Chapter 3 Quality Assurance (QA)

- QA as Dealing with Defect
- Defect Prevention
- Defect Detection and Removal
- Defect Containment

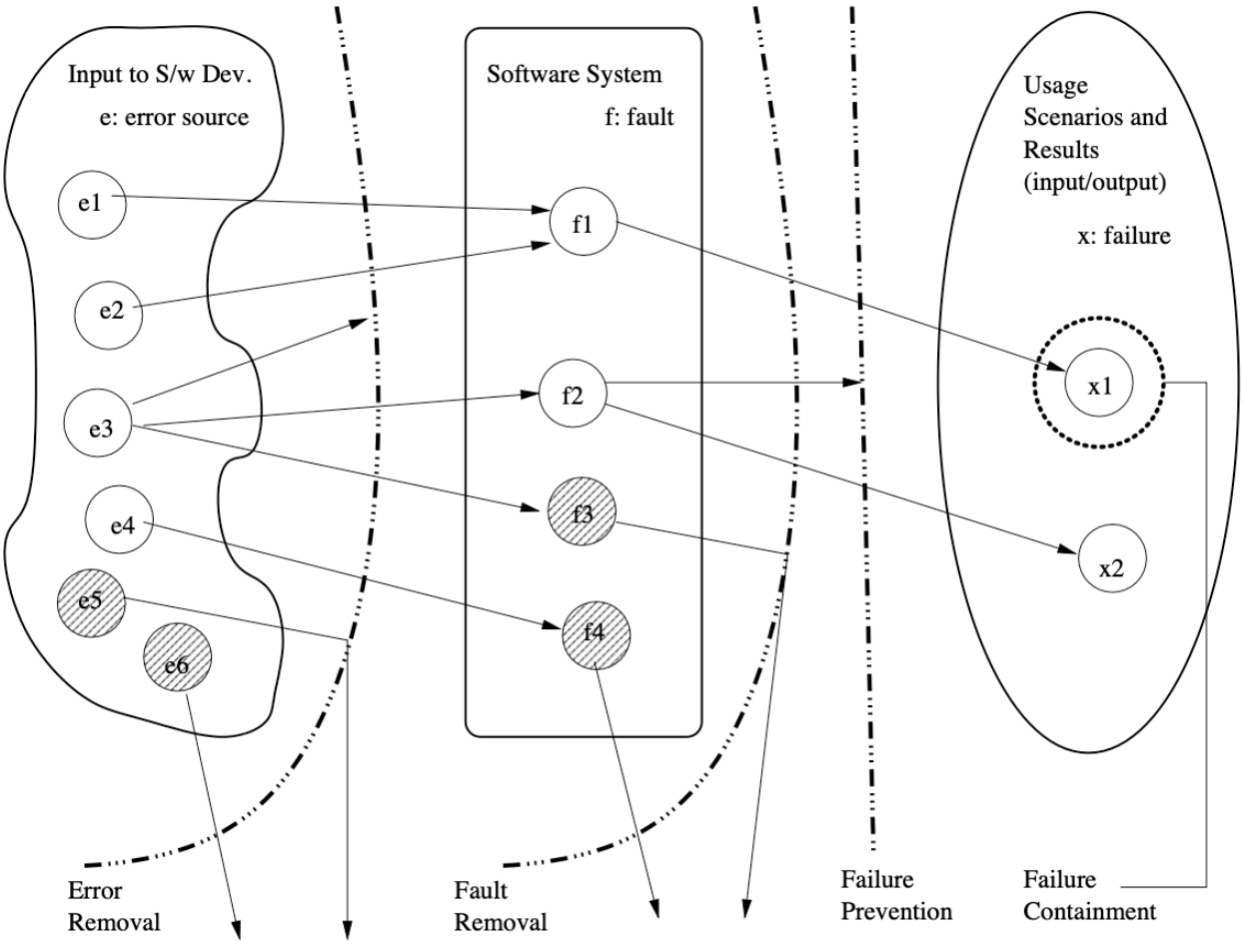
3.1 Defect vs. QA

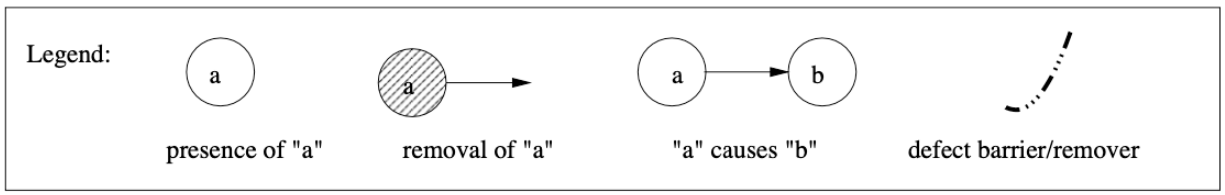
- QA: quality assurance
  - focus on correctness aspect of Q
  - QA as dealing with defects
    - post-release: impact on consumers
    - pre-release: what producer can do
  - what: testing & many others
  - when: earlier ones desirable (lower cost) but may not be feasible
  - how ⇒ classification below
- How to deal with defects:
  - prevention
  - removal (detect them first)
  - containment

> QA（质量保证）：
>
> - 关注质量的正确性方面
> - 将QA视为处理缺陷的过程
>   - 发布后：对消费者的影响
>   - 发布前：生产者可以做什么
> - 什么：测试及许多其他活动
> - 何时：较早的介入更为理想（成本较低）
>
>   但可能不可行
> - 如何 ⇒ 下面的分类
>
> 如何处理缺陷：
>
> - **预防**：通过提前的设计和编码标准、教育和工具来避免缺陷的产生。
> - **移除**（首先要检测到它们）：通过各种形式的测试、审查和分析来识别并修复缺陷。
> - **隔离**：当不能立即修复缺陷时，采取措施限制缺陷影响的范围和严重程度。

> 解释与例子
>
> - **预防**：这是最理想的处理缺陷的方法。例如，通过使用静态代码分析工具来识别潜在的编码问题，或者通过实施编码最佳实践和标准来减少错误的引入。此外，培训开发人员了解常见的软件开发陷阱也是预防缺陷的一种有效方法。
> - **移除**：这通常涉及到软件测试阶段，包括单元测试、集成测试和系统测试等，以发现并修复代码中的错误。代码审查和对等审查也是在软件发布前发现和移除缺陷的有效手段。
> - **隔离**：在某些情况下，可能无法立即修复发现的缺陷，特别是在软件即将发布时。在这种情况下，可能需要采取措施来隔离这些缺陷，比如通过软件的配置选项禁用引发缺陷的功能，或者在软件的发布说明中明确指出已知的限制和问题，从而减少对最终用户的影响。
>
> 质量保证的目标是通过这些方法最大限度地减少软件产品中的缺陷，从而提高产品的整体质量和可靠性。通过在软件开发生命周期的早期阶段采取预防和移除措施，可以显著降低修复缺陷的成本和复杂性。

3.2 QA Classification





- Fig 3.1 above (p.30): QA as barriers
  - dealing with errors, faults, or failures
  - removing or blocking defect sources
  - preventing undesirable consequences

> - **处理错误、故障或失败**：这意味着在软件开发的各个阶段，如需求分析、设计、编码、测试等，识别和解决可能导致软件行为不符合预期的问题。例如，在需求分析阶段，通过仔细审查和验证需求来发现并修正可能导致理解错误的模糊或不完整的需求。
> - **移除或阻止缺陷源**：这涉及到采取措施直接对抗可能引入缺陷的原因。比如，通过实施代码审查来识别和修正编码阶段可能引入的错误，或者使用自动化测试工具来检测和修复潜在的故障。
> - **防止不良后果**：最终目的是防止缺陷对用户造成影响，这可能包括软件崩溃、数据丢失或安全漏洞等。为了实现这一点，可以在软件发布前进行全面的测试，以确保所有已知的缺陷被修复或至少被识别和记录下来，以便用户了解。
>
> 例如，假设一个团队正在开发一个在线购物平台。在开发过程中，QA团队使用各种策略和工具，如单元测试来检测代码中的故障，安全测试来识别潜在的安全问题，以及性能测试来确保平台在高流量下仍能正常运行。通过这些措施，团队能够在软件发布之前识别并解决多数问题，从而减少最终用户遇到问题的风险。

3.3 Error/Fault/Failure & QA

- Preventing fault injection
  - error blocking (errors/⇒ faults)
  - error source removal
- Removal of faults (pre: detection)
  - inspection: faults discovered/removed
  - testing: failures trace back to faults
- Failure prevention and containment:
  - local failure/⇒ global failure
    - via dynamic measures to tolerate faults
  - failure impact↓ ⇒ safety assurance

> - 防止故障注入
>   - 阻止错误转化为故障
>   - 移除错误源
> - 移除故障（检测前提下）
>   - 审查：发现/移除故障
>   - 测试：追溯失败到故障的原因
> - 失败预防和控制：
>   - 局部失败不转化为全局失败
>   - 通过动态措施容忍故障

- 降低失败影响 ⇒ 确保安全

> 防止故障注入
> 阻止错误转化为故障：这意味着在软件开发过程中采取措施，防止设计或编程错误演变成软件的内部问题。例如，使用静态代码分析工具可以在编码阶段发现潜在的编码错误，并阻止这些错误成为实际的故障。
>
> 移除错误源：涉及到识别和解决可能导致错误的根本原因，如通过改进需求规格的清晰度和完整性，减少需求理解上的错误。
>
> 移除故障
> 审查：软件审查或代码审查是一种有效的故障发现和移除技术，它允许团队成员在早期阶段手动检查软件设计或代码，以发现并修正潜在的问题。
>
> 测试：软件测试是追踪故障的另一种方式，通过在运行时识别软件的异常行为（失败），然后追溯这些异常到具体的故障源头。
>
> 失败预防和控制
> 局部失败不转化为全局失败：通过设计软件以便在发生局部故障时不会影响整个系统的稳定性。例如，通过使用容错技术，如冗余和故障隔离，确保系统的关键部分在面对故障时仍能继续运行。
>
> 降低失败影响：采取措施减少故障可能导致的损害，从而确保系统的安全。这可能包括实施备份和恢复策略，或者在关键系统中引入紧急停机功能。
>
> 例如，假设在开发一个在线支付系统时，团队实施了代码审查和广泛的测试策略来识别和修复故障。同时，系统设计包括了故障隔离机制，确保即使支付处理组件出现故障，也不会影响到账户管理功能的运行，从而减少了单一故障点导致的风险，并提高了整个系统的可靠性和安全性。

3.4 Defect Prevention Overview

- Error blocking
  - error: missing/incorrect actions
  - direct intervention to block errors
    ⇒ fault injections prevented
  - rely on technology/tools/etc.
- Error source removal
  - root cause analysis
    ⇒ identify error sources
  - removal through education/training/etc.
- Systematic defect prevention via process improvement.
- Details: Chapter 13.

> - 阻止错误
>   - 错误：缺失或不正确的行为

    - 直接干预以阻止错误 ⇒ 防止故障注入
    - 依赖技术/工具等。
  - 移除错误源
    - 根本原因分析 ⇒ 识别错误源
    - 通过教育/培训等移除
  - 通过过程改进实现系统性缺陷预防。
  - 详细内容：第13章。

> 解释与例子
> 阻止错误
> 在软件开发过程中，阻止错误是一种直接干预的方法，旨在识别和阻断可能导致故障注入的错误行为。这通常涉及使用技术和工具，比如静态代码分析工具，它可以在代码提交前自动检测潜在的编码错误，从而防止这些错误转化为软件中的故障。
>
> 移除错误源
> 通过根本原因分析（Root Cause Analysis, RCA）来识别导致错误的深层次原因，然后通过教育和培训等手段来移除这些错误源。例如，如果发现软件缺陷频繁源于对某个特定技术或工具的误用，那么可以组织针对该技术或工具的培训课程，提高开发团队的使用技巧和理解度。
>
> 系统性缺陷预防
> 这种方法涉及到对软件开发过程本身进行持续的改进和优化，以减少缺陷的产生。这可以通过引入更好的开发实践、改进项目管理方法、采用新的技术工具等手段实现。目标是创建一个更加稳健、可靠且易于维护的软件开发环境。
>
> 例如，一个软件开发团队可能发现其缺陷主要源于需求不清晰或频繁变更。作为对策，团队决定改进需求管理过程，引入更严格的需求审查会议和更灵活的变更管理流程，从而减少因需求问题导致的错误和故障。
>
> 通过这些方法，软件团队不仅能够减少缺陷的发生，还能提升软件开发过程的整体质量和效率。

3.5 Formal Method Overview

- Motivation
  - fault present:
    - revealed through testing/inspection/etc.
  - fault absent: formally verify.

    (formal methods ⇒ fault absent)
- Basic ideas
  - behavior formally specified:
    - pre/post conditions, or
    - as mathematical functions.
  - verify “correctness”:

        - intermediate states/steps,
        - axioms and compositional rules.
    - Approaches: axiomatic/functional/etc.
        - Details: Chapter 15.

> 形式方法概述
>
> - 动机
>   - 存在的故障：
>   - 通过测试/审查等揭示。
>   - 不存在的故障：正式验证。 (形式方法 ⇒ 故障不存在)
> - 基本思想
>   - 行为被正式规定：
>   - 前置/后置条件，或
>   - 作为数学函数。
>   - 验证“正确性”：
>   - 中间状态/步骤，
>   - 公理和组合规则。
>   - 方法：公理化/函数式等。
>   - 详情：第15章。

> 解释与例子
>
> 动机
>
> 形式方法的动机是提供一种比传统测试和审查更系统、更可靠的方式来验证软件的正确性。在软件开发中，尽管测试和审查是识别故障的重要手段，但它们无法保证找出所有的故障。形式方法通过使用数学和逻辑来证明软件的某些属性，从而提供了一种确保软件不存在特定类型故障的手段。
>
> 基本思想
>
> - **行为被正式规定**：软件的行为通过前置条件（函数执行前必须为真的断言）和后置条件（函数执行后必须为真的断言），或者将软件功能定义为数学函数的方式进行规定。这些规定提供了一个精确的、非歧义的软件行为描述。
> - **验证“正确性”**：通过证明软件从初始状态到终止状态的每个中间步骤都符合预定义的规则和逻辑，来验证软件的正确性。这通常涉及到使用公理（被认为是自明真理的陈述）和组合规则（如何从已知事实推导出新事实的规则）。
>
> 方法
>
> - **公理化方法**：通过定义一组公理和推理规则来描述软件的行为，并通过逻辑推理来证明软件满足其规格说明。
> - **函数式方法**：将软件视为数学函数，使用数学证明技术来验证软件行为符合预期。

> 例如，使用形式方法验证一个排序算法的正确性，首先正式定义排序函数的预期行为（例如，输入是一个整数列表，输出是一个按升序排列的列表，且输出列表中的元素与输入列表相同）。然后，使用数学证明来证明无论输入列表如何，该排序算法都能产生符合预期行为的输出列表。
>
> 形式方法提供了一种强有力的工具，可以在软件开发的早期阶段就确保软件设计的正确性，减少后期测试和维护的成本和复杂性。然而，由于它们的应用通常需要专业的知识和较高的时间成本，因此主要用于对正确性要求极高的领域，如航空航天、核能控制系统等。

3.6 Inspection Overview

- Artifacts (code/design/test-cases/etc.) from req./design/coding/testing/etc. phases.
- Informal reviews:
  - self conducted reviews.
  - independent reviews.
  - orthogonality of views desirable.
- Formal inspections:
  - **Fagan inspection** and variations.
  - process and structure.
  - individual vs. group inspections.
  - what/how to check: techniques .
- Details: Chapter 14.

> - 来自需求、设计、编码、测试等阶段的产物（代码/设计/测试用例等）。
> - 非正式审查：
>   - 自我进行的审查。
>   - 独立审查。
>   - 观点的正交性是可取的。
> - 正式审查：
>   - Fagan审查及其变体。
>   - 流程和结构。
>   - 个人与团体审查。
>   - 如何检查：技术。
> - 详细内容：第14章。

> 解释与例子
>
> 非正式审查

> 非正式审查是一种较为灵活的审查方式，可以是开发人员对自己的代码进行自我审查，或者是由项目内部或外部的独立个体进行审查。这种审查的目的是从不同的视角识别潜在的问题和改进点，以提高软件质量。正交性的观点，即从不同的角度和专业知识进行审查，是非常有益的，因为它可以揭露那些可能被单一视角忽略的问题。
>
> 正式审查
>
> 正式审查，如Fagan审查，是一种结构化的审查过程，旨在系统地检查软件开发的各个产物，如代码、设计文档和测试用例等。这种审查通常涉及到明确的步骤、角色和责任，以及记录和追踪发现的问题。
>
> - **流程和结构**：正式审查具有定义良好的流程，包括准备、会议、审查和后续改进等阶段。
> - **个人与团体审查**：审查可以由个人独立完成，也可以通过团体会议的形式进行，团体审查有助于集中不同专家的知识和经验，提高发现问题的可能性。
> - **如何检查**：审查过程中，审查者会使用各种技术和检查清单来指导审查活动，确保系统性和全面性地覆盖所有重要方面。
>
> 例如，进行Fagan审查时，团队可能会聚集一组来自不同背景的人员（如开发人员、测试人员和设计师），共同审查软件设计文档。审查前，所有参与者都需要独立阅读文档，并准备好自己的反馈。在审查会议中，每个人都会提出自己的观点和发现的潜在问题，然后团队讨论这些问题并制定改进计划。
>
> 正式审查是提高软件质量、识别和修正问题的有效手段，尤其是在软件开发的早期阶段。通过结构化的审查过程，可以有效减少后期开发和维护成本。

3.7 Testing Overview

Product/Process characteristics:

- object: product type, language, etc.
- scale/order:

  unit, component, system, ...
- who: self, independent, 3rd party
- What to check:
  - verification vs. validation
  - external specifications (black-box)
  - internal implementation (white/clear-box)
- Criteria: when to stop?
  - coverage of specs/structures.
  - reliability ⇒ usage-based testing
  - Much, much more in Part II.

> 产品/过程特性：
>
> - 对象：产品类型、语言等。
> - 规模/顺序： 单元、组件、系统...

- 谁：自我、独立、第三方
- 检查内容：
  - 验证与确认
  - 外部规格（黑盒）
  - 内部实现（白盒/清晰盒）
- 准则：何时停止?
  - 规格/结构的覆盖。
  - 可靠性 ⇒ 基于使用的测试
  - 第二部分还有更多内容。

解释与例子

对象

测试的对象可以是软件开发的任何阶段产出，包括代码、设计文档、需求规格等。这些对象的特性，如产品类型（例如，Web应用、嵌入式系统）和编程语言（例如，Java、C++），会影响测试的方法和工具的选择。

规模/顺序

- **单元测试**：针对软件中的最小可测试单元（如函数或方法）进行的测试。
- **组件测试**：针对软件的单个组件或模块进行的测试。
- **系统测试**：对整个软件系统进行的综合测试，验证软件与其接口、硬件等其他元素的集成是否正确。

谁

测试可以由软件的开发者（自我测试）、同一组织内部但与项目开发团队不同的个体或团队（独立测试）或外部组织（第三方测试）执行。

检查内容

- **验证**：确保软件符合定义好的需求或规格。
- **确认**：确保软件满足用户的需求和期望。
- **外部规格测试（黑盒测试）**：不考虑软件内部结构和内部工作机制，只根据外部规格进行测试。
- **内部实现测试（白盒测试）**：基于软件内部的逻辑路径、代码结构等进行测试。

准则：何时停止?

确定测试何时结束是软件测试中的一个重要决策点，常见的准则包括：

- **规格/结构的覆盖**：确保所有定义的需求和软件结构都经过了测试。
- **可靠性**：基于使用的测试，即根据软件的实际使用情况或模拟使用情况进行测试，直到达到预定的可靠性目标。

> 例如，开发一个在线购物平台时，测试团队会进行单元测试来验证每个独立功能（如添加商品到购物车）的正确性；组件测试来验证各个服务（如支付服务、库存管理服务）的集成；系统测试来确保整个平台作为一个整体满足性能和安全性需求。测试过程中会使用黑盒测试来验证功能是否符合用户需求，同时使用白盒测试来确保代码内部结构的正确性和效率。测试持续进行，直到满足预定的覆盖率和可靠性标准为止。

3.8 Fault Tolerance Overview

- Motivation
  - fault present but removal infeasible/impractical
  - fault tolerance ⇒ contain defects
- FT techniques: break fault-failure link
  - recovery: rollback and redo
  - NVP: N-version programming
    - fault blocked/out-voted • Details: Chapter 16.
- Details: Chapter 16.

> 容错技术概述
>
> 动机
>
> - 存在的故障但移除不可行或不切实际
> - 容错 ⇒ 包含缺陷
>
> 容错技术：打破故障与失败之间的链接
>
> - **恢复**：回滚并重做
> - N版本编程（NVP): 通过多个版本的程序来确保至少有一个版本可以正确运行，故障被阻止或被多数表决排除。
>   - 详情：第16章。

> 解释与例子
>
> 动机
>
> 在软件系统中，总有一些故障是由于成本、时间或技术限制而难以或不可能完全移除的。在这种情况下，容错成为确保系统即使在部分组件发生故障的情况下仍能继续运行的关键策略。
>
> 容错技术
>
> - **恢复**：这种技术涉及到在检测到故障后，将系统状态回滚到一个已知良好的状态，然后重做操作。这是一种常见的技术，特别是在数据库管理系统中，通过事务日志来实现。
> - **N版本编程（NVP）**：这种方法通过开发同一个应用程序的多个功能上相互独立的版本，来提高系统的可靠性。当这些版本在运行时产生不一致的结果时，可以通过投票等机制来确定正确的结果。理论上，只要所有版本不会在相同的情况下失败，这种多样性就能提高系统的整体可靠性。

> 例子
>
> 假设有一个关键的金融交易系统，该系统必须确保高度的可靠性和正确性。使用**恢复**技术，系统可以在交易过程中的任何一点捕获快照，如果检测到故障（如数据不一致或系统崩溃），系统可以回滚到最近的一次快照并重新执行交易，以确保数据的一致性和完整性。
>
> 另一方面，通过实施**N版本编程**，金融机构可以开发几个独立团队编写的交易处理系统版本。这些版本独立运行，对于每个交易请求，它们可能会给出相同或不同的处理结果。然后，系统通过一个投票机制来决定最终的交易结果，从而在一个版本可能包含的故障面前提供额外的保护层。
>
> 这些容错技术提供了一种方式，以确保在面对不可避免的系统故障时，软件系统仍能继续提供服务，从而最大限度地减少潜在的服务中断和数据损失。

3.9 Safety Assurance Overview

- Extending FT idea for safety:
  - fault tolerance to failure “tolerance”
- Safety related concepts:
  - safety: accident free
  - accident: failure w/ severe consequences
- hazard: precondition to accident
- Safety assurance:
  - hazard analysis
  - hazard elimination/reduction/control
  - damage control
- Details: Chapter 16.

> 安全保障概述
>
> - 将容错思想扩展到安全领域：
>   - 从容错到“容忍”失败
> - 与安全相关的概念：
>   - 安全：无事故
>   - 事故：带有严重后果的失败
> - 危害：事故的前提条件
> - 安全保证：
>   - 危害分析
>   - 危害消除/减少/控制
>   - 损害控制
> - 详情：第16章。

> 解释与例子
> 将容错思想扩展到安全
> 容错技术旨在确保系统即使在部分组件失败的情况下也能继续运行。将这种思想扩展到安全领域，意味着不仅要确保系统的持续运行，还要确保系统运行的过程中不会发生严重后果的事故。
>
> 与安全相关的概念
> 安全指的是系统运行过程中没有发生事故的状态。
> 事故是指系统失败并导致严重后果，如人员伤亡、财产损失或环境破坏。
> 危害是导致事故的一系列条件或活动的组合。
> 安全保证
> 安全保证涉及到一系列活动，旨在识别潜在的危害、分析它们可能导致的事故，以及采取措施消除或减少这些危害，从而控制损害。
>
> 危害分析：这是识别和评估系统中可能导致事故的危害的过程。它涉及到系统的详细研究，以识别可能的故障模式及其对安全的影响。
>
> 危害消除/减少/控制：这涉及到采取措施来消除或减少已识别危害的潜在影响，或者在事故发生时控制损害。这些措施可以包括技术解决方案、过程改进、安全培训等。
>
> 例子
> 假设在一个化工厂中，通过危害分析识别了一种可能的危害——化学物质泄漏，这可能导致火灾或爆炸。为了消除或减少这种危害，工厂可能采取多种措施，如安装更先进的泄漏检测系统、改进化学物质的储存和处理程序、对员工进行安全操作培训等。此外，工厂还可能制定紧急响应计划，以便在化学物质泄漏发生时迅速采取行动，减少对人员和环境的损害。
>
> 通过这些安全保证活动，可以显著提高系统的安全性，减少事故的发生概率，以及在事故不可避免发生时减轻其后果。