SOFTWARE TESTING

Chapter 1

Overview of Software Engineering

School of Data & Computer Science
Sun Yat-sen University

Approaches & Technologies

OUTLINE

敏捷开发概述

敏捷开发概述

敏捷开发的起源

- 敏捷建模 (Agile Modeling, AM) 源于 *Scott W. Ambler* 的 Extreme Modeling (XM, 2000)。2001年以 *Kent Beck*, *Alistair Cockburn*, *Ward Cunningham*, *Martin Fowler* 等人为首在 Snowbird, Utah 发布《敏捷宣言》，决定将 Agile 作为新的轻量级软件开发过程的家族名称。
- 敏捷建模是一种态度，而不是一个说明性过程。它是从软件开发过程实践中归纳总结出来的一些价值观、原则和实践。
- 敏捷建模不是一个完整的方法论，而是对已有生命周期模型的补充，在应用传统的生命周期模型时可以借鉴敏捷建模的过程指导思想。
- http://agilemanifesto.org/

敏捷开发概述

- Manifesto for Agile Software Development m(敏捷宣言)

敏捷开发概述

- Manifesto for Agile Software Development
  - We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value
    - **Individuals and Interactions** *over processes and tools*
    - **Working Software** *over comprehensive documentation*
    - **Customer Collaboration** *over contract negotiation*
    - **Responding to Change** *over following a plan*
  - That is, while there is value in the items on the right, they value the items on the left more.

敏捷开发概述

- Manifesto for Agile Software Development
  - 我们一直在实践中探寻更好的软件开发方法，身体力行的同时也帮助他人。由此我们建立了如下价值观：
    - **个体和互动** 高于 流程和工具
    - **工作的软件** 高于 详尽的文档
    - **客户合作** 高于 合同谈判
    - **响应变化** 高于 遵循计划
  - 也就是说，尽管右项有其价值，我们更重视左项的价值。

敏捷开发概述

- Manifesto for Agile Software Development
  - As *Scott Ambler* elucidated:
    - Tools and processes are important, but it is more important to have competent people working together effectively.
    - Good documentation is useful in helping people to understand how the software is built and how to use it, but the main point of development is to create software, not documentation.
    - A contract is important but is no substitute for working closely with customers to discover what they need.
    - A project plan is important, but it must not be too rigid to accommodate changes in technology or the environment, stakeholders' priorities, and people's understanding of the problem and its solution.

敏捷开发概述

- Manifesto for Agile Software Development
  - We follow these 12 principles:

(1) Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

(2) Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

(3) Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

(4) Business people and developers must work together daily throughout the project.

(5) Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

(6) The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

敏捷开发概述

- Manifesto for Agile Software Development
  - We follow these 12 principles: (cont.)

(7) Working software is the primary measure of progress.

(8) Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

(9) Continuous attention to technical excellence and good design enhances agility.

(10) Simplicity—the art of maximizing the amount of work not done—is essential.

(11) The best architectures, requirements, and designs emerge from self-organizing teams.

(12) At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

敏捷开发概述

敏捷开发概述

- 敏捷宣言遵循的12条原则：
  - (1) 我们最重要的目标，是通过尽早和持续地交付有价值的软件来使客户满意。
  - (2) 欣然面对需求变化—即使是在项目开发后期。要善于利用需求变更，帮助客户获得竞争优势。
  - (3) 经常地交付可工作的软件，相隔几个星期或几个月不等，倾向于采取较短的周期。
  - (4) 在整个项目过程中，业务人员与开发人员必须经常在一起工作。
  - (5) 激励项目人员，以他们为核心构建项目，为他们提供需要的环境和支持，并相信他们能够完成任务。
  - (6) 无论团队内还是团队间，最有效的沟通方法是面对面的交谈。
  - (7) 可工作的软件是衡量进度的主要指标。
  - (8) 敏捷过程提倡可持续的开发。项目方、开发人员和用户应该能够保持恒久稳定的进展速度。
  - (9) 对技术的精益求精以及对设计的不断完善将提升敏捷性。
  - (10) 以简洁为本。简洁是尽可能减少不必要的工作量的艺术。
  - (11) 最佳的架构、需求和设计出自于自组织团队。
  - (12) 团队要定期反省如何能提高成效，并以此调整团队的行为。

敏捷开发概述

敏捷开发概述

敏捷开发的目标

- 敏捷开发的总体目标是通过“尽可能早地、持续地对有价值软件的交付”，使客户满意。
- 敏捷开发强调软件开发应当能够对未来可能出现的变化和不确定性作出全面反应。
- 敏捷开发主要用于在需求模糊或快速变化的前提下，支持小型开发团队的软件开发活动。

敏捷开发概述

- 敏捷开发的管理原则
  - 敏捷开发是一种以人为核心、迭代、循序渐进的开发过程指导思想。
  - 在敏捷开发过程中，软件项目的构建被切分成多个子项目分别实现。各个子项目之间相互联系、独立运行，子项目的成果经过测试，具备集成和可运行的特征。

敏捷开发概述

- 敏捷开发概述
  - 敏捷开发的核心实践
    - 项目关键利益方 (Project Stakeholder) 的积极参与
    - 正确使用工件
    - 集体所有制 (对代码、工件、模型的共有，包括使用和修改)
    - 测试性思维 (比如“测试优先”)
    - 并行创建模型 (为一个问题同时建立多种模型)
    - 创建简单的内容
    - 简单地建模
    - 公开展示模型 (使用 modeling wall 向项目参与各方展示)
    - 切换到另外的工件 (遇到困难时的敏捷切换)
    - 小增量建模
    - 和他人一起建模
    - 用代码验证模型
    - 使用最简单的建模工具

敏捷开发概述

- 敏捷开发概述
  - 敏捷开发的补充实践
    - 使用建模标准 (比如 UML)
    - 逐渐应用模式 (pattern)
    - 丢弃临时模型
    - 合同模型要正式
    - 为外部交流建模
    - 为帮助理解建模
    - 重用现有的资源
    - 不到万不得已不更新模型

敏捷开发方法分类

- XP
  - XP (极限编程) 提倡测试先行，目的在于将后面出现缺陷的概率降至最低。
- Scrum
  - Scrum 是一种迭代的增量化过程，用于产品开发或工作管理。
- Crystal Methods
  - Crystal Methods (水晶方法系列) 与 XP 一样，都是以人为中心，但不同类型的项目需要不同的实践方法。
- FDD
  - FDD (特性驱动开发) 是一套针对中小型软件开发项目的开发模式，采用模型驱动的快速迭代开发过程。

敏捷开发方法分类

- ASD
  - ASD (Adaptive Software Development, 自适应软件开发) 从复杂自适应系统理论派生出来，用于对需求多变、开发周期短项目的管理。
- DSDM
  - DSDM (动态系统开发方法) 倡导以业务为核心，快速而有效地进行系统开发。
- RUP
  - RUP 是一个过程框架，它可以包容许多不同类型的过程，但核心还是面向对象过程。

Scrum

- Scrum is an agile framework for managing knowledge work, with an emphasis on software development.
  - designed for teams of 3-to-9 members.
  - their work can be break into actions that can be completed within time-boxed iterations, called *sprints*, no longer than one month and most commonly two weeks.
  - then track progress and re-plan in 15-minute time-boxed stand-up meetings, called *daily scrums*.

Scrum

- Scrum is a lightweight, iterative and incremental framework for managing product development.
  - defines “a flexible, holistic (整体的) product development strategy where a development team works as a unit to reach a common goal“.
  - challenges assumptions of the “traditional, sequential approach” (*Hirotaka Takeuchi* and *Ikujiro Nonaka* 竹内弘高和野中郁次郎, 1986) to product development.
  - enables teams to self-organize by
    - encouraging physical co-location or close online collaboration of all team members.
    - daily face-to-face communication among all team members and disciplines involved.

Scrum

- History of Scrum
  - *Hirotaka Takeuchi* and *Ikujiro Nonaka* described a new approach to commercial product development that would increase speed and flexibility, based on case studies from manufacturing firms in the automotive, photocopier and printer industries.
    - They called this the *holistic or rugby approach*, as the whole process is performed by one cross-functional team across multiple overlapping phases, where the team "tries to go the distance as a unit, passing the ball back and forth".
  - In rugby football, a scrum refers to the manner of restarting the game after a minor infraction. In the early 1990s, *Ken Schwaber* used what would become Scrum at his company, Advanced Development Methods, and *Jeff Sutherland*, with *John Scumniotales* and *Jeff McKenna*, developed a similar approach at Easel Corporation, and were the first to refer to it using the single word *Scrum*.

Scrum

Scrum

History of Scrum

- In 1995, *Sutherland* and *Schwaber* jointly presented a paper describing the *Scrum methodology* at the Business Object Design and Implementation Workshop held as part of Object-Oriented Programming, Systems, Languages & Applications '95 (OOPSLA '95) in Austin, Texas, its first public presentation.
- *Schwaber* and *Sutherland* collaborated during the following years to merge the above writings, their experiences, and industry best practices into what is now known as Scrum.
- In 2001, *Schwaber* worked with *Mike Beedle* to describe the method in the book *Agile Software Development with Scrum*.
- Its approach to planning and managing projects is to bring decision-making authority to the level of operation properties and certainties.

Scrum

- History of Scrum
  - Although the word is not an acronym, some companies implementing the process have been known to spell it with capital letters as SCRUM. This may be due to one of *Ken Schwaber*'s early papers, which capitalized SCRUM in the title.
  - While the trademark on the term Scrum itself has been allowed to lapse, so that it is deemed as owned by the wider community rather than an individual, the leading capital is retained—except when used with other words (as in *daily scrum* or *scrum team*).
  - Hybridization of scrum is common as scrum does not cover the whole product development lifecycle; therefore, organizations find the need to add in additional processes to create a more comprehensive implementation.
    - For example, at the start of the project, organizations commonly add process guidance on requirements gathering and prioritization, initial high-level design, and budget and schedule forecasting.

Scrum

- Key ideas
  - A key principle of Scrum is the dual recognition.
    - Customers will change their minds about what they want or need (often called *requirements volatility* 需求波动).
    - There will be unpredictable challenges—for which a predictive or planned approach is not suited.
  - Scrum adopts an evidence-based empirical approach (经验主义的方法)—accepting that the problem cannot be fully understood or defined up front, and instead focusing on how to maximize the team's ability to deliver quickly, to respond to emerging requirements, and to adapt to evolving technologies and changes in market conditions.

Scrum

- Roles
  - There are three core roles:
    - Product Owner
    - Development Team
    - Scrum Master

    and a range of ancillary roles (辅助角色).
  - Core roles are often referred to as *pigs* and ancillary roles as *chickens* after the story The Chicken and the Pig.
    - The chicken asks the pig if he is interested in jointly opening a restaurant. The chicken says they could call it, "Ham-and-Eggs." The pig answers, "*No thanks. I'd be committed, but you'd only be involved!*"
  - It specifies a concrete set of roles, which are divided into two groups:
    - Committed
    - Involved.

- Scrum
  - Roles
    - Committed
      - those directly responsible for production and delivery of the final product. These roles include the team as a whole, its members, the scrum master, and the product owner.
      - The core roles are those committed to the project in the Scrum process—they are the ones producing the product (objective of the project). They represent the *scrum team*.
    - Involved
      - represents the other people interested in the project, but who aren't taking an active or direct part in the production and delivery processes. These roles are typically stakeholders and managers.

Scrum

- The Product Owner
  - The product owner represents the stakeholders and is the voice of the customer. He or she is accountable for ensuring that the team delivers value to the business.
  - The product owner writes (or has the team write) customer-centric items (typically user stories), ranks and prioritizes them, and adds them to the product backlog.
  - Scrum teams should have one product owner, and while they may also be a member of the development team, this role should not be combined with that of the scrum master. In an enterprise environment, though, the product owner is often combined with the role of project manager as they have the best visibility regarding the scope of work (products).

Scrum

- The Development Team
  - The development team is responsible for delivering potentially shippable product increments at the end of each Sprint (the Sprint Goal).
  - A development team is made up of 7 +/- 2 individuals with cross-functional skills who do the actual work:
    - analysis, design, develop, test, technical communication, document, etc.
  - The development team in Scrum is self-organizing, even though there may be some level of interface with project management offices (PMOs).

- Scrum
  - The Scrum Master
    - Scrum is facilitated by a scrum master, who is accountable for removing impediments (消除障碍) to the ability of the team to deliver the sprint goal/deliverables.
    - The scrum master is not the team leader, but acts as a buffer between the team and any distracting influences.
    - The scrum master ensures that the Scrum process is used as intended. The scrum master is the enforcer of the rules of Scrum, often chairs key meetings, and challenges the team to improve.
    - The role has also been referred to as a *servant-leader* to reinforce these dual perspectives.
    - The scrum master differs from a project manager in that the latter may have people management responsibilities unrelated to the role of scrum master. The scrum master role excludes any such additional people responsibilities.

Scrum

- Workflows in the Scrum framework
  - (1) A *Sprint* (or iteration) is the basic unit of development in Scrum. The sprint is a time-boxed effort; that is, it is restricted to a specific duration. The duration is fixed in advance for each sprint and is normally between one week and one month, with two weeks being the most common.
    - Each sprint starts with a sprint planning event that aims to define a sprint backlog, identify the work for the sprint, and make an estimated forecast for the sprint goal.
    - Each sprint ends with a sprint review and sprint retrospective, that reviews progress to show to stakeholders and identify lessons and improvements for the next sprints.
    - Scrum emphasizes working product at the end of the sprint that is really done. In the case of software, this likely includes that the software has been fully integrated, tested and documented, and is potentially releasable.

Scrum

Scrum

- Workflows in the Scrum framework

  (2) At the beginning of a sprint, the scrum team holds a *Sprint Planning* event to:

  - Mutually discuss and agree on the scope of work that is intended to be done during that sprint.
  - Select product backlog items that can be completed in one sprint.
  - Prepare a sprint backlog that includes the work needed to complete the selected product backlog items.
  - Once the development team has prepared their sprint backlog, they forecast (usually by voting) which tasks will be delivered within the sprint.

Scrum

Scrum

- Workflows in the Scrum framework
  - (2) At the beginning of a sprint, the scrum team holds a *Sprint Planning* event to:
    - The recommended duration is four hours for a two-week sprint (pro-rata for other sprint durations)
      - During the first half, the whole scrum team (development team, scrum master, and product owner) selects the product backlog items they believe could be completed in that sprint.
      - During the second half, the development team identifies the detailed work (tasks) required to complete those product backlog items; resulting in a confirmed sprint backlog.
      - As the detailed work is elaborated, some product backlog items may be split or put back into the product backlog if the team no longer believes they can complete the required work in a single sprint.

Scrum

- Workflows in the Scrum framework
  - (3) Each day during a sprint, the team holds a *Daily Scrum* (or stand-up) with specific guidelines:
    - All members of the development team come prepared. The daily scrum:
      - starts precisely on time even if some development team members are missing
      - should happen at the same time and place every day
      - is limited (time-boxed) to fifteen minutes
    - Anyone is welcome, though only development team members should contribute.

Scrum

Scrum

- Workflows in the Scrum framework

  (3) Each day during a sprint, the team holds a *Daily Scrum* (or stand-up) with specific guidelines:

  - During the daily scrum, each team member typically answers three questions:
    - What did I complete yesterday that contributed to the team meeting our sprint goal?
    - What do I plan to complete today to contribute to the team meeting our sprint goal?
    - Do I see any impediment that could prevent me or the team from meeting our sprint goal?
  - Any impediment identified in the daily scrum should be captured by the scrum master and displayed on the team's scrum board or on a shared risk board, with an agreed person designated to working toward a resolution (outside of the daily scrum). No detailed discussions should happen during the daily scrum.

Scrum

Scrum

- Workflows in the Scrum framework
  - (4) At the end of a sprint, the team holds two events: the *Sprint Review* and the *Sprint Retrospective*.
    - At the sprint review, the team:
      - reviews the work that was completed and the planned work that was not completed.
      - presents the completed work (aka the demo) to the stakeholders.
      - collaborates with the stakeholders on what to work on next.
    - Guidelines for sprint reviews:
      - Incomplete work cannot be demonstrated.
      - The recommended duration is two hours for a two-week sprint (proportional for other sprint-durations).

Scrum

- Workflows in the Scrum framework
  - (4) At the end of a sprint, the team holds two events: the *Sprint Review* and the *Sprint Retrospective*.
    - At the sprint retrospective, the team:
      - Reflects on the past sprint.
      - Identifies and agrees on continuous process improvement actions.
    - Three main questions are asked in the sprint retrospective:
      - What went well during the sprint?
      - What did not go well?
      - What could be improved for better productivity in the next sprint?
    - The recommended duration is one-and-a-half hours for a two-week sprint (proportional for other sprint duration(s)).
    - This event is facilitated by the scrum master.

Scrum

- Artifacts in the Scrum framework
  - Product Backlog
    - The product backlog is a model of work to be done and contains an ordered list of product requirements that a scrum team maintains for a product.
    - The format of product backlog items varies, common formats include user stories, use cases, or any other requirements format the team finds useful. These will define features, bug fixes, non-functional requirements, etc.
    - The product owner prioritizes product backlog items (PBIs) based on considerations such as risk, business value, dependencies, size, and date needed.
    - The product backlog is what will be delivered, ordered into the sequence in which it should be delivered. It is visible to everyone but may only be changed with the consent of the product owner, who is ultimately responsible for ordering product backlog items for the development team to choose.

Scrum

- Artifacts in the Scrum framework
  - Sprint Backlog
    - The sprint backlog is the list of work the development team must address during the next sprint.
    - The list is derived by the scrum team progressively selecting product backlog items in priority order from the top of the product backlog until they feel they have enough work to fill the sprint. The development team should keep in mind its past performance assessing its capacity for the new-sprint, and use this as a guideline of how much 'effort' they can complete.
    - The product backlog items may be broken down into tasks by the development team. Tasks on the sprint backlog are never assigned (or pushed) to team members by someone else; rather team members sign up for (or pull) tasks as needed according to the backlog priority and their own skills and capacity. This promotes self-organization of the development team and developer buy-in.

- Scrum
  - Artifacts in the Scrum framework
    - Product Increment
      - The potentially releasable increment is the sum of all the product backlog items completed during a sprint, integrated with the work of all previous sprints.
      - At the end of a sprint, the increment must be complete, according to the scrum team's definition of "done", fully functioning, and in a usable condition regardless of whether the product owner decides to actually release it.

- Scrum
  - Artifacts in the Scrum framework
    - Some Extensions
      - Sprint burn-down chart (燃尽图)
      - Release burn-up chart (燃起图)
      - Definition of done (DoD 完成标准)
      - Velocity
      - Spike
      - Research
      - Tracer bullet.

Scrum

- The following terminology is used in Scrum
  - Scrum Team
    - Product Owner, Scrum Master and Development Team.
  - Product Owner
    - The person responsible for maintaining the Product Backlog by representing the interests of the stakeholders, and ensuring the value of the work the Development Team does.
  - Scrum Master
    - The person responsible for the Scrum process, making sure it is used correctly and maximizing its benefits.
  - Development Team
    - A cross-functional group of people responsible for delivering potentially shippable increments of Product at the end of every Sprint.
  - Sprint Burn Down Chart
    - Daily progress for a Sprint over the sprint's length.

Scrum

- The following terminology is used in Scrum
  - Release Burn Down Chart
    - Sprint level progress of completed stories in the Product Backlog.
  - Product Backlog (产品积压工作列表)
    - A prioritized list of high-level requirements.
  - Sprint Backlog (Sprint 积压工作列表)
    - A prioritized list of tasks to be completed during the sprint.
  - Sprint
    - A time period (typically 1–4 weeks) in which development occurs on a set of backlog items that the team has committed to. Also commonly referred to as a Time-box or iteration.

Scrum

- The following terminology is used in Scrum
  - (User) Story
    - A feature that is added to the backlog is commonly referred to as a story and has a specific suggested structure:
      - "As a <user type> I want to <do some action> so that <desired result>"
    - This is done so that the development team can identify the user, action and required result in a request and is a simple way of writing easily understanding requests.
      - Example: As a wiki user I want a tools menu on the edit screen so that I can easily apply font formatting.
    - A story is an *independent, negotiable, valuable, estimable, small, testable* requirement ("INVEST").
    - Stories may be clustered into epics when represented on a product roadmap or further down in the backlog.

Scrum

- The following terminology is used in Scrum
  - Theme (主题)
    - A theme is a top-level objective that may span projects and products.
    - Themes may be broken down into sub-themes, which are more likely to be product-specific.
    - Themes can be used at both program and project level to drive strategic alignment and communicate a clear direction.
  - Epic (史诗)
    - An epic is a group of related stories, mainly used in product roadmaps and the backlog for features that have not yet been analyzed enough to break down into component stories, which should be done before bringing it into a sprint so to reduce uncertainty.
    - Epics can also be used at both program and project level.

Scrum

- The following terminology is used in Scrum
  - Spike
    - A spike is a time boxed period used to research a concept and/or create a simple prototype.
    - Spikes can either be planned to take place in between sprints or, for larger teams, a spike might be accepted as one of many sprint delivery objectives.
    - Spikes are often introduced before the delivery of large epics or user stories in order to secure budget, expand knowledge, and/or produce a proof of concept.
    - The duration and objective(s) of a spike will be agreed between the Product Owner and Delivery Team before the start. Unlike sprint commitments, spikes may or may not deliver tangible (切实的), shippable, valuable functionality.
    - E.g., the objective of a spike might be to successfully reach a decision on a course of action. The spike is over when the time is up, not necessarily when the objective has been delivered.

Scrum

- The following terminology is used in Scrum
  - Tracer Bullet (曳光弹)
    - The tracer bullet is a spike with the current architecture, current technology set, current set of best practices which results in production quality code.
      - It might just be a very narrow implementation of the functionality but is not throw away code.
      - It is of production quality and the rest of the iterations can build on this code.
    - The name has military origins as ammunition that makes the path of the weapon visible, allowing for corrections. Often these implementations are a 'quick shot' through all layers of an application, such as connecting a single form's input field to the back-end, to prove the layers will connect as expected.
  - Impediment
    - Anything that prevents a team member from performing work as efficiently as possible.

Scrum

- The following terminology is used in Scrum
  - Point Scale/Effort/Story Points (难度尺度)
    - Relates to an abstract point system, used to discuss the difficulty of the story, without assigning actual hours. The most common scale used is a rounded Fibonacci sequence (1,2,3,5,8,13,20,40,100), although some teams use linear scale (1,2,3,4...), powers of two (1,2,4,8...), and clothes size (XS, S, M, L, XL).
  - Task
    - Added to the story at the beginning of a sprint and broken down into hours. Each task should not exceed 12 hours, but it's common for teams to insist that a task take no more than a day to finish.
  - Definition of Done (DoD)
    - The exit-criteria to determine whether a product backlog item is complete. In many cases the DoD requires that all regression tests should be successful.

Scrum

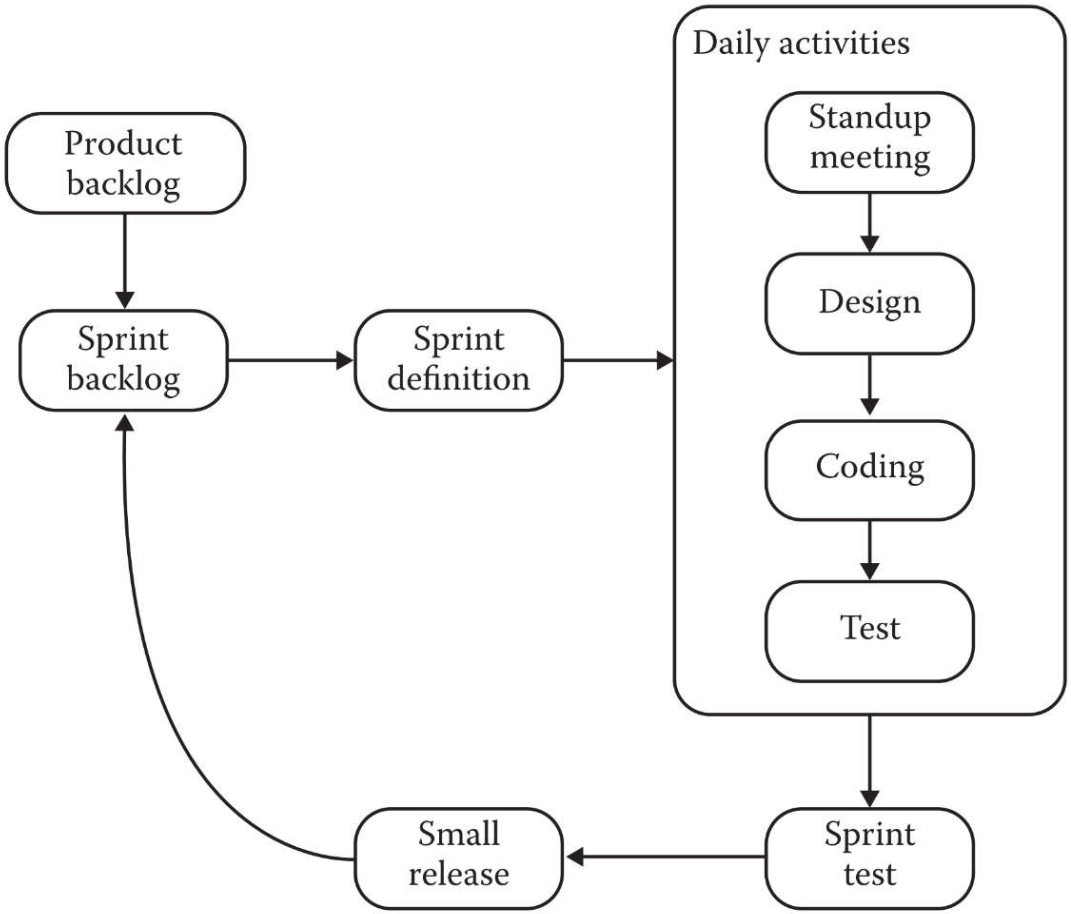
- The following terminology is used in Scrum
  - Velocity (团队速率)
    - The total effort a team is capable of in a sprint. The number is derived by evaluating the story points completed from the last few sprint's stories/features. This is a guideline for the team and assists them in understanding how many stories they can do in a future sprint.
  - Sashimi (成功的定义/生鱼片)
    - A report that something is "done". The definition of "done" may vary from one Scrum team to another, but must be consistent within one team.
  - Planning Poker (计划扑克游戏)
    - In the Sprint Planning Meeting, the team sits down to estimate its effort for the stories in the backlog. The Product Owner needs these estimates, so that he or she is empowered to effectively prioritize items in the backlog and, as a result, forecast releases based on the team's velocity.

Scrum

- The following terminology is used in Scrum
  - Abnormal Termination
    - The Product Owner can cancel a Sprint if necessary. The Product Owner may do so with input from the team, Scrum Master or management.
      - For instance, management may wish to cancel a sprint if external circumstances negate the value of the sprint goal.
    - If a sprint is abnormally terminated, the next step is to conduct a new Sprint planning meeting, where the reason for the termination is reviewed.
  - ScrumBut
    - A ScrumBut (or Scrum But) is an exception to the "pure" Scrum methodology, where a team has changed the methodology to adapt it to their own needs.

Scrum

- The Scrum life cycle.



- eXtreme Programming
  - 极限编程 (eXtreme Programming, XP) 是敏捷模型的一种实现过程，由 *Kent Beck* 在1996年提出。
  - 极限编程适合：
    - 小团队 (2-10 programmers)
    - 高风险
    - 快速变化或不稳定的需求
    - 强调可测试性
  - 格言
    - 沟通 Communication
    - 简化 Simplicity
    - 反馈 Feedback
    - 激励 Courage
    - *谦逊 Modesty

Kent Beck, 1996

最简单的可能就是最有效的

- eXtreme Programming
  - 极限编程方法的13个核心实践
    - 团队协作 (Whole Team)
    - 规划策略 (The Planning Game)
    - 结对编程 (Pair programming)
    - 测试驱动开发 (Testing-Driven Development)
    - 重构 (Refactoring)
    - 简单设计 (Simple Design)
    - 代码集体所有 (Collective Code Ownership)
    - 持续集成 (Continuous Integration)
    - 客户测试 (Customer Tests)
    - 小规模发布 (Small Release)
    - 每周40小时工作制 (40-hour Week)
    - 编码规范 (Code Standards)
    - 系统隐喻 (System Metaphor)

- eXtreme Programming
  - 极限编程的12个实践
    - 小版本
      - 小版本发布有利于高度迭代以及给客户展现开发的进展，客户可以针对性提出反馈。小版本也需要总体合理的规划，如果把模块缩得太小，会影响软件的整体思路。
    - 规划策略
      - 客户以故事的形式编写客户需求。极限编程不讲求统一的客户需求收集，客户需求不是由开发人员整理，而让客户编写，开发人员进行分析，设定优先级别，进行技术实现。规划策略可以进行多次，每次迭代完毕后再行修改。客户故事是开发人员与客户沟通的焦点，也是版本设计的依据，所以其管理必须是有效的、沟通顺畅的。

- eXtreme Programming
  - 极限编程的12个实践
    - 现场客户
      - 极限编程要求客户参与开发工作，客户需求就是客户负责编写的，所以要求客户在开发现场一起工作，并为每次迭代提供反馈。
    - 隐喻
      - 隐喻是让项目参与人员都必须对一些抽象的概念 (行业术语) 理解一致，因为业务本身的术语开发人员不熟悉，而软件开发的术语客户不理解，因此开始要先明确双方使用的隐喻，使用统一的术语描述问题，避免歧义。

- eXtreme Programming
  - 极限编程的12个实践
    - 简单设计
      - 极限编程体现跟踪客户的需求变化，既然需求是变化的，所以对于目前的需求不必过多考虑扩展性的开发，而讲求简单设计，实现目前需求即可。简单设计的本身也为短期迭代提供了方便，若开发者考虑“通用”因素较多，增加了软件的复杂度，将会加长开发的迭代周期。

- eXtreme Programming
  - 极限编程的12个实践
    - 重构
      - 重构是极限编程先测试后编码的必然需求，为了整体软件可以先进行测试，对于一些软件要开发的模块先简单模拟，让编译通过，到达测试的目的。然后再对模块具体“优化”，所以重构包括模块代码的优化与具体代码的开发。重构是使用了“物理学”的一个概念，是在不影响物体外部特性的前提下，重新优化其内部的机构。这里的外部特性就是保证测试的通过。

- eXtreme Programming
  - 极限编程的12个实践
    - 测试驱动开发
      - 极限编程是以测试开始的，为了可以展示客户需求的实现，测试程序优先设计，测试是从客户实用的角度出发，客户实际使用的软件界面着想，测试是客户需求的直接表现，是客户对软件过程的理解。测试驱动开发，也就是客户的需求驱动软件的开发。
    - 持续集成
      - 集成的理解就是提交软件的展现，由于采用测试驱动开发、小版本的方式，所以不断集成 (整体测试) 是与客户沟通的依据，也是让客户提出反馈意见的参照。持续集成也是完成阶段开发任务的标志。

- eXtreme Programming
  - 极限编程的12个实践
    - 结对编程
      - 这是极限编程最有争议的实践。就是两个程序员合用一台计算机编程，一个编码，一个检查，增加专人审计是为了提供软件编码的质量。两个人的角色经常变换，保持开发者的工作热情。这种编程方式对培养新人或开发难度较大的软件都有非常好的效果。
    - 代码共有
      - 在极限编程里没有严格文档管理，代码为开发团队共有，这样有利于开发人员的流动管理，因为所有的人都熟悉所有的编码。

- eXtreme Programming
  - 极限编程的12个实践
    - 编码规范
      - 编码是开发团队里每个人的工作，又没有详细的文档，代码的可读性很重要，所以规定统一的标准和习惯是必要的。
    - 每周40小时工作
      - 极限编程认为编程是愉快的工作，不要轻易加班，小版本的设计也是为了单位时间可以完成的工作安排。

eXtreme Programming

开发周期

Planning

- User stories are written.
- Release planning creates the schedule.
- Make frequent small releases.
- The Project Velocity is measured.
- The project is divided into iterations.
- Iteration planning starts each iteration.
- Move people around.
- A stand-up meeting starts each day.
- Fix XP when it breaks.

Coding

- The customer is always available.
- Code must be written to agreed standards.
- Code the unit test first.
- All code is pair programmed.
- Only one pair integrates code at a time.
- Integrate often.
- Use collective code ownership.
- Leave optimization till last.
- No overtime.

Designing

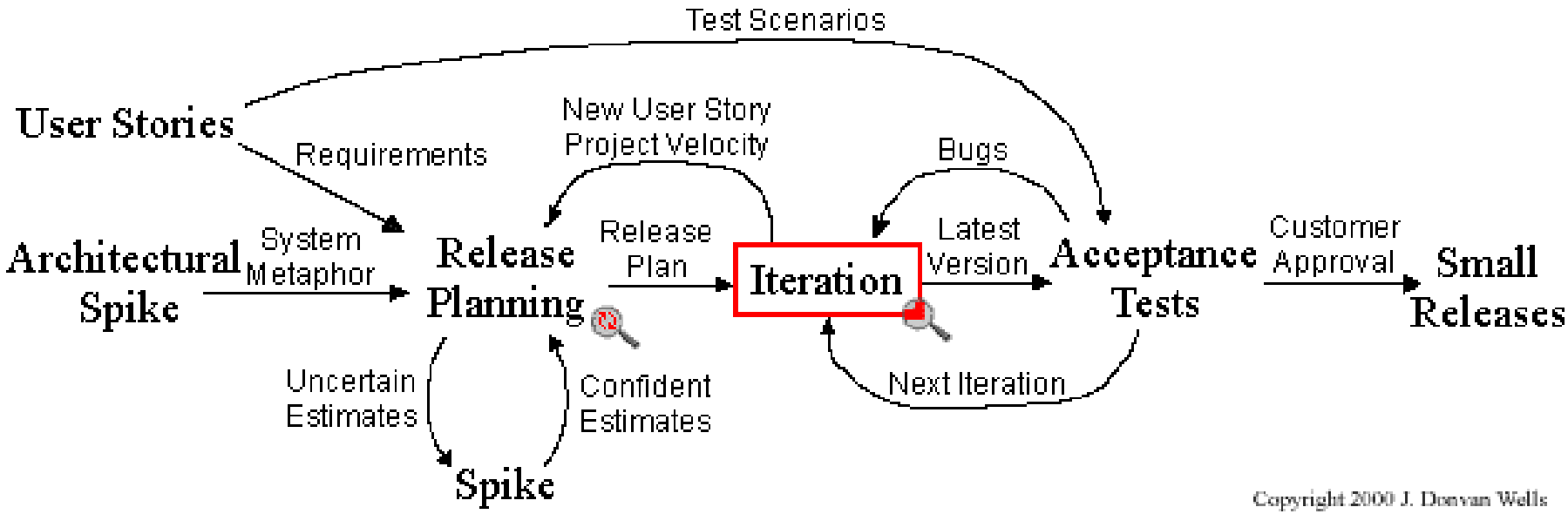
- Simplicity.
- Choose a system metaphor.
- Use CRC cards for design sessions.
- Create spike solutions to reduce risk.
- No functionality is added early.
- Refactor whenever and wherever possible.

Testing

- All code must have unit tests.
- All code must pass all unit tests before it can be released.
- When a bug is found tests are created.
- Acceptance tests are run often and the score is published.

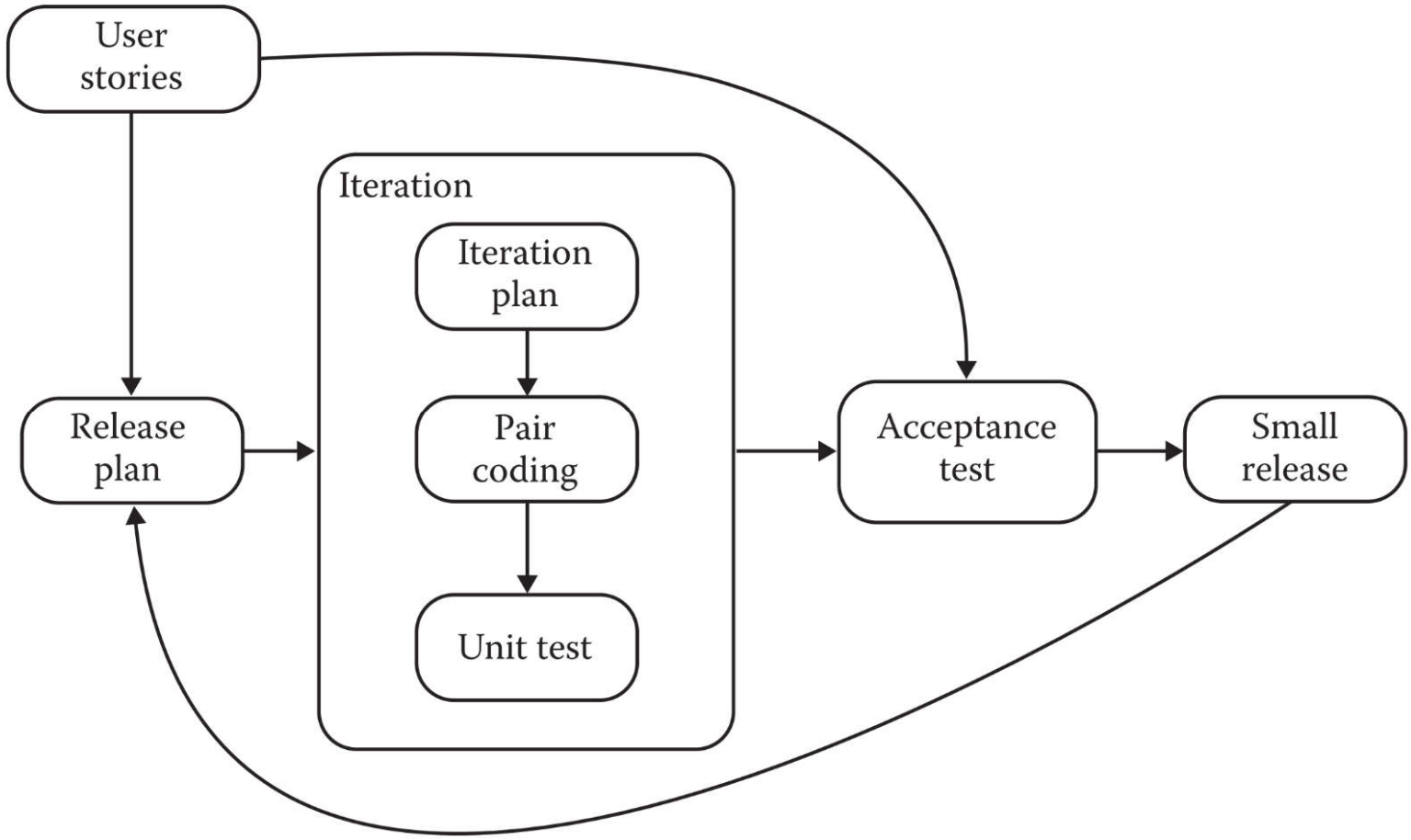
eXtreme Programming

The eXtreme Programming life cycle



eXtreme Programming

The eXtreme Programming life cycle



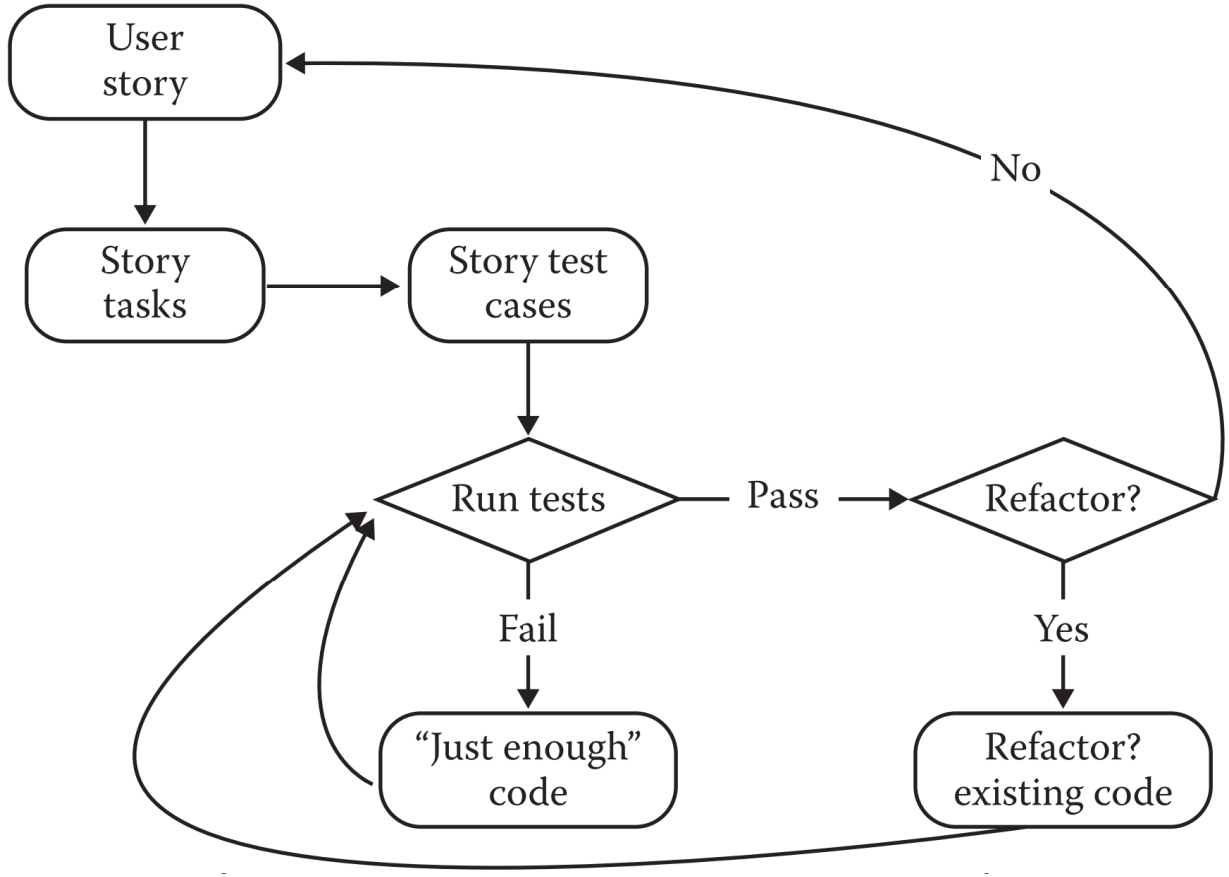
Test-driven Development (TDD)

- TDD is the extreme case of agility. It is driven by a sequence of user stories. A user story can be decomposed into several tasks, and this is where the big difference occurs. Before any code is written for a task, the developer decides how it will be tested. The tests become the specification. Now the tests are run on nonexistent code and naturally, they fail. But this leads to the best feature of TDD—greatly simplified fault isolation. Once the tests have been run (and failed), the developer writes just enough code to make the tests pass, and the tests are rerun. Once all the tests pass, the next user story is implemented.
- Occasionally, the developer may decide to refactor the existing code. The cleaned–up code is then subjected to the full set of existing test cases, which is very close to the idea of regression testing.
- For TDD to be practical, it must be done in an environment that supports automated testing, typically with a member of the nUnit family of automated test environments.

- Test-driven Development (TDD)
  - Testing in TDD is interesting. Since the story-level test cases drive the coding, they ARE the specification, so in a sense, TDD uses specification-based testing. But since the code is deliberately as close as possible to the test cases, we could argue that it is also code-based testing.
  - There are two problems with TDD.
    - The first is common to all agile flavors—the bottom–up approach prohibits a single, high-level design step. User stories that arrive late in the sequence may obviate earlier design choices. Then refactoring would have to also occur at the design level, rather than just at the code level. The agile community is very passionate about the claim that repeated refactoring results in an elegant design. Given one of the premises of agile development, namely that the customer is not sure of what is needed, or equivalently, rapidly changing requirements, refactoring at both the code and design levels seems the only way to end up with an elegant design. This is an inevitable constraint on bottom–up development.

- Test-driven Development (TDD)
  - There are two problems with TDD.
    - The second problem is that all developers make mistakes—that is much of the reason we test in the first place. But consider: what makes us think that the TDD developer is perfect at devising the test cases that drive the development? Even worse: what if late user stories are inconsistent with earlier ones? A final limitation of TDD is there is no place in the life cycle for a cross-check at the user story level.

- Test-driven Development (TDD)
  - TDD life cycle.



- 规模化敏捷开发
  - Agile at Scale (SEI Blog/SPRUCE—Systems and Software Producibility Collaboration Environment, SEI/CMU)
    - Why is Agile at Scale Challenging
      - Agile practices, derived from a set of foundational principles, have been applied successfully for well over a decade and have enjoyed broad adoption in the commercial sector, with the net result that development teams have gotten better at building software. Reasons for these improvements include
        - increased visibility into a project and the emerging product,
        - increased responsibility of development teams, the ability for customers and end users to interact early with executable code, and
        - the direct engagement of the customer or product owner in the project to provide a greater sense of shared responsibility.

- 规模化敏捷开发
  - Agile at Scale (SEI Blog/SPRUCE—Systems and Software Producibility Collaboration Environment, SEI/CMU)
    - Why is Agile at Scale Challenging (cont.)
      - Business and mission goals, however, are larger than a single development team. Applying Agile at Scale, in particular in DoD-scale environments, therefore requires answering several questions in three dimensions:
        (1) Team size
        (2) Complexity
        (3) Duration

规模化敏捷开发

- Agile at Scale Challenging

(1) **Team size**. What happens when Agile practices are used in a 100-person (or larger) development team? What happens when the development team needs to interact with the rest of the business, such as quality assurance, system integration, project management, and marketing, to get input into product development and collaborate on the end-to-end delivery of the product? Scrum and Agile methods, such as extreme programming (XP), are typically used by small teams of at most 7-to-10 people. Larger teams require orchestration of both multiple (sub)teams and cross-functional roles beyond development. Organizations have recently been investigating approaches, such as Scaled Agile Framework, to better manage the additional coordination issues associated with increased team size.

规模化敏捷开发

- Agile at Scale Challenging (cont.)

(2) **Complexity**. Large-scale systems are often large in scope relative to the number of features, the amount of new technology being introduced, the number of independent systems being integrated, the number and types of users to accommodate, and the number of external systems with which the system communicates. Does the system have stringent (严苛的) quality attributes (质量特性) needs, such as stringent real-time, high-reliability, and security requirements? Are there multiple external stakeholders and interfaces? Typically, such systems must go through rigorous verification and validation (V&V), which complicate the frequent deployment practices used in Agile development.

规模化敏捷开发

- Agile at Scale Challenging (cont.)

(3) **Duration**. How long will the system be in development? How long in operations and sustainment? Larger systems need to be in development and operation for a longer period of time than products to which agile development is typically applied, requiring attention to future changes, possible redesigns, as well as maintaining several delivered versions. Answers to these questions affect the choice of quality attributes supporting system maintenance and evolution goals that are key to system success over the long term.

规模化敏捷开发

- 10 Recommended Practices for Achieving Agile at Scale
  - (1) Make team coordination top priority
  - (2) Use an architectural runway to manage technical complexity
  - (3) Align feature-based development and system decomposition
  - (4) Use quality-attribute scenarios to clarify architecturally significant requirements
  - (5) Use test-driven development for early and continuous focus on verification
  - (6) Use end-to-end testing for early insight into emerging system properties.
  - (7) Use continuous integration for consistent attention to integration issues.
  - (8) Consider recent field study management as an approach to manage system development strategically.
  - (9) Use prototyping to rapidly evaluate and resolve significant technical risks.
  - (10) Use architectural evaluations to ensure that architecturally significant requirements are being addressed.

规模化敏捷开发

- 10 Recommended Practices for Achieving Agile at Scale
  - (1) Make team coordination top priority.
    - Scrum is the most common Agile project management method used today, and primarily involves team management practices. In its simplest instantiation, a Scrum development environment consists of a single Scrum team with the skills, authority, and knowledge required to specify requirements, architect, design, code, and test the system. As systems grow in size and complexity, the single team mode may no longer meet development demands. If a project has already decided to use a Scrum-like project-management technique, the Scrum approach can be extended to managing multiple teams with a "Scrum of Scrums," a special coordination team whose role is to (1) define what information will flow between and among development teams (addressing inter-team dependencies and communication) and (2) identify, analyze, and resolve coordination issues and risks that have potentially broader consequences (e.g., for the project as a whole).

规模化敏捷开发

- 10 Recommended Practices for Achieving Agile at Scale
  - (1) Make team coordination top priority.
    - A Scrum of Scrums typically consists of members from each team chosen to address end-to-end functionality or cross-cutting concerns such as user interface design, architecture, integration testing, and deployment. Creating a special team responsible for inter-team coordination helps ensure that the right information, including measurements, issues, and risks, is communicated between and among teams. Care needs to be taken, however, when the Scrum of Scrums team itself gets large to not overwhelm the team. This scaling can be accomplished by organizing teams--and the Scrum of Scrums team itself--along feature and service affinities. We further discuss this approach to organizing teams in our feature-based development and system decomposition practice. Such orchestration is essential to managing larger teams to success, including Agile teams.

规模化敏捷开发

10 Recommended Practices for Achieving Agile at Scale

(2) Use an architectural runway to manage technical complexity.

- Stringent safety or mission-critical requirements increase technical complexity and risk. Technical complexity arises when the work takes longer than a single iteration or release cycle and cannot be easily partitioned and allocated to different technical competencies (or teams) to independently and concurrently develop their part of a solution. Successful approaches to managing technical complexity include having the most-urgent system or software architecture features well defined early (or even pre-defined at the organizational level, e.g., as infrastructure platforms or software product lines).
- The Agile term for such pre-staging of architectural features that can be leveraged by development teams is "architectural runway." The architectural runway has the goal of providing the degree of stability required to support future iterations of development. This stability is particularly important to the successful operation of multiple teams.
- A system or software architect decides which architectural features must be developed first by identifying the quality attribute requirements that are architecturally significant for the system.

规模化敏捷开发

10 Recommended Practices for Achieving Agile at Scale

(2) Use an architectural runway to manage technical complexity.

- By initially defining (and continuously extending) the architectural runway, development teams are able to iteratively develop customer-desired features that use that runway and benefit from the quality attributes they confer (e.g., security and dependability).
- Having a defined architectural runway helps uncover technical risks earlier in the lifecycle, thereby helping to manage system complexity (and avoiding surprises during the integration phase). Uncovering quality attribute concerns, such as security, performance, or availability with the underlying architectural late in the lifecycle--that is, after several iterations have passed--often yields significant rework and schedule delay. Delivering functionality is more predictable when the infrastructure for the new features is in place, so it is important to maintain a continual focus on the architecturally significant requirements and estimation of when the development teams will depend on having code that implements an architectural solution.

规模化敏捷开发

- 10 Recommended Practices for Achieving Agile at Scale
  - (3) Align feature-based development and system decomposition.
    - A common approach in Agile teams is to implement a feature (or user story) in all the components of the system. This approach gives the team the ability to focus on something that has stakeholder value. The team controls every piece of implementation for that feature and therefore they need not wait until someone else outside the team has finished some required work. We call this approach "vertical alignment" because every component of the system required for realizing the feature is implemented only to the degree required by the team.
    - System decomposition could also be horizontal, however, based on the architectural needs of the system. This approach focuses on common services and variability mechanisms that promote reuse.

规模化敏捷开发

- 10 Recommended Practices for Achieving Agile at Scale
  - (3) Align feature-based development and system decomposition.
    - The goal of creating a feature-based development and system decomposition approach is to provide flexibility in aligning teams horizontally, vertically, or in combination, while minimizing coupling to ensure progress. Although organizations create products in very different domains (ranging from embedded systems to enterprise systems) similar architecture patterns and strategies emerge when a need to balance rapid progress and agile stability is desired. The teams create a platform containing commonly used services and development environments either as frameworks or platform plug-ins to enable fast feature-based development.

规模化敏捷开发

- 10 Recommended Practices for Achieving Agile at Scale
  - (4) Use quality-attribute scenarios to clarify architecturally significant requirements.
    - Scrum emphasizes customer-facing requirements--features that end users dwell on--and indeed these are important to success. But when the focus on end-user functionality becomes exclusive, the underlying architecturally significant requirements can go unnoticed.
    - Superior practice is to elicit, document, communicate, and validate underlying quality attribute scenarios during development of the architectural runway. This approach becomes even more important at scale when projects often have significant longevity and sustainability needs. Early in the project, evaluate the quality attribute scenarios to determine which architecturally significant requirements should be addressed in early development increments (see architectural runway practice above) or whether strategic shortcuts can be taken to deliver end-user capability more quickly.

规模化敏捷开发

- 10 Recommended Practices for Achieving Agile at Scale
  - (4) Use quality-attribute scenarios to clarify architecturally significant requirements.
    - For example, will the system really have to scale up to a million users immediately, or is this actually a trial product? There are different considerations depending on the domain.For example, IT systems use existing frameworks, so understanding the quality attribute scenarios can help developers understand which architecturally significant requirements might already be addressed adequately within existing frameworks (including open-source systems) or existing legacy systems that can be leveraged during software development. Similarly, such systems must address changing requirements in security and deployment environments, which necessitates architecturally significant requirements be given top priority when dealing with scale.

规模化敏捷开发

- 10 Recommended Practices for Achieving Agile at Scale
  - (5) Use test-driven development for early and continuous focus on verification.
    - This practice can be summarized as "write your test before you write the system." When there is an exclusive focus on "sunny-day" scenarios (a typical developer's mindset), the project becomes overly reliant on extensive testing at the end of the project to identify overlooked scenarios and interactions. Therefore, be sure to focus on rainy-day scenarios (e.g., consider different system failure modes), as well as sunny-day scenarios. The practice of writing tests first, especially at the business or system level (which is known as acceptance test-driven development) reinforces the other practices that identify the more challenging aspects and properties of the system, especially quality attributes and architectural concerns (see architectural runway and quality-attribute scenarios practices above).

规模化敏捷开发

- 10 Recommended Practices for Achieving Agile at Scale
  - (6) Use end-to-end testing for early insight into emerging system properties.
    - To successfully derive the full benefit from test-driven development at scale, consider early and continuous end-to-end testing of system scenarios. When teams test only the features for which they are responsible, they lose insight into overall system behavior (and how their efforts contribute to achieving it). Each small team could be successful against its own backlog, but someone needs to look after broader or emergent system properties and implications. For example, who is responsible for the fault tolerance of the system as a whole? Answering such questions requires careful orchestration of development with verification activities early and throughout development. When testing end-to-end, take into account different operational contexts, environments, and system modes.

- 规模化敏捷开发
  - 10 Recommended Practices for Achieving Agile at Scale
    (6) Use end-to-end testing for early insight into emerging system properties.
      - At scale, understanding end-to-end functionality requires its elicitation and documentation. These goals can be achieved through the application of agile requirements management techniques, such as stories, as well as use of architecturally significant requirements. If there is a need to orchestrate multiple systems, however, a more deliberate elicitation of end-to-end functionality as mission/business threads should provide a better result.

规模化敏捷开发

- 10 Recommended Practices for Achieving Agile at Scale
  - (7) Use continuous integration for consistent attention to integration issues.
    - This basic Agile practice becomes even more important at scale, given the increased number of subsystems that must work together and whose development must be orchestrated. One implication is that the underlying infrastructure developers will use day-to-day must be able to support continuous integration. Another is that developers focus on integration earlier, identifying the subsystems and existing frameworks that will need to integrate. This identification has implications for the architectural runway, quality-attribute scenarios, and orchestration of development and verification activities presented in our earlier blog posting. Useful measures for managing continuous integration include rework rate and scrap rate. It is also important to start early in the project to identify issues that can arise during integration. What this means more broadly is that both integration and the ability to integrate must be managed in the Agile environment.

规模化敏捷开发

- 10 Recommended Practices for Achieving Agile at Scale
  - (8) Consider recent field study management as an approach to manage system development strategically.
    - The concept of technical debt arose naturally from the use of Agile methods, where the emphasis on releasing features quickly often creates a need for rework later. At scale, there may be multiple opportunities for shortcuts, so understanding technical debt and its implications becomes a means for strategically managing the development of the system. For example, there might be cases where certain architectural selections made to accelerate delivery have long-term consequences. A recent field study the SEI conducted with software developers also strongly supports that the leading sources of technical debt are architectural choices. Such tradeoffs must be understood and managed based on both qualitative and quantitative measurements of the system.

- 规模化敏捷开发
  - 10 Recommended Practices for Achieving Agile at Scale
    - (8) Consider recent field study management as an approach to manage system development strategically.
      - Qualitatively, architecture evaluations can be used as part of the product demos or retrospectives that Agile advocates. Quantitative measures are harder but can arise from understanding productivity, system uncertainty, and measures of rework (e.g., when uncertainty is greater, it may make more sense to incur more rework later).

- 规模化敏捷开发
  - 10 Recommended Practices for Achieving Agile at Scale
    - (9) Use prototyping to rapidly evaluate and resolve significant technical risks.
      - To address significant technical issues, teams employing Agile methods will sometimes perform what in Scrum is referred to as a technical spike, in which a team branches out from the rest of the project to investigate a specific technical issue, develop one or more prototypes to evaluate possible solutions, and report what they learned to the project team so that they can proceed with greater likelihood of success. A technical spike may extend over multiple sprints, depending on the seriousness of the issue and how much time it takes to investigate the issue and report information that the project can use.

规模化敏捷开发

- 10 Recommended Practices for Achieving Agile at Scale
  - (9) Use prototyping to rapidly evaluate and resolve significant technical risks.
    - At scale, technical risks having severe consequences are typically more numerous. Prototyping (and other approaches to evaluating candidate solutions such as simulation and demonstration) can therefore be an essential early planning but also recurring. A goal of Agile methods is increased early visibility. From that perspective, prototyping is a valuable means of achieving visibility more quickly for technical risks and their mitigations. The practice of making team coordination top priority as mentioned earlier has a role here, too, to help orchestrate reporting what was learned from prototyping to the overall system.

规模化敏捷开发

- 10 Recommended Practices for Achieving Agile at Scale
  - (10) Use architectural evaluations to ensure that architecturally significant requirements are being addressed.
    - While not considered part of mainstream Agile practice, architecture evaluations have much in common with Agile methods in seeking to bring a project's stakeholders together to increase their visibility into and commitment to the project, as well as to identify overlooked risks. At scale, architectural issues become even more important, and architecture evaluations thus have a critical role on the project. Architecture evaluation can be formal, as in the SEI's Architecture Tradeoff Analysis Method, which can be performed, for example, early in the Agile project lifecycle before the project's development teams are launched, or recurrently. There is also an important role for lighter weight evaluations in project retrospectives to evaluate progress against architecturally significant requirements.

- ISO/IEC 12207 软件生命周期过程
  - ISO/IEC/IEEE 12207
    - ISO/IEC/IEEE 12207 *Systems and software engineering – Software life cycle processes* is an international standard for software lifecycle processes. First introduced in 1995, it aims to be a primary standard that defines all the processes required for developing and maintaining software systems, including the outcomes and/or activities of each process.
    - ISO/IEC/IEEE 12207:2017.
      - The IEEE Computer Society joined directly with the ISO in the editing process for 2017's version. A significant change is that it adopts a process model identical to the ISO/IEC/IEEE 15288:2015 process model with one name change that the 15288 "System Requirements Definition" process is renamed to the "System/Software Requirements Definition" process.

- ISO/IEC 12207 软件生命周期过程
  - ISO/IEC/IEEE 12207:2017
    - This harmonization of the two standards led to the removal of separate software development and software reuse processes, bringing the total number of 12207 processes from 43 down to the 30 processes defined in 15288. It also caused changes to the quality management and quality assurance process activities and outcomes. Additionally, the definition of "audit" and related audit activities were updated. Annex I of ISO/IEC/IEEE 12207:2017 provides a process mapping between the 2017 version and the previous version, including the primary process alignments between the two versions; this is intended to enable traceability and ease transition for users of the previous version.

ISO/IEC 12207 软件生命周期过程

- Software Life Cycle Processes
  - The ISO/IEC 12207 establishes a set of processes for managing the lifecycle of software. The standard “does not prescribe (规定) a specific software life cycle model, development methodology, method, modelling approach, or technique.". Instead, the standard (as well as ISO/IEC/IEEE 15288) distinguishes between a "stage" and "process" as follows:
    - stage: "period within the life cycle of an entity that relates to the state of its description or realization". A stage is typically a period of time and ends with a "primary decision gate".
    - process: "set of interrelated or interacting activities that transforms inputs into outputs". The same process often recurs within different stages.

- ISO/IEC 12207 软件生命周期过程
  - Software Life Cycle Processes
    - Stages (aka phases) are not the same as processes, and this standard only defines specific processes - it does not define any particular stages. Instead, the standard acknowledges that software life cycles vary, and may be divided into stages that represent major life cycle periods and give rise to primary decision gates. No particular set of stages is normative, but it does mention two examples:
      - The system life cycle stages from ISO/IEC TS 24748-1 could be used (concept, development, production, utilization, support, and retirement).
      - It also notes that a common set of stages for software is concept exploration, development, sustainment (支持), and retirement.

ISO/IEC 12207 软件生命周期过程

- Software Life Cycle Processes
  - ISO/IEC/IEEE 12207:2017 divides software life cycle processes into four main process groups
    - Agreement processes
    - Organizational project-enabling processes
    - Technical management processes
    - Technical processes.
  - Under each of those four process groups are a variety of sub-categories, including the primary activities of acquisition and supply (agreement); configuration (technical management); and operation, maintenance, and disposal 处置 (technical).
  - The life cycle processes the standard defines are not aligned to any specific stage in a software life cycle. Indeed, the life cycle processes that involve planning, performance, and evaluation "should be considered for use at every stage". In practice, processes occur whenever they are needed within any stage.

- ISO/IEC 12207 软件生命周期过程
  - Software Life Cycle Processes
    - Agreement processes
      - Here ISO/IEC/IEEE 12207:2017 includes the acquisition and supply processes, which are activities related to establishing an agreement between a supplier and acquirer. Acquisition covers all the activities involved in initiating a project. The acquisition phase can be divided into different activities and deliverables that are completed chronologically (按顺序). During the supply phase a project management plan is developed. This plan contains information about the project such as different milestones that need to be reached.

- ISO/IEC 12207 软件生命周期过程
  - Software Life Cycle Processes
    - Organizational project-enabling processes
      - Detailed here are life cycle model management, infrastructure management, portfolio management, human resource management, quality management, and knowledge management processes. These processes help a business or organization enable, control, and support the system life cycle and related projects. Life cycle mode management helps ensure acquisition and supply efforts are supported, while infrastructure and portfolio management supports business and project-specific initiatives during the entire system life cycle. The rest ensure the necessary resources and quality controls are in place to support the business‘ project and system endeavors (努力).

- ISO/IEC 12207 软件生命周期过程
  - Software Life Cycle Processes
    - Technical management processes
      - ISO/IEC/IEEE 12207:2017 places eight different processes here:
        - Project planning
        - Project assessment and control
        - Decision management
        - Risk management
        - Configuration management
        - Information management
        - Measurement
        - Quality assurance
      - These processes deal with planning, assessment, and control of software and other projects during the life cycle, ensuring quality along the way.

ISO/IEC 12207 软件生命周期过程

- Software Life Cycle Processes
  - Technical processes
    - The technical processes of ISO/IEC/IEEE 12207:2017 encompass (包含) 14 different processes, some of which came from the old software-specific processes that were phased out from the 2008 version.
    - The full list includes:

Business or mission analysis	Implementation
Stakeholder needs and requirements definition	Integration
Systems/Software requirements definition	Verification
Architecture definition	Transition
Design definition	Validation
System analysis	Operation
	Maintenance
	Disposal

- ISO/IEC 12207 软件生命周期过程
  - Software Life Cycle Processes
    - Technical processes
      - These processes involve technical activities and personnel (information technology, troubleshooters, software specialists, etc.) during pre-, post- and during operation. The analysis and definition processes early on set the stage for how software and projects are implemented. Additional processes of integration, verification, transition, and validation help ensure quality and readiness. The operation and maintenance phases occur simultaneously, with the operation phase consisting of activities like assisting users in working with the implemented software product, and the maintenance phase consisting of maintenance tasks to keep the product up and running. The disposal process describes how the system/project will be retired and cleaned up, if necessary.

Thank you!