Universidad Católica de Córdoba
Universidad Jesuita

Ingeniería de software III

Unidad 3 – Diseño en las metodologías ágiles

Programación Extrema

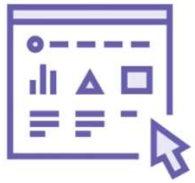
eXtreme Programming es una metodología de desarrollo de software la cual intenta mejorar la calidad del software y la velocidad de respuesta ante los requerimientos cambiantes del cliente.

Programación Extrema - Historia

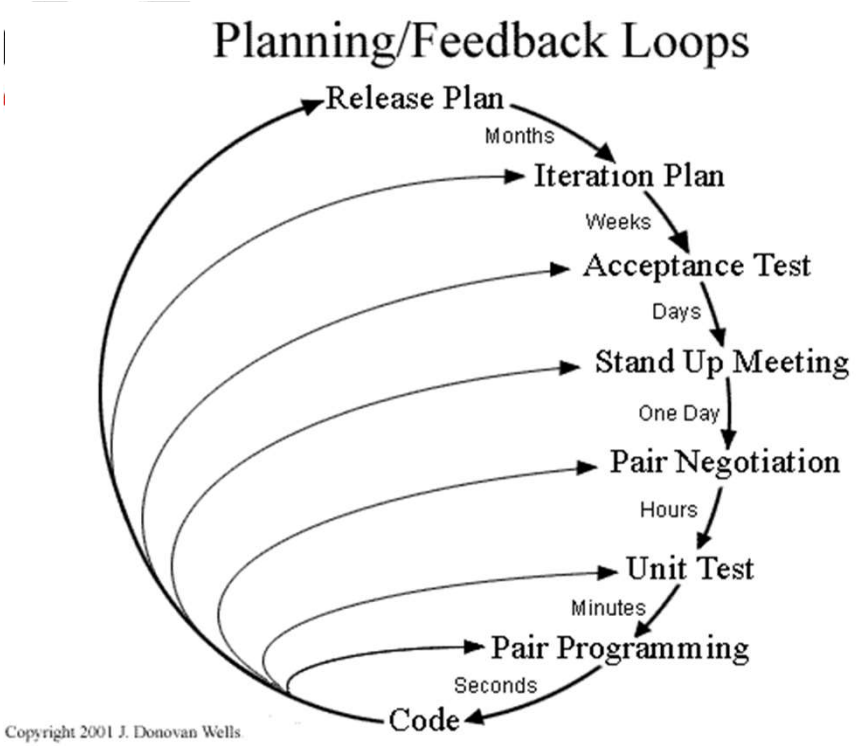
- Creada por Kent Beck
- C3 team (Chrysler)
- Refinamiento sucesivo
- Pequeños releases

- 4 Actividades
- 5 Valores
- 3 Principios
- 12 Prácticas
- 29 Reglas

Programación Extrema - Actividades

- Escribir código 
- Testear el sistema 
- Escuchar a los clientes y los usuarios 
- Diseñar para reducir el acoplamiento 

- La comunicación es esencial
- Simplicidad
- Aprender del feedback
- Tener coraje
- Respetar al team y al proyecto



Programación Extrema - Principios

- Feedback rápido
- Construir con simplicidad (DTSTTCPW, KISS, YAGNI)
- Cambio incremental
- Aceptar y valorar el cambio como algo positivo
- Trabajo de calidad

- Feedback detallado 
- Proceso continuo 
- Entendimiento común 
- Bienestar del programados 

Programación Extrema - Prácticas

Fine-scale Feedback

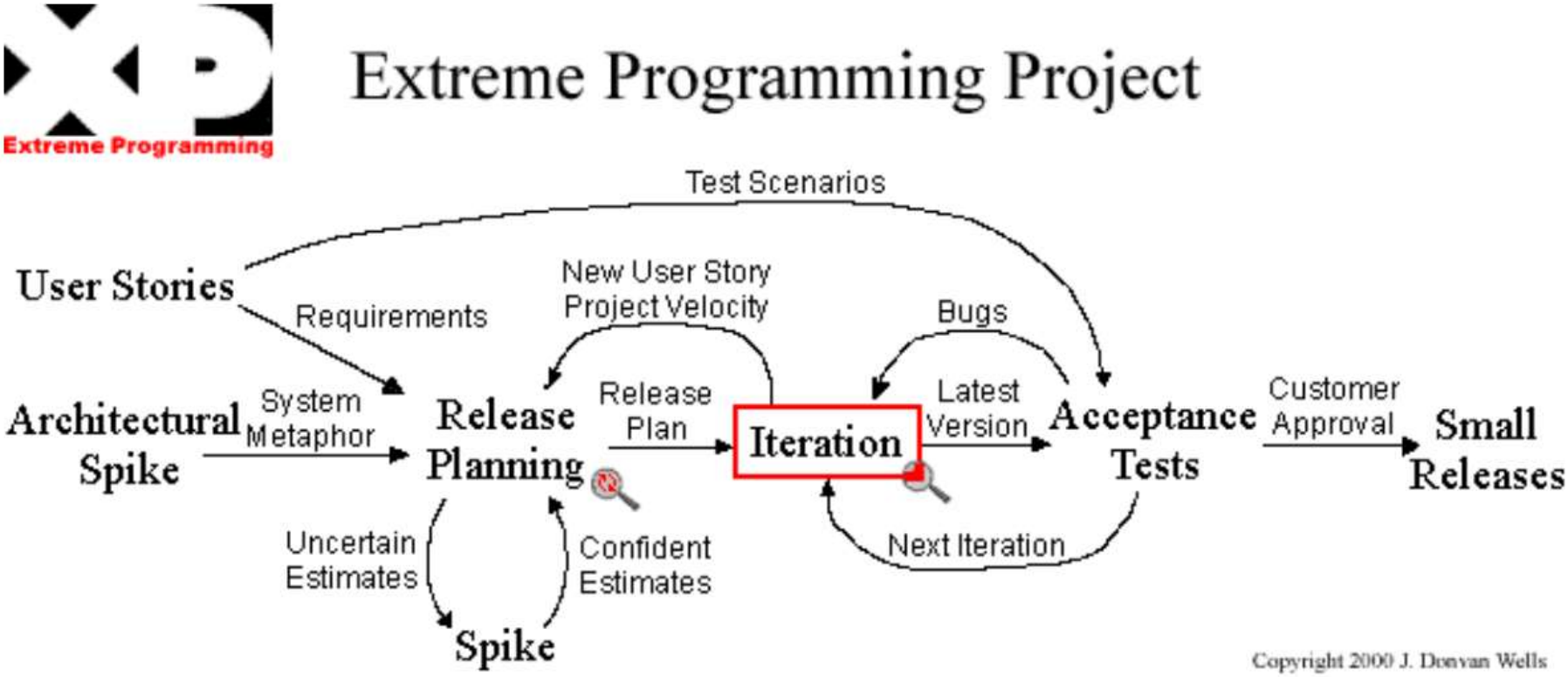
- Pair programming
- Planning game
- Test-driven development
- Whole team

Continuous Process

- Continuous integration
- Refactoring or design improvement
- Small releases

Programación Extrema - Prácticas

Programación Extrema – Flow



http://www.extremeprogramming.org/map/project.html

Prácticas de Extreme programming

Principio o práctica	Descripción
Planificación Incremental	Los requerimientos se registran en story cards y las stories que se van a incluir en un release se determinan por el tiempo disponible y la prioridad relativa. Los desarrolladores desglosan dichas historias en “tareas” de desarrollo.
Releases pequeños	Al principio se desarrolla el conjunto mínimo de funcionalidad útil, que ofrece valor para el negocio. Las liberaciones del sistema son frecuentes y agregan incrementalmente funcionalidad a la primera liberación.
Diseño Simple	El diseño suficiente se lleva a cabo para satisfacer las necesidades actuales y no más.
Desarrollo pruebas primero	Un marco de pruebas unitarias automatizadas es usado para escribir pruebas para una nueva función antes de que la funcion sea implementada.
Reconstrucción	Se espera de todos los desarrolladores de reconstruir el código continuamente ni bien las mejoras se hallan. Esto mantiene el codigo simple y mantenible.

Prácticas de Extreme programming

Programación en parejas	Los desarrolladores trabajan en parejas , revisando el trabajo mutuamente y dandose apoyo para hacer siempre un buen trabajo.
Propiedad colectiva	La pareja de desarrolladores trabaja en todas las áreas del sistema, para que no queden islas de conocimiento desarrolladas y todos los desarrolladores asumen responsabilidad por todo el código.
Integración continua	Tan pronto como el trabajo en una tarea esta completo, es integrado en el sistema. Luego de cada integración, se corren todas las pruebas unitarias.
Ritmo sostenible	Grandes cantidades de horas extras no se consideran aceptables como el efecto neto es a menudo para reducir la calidad del código y mediana productividad
Cliente en el sitio	Un representante de los usuarios finales del sistema (el cliente) debe estar disponible a tiempo completo para el equipo de XP. En un proceso de programación extrema, el cliente es miembro del equipo de desarrollo y es responsable de llevar requisitos del sistema para el equipo de implementación.

User stories en XP

Requerimientos: User Stories

✧En XP, el cliente o el usuario es parte del equipo de XP y es responsable de tomar decisiones sobre los requisitos

✧Solicitudes de los usuarios se expresan como escenarios o historias del usuario.

✧Estas se ordenan en un backlog por prioridades.

✧El equipo de desarrollo les asigna un costo estimado

✧El cliente elige las historias para su inclusión en el próximo release en base a sus prioridades y los costos estimados.

Requerimientos: User Stories + planeamiento incremental

✧En XP, el cliente o el usuario es parte del equipo de XP y es responsable de tomar decisiones sobre los requisitos

✧Solicitudes de los usuarios se expresan como escenarios o historias del usuario.

✧Estas se ordenan en un backlog por prioridades.

✧El equipo de desarrollo les asigna un costo estimado

Requerimientos: User Stories

✧User stories are part of an agile approach.

✧Helps shift the focus from writing about requirements to talking about them.

✧All agile user stories include a written sentence or two and, more importantly, a series of conversations about the desired functionality.

Requerimientos: User Stories

✧What is a user story?

✧User stories are short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system. They typically follow a simple template:

As a <type of user>, I want <some goal> so that <some reason>

✧User stories are often written on index cards or sticky notes, stored in a shoe box, and arranged on walls or tables to facilitate planning and discussion.

✧They strongly shift the focus from writing about features to discussing them.

✧These discussions are more important than whatever text is written.

Requerimientos: User Stories

✧Examples

✧One of the benefits of agile user stories is that they can be written at varying levels of detail. We can write a user story to cover large amounts of functionality. These large user stories are generally known as epics. Here is an epic agile user story example from a desktop backup product:

As a user, I can backup my entire hard drive.

✧Because an epic is generally too large for an agile team to complete in one iteration, it is split into multiple smaller user stories before it is worked on. The epic above could be split into dozens (or possibly hundreds), including these two:

As a power user, I can specify files or folders to backup based on file size, date created and date modified.

As a user, I can indicate folders not to backup so that my backup drive isn't filled up with things I don't need saved.

Requerimientos: User Stories

✧How is detail added to user stories?

✧Detail can be added to user stories in two ways:

By splitting a user story into multiple, smaller user stories.

By adding "conditions of satisfaction/success."

✧When a relatively large story is split into multiple, smaller agile user stories, it is natural to assume that detail has been added. After all, more has been written.

✧The conditions of satisfaction/success is simply a high-level acceptance test that will be true after the agile user story is complete.

Requerimientos: User Stories

✧Who writes user stories?

✧Anyone can write user stories. It's the product owner's responsibility to make sure a product backlog of agile user stories exists, but that doesn’t mean that the product owner is the one who writes them.

✧Over the course of a good agile project, you should expect to have user story examples written by each team member.

✧Also, note that who writes a user story is far less important than who is involved in the discussions of it.

Requerimientos: User Stories

✧When are user stories written?

- ✧User stories are written throughout the agile project.
- ✧Usually a story-writing workshop is held near the start of the agile project.
- ✧Everyone on the team participates with the goal of creating a product backlog that fully describes the functionality to be added over the course of the project or a three- to six-month release cycle within it.
- ✧Some of these agile user stories will undoubtedly be epics.
- ✧Epics will later be decomposed into smaller stories that fit more readily into a single iteration.
- ✧Additionally, new stories can be written and added to the product backlog at any time and by anyone.

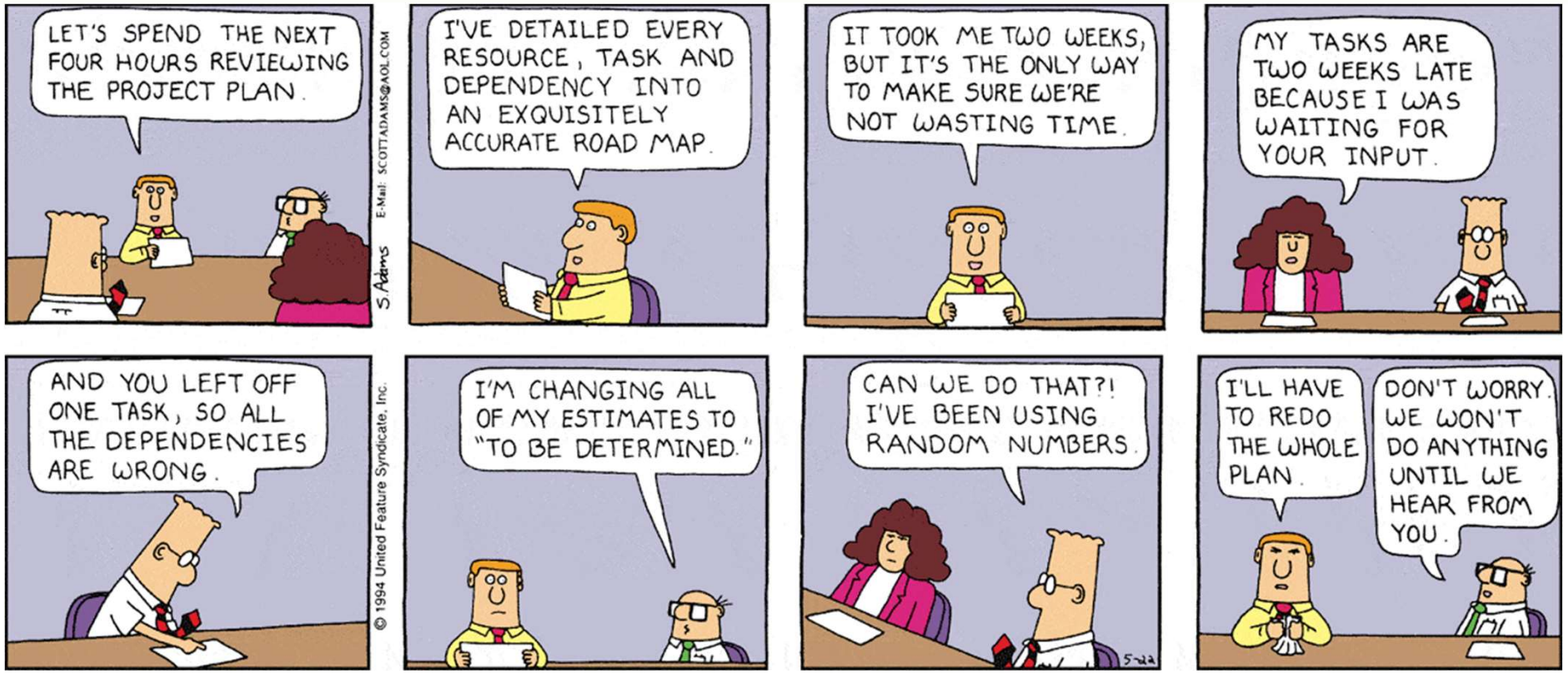

Generic Data Model

Requerimientos: User Stories

- User Stories Aren't Use Cases
  - Both deliver business value
  - Scope: stories are smaller, should fit an iteration
  - Level of completeness
  - Longevity: use case are long lived, stories are for an iteration
  - Use case prone to include user interface details

Requerimientos: User Stories

- User Stories Aren't Requirements Statements
  - List of "the system shall..." statements
    - Tedious, error-prone, time-consuming, up-front, completeness?
    - Focused on system and not user goals
  - Example:
    - *3.4) The product shall have a gasoline-powered engine.*
    - *3.5) The product shall have four wheels.*
    - *3.5.1) The product shall have a rubber tire mounted to each wheel.*
    - *3.6) The product shall have a steering wheel.*
    - *3.7) The product shall have a steel body.*
  - What would you build?


LET'S SPEND THE NEXT FOUR HOURS REVIEWING THE PROJECT PLAN.
I'VE DETAILED EVERY RESOURCE, TASK AND DEPENDENCY INTO AN EXQUISITELY ACCURATE ROAD MAP.
IT TOOK ME TWO WEEKS, BUT IT'S THE ONLY WAY TO MAKE SURE WE'RE NOT WASTING TIME.
MY TASKS ARE TWO WEEKS LATE BECAUSE I WAS WAITING FOR YOUR INPUT.
AND YOU LEFT OFF ONE TASK, SO ALL THE DEPENDENCIES ARE WRONG.
I'M CHANGING ALL OF MY ESTIMATES TO "TO BE DETERMINED."
CAN WE DO THAT?! I'VE BEEN USING RANDOM NUMBERS.
I'LL HAVE TO REDO THE WHOLE PLAN.
DON'T WORRY. WE WON'T DO ANYTHING UNTIL WE HEAR FROM YOU.
S. Adams
E-Mail: SCOTTADAMS@AOL.COM
© 1994 United Feature Syndicate, Inc.

Diseño en XP

Diseño en XP

- El sistema debe ser diseñado tan simple como sea posible

- El diseño correcto
  - Hace pasar todos los tests
  - No tiene lógica duplicada
  - Comunica las intenciones del desarrollador
  - Tiene la menor cantidad de unidades (clases, métodos, funciones, etc)

- Opuesto a
  - “Implementa para hoy, diseña para mañana”

- Porque
  - “El futuro es incierto”

- Ventajas del diseño simple
  - No se pierde tiempo en funcionalidad superflua
  - Más fácil de entender
  - Más fácil de “refactorizar” y de crear “propiedad colectiva” del código
  - Ayuda a mantener a los programadores en schedule

Refactoring en XP

XP y el cambio

✧Un precepto fundamental de la ingeniería de software tradicional es que se tiene que diseñar para cambiar. Esto es, deben anticiparse cambios futuros al software y diseñarlo de manera que dichos cambios se implementen con facilidad.

- ✧Vale la pena el gasto de tiempo y esfuerzo anticipando los cambios ya que esto reduce los costos más tarde en la vida del ciclo

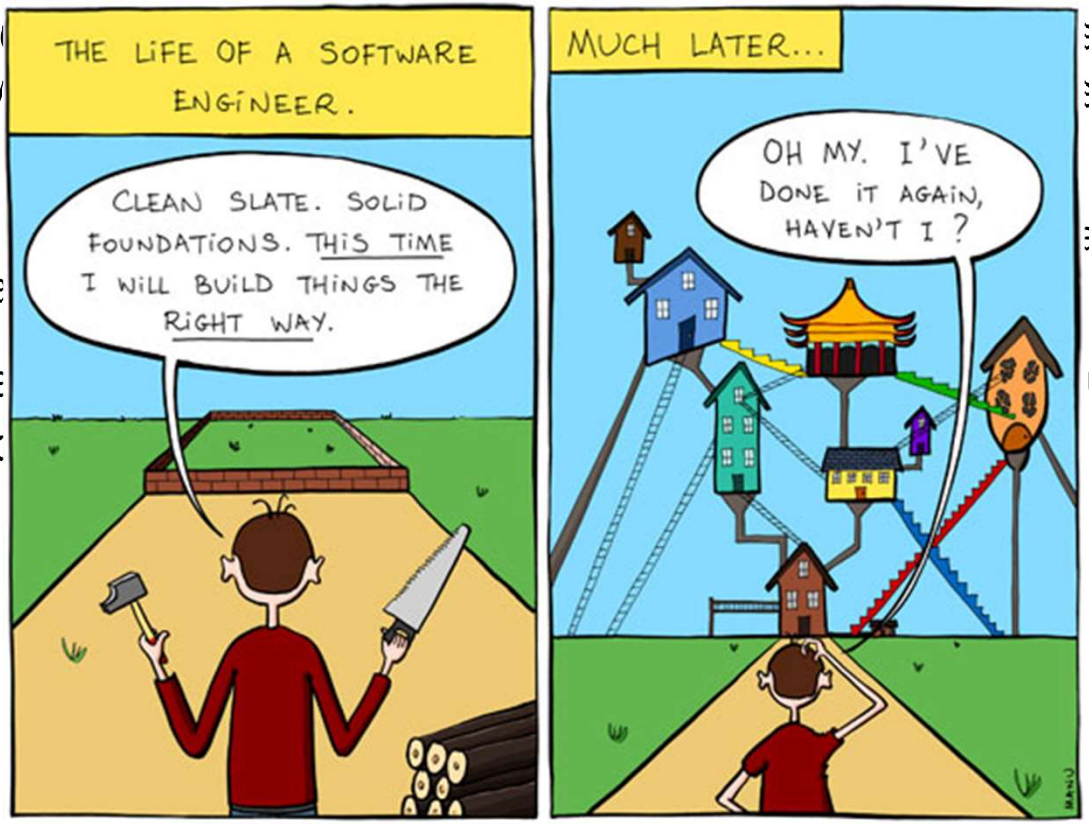
✧ XP, sin embargo, descarta este principio ya que sostiene que los cambios no se pueden prever de forma fiable

XP y el cambio

✧Un precepto fundamental de la ingeniería de software tradicional es que se tiene que (ambios futuros al software y ementen con facilidad.

✧Vale ambios ya que esto reduce

✧ XP, sin e ue los cambios no se pued



XP y el cambio

✧Un precepto fundamental de la ingeniería de software tradicional es que se tiene que diseñar para cambiar. Esto es, deben anticiparse cambios futuros al software y diseñarlo de manera que dichos cambios se implementen con facilidad.

- ✧Vale la pena el gasto de tiempo y esfuerzo anticipando los cambios ya que esto reduce los costos más tarde en la vida del ciclo

✧ XP, sin embargo, descarta este principio ya que sostiene que los cambios no se pueden prever de forma fiable

✧En su lugar, propone la mejora constante de código (Refactoring/Reconstrucción).

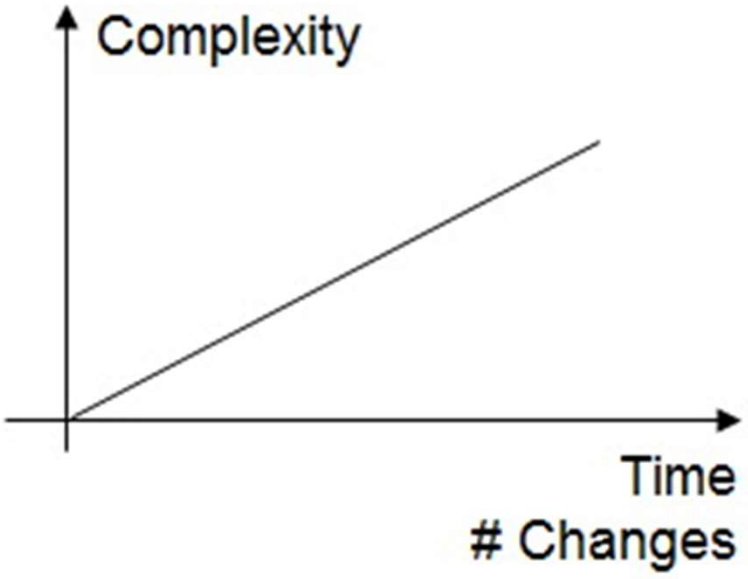
Refactoring

- El refactoring está motivado por los smells
  - Design/Code smells
  - Rotting software

✧Los cambios constantes degradan el diseño y el código

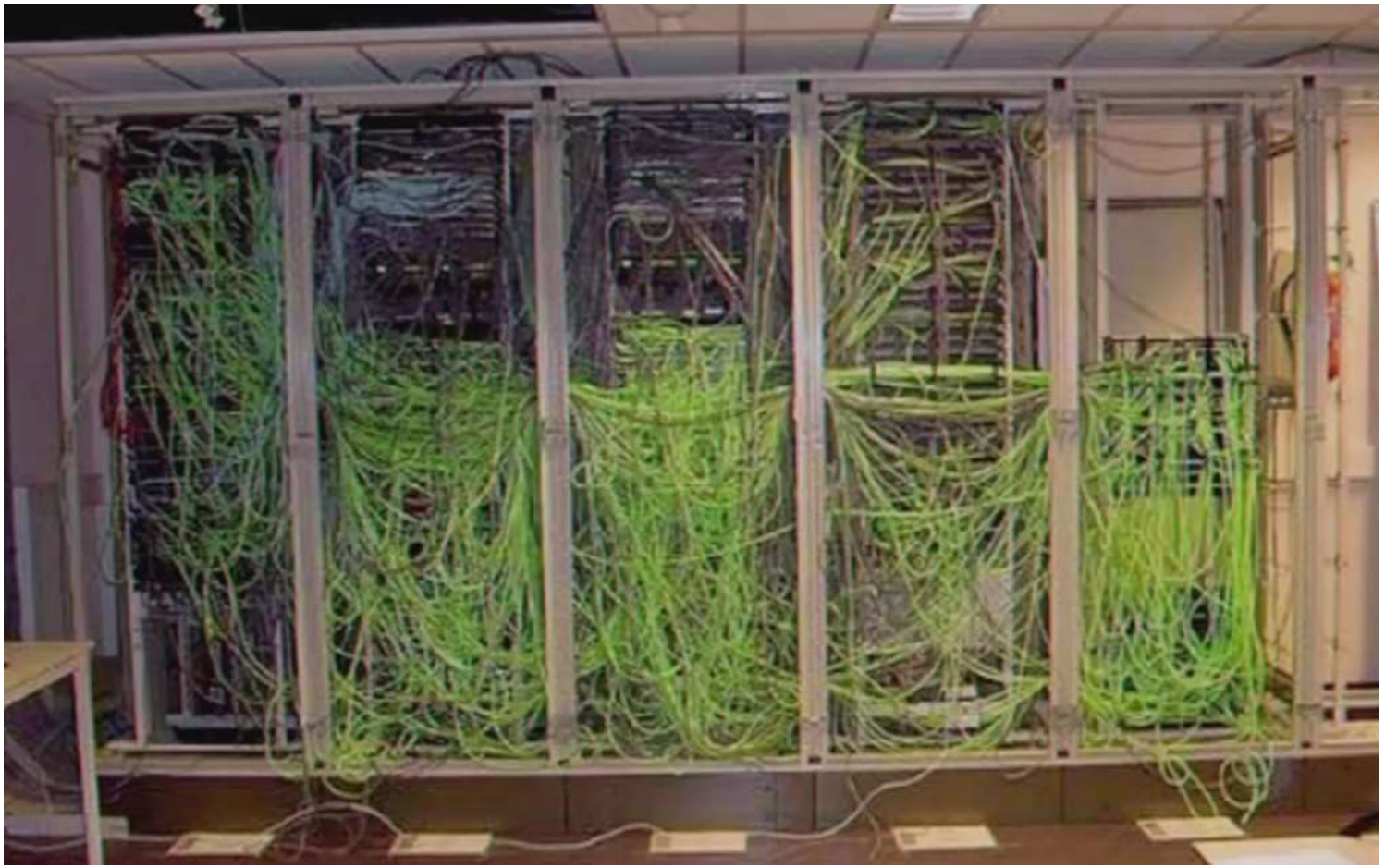
✧Signos de un mal diseño o un diseño degradado

- RIGIDEZ
- FRAGILIDAD
- IMMOBILIDAD
- VISCOSIDAD



✧Rigidez

✧Fragilidad



✧Inmovilidad

✧Viscosidad

Code Smell

- Code smell
  - Característica de un programa que indica un posible problema más profundo
  - Análisis subjetivo
  - Ejemplos:
    - Métodos muy largos
    - Métodos muy similares

Code Smell

- Code smell
  - Ciertas estructuras en el código que indican la violación de principios fundamentales de diseño e impactan de manera negativa la calidad del diseño
- No es un bug
- El código funciona correctamente
- Indican debilidad en el diseño que pueden
  - Hacer el desarrollo más lento
  - Aumentar la posibilidad de la introducción de bugs en el desarrollo

Code Smell

- Application-level smells:
  - Duplicated code: identical or very similar code exists in more than one location.
  - Contrived complexity: forced usage of overcomplicated design patterns where simpler design would suffice.
  - Shotgun surgery : a single change needs to be applied to multiple classes at the same time.

Code Smell

- Class-level smells:
  - Large class: a class that has grown too large. God object.
  - Feature envy: a class that uses methods of another class excessively.
  - Inappropriate intimacy: a class that has dependencies on implementation details of another class.
  - Refused request: a class that overrides a method of a base class in such a way that the contract of the base class is not honored by the derived class. Liskov substitution principle.
  - Lazy class / freeloader: a class that does too little.
  - Excessive use of literals: these should be coded as named constants, to improve readability and to avoid programming errors. Additionally, literals can and should be externalized into resource files/scripts, or other data stores such as databases where possible, to facilitate localization of software if it is intended to be deployed in different regions.

Code Smell

- Class-level smells:
  - Cyclomatic complexity: too many branches or loops; this may indicate a function needs to be broken up into smaller functions, or that it has potential for simplification.
  - Downcasting: a type cast which breaks the abstraction model; the abstraction may have to be refactored or eliminated.
  - Orphan variable or constant class: a class that typically has a collection of constants which belong elsewhere where those constants should be owned by one of the other member classes.
  - Data clump: Occurs when a group of variables are passed around together in various parts of the program. In general, this suggests that it would be more appropriate to formally group the different variables together into a single object, and pass around only this object instead.

Code Smell

- Method-level smells:
  - Too many parameters: a long list of parameters is hard to read, and makes calling and testing the function complicated. It may indicate that the purpose of the function is ill-conceived and that the code should be refactored so responsibility is assigned in a more clean-cut way.
  - Long method: a method, function, or procedure that has grown too large.
  - Excessively long identifiers: in particular, the use of naming conventions to provide disambiguation that should be implicit in the software architecture.
  - Excessively short identifiers: the name of a variable should reflect its function unless the function is obvious.
  - Excessive return of data: a function or method that returns more than what each of its callers needs.
  - Excessively long line of code (or God Line): A line of code which is so long, making the code difficult to read, understand, debug, refactor, or even identify possibilities of software reuse.

Technical debt

- Code smells son indicadores de factores que contribuyen al technical debt
- Technical debt
  - Un concepto en el desarrollo de software que refleja el costo implícito de rework causado por elegir una solución fácil en lugar de usar una mejor solución que hubiera tomado más tiempo en desarrollarse
- Se puede comparar con las deudas financieras, si el technical debt no se paga acumula intereses, haciendo cambios futuros más difíciles

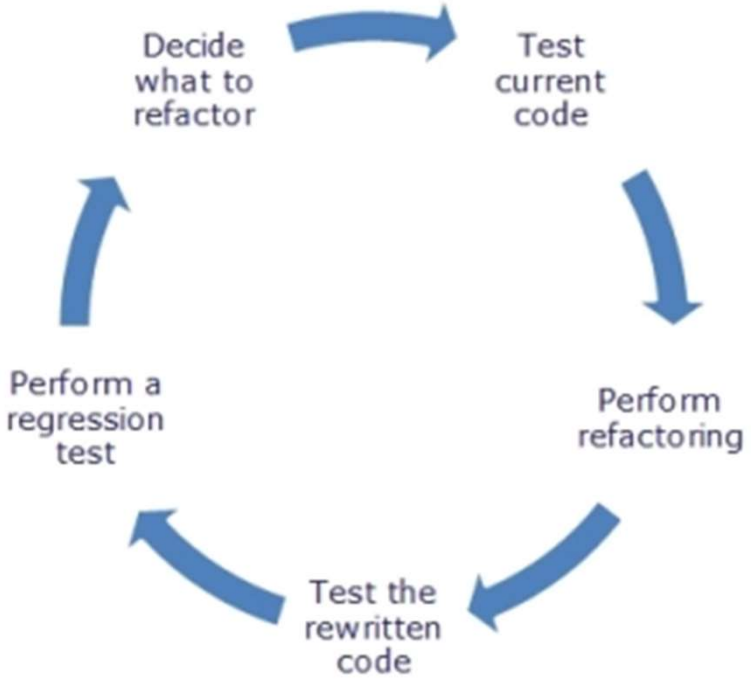
Refactoring

- Los code smells son un indicador de que se necesita refactoring
- Refactoring es una forma de pagar el technical debt
- Beneficios del refactoring:
  - Maintainability. It is easier to fix bugs because the source code is easy to read and the intent of its author is easy to grasp.
  - Extensibility. It is easier to extend the capabilities of the application if it uses recognizable design patterns, and it provides some flexibility where none before may have existed.

Refactoring

✧Un arnés de prueba automatizado con suficiente cobertura es obligatorio para poder hacer un buen refactoring

✧El refactoring se hace mediante ciclos iterativos de pequeñas transformaciones del programa y testing intercalados



Técnicas de refactoring

- Techniques that allow for more abstraction
  - Encapsulate field – force code to access the field with getter and setter methods
  - Generalize type – create more general types to allow for more code sharing
  - Replace type-checking code with state/strategy
  - Replace conditional with polymorphism

Técnicas de refactoring

- Techniques for breaking code apart into more logical pieces
  - Componentization breaks code down into reusable semantic units that present clear, well-defined, simple-to-use interfaces.
  - Extract class moves part of the code from an existing class into a new class.
  - Extract method, to turn part of a larger method into a new method. By breaking down code in smaller pieces, it is more easily understandable. This is also applicable to functions.

Técnicas de refactoring

- Techniques for improving names and location of code
  - Move method or move field – move to a more appropriate class or source file
  - Rename method or rename field – changing the name into a new one that better reveals its purpose
  - Pull up – in object-oriented programming (OOP), move to a superclass
  - Push down – in OOP, move to a subclass

Pruebas en XP

Pruebas en XP

- Los desarrolladores escriben UT
  - Deben pasar para poder progresar
- Los clientes escriben pruebas de aceptación con los desarrolladores
  - Deben pasar para saber cuando se terminó la tarea
- Los tests son automatizados
  - Se usan como regresión
- Crean un sistema que acepta el cambio

Pruebas en XP

- Ventajas de las pruebas en XP
  - Promueven un set de pruebas completo
  - Le dan al desarrollador un objetivo
  - La automatización crea un suite de regresión que como seguro durante el “refactoring"

Las pruebas en XP

✧Las pruebas son fundamentales para XP y XP ha desarrollado un enfoque en el que el programa se comprueba después de que cada cambio se ha realizado.

✧Funciones de prueba de XP:

- Desarrollo de las pruebas en primer lugar
- Desarrollo de pruebas incrementales a partir de escenarios.
- Participación de los usuarios en el desarrollo de la prueba y validación.
- Arneses de pruebas automatizadas se utilizan para ejecutar todas las pruebas de componentes cada vez que una nueva versión está construida.

Desarrollo de las pruebas primero

- ✧Pruebas antes del código => aclara los requerimienots
- ✧Pruebas == programas != texto => se pueden ejecutar de forma automática
- ✧Pruebas anteriores y nuevas =>
  - ✧se ejecutan automáticamente
  - ✧Cada ves que se agrega una nueva funcionalidad
  - ✧Compruebo que la nueva funcionalidad
    - ✧Funciona
    - ✧No agregó errores (regression)

Participación de los clientes

✧El papel del cliente en el proceso de pruebas es ayudar a desarrollar pruebas de aceptación de las historias que han de ser implementadas en la próxima versión del sistema

✧El cliente que es parte del equipo de pruebas, escribe pruebas simultáneamente al desarrollo. Por consiguiente, todo nuevo código es validado para asegurarse de que es lo que necesita el cliente.

✧No obstante, las personas que adoptan el papel de clientes tienen tiempo disponible limitado y por lo tanto no pueden trabajar a tiempo completo con la equipo de desarrollo. Pueden pensar que la presentación de la requerimientos es suficiente contribución y por tanto pueden ser reacios a involucrarse en el proceso de prueba.

La automatización de pruebas

✧ La automatización de pruebas significa que las pruebas se escriben como componentes ejecutables antes de que la tarea se implemente

- Estos componentes de prueba deben ser independientes, deberían simular la presentación de la entrada para ser probado y debe comprobar que el resultado cumple con las especificaciones de salida. Un marco de prueba automatizada (por ejemplo Junit) es un sistema que hace que sea fácil de escribir pruebas ejecutables y presentar un conjunto de pruebas para su ejecución

✧Como se automatiza las pruebas, siempre hay un conjunto de pruebas que puede ser rápida y fácilmente ejecutado

- Siempre que se agrega alguna funcionalidad al sistema, las pruebas se pueden corren y los problemas que ha introducido el nuevo código puede ser atrapadas inmediatamente

Dificultades en pruebas XP

✧Los programadores prefieren programación a las pruebas y a veces se toman atajos al escribir pruebas. Por ejemplo, pueden escribir ensayos incompletos que no comprueban todas las posibles excepciones que puedan ocurrir.

✧Algunas pruebas pueden ser muy difíciles de escribir de forma incremental. Por ejemplo, en una interfaz de usuario compleja, es a menudo difícil de escribir pruebas unitarias para el código que implementa la 'lógica de visualización "y flujo de trabajo entre las pantallas

✧Es difícil juzgar la integridad de un conjunto de pruebas. Aunque usted puede tener un montón de pruebas del sistema, la prueba de conjunto puede no proporcionar una cobertura completa.

Programación en parejas en XP

Programación en parejas

- En XP, los programadores trabajan en parejas,
  - Sentados juntos a desarrollar código.
  - Los pares se crean dinámicamente
- Ventajas
  - Ayuda a desarrollar la propiedad común de código
  - Difunde el conocimiento a través del equipo.
  - Sirve como un proceso de revisión informal
    - Cada línea de código es visto por más de 1 persona.
  - Alienta a la reconstrucción
- Productividad programación en pareja similar a dos personas trabajando de forma independiente

Programación en parejas

✧En XP, los programadores trabajan en parejas, sentados juntos a desarrollar código.

✧Esto ayuda a desarrollar la propiedad común de código y difunde el conocimiento a través del equipo.

✧Sirve como un proceso de revisión informal, ya que cada línea de código es visto por más de 1 persona.

Puntos clave

✧Los métodos ágiles

✧Son métodos incrementales de desarrollo que se centran en el desarrollo rápido, versiones frecuentes del software, reduciendo los gastos generales del proceso.

✧Implican al cliente directamente en el proceso de desarrollo.

✧ La decisión sobre si se debe utilizar un enfoque ágil o un enfoque dirigido por un plan para el desarrollo debe depender del tipo de software que está desarrollado, las capacidades del equipo de desarrollo y la cultura de la empresa que desarrolla el sistema.

✧ La programación extrema es un método ágil bien conocido que integra una serie de buenas prácticas de programación tales como versiones frecuentes del software, el software de mejora continua y participación del cliente en el equipo de desarrollo.

SCRUM
Gestión de proyectos ágiles

Gestión de proyectos ágil

- La principal responsabilidad de los directores de proyectos de software es la gestión del proyecto para que el software se entregue a tiempo y dentro del presupuesto previsto para el proyecto
- El enfoque estándar para la gestión de proyectos es el direccionado por plan.
  - Los gerentes elaboran un plan para el proyecto mostrando qué debería ser entregado, cuándo debería ser entregado y quién trabajara en el desarrollo del proyecto
- La gestión de proyectos ágil requiere un enfoque diferente, que está adaptado para desarrollo incremental y la fortalezas particulares de los métodos ágiles.

Gestión de proyectos ágil

- Manifiesto Ágil
  - Valores y Principios
  - Énfasis en colaboración y comunicación, software funcionando, etc
  - En contraste con el PMBOK
  - No provee pasos concretos
- Se busca una metodología o marco de trabajo ágil
  - Generalmente scrum más prácticas de XP u otras metodologías

Scrum

Universidad Católica de Córdoba
Universidad Jesuita

Scrum es un marco de trabajo ágil iterativo e incremental para gestionar el desarrollo de un producto.

Define una estrategia flexible donde el equipo de desarrollo trabaja como una unidad para alcanzar una meta común.

Scrum - Historia

- Creado por
  - Ken Schwaber

  - Jeff Sutherland

- Hereda el nombre de scrum del paper "The new product development game"

Scrum

- ✧Un marco de trabajo por el cual
  - ✧Se puede acometer problemas complejos adaptativos
  - ✧Entregar productos del máximo valor posible productiva y creativamente
  - ✧Ligero
  - ✧Fácil de entender
  - ✧Extremadamente difícil de llegar a dominar
- ✧No es un proceso o una técnica para construir productos
- ✧Es un marco de trabajo dentro del cual se pueden emplear varias técnicas y procesos
- ✧Scrum muestra la eficacia relativa de las prácticas de gestión de producto y las prácticas de desarrollo, de modo que podamos mejorar
- ✧Consiste de
  - ✧Equipo, Roles, Eventos y Artefactos

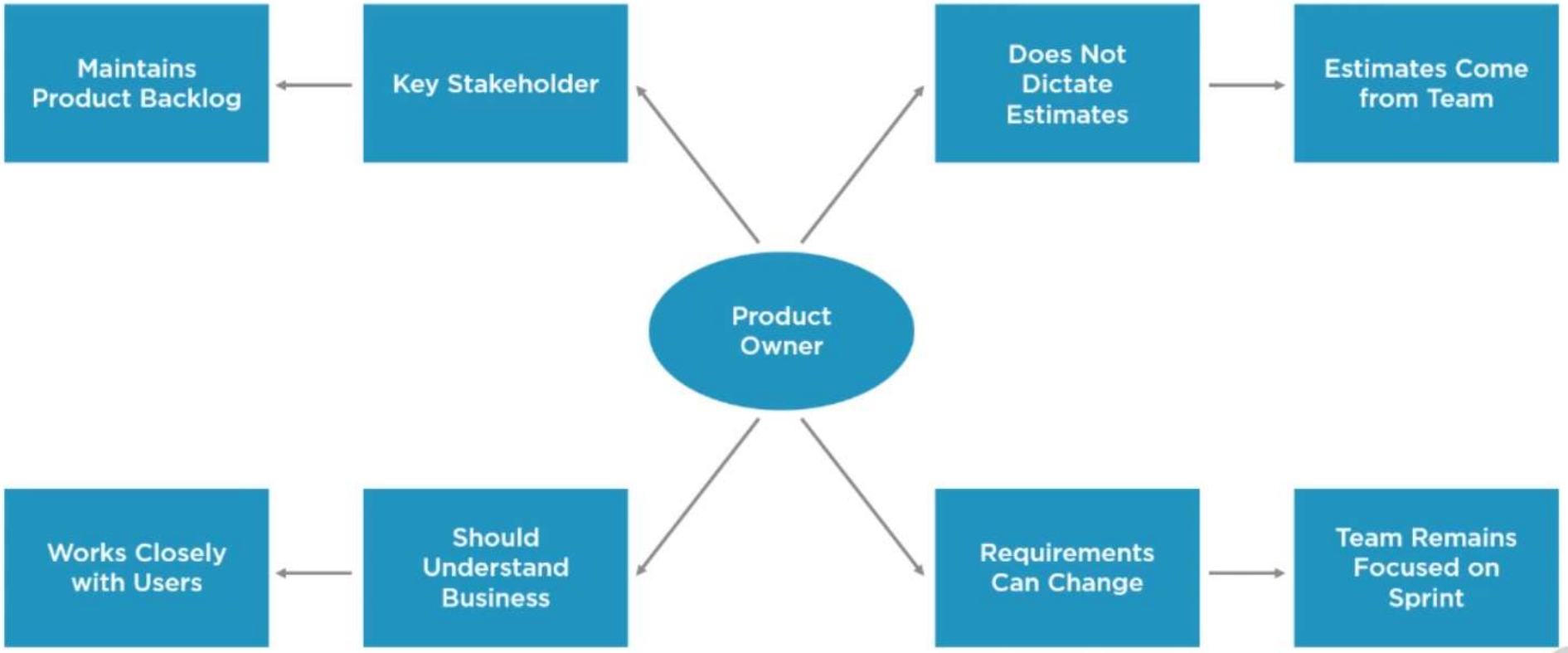
Scrum

- Teoría del control de proceso empírica o empirismo
  - Tomar decisiones
    - basados en hechos reales y no especulaciones
    - basados en la experiencia
- Iterativo e incremental
- Pilares del control empírico:
  - Transparencia
  - Inspección (en busca de variaciones)
  - Adaptabilidad

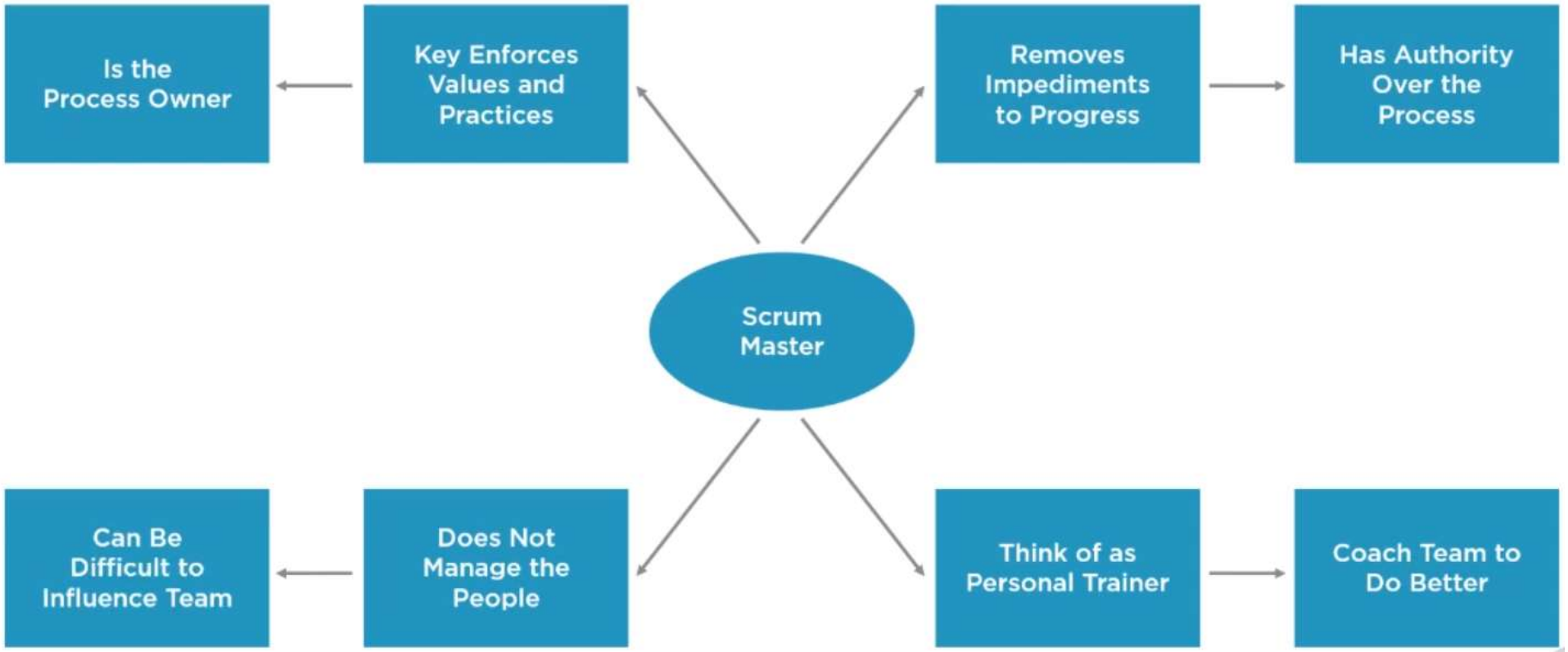
Scrum

- *Ciclos* de feedback *"inspect and adapt"*
- El tiempo se divide en ciclos cortos conocidos como sprints
- El producto está en estado "potencialmente entregable" en todo momento
- Al final de cada ciclo se muestran los resultados y se planea el próximo
- Eventos formales:
  - Reunión de Planificación del Sprint (Sprint Planning Meeting)
  - Scrum Diario (Daily Scrum)
  - Revisión del Sprint (Sprint Review)
  - Retrospectiva del Sprint (Sprint Retrospective)

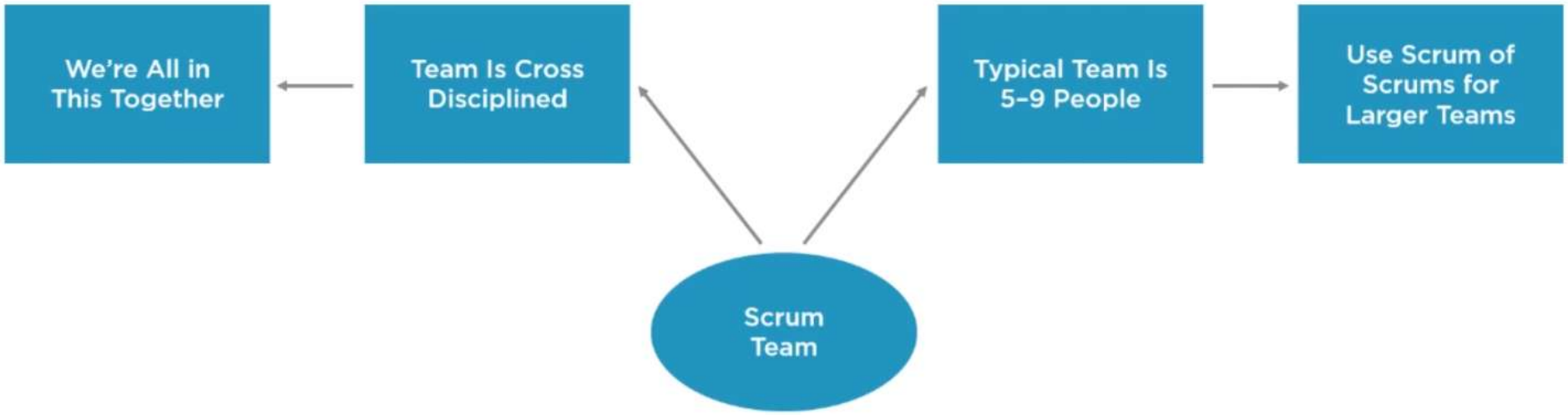
Scrum – Roles – Product Owner



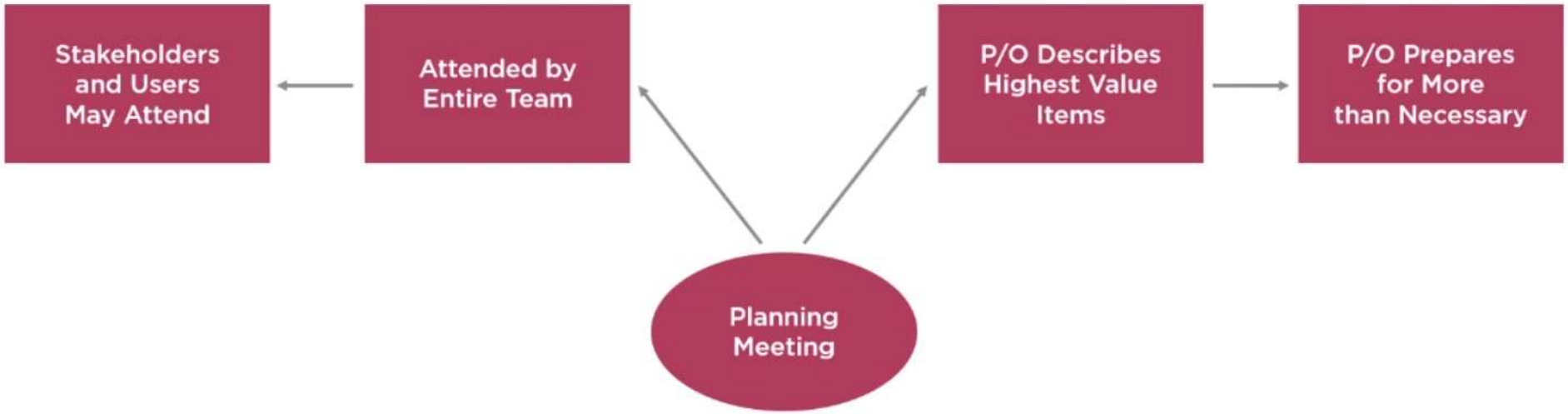
Scrum – Roles – Scrum Master



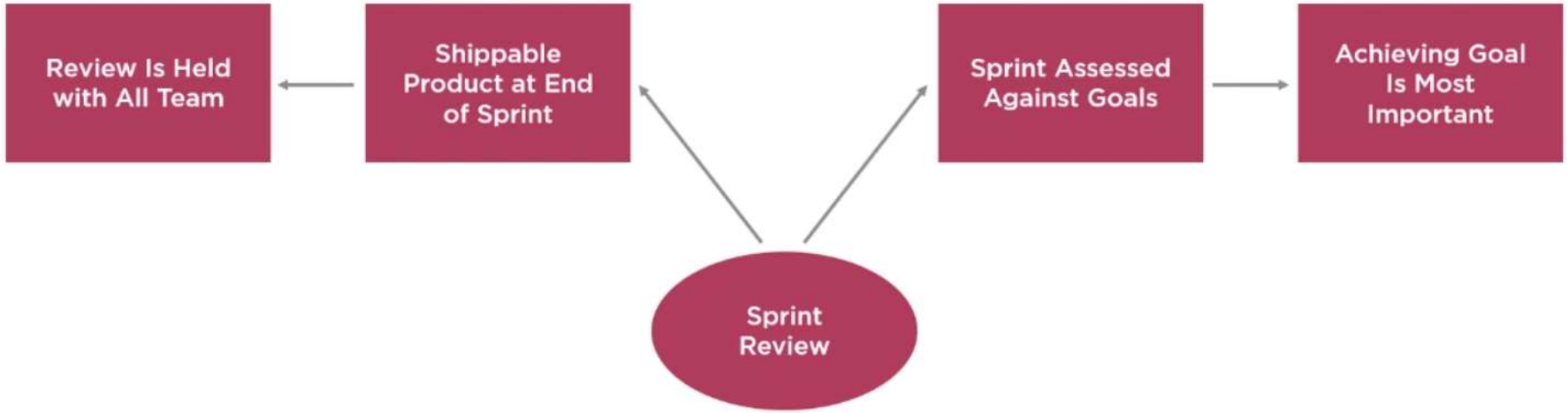
Scrum – Roles – Scrum Team



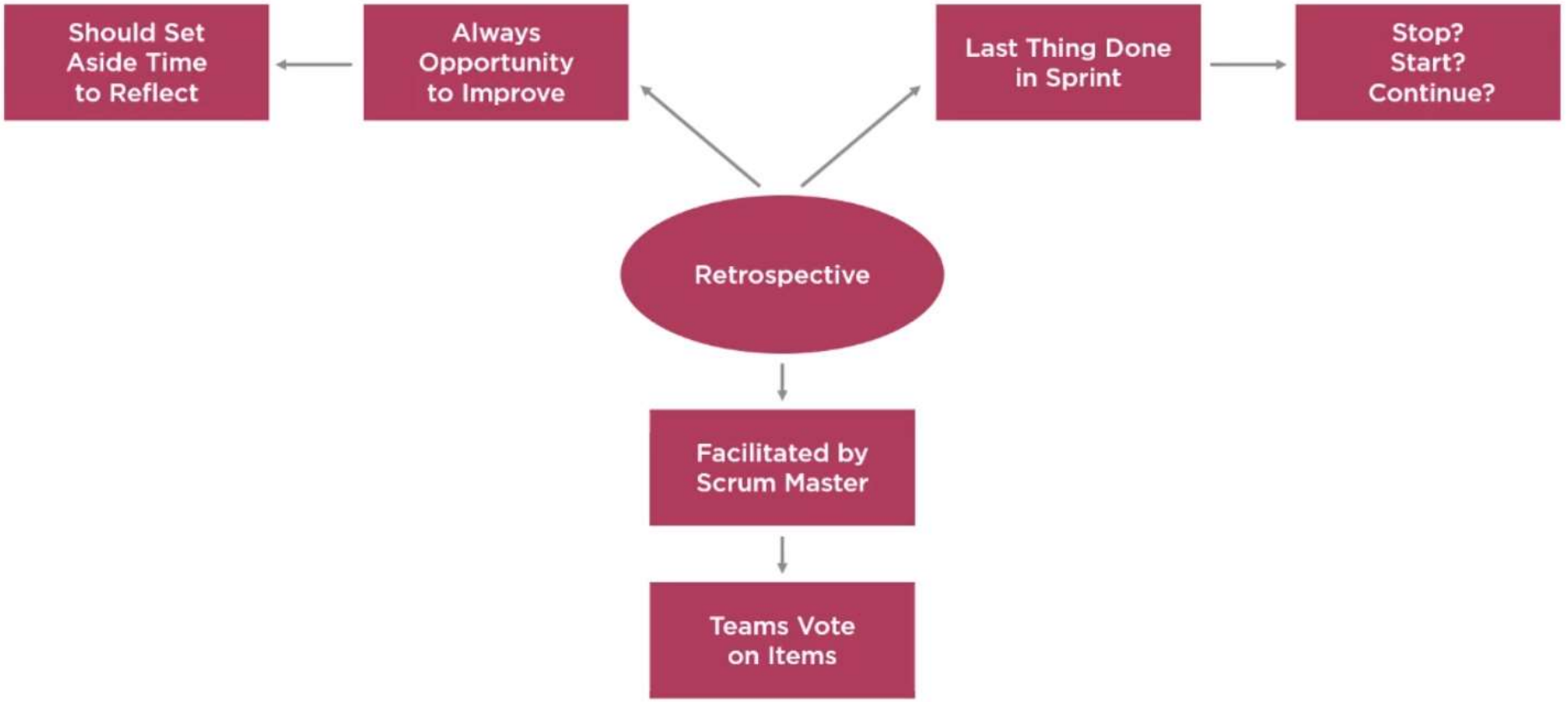
Scrum – Eventos - Planning



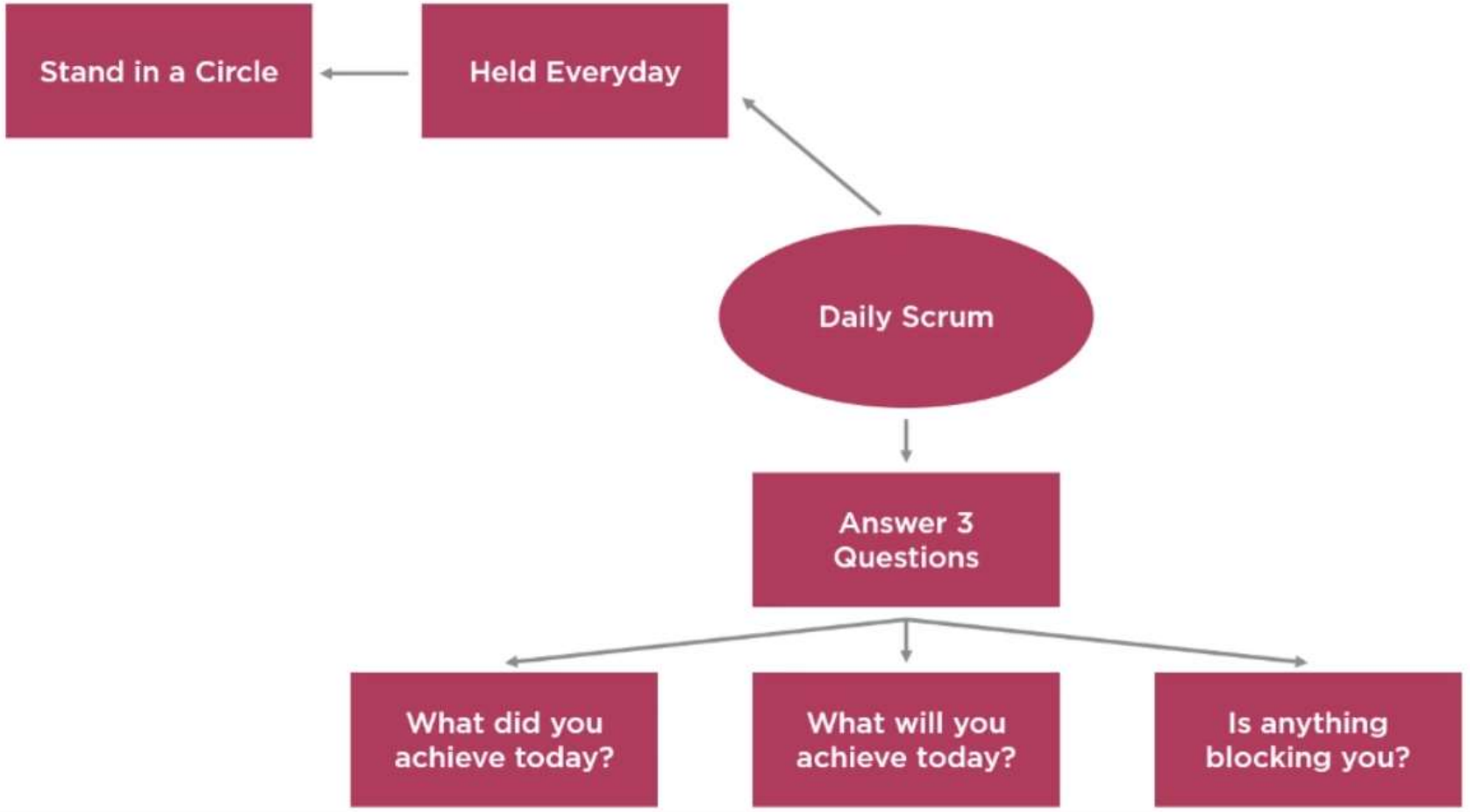
Scrum – Eventos – Sprint Review



Scrum – Eventos – Sprint Retrospective

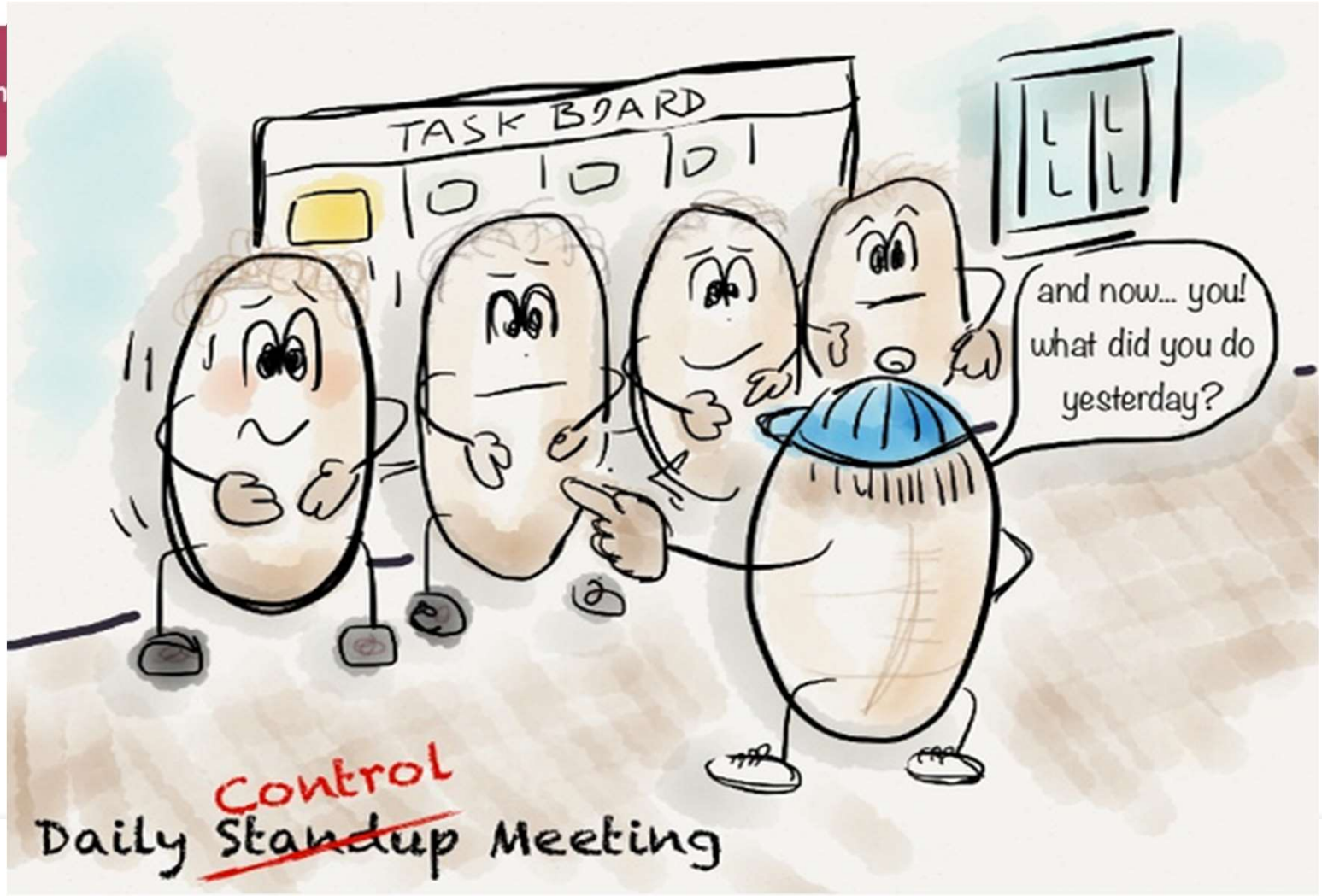


Scrum – Eventos – Daily Scrum

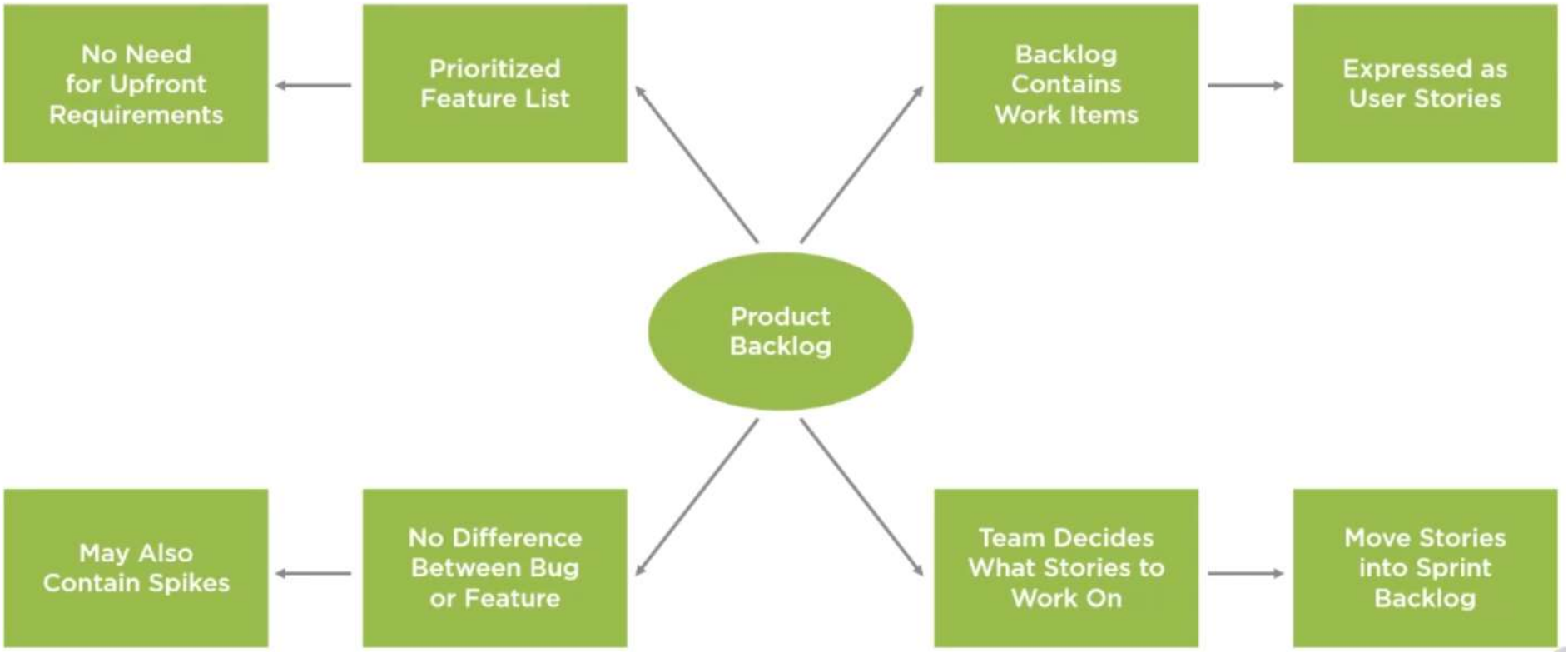


Scrum – Eventos – Daily Scrum

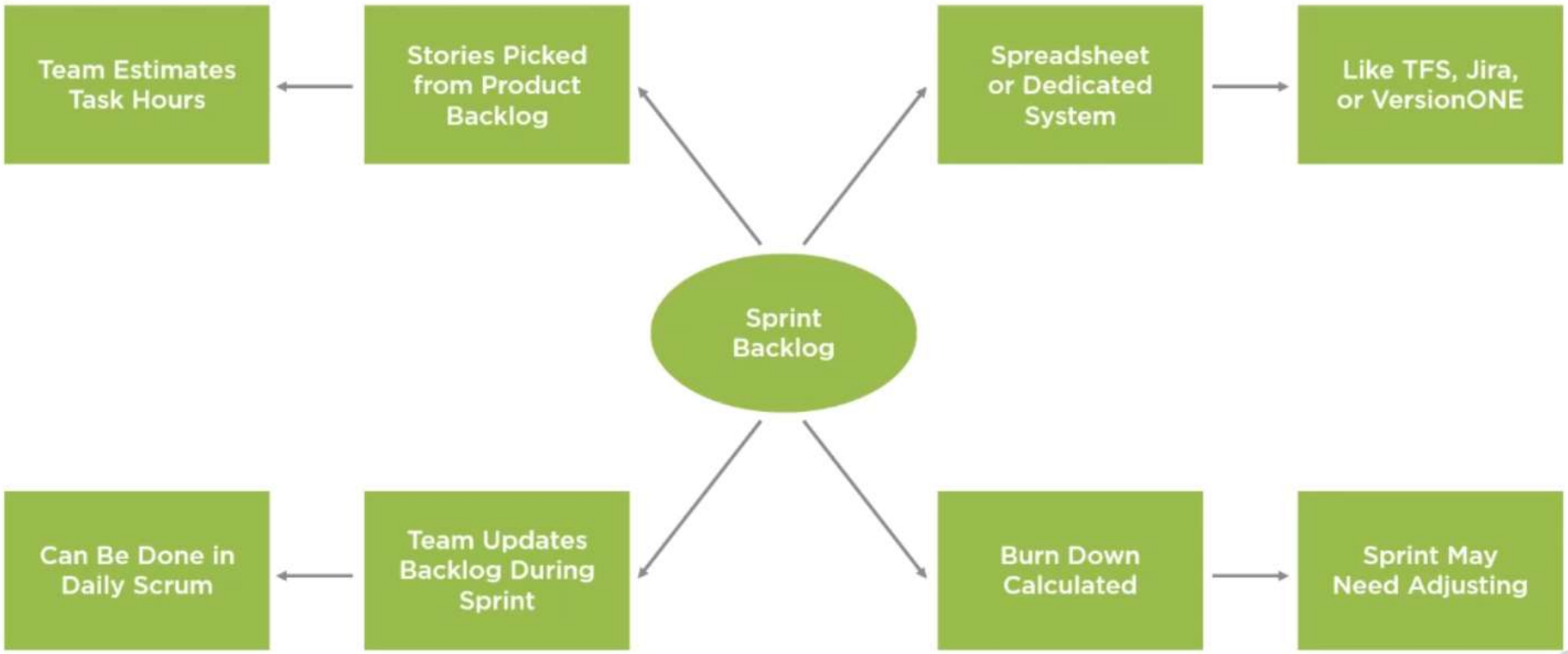
Stand in



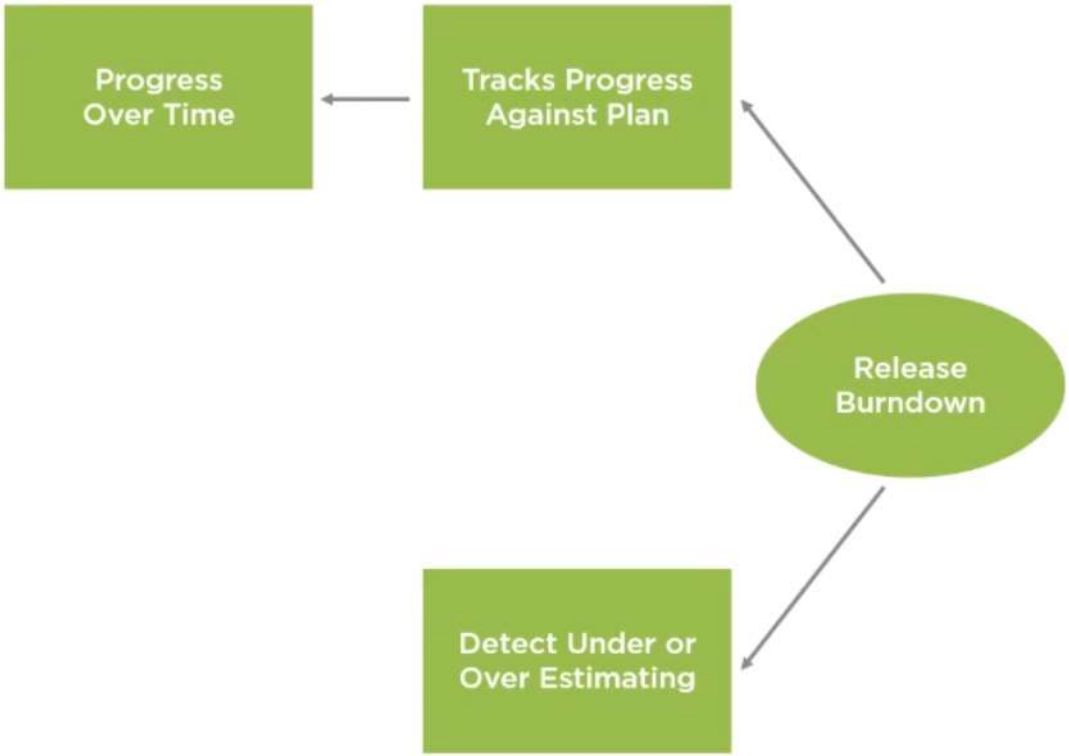
Scrum – Artifacts – Product Backlog



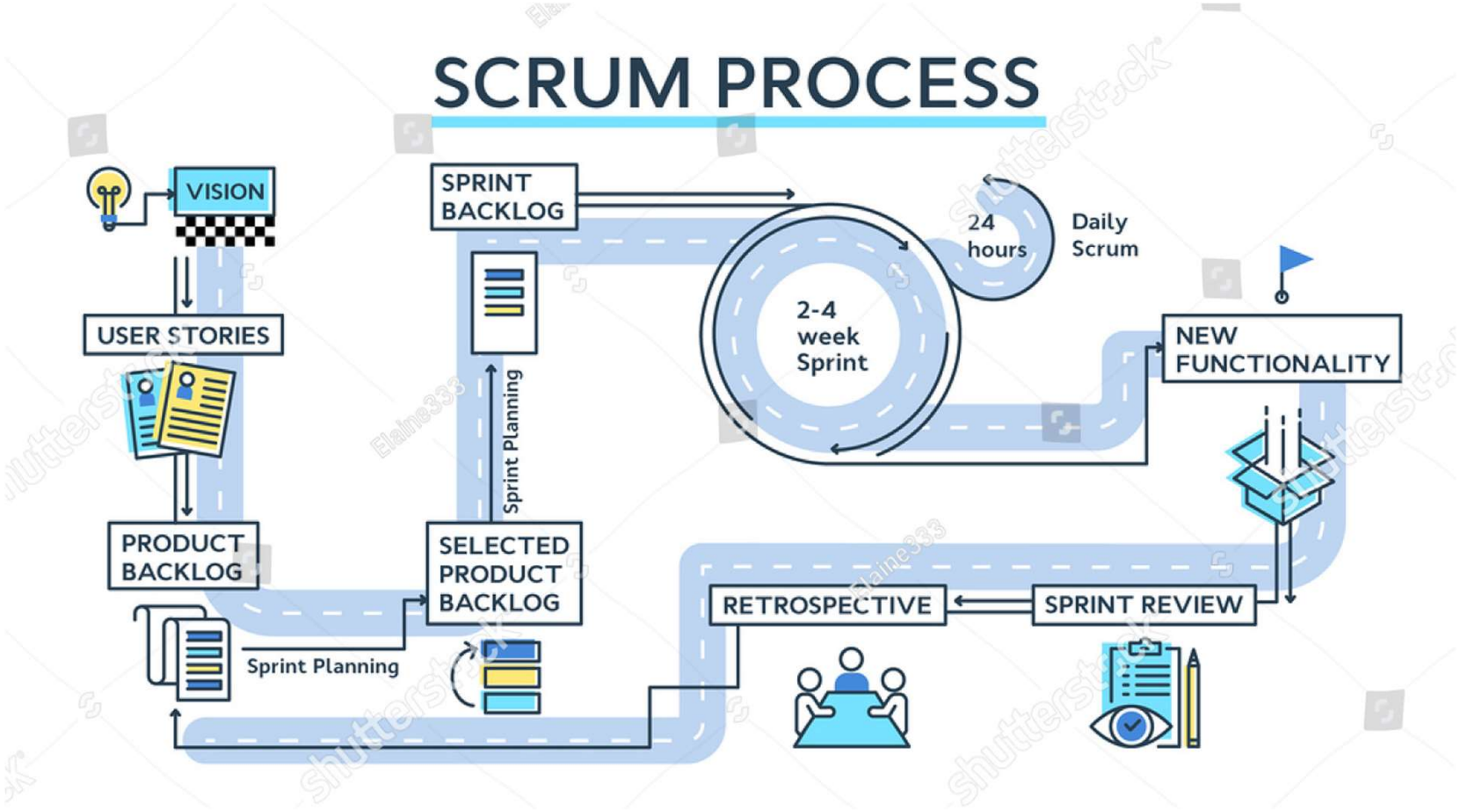
Scrum – Artifacts – Sprint backlog



Scrum – Artifacts – Burndown Chart



El Proceso Scrum



Scrum terminology (a)

Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than 7 people. They are responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a sprint. The idea is that this should be 'potentially shippable' which means that it is in a finished state and no further work, such as testing, is needed to incorporate it into the final product. In practice, this is not always achievable.
Product backlog	This is a list of 'to do' items which the Scrum team must tackle. They may be feature definitions for the software, software requirements, user stories or descriptions of supplementary tasks that are needed, such as architecture definition or user documentation.
Product owner	An individual (or possibly a small group) whose job is to identify product features or requirements, prioritize these for development and continuously review the product backlog to ensure that the project continues to meet critical business needs. The Product Owner can be a customer but might also be a product manager in a software company or other stakeholder representative.

Scrum terminology (b)

Scrum term	Definition
Scrum	A daily meeting of the Scrum team that reviews progress and prioritizes work to be done that day. Ideally, this should be a short face-to-face meeting that includes the whole team.
ScrumMaster	The ScrumMaster is responsible for ensuring that the Scrum process is followed and guides the team in the effective use of Scrum. He or she is responsible for interfacing with the rest of the company and for ensuring that the Scrum team is not diverted by outside interference. The Scrum developers are adamant that the ScrumMaster should not be thought of as a project manager. Others, however, may not always find it easy to see the difference.
Sprint	A development iteration. Sprints are usually 2-4 weeks long.
Velocity	A measure of how much product backlog effort that a team can cover in a single sprint. Understanding a team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring improving performance.

El Ciclo de Sprint

✧Los sprints son de longitud fija, normalmente 2-4 semanas. Se corresponden al desarrollo de una versión del sistema en XP.

✧El punto de partida para la planificación es la acumulación de stories, que es la lista de trabajo a realizar en el proyecto.

✧La fase de selección involucra a todo el equipo del proyecto, que trabajan con el cliente para seleccionar las funciones y funcionalidad que se desarrollará durante el sprint. Basados en:

- ✧Prioridades
- ✧Esfuerzo estimado
- ✧Velocidad del team

Planeamiento

- ✧Estimaciones
- ✧Punto de Historia
- ✧Poker planning
- ✧Velocidad

El Ciclo de Sprint

✧Una vez que éstos están de acuerdo, el equipo se organizan para desarrollar el software. Durante esta etapa, el equipo está "aislado" del cliente y la organización, con toda comunicaciones canalizadas a través del denominado 'Scrum Master‘.

✧El papel del Scrum Master es proteger el equipo de desarrollo de las distracciones externas.

✧Al final del sprint, el trabajo realizado es revisado y se presentó a las partes interesadas. El siguiente ciclo de comienza nuevamente

Trabajo en equipo en Scrum y Daily Stand-ups

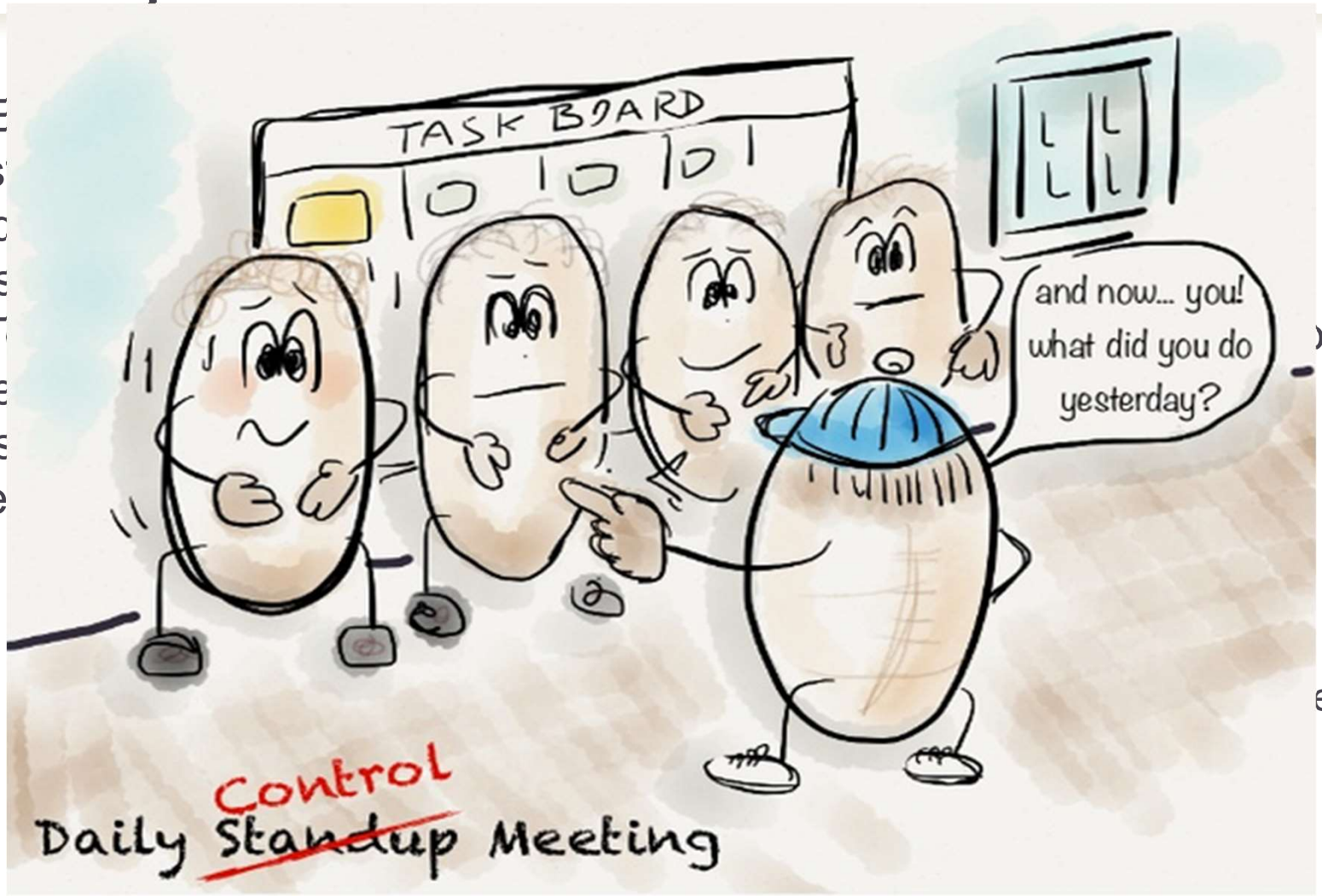
✧El 'Scrum Master' es un facilitador que organiza reuniones diarias, rastrea la acumulación de trabajo por hacer, registra las decisiones, mide el progreso contra el atraso y se comunica con los clientes y la gestión fuera del equipo.

✧Todo el equipo asiste a las reuniones diarias cortas donde todos los miembros del equipo comparten información, describen su progreso desde la última reunión, los problemas que han surgido y que se ha previsto para el día siguiente.

- ▪Esto significa que todos en el equipo saben lo que está pasando y, si surgen problemas, puede volver a planear el trabajo a corto plazo para hacer frente a ellos.
- ▪Preguntas en los daily stand ups:¿Qué hice ayer?¿Qué planeo hacer hoy?¿Tengo algún impedimento?

Trabajo en equipo en Scrum y Daily Stand-ups

✧E
ras
mic
ges
✧T
mie
des
pre



os

er

Beneficios de Scrum

✧El producto se divide en un conjunto de fragmentos manejables y comprensibles.

✧Requerimientos inestables no retrasan el progreso.

✧Todo el equipo tiene visibilidad de todo y por lo tanto se mejora la comunicación del equipo.

✧Los clientes ven la entrega a tiempo de los incrementos y obtienen retroalimentación sobre cómo funciona el producto.

✧La confianza entre los clientes y los desarrolladores se establece y una cultura positiva se crea en la que todo el mundo espera que el proyecto tenga éxito.

Preguntas