什么是SDD（规格驱动开发）？

Spec-Driven Development（规格驱动开发）介绍材料

> 整理自以下 11 份原始资料（均收录于 raw/ 目录）：
>
> - Thoughtworks Technology Radar（2025年11月）
> - Birgitta Böckeler（Thoughtworks 卓越工程师），博客文章 *Understanding Spec-Driven Development: Kiro, spec-kit, and Tessl*
> - GitHub spec-kit 官方 README
> - OpenSpec（Fission AI）官方 README
> - BMAD METHOD（bmad-code-org）官方 README
> - Kevin Ryan，《Spec Driven Development: AI Native Software Engineering》（2026，早期 beta 版）及 sddbook.com
> - Bezael Pérez，《Spec-Driven Development: Build With AI Without Losing Control》（2026）
> - Enrico Papalini，Medium 文章 *The Book I Didn't Plan to Write: Why Spec-Driven Development Deserves Its Own Manual*
> - Udemy 课程 *Spec-Driven Development: From Vibe-Coding to AI Engineering*
> - O'Reilly 培训 *Spec-Driven Development with Claude Code*（讲师：Harshit Tyagi）

一、定义

各方定义汇总

来源	定义表述	核心侧重
Thoughtworks Technology Radar（2025年11月）	"SDD 是一种新兴的 AI 辅助编码工作流方式。定义仍在演化中，通常指以结构化的功能规格说明为起点，再经过多步骤将其分解为更小的片段、解决方案和任务的工作流。"	结构化规格 → 逐步分解

来源	定义表述	核心侧重
Birgitta Böckeler（Thoughtworks）	"SDD 意味着在使用 AI 编写代码之前先写一份'规格说明'（文档优先）。规格说明成为人和 AI 共同的真相来源。"	文档优先；规格作为真相来源
GitHub spec-kit	"SDD 颠覆了传统软件开发。数十年来代码是核心——规格说明只是写完就丢的脚手架。SDD 改变了这一切：规格说明变得可执行，直接生成可运行的实现，而不只是指导实现。"	规格可执行；维护软件即维护规格
Tessl（经 Birgitta 引用）	"一种以规格说明——而非代码——为主要制品的开发方式。规格说明用结构化、可测试的语言描述意图，代理生成代码与其匹配。"	规格是主制品；代码由规格生成
Kevin Ryan（sddbook.com / 书）	"SDD 反转了与 AI 协作时的传统软件开发模式：不再是思考→编码→文档，而是规格→生成→验证。规格说明是主要工作成果，生成的代码是可再生成、可修改、可丢弃的派生输出。规格是制品，代码是副作用。这不是提示工程，提示是战术性的，SDD 是一种专业方法论——一种使规格可执行且可强制执行的架构模式。"	规格为制品；代码为副作用；专业方法论
Bezael Pérez	"SDD 是一个方法：你先写规格说明，再让 AI 构建，然后审查输出，如果有什么不对，先更新规格再回到 AI。规格是你面对不确定性的 AI 时唯一的确定性：一份书面的、有版本的文档，让代理对照它做每一个决定。"	规格是对抗 AI 不确定性的确定性锚
Enrico Papalini	"我们正在从'代码是真相来源'的世界转向'意图是真相来源'的世界。规格说明书是成功或失败在最上游就被决定的地方，其后一切都是下游。"	意图即真相来源；规格决定成败
Udemy 课程	"SDD 是一门将 AI 从聪明的代码生成器变成可预测的工程能力的纪律。让意图成为记录系统，将意图转化为可执行的计划和任务，并让 AI 代理与你的架构、标准和质量要求保持一致。"	意图系统化；工程控制
O'Reilly 培训（Harshit Tyagi）	"SDD 在实现之前先定义意图，通过 spec → plan → work → review 的工作流提升正确性、安全性和可维护性。"	意图先行；规格驱动工作流
OpenSpec（Fission AI）	"AI 编码助手功能强大，但当需求只存在于聊天记录中时就难以预测。OpenSpec 添加了一个轻量级规格层，在编写任何代码之前，让你和 AI 就要构建的内容达成一致。"	轻量级规格层；先对齐再编码

各方定义的差异与共识

✅ 共识点

所有来源都认同以下核心：

- **规格说明优先于代码**（spec-first），先写规格，再让 AI 编写代码；
- **规格说明是人与 AI 之间的契约和沟通媒介**；
- **SDD 是对"vibe coding"（随兴编码、靠感觉编码）的改进**，带来可预测性和可重复性。

⚠️ 差异点

差异 1：规格说明的持久化程度

Birgitta Böckeler（Thoughtworks）明确提出了三个层级，其他来源通常并不作此区分：

层级	含义	哪些工具/作者达到此层级
Spec-first（规格优先）	任务开始时写规格，完成后可丢弃	大多数工具和作者（Kiro、OpenSpec、Papalini、Udemy 课程、O'Reilly 培训、Pérez）
Spec-anchored（规格锚定）	任务完成后保留规格，后续演进时继续使用	GitHub spec-kit（声称但实践上存疑，见 Birgitta 分析）
Spec-as-source（规格即源码）	规格是持续维护的主制品，代码完全由规格生成，人不直接修改代码	Tessl、Kevin Ryan（sddbook.com 立场）

Thoughtworks/Birgitta 的结论是：**目前大多数 SDD 工具实际上只是 spec-first，而非 spec-anchored 或 spec-as-source，尽管部分工具声称后者。**

差异 2：规格说明的重量级程度

- **OpenSpec**：明确倡导"fluid not rigid"（流动不僵化），轻量，适合棕地（brownfield）项目；
- **GitHub spec-kit**：重量级，一个功能会生成大量 markdown 文件（spec.md、plan.md、tasks.md、data-model.md、research.md 等）；
- **Bezael Pérez**：极简主义，"一个 .md 文件，仅此而已"；
- **Kevin Ryan / Papalini / Udemy 课程**：中间立场，四阶段工作流（规格→计划→任务→实现）。

差异 3：代码的定位

- **Kevin Ryan、Tessl**：代码是"副作用"（side effect），最终目标是只维护规格；
- **GitHub spec-kit**：代码是"最后一步"（last-mile approach）；
- **Thoughtworks/Birgitta**（批判性视角）：上述愿景与实际情况存在差距，目前 AI 的不确定性使得"规格即源码"面临挑战。

差异 4：工作流阶段划分

来源	阶段划分
Kevin Ryan、Papalini、Udemy 课程	Specify → Plan → Tasks → Implement（4阶段）
O'Reilly 培训	Spec → Plan → Work → Review（4阶段）
GitHub spec-kit	Constitution → Specify → Plan → Tasks → Implement（5步，含宪法）
Kiro（AWS）	Requirements → Design → Tasks（3步）
OpenSpec	Propose → Apply → Archive（3步）
BMAD	未明确标榜 SDD，但有 34+ 结构化工作流（PM、架构师、开发者等多专家角色协作）

差异 5：BMAD 与 SDD 的关系

BMAD（Breakthrough Method for Agile AI Driven Development）在原始资料中**并不将自己定义为 SDD 工具**，而是定位为"敏捷 AI 驱动开发框架"。它被 Bezael Pérez 在书中列为与 OpenSpec、spec-kit 并列的"工具无关流程"之一，可与 SDD 结合使用，但本身更强调多专家代理协作和规模自适应，而非规格说明的主制品地位。

二、特点

1. **规格说明优先**：在编写任何代码之前，先用自然语言（通常为 Markdown 文件）描述"要构建什么"和"为什么构建"，重功能意图而非技术实现细节。
2. **规格是人与 AI 的共同真相来源**：规格说明书既是人类团队成员之间的沟通媒介，也是约束 AI 代理行为的"契约"。
3. **结构化多阶段工作流**：通常遵循"规格→计划→任务→实现"的门控工作流，每个阶段有明确产物，防止过早进入下一阶段。
4. **治理层（宪法/原则）**：通过"宪法"（constitution）文件定义项目级不可变原则，约束所有功能、所有 AI 代理的行为，确保一致性。
5. **版本控制的活文档**：规格说明书随项目演进而更新，纳入版本控制，保持与代码的可追溯性。

6. **渐进式澄清机制**：在规格书写完成后、技术规划开始前，有结构化的澄清工作流（clarification workflow），在技术决策嵌入前消除歧义。
7. **人机协作中人居主导**：人负责写规格、审查规格质量、验证 AI 输出是否符合规格，AI 负责将规格转化为代码；最终判断权在人。
8. **可追溯性（Provenance）**：Kevin Ryan 的方法论明确引入"出处追踪"（provenance chain），使每一段生成的代码都可追溯到其对应规格。

三、优势

1. **可预测性与可重复性**：有了书面规格，同样的功能可以反复可靠地生成，不再依赖随机的"灵感提示"。
2. **减少返工**：规格说明在编码前消除歧义，让错误在最廉价的阶段被发现，而不是在实现完成后。
3. **跨会话的上下文保持**：AI 会话之间上下文丢失是常见问题，规格文件作为外部持久化的"记忆"，确保 AI 代理在新会话中能够重新对齐意图。
4. **防止 AI 漂移（Drift Detection）**：规格定义了可接受行为的边界，使得 AI 引入未经批准的依赖或架构反模式变得可检测。
5. **提升可维护性与可审计性**：生成的代码有清晰的"来源文档"，便于他人理解意图，降低长期维护成本。
6. **团队协作的共同语言**：规格文件为开发者、产品经理、QA 提供统一的沟通载体，取代了模糊的口头需求。
7. **应对 AI 不确定性的确定性锚**：AI 输出本质上不确定，而书面规格提供了确定的验收标准，是评估输出质量的客观依据（Pérez 的核心论点）。
8. **可扩展到团队和企业**：宪法模式使组织级原则（安全要求、技术标准、合规要求）可以系统性地应用到所有 AI 辅助开发中。

四、劣势

以下劣势主要来自 **Birgitta Böckeler（Thoughtworks）的批判性分析**，是目前对 SDD 最系统的质疑，值得重点关注：

1. **前期工作量大**：在真正开始编码之前，需要创建和审查大量规格文档，尤其是 GitHub spec-kit 会生成数量繁多的 Markdown 文件，审查成本高。
2. **对任务规模缺乏适应性**：现有 SDD 工具（如 Kiro、spec-kit）提供的是单一的重型工作流，不区分任务大小。对一个简单 bug 用 SDD 工作流，就像"用大锤敲坚

果"（sledgehammer to crack a nut）。

3. **虚假的控制感**：即使有详细的规格说明、宪法和检查列表，AI 代理也不能保证完全遵守所有指令。规格说明的存在不等于 AI 会按规格行事，可能让团队高估了对 AI 的控制程度。
4. **功能规格与技术规格的边界模糊**：在"纯功能规格"与"包含技术细节的规格"之间划线是困难的，现有工具的文档和教程对此也不一致。
5. **目标用户不清晰**：SDD 工具的演示通常涵盖产品目标定义、用户故事撰写等本属于产品经理职能的工作，但工具却以开发者为目标用户，角色定位模糊。
6. **对棕地项目不够友好**：多数 SDD 工具的教程基于从零开始的新项目，将其引入已有代码库难度更大（OpenSpec 是例外，明确针对棕地场景设计）。
7. **"幸存者偏差"风险**：SDD 社区中的成功案例可能主要来自新项目，对复杂遗留系统的实际效果尚待验证。
8. **历史教训（MDD 的前车之鉴）**：Birgitta 指出，Tessl 的"规格即源码"理念与上世纪的模型驱动开发（MDD）高度相似，而 MDD 最终未能普及，原因是其处于一个尴尬的抽象层级，产生了过多的额外负担和限制。LLM 虽然移除了部分约束，但引入了不确定性，"可能同时带来 MDD 和 LLM 的双重缺点：既不灵活，又不确定"。

五、适用场景

场景	说明	推荐工具/框架
新项目（Greenfield）从零构建	SDD 最自然的应用场景，可以从一开始就建立规格体系和宪法	GitHub spec-kit、BMAD、OpenSpec
现有项目功能增强（Brownfield）	规格说明帮助 AI 理解现有代码库边界，防止引入破坏性变更	OpenSpec（专为棕地设计）、spec-kit（需要更多配置）
遗留系统现代化	先对现有功能编写规格，再逐步迁移，防止"继承旧债"	Kevin Ryan 书中有专门章节；Udemy 课程 Module 7
多人团队协作	规格作为共同语言，减少人与人之间、人与 AI 之间的歧义	GitHub spec-kit（有宪法机制和团队协作设计）
对 AI 生成代码质量有严格要求的项目	通过宪法、验收标准和审查门控确保代码质量	spec-kit、BMAD（有测试架构模块）
中大型功能开发（3-5 point 以上故事）	Birgitta 指出，SDD 工具对小任务过于繁重，适合有一定规模的功能	根据工具重量级程度选择
需要长期上下文保持的项目	跨会话、跨开发者保持 AI 对系统理解的一致性	任何实现了规格文件版本控制的 SDD 工具

场景	说明	推荐工具/框架
企业级合规和治理要求高的项目	宪法机制可以强制执行安全要求、技术标准等组织级约束	GitHub spec-kit（有详细宪法模板）

不适合 SDD 的场景

- **探索性原型（Spike/POC）**：目标不明确时，写规格反而增加摩擦；可以明确跳过规格阶段。
- **极小任务（simple bug fix）**：Birgitta 的实验表明，用完整 SDD 工作流修复一个简单 bug 是杀鸡用牛刀。OpenSpec 的 TinySpec 扩展试图解决此问题（单文件轻量工作流）。
- **需求极度不稳定的初期阶段**：如果业务方向每天变化，维护规格文档的成本会超过收益。

六、相关框架对比摘要

框架	主体	SDD 层级	重量级程度	特色
GitHub spec-kit	GitHub	Spec-anchored（声称），实践上偏 spec-first	重	宪法机制；大量结构化 Markdown；丰富社区扩展
OpenSpec	Fission AI	Spec-first	轻	流动不僵化；专为棕地优化；25+ AI 工具支持
BMAD METHOD	bmad-code-org	未自称 SDD；敏捷 AI 驱动	中重	12+ 专家角色代理；规模自适应；完整项目生命周期
Kiro（AWS）	Amazon	Spec-first	中	IDE 集成（VS Code 发行版）；三步工作流（需求→设计→任务）
Tessl	Tessl	Spec-as-source（实验性）	重	规格与代码双向同步；生成代码注释"DO NOT EDIT"；私测阶段

本文档整理于 2026年4月21日，基于截至整理时可获取的资料。SDD 领域仍在快速演化，定义和工具均处于活跃变化中。