
# Equipos de desarrollo de software: sus prácticas representadas en Semat.

**Jorge Orlando Muñoz Rengifo**

Universidad Nacional de Colombia

Facultad de Ingeniería

Departamento de Ciencias de la Computación y de la Decisión

Medellín, Colombia

2015

# Equipos de desarrollo de software: sus prácticas representadas en Semat.

**Jorge Orlando Muñoz Rengifo**

Trabajo de investigación presentado como requisito parcial para optar al título de:

**Magister en Ingeniería de Sistemas**

Director (a):

Carlos Mario Zapata Jaramillo, Ph.D.

Línea de Profundización:

Ingeniería de Software

Universidad Nacional de Colombia

Facultad de Ingeniería

Departamento de Ciencias de la Computación y de la Decisión

Medellín, Colombia

2015

*Elegance is not a dispensable luxury but a factor that decides between success and failure.*

*Edsger Dijkstra*

# Resumen

En este Trabajo Final de Maestría se analizan algunos factores críticos de éxito de los proyectos de desarrollo de software y las formas en que se busca mitigar el fracaso en dichos proyectos. Se propone una forma de trabajo, en métodos de desarrollo no ágiles, de las prácticas de los equipos auto-administrados de desarrollo de software, con el fin de reducir los factores de fracaso en los proyectos de software. La forma de reutilización de las prácticas, consiste en emplear *Semat* (*Software Engineering Method and Theory*) para representar las prácticas dentro de su núcleo y aplicarlas en el método de desarrollo en cascada. Se desarrolla un caso de estudio donde se modificó un método de desarrollo, removiendo elementos propios del método y adicionando prácticas externas al método de desarrollo.

**Palabras clave: *Semat*, prácticas de desarrollo de software, métodos de desarrollo de software.**

# Abstract

In this M.Sc. final report, we analyze some critical success factors of software development projects and the ways to deal with failure in such projects. We also propose a way of working to use practices of self-managing teams on non agile development methods, with the aim of reducing failure factors in software development projects. We use Semat (Software Engineering Method and Theory) as a way to reuse practices by representing the practices inside the Semat kernel and applying then to the waterfall development method. A case study with the modification of a software development method is developed. In such a case, some elements of the method were removed and some external practices were added.

**Keywords: Semat, Software development practices, Software development methods.**

# Contenido

Pág.

# Lista de figuras

# Lista de tablas

# Introducción

El desarrollo de software se considera una actividad propensa al fracaso y, desde sus inicios en los años 60's, se busca disminuir el fracaso en los proyectos de software (Brooks, 1986; Cerpa & Verner, 2009). Entre las soluciones para disminuir ese fracaso se encuentran los métodos de desarrollo de software, los cuales se pueden separar en dos grupos. El primer grupo lo constituyen los métodos lineales, donde se hace un gran esfuerzo de planeación antes de iniciar el desarrollo del proyecto o también llamados métodos tipo *Plan-driven* o *Big Design Upfront* (BDUF). El segundo grupo de métodos de desarrollo de software lo conforman métodos iterativos donde se parte de un prototipo funcional que se modifica en ciclos cortos de planeación y desarrollo, usualmente llamados *Sprints* o iteraciones. Estos métodos se consideran métodos ágiles de desarrollo de software. Los métodos de desarrollo de software, sin importar si se clasifican en el grupo de métodos tipo *Plan-driven* o *Big Design Upfront* o métodos ágiles, comparten características comunes (Dyck & Majchrzak, 2012). Dentro de los métodos ágiles de desarrollo de software existen prácticas que demuestran reducir el fracaso en los proyectos de software (Arisholm, Gallis, Dybå, & Sjøberg, 2007; Dingsoyr & Dyba, 2012; Falessi, Babar, Cantone, & Kruchten, 2009).

En este Trabajo Final de Maestría se propone la representación de prácticas de métodos ágiles de desarrollo de software dentro del núcleo de *Semat* (*Software Engineering Method and Theory*), para luego introducir estas prácticas dentro de un método de desarrollo tipo *Plan-driven* o *Big Design Upfront.* Al realizar esta representación, se espera comprobar que se pueden obtener los beneficios de un método ágil de desarrollo de software al aplicar sus prácticas, sin necesidad de cambiar el método de desarrollo de software que se viene empleando en un proyecto u organización. El motivo del uso de *Semat* sobre otras formas de representación de prácticas de ingeniera de software como Patrones de Proceso (sdPP; García Guzmán, Martín, Urbano, & Amescua, 2012) o *Software and Systems Process Engineering Metamodel* (SPEM; Shen, Hsueh, & Chu, 2011), es que, aunque se emplee sdPP o SPEM para realizar representaciones de prácticas de software, estas formas de representación no poseen elementos para mostrar

las prácticas de software en un lenguaje común, es decir, el lenguaje empleado para la representación de una práctica en sdPP o SPEM lo define la persona que está creando la representación. Esto limita la posibilidad de comparar prácticas de software y también limita la posibilidad de combinar elementos de diferentes prácticas de software.

*Semat* incluye un núcleo que define elementos universales para cualquier práctica de software y que emplean un lenguaje común. Este lenguaje común restringe los elementos y el vocabulario de quien desee representar una práctica de software en *Semat*. Sin embargo, la restricción del lenguaje no es una limitante sobre una práctica representada en *Semat*, sino una característica de *Semat* que permite comparar diferentes prácticas de software y compartir elementos entre dichas prácticas, lo que hace las prácticas reutilizables.

En este trabajo también se desarrolla un caso de estudio. Este caso de estudio toma el método de desarrollo de software descrito en el estándar DOD-STD-2167A del departamento de defensa de los Estados Unidos (Defense, 1988). Este caso de estudio busca ejemplificar de qué manera, modificando un método de desarrollo para mejorarlo adoptando prácticas externas al método, se pueden obtener proyectos de software exitosos, y cómo *Semat* se puede convertir en una herramienta para la gestión del proyecto.

# 1. Definición del problema

El problema que se aborda en este Trabajo Final de Maestría es el fracaso de los proyectos de software. En este Capítulo se muestra el porcentaje de proyectos de software que fracasan o muestran dificultades en su desarrollo. Adicionalmente, se muestra la motivación detrás de este Trabajo Final de Maestría, las áreas de estudio que procuran resolver el problema del fracaso de los proyectos de software y por qué se eligieron las prácticas que emplean los equipos de desarrollo de software para solucionar el problema.

## 1.1 Porcentaje de fracaso de los proyectos de software

El desarrollo de software es una actividad altamente susceptible al fracaso. El *CHAOS Report* que realizó el *Standish Group* en 2007 muestra que, de los proyectos iniciados en 2006: 35% de ellos fueron exitosos, 19% fueron fracasos (es decir, no lograron terminar) y 46% presentaron dificultades (Cerpa & Verner, 2009). Estas cifras son alarmantes, ya que el fracaso de un proyecto de software conlleva pérdidas tanto de tiempo como de dinero. Dos ejemplos de las pérdidas que puede ocasionar el fracaso de un proyecto de software son: un sistema de compras de *Ford Motor Company*, que abandonaron luego de su implementación, y otro sistema de compras de *McDonald's* que abandonaron durante su desarrollo; el primero tuvo un costo de aproximadamente 400 millones de dólares y en el segundo se invirtieron 170 millones de dólares antes de su cancelación (Charette, 2005).

## 1.2 Motivación

La motivación para realizar este trabajo se origina al observar que, entre los ingenieros de los diferentes proveedores de software de una entidad financiera, existe una creencia común; "que el desarrollo de software requiere un sacrificio de los programadores", es decir, trabajar hasta largas horas de la noche o la madrugada, laborar domingos y

festivos e inclusive posponer o interrumpir el tiempo destinado para las vacaciones. Sin embargo, incluso con dichos sacrificios, se observa que los proyectos se entregan tarde, con sobrecostos o excluyendo algunas de las funcionalidades que se planearon.

Luego de hablar sobre esta situación con ingenieros de uno de los proveedores del banco, quien tiene amplia experiencia en el desarrollo de software en países como Alemania, Reino Unido y Estados Unidos, se aclaró que dichos problemas son comunes y la forma de solucionarlos es mediante el cambio y la optimización de procesos, pero afirman que esto toma tiempo, ya que se debe realizar mediante prueba y error, pues no existe una solución general. La revisión de literatura que se hizo para este Trabajo Final de Maestría confirma dicha afirmación.

## 1.3 Trabajos previos

En la revisión de la literatura se identifican dos áreas de estudio que se pueden explorar para hallar una solución al fracaso de los proyectos de desarrollo de software, como son: los factores humanos y los métodos o procesos empleados en los proyectos (Cerpa & Verner, 2009). Debido a que los factores humanos requieren una preparación previa del investigador sobre la psiquis y el comportamiento humano, este Trabajo Final de Maestría se centra en las partes que integran los métodos o procesos aplicados en el desarrollo de software (que de ahora en adelante se denominan prácticas).

## 1.4 Problema a abordar

Dentro de las prácticas de desarrollo se encuentran las que emplean los equipos auto-administrados de desarrollo de software, que se seleccionaron para el desarrollo de este Trabajo Final de Maestría debido a que demuestran ser útiles para ayudar a proyectos de software en dificultades. Dichas prácticas, como el *Pair Programming*, se pueden encontrar dentro de los diferentes métodos de desarrollo de software que, aunque presentan diferentes orígenes, poseen prácticas comunes (Dyck & Majchrzak, 2012).

El problema que se aborda en este Trabajo Final de Maestría es el fracaso de los proyectos de software, específicamente los que se pueden disminuir mediante el uso de prácticas de desarrollo de software. En este Trabajo Final de Maestría se muestra cómo las prácticas de desarrollo de los equipos auto-administrados reducen el fracaso en los proyectos de software. Para aumentar la flexibilidad de dichas prácticas, se representan empleando el núcleo de *Semat* (*Software Engineering Method and Theory*), lo que

permite hacerlas comparables con otras prácticas y adaptarlas a prácticas ya existentes en las organizaciones. *Semat* busca cambiar la forma en que las personas piensan sobre los métodos y prácticas de software (Jacobson, Ng, McMahon, Spence, & Lidman, 2012) y se compone de elementos y bases teóricas diseñadas para representar cualquier proceso o actividad que ocurra dentro de un proyecto de software, lo cual facilita la obtención de los objetivos de este Trabajo.

# 2. Objetivos

## 2.1 Objetivo general

Representar las prácticas de los equipos de desarrollo auto-administrados en el núcleo de SEMAT (*Software Engineering Method and Theory*).

## 2.2 Objetivos específicos

- Identificar las prácticas comunes de las metodologías que emplean los equipos auto-administrados de desarrollo de software.
- Seleccionar las prácticas que emplean los equipos auto-administrados de desarrollo de software, que tienen más influencia en el éxito de los proyectos de software.
- Adaptar las prácticas de los equipos auto-administrados de desarrollo de software a los alfas del núcleo de Semat.
- Validar la correcta representación de las prácticas en el lenguaje de Semat mediante la creación de un caso de estudio.

# 3. Marco teórico

Para poner en contexto el tema de este Trabajo Final de Maestría, en este Capítulo se explica por qué fallan los proyectos de software, se exponen conceptos básicos sobre métodos agiles de desarrollo de software y se hace una introducción a *Semat* (*Software Engineering Method and Theory*).

## 3.1 Fracasos en los proyectos de software

Los estudios realizados sobre el fracaso de los proyectos de software agrupan las causas de los fracasos en factores humanos y factores atribuidos a procesos (Cerpa & Verner, 2009; Charette, 2005).

Los factores humanos identificados son:

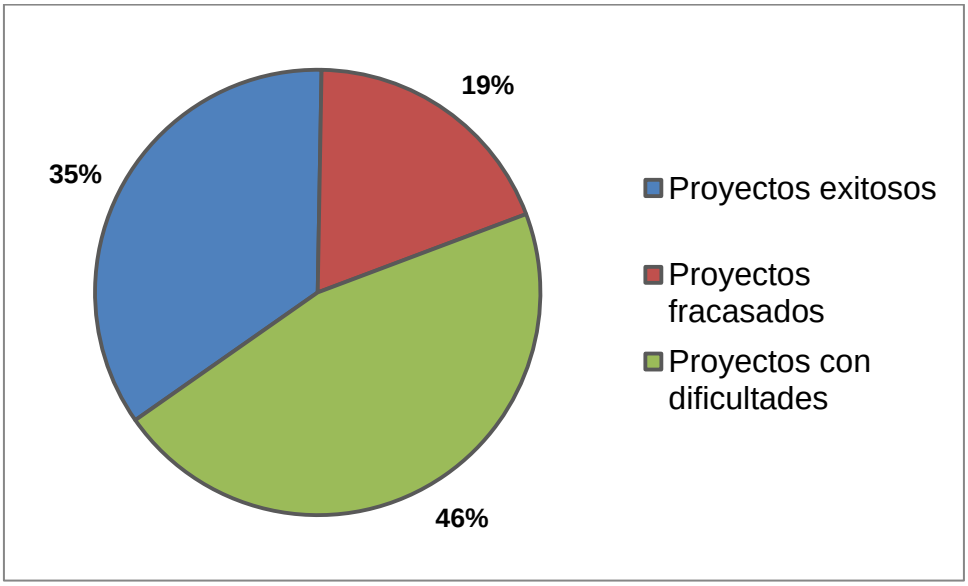
- La falta de recompensas por trabajar horas extra.
- Experiencias desagradables mientras se trabajaba en el proyecto.
- Un cronograma agresivo que afectó la motivación del equipo.
- El personal no era el adecuado para el proyecto.

Los factores atribuidos a procesos son:

- El proyecto se estimó mal.
- Los riesgos no se estimaron o controlaron.
- No se hizo revisión de las fases del proyecto.
- No se empleó el método adecuado.
- La meta se cambió durante el desarrollo del proyecto.
- No se concedió tiempo suficiente para el levantamiento de requisitos.

No todos los factores se presentan al mismo tiempo en un proyecto de software, pero la presencia de uno de estos factores puede poner en riesgo un proyecto de software. La presencia de varios de los factores presentados puede significar el fracaso de un proyecto. Para poder conocer en cifras cuántos proyectos de software fracasan, cuantos presentan retrasos y cuantos son exitosos, en 1994 el *Standish Group* creó el *CHAOS Report,* al ver que el fracaso de los proyectos de software era algo común. Este reporte se genera en periodos de dos a tres años; el CHAOS *Report* de 2005 mostró que 35% de los proyectos fueron exitosos, 19% fueron fracasos y no lograron terminar y 46% presentaron dificultades, como se muestra en la Figura 3-1.

**Figura 3-1:** Porcentaje de proyectos exitosos, fracasados o en dificultades (Cerpa & Verner, 2009).




Las cifras son más impactantes si se conocen las sumas de dinero invertidas en algunos de los proyectos que fracasaron, que se incluyen en la Tabla 3-1.

**Tabla 3-1**: Proyectos de software que fracasaron y originaron grandes pérdidas de dinero (Charette, 2005).

| Proyecto | Empresa | Perdidas en millones de dólares ($USD) |
|---|---|---|
| Sistema de compras | Ford *Motor Company* | 400 |
| Sistema de control aéreo | Administración Federal de Aviación de los Estados Unidos | 400 |
| Sistema de planeación de recursos (ERP) | Avis Europa | 54.5 |
| Sistema de compras | MacDonald's | 117 |
| Sistema de manejo de suministros | Kmart | 130 |

Todos los proyectos de software listados se cancelaron, es decir, el dinero y tiempo invertido se perdió. Dadas las pérdidas millonarias que puede representar el fracaso de un proyecto de software, investigadores y profesionales en el desarrollo de software buscan mitigar los factores tanto humanos como de procesos que afectan los proyectos de software. A continuación, se exponen prácticas de desarrollo de software y métodos de desarrollo ágiles que demuestran ser útiles para disminuir el fracaso en los proyectos de software.

## 3.2 Prácticas que disminuyen el riesgo de fracaso en proyectos de software y barreras que impiden su aplicación.

Las grandes pérdidas que pueden producir el fracaso de un proyecto de software llevan a realizar investigaciones que identifiquen las causas de estos fracasos y a investigar los proyectos exitosos, para conocer qué hicieron bien.

No existe un método, proceso o práctica que garantice el éxito de un proyecto de software (Brooks, 1986; Fraser & Mancl, 2008). Sin embargo, existen herramientas y prácticas que ayudan a reducir los factores de fracaso; entre estas están el uso de lenguajes de programación de alto nivel que no necesiten tiempo de compilación, el desarrollo incremental de aplicaciones, el uso de librerías existentes y cualquier práctica

que ayude a promover la colaboración y el trabajo en equipo (Brooks, 1986; Fraser & Mancl, 2008). Entre las herramientas se cuentan:

- Lenguajes de programación orientados a objetos.
- *Frameworks* de desarrollo de software.
- Automatización de pruebas.

Entre las prácticas se puede encontrar:

- *Pair Programming*.
- Reuniones de *Standup.*
- Asignación de tareas en *Kanban*.
- Construcción iterativa de software.
- Revisión de código.

El descubrimiento de estas prácticas se basa en evidencia anecdótica, es decir, en experiencias y observaciones de procesos o prácticas que realizan miembros de un proyecto (Dybå & Dingsøyr, 2008; Falessi *et al.*, 2009). Estas prácticas, cuando se aplican, pueden llegar a reducir el fracaso en los proyectos de software, pero el uso de estas prácticas no garantiza el éxito de un proyecto de software (Dybå & Dingsøyr, 2008). Existen barreras que dificultan la aplicación de estas prácticas dentro de un equipo de desarrollo. Estas barreras se clasifican en barreras organizacionales y barreras en el equipo de trabajo (Moe, Dingsøyr, & Dybå, 2009), según se presenta en la Tabla 3-2.

**Tabla 3-2**: Barreras que impiden la aplicación de prácticas desarrollo de software en las organizaciones (Moe *et al.*, 2009).

| Tipo de barrera | Nombre | Descripción |
|---|---|---|
| Barrera Organizacional | Recursos compartidos | Se refiere a la asignación de programadores a dos o más proyectos simultáneamente. Esto genera un dilema en el programador pues no sabe a cuál de los proyectos debe darle prioridad. |
| Barrera Organizacional | Control organizacional | Los documentos o productos de trabajo, que requiere la organización para controlar el avance del proyecto, pueden ocasionar que se emplee valioso tiempo en llenar reportes que se consideran innecesarios para el equipo de trabajo. |
| Barrera Organizacional | Cultura de la especialización | Se refiere a una actitud colectiva de querer especializarse en algo importante, ya sea para asegurar el empleo o para que la gerencia conozca fácilmente en que está trabajando cada programador. |
| Barrera en el equipo de trabajo | Compromiso individual | Se refiere a darle prioridad a los objetivos individuales sobre los objetivos comunes. Esto limita las oportunidades de interacción entre los miembros del equipo de trabajo y, por consiguiente, la oportunidad de aplicar prácticas como el *Pair Programming*. |
| Barrera en el equipo de trabajo | Liderazgo individual | Una de las prácticas que ayuda a reducir el fracaso en los proyectos de software es la toma de decisiones en conjunto, es decir, tomar decisiones donde todos los miembros del equipo pueden aportar su opinión. Sin embargo, puede dar la impresión que si las decisiones son tomadas por una sola persona, el proyecto avanzará más rápido. |
| Barrera en el equipo de trabajo | Dificultad para aprender | Se refiere a la dificultad que tienen los miembros del equipo de emplear las nuevas prácticas, volviendo a realizar las labores como se venían haciendo anteriormente. |

Algunas de prácticas que ayudan a reducir el fracaso en los proyectos de software se agrupan en métodos ágiles de desarrollo de software. A continuación, se describe en qué consisten los métodos ágiles de desarrollo de software y cuáles son sus orígenes.

## 3.3 Métodos ágiles de desarrollo de software.

En la búsqueda de disminuir el fracaso en los proyectos de software, se desarrollan procesos y prácticas que, en conjunto, buscan estandarizar la forma como se desarrolla el software. Este conjunto de procesos y prácticas se denominan métodos de desarrollo de software; entre los primeros métodos de desarrollo con mejor acogida entre las organizaciones, se encuentran el método de desarrollo en cascada y el método en espiral (B. W. Boehm, 1988). Estos métodos también se llaman *Plan-driven Methods* o *Big Design Upfront Methods* (B. Boehm, 2002; Cao, Mohan, Xu, & Ramesh, 2004) y se caracterizan por hacer una descripción detallada de los requisitos que debe cumplir el software resultante, evaluar los riesgos que se pueden presentar durante el desarrollo del proyecto y planear los recursos necesarios en tiempo y personal para poder culminar el proyecto. Esto se hace antes de iniciar formalmente el desarrollo del software, es decir, antes de escribir la primera línea de código.

Sin embargo, aunque se empleen métodos tipo *Plan-driven* o *Big Design Upfront*, los proyectos de software aún pueden fracasar (Charette, 2005). Debido a esto, a finales de los 90's surgieron métodos de desarrollo de software como *XP* y *Scrum*, que buscan crear software de forma iterativa, es decir, no hacer un gran plan del proyecto con antelación, sino ir construyendo el software con base en un prototipo funcional, donde en cada iteración se planea qué se va a adicionar a la última revisión del prototipo. Estos métodos se conocen como métodos ágiles de desarrollo de software. Para que un método de desarrollo de software se considere ágil debe cumplir doce principios ágiles, que se incluyen en la Tabla 3-3

**Tabla 3-3**: Doce principios ágiles que debe cumplir un método ágil de desarrollo de software (Williams, 2012).

| **Doce principios ágiles.** |
| --- |
| “Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor”. |
| “Aceptamos que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente”. |
| “Entregamos software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible”. |
| “Los responsables de negocio y los desarrolladores trabajamos juntos de forma cotidiana durante todo el proyecto”. |
| “Los proyectos se desarrollan en torno de individuos motivados. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo”. |
| “El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la conversación cara a cara”. |
| “El software funcionando es la medida principal de progreso”. |
| “Los procesos ágiles promueven el desarrollo sostenible. Los promotores, desarrolladores y usuarios debemos ser capaces de mantener un ritmo constante de forma indefinida”. |
| “La atención continua a la excelencia técnica y al buen diseño mejoran la agilidad”. |
| “La simplicidad, o el arte de maximizar la cantidad de trabajo no realizado, es esencial”. |
| “Las mejores arquitecturas, requisitos y diseños emergen de equipos auto-organizados”. |
| “A intervalos regulares, el equipo reflexiona sobre cómo ser más efectivo para a, continuación, ajustar y perfeccionar su comportamiento en consecuencia”. |

Aunque se demuestra éxito al emplear métodos ágiles de desarrollo de software, un proyecto puede fracasar aún empleando estos métodos (Dybå & Dingsøyr, 2008). También, se considera que los métodos ágiles de desarrollo de software no son adecuados para desarrollar grandes proyectos de software (B. Boehm, 2002; Cao *et al.*, 2004).

En el desarrollo de este Trabajo Final de Maestría se muestra cómo se pueden extraer prácticas de los métodos de desarrollo de software ágiles e inyectarlos en un método de desarrollo tipo *Plan-driven* o *Big Design Upfront,* como el método en cascada. Para esto se emplea *Semat* (*Software Engineering Method and Theory*).

## 3.4 *Semat* (*Software Engineering Method and Theory*)

Los métodos de desarrollo de software son herramientas que permiten gestionar los proyectos de software para poder asegurar que terminen satisfactoriamente. Estos métodos, si se comparan, tienen elementos comunes. Por ejemplo, entre los métodos ágiles de desarrollo de software como XP, Scrum y DSDM (*Dynamic System Development Methodology*) se encuentran elementos o prácticas comunes (Aalst, Mylopoulos, Rosemann, & Shaw, 2012); entre éstas se cuentan:

- *Pair Programnning.*
- *Refactoring* (Re-estructurar el código para reducir la complejidad y hacerlo más entendible).
- Entregas pequeñas y continuas del software en desarrollo.
- Revisiones de código.

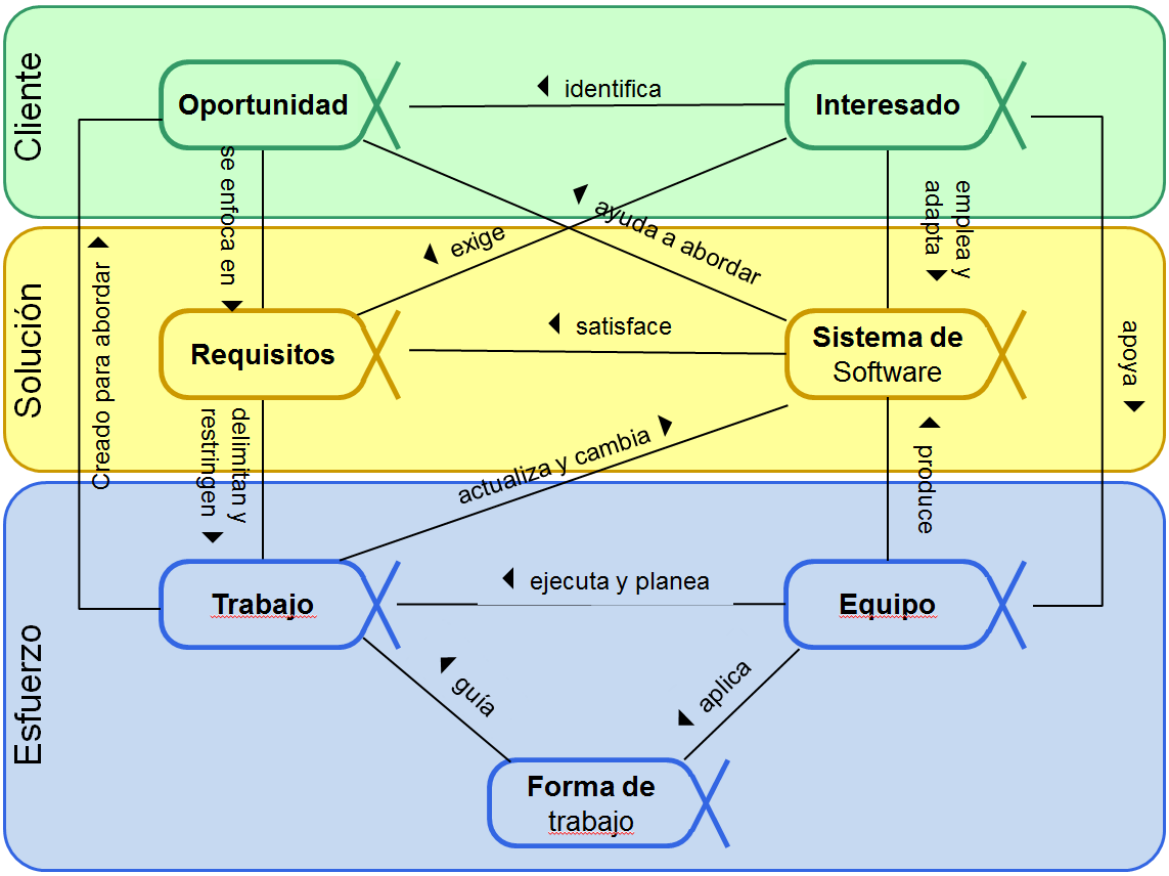
De igual forma, los métodos de desarrollo tipo *Plan-driven* o *Big Design Upfront* también poseen características comunes con los métodos ágiles de desarrollo, pues todos requieren requisitos, definen una etapa donde se debe crear el código y otra donde se deben probar las funcionalidades desarrolladas. El proceso que empleaba Lockheed en los 60’s y que sirvió de inspiración para los métodos de cascada y espiral, es un método iterativo de desarrollo de software (Royce, 1970). Por ello, si esta característica la hubieran tomado los métodos de cascada y espiral, sería otra característica común entre los métodos ágiles de desarrollo de software y los métodos tipo *Plan-driven* o *Big Design Upfront*.

Aunque estas similitudes son aparentes, es difícil hacer un comparativo de las prácticas que emplean los diferentes métodos de desarrollo de software para lograr los mismos objetivos. *Semat* (*Software Engineering Method and Theory*) nace como una iniciativa que busca atender diferentes problemas en la ingeniería de software. Según sus autores, estos problemas son (Jacobson *et al.*, 2012):

- "La prevalencia de bogas más típicas en la industria de la moda que en una disciplina ingenieril".
- "La carencia de una base teórica sonora y ampliamente aceptada".
- "La gran cantidad de métodos y variantes de métodos, con diferencias que poco se entienden y que se magnifican artificialmente".
- "La carencia de evaluación y validación experimentales y creíbles".
- "La separación entre la práctica industrial y la investigación académica".

Entre los problemas que busca solucionar *Semat* y que nos interesa en el desarrollo de este Trabajo Final de Maestría está "la gran cantidad de métodos y variantes de métodos, con diferencias que poco se entienden y que se magnifican artificialmente". Como se expuso en el párrafo anterior, los métodos de desarrollo de software poseen prácticas comunes, pero dichas prácticas son difíciles de comparar. *Semat* presenta un marco de trabajo que permite representar en gráficas, prácticas empleadas en el desarrollo de software; al hacer esto, se pueden comparar las características de las prácticas y su relación con otros elementos que componen un método de desarrollo de software, como las personas, procesos y productos de trabajo. Estos elementos se organizan en 'Alfas', que son contextos esenciales en desarrollo de software, es decir, contextos que existen en todos los métodos de desarrollo de software. Los alfas dentro del núcleo de *Semat* son; oportunidad, interesados, requisitos, sistema de software, trabajo, equipo y forma de trabajo. Jacobson *et al.* (2012) interconectan estos alfas en áreas de interés, como se muestra en la Figura 3-2:

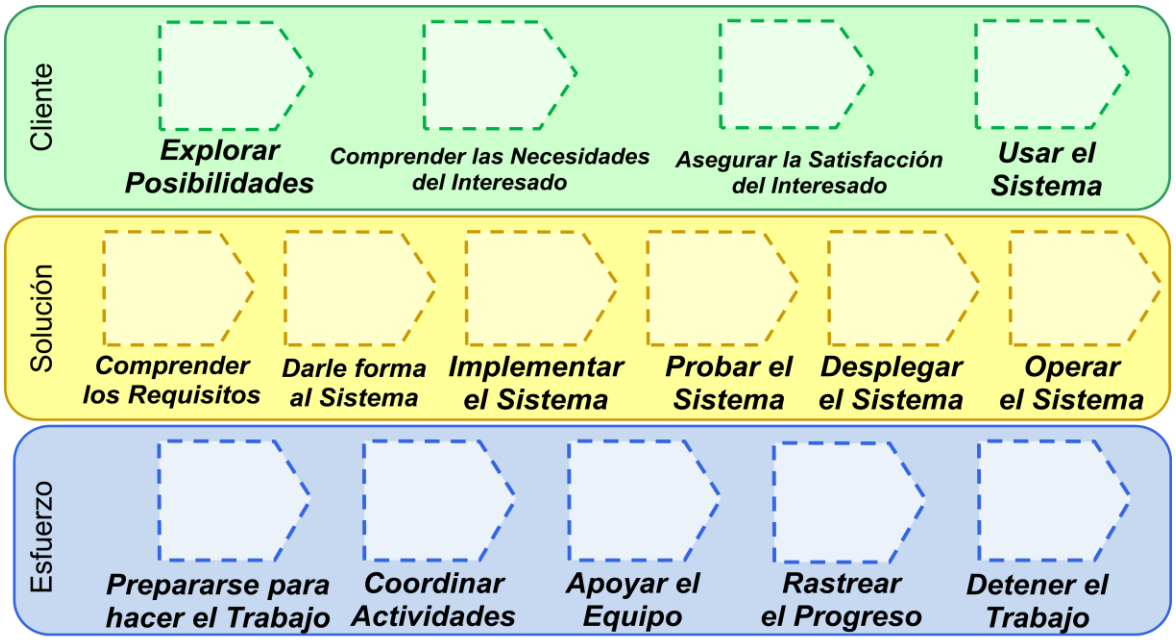
**Figura 3-2:** Alfas dentro del núcleo de *Semat* (Jacobson *et al.*, 2012).




Al organizar los contextos y elementos esenciales de la ingeniería de software dentro de estos alfas, se facilita la comparación de las prácticas entre los diferentes métodos de desarrollo de software y se posibilita el conocimiento de las interacciones de los elementos, el estado de estos elementos y las tareas que faltan por cumplir para avanzar el estado de dichos elementos, convirtiendo a *Semat* en una ayuda para la toma de decisiones dentro del proyecto de software (Jacobson *et al.*, 2012).

*Semat* también provee elementos que permiten representar las actividades que ocurren dentro de los alfas, estos son los espacios de actividad. La Figura 3-3 muestra los espacios de actividad dentro del núcleo de *Semat*. Al igual que los alfas, los espacios de actividad se agrupan en las áreas de interés de la ingeniería de software.

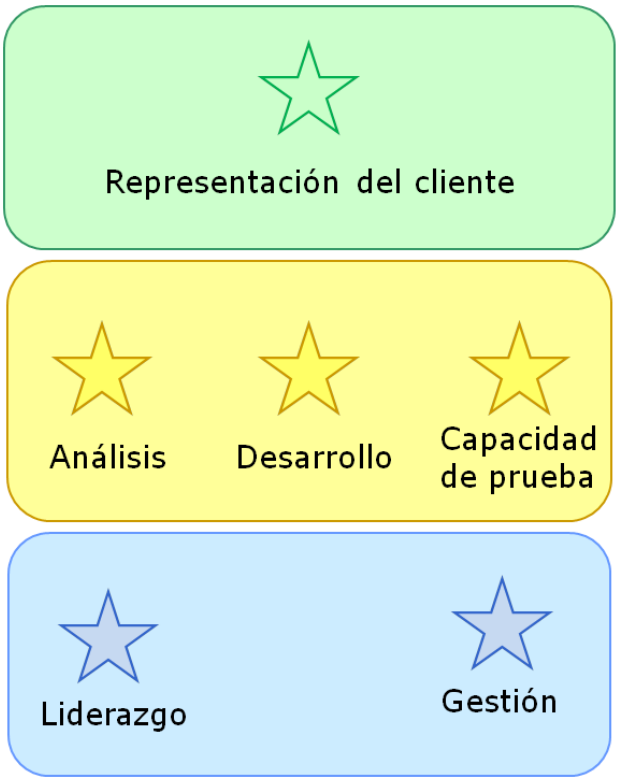
**Figura 3-3:** Espacios de actividad dentro del núcleo de *Semat*. (Jacobson *et al.*, 2012)




Un método de desarrollo de software puede agrupar sus actividades dentro los espacios de actividad. No es obligatorio emplear todos los espacios de actividad que se definen en *Semat,* por lo que pueden quedar espacios de actividad vacios, es decir, sin actividades relacionadas con ellos.

En *Semat* también se definen las competencias o habilidades necesarias para llevar a cabo una actividad dentro de un proceso de desarrollo de software. Al igual que los alfas y los espacios de actividad, las competencias se agrupan en los contextos esenciales de la ingeniería de software. La Figura 3-4 muestra las competencias dentro de cada una de las áreas de interés.

**Figura 3-4:** Competencias para el desarrollo de actividades.




En el desarrollo de este Trabajo se usa *Semat* para representar prácticas de software dentro de su núcleo.

# 4. Análisis y representación de prácticas de software en *Semat*

En este Capítulo se desarrollan los objetivos planteados en el Trabajo Final de Maestría. Se inicia por la selección de las prácticas de desarrollo de software que se representarán en el núcleo de *Semat*, explicando por qué se seleccionaron y su relevancia para el desarrollo de este Trabajo. Luego, se muestra la representación de estas prácticas dentro del núcleo de *Semat* y la manera de introducir las prácticas seleccionadas en un método de desarrollo que no emplea dichas prácticas. Por último, se desarrolla un caso de estudio donde se modifica el método de desarrollo descrito en el estándar DOD-STD-2167A del departamento de defensa de los Estados Unidos (Defense, 1988) para incluir prácticas de equipos auto-administrados.

## 4.1 Selección de prácticas de software

Los métodos de desarrollo de software se componen de prácticas empleadas para cada una de las etapas del proceso de desarrollo. Existen prácticas para la planeación de las actividades a realizar, para el desarrollo del código, para el aseguramiento de la calidad de software y para la interacción con los clientes o interesados en el desarrollo del software (B. W. Boehm, 1988; Dyck & Majchrzak, 2012).

Para el desarrollo de este Trabajo, se realizó una selección entre las prácticas que emplean los equipos auto-administrados de desarrollo de software, debido a que muestran ser útiles para reducir el fracaso en los proyectos de software (Dybå & Dingsøyr, 2008). De igual forma, se considera la importancia que los equipos de desarrollo de software le dan a los diferentes tipos de prácticas. Se evidencia que entre las más importantes están las relacionadas con la interacción entre los miembros de equipo y las que permiten alcanzar objetivos realistas al final de cada iteración o cada fase del proceso de desarrollo (Williams, 2012). Otra consideración para la selección de las prácticas de desarrollo de software a emplear en este trabajo, la constituyen las

barreras organizacionales que pueden existir dentro del equipo de desarrollo, es decir, limitantes que, por políticas o costumbres de la organización, pueden impedir que se empleen prácticas de software dentro del desarrollo del proyecto. Estas barreras, como el control organizacional, es decir, la aplicación de procesos de control que van en contra de procesos ágiles de planeación, exhiben dificultades para el avance de los proyectos de software (Moe *et al.*, 2009).

Teniendo en cuenta las posibles restricciones descritas en el párrafo anterior y la manera como dichas restricciones pueden dificultar la aplicación de prácticas de software en un proceso de desarrollo, se eligen dos prácticas que se pueden aplicar sin afectar los procesos existentes de la organización y, además, se encuentran dentro las prácticas consideradas de mayor importancia entre los equipos de desarrollo auto-administrados. Estas prácticas son el *Pair Programming* y la asignación de tareas en *Kanban*.

## 4.2 Representación en el núcleo de *Semat* de las prácticas seleccionadas

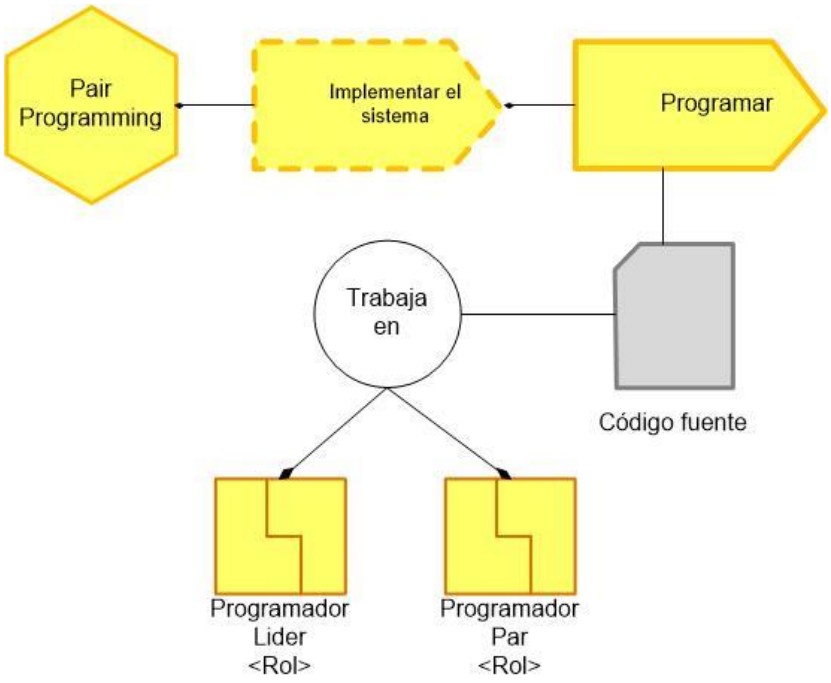
A continuación, se explica la importancia del *Pair Programming* y la asignación de tareas en *Kanban*. En esta Sección también se realiza la representación de las prácticas dentro del núcleo de *Semat*. Luego, estas representaciones se usan para mostrar cómo emplear *Semat* para introducir las prácticas en un método de desarrollo que no emplea dichas prácticas.

### *4.2.1 Pair Programming*

*Pair programming* se traduce como programación en pares y es una práctica de programación donde dos miembros de un equipo se reúnen en una sola estación de trabajo para escribir código. Las formas más comunes de realizar el trabajo en pares son dos: uno de los dos programadores dirige mientras el otro introduce el código empleando el teclado, o los dos programadores se turnan en el teclado cada vez que sientan que es necesario. Se espera que, al tener dos miembros del equipo trabajando en una misma funcionalidad o desarrollo de un requisito o historia de usuario, el código resultante sea mejor que si el código lo produjera un solo programador. En la práctica, se demuestra que las ventajas del *pair programming* varían según la complejidad del código y la experiencia de los programadores (Arisholm *et al.*, 2007). En el estudio que evidencia

esta variación en la efectividad del *pair programming* se recomienda que, si la complejidad de la tarea a realizar es alta, se debe realizar entre un programador *Senior* y un programador *Junior* (Arisholm *et al.*, 2007). Una revisión a varios estudios empíricos sobre desarrollo de software empleando métodos ágiles muestra que la aplicación de *pair programming* permite que el conocimiento se distribuya entre los miembros del equipo. En otras palabras, que el nivel de los programadores se tienda a nivelar hacia el nivel de los programadores con más experiencia (Dybå & Dingsøyr, 2008). Esta evidencia se menciona para mostrar la importancia del *pair programming* dentro de un proceso de desarrollo de software. En la Figura 4-1 se muestra la representación propuesta de la práctica *pair programming* en el núcleo de *Semat*.

**Figura 4-1:** Representación de *pair programming* en *Semat*.



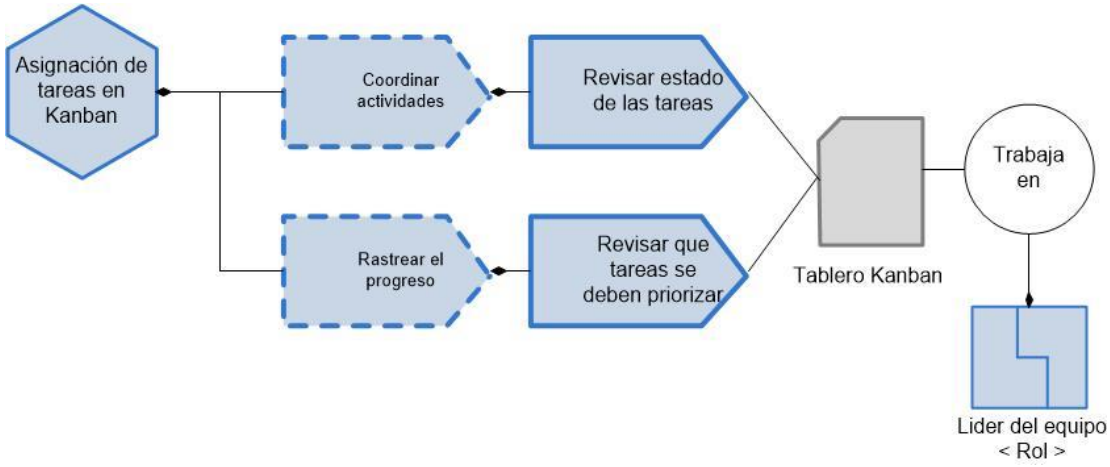

## 4.2.2 Asignación de tareas en *Kanban*

*Kanban* es una palabra utilizada en japonés para describir un tablero. Toyota, en su proceso de producción, utilizó en un inicio la asignación de tareas en *Kanban*. El concepto se basa en tener un tablero donde se indique, mediante tarjetas, qué partes se

están produciendo actualmente para poder minimizar el inventario, solo producir las partes que en realidad se necesitan y evitar cuellos de botella en la producción (Ono, 1988). De igual forma, en el desarrollo de software la asignación de tareas en *Kanban* permite saber en qué está trabajando cada programador, qué tareas se completaron y cuales aún no inician. Esto permite evitar cuellos de botella y, principalmente, conocer la capacidad del equipo, es decir, cuál es el máximo de tareas que el equipo puede trabajar simultáneamente (Ikonen, Pirinen, Fagerholm, Kettunen, & Abrahamsson, 2011).

*Kanban* es una práctica que inició en un proceso de manufactura y se adaptó al desarrollo de software. El estudio de Ikonen *et al.* muestra que no existe una forma única de aplicar *Kanban* a un proceso de desarrollo de software. Es por esto que surgen métodos de desarrollo de software como *Scrumban* (Nikitina, Kajko-Mattsson, & Strale, 2012), que busca estandarizar el uso de *Kanban* en el método de desarrollo *Scrum*. Uno de los objetivos de este Trabajo es la representación de la asignación de tareas en *Kanban* en el núcleo de *Semat*, con el propósito de poder mostrar que, mediante *Semat*, se pueden incluir prácticas dentro de los diferentes métodos de desarrollo, sin la necesidad de crear nuevos métodos por cada inclusión de una práctica a un método existente, evitando que se creen nuevos métodos como *Waterfallban*, *XPban*, *Spiralban*, entre otros. En la Figura 4-2 se muestra la representación propuesta de la asignación de tareas en *Kanban* en el núcleo de *Semat*.

**Figura 4-2:** Representación de asignación de tareas en *Kanban*, en *Semat*.

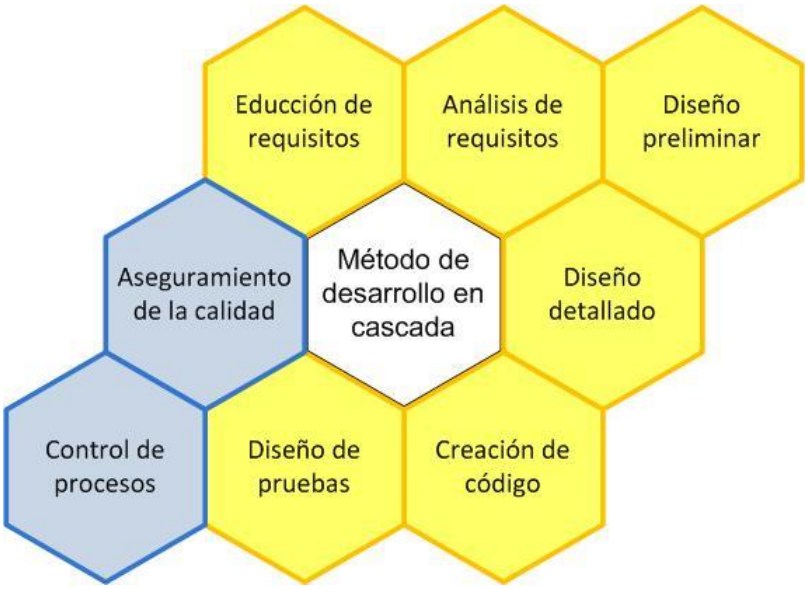



## 4.3 Introduciendo prácticas ágiles en el método de desarrollo en cascada y su representación en *Semat*

Para ejemplificar como *Semat* permite extender un método de desarrollo de software, añadiendo prácticas provenientes de otro método de desarrollo, en este Capítulo se toma el método de desarrollo en cascada y se añaden las prácticas de *Pair Programming* y asignación de tareas en *Kanban*. Estas son prácticas empleadas en métodos ágiles y la ejemplificación permite mostrar que también se pueden emplear en métodos tipo *Plan-driven* o *Big Design Upfront*.

Inicialmente, en la Figura 4-3 se presentan las prácticas generales del método en cascada.

**Figura 4-3:** Prácticas generales del método en cascada.




En la Figura 4-4 se detallan las actividades que ocurren dentro del alfa 'sistema de software' y en la Figura 4-5 se detallan las actividades que ocurren dentro de los espacios de actividades de 'Rastrear el progreso' y 'Apoyar el equipo' para un método tradicional de desarrollo de software en cascada.

**Figura 4-4:** Actividades que ocurren dentro del alfa de 'sistema de software' para un método tradicional de desarrollo de software en cascada.

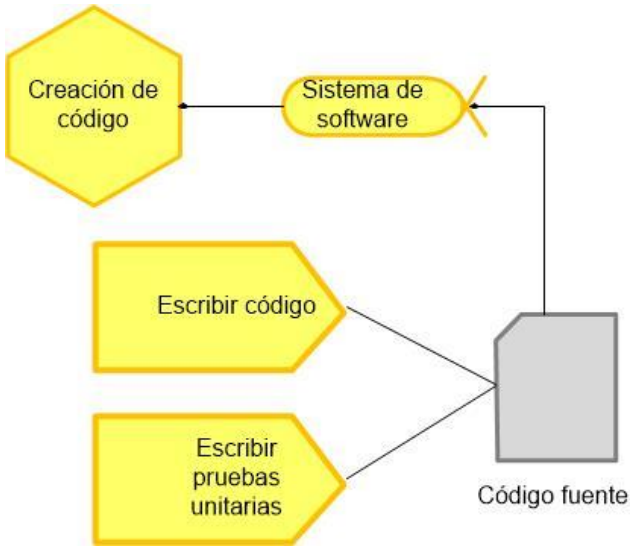
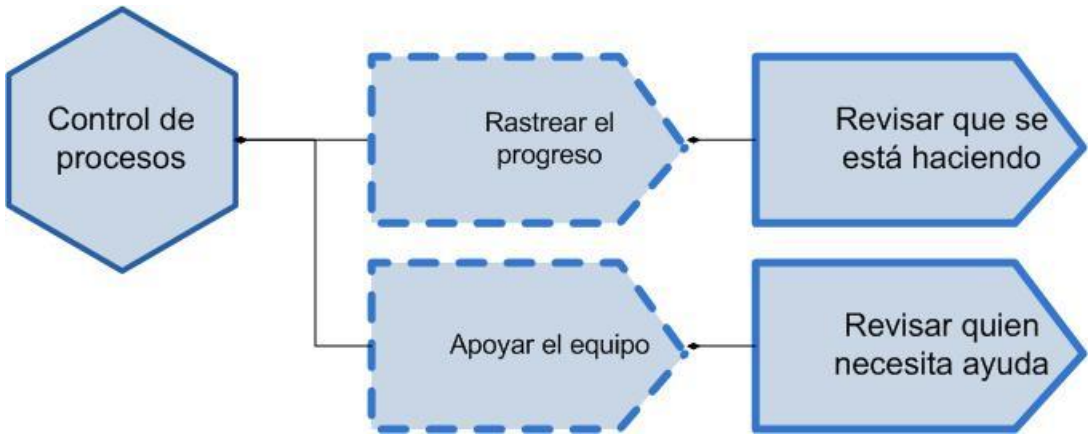
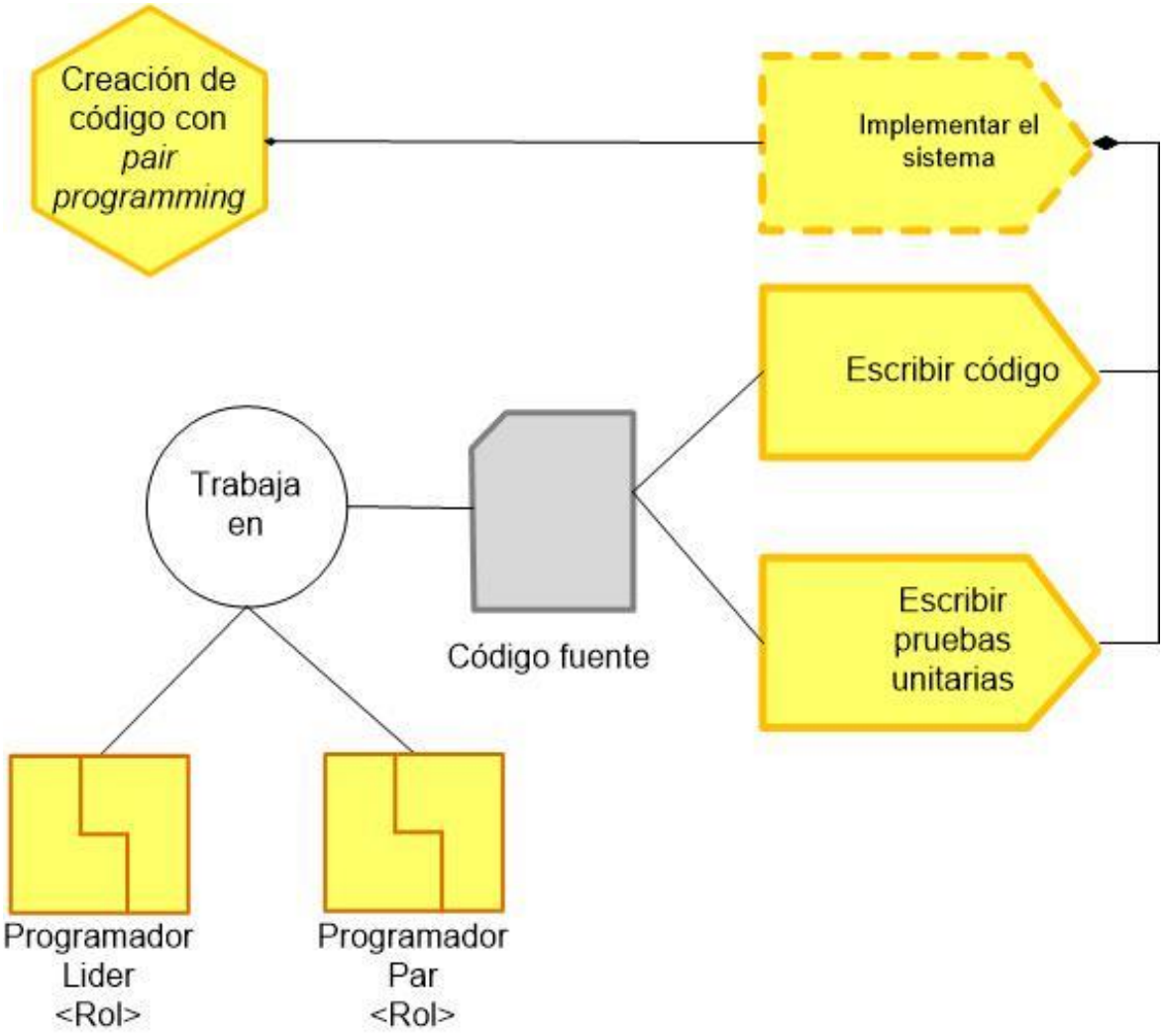


**Figura 4-5:** Actividades que ocurren dentro espacios de actividades de ‘Rastrear el progreso’ y ‘Apoyar el equipo’ para un método tradicional de desarrollo de software en cascada.



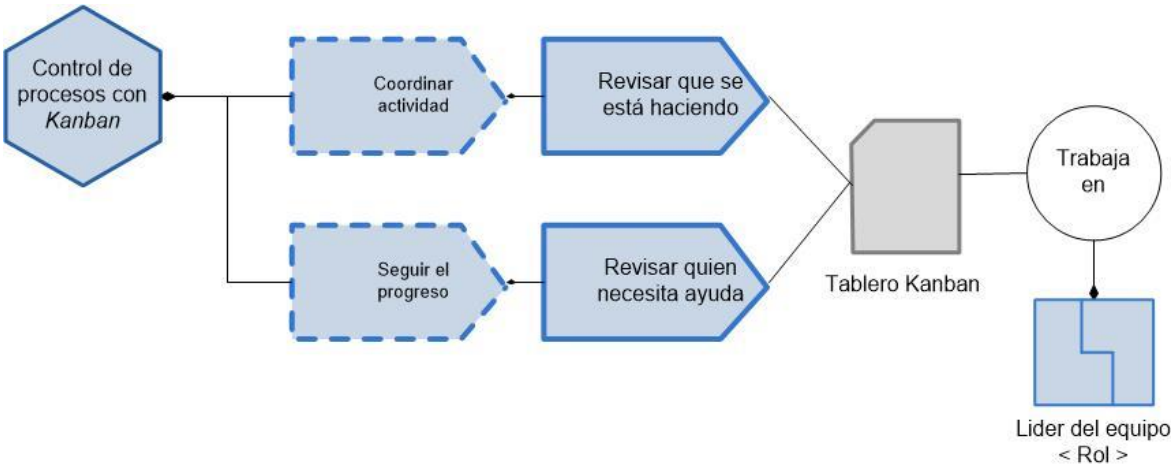

En la figura 4-6 se muestra la inclusión de la práctica *pair programming* dentro de la actividad descrita anteriormente en la figura 4-4. Con la inclusión de *pair programming* se consigue mejorar la actividad, incluyendo una práctica que aumenta la calidad del código y que permite redistribuir el conocimiento entre los miembros de equipo de trabajo (Arisholm *et al.*, 2007).

**Figura 4-6:** *Pair programming* incluido en el alfa de 'sistema de software' para un método tradicional de desarrollo de software en cascada.




En la Figura 4-7 se muestra la inclusión de la práctica asignación de tareas en *Kanban* en los espacios de trabajo de 'Coordinar actividades' y 'Seguir el proceso' para un método tradicional de desarrollo de software en cascada. La inclusión de la asignación de tareas en *Kanban* permite tener un mejor conocimiento de en qué estado se encuentran las funcionalidades que están en desarrollo y quiénes las están desarrollando. Lo anterior permite tomar mejores decisiones sobre cuáles funcionalidades, requisitos o historias de usuario deben tener prioridad y cuáles miembros del equipo pueden necesitar ayuda para completar la funcionalidad que están trabajando actualmente.

**Figura 4-7:** Asignación de tareas en *Kanban* incluida en los espacios de trabajo de 'Coordinar actividades' y 'Seguir el proceso' para un método tradicional de desarrollo de software en cascada.




Como se muestra en los ejemplos, las actividades del método de desarrollo de software mejoraron y no se cambiaron con actividades nuevas, lo que permite incluir prácticas ágiles sin necesidad de migrar de método de desarrollo de software.

## 4.4 Caso de estudio: Prácticas de los equipos auto-administrados adaptadas al estándar DOD-STD-2167A para desarrollo de software

Para mostrar un ejemplo de cómo *Semat* permite extender métodos de desarrollo existentes adicionando prácticas de otros métodos, a continuación se muestra un caso de estudio donde se toman prácticas de los equipos auto-administrados de desarrollo de software y de adaptan al estándar DOD-STD-2167A. El estándar DOD-STD-2167A lo desarrolló y utilizó el ejército de los Estados Unidos para el desarrollo de software.

### 4.4.1 Descripción del estándar DOD-STD-2167A

El estándar DOD-STD-2167A describe la forma en que los contratistas del ejército de los Estados Unidos debían desarrollar el software que les era encargado. El estándar

también es base para del estándar ISO/IEEE 12207 (Sheard, 2001). Para este caso de estudio se emplea el estándar DOD-STD-2167A debido a que es más específico en las actividades que debe realizar el equipo de trabajo en cada una de las etapas del ciclo de vida del software, en comparación con estándares más recientes y similares como el ISO/IEEE 12207.

El estándar DOD-STD-2167A describe las siguientes etapas en el ciclo de software:

- Análisis de los requisitos del sistema o software.
- Diseño preliminar.
- Diseño detallado.
- Codificación, pruebas unitarias y pruebas de integración.

Adicional a las etapas listadas, el estándar solicita que se tenga un proceso de revisión de cada una de las etapas, donde se verifica que:

- Los requisitos tengan el suficiente nivel de detalle.
- El diseño de los casos de prueba cubran todos los requisitos planeados para el software.
- El código creado cumpla con los estándares de código elegidos para el proyecto.
- Existan pruebas unitarias para el código creado.
- Existan pruebas de integración para los requisitos planeados.

El estándar no especifica cómo se debe realizar el proceso de revisión. En este caso de estudio se modifican prácticas de los equipos auto-administrados, adaptándolos al estándar DOD-STD-2167A para cumplir con los objetivos impuestos por el estándar.

### 4.4.2 Desarrollo del caso de estudio

Para que “los requisitos tengan el suficiente nivel de detalle” y “el diseño de los casos de prueba cubran todos los requisitos planeados para el software”, se propone la asignación de tareas en *Kanban*. Esta práctica permite conocer la relación y el estado de desarrollo de los requisitos y los casos de prueba. Adicionalmente, permite saber cuál de los requisitos puede ser un cuello de botella para el desarrollo de un caso de prueba.

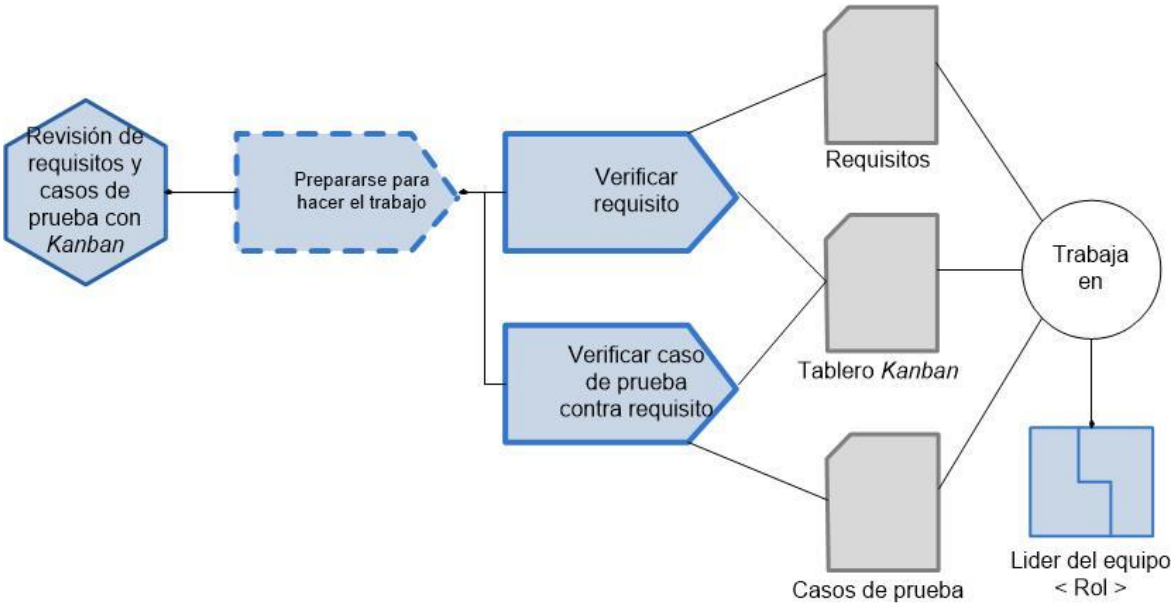
Para que “el código creado cumpla con los estándares de código elegidos para el proyecto”, se propone emplear *Pair Programming*. Esta práctica permite mejorar la calidad del código y, adicionalmente, otro miembro del equipo de trabajo hace una revisión de si se están cumpliendo los estándares de código.

Para que “existan pruebas unitarias para el código creado” y para que “existan pruebas de integración para los requisitos planeados”, se propone emplear desarrollo orientado a pruebas (*Test Driven Development*), debido a que esta práctica obliga a crear las pruebas unitarias antes de escribir el código que implementará la funcionalidad o requisito (Janzen & Saiedian, 2005).

Para la representación de las prácticas se emplea *Semat*, y se incluyen en las representaciones los productos de trabajo exigidos por el estándar.

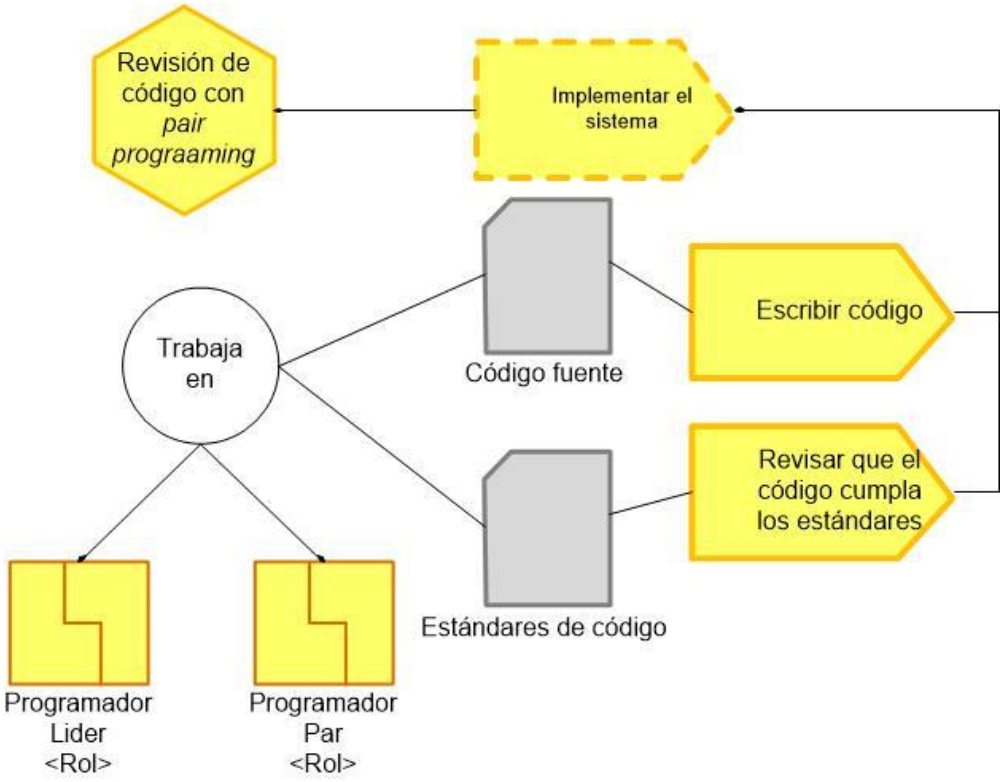
En la Figura 4-8 se muestra la asignación de tareas en *Kanban* modificada para cumplir el estándar DOD-STD-2167A para las revisión de requisitos y casos de prueba. Mediante la aplicación de una práctica de los equipos auto-administrados se logran los objetivos que exige el estándar en relación con que “los requisitos tengan el suficiente nivel de detalle” y “el diseño de los casos de prueba cubra todos los requisitos planeados para el software”.

**Figura 4-8:** Revisión de requisitos y casos de prueba con *Kanban*.



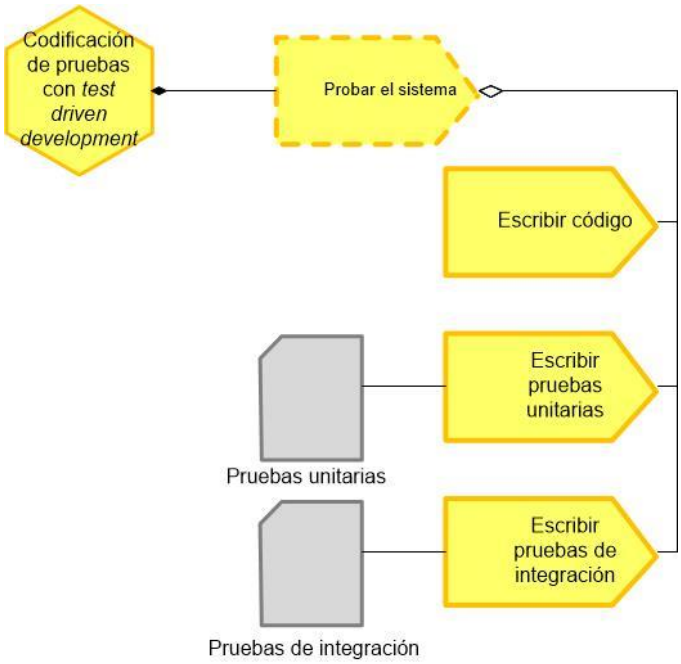

En la Figura 4-9 se muestra como mediante la aplicación de la práctica de *Pair Programming* como mecanismo de revisión de código fuente, se asegura que el código cumpla con los estándares de código elegido para el proyecto.

**Figura 4-9:** Revisión de código con *pair programming*.




En la Figura 4-10 se muestra la aplicación de la práctica de desarrollo orientado a pruebas (*Test Driven Development*) como mecanismo para la codificación de pruebas unitarias y de integración. Esta práctica permite obtener las pruebas unitarias y pruebas de integración exigidas por el estándar al mismo tiempo que se desarrolla el software.

**Figura 4-10:** Codificación de pruebas con *test driven development*.




## 4.4.3 Conclusiones sobre el caso de estudio.

En el caso de estudio se concluye que las prácticas de los equipos auto-administrados y las prácticas ágiles de desarrollo de software se pueden adaptar a métodos de desarrollo no ágiles como el descrito en el estándar DOD-STD-2167A. El estándar no define concretamente las actividades que se deben hacer para la revisión de los productos de trabajo en cada una de las etapas del ciclo de vida del software. Mediante el uso de *Semat* se pudieron representar las adaptaciones de prácticas ágiles de desarrollo de software y cómo se pueden utilizar para hacer las revisiones de los productos de trabajo, como lo exige el estándar DOD-STD-2167A. Al hacer este caso de estudio, se puede considerar hacer un ejercicio similar con un estándar más reciente como el ISO/IEEE 12207; este estándar lo deben seguir las empresas de desarrollo que deseen certificarse en ISO 12207. *Semat* puede servir como herramienta para una conocer si una empresa cumple con las prácticas necesarias para certificarse en ISO 12207 o conocer qué prácticas deben añadir o modificar para poder alcanzar la certificación.

# 5. Conclusiones

En este Trabajo Final de Maestría se hizo una revisión de las causas de los fracasos de los proyectos de software, se comprobó que prácticas de desarrollo de software como *pair programming* y asignación de tareas en *Kanban* pueden ayudar a reducir el fracaso en los proyectos de software y se planteó cómo mediante el uso de *Semat* se pueden incluir dichas prácticas dentro de las organizaciones sin cambiar de método de desarrollo de software que emplea la organización. Se analizó un caso de estudio donde se mejoró un estándar de desarrollo de software mediante la adaptación de prácticas ágiles de desarrollo que emplean los equipos auto-administrados. Queda por demostrar si, al poner en práctica el estándar mejorado, una organización o equipo de trabajo lo acepta sin inconvenientes. Al hacer el caso de estudio, se considera que es posible hacer un ejercicio similar con un estándar más reciente como el ISO/IEEE 12207. En ese caso *Semat* podría servir para conocer qué prácticas debe adicionar o cambiar una organización para poder lograr la certificación ISO 12207.

La aplicación de *Semat* se limitó a la modificación de actividades de un método de desarrollo de software dentro de los alfas de ‘Trabajo’ y ‘Sistema de software’. Aún quedaría mostrar su aplicación en otros alfas y en la gestión completa de un proyecto de software.

# Bibliografía


Aalst, W., Mylopoulos, J., Rosemann, M., & Shaw, M. J. (2012). *Agile Processes in Software Engineering and Extreme Programming*. (C. Wohlin, Ed.) (Vol. 111). Berlin, Heidelberg: Springer Berlin Heidelberg. http://doi.org/10.1007/978-3-642-30350-0

Arisholm, E., Gallis, H., Dybå, T., & Sjøberg, D. I. K. (2007). Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Transactions on Software Engineering*, *33*(2), 65–86. http://doi.org/10.1109/TSE.2007.17

Boehm, B. (2002). Get ready for agile methods, with care. *Computer*, *35*(1), 64–69. http://doi.org/10.1109/2.976920

Boehm, B. W. (1988). A Spiral Model of Software Development ans Enhancement. *Computer*, *21*(5), 61–72. http://doi.org/10.1109/2.59

Brooks, F. P. J. (1986). No silver bullet-essence and accidents of software engineering. *Proceedings of the IFIP Tenth World Computing Conference*, 1069–1076. http://doi.org/10.1109/MC.1987.1663532

Cao, L., Mohan, K., Xu, P. X. P., & Ramesh, B. (2004). How Extreme does Extreme Programming Have to be? Adapting XP Practices to Large-scale Projects. *37th Hawaii International Conference on System Sciences (HICSS)*, *00*(C), 1–10.

Cerpa, N., & Verner, J. (2009). Why did your project fail? *Communications of the ACM*, *52*(12), 130–134. http://doi.org/10.1145/1610252.1610286

Charette, B. R. N. (2005). Why software fails. *IEEE Spectrum*, *42*(9), 42–49. http://doi.org/10.1109/MSPEC.2005.1502528

Defense, D. of. (1988). *Department of Defense Standard 2167A*. Department of Defense. Retrieved from http://www.product-lifecycle-management.com/download/DOD-STD-2167A.pdf

Dingsoyr, T., & Dyba, T. (2012). Team effectiveness in software development: Human and cooperative aspects in team effectiveness models and priorities for future studies. *Cooperative and Human Aspects of …*, (7465), 27–29. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6223016

Dybå, T., & Dingsøyr, T. (2008). Empirical studies of agile software development: A systematic review. *Information and Software Technology*, *50*(9-10), 833–859. http://doi.org/10.1016/j.infsof.2008.01.006

Dyck, S., & Majchrzak, T. A. (2012). Identifying Common Characteristics in Fundamental, Integrated, and Agile Software Development Methodologies. *2012 45th Hawaii International Conference on System Sciences*, 5299–5308. http://doi.org/10.1109/HICSS.2012.310

Falessi, D., Babar, M. A., Cantone, G., & Kruchten, P. (2009). Applying empirical software engineering to software architecture: challenges and lessons learned. *Empirical Software Engineering*, *15*(3), 250–276. http://doi.org/10.1007/s10664-009-9121-0

Fraser, S., & Mancl, D. (2008). No silver bullet: Software engineering reloaded. *IEEE Software*, *25*(May 1972), 91–94. http://doi.org/10.1109/MS.2008.14

García Guzmán, J., Martín, D., Urbano, J., & Amescua, A. (2012). Practical experiences in modelling software engineering practices: The project patterns approach. *Software Quality Journal*, *21*(2), 325–354. http://doi.org/10.1007/s11219-012-9177-8

Ikonen, M., Pirinen, E., Fagerholm, F., Kettunen, P., & Abrahamsson, P. (2011). On the Impact of Kanban on Software Project Work: An Empirical Case Study Investigation. *2011 16th IEEE International Conference on Engineering of Complex Computer Systems*, 305–314. http://doi.org/10.1109/ICECCS.2011.37

Jacobson, I., Ng, P.-W., McMahon, P. E., Spence, I., & Lidman, S. (2012). The essence of software engineering. *Communications of the ACM*, *55*(12), 42. http://doi.org/10.1145/2380656.2380670

Janzen, D., & Saiedian, H. (2005). Test-driven development concepts, taxonomy, and future direction. *Computer*, *38*(9), 43–50. http://doi.org/10.1109/MC.2005.314

Moe, N. B., Dingsøyr, T., & Dybå, T. (2009). Overcoming Barriers to Self-Management in Software Teams. *IEEE Software*, *26*(6), 20–26. http://doi.org/10.1109/MS.2009.182

Nikitina, N., Kajko-Mattsson, M., & Strale, M. (2012). From scrum to scrumban: A case study of a process transition. *Software and System Process (ICSSP), 2012 International Conference on*, 140–149. http://doi.org/10.1109/ICSSP.2012.6225959

Ono, T. (1988). *Toyota production system: beyond large-scale production*. Productivity Press.

Royce, W. W. (1970). Managing the Development of Large Software Systems. *IEEE WESCON (Reprinted in Proceedings Ninth International Conference on Software Engineering.)*, (August), 1–9.

Sheard, S. a. (2001). Evolution of the frameworks quagmire. *Computer*, *34*(7), 96–98.

http://doi.org/10.1109/2.933516

Shen, W.-H., Hsueh, N.-L., & Chu, P.-H. (2011). Measurement-based Software Process Modeling. *Journal of Software Engineering*, *5*(1), 20–37. http://doi.org/10.3923/jse.2011.20.37

Williams, L. (2012). What agile teams think of agile principles. *Communications of the ACM*, *55*(4), 71. http://doi.org/10.1145/2133806.2133823