SOFTWARE TESTING

Software Testing

Agile Modeling & SLCP (2)

School of Computer Science & Engineering
Sun Yat-sen University

Instructor: Guoyang Cai
email: isscgy@mail.sysu.edu.cn

Approaches & Technologies

中山大學
SUN YAT-SEN UNIVERSITY

OUTLINE

eXtreme Programming

- 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

- 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

- 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 (story) 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

- 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

- 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

- 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

- 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

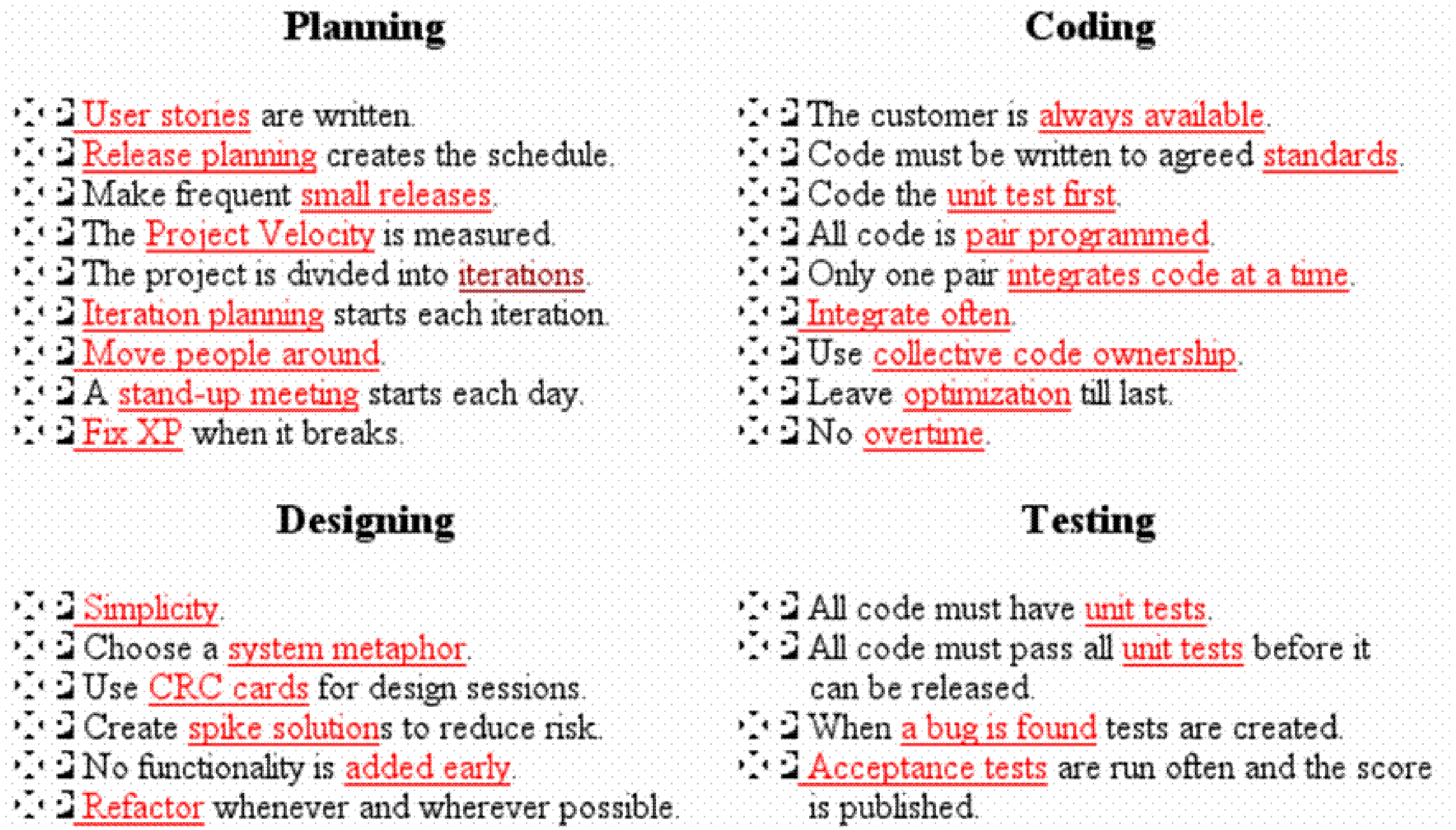
- eXtreme Programming
  - 极限编程的12个实践
    - 结对编程
      - 这是极限编程最有争议的实践。就是两个程序员合用一台计算机编程，一个编码，一个检查，增加专人审计是为了提供软件编码的质量。两个人的角色经常变换，保持开发者的工作热情。这种编程方式对培养新人或开发难度较大的软件都有非常好的效果。
    - 代码共有
      - 在极限编程里没有严格文档管理，代码为开发团队共有，这样有利于开发人员的流动管理，因为所有的人都熟悉所有的编码。

eXtreme Programming

- 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.

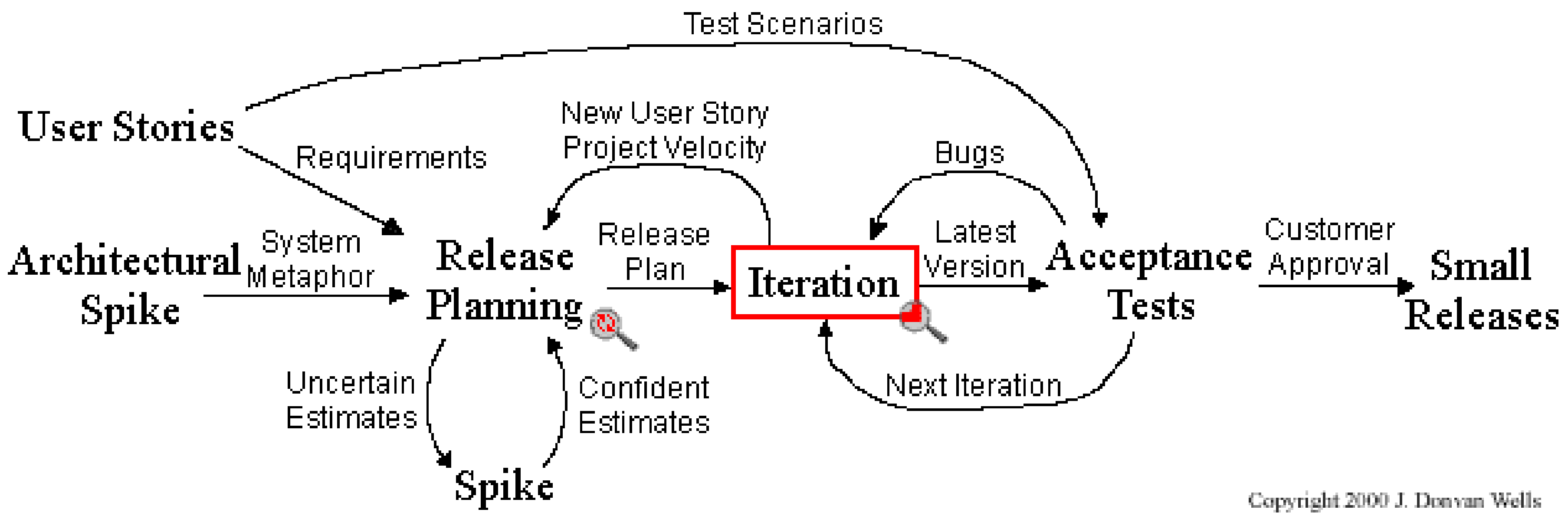
Designing

- 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

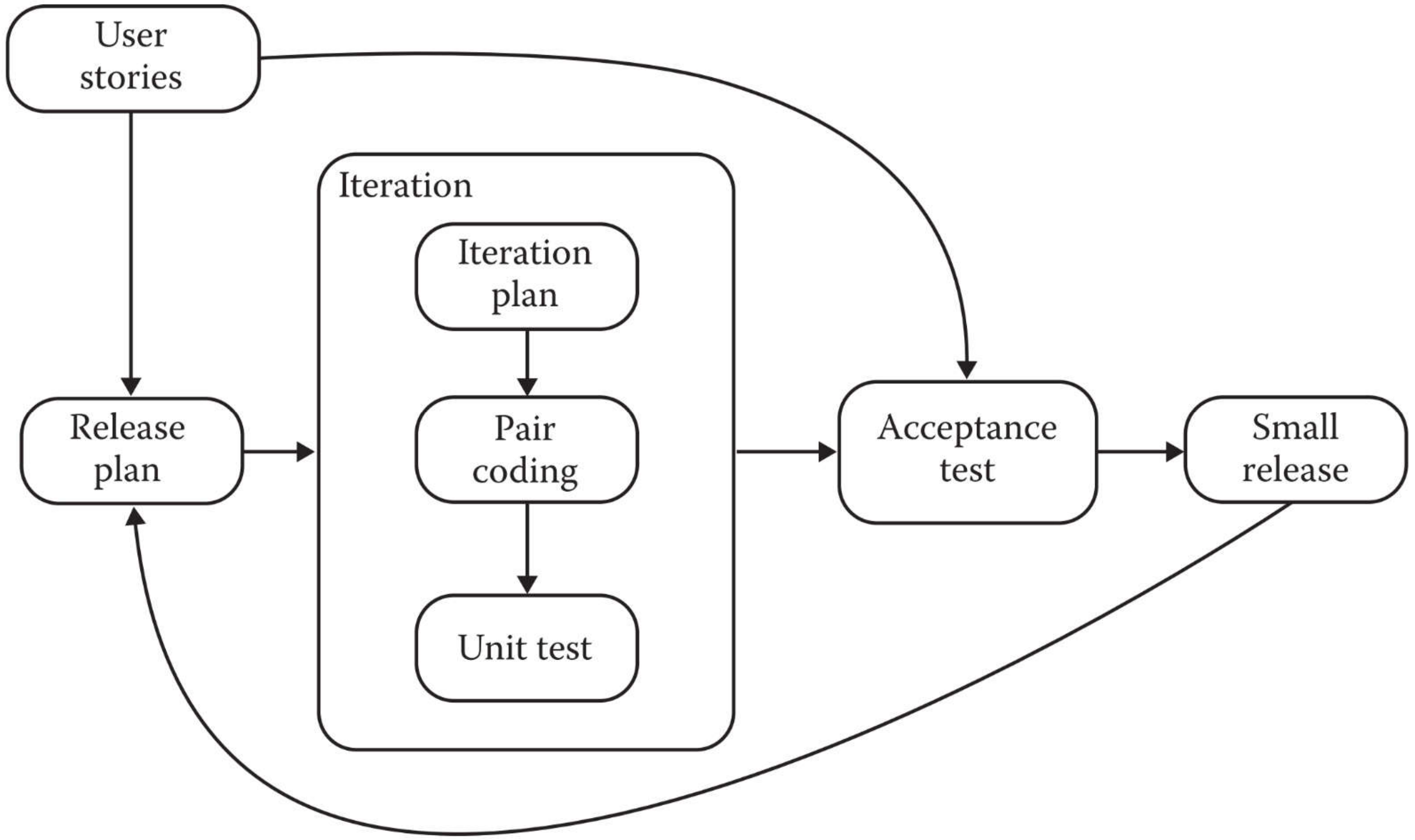
- All code must have unit tests.
- All code must pass all unit tests before it can be released.
- When a bug is found tests are created.
- Acceptance tests are run often and the score is published.

- eXtreme Programming
  - The eXtreme Programming life cycle



eXtreme Programming

- eXtreme Programming
  - The eXtreme Programming life cycle



TDD

- 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.

TDD

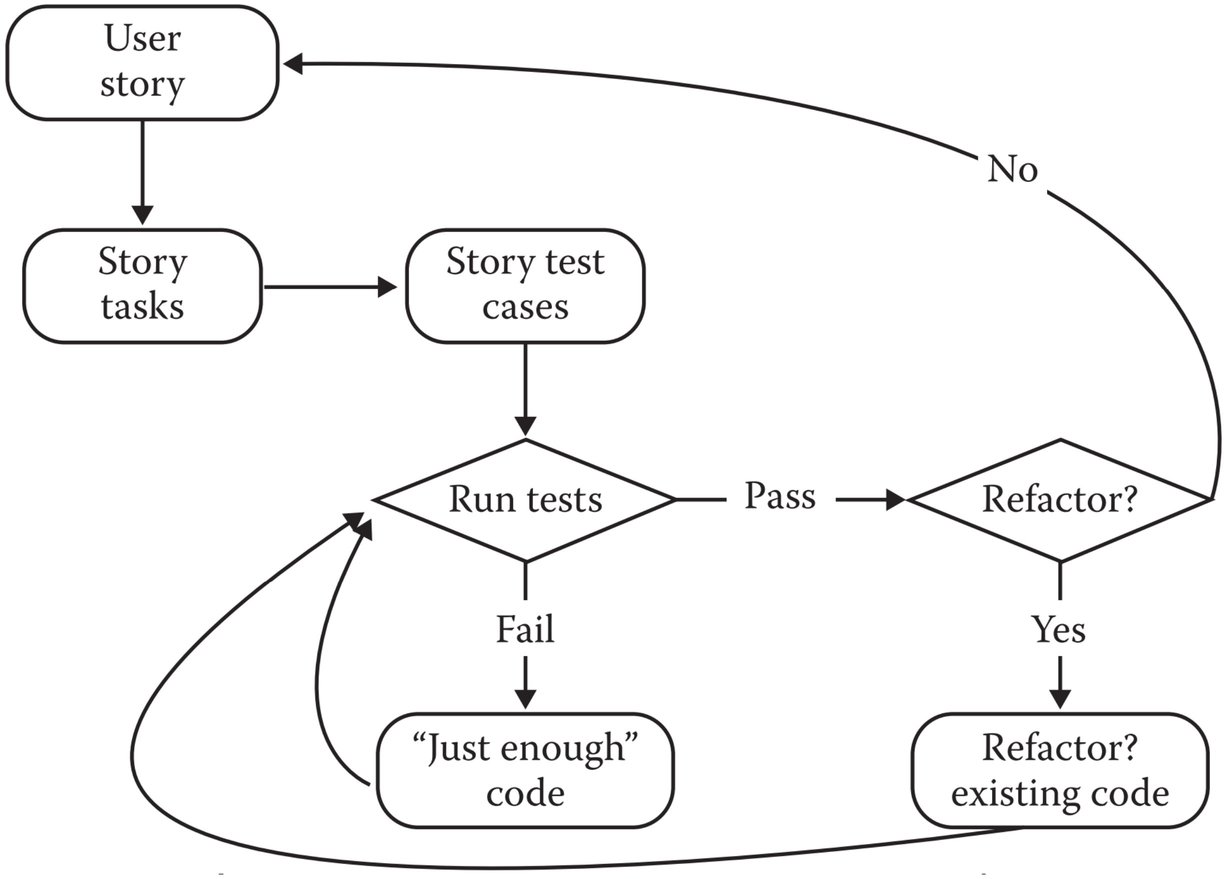
- Test-driven Development (TDD)
  - Testing in TDD is interesting. Since the story-level test cases drive the coding, they ARE the specification, so in a sense, TDD uses specification-based testing. But since the code is deliberately as close as possible to the test cases, we could argue that it is also 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.

TDD

- 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.

TDD

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



Agile at Scale

- 规模化敏捷开发
  - Agile at Scale (SEI Blog/SPRUCE — Systems and Software Producibility Collaboration Environment 系统和软件可生产性协作环境, 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.

Agile at Scale

- 规模化敏捷开发
  - Agile at Scale (SEI Blog/SPRUCE — Systems and Software Producibility Collaboration Environment 系统和软件可生产性协作环境, 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

Agile at Scale

- 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.

Agile at Scale

- 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 quality attributes (严苛的质量特性) 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.

Agile at Scale

- 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.

Agile at Scale

- 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.

Agile at Scale

- 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 (实例化), 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".

Agile at Scale

- 规模化敏捷开发
  - 10 Recommended Practices for Achieving Agile at Scale
    - (1) Make team coordination top priority.
      - "Scrum of Scrums" is a special coordination team to
        - (a) define what information will flow between and among development teams (addressing inter-team dependencies and communication)
        - (b) identify, analyze, and resolve coordination issues and risks that have potentially broader consequences (e.g., for the project as a whole).
      - 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.

Agile at Scale

- 规模化敏捷开发
  - 10 Recommended Practices for Achieving Agile at Scale

    (1) Make team coordination top priority.

    - 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.

Agile at Scale

- 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).

Agile at Scale

- 规模化敏捷开发
  - 10 Recommended Practices for Achieving Agile at Scale

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

Agile at Scale

- 规模化敏捷开发
  - 10 Recommended Practices for Achieving Agile at Scale

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

    - 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.

Agile at Scale

- 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.

Agile at Scale

- 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.

Agile at Scale

- 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.

Agile at Scale

- 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.

    - Early in the project, evaluate the quality attribute scenarios to determine which architecturally significant requirement 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.
      - 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.

Agile at Scale

- 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, 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.

Agile at 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).

Agile at Scale

- 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.

Agile at Scale

- 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.

Agile at Scale

- 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.

Agile at Scale

- 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.

    - 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.

Agile at Scale

- 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 (技术债务) 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.

Agile at Scale

- 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.

    - 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.
    - 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).

Agile at Scale

- 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.

Agile at Scale

- 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.

Agile at Scale

- 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.

软件生命周期过程

- ISO/IEC 12207 软件生命周期过程
  - ISO/IEC/IEEE 12207: 2017
    - 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.

软件生命周期过程

- 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.

软件生命周期过程

- 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.

软件生命周期过程

- 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.

软件生命周期过程

- 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.

软件生命周期过程

- 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.

软件生命周期过程

- 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).

软件生命周期过程

- 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.

软件生命周期过程

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

软件生命周期过程

- 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.

Lecture 7. Agile Modeling & SLCP (2)

End of Lecture