ICS 35.080
CCS L 77

T/CICC

中国指挥与控制学会团体标准

T/CICC 35016—2025

复杂软件系统维护性技术要求

Technical requirments for maintainability of complex software systems

2025-11-20 发布　　2025-11-20 实施

中国指挥与控制学会　发布

目　　次

前　　言

本文件按照GB/T 1.1-2020《标准化工作导则 第1部分：标准化文件的结构和起草规则》的规定编写。

请注意本文件的某些内容可能涉及专利。本文件的发布机构不承担识别专利的责任。

本文件由中国指挥与控制学会提出并归口。

本文件起草参与单位：北京航空航天大学、杭州市北京航空航天大学国际创新研究院（北京航空航天大学国际创新学院）、中国船舶集团有限公司综合技术经济研究院、可靠性与环境工程技术国家级重点实验室、北京航空航天大学可靠性工程研究所、南京大学、中国科学院空间应用工程与技术中心、昆仑数智科技有限责任公司。

本文件主要起草人：杨顺昆、颜思玮、侯展意、刘泊江、吴梦丹、曾福萍、陈振宇、金旷宇、张亚铭、钟红恩、陶新、王志伟、王铁成、王亦风。

复杂软件系统维护性技术要求

1 范围

本标准规定了复杂软件系统维护性的定量和定性要求、支撑技术与方法以及全生命周期的过程与活动。

本文件适用于复杂软件系统在需求分析、设计与实现、测试、使用和维护更新阶段的维护性保证，可供相关行业组织和研究机构使用。

2 规范性引用文件

下列文件中的内容通过文中的规范性引用而构成本文件必不可少的条款。其中，注日期的引用文件，仅该日期对应的版本适用于本文件；不注日期的引用文件，其最新版本（包括所有的修改单）适用于本文件。

GB/T 14394-2008 计算机软件可靠性和可维护性管理

GB/T 20157-2006 信息技术 软件维护

GB/T 29834.1-2013 系统与软件维护性 第1部分：指标体系

GB/T 29834.2-2013 系统与软件维护性 第2部分：度量方法

GB/T 29834.3-2013 系统与软件维护性 第3部分：测试方法

GB/T 40473.8-2021 银行业应用系统非功能需求 第8部分：可维护性

GJB 438C-2021 军用软件开发文档通用要求

GJB 439A-2013 军用软件质量保证通用要求

GJB 841A-2024 通用质量特性问题报告、分析和纠正措施系统

GJB 2434A-2004 军用软件产品评价

GJB 5235A-2021 军用软件配置管理

GJB 6389-2008 军用软件评审

T/CFEII 0016-2023 汽车软件开发能力要求

QJ 2543A-2008 航天型号软件维护

3 术语与定义

GB/T 14394-2008、GB/T 20157-2006、GB/T 29834.1-2013和T/CICC 35008-2025术语和定义适用于本文件。

3.1

复杂软件系统 complex software system

由大量相互依赖、相互作用的软件组件（计算机程序、模块、服务等）通过复杂的逻辑和物理关系连接而成，并遵循严格的规程（流程、协议、策略）进行协同运作，需动态应对内外部软件、硬件与环

境的变化以实现复杂业务或关键领域目标的软件集成。其本质特征在于结构庞大、功能多样、环境多变、动态交互、任务复杂，通常具备多层次架构、模块协作、高度耦合以及较长的生命周期。

［来源：T/CICC 35008-2025，3.1］

3.2

软件可维护性 software maintainability

系统或软件或其部件能修改以排除故障、改进性能或其他属性或适应变更了的环境的容易程度。

［来源：GB/T 29834.1-2013，3.4］

3.3

软件可维护性大纲 software maintainability program

描述为保证软件满足规定的可维护性要求所采取的技术和管理方法的文档，典型地描述要做的工作、所需要的资源、使用的方法、采用的过程、要满足的进度表和项目组织方法。

［来源：GB/T 14394-2008，3.2］

3.4

维护性计划 maintainability plan

一种文档，其中记述了与软件有关的特定的维护性惯例、资源以及活动序列。

［来源：GB/T 20157-2006，4.4］

3.5

适应性维护 adaptive maintenance

在交付后执行的软件产品的修改，以保持这个软件产品可以在已变更或正在变更的环境中使用。

［来源：GB/T 20157-2006，4.1］

3.6

预测性维护 predictive maintenance

根据观测到的状况而决定的连续或间断进行的维护，以监测、诊断或预测构筑物、系统或部件的条件指标。

［来源：GB/T 40571-2021，3.5］

3.7

纠正性维护 corrective maintenance

软件产品交付后执行的反应性修改，以纠正发现的问题。

［来源：GB/T 20157-2006，4.3］

3.8

完善性维护 perfective maintenance

软件产品交付后为改进性能或维护性所作的修改。

［来源：GB/T 20157-2006，4.10］

3.9

预防性维护 preventive maintenance

软件产品交付后的修改,用来检测和纠正软件产品中的潜在故障,使其不致成为有效故障。

［来源：GB/T 20157-2006，4.11］

4 缩略语

MBSwE	基于模型的软件工程（Model-Based Software Engineering）
ORM	对象关系映射（Object-Relational Mapping）
RBAC	基于角色的访问控制（Role-Based Access Control）
NFS	网络文件系统（Network File System）
CI/CD	持续集成/持续交付（Continuous Integration / Continuous Delivery）
CVE	公开漏洞和暴露（Common Vulnerabilities and Exposures）
BIT	内置测试（Built-In Test）
OTA	空中下载技术（Over-The-Air Technology）

5 复杂软件系统维护性定量要求

根据GB/T 29834.1-2013第四章，系统与软件的维护性指标的定量要求宜从易分析性、模块化、规范性、易改变性、稳定性、可验证性和维护性的依从性七个方面来描述：

a） 易分析性：系统与软件对维护过程提供动态分析的支持能力；

b） 模块化：系统与软件为维护过程通过模块化对维护实施的支持能力；

c） 规范性：系统与软件对维护过程所涉及的代码、数据和文档等静态要素的可理解性的支持能力；

d） 易改变性：对系统与软件实施维护的容易程度的支持能力；

e） 稳定性：系统与软件在维护后，对防止因维护而带来系统意外的能力；

f） 可验证性：系统与软件对软件维护结果的可验证性的支持能力；

g） 维护性的依从性：软件产品遵循与维护性相关的标准或约定的能力。

维护性指标及其计算方法如表1所示：

表 1　维护性指标计算方法

序号	类别	指标	含义	计算方法	来源	备注
1	易分析性	活动的记录	系统状态记录的完整程度。	X = A/B A=在评审中已证实按照规定已实现数据记录的项数； B=在规格说明中定义的要记录的数据项数。	GJB 5236-2004	0 ≤ X ≤ 1，越接近1越好。
2	易分析性	诊断功能准备的情况	诊断功能准备的全面程度。	X = A/B A=在评审中已证实按照规定实现诊断功能的项数； B=要求的诊断功能数。	GJB 5236-2004	0 ≤ X ≤ 1，越接近1越好。
3	规范性（代码易读性）	注释的充分性	代码的注释行的数量能确保维护人员的理解。	X = A/B A=抽样模块中已加注释的方法个数； B=抽样模块中所有的方法个数。	GB/T 29834.2-2013	0 ≤ X ≤ 1，越接近1越好。
4	规范性（代码易读性）	注释的规范性	代码的注释是否规范、易于理解和分析。	X = A/B A=抽样注释中，准确、易理解的注释个数； B=抽样注释的个数。	GB/T 29834.2-2013	0 ≤ X ≤ 1，越接近1越好。
5	规范性（代码易读性）	代码的规范性	代码的编写遵从代码编写规范的程度。	X = A/B A=抽样代码中，遵守代码编写规范的代码行数； B=抽样代码的行数。	GB/T 29834.2-2013	0 ≤ X ≤ 1，越接近1越好。
6	规范性（文档维护指导性）	代码规范的符合性	代码的编写是否遵从相应的规范。	X = A/B A=程序中已经遵守的代码编写规范数； B=应遵守的代码编写规范数。	GB/T 29834.2-2013	0 ≤ X ≤ 1，越接近1越好。
7	规范性（文档维护指导性）	文档与软件的符合程度	文档与软件的实际功能间的一致程度。	X = A/B A=抽样模块中，有正确文档描述的模块个数； B=抽样模块个数。	GB/T 29834.2-2013	0 ≤ X ≤ 1，越接近1越好。
8	规范性（数据的规范性）	数据的规范性	符合预定义格式的数据量。	X = A/B A=符合规范格式的数据类型的个数； B=所有的数据类型的个数。	GB/T 29834.2-2013	0 ≤ X ≤ 1，越接近1越好。
9	易改变性	更改的记录	规格说明和程序模块的更改是否在代码的注释行中被恰当记录。	X = A/B A=在评审中已证实有更改注释内容的功能/模块的变化数； B=按源代码更改的功能/模块的总数。	GJB 5236-2004	0 ≤ X ≤ 1，越接近1越好。
10	稳定性	更改的影响	修改后发生不利影响的频率。	X = A/B A=检测到修改之后发生不利影响的次数； B=实施修改的次数。	GJB 5236-2004	0 ≤ X ≤ 1，越接近1越好。
11	稳定性	受到修改影响的范围	软件产品修改后，受到的影响大小。	X = A/B A=在评审中已证实受修改影响的变量数； B=变量的总数。	GJB 5236-2004	0 ≤ X ≤ 1，越接近0越好。
12	可验证性	内置测试功能的完整性	内置测试能力的完整程度。	X = A/B A=在评审中证实按照规定实现内置测试的功能数； B=要求内置测试的功能数。	GJB 5236-2004	0 ≤ X ≤ 1，越接近1越好。
13	可验证性	自主性	软件测试的独立程度。	X = A/B A=依赖于其他系统的已经用桩模块模拟的测试项数； B=依赖于其他系统的测试总数。	GJB 5236-2004	0 ≤ X ≤ 1，越接近1越好。

表 1　维护性指标计算方法（续）

序号	类别	指标	含义	计算方法	来源	备注
14	可验证性	维护完整性	修改后的软件是否修复、纠正或完成需求中提出的要求。	X = A/B A=在评审中证实按规定已实现的检查点数； B=设计中的检查点总数。	GJB 5236-2004	0 ≤ X ≤ 1，越接近1越好。
15	依从性	维护性的依从性	遵循与产品的维护性有关的法规、标准和约定的程度。	X = A/B A=在评价中已证实的正确实现与维护性的依从性相关的项； B=依从性的总项数。	GJB 5236-2004	0 ≤ X ≤ 1，越接近1越好。
16	易分析性（失效诊断的效率）	失效诊断的准确性	系统与软件是否能够有效地定位失效。	X = A/B A=用户成功定位的失效数； B=系统实际失效个数。	GB/T 29834.2-2013 GB/T 16260.2-2006 GJB 5236-2004	0 ≤ X ≤ 1，越接近1越好。
17	易分析性（失效诊断的效率）	失效诊断的时间	系统与软件有效定位失效的时间。	$X = (\sum_{i=1}^{n} T_i)/n$ T_i=成功定位第i个失效的时间。	GB/T 29834.2-2013 GB/T 16260.2-2006 GJB 5236-2004	X越小越好。
18	易分析性（对失效诊断的支持）	对诊断功能的支持	支持原因分析方面诊断功能的能力；用户能否标识引起失效的是哪个具体功能；维护者能否容易地发现失效的原因。	X = A/B A=维护者能（利用诊断功能）诊断并理解因果关系的失效数； B=登记的失效总数用户成功定位的失效数。	GJB 5236-2004	0 ≤ X ≤ 1，越接近1越好。
19	易分析性（对失效诊断的支持）	有效线索比例	系统与软件能否提供充分的维护线索以支持维护实施。	X = A/B A=软件系统实际提供的有效线索数； B=软件系统计划维护（及相关）点提供的线索数。	GB/T 29834.2-2013 GB/T 16260.2-2006	0 ≤ X ≤ 1，越接近1越好。
20	易分析性（对失效诊断的支持）	可理解线索比例	实施方能否正确、有效的理解该次维护过程提供的线索。	X = A/B A=实施方能正确理解的线索数； B=软件系统提供的有效线索数。	GB/T 29834.2-2013 GB/T 16260.2-2006	0 ≤ X ≤ 1，越接近1越好。
21	易分析性（对失效诊断的支持）	状态监视的能力	通过取得运行时的监视数据，用户是否能标识引起失效的具体操作；通过取得运行时的监视数据，维护者是否容易找到失效的原因。	X = A/B A=维护者（或用户）未能取得监视数据的事例个数； B=维护者（或用户）企图取得运行时记录软件状态的监视数据的事例个数。	GJB 5236-2004	0≤X≤1，越接近1越好。
22	易分析性（对失效诊断的支持）	审核追踪的能力	当系统与软件发生失效时能否有效的追踪其失效的具体位置。	X = A/B A=在运行中实际记录到的数据数； B=计划在运行中要记录的足以监视系统与软件状态的数据数。	GB/T 29834.2-2013 GB/T 16260.2-2006 GJB 5236-2004	0 ≤ X ≤ 1，越接近1越好。

表 1　维护性指标计算方法（续）

序号	类别	指标	含义	计算方法	来源	备注
23	模块化	模块间的耦合性	模块间存在的依赖关系的强弱。	$X = \sum_{i=1}^{n} B_i/n$ $B_i=\sum_{j=1}^{n} C_{ij(i\neq j)}/(n-1)$，是模块i与其他模块的关联强度； C_{ij}=模块i和模块j间的关联强度； n=软件的模块数。	GB/T 29834.2-2013	X越小越好。
24	模块化	模块结构的合理性	模块结构符合要求的程度，包括代码、预定义的代码。	X = A/B A=合理的维护模块数； B=系统维护模块数。	GB/T 29834.2-2013	$0 \leq X \leq 1$，越接近1越好。
25	规范性（文档维护指导性）	对维护的指导性	在维护分析和实施的过程中，文档能提供的指导程度。	X = A/B A=用户通过查找文档能解决的问题数； B=在维护分析过程（诊断、修改和验证）中相关人员（提出方、实施方和验证方）发现问题的总数。	GB/T 29834.2-2013	$0 \leq X \leq 1$，越接近1越好。
26	易改变性（可修改性）	代码的可修改性	维护实施方能否通过修改代码来维护指定的功能。	X = A/B A=维护实施方通过修改代码正确维护功能的个数； B=计划维护的功能点个数。	GB/T 29834.2-2013 GJB 5236-2004	$0 \leq X \leq 1$，越接近1越好。
27	易改变性（可修改性）	可配置性	维护实施方能否容易地变更配置参数来实施修改。	X = A/B A=维护实施方通过修改（配置）参数完成维护的功能的个数； B=计划维护的功能点个数。	GB/T 29834.2-2013 GB/T 16260.2-2006 GJB 5236-2004	$0 \leq X \leq 1$，越接近1越好。
28	易改变性（修改实施的效率）	变更周期的效率	当系统与软件发生失效时能否在可接受的时间内解决。	$X = \sum_{i=1}^{n} T_{u_i}/n$ $T_{u_i}=T_{r_i}-T_{s_i}$； T_{r_i}=发现问题的时间； T_{s_i}=解决问题的时间维护实施方维护完成的时间； n=修订版本的次数。	GB/T 29834.2-2013 GB/T 16260.2-2006 GJB 5236-2004	$0 < X$，越短越好，除非修订版本的次数太多。
29	易改变性（修改实施的效率）	修改实施的效率	维护过程是否能在可接受的时间限度内完成。	$X = \sum_{i=1}^{n} T_{u_i}/n$ $T_{u_i}=T_{r_i}-T_{s_i}$； T_{r_i}=第i次维护完成的时间； T_{s_i}=第i次维护实施方计划开始维护的时间； n=修订版本的次数。	GB/T 29834.2-2013 GB/T 16260.2-2006 GJB 5236-2004	$0 < X$，越短越好，除非修订版本的次数太多。
30	易改变性（修改实施的效率）	修改的复杂度	维护是否能容易的执行以解决问题。	$X = (\sum_{i=1}^{n} A_i/B_i)/n$ B_i=变更所花费的工作时间； B_i=软件变更后的规模； n=系统与软件变更的次数。	GB/T 29834.2-2013 GB/T 16260.2-2006 GJB 5236-2004	$0 < X$，越短越好，除非修订版本的次数太多。

表 1　维护性指标计算方法（续）

序号	类别	指标	含义	计算方法	来源	备注
31	易改变性（修改的可控制性）	修改的可还原性	具有撤销等类似功能的软件系统，在完成修改后，是否可以正常还原到修改前状态。	X = A/B A=成功还原的功能点数； B=试图还原修改的功能点数。	GB/T 29834.2-2013	0 ≤ X ≤ 1，越接近1越好。
32	易改变性（修改的可控制性）	软件变更控制的能力	用户能否容易地标识修订的版本。	X = A/B A=具有明确修订标识的版本个数； B=维护过程中所有的软件版本（包括初始版本）个数。	GB/T 29834.2-2013 GB/T 16260.2-2006	0 ≤ X ≤ 1，越接近1越好。
33	稳定性	变更成功的比率	系统与软件在维护之后是否不再失效。	X = A/B A=系统与软件在维护后失效的次数； B=系统与软件在维护前失效的次数。	GB/T 29834.2-2013 GB/T 16260.2-2006 GJB 5236-2004	0 ≤ X ≤ 1，越接近0越好。
34	稳定性	修改影响的局部化	系统与软件在变更后对于系统与软件自身其他功能的影响。	X = A/B A=系统与软件变更解决失效后再次出现的失效数； B=系统与软件解决的失效数。	GB/T 29834.2-2013 GB/T 16260.2-2006 GJB 5236-2004	0 ≤ X ≤ 1，越接近0越好。
35	可验证性	可自动验证性	是否可以通过软件的自动验证来完成。	X = A/B A=能自动验证的功能点个数； B=需维护的软件功能点个数。	GB/T 29834.2-2013 GB/T 16260.2-2006	0 ≤ X ≤ 1，越接近1越好。
36	可验证性	重新测试的效率	用户与维护者能否容易地进行运行测试并确定软件是否准备好运行。	$X = \sum_{i=1}^{n} T/n$ T=为确认是否已经解决所报告的失效而花费的测试时间； N=解决的失效总数。	GJB 5236-2004	0 ≤ X，越小越好。
37	可验证性	内置测试功能的有效性	用户与维护者能否不必准备附加的测试设施就可容易地做运行测试。	X = A/B A=维护者能利用合适的内置测试功能实； B=测试机会的实例数。	GJB 5236-2004	0≤X≤1，越接近1越好。
38	可验证性	测试的重启性	在维护后，能否容易地用检测点执行测试。	X = A/B A=在所希望的点上逐步检测时维护者能够暂停并重新开始测试的事例数； B=在测试中暂停的总次数。	GB/T 29834.2-2013 GB/T 16260.2-2006 GJB 5236-2004	0 ≤ X ≤ 1，越接近1越好。
39	可验证性	维护完整性	修改后的软件是否修复、纠正或完成需求中提出的要求。	X = A/B A=正确维护的功能点个数； B=计划维护地功能点个数。	GB/T 29834.2-2013	0 ≤ X ≤ 1，越接近1越好。
40	依从性	维护性的依从性	遵循与产品的维护性有关的法规、标准和约定的程度。	X = A/B A=在测试期间规定的可维护性的依从性还未实施的项数； B=规定的维护性的依从性的项目总数。	GJB 5236-2004	0 ≤ X ≤ 1，越接近1越好。

6　复杂软件系统维护性定性要求

6.1　软件维护性分析要求

进行维护性分析时应充分考虑如下因素：

a） 软件所处寿命周期阶段；

b） 规定的软件可维护性要求；

c） 软件的维护方式，包含预防性维护、周期性维护和修改性维护等；

d） 每项活动的主要任务；

e） 拟采用的开发技术和类似软件的历史状况；

f） 时间进度、经费与其它资源，存贮空间与运行时间，程序设计语言，软件运行的软硬件环境等各种限制条件。

6.2 软件设计与开发过程中的维护性要求

为保证软件的维护性，软件的设计与开发需满足以下要求：

a） 软件表示应易于理解，逻辑清晰，命名规范，便于维护期间的代码阅读、变更定位与影响分析；

b） 软件结构应模块化，高内聚、低耦合，支持模块的独立替换与局部修改；

c） 应提供完整、准确且及时的文档，并随变更同步更新，覆盖架构说明、接口契约、数据/配置、部署与运维/故障处置信息，确保维护可追溯；

d） 代码应具备良好可读性和一致性，采用统一编码风格与目录/提交规范，配合静态检查与评审，降低后续维护难度；

e） 系统应易于修改和扩展，以适应需求变化；

f） 应具备可测试性，方便进行单元测试和集成测试；

g） 错误处理机制应完善，便于定位和修复问题；

h） 应遵循统一的程序设计规范和标准。

6.3 软件维护性保证及软件维护工作的人力资源要求

为了保证软件的维护性，有效的开展软件维护工作，相关人力资源需满足以下要求：

a） 应准确地确定维护人员数量，维护人员数量应该由管理人员和开发人员在维护机构的帮助下确定；

b） 软件维护必须有专人负责，建立维护计划，定期进行维护；

c） 软件维护从建立原型系统后开始；

d） 定期清理系统和备份；

e） 建立运行日志档案，更新系统设置说明和用户文档；

f） 设置全面、准确和易理解的系统文档。

6.4 软件开发及维护活动的维护性要求

6.4.1 开发与维护工具要求

6.4.2 软件审查要求

在软件生命周期的不同阶段，需开展以下审查工作，以减少不利于保证软件维护性的因素：

a） 需求阶段：关注潜在变更需求，避免在开发后期进行被动的设计补充与返工；

b） 设计阶段：检查设计文档与模型的逻辑清晰性，减少歧义、降低思维复杂度；

c） 编码阶段：确保遵循编码规范，如命名规则、注释完整性等；

d） 测试阶段：需通过自动化工具实现持续集成和代码/可执行文件的静态分析，结合人工审查确认，针对软件及其测试用例进行审查，提前发现可维护性问题；

e） 部署与维护阶段：在软件修改、改进和重构的过程中通盘考虑维护性的保证，并对软件变更进行维护性的审查与确认。

6.4.3 软件测试活动要求

为保证软件维护性，需开展充分且有效的测试活动，以保证软件的功能稳定性、避免代码变更引入新缺陷。

6.4.4 可重用性要求

各个组件和模块应当可以重用在其它项目，避免重复开发。

6.4.5 技术债务清理要求

周期性执行逆向工程分析遗留系统，结合需求变更，重构或重新设计高风险、难维护的模块。

6.5 软件维护相关文档编制要求

6.5.1 总体要求

软件维护相关文档需满足以下的编制要求：

a） 清晰说明核心目标：指导用户正确操作软件，并为维护人员提供故障诊断、版本升级及系统优化的标准化流程。此外，明确区分用户操作与维护任务两部分内容，避免交叉混淆；

b） 文档完整性：必须覆盖软件全生命周期支持，包括安装、配置、日常操作、异常处理、数据备份、迁移和退役流程；此外还需提供与软件版本严格对应的配套文档清单（如设计文档、接口规范、测试报告），确保维护时可追溯原始设计意图；

c） 格式标准化：采用统一模板，强制要求包含章节编号、版本标识、修订历史表。

6.5.2 文档结构及内容要求

6.5.2.1 引言部分

引言部分需满足以下要求：

a） 说明手册适用范围、目标读者（用户/维护工程师）、软件功能概要及兼容性环境（操作系统、硬件配置、依赖库）；

b） 术语表需包含理解文档所必需的专业缩写及行业术语定义。

6.5.2.2 软件使用指南

软件使用指南需满足以下要求：

a） 操作流程：分步骤描述功能操作，配界面截图及输入/输出示例；

b） 运行控制：列出所有操作命令、参数说明、预期系统响应及超时处理机制。

6.5.2.3 软件维护指南

软件维护指南需满足以下要求：

a） 故障诊断树：提供常见错误代码表、排查逻辑图及解决方案（如：数据库连接失败→检查端口占用/凭证有效性）；

b） 维护接口规范：公开API调用方法、日志格式（含错误级别定义）、监控指标阈值、配置项清单等；

c） 数据管理指南：备份周期建议、存储路径规则、迁移工具使用说明。

6.5.2.4 软件升级与扩展指南

软件升级与扩展的指南部分需满足以下要求：

a） 版本升级步骤：强调回滚流程及兼容性检查点（如：V2.0升级前需验证插件版本≥1.5）；

b） 模块扩展说明：描述二次开发接口、插件加载机制及安全约束条件。

6.5.2.5 软件使用与维护说明要求

软件使用与维护手册需载明可维护性相关的注意事项：

a） 代码示例需标注维护关键点，例如：“配置项config_timeout修改后需重启服务生效”；

b） 当软件功能或数据结构变更时，需在手册中同步更新说明内容及影响分析，例如字段删除导致的历史数据兼容方案等。

6.5.2.6 变更追踪机制

软件文档需具备修改需记录变更日期、版本号、修改内容及责任人，关联需求变更编号。

6.5.2.7 可验证性

有关软件使用与维护的操作步骤需附带验证方法，描述进行操作后，软件产生何种响应才表示成功；此外，维护案例需提供测试数据集及预期输出样例。

6.5.3 文档质量保证

软件文档需通过评审改正并持续改进，保证文档质量，具体要求如下：

a） 评审机制：交付前需通过三方评审（开发、测试、维护团队），重点验证维护场景的覆盖完整性；

b） 持续改进：根据用户反馈（如工单高频问题）定期增补手册内容，每季度发布补遗附录；

6.6 软件维护性评审要求

6.6.1 需求分析的维护性评审要求

针对需求分析文档和维护性大纲等资料，需评审以下事项：

a） 可维护性指标分配；

b） 操作顺序和不可逆操作顺序的保障要求；

c） 功能降级使用方式下，软件产品最低功能保证的规格说明；

d） 系统提供的运行条件；

e） 人机界面要求，对人的因素的考虑；

f） 大纲及其实施计划、软件确认测试计划；

g） 选用或制定的规范和准则。

6.6.2 软件设计的维护性评审要求

在概要设计评审时要评价软件可维护性设计方案，而在详细设计评审时，要评价模块可维护性详细设计的正确性、充分性、完整性，并分析有关规范、准则的实施情况，评价与其它软件及硬件交互作用的协调一致性。具体评审内容为：

a） 设计方案中所采用的软件可维护性专门技术和措施；

b） 设计方法，重点在对过载情况和软件操作顺序的考虑；

c） 与关键时序要求、预估的运行时间。错误恢复和有关性能要求的一致性；

d） 人机界面和人的因素的要求；

e） 测试原理、要求和工具；

f） 测试文件的技术准确性和协调一致性，以及它们与测试要求的一致性。

g） 保证设计完整性所需的设计资料，例如逻辑图、算法、存贮器分配图、以及用流程图和程序设计语言等表示的详细设计；

h） 外部和内部界面，包括同数据库的交互作用；

i） 测试文件的技术准确性和协调一致性，以及它们与测试要求的一致性；

j） 软件支持要求和维护工具；

k） 软件现场支持要求.

6.6.3 验证、确认和测试过程的维护性评审要求

对软件的验证和确认过程进行可维护性评审，需评审以下事项：

a） 保证可追踪性的验证方法和判据；

b） 用于检验、演示、分析和测试的软件确认方法；

c） 评审报告、测试结果和审核等文件；

d） 待测试的软件配置说明，包括支持测试的硬件和软件；

e） 测试计划、测试规程和测试用例；

f） 每项测试的验收判据；

g） 日程表和责任分工；

h） 软件可维护性分析报告。

7 软件维护性支撑技术与方法

7.1 软件维护性保证流程

7.1.1 维护性分析

维护性分析是在软件生命周期早期开展的一系列前瞻性规划与评估活动，它通过制定详尽的维护计划、明确软件运行条件、评估维护要求的可行性与成本风险、建立相应的开发规范与准则。

7.1.1.1 制定软件维护工作计划

实施计划应在需求分析阶段制定，其内容应包括：

a） 大纲实施的组织机构及职责；

b） 定性、定量的可维护性目标；

c） 各项具体任务的进度表；

d） 评价和验证判据；

e） 标准化要求；

f） 软件版本控制；

g） 评审计划；

h） 文件编制要求；

i） 培训与支持保障计划。

7.1.1.2 确定运行条件

在系统分析与软件定义及需求分析阶段应确定软件运行条件，在概要设计和详细设计阶段对运行条件可作必要的修改。可从以下角度进行分析：

a） 运行的系统及其体系结构；

b） 运行和维护方式；

c） 加载情况；

d） 运行和维护环境(如电磁发射和感应);

e） 运输和安装条件；

f） 操作和维护人员要求；

g） 新版本安装或升档；

h） 恢复的规程和要求；

i） 终端和通信介质的类型，

7.1.1.3 进行可维护性要求的可行性分析

在系统分析与软件定义阶段应对软件可维护性要求进行可行性分析。确定设计工作的起点，估计它对开发技术选择、设计配置和软件性能的影响，并估计从现有软件可维护性要求提高到新软件要求所需的费用和承担的风险。应当考虑以下因素：

a） 软件的功能要求；

b） 新软件的市场潜力；

c） 现有软件的技术状况；

d） 寿命周期费用；

e） 开发新软件所需的工作量与改进现有软件的工作量进行比较，还应评审合同(或任务书)中有关要求的可行性，评价其符合系统要求的能力，并评 价它对软件的其它要求(如其它质量特性、现场保障、性能等)的关系。

7.1.1.4 选定或制定规范和准则

在维护性分析时，应当选定或制定软件维护性的规范和准则，要求如下：

a） 确定保证软件可维护性所必须的软件工程规范；

b） 制定软件开发必须遵守的技术准则；

c） 制定软件的支持和维护要求；

d） 必要时制定对外购、转承开发和重用原有软件的可维护性控制规范。

7.1.1.5 进行可维护性分析

在软件开发过程各阶段应进行有关的可维护性分析并编写分析报告，分析的工作量和日程应包括在大纲计划中。分析时应当考虑：

a） 可维护性指标分配情况；

b） 软件使用需求量过载情况；

c） 程序设计的实施情况；

d） 对外购、转承开发和重用原有软件的控制规范实施情况；

e） 故障原因和故障模式影响及危害度分析；

f） 关键模块分析；

g） 不可靠因素的分析及预防措施；

h） 故障定位和隔离技术的应用；

i） 测试环境、测试系统、测试用例和测试覆盖情况；

j） 可维护性预计情况；

k） 维护实施简易性。

7.1.2 维护性设计与实现

在软件的设计阶段，需专门开展维护性设计。通过这些综合措施，构建一个易于理解、便于修改和扩展的软件系统，有利于软件的长期维护。

7.1.3 维护性测试与评价

7.1.3.1 专家评审法

根据被评审对象和评审目的，设计评审项目表，列出打分栏目、分值、权重和打分规则。可由N个专家组成一个评审组，专家根据自身的经验与认知，进行判断打分。然后，依据专家的权重和统计规则，进行分值汇总计算，其计算得出的值作为评审的结果。

7.1.3.2 技术测试法

技术测试时，可依据被测对象和测试目的，选择采用适用的自动化测试工具进行，也可由人工进行手工测试。技术测试获得的结果通常是一种量化的测量结果。

7.1.3.3 用户调查法

用户调查时，应根据调查的目的和特定的用户群，设计调查表，让被调查对象填写并反馈，调查表的回收数应达到一定的数量，并不低于发出数的适当比例。然后，对回收的调查表进行汇总计算，其计算得出的值作为用户调查的结果。

7.2 软件维护性设计与保证方法

7.2.1 软件体系结构的维护性设计与保证方法

本节系统性地阐述了提升软件体系结构维护性的核心设计方法，其根本目标是通过降低系统复杂度、隔离变更影响，从而构建一个易于理解、修改和扩展的软件系统。

7.2.1.1 分层设计

分层架构是一种常见的结构化设计方法，可分离关注点，使每层专注于特定职责；限制层间依赖，通常只允许上层调用下层，禁止逆向调用；增加替换灵活性，更改某层而不影响其他层。

7.2.1.1.1 软件分层方式

分层架构设计中，通常对于运行在通用操作系统和通用计算机平台的软件，可根据软件实际，选择性的包含如下层次：

a） 用户界面层：处理用户界面和交互，包含图形界面或命令行界面；

b） 业务逻辑层：包含核心业务规则和处理；

c） 数据访问层：负责数据持久化和检索；

d） 网关服务层：对外提供通用统一的服务，如Restful API、GraphQL和RPC等服务；

e） 硬件及外设访问层：访问和管理特殊硬件和外设，如飞行训练软件的操作杆等。

对于嵌入式软件，可根据软件实际，选择性的包含以下层次：

a） 业务逻辑层：实现核心算法和业务流程控制等；

b） 外部接口层：提供与外部软件的信息沟通，如串口、IIC、SPI和TCP/UDP等；

c） 通用服务层：提供通用的底层基础设施服务，如网络协议栈、通用加密算法、文件系统等；

d） 系统服务层：基于定时器、实时操作系统等方案，实现任务调度，并进行内存管理、任务间通信等核心服务接口的提供；

e） 硬件驱动层：在硬件抽象层基础上进一步封装外设驱动（如SPI、I2C通信），提供更高级的操作接口，实现具体的系统功能；

f） 硬件抽象层：直接与硬件交互，封装寄存器操作、中断管理和外设驱动等，为硬件的操作提供统一的硬件接口。

7.2.1.1.2 软件分层设计准则

分层设计的核心是职责分离、层间解耦，从而保证软件的维护性，可按照以下方法和准则进行设计：

a） 单一职责：每一层仅负责一项明确的职责，避免功能混杂。例如，表现层仅处理用户交互，业务逻辑层专注核心业务规则；

b） 单向依赖：依赖关系单向流动，高层（如表现层）依赖低层（如数据访问层），禁止逆向或跨层调用，必要时使用依赖倒置的方式进行解耦；

c） 高内聚低耦合：层内功能紧密相关，如数据访问层仅包含数据库操作，不应涉及业务规则；同时层间耦合尽量减少，如MVC模式中模型与视图通过控制器进行解耦，避免直接交互；

d） 接口隔离与抽象化：每层对外提供明确的接口，隐藏实现细节。例如，服务层通过REST API暴露功能，内部逻辑变化不影响调用方；

e） 开闭原则：软件实体应对扩展开放，通过分层设计支持功能扩展，如新增支付方式时，只需扩展支付服务层，无需修改其他层；应对修改关闭，底层实现变更不应影响上层代码，例如通过ORM框架屏蔽不同数据库之间的差异；

f） 边界清晰：明确每层的输入和输出，避免如业务逻辑分散在数据访问层等问题；

g） 易测试：每层可独立测试。例如，通过Mock工具隔离业务层与数据层依赖；

h） 异常友好：底层异常（如数据库连接失败）应转换为高层可识别、理解和分析的业务异常；

7.2.1.2 模块化及组件化设计

7.2.1.2.1 模块化设计

模块化设计是将复杂系统划分为多个相对独立、功能完整、可替换的模块的方式，降低系统不同功能之间的互相影响，将软件修改限制在单个或少数几个模块内，并有效隔离修改某一模块对其它模块的影响，从而有效提升软件的维护性。常见的模块化设计主要体现在以函数、类和包等形式来封装程序逻辑，设计时应遵循以下方法或准则：

a） 高内聚低耦合：模块内部元素紧密相关(高内聚)，模块之间依赖最小化(低耦合)；

b） 单一职责：每个模块只有一个引起内部状态变化的原因，即只负责一个功能；

c） 接口定义清晰：模块间通过定义良好的接口进行通信，隐藏实现细节；

d） 模块大小适当：模块大小适中，不应过大导致难以理解，也不应过小导致系统碎片化。

7.2.1.2.2 组件化设计

组件化设计是模块化设计的进一步延伸，将系统拆分为独立部署和运行的组件，可独立部署、安装、维护和替换，并隔离各个组件之间的故障，从而提升软件的可维护性。组件化软件的基本实现方法及其对维护性的影响列举如下：

a） 基于进程内调用的架构：通过动态链接库、软件包等方式，在主程序进程内加载和调用各个组件。此种方式调用效率高，但组件与主程序的隔离度低，不利于维护；

b） 基于进程间调用的架构：各个组件为可执行文件，通过主程序启动可执行文件形式的组件来实现调用。此种方法无需网络通信，且组件的失效一般不会影响主程序，主要通过命令参数、进程间通信或文件读写的方式进行数据传递，更有利于维护。但这种方式的执行效率低于进程内调用，且无法跨多机部署，隔离度较低。例如，子组件的内存占用高时，则可能造成主程序卡顿甚至因内存不足而异常退出；

c） 基于服务的架构：以服务的形式实现不同组件，实现不同组件间的完全解耦，所有组件均可作为独立的软件工程项目进行维护，且不同组件可部署在不同容器或主机，隔离度最高。但其调用效率低，且跨多服务的调用或消息传递过程难以使用常规单体软件的方式进行排查，需要完备的日志记录、链路追踪等方案，从而提升维护性。

组件的设计除了要满足模块化设计的准则外，还需满足以下准则：

a） 接口定义明确：接口需进行明确定义，便于实现组件间的松耦合；

b） 数据统一：组件间的数据传输格式应进行统一化；

c） 可替换性：组件应设计为可被其他实现相同接口的组件替换而不影响系统；

d） 独立部署：组件可以独立更新和部署，降低系统维护风险。

7.2.1.3 通用维护性保证

7.2.1.3.1 软件依赖管理

针对软件开发所需的依赖项，如编程语言、开发框架和软件包等，可依据以下的具体方法与原则进行管理：

a） 使用流行且在目标业务场景下广泛应用的编程语言、开发框架和软件包，从而便于获取相似的开发和维护的知识与资源，有利于软件维护；

b） 在软件早期设计阶段应充分识别各依赖项的使用许可，避免在开发中后期发现许可证不符而进行依赖项变更；

c） 使用稳定的依赖项版本，减少依赖项不稳定造成的技术债务；

d） 使用自动化工具管理软件依赖项，若没有此类工具，则须人工记录清晰的依赖清单；

e） 使用开源依赖时，优先选择有专职团队维护或社区活跃的开源依赖，不要使用已停止支持的开源依赖；

f） 确保依赖项具备完整的文档（如API文档、配置指南），并以软件维护团队的所有成员均可有效阅读和理解的语言和形式呈现，便于软件维护时查阅。若文档没有适合软件维护团队所有成员阅读的语言或形式，则需要进行翻译或转换；

g） 依赖项的替换或升级须经过充分的测试，从而保证功能一致性；

h） 对组件化开发的项目，可根据项目实际，使不同组件独立管理依赖，避免全局依赖冲突；

i） 定期检查项目依赖项的支持情况，在软件升级时，须对即将停止维护的依赖项制定迁移计划，逐步替换为替代方案；

j） 及时移除未使用的依赖项，减少技术债务和潜在冲突；

k） 对于自研的依赖项如编程语言、开发框架或软件包，须对自研依赖项进行有效的版本控制，规范区分主版本、次版本和补丁版本的更新影响，规范管理版本发布。

7.2.1.3.2 日志系统

针对不同类型的软件，如单体软件、组件化软件等，均需要设计标准化、自动化、可追溯和安全的日志系统。一旦软件运行过程中出现故障，排错人员可在运行日志的支持下，定位软件故障位置，然后基于流程图或伪代码，提出修改方案，完成维护工作。同时，完善的日志信息也有利于判断系统状态、预测未来趋势，利于预防性维护的开展。

日志系统应按照以下的准则与方法进行设计：

a） 保证日志级别统一且清晰等，根据项目实际明文规定各级别日志的使用场景；

b） 使用清晰的结构化格式，便于机器自动化解析，包含时间戳、线程名、日志级别、类名、消息体、异常堆栈等基础字段，根据项目实际进行增删；同时也需要在保证性能和信息安全的基础上，尽量记录完整的上下文（如用户ID、请求参数、操作时间），便于日志数据的处理和挖掘；

c） 支持自动日志收集，降低人工日志收集的工作量，同时注意支持异步传输以避免阻塞主业务流程；

d） 日志进行集中存储，并对高频查询日志建立索引，提升日志查询的速度；

e） 历史日志的清理时间需根据项目实际确定，但不宜过短，避免丢失近期的故障信息；

f） 日志系统应具备监控和告警装置，如对日志中错误率进行告警，并通过邮件、短信等方式通知管理和维护人员；

g） 针对微服务系统，日志系统需支持链路追踪，便于故障的定位和排查。

实际实施时需结合软件类型（如单体软件或微服务软件）调整日志系统设计的侧重点，例如嵌入式系统需侧重轻量级采集和本地存储优化，而企业级应用需强化日志的集中化管理和合规性审计。

7.2.2 软件用户界面的维护性设计与保证方法

针对软件用户界面的维护性设计与保证，除了需要遵循软件本身的维护性设计与保证方法外，还可使用如下针对用户界面的方法：

a） 界面组件独立封装，通过标准化接口交互，支持单独修改或替换而不影响整体功能，降低维护风险与成本；

b） 保证布局、色彩、图标及交互逻辑跨平台、跨版本一致性，遵循设计系统规范，减少用户学习成本及维护复杂度；

c） 保持代码结构清晰、命名规范、注释完整，采用分层架构分离视图与逻辑，提升界面及其交互逻辑的可理解性与可修改性；

d） 建立健全用户反馈机制，并通过埋点、日志分析等收集界面使用问题，驱动迭代优化，确保维护方向符合用户需求。

7.2.3 软件设计模型的维护性设计与保证方法

7.2.3.1 软件建模

在软件建模阶段，需按照以下方法进行设计，以确保模型的可维护性、可扩展性和技术规范性：

a） 模型分层设计：采用层次化建模，将业务逻辑与实现细节分离，便于维护时聚焦特定层级；

b） 模块化分解：将模型划分为功能独立的模块（如Simulink中的子系统或模型引用），每个模块职责明确，减少修改时的连锁影响；

c） 接口标准化：明确定义模块间的输入输出接口，采用统一的数据类型和通信协议，便于替换和扩展；

d） 设计模式应用：在建模中引入设计模式（如状态模式、观察者模式），提升模型结构的灵活性和可理解性；

e） 控制圈复杂度：避免过度嵌套的逻辑结构，通过子模块拆分降低单个模块的复杂度；

f） 需求-模型映射：建立需求与模型元素的追踪关系，确保修改时能快速定位影响范围。

7.2.3.2 模型维护

在模型维护阶段，需按照以下方法进行维护，以保证模型的可维护性、可扩展性和技术规范性，减少技术债务的引入：

a） 模型驱动测试：基于模型自动生成测试用例，覆盖正常和异常场景，在模型修改后需对模型相关的测试用例全量测试通过后才可进行系统级测试，确保修改后功能稳定性；

b） 形式化验证：使用形式化验证工具，如模型检查工具或符号执行工具，验证模型是否满足时序或逻辑约束，减少后期维护风险；

c） 静态分析工具：使用模型静态检查工具检测模型中的冗余、耦合、类型错误等问题；

d） 依赖关系分析：在模型变更前，通过依赖关系分析工具（如Simulink Dependency Analyzer），评估修改对其他模块的影响。

7.2.4 软件代码的维护性设计与保证方法

7.2.4.1 简易性设计

软件源代码实现的过程中，应使用合适的方法，保证符合以下的简易性要求：

a） 模块中可执行语句的行数适当；

b） 模块中操作符的数量适当；

c） 模块包含大量的控制分支语句；

d） 模块中使用的嵌套语句可以控制；

e） 模块中使用的大量数据项易于处理；

f） 模块中使用的大量复合数据结构易于处理；

g） 模块中使用的复杂公式易于处理；

h） 模块内不存在过于技巧性、难以理解的程序设计。

7.2.4.2 一致性设计

软件代码实现过程中，需保证以下一致性：

a） 代码注释块信息与相关的源代码一致；

b） 软件文档的输入/输出信息与源代码一致。

7.2.4.3 可扩展性设计

软件在进行源代码的设计过程中应采取以下措施，保证快捷、方便地对模块源代码的数量和类型等进行增删或者修改：

a） 保证模块内常量和数据结构大小、维数等参数化、可配置；

b） 保证可用的内存容量、处理器性能满足软件修改的额外要求；

c） 保证可用的输入/输出带宽应满足软件修改后的需求。

7.2.5 软件文档的维护性设计与保证方法

7.2.5.1 可描述性设计

软件在进行文档的设计过程中应考虑软件的可描述性。模块中必须包含输入/输出、条件、构件、约束、与其它模块间关系等信息。此外，详细的标识、规范的缩进和注释也是必不可少的，这些有助于维护人员对源代码的理解。软件可描述性设计方法如下：

a） 注释块包含充分的软件开发基本信息；

b） 注释块包含充分的模块处理信息；

c） 注释块包含充分的接口信息；

d） 标识符名称有相关解释信息；

e） 有充分描述局部数据的注释；

f） 有充分描述处理和模块内控制转换的注释；

g） 有充分描述错误处理的注释；

h） 有充分描述源代码对计算机要求的注释；

i） 语言标号使用合理，含义贴切；

j） 源代码使用缩进和空行，以增强其可读性；

k） 源代码采用结构化编程。

7.2.5.2 可追溯性设计

软件在进行文档的设计过程中应考虑软件的可追溯性。要求源代码的描述文档之间以及文档和模块源代码之间都有可追溯性信息，以实现信息在顶层需求到底层详细实现之间的追溯，有助于维护人员对底层代码进行维护。软件可追溯性设计要求如下：

a） 文档中包含对源代码的详细描述；

b） 文档中包含对数据项的可追溯性描述；

c） 数据库文档中包含对数据库数据项的可追溯性描述信息。

7.2.6 软件配置项体系的维护性设计与保证方法

软件配置项体系设计需确保所有配置项在整个生命周期内得到系统化管理和控制，以维护产品的完整性和一致性，提升可维护性。具体方法如下：

a） 唯一标识设计：为每个配置项分配全局唯一的标识符，采用结构化命名规则（如"模块名类型序列号"），确保配置项可明确区分和追踪；

b） 分类与基线管理：基线配置项（如源代码、设计文档）必须通过正式评审后冻结，任何修改需遵循变更控制流程；非基线配置项（如项目计划、报告）需定期归档，并限制访问权限；

c） 配置项库权限设计：配置项库应设置权限，避免随意改动。如开发库允许开发人员自由修改，但需自动版本记录；受控库仅授权人员可通过变更流程修改基线配置项；产品库为已发布配置项，禁止直接修改，紧急修复需审批等；

d） 状态可追溯性：应记录配置项的完整生命周期状态，并通过相应的文档或专用管理数据库，实时反映变更历史和当前版本。

7.2.7 软件日志系统维护性设计与保证方法

7.2.7.1 日志内容

软件日志内容可使用以下方法进行设计：

a） 分级控制：定义多级日志，如，按环境动态调整输出级别（如生产环境仅输出WARNING以上）；

b） 结构化格式：每条日志必须包含时间戳、模块名、日志级别、操作详情、错误堆栈（若异常），示例：2024-08-14 10:30:25,789 - ERROR - AuthService - Login failed: user_id=123, error_code=502；

c） 关键操作审计：对数据修改、权限变更等高风险操作记录完整上下文（如操作者IP、目标对象、前/后值）。

7.2.7.2 日志系统功能设计

软件日志系统功能可使用以下方法进行设计：

a） 收集机制：采用轻量级代理异步采集日志，避免阻塞主业务流程；

b） 存储与检索：热数据可建立索引，支持关键词/字段组合检索；冷数据：压缩归档至分布式存储，保留周期可配置；

c） 实时分析能力：集成流处理框架，实现错误率突增告警或性能瓶颈自动检测。

7.2.7.3 日志系统性能与安全设计

日志系统的性能与安全设计可采用以下方法：

a） 容量控制：实现日志轮转、自动清理旧日志；

b） 传输加密：敏感日志在传输/存储中必须加密；

c） 访问控制：需限制日志访问权限。

7.2.7.4 可维护性设计

日志系统的可维护性设计可采用以下方法：

a） 动态配置：支持运行时调整日志级别或过滤规则，无需重启服务；

b） 模块化过滤：按子模块独立启用/关闭日志，减少冗余输出。

7.2.8 配置文件的维护性设计与保证方法

软件版本控制可采用以下方法以保证维护性：

a） 版本号格式要求：版本号格式需按状态区分（如草稿、正式、修改），并规定主/次/修订的递增规则与触发条件；

b） 历史版本保留：保留所有历史版本，支持版本回溯与差异对比，防止版本丢失或混淆；

c） 分支与发布策略：明确主干、开发、特性、修复与发布分支的职责与合流节奏，避免长期漂移与大爆炸合并；

d） 变更记录与追溯：统一提交信息与变更日志（Changelog）规范，版本标签（Tag）关联发布说明与缺陷/需求编号；

e） 标签与签名：采用带注释/签名的版本标签，关键里程碑打标并归档，确保版本可识别、可校验；

f） 审核与合并控制：启用受保护分支与合并审批（评审/CI 通过方可合并），禁止直接向主干推送；

g） 回滚与热修流程：为每次发布建立可回滚点与紧急修复（Hotfix）流程，提供一键回退与发布前后校验；

h） 基线与归档：形成发布基线（代码、依赖、构建产物、配置清单），按版本受控归档并保留审计记录。

7.2.9 软件版本控制的维护性保证方法

软件版本控制可采用以下方法以保证维护性：

i） 版本号格式要求：版本号格式需按状态区分，如草稿状态、正式状态、修改状态，并规定版本递增规则；

j） 历史版本保留：保留所有历史版本，支持版本回溯与差异对比，避免版本丢失或混淆。

7.2.10 软件安装及卸载的维护性设计与保证方法

7.2.10.1 通用的软件安装与卸载

对于任何常用的软件的安装和卸载方式，其维护性设计应采用以下方法：

a） 清晰的安装过程与信息告知：安装前应明确告知用户磁盘空间、系统依赖及安装路径等关键信息，确保过程透明、用户知情；

b） 安装路径与注册表规范化：应使用标准目录结构存放文件，并在注册表中创建清晰、可追溯的条目，为后续维护和卸载提供明确依据；

c） 组件化与按需安装：将软件功能模块化，允许用户按需选择安装，既能节省系统资源，也便于未来对单个组件进行维护或升级；

d） 卸载（清理）的彻底性保证：卸载过程必须彻底删除所有文件、目录及注册表项，确保系统环境干净，避免残留文件影响系统性能或未来软件安装。

7.2.10.2 传统桌面/服务器软件安装与卸载

面向传统桌面/服务器软件安装与卸载过程的维护性设计应采用以下方法：

a） 事务性设计：安装程序应具备事务性能力，确保安装过程可中断、可回滚，避免因安装失败导致系统处于不稳定状态；

b） 依赖管理设计：应能自动检测并引导用户安装缺失的系统依赖，或采用静态链接等方式减少对外部环境的依赖，简化部署；

c） 配置文件隔离设计：用户配置文件应与程序文件分离存放于用户目录下，确保软件升级或重装时用户数据不丢失；

d） 标准化接口调用设计：系统服务和驱动程序的安装必须通过标准服务管理接口，并提供启动失败时的诊断信息与回滚机制。

7.2.10.3 容器化与云原生软件的安装与卸载

面向容器化与云原生软件安装与卸载过程的维护性设计应采用以下方法：

a） 声明式部署：应使用容器编排系统的声明式配置定义应用状态，实现部署的自动化、标准化与可重复性；

b） 数据卷持久化：需将所有持久化数据挂载至外部存储卷，确保容器销毁与重建时数据不丢失；

c） 优雅启停与健康检查：应用必须能处理系统信号以实现优雅启停，并配置健康检查探针，使编排系统能自动进行流量管理与故障自愈；

d） 资源清理彻底性：卸载时应删除所有相关的编排资源对象、容器镜像，并根据策略清理持久化卷，确保无任何资源残留。

7.2.11 软件升级的维护性设计与保证方法

7.2.11.1 通用升级设计

软件升级过程可采用以下的通用维护性设计与保证方法保证维护性：

a） 版本兼容性验证：升级前需确认目标系统硬件、操作系统及依赖库满足新版本要求，避免因环境不兼容导致升级失败或功能异常；

b） 关键基础功能保障：升级过程中的耗时流程应允许后台运行，或允许中断升级流程。此外，应保障产品的关键基础功能的正常使用（如汽车的车载软件系统中，当系统升级时，应保持车辆的动力、转向和制动功能可用）；

c） 数据备份机制：对所有关键数据（用户配置、数据库、业务文件）进行完整备份，并验证备份可恢复性，确保升级中断时可回滚至稳定状态；

d） 数据兼容要求：高版本软件应向后兼容低版本软件的关键用户数据。对于版本差别过大、强行兼容低版本将极大损害代码维护性的，可提供专用数据转换适配工具，将低版本软件用户数据转换为高版本软件用户数据；

e） 升级计划制定：须明确升级时间窗口、影响范围及回滚策略，非关键业务时段执行以减少业务中断风险；

f） 升级提示与告知：重大升级需提前通知用户，说明升级内容、耗时及注意事项；

g） 回滚设计：保留旧版本卸载入口，支持一键回退至升级前状态；

h） 允许手动禁用：需允许用户手动禁止软件升级、关闭更新提示等；

i） 升级内容验证：升级后需验证核心功能、性能指标及数据一致性；

j） 安全防护：升级包传输与存储须验证数字签名及哈希值防止篡改；敏感操作（如权限变更等）需记录审计日志，支持事后追溯；

k） 依赖最小化：在合规且发行包大小可接受的前提下，内置运行时依赖库和第三方程序，避免调用外部路径导致冲突；

l） 电源安全性：对于使用电池的产品，需验证设备电量不低于阈值或确认设备已连接符合要求的外部电源，才可启动安装，避免安装过程中电量耗尽；

m） 存储空间要求：软件升级后应删除升级过程中下载或产生的中间文件，节约存储空间；

n） 合规性要求：特殊领域（如车载、医疗）升级需符合行业标准，部分行业需经过审批。

7.2.11.2 本地安装升级设计

软件的本地安装升级可采用以下的方法保证维护性：

a） 权限最小化：尽量允许以普通用户权限安装和升级，不强制要求管理员权限；

b） 环境隔离：使用明显标识提示用户关闭杀毒软件等可能干扰安装的进程，确保安装过程无外部中断。

7.2.11.3 在线升级设计

软件的在线安装升级，如桌面/服务器软件的在线升级、车载/移动端软件的OTA升级等，可采用以下的方法保证维护性：

a） 升级流程控制：分批次推送更新，按设备类型、区域分批降低服务器负载，实时监控异常率；

b） 升级包传输支持：升级包须支持断点续传，避免重复消耗网络流量；同时在升级包较大时应支持增量更新；

c） 错误日志上报：升级日志信息需实时推送至运维平台，便于分析和管理。

7.2.11.4 容器化升级

7.2.11.4.1 容器镜像构建

容器镜像构建可采用以下的方法：

a） 基础镜像安全性保证：基础镜像必须使用官方或经安全审核的定制镜像（如Alpine、Distroless）；

b） 基础设施不可变保证：禁止直接修改运行中容器的文件系统；

c） 应用代码与依赖库分离：依赖库通过卷挂载或置于基础镜像中，应用层（如JAR包）作为独立镜像层，减少升级时的镜像体积；

d） 语义化镜像标签：生产环境禁止使用latest标签，需采用语义化版本（如v1.2.3）或代码提交版本的唯一标识；镜像名称需包含环境标识（如-prod、-staging等）；

e） 敏感配置分离：敏感配置通过外部数据卷或者环境变量等方式注入，禁止硬编码至镜像。

7.2.11.4.2 镜像仓库管理

容器镜像仓库管理时，可采用以下的方法：

a） 镜像仓库安全性保证：镜像仓库需启用镜像漏洞扫描（如CVE检查）和访问控制（RBAC）；

b） 升级源验证：拉取镜像前需校验数字签名及SHA256哈希值，防止篡改；

c） CI/CD集成：升级流程必须集成至流水线（如GitLab CI、Jenkins），包括镜像构建、测试、推送及部署。

7.2.11.4.3 升级策略与流程

容器镜像升级可采用以下的方法或策略：

a） 分批次替换：有多个同一功能容器时，可每次仅替换不超过一定数量的容器实例，监控健康状态通过后再继续升级，两次升级间时间不低于一定的间隔；

b） 资源预留：升级前预留充足的CPU/内存资源，避免升级过程中资源争用导致服务降级；

c） 健康检查集成：容器需内置健康检查指令，失败时可自动回滚；

d） 蓝绿版本隔离：同时运行新旧版本容器集群（蓝绿版本），通过服务流量切换实现零停机升级；

e） 流量切换条件：新版本需通过自动化测试（如API兼容性、性能压测）且错误率低于阈值后，方可切换流量；

f） 环境约束：生产环境升级需在业务低峰期执行。

7.2.11.4.4 编排系统集成

容器镜像编排系统可采用以下的方法或策略进行集成：

a） 规范要求：遵守容器编排系统的规范，如Kubernetes/Openshift规范等；

b） 数据卷持久化：数据库类容器需挂载外部存储卷（如NFS、云盘），升级前执行快照备份；

c） 顺序控制：有服务间依赖关系的集群需按依赖顺序升级节点。

7.2.11.4.5 监控与验证

容器化升级过程可采用以下方法进行监控与验证：

a） 指标实时监控：基础指标包含CPU/内存使用率、容器重启次数、网络延迟等；业务指标包含请求错误率、事务吞吐量等。

b） 升级后验证：升级后须通过自动化测试脚本校验核心业务流程（如用户登录、支付接口）等，并进行升级前后的数据一致性检查。

7.2.11.4.6 升级回滚机制

容器化升级过程可采用以下方法，保证可恢复到旧版本：

a） 快速回滚设计：镜像仓库至少保留最近多个稳定版本，支持短时间回滚至旧镜像；

b） 流量切换回滚：在蓝绿版本部署时，在应支持短时间内将流量切回旧版本集群；

c） 回滚触发条件：应结合软件的功能和性能要求，制定易自动化探测的回滚触发条件；

7.2.11.4.7 安全与合规保证

容器化升级过程可采用以下方法，保证升级过程中的安全与合规：

a） 安全补丁集成：容器基础镜像每月同步一次操作系统的安全补丁；

b） 合规性审计：记录升级操作日志（如kubectl命令、镜像拉取时间），留存较长时间段供审计；

c） 医疗/车载设备需符合行业标准（如ISO 26262），升级包经签名加密。

7.3 内置测试（BIT）技术与方法

7.3.1 简介

内置测试（BIT）指系统内部集成的自动测试功能，用于故障检测、隔离和诊断，覆盖硬件、软件及接口状态。BIT可以缩短维护时间、降低保障成本、提高装备战备完好率。

BIT的主要实现方式通过传感器、逻辑电路和算法实时采集系统状态（如电压、信号完整性、程序流），与预设阈值或冗余通道数据比对，识别异常。

7.3.2 BIT 测试分类

BIT测试方法主要包含以下三种：

a） 在线监控：实时同步运行，不中断主任务（如飞控系统中的周期BIT）；

b） 激励注入：主动施加测试信号（如电流、模拟故障），验证硬件响应（如维护BIT）；

c） 回绕测试：信号输出后环回输入端口，检测通路完整性。

7.3.3 BIT 工作方式

按工作方式分类，BIT可分为：

a） 主动BIT：主动向软件系统注入测试信号，可能短暂中断软件执行。典型用途如CPU自检等深度硬件诊断；

b） 被动BIT：后台运行检测，通常不中断软件执行。典型用途为实时状态监控，如温湿度监控等。

7.3.4 BIT 工作的时段和功能

按工作的时段和功能，BIT可分为：

a） 启动BIT：软件启动时运行的BIT，主要用于快速基础检测；

b） 周期BIT：与主任务协同运行的BIT，通常用于实时故障监控；

c） 维护BIT：可手动触发，用于进行全面诊断与校准。

7.3.5 BIT 应用案例

BIT在安全关键软件系统中有典型的大量应用。如电传飞控系统中，BIT可实现浮点/整型运算错误检测、各个通道数据一致性校验等。同时具备联锁保护，避免飞行中误触发，确保安全。

7.4 影响域分析技术与方法

7.4.1 定义与目的

影响域分析是复杂软件系统维护过程中识别变更范围、评估修改内容对系统其他部分潜在影响的系统性活动。其核心目标包括：

a） 精准定位受变更影响的软件模块、功能及数据实体；

b） 量化评估变更对性能、安全性、兼容性的风险等级；

c） 为回归测试范围划定提供客观依据，平衡测试充分性与成本效率。

7.4.2 分析内容

影响域分析需覆盖以下维度：

a） 代码级影响：构建多层级代码耦合关系网络，并通过比对变更前后代码版本，识别修改的代码行、函数及类，进而追溯受影响的节点及关联路径；

b） 功能级影响：分析新增/修改功能对其他功能的依赖性影响，包括接口调用关系、数据流传递链及控制流交互；

c） 非功能性影响：评估变更对性能、资源占用率、安全机制（如权限模型）的连锁影响，需量化指标（如响应延迟增幅、并发容量阈值）。

7.4.3 分析方法

可采用以下方法之一或组合使用，从而进行分析：

a） 静态分析：基于代码依赖图（如方法依赖图、类调用树）识别直接/间接关联模块，适用于面向对象软件的精细化影响定位；

b） 动态追踪：通过插桩监控运行时数据流与控制流，捕获实际执行路径的影响范围；

c） 三元闭包理论应用：利用信息流传递关系建立组件依赖网络，消除传递冗余，提升影响域预测准确性；

d） 量化分析模型：定义变更影响深度（依赖层级）、广度（关联模块数）、时效（影响持续时间）三维参数，生成影响域热力图。

7.4.4 分析过程特点

a） 自动化：采用工具实现耦合关系网络自动构建与影响节点追溯，消除人工分析主观性；

b） 可追溯：分析过程需关联需求变更编号、测试用例ID及配置项版本，支持全生命周期审计；

c） 可验证：可通过注入故障用例验证影响域分析的完备性，判断漏检率是否低于目标值。

7.4.5 输出物

分析结果为《变更影响域分析报告》，包含：

a） 变更描述及修改范围边界；

b） 受影响模块清单（含依赖层级与关联类型）；

c） 风险矩阵（高/中/低风险区域及依据）；

d） 推荐的测试覆盖策略（如必备回归测试用例集）；

e） 潜在回滚点及兼容性约束说明。

7.5 预防性维护技术与方法

预防性维护可采用以下的的技术与方法：

a） 定期分析日志与监控数据（如错误率趋势），预测潜在故障并提前修复；

b） 清理冗余代码、移除未使用依赖项，降低系统复杂度。

c） 执行容量与性能评估（压测/基准测试），根据趋势进行扩缩容与参数调优；

d） 开展数据库日常维护（统计信息更新、索引重建/重组、碎片清理、归档与分区管理）；

e） 按计划进行安全加固与补丁更新，执行漏洞扫描与基线加固，关闭过期接口与弱加密套件；

f） 实施配置漂移检测与基线对比，发现偏差及时纠正；

g） 按周期进行备份与恢复演练，验证恢复点目标（RPO）与恢复时间目标（RTO）；

h） 建立依赖与组件更新机制（含语言运行时、中间件、容器镜像），评估兼容性并提供回滚方案；

7.6 周期性维护技术与方法

周期性维护可采用以下的技术与方法：

a） 定期对系统进行健康检查，包括运行状态、资源占用情况、安全漏洞扫描等，形成健康报告；

b） 根据系统运行情况，定期更新系统补丁、升级依赖库，确保系统兼容性与安全性；

c） 执行周期性备份与恢复演练，确保关键数据可及时恢复；

d） 对系统配置进行复核与优化，调整参数以适应运行环境变化。

7.7 修改性维护技术与方法

修改性维护可采用以下的技术与方法：

a） 针对用户反馈或系统运行中发现的缺陷，实施纠错性修改，确保系统功能正常运行；

b） 根据业务需求变化，对系统功能进行适应性调整，包括接口变更、逻辑优化等；

c） 在修改过程中严格遵循变更管理流程，记录修改内容、影响范围及验证结果；

d） 修改完成后进行回归测试，确保系统原有功能未受影响，新功能符合预期。

8 软件系统维护性全生命周期过程与活动

8.1 需求分析阶段

8.1.1 可维护性指标要求

明确系统模块替换时间、故障定位效率、文档更新响应周期等定性或量化指标。

8.1.2 维护场景覆盖

识别典型维护场景（如模块升级、数据迁移、故障恢复），确保需求覆盖未来可能的变更需求。

8.1.3 可维护性约束条件

规定系统需支持的维护接口（如日志输出格式、监控API）、兼容性要求（如新旧版本数据结构兼容）及安全约束。

8.2 设计与实现阶段

将可维护性原则融入架构设计、文档编写与代码实现，遵循本标准与其它标准中的相关要求，参考本标准与其它标准中维护性设计与实现方法，设计与实现高维护性的软件。

8.3 测试阶段

8.3.1 测试环境保证

建立模拟生产环境的可靠性测试平台，覆盖硬件、网络及负载场景。

8.3.2 可维护性专项测试

对软件可维护性进行专项测试，如组件替换测试、新旧版本兼容性测试等。

8.4 交付阶段

软件交付阶段需提供行之有效的软件维护性及维护活动说明，并在相应文档中标注软件维护性及维护活动的关键点（如配置修改需重启服务），以及数据变更兼容方案。此外，软件维护团队需经过充分的培训。

8.5 维护与更新阶段

维护与更新阶段的主要维护性活动如下：

a） 主动管理技术债务并优化可维护性，定期重构高风险模块，优化架构以适应新需求；

b） 所有修改需遵循变更追踪机制，升级上线前须执行兼容性检查并制定回滚预案；

c） 基于用户反馈（如工单高频问题）每季度更新维护手册，补充故障案例与解决方案。

参 考 文 献

[1] GB/T 11457-2006 信息技术 软件工程术语

[2] GJB/Z 141-2004 军用软件测试指南

[3] GJB 439A-2013 军用软件质量保证通用要求

[4] GJB 841A-2024 通用质量特性问题报告、分析和纠正措施系统

[5] GJB 2725B-2024 军用校准和测试实验室能力通用要求

[6] GJB 2786A-2009 军用软件开发通用要求

[7] GJB 5716-2006 军用软件开发库、受控库和产品库通用要求

[8] GJB 5880-2006 软件配置管理

[9] ISO/IEC 25010-2023 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product Quality Model