1. ISO/IEC 12207:2010: Жизненный цикл ПО. Группы процессов ЖЦ.

ISO/IEC 12207:2010：软件生命周期。 生命周期过程组。

生命周期：

Requirements development 需求开发

Analysis 分析

Design 设计

Development 发展

Testing 测试

Implementation 执行

Exploitation开发

removal from service 从服务中删除

启动过程组：作用是设定项目目标，让项目团队有事可做  
规划过程组：作用是制定工作路线，让项目团队“有法可依”  
执行过程组：作用是“按图索骥”，让项目团队“有法必依”  
监控过程组：作用是测量项目绩效，让项目团队“违法必究”，并且尽量做到“防患于未然”  
收尾过程组：作用是了结项目（阶段）“恩怨”，让一切圆满

1. Модели ЖЦ (последовательная, инкрементная, эволюционная).

生命周期模型（顺序、增量、进化）

定义整个生命周期完成顺序以及进程，操作，任务之间的相互关系的结构

Последовательная:

​ 定义所有要求，一次性完成所有代码

Инкрементная:

​ 定义所有要求（分几步），分几步完成

Эволюционная:

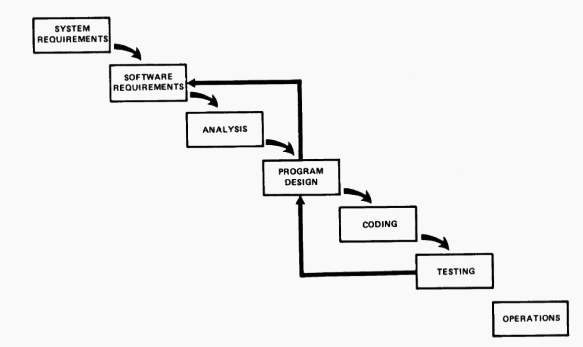
​ 几步完成，没有定义所有的要求

1. Водопадная (каскадная) модель.

瀑布模型 waterfall

开发过程是通过设计一系列阶段顺序展开的，如果有信息未被覆盖或者发现了问题，那么最好 “返回”上一个阶段并进行适当的修改

System requirement -> software requirement -> analysis -> program design -> coding -> testing -> operations



1. Методология Ройса.

Royce模型

普通的瀑布模型，如果我们在testing环节出现错误，一般不能仅仅靠改动几行代码来解决，通常需要重新设计。但设计上的改动可能是毁灭性的，甚至可能连需求分析都被波及，这时候要么重新设计，要么更改需求，甚至从头来过，这意味着巨大的额外投入与日程延期

阶段间的交互不可能被局限在前后相继的顺序上

改进方面：

preliminary desig初步设计

需求生成阶段与分析阶段插入一个预设计阶段，我们可以提前确保程序不会因为时间，储存或者数据流方面出现问题而失败，而在分析时，确保设计师把限制提交给分析师面前，让分析师分析时必须考虑这些限制。

documentation文档

管理需要大量的文档，没有足够的文档，根本无法对项目做出有效的管理。

do it twice做两次

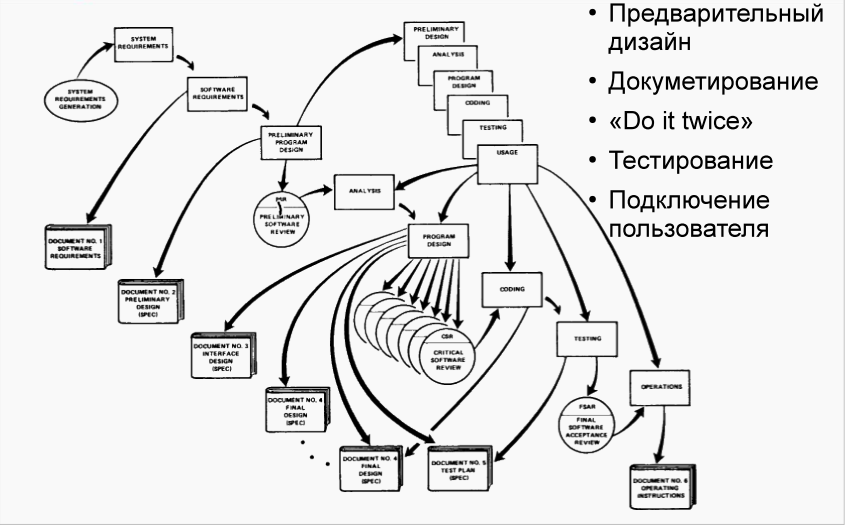
如果是第一次开发某个计算机程序，先整理相关问题，考虑设计/运营方面的问题，再做一次，确保交付给客户的实际上是第二个版本

testing测试

有很多错误是可以通过视觉上的审查来发现的，应该把分析和设计提交给除分析者和编码者之外的人来评审

user connection用户连接

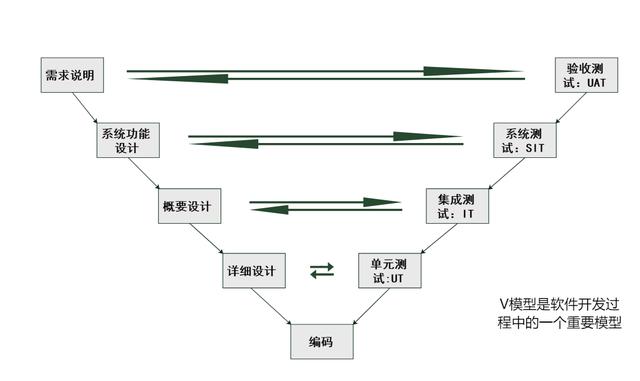
使用户尽早参与开发过程，可以澄清一些不确定的问题，初步软件审评，关键软件审评，最终软件审评



1. Традиционная V-chart model J.Munson, B.Boehm.

传统 V 型图模型 J.Munson、B.Boehm

V-Chart模型是由J.Munson和B.Boehm提出的一种传统软件开发生命周期模型。该模型强调软件开发的迭代和增量式方法

顺序基于瀑布模型，但是给每一个步骤加上了属于自己的测试。模块，集成，系统，使用测试。右边是动态测试（包含阿尔法测试），左边是静态测试（非功能性）。

1. Многопроходная модель (Incremental model).

多遍模型（增量模型）

增量模型是把待开发的软件系统模块化，将每个模块作为一个增量组件，从而分批次地分析、设计、编码和测试这些增量组件。相对于瀑布模型而言，采用增量模型进行开发，开发人员不需要一次性地把整个软件产品提交给用户，而是可以分批次进行提交。

1. Модель прототипирования (80-е).

原型模型

原型是一个可以实际运行、反复修改，可以不断完善的系统

很多时候，客户定义了软件的一些基本任务，但是没有详细定义功能和特性需求。另一种情况下，开发人员可能对算法的效率、操作系统的适用性和人机交互的形式等情况并没有把握。在这些情况和类似情况下，采用原型开发模型。

1. RAD методология.

RAD方法论

RAD模型是指快速应用开发，它是一种增量模型,组件或功能是平行开发的，它可以处理小型项目和中型项目。在这种模式下，任何变化都可以在任何阶段进行。由于人数较少，它具有较高的生产率。

1. Спиральная модель.

螺旋模型

螺旋的每一圈都代表产品开发的一个阶段，旨在构建下一个版本的软件或原型。 首先，为每次迭代设定目标，确定在此阶段需要考虑的备选解决方案和约束条件。

螺旋模型最大的特点在于引入了其他模型不具备的风险分析，使软件在无法排除重大风险时有机会停止，以减小损失

1. UML Диаграммы: Структурные и поведенческие.

UML 图：结构和行为。

UML（统一建模语言）是一种广泛使用的软件工程工具，它支持以图形化方式表示和描述软件系统的各个方面。UML包含多种类型的图表，可以分为两大类：结构图和行为图。

结构图用于描述系统的静态结构和组成部分，包括系统的类、对象、接口、包等。常见的结构图有以下几种：

·类图（Class Diagram）：描述系统中类的结构和它们之间的关系，如继承、关联、聚合等。

·对象图（Object Diagram）：展示特定时刻系统中对象的状态以及它们之间的关系。

·组件图（Component Diagram）：表示系统的物理组件和它们之间的依赖关系，用于展示系统的实现层面。

行为图用于描述系统的动态行为，包括系统的交互、状态变化、事件触发等。常见的行为图有以下几种：

·用例图（Use Case Diagram）：展示系统的功能需求和用户之间的交互。

·活动图（Activity Diagram）：描述系统中业务流程或算法的控制流程。

·时序图（Sequence Diagram）：表示对象之间消息交互的时间顺序，强调对象之间的交互和时序。

1. UML: Use-case модель.

UML：用例模型。

UML中的Use-case模型是用例图（Use Case Diagram）的概念，它是一种用于描述系统功能需求和用户之间交互的图表。用例图是UML中最常见的图之一，用于在需求分析阶段捕捉系统的功能需求，帮助团队理解系统的功能和用户与系统之间的交互。

用例图主要由以下几个元素组成：

·用例（Use Case）：用例代表了系统中的一个功能或一个用户场景。每个用例描述了一个特定的系统功能，描述了系统对外的行为。用例通常以动词开始，例如"登录系统"、"查询订单"等。

·参与者（Actor）：参与者是系统的外部用户或其他系统，它们与系统进行交互以实现一个或多个用例。参与者可以是人、其他系统或硬件设备等。

·关系（Relationship）：用例图中的关系表示参与者和用例之间的联系。

常见的关系有：

1关联关系（Association）：表示参与者与用例之间的关联，通常用一条实线连接。

2包含关系（Include）：表示一个用例包含另一个用例，表示为一个用例的箭头指向另一个用例。

3扩展关系（Extend）：表示一个用例可以扩展另一个用例的功能，表示为一个用例的箭头指向另一个用例。

1. UML: Диаграмма классов.

UML：类图。

UML中的类图（Class Diagram）是一种用于表示系统静态结构的图表，它是UML中最常见、最重要的图之一。类图描述了系统中的类、对象、接口以及它们之间的关系，是面向对象分析和设计的重要工具。

类图主要由以下几个元素组成：

·类（Class）：类是面向对象编程中的基本概念，用于表示具有相似属性和行为的一组对象。类由类名、属性和方法组成。

·对象（Object）：对象是类的实例，表示具体的实体。类图中通常不直接显示对象，而是通过类来表示对象的属性和行为。

·属性（Attribute）：属性是类的特征或数据成员，用于描述类的状态。在类图中，属性通常以名称：类型的形式表示，例如"姓名：字符串"。

·方法（Method）：方法是类的行为或操作，用于描述类可以执行的操作。在类图中，方法通常以名称(参数列表)：返回类型的形式表示，例如"登录(用户名, 密码)：布尔型"。

·关系（Relationship）：类图中的关系表示类之间的联系和依赖关系。常见的关系有：

1继承关系（Inheritance）：表示一个类继承自另一个类，用于表示类之间的继承关系，通常用一个空心三角箭头表示。

2关联关系（Association）：表示类之间存在关联关系，用于表示类之间的关联，通常用一条实线连接。

3聚合关系（Aggregation）：表示整体与部分之间的关系，用于表示类之间的聚合关系，通常用一条带空心菱形的实线连接。

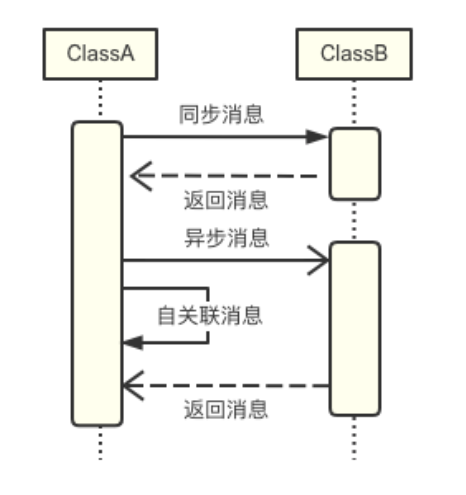
4组合关系（Composition）：表示整体与部分之间的强关系，用于表示类之间的组合关系，通常用一条带实心菱形的实线连接。

5依赖关系（Dependency）：表示一个类依赖于另一个类，用于表示类之间的依赖关系，通常用一条带箭头的虚线连接。

1. UML: Диаграмма последовательностей

UML：序列图

序列图（Sequence Diagram），亦称为循序图、时序图，是一种UML行为图。描述物件在时间序列中的交叉作用。序列图会描绘在此情境下有关的物件，以及此物件和其他物件交换讯息的顺序。序列图一般和待开发系统逻辑视图上，用例的实现有关。序列图有时也称为事件图或事件情境。



1. UML: Диаграмма размещения

UML：布局图

1. \*UP методологии (90-е). RUP: основы процесса.

UP 方法（90 年代）。 RUP：流程基础知识。

统一过程：是一种现代的软件开发过程模型。它是基于构件（Component-based）的，即所构造的软件系统是由软件构件通过明确定义的接口相互链接所建造起来的。并且它使用统一建模语言（Unified Modeling Language，UML）来制定系统的所有蓝图。

Inception（开始阶段）、Elaboration（设计阶段）、Construction（产品创建阶段）和Transition（客户端的产品实施阶段）

任何开发的组织不仅需要每个角色遵守正式的工作流程，还需要对履行职责的特殊态度，这被称为“RUP 精神”。

1尽早发现风险2 注意用户的需求3 专注代码4组件构造系统5 尽早设计架构6 团队合作7注意质量

1. RUP: Фаза «Начало».

开始阶段：主要工作方向是评估项目，了解需要花费多少资源和时间，以及用户有哪些问题以及项目可以解决哪些问题，在这个阶段中，我们需要关注整个项目进行中的业务逻辑和需求方面的主要风险

1. RUP: Фаза «Проектирование».

设计阶段：主要任务是开发和测试一个稳定不变的系统架构，创建一个或多个定义可执行架构的系统原型，淘汰高风险的元素

1. RUP: Фаза «Построение».

产品创建阶段：在构建阶段之前，必须开发应用程序架构； 此时应该排除对其进行重大修改的可能性。 此阶段的主要目标是尽可能快地开发具有适当质量且具有成本效益的软件产品。所有的功能都被详细的测试。构架阶段是一个创造的过程。

1. RUP: Фаза «Внедрение».

客户端的产品实施阶段：确保软件对用户是用的，由用户最终确认产品是否适合他们的实际需求。

1. Манифест Agile (2001).

敏捷宣言

我们的首要任务是通过定期和及早交付有价值的软件来满足客户的需求。

欢迎更改需求，即使是在开发后期。 敏捷流程允许您使用变更为客户提供竞争优势。

工作产品应尽可能频繁地发布，频率为几周到几个月。

在整个项目中，开发人员和业务代表必须每天一起工作。

1. Scrum.

Scrum 主要且唯一的服务工件是 backlog（来自英文。Backlog - 债务）——按优先级排序的需求列表，其中包含对开发劳动强度的评估。

Scrum的核心理念是将复杂的项目划分为一系列较小、可管理的时间段，称为Sprint（冲刺）。每个Sprint通常持续2至4周，其间团队集中精力完成特定功能或目标。

1. Disciplined Agile 2.X (2013).

Scott Ambler 在 2013 年提出了一种新方法——Disciplined Agile 2.X (DAD)。 DAD 中的阶段和规程方法与 RUP 非常相似，但主要开发周期基于敏捷方法，包括 Scrum。

1. Требования. Иерархия требований.

要求。 需求的层次结构。

在软件开发和项目管理中，"Требования"（需求）是指对产品或项目的功能、性能、限制和其他特性的描述和规范。需求是项目成功的基础，它们对于理解和满足客户的期望，以及确保项目交付符合预期目标非常重要。

需求可以分为不同层次，形成一种"Иерархия требований"（需求层次结构）。这种层次结构使得需求可以更加系统地组织和管理，从而更好地满足项目的目标。

在需求层次结构中，一般有以下几个层次：

·业务需求（Business Requirements）：这是最高层次的需求，通常由项目的利益相关者（Stakeholder）提供。业务需求描述了项目的高层目标和对业务价值的贡献，强调项目为何被启动以及预期的商业效益。

·用户需求（User Requirements）：用户需求是从业务需求派生出来的，它们更加具体，描述了用户对系统或产品的功能和性能的期望。用户需求侧重于从最终用户的角度描述需求。

·功能需求（Functional Requirements）：功能需求是用户需求的进一步细化，详细描述系统或产品应该提供的具体功能。功能需求通常以用户故事、用例或功能规范的形式来表达。

·非功能性需求（Non-functional Requirements）：非功能性需求描述了系统或产品的非功能性特性，如性能、安全性、可用性、可靠性等。它们不直接关联具体的功能，但对于确保系统的质量和用户体验至关重要。

1. Свойства и типы требований (FURPS+).

要求的属性和类型 (FURPS+)。

FURPS+ 是一种常用的软件需求分类模型，用于帮助团队全面地考虑和分类各种需求。它将软件需求划分为五个主要类别，每个类别由首字母缩写表示：

·Functionality（功能性）：描述了软件系统应该提供的功能和行为。这包括用户界面、系统的操作、数据处理和业务逻辑等。功能性需求是满足用户期望的核心需求。

·Usability（可用性）：描述了软件的易用性和用户体验。可用性需求关注用户界面的友好性、操作的简单性、帮助文档的清晰性等，旨在提供良好的用户体验。

·Reliability（可靠性）：描述了软件的稳定性和可靠性。可靠性需求关注系统的稳定性、容错性、恢复能力，以及在面对异常情况时的表现。

·Performance（性能）：描述了软件的性能要求。性能需求涉及响应时间、吞吐量、并发性等指标，旨在确保软件在运行时能够满足性能要求。

·Supportability（支持性）：描述了软件的可维护性和可支持性。支持性需求涉及软件的易维护性、可扩展性、文档化程度，以及与其他系统的集成等。

1. Требования к удобству использования и надежности.

可用性和可靠性的要求。

对于软件系统的用户体验和可靠性，通常有两类重要的需求：用户体验（Usability）和可靠性（Reliability）。

1用户体验（Usability）需求：这些需求关注软件系统的易用性和用户体验。用户体验是指用户在使用软件时的感受和互动。以下是一些用户体验需求的例子：

用户界面应该简洁明了，易于理解和导航。

操作流程应该简单和直观，减少用户的学习成本。

系统应该提供明确的反馈和状态指示，让用户了解其当前状态。

字体和颜色应该易于阅读和识别，避免眼睛疲劳。

系统应该支持不同的设备和屏幕尺寸，以便用户在不同平台上获得一致的体验。

2可靠性（Reliability）需求：这些需求关注软件系统的稳定性和可靠性。可靠性是指系统在特定时间段内保持正常运行的能力。以下是一些可靠性需求的例子：

系统应该在正常工作状态下保持高可用性，尽量避免系统宕机或崩溃。

系统应该能够正确处理和处理异常情况，避免出现未处理的错误。

系统应该能够正确保存和恢复用户数据，以防止数据丢失或损坏。

系统应该具有良好的性能，保证响应时间合理，不会让用户感到卡顿或延迟。

1. Требования к производительности и поддерживаемости.

性能和可维护性要求。

对于软件系统的性能和可维护性，通常有两类重要的需求：性能需求（Performance）和可维护性需求（Maintainability）。

性能需求（Performance）：这些需求关注软件系统在运行时的性能表现，包括响应时间、吞吐量、资源利用率等方面。以下是一些性能需求的例子：

·系统的响应时间应该在用户可接受的范围内，确保用户能够快速地获取结果。

·系统的吞吐量应该足够高，能够同时支持大量用户的并发访问。

·系统在处理大数据量时应该保持高效，避免因数据量过大而导致性能下降。

·系统应该有效利用计算资源，确保在满足性能要求的前提下节约资源消耗。

可维护性需求（Maintainability）：这些需求关注软件系统的易维护性和可扩展性，确保系统在未来能够持续演进和更新。以下是一些可维护性需求的例子：

·系统的代码应该清晰、可读性高，方便团队成员理解和维护。

·系统的模块应该相互独立，确保修改一个模块不会影响其他模块的功能。

·系统应该有适当的文档和注释，方便新成员快速上手和了解系统结构。

·系统的架构应该具有良好的可扩展性，能够方便地引入新功能和技术。

1. Описание прецедента.

用例描述

id

краткое описание简短的介绍

Главные actor主角

Второстепенные actor次要演员

Предусловия前提条件

Основной поток主流

ppt 56

1. Риски. Типы Рисков.

风险。 风险类型

Потенциально опасный фактор

潜在危险因素

Прямые и непрямые:直接和间接

可不可控制

Ресурсные:资源

人，时间，财政

Бизнес-риски:商业风险

竞争，承包人

Технические-риски:技术风险

项目边界，技术风险，外部关系

Политические:政治的

Форс-мажор:不可抗力

1. Управления рисками. Деятельности, связанные с оценкой.

风险管理。 评估相关活动。

控制和管理风险：

计划控制，应对反应

跟踪

解决与风险有关的不确定

评估风险：

分级

分析

识别

1. Изменение. Общая модель управления изменениями.

改变。 一般变更管理模型。

控制或者文档上的系统更新。更新内容可能是Id，日期，职责，描述，更改日志等等

引入改变或者增加 -> 生成文档->产生更改请求->确定技术的必要性以及可实现性并分析花费和优点->确认请求（commit)->分析对其他部分的影响->计划->实现->测试->测试的报告->发布

1. Системы контроля версий. Одновременная модификация файлов.

版本控制系统。 同时修改文件。

控制对于代码的修改，支持团队工作。分为集中式，基于文件系统式和分布式。同时修改文件有两种方法，第一种copy-modify-merge，第二种Lock-modify-unlock

1. Subversion. Архитектура системы и репозиторий.

Subversion。 系统架构和存储库。

集中式版本控制。仓库存在一个远程服务器上。主要有trunk，branches，tag

1. Subversion: Основной цикл разработчика. Команды.

Subversion：主要的开发周期。 团队。

周期：

Create

Checkout

Update

Status

Revert

Merge

Commit

svn update

svn (add delete copy move mkdir)

svn status and svn diff

svn revert

svn update

svn resolved/svn resolve

svn commit

1. GIT: Архитектура и команды.

Git 架构和命令。

1. Git 的基本架构：

·仓库（Repository）：Git 仓库是存储代码和历史变更的地方。一个仓库可以包含完整的项目代码，包括所有分支和历史记录。

·提交（Commit）：提交是 Git 中最小的版本单位。每次对项目文件的修改都需要进行提交，形成代码的版本历史。

·分支（Branch）：分支是代码开发的不同线路。通过创建分支，可以在不影响主线代码的情况下进行新功能的开发和实验。

·主分支（Master/Main）：主分支通常是项目的主要代码线路，代表了稳定版本的代码。

·远程仓库（Remote Repository）：远程仓库是位于网络中的中央代码存储库，用于多人协作开发。

2. 常用 Git 命令：

git init：初始化一个新的本地仓库。

git clone URL：从远程仓库克隆代码到本地。

git add filename：将修改后的文件添加到暂存区。

git commit -m "commit message"：将暂存区的文件提交到本地仓库，并附上提交信息。

git status：查看仓库中文件的状态，包括已修改、已暂存和未跟踪等。

git log：查看提交历史记录。

git branch：列出本地仓库中所有分支。

git checkout branch-name：切换到指定分支。

git merge branch-name：合并指定分支到当前分支。

git pull：从远程仓库获取最新代码，并合并到当前分支。

git push：将本地分支的代码推送到远程仓库。

1. GIT: Организация ветвей репозитория.

GIT：分支存储库。

主分支（Main Branch）： 主分支通常称为"master"或"main"，是仓库中的主要代码线路。它包含了稳定版本的代码，即可随时发布的代码。在主分支上进行的更改和提交应该经过严格的代码审查和测试。

开发分支（Development Branch）： 开发分支通常称为"develop"，是用于日常开发的主要分支。在开发分支上进行新功能的开发和实验。当功能开发完成且经过测试后，可以将开发分支合并到主分支中。

功能分支（Feature Branches）： 功能分支是用于开发单个功能或任务的临时分支。每个新功能通常都会创建一个独立的功能分支，以便独立开发和测试。功能开发完成后，可以将功能分支合并到开发分支。

修复分支（Bug Fix Branches）： 修复分支是用于解决特定问题或缺陷的分支。当在主分支或开发分支上发现问题时，可以创建一个修复分支来解决问题。完成修复后，可以将修复分支合并回主分支和开发分支。

发布分支（Release Branches）： 发布分支是用于准备发布新版本的分支。在发布分支上进行最后的测试和修复，确保代码稳定性。一旦准备好发布，可以将发布分支合并到主分支，并标记版本号。

热修复分支（Hotfix Branches）： 热修复分支是用于紧急修复已发布版本中的问题的分支。当在已发布版本中发现严重问题时，可以创建热修复分支来进行紧急修复。完成修复后，可以将热修复分支合并回主分支和开发分支。

1. GIT: Плагин git-flow.

git-flow 插件。

Git-flow 是一个 Git 扩展插件，它为 Git 提供了一套高级命令，用于管理代码库中的分支和版本发布。Git-flow 插件基于一种称为 "Git-flow 工作流" 的模型，旨在简化团队协作开发和版本管理的过程。

1. Системы автоматической сборки: предпосылки появления

自动装配系统：出现的先决条件

自动化构建系统是为了解决软件开发中频繁构建和部署的问题而产生的。它们为开发团队提供了一种自动化的方法来构建、测试和部署软件，从而提高开发效率和软件质量