一、CMMI 层次成熟度模型简述

CMMI（Capability Maturity Model Integration）是一套衡量和提升组织软件开发与管理能力的框架。其核心为五级成熟度模型，每级代表过程管理能力的跃升：

初始级 (Level 1 - Initial)：

特征： 过程是临时的、混乱的、不可预测的。成功高度依赖个人能力。缺乏稳定环境，项目常超期超预算。

焦点： 救火式管理，无稳定过程。

已管理级 (Level 2 - Managed)：

特征： 在项目层面建立了基本的管理过程。能够对项目的需求、计划、监督、配置、质量等进行管理。过程在类似项目上可重复。

焦点： 项目级管理，关注基本纪律（需求管理、项目计划、项目监督与控制、供应商协议管理、过程与产品质量保证、配置管理）。

已定义级 (Level 3 - Defined)：

特征： 组织拥有标准化的、裁剪适应的过程资产库。过程在组织层面被清晰定义、文档化、理解并一致执行。项目根据组织标准过程进行裁剪。

焦点： 组织级标准化过程，关注工程过程（需求开发、技术解决方案、产品集成、验证、确认）、支持过程（决策分析与解决、组织级过程定义、组织级培训）、项目管理（集成项目管理、风险管理）等。

定量管理级 (Level 4 - Quantitatively Managed)：

特征： 组织建立了对过程和产品质量的定量目标。利用统计技术和度量数据来理解过程变异，并稳定地预测过程和产品质量。

焦点： 量化管理，利用数据驱动决策（组织级过程性能、定量项目管理）。

优化级 (Level 5 - Optimizing)：

特征： 组织持续基于量化反馈，识别过程改进机会，并采用创新方法和技术进行持续优化。关注预防缺陷和提升效率。

焦点： 持续过程改进（组织级绩效管理、因果分析与解决）。

二、个人过往开发过程成熟度评估

回顾参与的课程大作业（如数据库系统、操作系统）、结合 CMMI 模型进行反思，我认为我的软件过程成熟度总体上处于 CMMI Level 1 (初始级) 向 Level 2 (已管理级) 过渡的阶段，部分项目勉强达到 Level 2 的局部要求，但远未达到 Level 2 的全面要求

Level 1 (初始级) 特征显著存在：

过程高度临时性： 大部分项目启动时缺乏明确、文档化的计划。任务分配、时间估算、风险识别往往在第一次小组会议口头讨论后即开始编码，计划文档（若有）也极其简陋且不更新。

需求管理薄弱： 课程作业需求通常由老师提供，看似明确，但在实现过程中常因理解偏差或技术难点而自行“打折”或变更。大创项目需求初期模糊，在开发过程中频繁、随意变更，缺乏记录、评审和影响分析。（Level 2 需求管理关键实践缺失）

2 配置管理实践初步使用但远不规范

质量保证缺失： 测试通常是编码完成后的“附加动作”。单元测试覆盖率极低（甚至为零），集成测试和系统测试主要靠人工点击和肉眼观察。几乎没有正式的质量评审（如设计评审、代码走查）。Bug 修复过程随意。（Level 2 过程与产品质量保证实践严重缺失）

项目监控形式化： 进度跟踪主要靠口头询问或简单的任务列表（如 Trello 看板），缺乏基于计划的定量监控（如 Earned Value）。风险识别后常无应对计划。（Level 2 项目监督与控制实践部分存在但肤浅）

Level 2 (已管理级) 的初步尝试与局部实现：

项目计划意识萌芽： 在后期项目（特别是大创）中，开始有意识制定相对详细的项目计划，包含任务分解、初步时间表、人员分工（使用工具如甘特图、在线看板）。虽然计划常与实际脱节且更新不及时，但已开始实践。（Level 2 项目计划实践的雏形）

版本控制工具普及： Git 成为标配（主要用 GitHub/Gitee），初步实现了代码的集中管理和历史追溯，避免了完全的手工备份。（Level 2 配置管理的核心工具应用）

虽然我在一些项目中应用了基本的项目管理过程，但这些过程还不够完善和规范。例如，我的项目计划往往不够详细，缺乏对风险的评估和应对措施；我虽然使用了版本控制系统，但对分支管理和合并策略的应用还不够熟练；我进行的测试主要是功能测试，缺乏对性能、安全等方面的测试；我没有对项目过程进行有效的总结和改进，导致类似的问题在不同项目中重复出现。

三、基于现有成熟度的过程改进计划

当前我的软件过程成熟度仍处于 CMMI 初始级向已管理级过渡的阶段，过程中存在明显的临时性和随意性。为系统性提升能力，改进目标将聚焦于建立项目级的基础管理规范，重点突破需求管理。以下是具体的改进路径：

1. 需求管理规范化

过往项目中需求变动频繁且缺乏追踪，导致功能遗漏或偏离。改进方向包括：强制推行需求文档化，即使是小型项目也需明确记录核心功能、用户场景和验收标准；建立简易变更流程，任何需求调整必须通过团队评审，记录变更原因及影响范围（可利用 GitHub Issues 或钉钉协作表格实现）；同时尝试建立需求追溯表，将需求条目与设计文档、测试用例关联，确保端到端覆盖。

2. 计划与监控的实效性提升

避免“计划仅存于立项时”的现状。改进关键在于：制定分阶段可执行计划，基于任务拆解（WBS）估算时间并分配责任人，在飞书文档或 Trello 看板中动态维护；推行双周进度闭环机制——每两周召开 15 分钟站会核对进度偏差，每月进行里程碑评审并更新计划；引入可视化监控工具如燃尽图（通过 GitLab 自带功能生成），量化跟踪完成度；同步启动风险管理实践，在计划阶段识别 3-5 项关键风险（如技术选型风险、依赖延误）并预设应对方案。

3、 质量保证机制落地

扭转“测试即事后补救”的惯性：代码评审制度化，所有合并到主干的代码必须经他人审查（利用 GitHub PR 机制强制执行）；推行自动化质量门禁，在 CI 流水线（如 GitHub Actions）中集成单元测试（Pytest/JUnit）、静态检查（SonarQube）和构建验证，失败则阻断合并；分层测试策略方面，要求核心模块单元测试覆盖率达 60%以上，接口测试覆盖主业务流程，UI 测试至少保障登录/核心功能路径；最后，在版本发布前进行质量会议，基于缺陷分布和测试报告决策是否放行。