证测试优先级因子
在确定和验证每一个测试需求的测试风险和实施概要之后，就应该确定和验证测试优先级因子。测试优先级因子确定测试需求的相对重要性以及测试其的顺序或时序。
测试优先级因子通过利用风险因子、实施概要、合同义务、其他约束或者它们的组合来确定。在确定测试优先级时，考虑所有的因素十分重要，这样可以确保测试适当而有重点。
请参见指南：测试计划以获取关于评估风险和确定测试优先级的详细信息。

制定测试策略

目的
确定并传达测试手段和工具
确定并传达用于确定产品质量和测试是否完成的评估方法

测试策略的目的是向每一个人传达如何进行测试以及采用何种评测标准来确定测试的完成和成功程度。策略不必十分详尽，但它应该向读者指明如何进行测试。
制定测试策略包括：
确定和描述测试方法
确定测试标准
确定测试的特殊事项
确定和描述测试方法
测试方法是对如何实施测试的说明（陈述）。它应该说明或指出测试对象、测试时采取的主要操作以及如何核实结果等。说明应该为读者提供足够的信息以便他们能够理解测试的对象（尽管尚不了解测试深度），如以下说明陈述：
对于每一个用例，确定并执行测试用例，包括有效和无效的输入数据。
对于每一个用例都将设计和开发测试流程。
利用三个月的时间来实施测试过程以模拟客户账号管理。测试过程包括添加、修改和删除账号以及客户。
实施测试过程并通过 1500 个虚拟用户来执行测试脚本，每个用户都执行功能 A、B 和 C，并

且使用不同的输入数据。
确定测试标准
测试标准是关于测试的客观说明，它指明那些用于确定/识别测试完成时间的值和被测试应用程序质量。测试标准可能包括一系列说明或对其他文档（比如方法指南或测试标准等）的引用。测试标准应确定：
测试对象（具体测试目标）
评测方法
评估评测方法所采用的标准
测试标准示例：
对于每一高优先级用例：
所有计划好的测试用例和测试过程已被执行。
所有确定的缺陷已被解决。
所有计划好的测试用例和测试过程已被重新执行并且没有发现新的缺陷。
在上例中，“对于每一高优先级用例”说明测试对象。“所有计划好的测试用例和测试过程已被执行”说明评测的方法。评估标准包含在最后两个陈述“所有确定的缺陷已被解决”和“所有计划好的测试用例和测试过程已被重新执行并且没有发现新的缺陷”之中。
确定测试的特殊事项
应列出所有关于测试或者依赖关系的特殊事项，如下所示：
测试数据库将由操作资源恢复。
测试（性能）必须在办公结束后开始（不受日常的正常操作影响）并且在凌晨 5:00 以前结束。
必须与遗留系统同步（或模拟同步）。
请参照指南：测试计划以获取关于制定测试策略的其他信息。

确定资源 

目的
确定测试所需的资源，包括人力资源（技能、知识、可用性）、硬件、软件、工具等。

一旦确定测试对象和测试方法，就需要确定执行测试人员及测试活动所需支持。确定资源需求包括确定所需的资源，资源包括如下：
人力资源（人员数量和技能）
测试环境（包括硬件和软件）
工具
数据
确定人力资源需求
对于大多数测试工作而言，您需要符合下列条件的人力资源：
管理和制定测试计划
设计测试及数据
实施测试和数据
执行测试并评估结果
管理和维护测试系统
确定非人力资源需求
测试环境（包括硬件和软件）
推荐使用两个不同的物理环境（尽管这不是必需的）：
执行测试管理、设计和实施活动的实施环境，以及

执行所有测试的执行环境，它是一个独立的执行系统（通常是生产系统的克隆）。
软件也有必要进行测试。需要测试的软件最少应包括所测试的应用程序、客户机操作系统、网络以及服务器端操作系统。此外，可能还需要精确地模拟/复制生产环境的软件，这类软件包括：
与其他系统（如遗留系统）的接口。
其他桌面应用程序，如 Microsoft Office、Lotus Notes 等。
其他驻留于文件服务器和网络或在其中执行的应用程序。当所测试的应用程序不需要这些应用程序时，它们就驻留在其执行环境中。
工具
应该声明何种软件工具（供测试用）将被使用、被谁使用，以及使用各种工具所能够获得哪些信息或好处。
数据
软件测试在很大程度上取决于输入数据（创建或支持某一测试条件）和输出数据（同预期结果作比较）的使用。应确定解决以下与测试数据有关的问题的策略：
收集或生成用于测试的数据（输入和输出）。
数据库构架（隔离外界影响的手段以及在测试完成后将数据返回初始状态的方法）。

创建时间表 

目的
确定并传达测试工作、时间表和里程碑。

创建时间表包括：
估计测试工作
制定测试进度
估计测试工作
估计测试工作时，应考虑如下假设：
投入到项目中的人力资源的生产率和技能/知识水平（比如他们使用测试工具或程序的能力）。
要构建的应用程序的有关参数（比如窗口数目、构件、数据实体和相互关系以及重复使用的百分比等）。
测试覆盖（实施并执行测试的可接受深度）。如果只实施和执行一个测试用例（对每一个用例/需求），则表述每一个用例/需求是不同的。经常需要多个测试用例来对某一个用例/需求进行可以接受的测试。
测试估计还应考虑到在测试生命周期的各个阶段使用不同方式对工作进行划分，这是因为某些类型的（工作）量在生命周期内是变化的。例如，性能测试工作，由于其包含在复杂环境中建立测试系统并执行测试的工作，因此该测试执行活动就占了工作估计的很大比重。

此划分对于安排时间是很重要的。测试设计和测试实施工作需要一个单独的调度时期，并增加一些小的改进。相比之下，每一个应用程序工作版本都需要重复执行测试，因此必须对测试执行工作做相应的调整。

测试工作需要包含回归测试的时间。下表显示了经过不同测试阶段的几次迭代之后回归测试用例是如何进行积累的。

迭代与阶段	系统	集成	单元
第一次迭代	本次迭代中以系统为目标的测试	本次迭代中以工作版本为目标的测试用例	本次迭代中以单元为目标的测试用例的测试

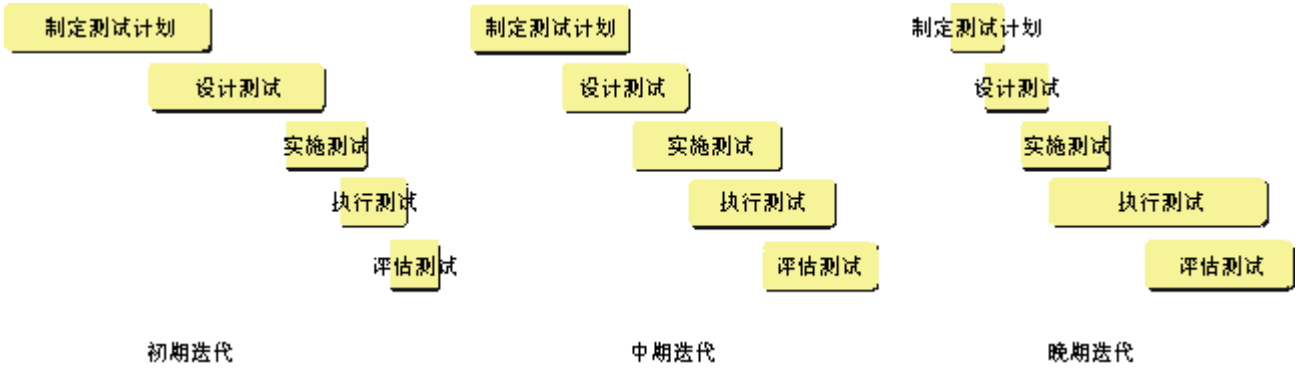
	用例的测试	的测试	
下一次迭代	本次迭代测试用例以及用于回归测试的先前迭代的测试用例的测试。	对本次迭代测试用例以及用于回归测试的先前迭代的测试用例的测试。	本次迭代测试用例以及用于回归测试的先前迭代的测试用例的测试。

制定测试进度

测试项目时间表可以通过工作估计和资源分配来建立。在迭代开发环境中，每一迭代都需要一个独立的测试项目时间表。在每一迭代中都将重复所有的测试活动。

在某一特定迭代中，测试计划和测试设计活动处理软件中的新功能。测试实施活动包括为新功能创建新的测试用例，并为已变更的功能修改测试用例。测试执行和评估步骤验证新功能并为现有功能执行回归测试。

早期迭代引入许多新功能和新测试。随着集成流程继续进行，新测试的数目将减少，而需要执行以检验累计功能的回归测试的数目将增加。因此，早期迭代更多地要求在测试计划和设计上进行工作，而后期迭代则偏重于测试执行和评估。



为每一迭代提供详尽的时间表是不可能的。通常并不知道将会有多少迭代，或者在哪一次迭代中将达到某一测试标准。

使用估计好的工作和已分配的资源创建测试工作的时间表。

以下示例表概述了所有测试活动。显示的工作估计是各项任务的相对工作数量。

在制定时间表时，必须确保它符合实际。有些时间表很有野心，但人们没有足够的时间或精力来按时间安排行动，从而导致无法成功执行测试。这样的时间表是再令人沮丧不过的了。

任务	有关工作
工作总计	**38d**
测试计划	**7d**
确定测试项目	1d
确定测试策略	1d
估计工作	1d

确定资源	1d
安排测试活动的日程	1d
记录测试计划	2d
指定测试用例	**5d**
确定测试用例	5d
设计测试	**7d**
分析测试需求	2d
指定测试过程	3d
指定测试用例	1d
复审测试需求覆盖	1d
实施测试	**12d**
建立测试实施环境	1d
记录并回放原型脚本	1d
制定测试过程	5d
测试与调试测试过程	1d
修改测试过程	2d
建立外部数据集	1d
重新测试与调试测试过程	1d
执行系统测试	**6d**
设置测试系统	1d
执行测试	2d
核实预期结果	1d
调查意外结果	1d
记录缺陷	1d
评估测试	**1d**
复审测试记录	0.25d
评估测试用例覆盖	0.25d
评估缺陷	0.25d
确定是否满足测试完成标准	0.25d

制定测试计划

目的
组织并向其他人员传达测试计划信息。

要生成测试计划，请执行如下步骤：
复审/改进现有材料
确定测试可交付工件

生成测试计划
复审/改进现有材料
在生成测试计划之前，应该复审所有现有项目信息以确保测试计划包含最新和最准确的信息。如果需要，应修改测试相关信息（测试需求、测试策略、资源等）以反映所有变更。
确定测试可交付工件
测试可交付工件部分的目的在于落实和规定创建、维护以及如何向其他人提供测试工件的方法。这些工件包括：
测试模型
测试用例
测试过程
测试脚本
变更请求
生成测试计划
制定测试计划活动的最后步骤是生成测试计划。它通过集中收集到的所有测试信息来完成，并生成一份报告。
测试计划应至少分发到以下对象：
所有测试角色
开发人员代表
股东代表
涉众代表
客户代表
最终用户代表

设计测试：

活动：设计测试**目的**
为每个工作版本确定可验证的测试用例集。 确定如何实现测试用例的测试过程。
步骤 工作量分析（仅用于性能测试） 确定并说明测试用例 确立并结构化测试过程 复审并评估测试覆盖

输入工件：	**生成工件：**
测试指南 测试计划 迭代计划 产品验收计划 质量保证计划 补充规约 用例模型 设计模型 类与设计包 构件	测试模型，设计视图 测试用例 测试过程 工作量分析文档，（仅用于性能测试）

用例实现 软件构架文档	
角色：测试设计员	
指南： 指南：单元测试 指南：测试用例 指南：测试过程 指南：工作量分析文档 指南：测试模型	

工作流程明细：
核心工作流程：实施
实施构件
核心工作流程：测试
设计测试

工作量分析

目的
确定并说明影响系统使用与系统性能的各种可变因素。
确定用于性能测试的用例子集。

工具向导
Rational LoadTest™：
制定性能测试计划
制定表示性能测试工作量的性能测试进度

执行工作量分析将生成工作量分析文档，该文档可用于性能测试。工作量分析文档包括以下主要输入：
软件开发计划
用例模型
设计模型
补充规约
工作量分析内容包括：
明确性能测试的目标与用例。
确定模型中要实施的用例。
确定性能测试中要模拟的主角和主角特征。
确定性能测试中要模拟的工作量（以主角、主角类和主角简档的数目计算）。
确定性能评测方法与标准。
复审待实施的用例，确定执行频率。
选择最频繁调用的用例及给系统带来最大负载的用例。
为上一步中确定的每个用例生成测试用例。
为每个测试用例确定关键评测点。
请参见工件：工作量分析文档与指南：工作量分析文档以获得其他信息。

确定并说明测试用例

目的
确定并说明要用于测试的条件 确定测试所需的特定数据 确定测试的预期结果
工具向导 **Rational LoadTest:** 为符合实际性能测试的数据生成数据池 制定表示性能测试工作量的性能测试进度

对于每项测试要求，应：
分析应用程序工作流程
确定并说明测试用例
确定测试用例数据
分析应用程序工作流程
该步骤的目的在于确定并说明主角与系统交互时的操作和/或步骤。这些测试过程说明将进一步用于确定与描述测试应用程序所需的测试用例。
请注意：这些初期的测试过程说明应是较概括的说明，即：对操作的说明应尽可能笼统，而不应具体引用实际构件或对象。
对各个用例或需求，应
复审用例事件流，或
走查并描述主角与系统交互时采取的操作/步骤
确定并说明测试用例
该步骤的目的是为每项测试需求编写适当的测试用例。
请注意：如果已测试过以前的版本，则测试用例已经存在。应复审这些测试用例，供回归测试及其设计使用。回归测试用例应包括在当前迭代中，并应与处理新行为的新测试用例结合使用。
用于确定测试用例的主要输入包括：
在某一点可遍历测试对象（系统、子系统或构件）的用例。
设计模型。
任何技术或补充需求。
测试对象应用程序映射表（由自动测试脚本生成工具生成）。
要说明测试用例，应阐明：
受测试的测试条件（或对象状态、应用程序状态）。
用例、用例场景或测试用例所依据的技术与补充需求。
以输出状态、条件或数据值表示的预期结果。
请注意：没有必要对所有用例、用例场景及技术或补充需求都进行测试。
执行本步骤将得到一份测试用例表格。表中说明测试条件、构成测试条件的对象、数据或对系统的影响，以及预期的结果。
请参见工件：测试用例与指南：测试用例以获得其他信息。
确定测试用例数据
使用上面生成的表格，复审测试用例，并确定支持这些测试用例的实际值。本步骤将确定用于以下三种目的的数据：
用作输入的数据值
用作预期结果的数据值
用作支持测试用例所需的数据（但是对一个特定测试用例，该数据不会用作输入或输出数据）
请参见工件：测试用例、指南：测试用例与指南：测试数据以获得其他信息。

确立并结构化测试过程 

目的 分析用例工作流程与测试用例，以此确定测试过程 在测试模型中确定生成该模型的测试用例与测试过程的关系
工具向导 **Rational TestManager™** 使用 Rational TestManager 获取确定并结构化测试过程的结果 **Rational TestFactory™** 使用 Rational TestFactory 获取自动测试的测试设计结果

执行下列操作：

复审应用程序工作流程或应用程序映射表

开发测试模型

使测试过程结构化

复审应用程序工作流程或应用程序映射表

复审应用程序工作流程及先前说明的测试过程，查看是否已对用例作出任何改动，影响到测试过程的确定和结构安排。

如果利用自动测试脚本生成工具，则应复审生成的应用程序映射表（用于生成测试脚本），以确保 UI 对象分层列表（代表测试对象用户界面中的控件）正确无误，并与您的测试和/或受测试的用例相关。

复审进行的方式与前面的分析方式相同：

复审用例事件流，并且

复审所描述的测试过程，还要

走查主角在与系统交互时采取的步骤，还可选择

复审应用程序映射表

开发测试模型

测试模型的目的在于传达以下信息：测试内容、测试方式以及实施测试的方式。对各个说明的测试过程（或应用程序映射表与生成的测试脚本），需要完成以下步骤以生成测试模型：

确定本测试过程与其他测试过程（或生成的测试脚本之间）的关系或顺序。

确定本测试过程的起始条件/状态与结束条件/状态。

指明本测试过程（或生成的测试脚本）要执行的测试用例。

开发测试模型时应考虑以下问题：

很多测试用例互为变体，这意味着可以通过同一测试过程实现这些用例。

很多测试用例要求执行的行为可能会交叉。为了能够重复实施这些行为，可使测试过程结构化，使同一测试过程可用于几个测试用例。

很多测试过程包含了大多数测试用例或其他测试过程常用的操作与步骤。这种情况下，应决定是否需要专门创建一个结构化的测试过程（对那些常用步骤），而测试用例特有的步骤仍另外保留在一个结构化的测试过程中。

使用自动测试脚本生成工具时，应复审应用程序映射表和生成的测试脚本，以确保测试模型反映了以下内容：

正确的（或期望的）控件已包含在应用程序映射表和测试脚本中。

控件按照所需顺序执行。

为那些需要测试数据的控件确定了测试用例。

指定了要在其中显示控件的窗口或对话框。

请参见工件：测试模型。
使测试过程结构化
上述测试过程的说明对实施与执行测试来说并不充分。需要调整测试过程的结构，即修改已说明的测试过程，让它至少应包括以下信息：
设置：如何为接受测试的测试用例创建条件，需要哪些数据（无论是输入数据还是测试数据库中的数据）。
结构化测试过程的起始条件、状态或操作。
对执行的说明：测试员实施与执行测试所采取的详细步骤/操作（说明应详细到对象或构件这一级）。
输入的数据值（或引用的测试用例）。
每个操作/步骤的预期结果（条件或数据，或引用的测试用例）。
评估结果：对得到的实际结果与预期结果进行比较的分析方法与步骤。
结构化测试过程的结束条件、状态或操作。
请注意：一个已说明的测试过程，在进行结构化时，可能分解为几个结构化测试过程，它们必须按顺序执行，这样做是为了获取最大的复用性，并使测试过程的维护成本最小化。
测试过程可象测试脚本（用于自动执行）一样手动执行或实施。自动执行测试过程时，生成的计算机可读文件就是测试脚本。
请参见工件：测试过程。
请参见工件：测试脚本。

复审并评估测试覆盖

目的
确定并说明要用于判断测试的完全程度的测试评测方法。
工具向导 **Rational TestManager** 使用 Rational TestManager 复审和评估基于需求的测试覆盖

应执行以下操作：
确定测试覆盖评测方法
生成并分发测试覆盖报告
测试覆盖评测方法用于确定测试当前或将要达到的完全程度。
确定测试覆盖评测方法
有两种确定测试覆盖的方法：
基于需求的覆盖。
基于代码的覆盖。
这两种方法都确定了将接受（或已接受）测试的全部可测试项的百分比，但它们使用不同的收集与计算方式。
基于需求的覆盖依据：使用用例、需求、用例流或测试条件作为全部测试项的评测方法，这种方法可在测试设计中应用。
基于代码的覆盖将生成的代码作为全部测试项，并评测测试中执行的代码特征（如执行的代码行或遍历的分支数）。这种覆盖评测方法只有当代码生成后才能够实施。
确定要使用的方法，并指明如何收集评测结果、如何解译数据及如何在流程中运用指标。
生成并分发测试覆盖报告
测试计划中确定了时间表，说明了何时生成、何时分发测试覆盖报告。这些报告至少应分发给以下角色：

所有测试角色
开发人员代表
股东代表
涉众代表

实施测试：

活动：实施测试**目的** 创建或生成可复用的测试脚本 维护从测试实施工件到相关的测试用例及用例或测试需求的可追踪性
步骤 记录、生成或通过编程创建测试脚本 确定设计与实施模型中的测试专用功能 建立外部数据集

输入工件：	**生成工件：**
补充规约 编程指南 设计指南 设计模型 设计类和包 软件构架文档 构件 测试模型 测试用例 测试构件 测试过程 测试脚本 工作量分析文档 实施模型 软件工作版本	完整的测试脚本 测试构件

角色：测试设计员
指南： 指南：单元测试 指南：测试用例 指南：测试模型 指南：测试脚本 指南：测试数据 指南：工作量分析文档
工作流程明细： **核心工作流程：测试** 实施测试

记录、生成或通过编程创建测试脚本 

目的 创建或自动生成适当的测试脚本，以便按照预期的方式实施（并执行）测试用例和测试过程
工具向导： **Rational Robot™** 在 Rational Robot 中设置测试环境 使用 Rational Robot 创建测试脚本 **Rational TestManager™** 使用 Rational TestManager 评估基于需求的代码覆盖 **Rational TestFactory™** 在 Rational TestFactory 中设置测试环境 使用 Rational TestFactory 自动生成测试脚本 使用 Rational TestFactory 评测与评估 Rational Robot 脚本的基于代码的测试覆盖 **Rational Purify™** 使用 Rational Purify 检测运行时错误 **Rational PureCoverage™** 使用 Rational PureCoverage 评估代码覆盖 **Rational Quantify™** 使用 Rational Quantify 查找性能瓶颈 **Rational LoadTest™** 记录性能测试会话 制定表示性能测试工作量的性能测试进度

执行下列操作：

创建、生成或获取测试脚本

测试/调试测试脚本

复审和评估测试覆盖

创建、生成或获取测试脚本

对于测试模型中的每个结构化测试过程，需创建或生成至少一个测试脚本。

在创建、生成或获取测试脚本时，应考虑以下方面：

尽量增大测试脚本的复用程度

尽量减小测试脚本的维护程度

使用现有脚本（如果可行）

使用测试工具（而不通过编程）创建测试脚本（如果可行）

以最稳定的方法（如通过对象名或鼠标单击）访问应用程序 GUI 对象和操作

要创建、生成或获取测试脚本，请执行以下步骤：

复审现有的测试脚本，了解其使用潜力

设置测试环境（包括所有硬件、软件、工具、数据与应用程序工作版本）

将测试环境初始化（以确保环境处于适当的测试状态或条件）

创建或获取测试脚本：

记录/获取：对于各个结构化测试过程，在执行测试过程以创建新的测试脚本时，应遵循结构化测试过程中所确定的步骤/操作，并使用适当的记录技术来尽量增大复用性并减小维护程度。

修改现有脚本：手工编辑现有脚本，或按照上述记录说明删除不需要的指令，并重新记录新的指令

编程：对于每个结构化测试过程，使用适当的编程技术来生成指令
要自动生成测试脚本，请参考具体的测试脚本生成工具。
继续创建、生成或获取代码，直至创建了所期望的/需要的测试脚本
根据需要修改这些测试脚本（测试模型中有所说明）
测试/调试测试脚本
在完成测试脚本的创建、生成或获取时，应该对测试脚本进行测试/调试，以确保这些测试脚本能正确地实施和执行测试。执行本步骤所用的软件工作版本应与创建/获取测试脚本所用的版本相同。
要测试/调试测试脚本，请执行以下步骤：
设置测试环境（如果需要）
重新将环境初始化
执行测试脚本
评估结果
确定下一步应执行的操作：
得到需要/期望的结果：无需执行操作
得到意外结果：确定问题的原因并予以解决
复审和评估测试覆盖
在完成测试脚本的创建、生成或获取时，应生成测试覆盖报告，以核实测试脚本已实现预期的测试覆盖。

确定设计与实施模型中的测试专用功能

目的
指定实施或执行测试所需的软件支持功能方面的需求

确定设计模型与实施模型中应包括的测试专用功能。测试专用功能最常用于集成测试中，因为集成测试需要为未包括与未实施的构件或系统提供桩模块或驱动程序。
有两种形式的桩模块或驱动程序：
只是“哑元”的桩模块和驱动程序，除了输入特定值或返回预定义值以外，它们不具有任何其他功能。
更加智能化的桩模块和驱动程序，它们可以模拟较为复杂的行为。
第二种形式应慎重使用，因为它需要较多的资源才能实施。所以，务必要在附加的价值（通过创建复杂的桩模块/驱动程序）和实施与测试桩模块/驱动程序所需的工作量之间保持平衡。
构建测试专用构件的其他原因包括：
在没有测试自动化工具的情况下实现回归测试过程的自动化，或
构建测试对象与测试自动化工具之间的接口。
按照对（移交给设计员与实施员的）设计与实施模型的需求，描述您的结果。

建立外部数据集

目的
为了创建和维护数据，应将数据保存在测试脚本的外部，由测试脚本在执行测试时使用。

外部数据集以如下方式向测试提供值：
数据位于测试脚本之外，从而消除了测试脚本中的硬编码引用
外部数据可以很容易地进行修改，修改后测试脚本不会受到影响（或只受到很小的影响）
其他测试用例可以很容易地添加到测试数据中，添加后无需修改测试脚本（或仅作很小的修改）

外部数据可以由许多测试用例共享
外部数据集可以包括用于控制测试脚本的数据值（有条件的分支逻辑）
创建/维护外部数据集
要创建外部数据集，请执行以下步骤：
复审测试模型、测试用例和结构化测试过程
使用适当的工具和方法创建数据集
修改测试脚本以便使用数据集
测试/调试测试脚本
在完成测试脚本的创建、生成或获取时，应该对测试脚本进行测试/调试，以确保这些测试脚本能正确地实施和执行测试。执行本步骤所用的软件工作版本应与创建/获取测试脚本所用的版本相同。
要测试/调试测试脚本，请执行以下步骤：
设置测试环境（如果需要）
重新将环境初始化
执行测试脚本
评估结果
确定下一步应执行的操作：
得到需要/期望的结果：无需执行操作
得到意外结果：确定问题的原因并予以解决

评估测试：

活动：评估测试**目的** 评估测试结果并记录变更请求。 计算并交付测试的主要评测方法。 生成测试评估摘要。
步骤 分析测试结果并提交变更请求 评估基于需求的测试覆盖 评估基于代码的测试覆盖 分析缺陷 确定是否达到了测试的完成标准和成功标准 生成测试评估摘要

输入工件：	**生成工件：**
测试计划 测试用例 测试结果	测试评估摘要 变更请求

角色：测试设计员

工作流程明细：
核心工作流程：测试 评估测试 **核心工作流程：项目管理** 管理迭代

分析测试结果并提交变更请求

目的 确定并评估预期结果和实际结果间存在的差异 将变更请求信息输入用于评估、管理与解决的跟踪工具
工具向导： **Rational LogViewer™:** 使用 Rational LogViewer 评估测试的执行情况 **Rational TestFactory™:** 使用 Rational TestFactory 评估执行一组测试脚本的结果 **Rational Purify™:** 使用 Rational Purify 检测运行时错误 **Rational Quantify™:** 使用 Rational Quantify 查找性能瓶颈 **Rational PureCoverage™:** 使用 Rational PureCoverage 评估代码覆盖 **Rational ClearQuest™:** 使用 Rational ClearQuest 提交变更请求（缺陷） **Rational LoadTest™:** 监测性能测试进度 报告性能测试

在测试执行活动中，测试结果要经过复审以确保测试已执行完全，并确保报告的测试结果没有受到非测试对象因素的影响。

在此活动中，经过分析的测试结果可以确定预期测试结果与实际测试结果之间存在的细微差异。这些差异表明了测试对象中存在潜在缺陷，应将差异作为变更请求输入跟踪系统,下一步则应采取相应的纠正操作。

复审各个已报告的测试故障，并确定将变更请求输入相应的跟踪系统所需的信息。输入的信息应准确、适当、并满足既定的变更请求跟踪标准或指南。

评估基于需求的测试覆盖

目的 确定是否实现了要求的（或适当的）基于需求的测试覆盖
工具向导： **Rational TestManager：** 使用 Rational TestManager 复审和评估基于需求的测试覆盖 **Rational TestFactory：** 使用 Rational TestFactory 评估测试覆盖

要评估基于需求的测试覆盖，您应复审测试结果并决定：

此次迭代中执行的基于需求的测试（测试用例）的数量与测试对象的总测试数量的比例。

成功执行的测试用例的比例。

目的是要确保要在本次迭代中进行的基于需求的测试能够百分之百成功执行。如果这是不可能或不可行的，则应确定一个不同的测试覆盖标准，该标准的基础可以是：

风险或优先级
可接受的覆盖百分比
请参见测试简介中的“测试的主要评测方法”。
在测试评估报告中记录本次迭代的结果。

评估基于代码的测试覆盖

目的 确定是否实现了要求的（或适当的）基于代码的测试覆盖。
工具向导： **Rational TestFactory:** 使用 Rational TestFactory 评估测试覆盖 使用 Rational TestFactory 评估执行一组测试脚本的结果 **Rational PureCoverage：** 使用 Rational PureCoverage 评估代码覆盖

要评估基于代码的测试覆盖，您应复审测试结果并决定：
在本次迭代的测试期间执行的代码（如代码行或语句）与测试对象中总代码的比例。
目的是要确保要在本次迭代中测试的代码百分之百成功执行。如果这是不可能或不可行的，则应确定一个不同的测试覆盖标准，该标准的基础可以是：
风险或优先级
可接受的覆盖百分比
请参见“概念：测试的主要评测方法”。
在测试评估报告中记录本次迭代的结果。

分析缺陷

目的 评估缺陷并推荐适当的后续活动 撰写客观报告，传达测试结果
工具向导 使用 Rational ClearQuest 报告缺陷趋势和状态

要分析缺陷，应将复审和分析所选评测方法作为缺陷分析战略的一部分。最常用的缺陷评测方法包括（通常以图的形式显示）：
缺陷密度 - 缺陷的数量以一个或两个缺陷属性（如状态或严重性）的函数显示。
缺陷趋势 - 缺陷数目以随时间变化的函数表示。
缺陷龄期 - 是特殊的缺陷密度报告，它通过缺陷在一定时间内保持某特定状态（打开的、新的、待测试的缺陷等）的函数表示。
将本次迭代的评测方法与先前各次迭代的分析结果进行比较，判断缺陷的走势。
建议用图来显示结果。
请参见“概念：测试的主要评测方法”。

确定是否达到了测试的完成标准和成功标准

目的 判定测试的执行是否完全、是否可接受 确定适当的后续测试活动

复审测试计划中制定的测试战略。应根据测试覆盖和/或缺陷评估结果说明测试标准。请检验测试结果、缺陷与缺陷分析，并判断是否已达标。
如果未达标，请参考以下备选方案：
收集进一步的信息：
另行撰写报告，如不同的缺陷密度报告
通过研究流程，判断意外条件是否导致背离已确定的测试标准，并在这一新信息的基础上再次评估标准
建议安排进一步测试：
实施新测试以进一步执行测试用例
实施新测试以扩大测试覆盖面
修改测试标准：
复审并评估测试后变更标准会带来的风险
确定满足测试标准的软件子集，并决定是否可以部署该子集
请参见测试简介中的“测试的主要评测方法”。

生成测试评估摘要 

目的 撰写客观报告，传达测试结果、主要评测方法与测试建议

评估测试活动中的最后一步是撰写测试评估摘要。 此活动通过以下方式进行：将上述信息编入一个报告，然后将其分发给相应的角色以便复审。

制订测试指南：

活动：制定测试指南**目的** 制定测试指南。	
步骤 定制测试指南 记录决定	
输入工件： 测试指南 开发案例 工具	**生成工件：** 测试指南
频率：在测试开始之前。	
角色：测试设计员	
工作流程明细： **核心工作流程：环境** 准备迭代指南	

定制测试指南 

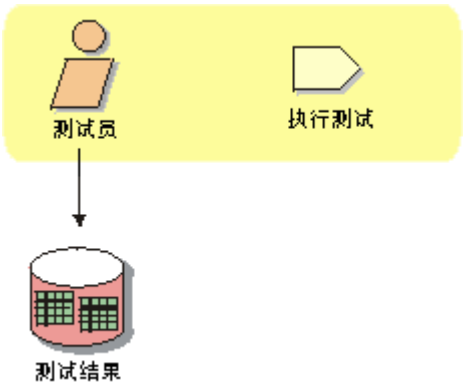
您最好定制测试指南，在其中包含与项目有关的领域。请参见测试指南中的“定制”部分。开发案例将作为一种输入来说明项目将如何进行测试。

记录决定 

在测试开始前，您应作出关于测试指南与策略的总体决策，并将这些决定记录在测试指南中。

测试员：

测试员负责执行测试，其职责包括：
设置和执行测试
评估测试执行过程并修改错误



人员配备

测试员应具备的知识和技能可能会因为他们所执行的测试类型和/或测试阶段的不同而有所差异。例如，在执行性能测试或集成阶段的测试时，需要更高级的技能。在执行功能测试或系统测试阶段的测试时，则不需要太高级的技能。
以下是测试员所需知识和技能的一些标准：
高级测试员：
了解系统或所测试的应用程序
了解联网和系统构架
了解测试及测试自动化工具
具备诊断和解决问题的技能
编程技能（必备）
初级测试员：
了解系统或所测试的应用程序
了解测试及测试自动化工具
具备诊断和解决问题的技能
编程技能（最好具备）

执行测试：

目的 执行测试并获取测试结果。
步骤 执行测试过程 评估测试的执行情况 核实测试结果

恢复暂停的测试	
输入工件： 构件 开发基础设施 实施子系统 软件工作版本 测试用例 测试模型 测试过程 测试脚本 测试构件 测试子系统	**生成工件：** 测试结果
角色：测试员	

工作流程明细：

核心工作流程：实施

集成每个子系统

集成系统

核心工作流程：测试

在集成测试阶段执行测试

在系统测试阶段执行测试

核心工作流程：部署

管理验收测试

核心工作流程：配置与变更管理

管理变更请求

执行测试过程

目的

执行测试过程（对于自动测试，则执行测试脚本）

工具向导：

Rational Robot™

在 Rational Robot 中设置测试环境

使用 Rational Robot 执行测试

Rational TestFactory™

使用 Rational TestFactory 执行一组测试脚本

Rational LoadTest™

执行性能测试进度

执行测试时应遵循以下步骤：

设置测试环境，确保所需的全部构件（硬件、软件、工具、数据等）都已实施并处于测试环境中。

将测试环境初始化，以确保所有构件都处于正确的初始状态，可以开始测试。

执行测试过程。

注：测试过程的执行方式将依据测试是自动测试还是手工测试而有所不同。

自动测试：执行在实施测试活动中创建的测试脚本。
手工测试：按照在设计测试活动中制定的结构化测试过程来手工执行测试。

评估测试的执行情况 

目的 确定是将测试执行完毕还是暂停测试 确定是否需要纠正措施
工具向导： **Rational LogViewer™:** 请参见工具向导：使用 Rational LogViewer 评估测试的执行情况中的“评估测试的执行情况”部分 **Rational TestFactory:** 使用 Rational TestFactory 评估执行一组测试脚本的结果 使用 Rational TestFactory 评测与评估 Rational Robot 脚本的基于代码的测试覆盖 使用 Rational TestFactory 评估测试覆盖 使用 Rational TestFactory、Rational Robot 和 Rational LogViewer 分析自动生成的测试脚本 **Rational LoadTest:** 监测性能测试进度

测试执行活动结束或终止时，以下两种情况之一会出现：
正常终止：所有测试过程（或脚本）按预期方式执行至结束。
如果测试正常结束，则继续核实测试结果：
异常或提前结束：测试过程（或脚本）没有按预期方式执行或没有完全执行。当测试异常终止时，测试结果可能不可靠。在执行任何其他测试活动之前，应确定并解决异常/提前终止的原因，然后重新执行测试。
如果测试异常终止，则继续恢复暂停的测试。

核实测试结果 

目的 确定测试结果是否可靠 针对测试结果表明的测试工作或测试工件中存在的缺陷，确定合适的纠正措施
工具向导： 请参见使用 Rational LogViewer 评估测试的执行情况中的“核实测试结果”和“调查意外结果”部分

测试完成后，应当复审测试结果以确保测试结果可靠，确保所报告的故障、警告或意外结果不是（对测试对象的）外部影响（例如，不正确的设置或数据等）造成的。
在测试过程和测试脚本完全执行时所报告的故障中，最常见的故障及其纠正操作为：
测试核实故障 - 通常发生在实际结果与预期结果不匹配时。验证所用的核实方法仅侧重于基本项与/或特征，并在必要时进行修改。
意外的 GUI 窗口 - 发生这种情况有好几种产生原因。最常见的原因是当前的 GUI 窗口并不是预期的窗口，或所显示的 GUI 窗口数目大于预期的数目。确保为正确执行测试而设置了测试环境并对其进行了初始化。
GUI 窗口遗失 - 如果某个 GUI 窗口应该可用（不一定是当前窗口）但实际上却不可用，则应记

录该故障。确保为正确执行测试而设置了测试环境并对其进行了初始化。确保实际遗失的窗口已从测试对象中删除。
如果所报告的故障是在测试工件中确定的错误导致的，或者是测试环境的问题造成的，则应当采取适当的纠正措施进行纠正，然后重新执行测试。有关其他信息，请参见下面的“恢复暂停的测试”。
如果测试结果表明故障确实是由测试对象造成的，就可认为执行测试活动已完成。

恢复暂停的测试

目的 确定适当的纠正措施，以便恢复暂停的测试 纠正问题，恢复并重新执行测试
工具向导： 请参见工具向导：使用 Rational LogViewer 评估测试的执行情况中的“恢复暂停的测试”部分

暂停的测试主要有两种类型：
致命错误 - 系统故障（网络故障、硬件崩溃等）。
测试脚本命令故障 - 针对自动测试，指测试脚本无法执行某条命令（或代码行）。
这两种类型的测试异常终止可能会表现出相同的故障现象：
当执行测试脚本时，出现许多意外的操作、窗口或事件。
测试环境没有响应或处于非理想状态（如悬挂或崩溃）。
要恢复暂停的测试，请执行如下步骤：
确定问题的实际原因
纠正问题
重新设置测试环境
重新初始化测试环境
重新执行测试

测试结果：

测试结果	该工件包括测试执行期间捕获的数据存储库，用于计算多种测试的主要评测方法。
角色：	测试员
详细信息：	概念：测试的主要评测方法 概念：测试自动化和工具 工件：测试计划
目的 提要 时机 职责 定制	
活动的输入： 评估测试	**活动的输出：** 执行测试 执行单元测试

目的

测试结果是在测试执行期间捕获的。它用作评估测试和计算测试的主要评测方法的输入。

提要

测试结果中包含的数据，将由于在捕获数据的测试执行期间所应用的技术和工具的不同而异。但至少应捕获以下数据，以便将其用于复审和评估：
测试结果标识符（即这些结果区别于其他测试结果的 ID）
时间、日期、测试员姓名和环境信息（如 O/S、机器特征等等）
测试对象的特定标识（如版本、对象、文件等等）
预期执行（并追踪到测试需求）的测试用例
已执行（并追踪到测试需求）的测试用例
要执行的测试对象的评测范围
已执行的测试对象的评测范围
指定事件序列的响应时间
包含主角和测试对象之间的会话详细信息的追踪数据，和/或包含测试对象中的对象之间的会话详细信息的追踪数据
每个已执行测试用例的实际结果
预期结果和实际结果之间的差异
每个所执行的测试用例通过/失败的指标
任何意外或反常的结果或行为

时机

测试结果是在测试执行期间捕获的。因为在开发生命周期内将多次执行测试，所以必须以能够对每一测试执行实例都可分别复审和评估的方式，捕获和存储测试结果。

职责

测试员负责测试结果工件的完整性，确保捕获数据的精确和完全。

定制

如果正在使用自动测试工具，例如在 Rational Studio 或 Test Studio 中的工具，那么以上列出的很多数据（甚至更多数据）都可以自动捕获和存储。出于报告的目的，这些工具还提供已保存和自定义数据视图。