PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
ESCUELA DE INGENIERIA

PRÁCTICAS, METODOLOGÍAS Y HERRAMIENTAS PARA LA GESTIÓN Y EJECUCIÓN DE PROYECTOS DE DESARROLLO DE SOFTWARE QUE UTILICEN EQUIPOS DE TRABAJO GEOGRÁFICAMENTE DISTRIBUIDOS

FRANCISCO JAVIER RAMÍREZ MORALES

Tesis para optar al grado de
Magister en Ciencias de la Ingeniería

Profesor Supervisor:
DAVID FULLER

Santiago de Chile, enero, 2008

PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
ESCUELA DE INGENIERIA

PRÁCTICAS, METODOLOGÍAS Y HERRAMIENTAS PARA LA GESTIÓN Y EJECUCIÓN DE PROYECTOS DE DESARROLLO DE SOFTWARE QUE UTILICEN EQUIPOS DE TRABAJO GEOGRÁFICAMENTE DISTRIBUIDOS

FRANCISCO JAVIER RAMÍREZ MORALES

Tesis presentada a la Comisión integrada por los profesores:

DAVID FULLER

YADRAN ETEROVIC

LUÍS GUERRERO

SERGIO MATURANA

Para completar las exigencias del grado de
Magíster en Ciencias de la Ingeniería

Santiago de Chile, enero, 2008

A mi familia, novia y amigos, su apoyo fue fundamental.

AGRADECIMIENTOS

Agradezco los comentarios, concejos y apoyo de mis profesores Rosa Alarcón, David Fuller, Yadran Eterovic, Luís Guerrero y Sergio Maturana. También estoy muy agradecido de todo el apoyo que recibí de mis seres queridos en especial de Mónica Morales (madre), Claudia Forni (novia), Pedro Ramírez (padre), Pablo Ramírez (hermano), Humberto Urzúa (tío), Bárbara Vallejos (amiga), Mario Zúñiga (amigo) y Miguel Ángel Suarez (amigo). Tampoco puedo dejar de mencionar a mis mejores amigos: Paula Abarca, María Dolores Yáñez, Ignacio Berrios, Patricio Pérez-Cotapos, Andrés Crisosto, Felipe Revello, Eduardo Carvajal, Pablo Carrera, Diego Rivera, Julián Loma-Osorio, Juan Sebastián Espejo y tantos más, ellos siempre estuvieron para brindarme toda su alegría y darme fuerzas para seguir con este gran desafío.

INDICE GENEAL

Pág.

INDICE DE TABLAS

INDICE DE FIGURAS

RESUMEN

Durante la última década, la elaboración de software a través de equipos de trabajo geográficamente dispersos -también llamado ***desarrollo distribuido de software***- ha tomado notable interés por organizaciones que desean aumentar su competitividad local y global, dado los beneficios potenciales que presenta esta forma de elaboración como por ejemplo: poder atender a clientes con alta disposición de pago ubicados en otras regiones geográficas, capacidad para disminuir los tiempos de fabricación, reducir los costos en recursos humanos y poder reclutar personal altamente calificado. No obstante, el ***desarrollo distribuido de software*** no se ha masificado por sus altas barreras de gestión y ejecución. Es así como, el objetivo central de esta investigación consiste en identificar y abordar los principales problemas que impidan lograr exitosamente un ***desarrollo distribuido de software***. Para lograr el objetivo planteado se seleccionaron las prácticas básicas para elaborar software de calidad; se realizó un análisis que identificó las características de desarrollo más ventajosas para la realidad chilena; se seleccionaron y analizaron los principios y prácticas aplicadas en los proyectos de software más exitosos; y se analizó el ***estado del arte del desarrollo distribuido de software***. En función de los puntos anteriores, se diseño un ***marco de trabajo*** que define principios, prácticas, metodologías y herramientas que, según las argumentaciones expuestas, son las más apropiadas para utilizarlas en un escenario distribuido. Finalmente, se modeló un ***desarrollo distribuido de software*** en el cual se aplica el ***marco de trabajo*** diseñado.

De la investigación se concluye que, en base a la experiencia de proyectos exitosos, es posible que los desarrollos distribuidos sean factibles si se atienden las variables principales para permitir una adecuada ***comunicación***, ***coordinación***, ***control***, ***trazabilidad*** y ***visibilidad*** de los proyectos. Lo anterior, apoyándose en un conjunto adecuado de ***principios***, ***prácticas***, ***técnicas***, ***metodologías*** y ***herramientas***.

Palabras Claves: Desarrollo Distribuido de Software, Desarrollo Disperso de Software, Offshoring, Exportación de Servicios, Equipos Virtuales, Ingeniería de Software.

ABSTRACT

During the last decade, software development across geographically dispersed work teams - also called distributed software development -has taken considerable interest by organizations wishing to enhance their competitiveness locally and globally, given the potential benefits which this form of development presents, such as: serving customers with high provision for payment located in other geographic regions, ability to reduce manufacturing time, reducing costs in human resources and being able to recruit highly qualified staff. However, distributed software development has not been able to spread widely due to its high management and execution barriers. Thus, the central objective of this research is to identify and address major problems that prevent achieving successfully distributed software development. To achieve the above goal, practices to develop basic quality software were selected; an analysis that identified more advantageous developing characteristics for the Chilean reality was carried out; the principles and practices applied in the most successful software projects were selected and analyzed; and the state of the art of distributed software development was analyzed. Based on the above points, a framework was designed. It defines principles, practices, methodologies and tools that, according to the arguments stated, are most appropriate for use in a distributed scenario. Finally, a distributed software development was modeled in which the framework designed is applied.

The investigation concluded that, based on the experience of successful projects, it is possible that distributed developments are feasible if the main variables to enable proper communication, coordination, monitoring, traceability and visibility of projects are considered. The above, relying on a suitable set of principles, practices, techniques, methodologies and tools.

Keywords: Distributed Software Development, Dispersed Software Development, Offshoring, Virtual Teams, Global Software Development, Software engineering.

1. INTRODUCCIÓN

1.1 Motivación

Históricamente la demanda por software ha sobrepasado la oferta (Kotlarsky, 2005; Procópio, 2005). De la misma forma, cada vez más las personas y las organizaciones requieren más software en sus procesos y artefactos (a través del software embebido). Lo anterior puede corroborarse mediante el crecimiento mundial de la industria del software, según el estudio Development of Global Software Industry Annual Report 2006-2007, elaborado por la consultora FriedlNet and Partners (2007), la tasa de crecimiento mundial de esta industria para el periodo 2002-2006, ha sido en promedio de un 7,12%. Por su parte Larraguibel (2007) señala que existe un mercado potencial nueves superior al actual mercado Outsourcing TI, y hace una estimación del crecimiento para el 2009 de un 6% anual.

Por otro lado, las compañías elaboradoras de software de los países prósperos, están emigrando a los *proyectos distribuidos* para producir software más barato y rápido (Kotlarsky, 2005), aumentando así su competitividad mundial. Este fenómeno también ha facilitado la creación de un conjunto de nuevas compañías provenientes de países emergentes, como las empresas indias ***Tata Consultancy Services*** (*TCS*), ***Infosys*** (Carmel, 2003; Tschang, 2001), entre otras. Mediante la creación y el crecimiento de una industria emergente del software, algunas economías como las de India, China, Rusia, Irlanda e Israel han sido positivamente influidas por este fenómeno (Carmel, 2003).

Considerando lo anteriormente señalado para las economías emergentes, se puede formular la siguiente pregunta:

¿Qué puede hacer un país como Chile para convertirse en una nación exportadora de software?

Una posible respuesta es: ***desarrollando una industria de software mundialmente competitiva***.

Por ejemplo, India logró desarrollar su industria en función de dos hechos: creó un cúmulo de programadores de clase mundial y satisfizo la demanda por personal altamente calificado que requirió la industria del software de EEUU (Tschang, 2001). Sin embargo, otros países también han tenido éxito en los proyectos distribuidos de software, sin la necesidad de contar con un gran número de ingenieros de software, ni con costos tan competitivos en recursos humanos como los de India. Ejemplo de esto son Irlanda e Israel (Carmel, 1999).

En síntesis, cada país tiene sus propias ventajas para poder incentivar una industria de software mundialmente competitiva (por ejemplo: el idioma, la infraestructura TIC, la misma zona horaria con el cliente, entre otras). Sin embargo, un factor preponderante para el éxito de una industria de esta naturaleza es la capacidad de sus empresas para captar y gestionar la demanda internacional. Por tal motivo, la presente investigación procura efectuar un aporte concreto al plantear un ***marco de trabajo*** que pueda guiar exitosamente a los ***desarrollos distribuidos de software***.

1.2 El desarrollo de software

El impacto del software ha sido profundo en la mayoría de las sociedades y culturas del mundo, al mismo tiempo su importancia crece constantemente. Sin embargo, existe una gran cantidad de recursos mal utilizados en proyectos de software fracasados o software que no pudieron satisfacer las necesidades de sus usuarios (Azher, 2007).

Pero ***¿Cuándo se considera que un proyecto de software es exitoso?***
Pressman (2002) señala que estos se consideran exitosos cuando elaboran un producto que satisface las necesidades de los usuarios y se mantienen dentro de los plazos y costos presupuestados.

Este mismo autor señala que, para tener éxito al diseñar y construir software, es necesario aplicar una disciplina.

Pero ***¿Dónde se puede obtener una disciplina para diseñar y elaborar software?***
La ***ingeniería de software*** es un área del conocimiento, que ofrece métodos y técnicas para desarrollar y mantener software de calidad, abordando todas las ***fases del ciclo de desarrollo*** de cualquier tipo de sistema de información. Esta disciplina también procura utilizar racionalmente los recursos tratando de establecer principios y métodos a fin de obtener un producto rentable (Bauer, 1972).

Adicionalmente Wikipedia (2007) señala que esta disciplina incorpora conocimientos de diversas áreas del saber, como lo son: *la gestión de proyectos*, *la gestión de calidad*, *la ingeniería de sistemas*, *las matemáticas*, entre otras. Pressman (2002) afirma que esta disciplina también comprende las tecnologías aplicadas al proceso de elaboración y mantención, como lo son los ***métodos*** técnicos y las ***herramientas*** automatizadas. Finalmente, Wikipedia (2007) también

señala que ésta es una disciplina joven, en permanente evolución y que presenta diversas tendencias. Por ejemplo, algunas corrientes presentes en el área de los ***modelos de procesos*** son: ***Waterfall***, ***Agile***, ***RUP***, ***Spiral***, ***RAD***, ***XP***, ***Cleanroom***, ***Scrum***, ***MSF***, entre otras**.**

Sin embargo, al ser una disciplina joven que presenta un amplio conjunto de tendencias, puede surgir la siguiente interrogante:

¿Qué tendencia debe aplicarse para un exitoso desarrollo distribuido de software?

La respuesta no es trivial, por este motivo la presente investigación provee información para poder tomar una decisión adecuada al momento de establecer un desarrollo remoto.

Toda actividad se facilita cuando se sigue un curso de acciones predecibles. Por este motivo la ***ingeniería de software*** proporciona una serie de ***procesos***, ***métodos*** y ***herramientas*** para que el equipo desarrollador pueda adaptarlas a sus necesidades, dependiendo principalmente del tipo de software que se requiera construir.

Figura 1- 1: Representación del vínculo entre los elementos más destacados en un desarrollo de software basado en la calidad. Extraído de Pressman (2002)

- Las ***herramientas,*** proporciona un enfoque automático o semi-automático para los ***procesos*** y los ***métodos*** de desarrollo.

- Los ***métodos***, indican cómo se debe construir técnicamente el software, comprendiendo una gran gama de actividades.

- Los ***Procesos de desarrollo,*** vinculan a las metodologías con las herramientas y permiten una elaboración racional y oportuna.

- ***Enfocarse en la calidad***, se relaciona con entregar un producto que satisface plenamente las necesidades del cliente basándose en un diseño de calidad y una elaboración que se ajusta a dicho diseño.

Un ***marco de trabajo*** común para el ***proceso de desarrollo***, definido por actividades aplicables a todos los proyectos (independientemente al área de aplicación, tamaño y complejidad), se representa de la siguiente manera:

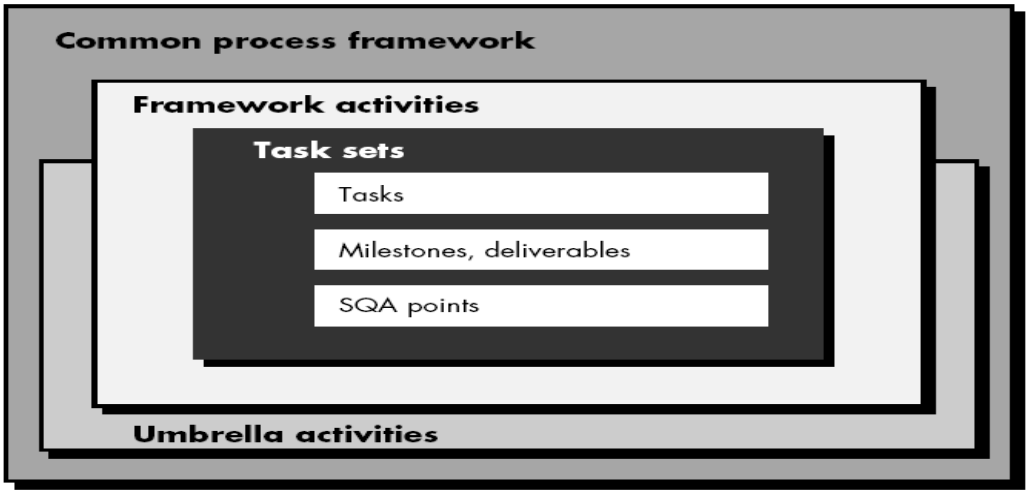

Figura 1- 1: Representación de las actividades aplicables a los proyectos de software. Extraído de Pressman (2002)

- ***El conjunto de tareas*** (*Task Sets*)***:*** Comprende todas las actividades que pueden ser adaptadas a las características del proyecto y a los requisitos del equipo desarrollador. Como lo son las tareas y productos de trabajo, los hitos del proyecto y los puntos de aseguramiento de la calidad.

- ***Las actividades de protección*** (*Umbrella activities*)***:*** Son independientes al proyecto y es recomendable aplicarlas durante todo el proceso de desarrollo. Entre las actividades típicas de esta categoría se incluyen:

  - Seguimiento y control del proyecto
  - Revisiones técnicas formales
  - Garantía de la calidad del software
  - Gestión de la configuración
  - Preparación y producción de documentos
  - Gestión de reutilización
  - Mediciones
  - Gestión de riesgo

Como se ha señalado anteriormente, desde los comienzos de la *ingeniería de software* se han propuesto distintos ***modelos de procesos***. Sin embargo, la elección dependerá de diversos factores, entre ellos:

- La naturaleza del proyecto y su aplicación
- Los métodos y las herramientas que se utilizarán para su desarrollo
- Los controles y entregas que se requieran

Algunos ***modelos de procesos*** son:

- El modelo lineal secuencial

- El modelo de construcción de prototipos
- El modelo de desarrollo rápido de aplicaciones
- El modelo espiral
- El modelo de desarrollo concurrente
- El modelo de desarrollo basado en componentes

Una de los objetivos de esta investigación es proponer un ***modelo de proceso***, que incluya un conjunto de ***tareas*** y ***actividades*** adecuadas para aplicarlas a un ***desarrollo distribuido de software***.

1.3 La gestión de los proyectos de desarrollo de software

Es una disciplina que sirve para guiar la elaboración de productos y sistemas basados en computadoras. Esto implica la planificación, supervisión y el control de: ***el personal***, los ***eventos*** y ***procesos***, desde el inicio del proyecto hasta la implementación operacional.

La relevancia de esta actividad es respaldada por diversos autores como Azher (2007), Kroll y Maclsaac (2006) y Pressman (2002). Como también por modelos de procesos que incluyen las mejores prácticas, ejemplo de esto son RUP y CMMI. Por su parte, Reel (1999) hizo un estudio en 1998 indicando que un 26% de los proyectos de desarrollo de software se suspendieron y un 46% experimentó problemas en los plazos y costos presupuestados. Azher (2007), ratifica dichas cifras mediante un estudio hecho en el 2004 por Standish Group, el cual señala que tan sólo el 29% de estos proyectos se mantuvo dentro de los tiempos y costos estimados. Pressman (2002) atribuye dichos problemas, principalmente, a problemas propios en la gestión de esta clase de proyectos.

Como respuesta a esta problemática, la *ingeniería de software* de los años próximos al 2000 plantea una serie de ítems claves que permiten una gestión más efectiva, por ejemplo:

- Métricas que sirven como base para tomar decisiones de gestión más efectivas
- Técnicas planteadas para estimar los costos y requisitos de recursos
- Técnicas para establecer un plan efectivo del proyecto
- Actividades de gestión para una correcta supervisión, reducción y gestión de riesgos
- Actividades necesarias para definir las tareas y establecer una programación de proyecto realista

- Técnicas para asegurar la calidad mientras se dirige el proyecto
- Técnicas para el control de cambios a lo largo del ciclo de vida de una aplicación

Sin embargo, implementar la mayoría de estas prácticas, en esos años, era costoso y difícil, porque las pocas herramientas de apoyo tenían altos precios, las actividades y técnicas no presentaban un grado de madurez adecuado, y era compleja la capacitación de los miembros del equipo.

Para el *Instituto de Ingeniería de Software* (***SEI***), el ***personal*** también es un factor clave de la gestión, por tal motivo ha desarrollado un *Modelo de Madurez de la Capacidad de Gestión del Personal* (***PM-CMM***), definiendo las siguientes áreas: reclutamiento; selección; desarrollo de la carrera; y diseño de la organización, trabajo, cultura y espíritu de equipo.

Más recientemente, en el 2006, el ***SEI*** también respalda la Gestión de Proyectos como una disciplina clave, sin embargo, realiza una distinción entre ésta y la Gestión de Procesos. Esto se puede apreciar en el *Modelo Integrado de Madurez de la Capacidad para el Desarrollo* (***CMMI-DEV, V1.2***). La finalidad de éste es entregar un *marco de trabajo* para mejorar la madurez de los procesos de desarrollo de productos y servicios. Generalmente, es utilizado por las empresas elaboradoras de software, para guiar sus procesos de desarrollo. También se utiliza para certificar la calidad de los productos elaborados.

Definir los aspectos claves de la Gestión de Proyectos de software para minimizar el riesgo en los desarrollos distribuidos, también es un objetivo que plantea y aborda la presente investigación.

1.4 El desarrollo distribuido de software

El ***desarrollo distribuido de software*** ha sido una tendencia creciente en la última década (Damian y Zowghi, 2003), producido por una serie de *condiciones técnicas* e *incentivos económicos* que han acelerado este proceso (Kotlarsky, 2005). Las *condiciones técnicas* corresponden, principalmente, a herramientas tecnológicas que han permitido la ***colaboración***, ***coordinación*** y ***comunicación*** entres las personas que trabajan remotamente. En cambio, los *incentivos económicos* se han producido por la posibilidad de contratar, a bajo costo, a personal experimentado y/o con alta capacidad. La relevancia de este último punto radica en que; como Brooks (1995) afirma, un buen programador es entre 5 a 10 veces más productivo que uno mediocre; también Carmel (2002) señala que aplicando un ***desarrollo distribuido*** los costos en personal pueden reducirse desde un 30 a un 50%. Otro *incentivo económico* presente en el escenario distribuido, es poder atender las necesidades de clientes que poseen alto poder adquisitivo, pero que se encuentran geográficamente distantes del equipo desarrollador.

Cabe indicar que los estudios sobre el ***desarrollo distribuido de software*** comenzaron como un subconjunto de las investigaciones de proyectos geográficamente dispersos (Kotlarsky, 2005). A fines de la década de los noventa esta área incipiente se estableció como una rama de investigación independiente, sus precursores fueron Carmel (1999) y Karolak (1999).

Aunque la experiencia en este campo se ha incrementado, las investigaciones hechas al respecto todavía son muy limitadas, ya que sólo abordan algunos aspectos del desarrollo de software (Kotlarsky, 2005). Por ejemplo, algunos estudios se enfocan únicamente en alguna ***fase del ciclo de desarrollo***, tal como el ***análisis de requisitos***: Crowston y Kammer (1998) y Damian (2002).

Yuska Costa (2005), de la Universidad Federal de Campina Grande, señala que las interrogantes en este nuevo campo de investigación abordan preguntas desde aspectos amplios, como por ejemplo *¿Qué metodología es la más adecuada para aplicarla en un escenario distribuido?*; hasta aspectos más aplicados, por ejemplo: *¿Qué funcionalidades y características deben poseer las herramientas tecnológicas para posibilitar un exitoso desarrollo distribuido de software?*

Aunque aún no hay consenso en dichas respuestas, se debe tener presente que este nuevo campo de investigación aborda, principalmente, problemas prácticos para efectuar un exitoso ***desarrollo distribuido de software***, ya que busca lograr los beneficios potenciales de estos proyectos (menores costos, mayor calidad y satisfacer a clientes con mayor poder adquisitivo); dado que el escenario distribuido surge como vía alternativa que optimice los recursos necesarios para concretar un proyectos y no para atender problemas irresolubles en el escenario co-localizado (presencial).

Considerando el contexto anterior, las problemáticas que aborda este campo de investigación están modificándose constantemente en el tiempo, dado que frecuentemente surgen nuevas tecnologías y prácticas que facilitan la colaboración, coordinación y control de los equipos remotos. Sin embargo, la experiencia muestra que sólo aplicando tecnología o prácticas adecuadas, no se logran proyectos exitosos porque que existe un preponderante factor humano que debe tomarse en cuenta al momento de gestionar un proyecto de desarrollo de software. Es así como, este campo de investigación pretende resolver diversos problemas, algunos de ellos se expresarán mediante las siguientes interrogantes:

- ¿Qué tecnologías deben aplicarse en los escenarios distribuidos?
- ¿Cómo deben interactuar los miembros de los equipos remotos a través de estas tecnologías?

- ¿Qué conocimientos, prácticas y habilidades adicionales deben poseer las personas involucradas en los desarrollos distribuidos de software?
- ¿Cómo deben organizarse los equipos distribuidos?
- ¿Qué procesos y metodologías de desarrollo de software son adecuados para aplicarlos a un escenario distribuido?
- ¿Cómo deben interactuar los equipos remotos para que los proyectos distribuidos sean exitosos?
- ¿Qué incentivos adicionales deben poseer los integrantes de los equipos remotos para que los proyectos distribuidos sean exitosos? ¿Cómo lograr esto?
- ¿Qué condiciones adicionales deben presentarse en los proyectos distribuidos de software para incrementar el bienestar de los desarrollares involucrados? ¿Cómo lograr esto?
- ¿Qué características propias del software que se desarrollará afectará al exitoso del proyecto distribuido?
- ¿Cómo realizar estimaciones adecuadas de presupuesto y plazos para los desarrollos distribuidos?
- ¿Cuáles prácticas de gestión y cuáles tecnologías deben aplicarse conjuntamente para lograr un desarrollo distribuido exitoso?
- ¿Cuáles prácticas de gestión y herramientas tecnológicas deben aplicarse primero y cuales posteriormente?
- ¿Cómo deberían las empresas desarrolladoras de software pasar desde un escenario co-localizado a uno distribuido sin afectar su integridad operacional?

En particular, Kotlarsky (2005) identifica ciertas causales que atentan contra el éxito en los proyectos distribuidos de software. Es así como, problemas de fácil resolución en los escenarios co-localizados pasan a ser problemas serios o críticos en los escenarios distribuidos, las principales causales identifica por este auto son:

- Distancia:
    - Causa problemas en los mecanismos de control y coordinación
    - Causa perdida de riqueza comunicacional
    - Causa malos entendidos y falta de comprensión por las contrapartes remotas (equipos geográficamente distantes)
    - Causa asimetrías de información entre los equipos distribuidos
    - Causa carencias de comunicación informal e interpersonal
    - Causa desconfianza entre los equipos y contrapartes remotas

- Husos horarios
    - Causa dificultades para coordinar eficientemente el trabajo cuando hay equipos remotos ubicados en diferentes husos horarios
    - Causa retrasos en los trabajos colaborativos

- Diferencias culturales
    - Causa barreras para comunicarse efectivamente produciendo malos entendidos y afectando las sinergias del trabajo colaborativo

Adicionalmente Kotlarsky (2005) señala que no todas las *tecnologías* y *prácticas* sirven para todos los proyecto, tomando un rol fundamental el *contexto* en el cual se desenvolverá el equipo. Actualmente la Ingeniería de Software no atiende estas problemáticas dado que no señala qué *tecnologías* y *prácticas* son adecuadas para un *contexto* en particular. Por este motivo, la presente tesis pretende abordar esta problemática y propone la utilización incremental de un conjunto de *prácticas* y *tecnologías* según el *contexto* del proyecto que se realizará. Dicha propuesta se diseño mediante un riguroso análisis de casos y en base a la aceptación de expertos en Ingeniería en Software Co-localizada y Gestión, como son los profesores de la Escuela de Ingeniería de la Pontificia Universidad Católica de Chile (PUC) y Universidad de Chile (U Chile): David Fuller (Dpto. Ciencias de la Computación,

PUC), Yadran Eterovic (Dpto. Ciencias de la Computación, PUC), Rosa Alarcón (Dpto. Ciencias de la Computación, PUC), Sergio Maturana (Dpto. Ingeniería Industrial y Sistema, PUC) y Luís Guerrero (Dpto. Ciencias de la Computación, U Chile).

Con respecto al modelo de distribución que aborda la presente investigación, se debe tener presente que la distribución de un equipo desarrollador está dada por la cantidad de miembros distantes que posee el equipo. Usualmente, se denomina desarrollo disperso de software cuando el equipo que elabora la aplicación está completamente distribuido (todos los integrantes tienen escasas posibilidades de reuniones presenciales con sus compañeros de proyecto).

La presente investigación analiza las ventajas y desventajas de los distintos grados de dispersión posibles en un equipo distribuido. Para calcular el grado de dispersión, Kroll y Maclsaac (2006) proponen utilizar una fórmula que cuantifica la cantidad de canales de comunicación que se presentan en dichos equipos.

Uno de los objetivos de la investigación es proponer una forma de determinar el grado de dispersión óptima que minimice los riesgos[1] que afecten el éxito del proyecto[2] y maximice las utilidades que percibirá la empresa desarrolladora.

Por su parte, Berenbach (2006) señala cómo se puede

[1] Kroll y Maclsaac (2006) señala que un riesgo es un problema potencial que puede suceder si ocurren ciertas circunstancias. Dependiendo del alcance del riesgo puede afectar las metas del proyecto. Existen diferentes fuentes de riesgos que pueden transformarse en problemas como: técnicos, de gestión, de modificación en los requisitos, entre otros.

[2] Un proyecto exitoso es el que logra las metas auto-impuestas. Las metas más utilizadas corresponden a: cumplir los plazos comprometidos, no sobrepasar el presupuesto destinado al proyecto, suministrar las funcionalidades requeridas por los usuarios y proveer un producto con la calidad adecuada según las necesidades funcionales de usuarios y necesidades económicas de los clientes.

distribuir al equipo desarrollador según los roles de cada miembro y plantea cuáles son las ventajas y desventajas de cada *modelo de distribución para un equipo desarrollador*.

En la presente tesis se consideran los argumentos provistos por Berenbach (2006) proponiendo un grado de dispersión factible para la mayoría de los proyectos de desarrollo distribuido de software.

2 PRÁCTICAS SUGERIDAS PARA MINIMIZAR LOS RIESGOS

2.1 En los proyectos co-localizados de software

2.1.1 Conceptos y problemáticas en la gestión de proyectos

La complejidad de la gestión está presente desde los inicios de los proyectos (en la fase de análisis de requisitos), en cuanto se requieran estimaciones cuantitativas y un plan organizado, sin embargo, en esta etapa no se dispone de información suficientemente sólida para sustentarlos. Un análisis detallado de los requisitos del software proporciona la información necesaria, pero en diversas ocasiones los requisitos se modifican en la medida que avanza el proyecto. Un análisis detallado de los requisitos también toma tiempo y es costoso (Pressman 2002).

Los pasos establecidos para gestionar un proyecto de software son:

- Determinar el ámbito del software
- Descomponer el problema en:
  - Funcionalidades que deben entregarse
  - Los procesos que se emplearán para realizar dichas funcionalidades
- Establecer el ***modelo de proceso*** más adecuado para efectuar el proyecto, basándose en:
  - Los clientes que han solicitado el producto
  - Las características del producto
  - El entorno del proyecto donde trabajarán los miembros del equipo

- El personal que realizará la elaboración del software

- Descomponer el proyecto en tareas

- Emplear actividades estructurales para la maduración del producto y los procesos. Estableciendo por ejemplo:

  - Comunicación y evaluación del cliente
  - Planificación
  - Análisis del riesgo
  - Diseño, implementación y entrega

2.1.1.1 Realizar mediciones

Una de las características principales de las mediciones es proporcionar un mecanismo para realizar evaluaciones objetivas. Asimismo, las ***métricas de software*** corresponden a un amplio conjunto de mediciones para el software de computadora, éstas generalmente se aplican al ***proceso de software*** para perfeccionarlo en base a valores determinados.

Las mediciones son utilizadas por el *jefe de proyecto* para sustentar las *estimaciones*, el *control de calidad*, la *evaluación de la productividad* y el *control del proyecto*. En cambio, los *ingenieros de software*, las utilizan para corroborar la calidad técnica del producto y argumentar las decisiones.

Park, Goethert y Florac (1996) señalan la existencia de cuatro razones para medir el software, éstas son:

- ***Caracterizarlo***: Para comprender mejor los procesos, productos, recursos y entornos.

- ***Evaluarlo***: Para determinar el estado de implementación con respecto al diseño, determinar el logro de los objetivos de calidad y para evaluar indirectamente el impacto de las tecnologías y las mejoras en el proceso de desarrollo.

- ***Predecirlo:*** Para poder planificar y estimar costos, tiempo y calidad**.**

- ***Mejorarlo***: Para identificar obstáculos e ineficiencias a través de información cuantitativa, que permita tomar las medidas necesarias.

Ragland (1996) señala que un ***indicador*** es una métrica o una combinación de ellas, que proporciona una visión profunda del *proceso*, *proyecto* y/o *producto de software*. Los ***indicadores del proceso*** permiten tener una visión objetiva de su eficacia, es así como, el *jefe de proyecto* puede evaluar lo que realmente funciona. En cambio, los ***indicadores del proyecto*** permiten:

- Evaluar el estado actual del proyecto
- Rastrear los riesgos potenciales
- Detectar los problemas antes de que se vuelvan críticos
- Ajustar el flujo y las tareas
- Evaluar el control, de la calidad de los productos de trabajo

Algunas posibles mediciones señaladas por Park, Goethert y Florac (1996) son:

Tabla 2-1: Mediciones señaladas por Park, Goethert y Florac (1996)

Entidad de recurso	Atributo	Medición posible
Personal asignado	Tamaño del equipo	Cantidad de personas asignadas
	Experiencia	Años de experiencia en el área
		Años de experiencia programando
Tiempo	Fecha de inicio y límite	Fechas Calendario
	Tiempo transcurrido	Días
Sistema	Tamaño	Cantidad de módulos
		Cantidad de procesos en un diagrama de flujos de datos
		Cantidad de puntos de función
		Cantidad de líneas de código fuente físicas (SLOC)
		Cantidad de bytes de memoria
	Densidad de los defectos	Cantidad de defectos por KSLOC
		Cantidad de defectos por punto de función
Módulo	Longitud	Cantidad de líneas de código fuente físicas (SLOC)
		Estamentos lógicos de fuente
	Porcentaje de reutilización	Razón entre las líneas físicas no cambiadas y las totales
Documentos	Longitud	Número de páginas
Líneas de código	Tipo de estamento	Nombre del tipo
	Como se produjo	Nombre del método de producción
	Lenguaje de programación	Nombre del lenguaje
Defectos	Tipo	Nombre del tipo
	Origen	Nombre de la actividad cuando fue introducido
	Severidad	Prioridad según las clases de severidad.
	Esfuerzo para repararlo	Horas del personal
	Edad (desde que se abrió)	Tiempo desde que se recibió el reporte del defecto

En ciertos casos, se utilizan las ***métricas del software*** para determinar los ***indicadores del proyecto y de los procesos***. Ejemplo de esto, es la práctica de seguimiento y análisis de defectos (desperfectos antes de la entrega al cliente) y errores (desperfectos posteriores a la entrega).

Las ***métricas del software*** pueden categorizarse en ***directas*** (por ejemplo: *líneas de código*, *velocidad de ejecución*, *tamaño de memoria* y *defectos encontrados*) o ***indirectas*** (por ejemplo: *funcionalidades*, *calidad*, *complejidad*, *eficiencia*, *fiabilidad*, *facilidad de mantenimiento*). Finalmente, las ***métricas*** también se utilizan para mostrar la ***calidad del software***.

Para rastrear el progreso de los productos de trabajo, mientras estos se realizan y aprueban, Pressman (2002) señala que el progreso de un proyecto también puede ser examinado a través de los informes de estado de actividades como: ***control de cambios***, ***control de versiones*** y ***auditorías de la configuración***.

2.1.1.2 Planificar el proyecto

Implica ***estimar*** cuánto dinero, esfuerzo, tiempo y recursos, se deben destinar para construir un sistema o producto específico de software. Las estimaciones en sí conllevan un alto riesgo de ***incertidumbre***. Existen ciertos factores que inciden en el grado de fluctuación de las estimaciones, éstos son:

- ***Complejidad*:** Se atenúa con la experiencia obtenida en proyectos anteriormente ejecutados y de similares características.

- ***Tamaño del proyecto*:** A mayor tamaño del software crecen las interdependencias entre sus elementos, lo que incrementa el problema de la descomposición.

- ***Estructura del proyecto*:** El grado de detalle de los requisitos, la facilidad para la subdivisión de funciones y la naturaleza jerárquica de la información que debe procesarse.

Del mismo modo, cuando los requisitos del proyecto están sujetos a cambio, la ***incertidumbre*** del proyecto se incrementa (Pressman 2002). Los enfoques modernos de *ingeniería de software* (como los ***modelos de procesos evolutivos***) toman un punto de vista ***iterativo***, donde se revisan las estimaciones hechas en cada iteración.

Los pasos básicos para realizar una planificación satisfactoria son:

- Delimitar el *ámbito del software*, al describir:

  - El control
  - Los datos a procesar
  - Las funciones
  - El rendimiento
  - Las restricciones
  - Las interfaces
  - La fiabilidad

- Analizar la *viabilidad del producto y del proyecto* al basarse en cuatro dimensiones:

  - Tecnología
  - Financiamiento
  - Tiempo
  - Recursos

- Determinar los *recursos* requeridos para comenzar el desarrollo del producto. Los principales son:

  - Recursos humanos, habilidades necesarias y cantidad de personas.
  - Recursos del entorno, hardware y software.
  - Recursos reutilizables.

Existen ciertas técnicas que facilitan el proceso de estimación, por ejemplo: ***técnicas de descomposición***, ***técnicas empíricas*** y la utilización de algunas ***herramientas***. Sin embargo, las técnicas mencionadas sólo ayudan parcialmente, siendo el método más efectivo la evaluación permanente de los costos, recursos y plazos (Pressman 2002).

2.1.1.3 Gestionar los riesgos

La ***Gestión de riesgos*** consiste en una serie de pasos que ayudan al equipo desarrollador a comprender y gestionar la ***incertidumbre***.

Como señalan Demarco y Lister (2003) "*Los proyectos que han sido realizado son los que han estado libres de riesgo*". Sin embargo, diversas

organizaciones siguen negando los *aspectos desconocidos* del desarrollo de software, enfocándose sólo en los *aspectos conocidos* (Kroll y Maclsaac 2006). Generalmente los jefes de proyecto cometen el error de estimar y planificar considerando que todas las variables son conocidas, asumen que la elaboración de software es un trabajo mecánico, que el personal es fácilmente reemplazable, entre otros supuestos erróneos. Siendo que cuando los riesgos no se atienden tempranamente se pueden transformar en serio problemas, requiriendo cada vez más recursos para solucionarlos.

Comprender los ***riesgos*** y tomar las medidas proactivas para evitarlos o gestionarlos son elementos claves para asegurar el éxito de los proyectos de software (Pressman 2002). Cabe mencionar que todavía está abierto el debate sobre el ***riesgo*** en esta área del conocimiento, sin embargo, éstos presentan dos características ampliamente aceptadas (Higuera, Dorofee, Walker, y Williams, 1994):

- ***Incertidumbre***: Puede o no ocurrir, se deben diferenciar de los eventos que tengan probabilidad 1 de ocurrencia, porque dichos eventos son limitaciones del proyecto y no deben gestionarse igual que los *riesgos*.

- ***Pérdidas*:** Si el riesgo se convierte en realidad, ocurren consecuencias no deseadas que pueden producir pérdidas.

Pressman (2002) señala que hay riesgos imposibles de predecir, sin embargo, también existen riesgos que se pueden agrupar en:

- ***Riesgos del proyecto***: Son problemas potenciales de presupuesto, planificación temporal, recursos, clientes, requisitos y personal (asignación y organización). Amenazan la planificación temporal y los costos del proyecto.

- ***Riesgos técnicos*:** Son problemas potenciales de diseño, implementación, interfaces, verificación y mantenimiento. Amenazan la calidad, los costos y la planificación temporal del software que se construye.

- ***Riesgos del negocio*:** Son problemas como construir un sistema que no es útil para los usuarios o construir un producto que no se alinea con la estrategia organizacional del cliente. Amenazan la viabilidad del software a construir.

Existen otras formas de categorizar los riesgos como: ***riesgos genéricos*** y ***riesgos específicos del producto***. Ambas categorías presentan distintas formas de identificarlos y gestionarlos.

También existen un subconjunto de ***riesgos genéricos***, que se agrupan en:

- Tamaño del producto
- Limitaciones del negocio
- Características del cliente
- Definición del proceso
- Entorno del desarrollo
- Tecnología a construir
- Tamaño y experiencia del equipo

2.1.1.4 Planificación temporal y trazabilidad

Corresponde a crear una ***red de tareas*** necesarias para lograr el trabajo propuesto en un plazo estipulado. Una vez creada la *red* (determinando el ***esfuerzo***, la ***duración*** y las ***interdependencias*** entre las tareas), se asignan los respectivos responsables para que éstas puedan ser realizadas y rastreadas. Por lo

tanto, ***gestionar el riesgo*** implica adaptar la ***red de tareas*** antes de que los riesgos se vuelvan realidad. La complejidad radica en tratar de comprender las ***interdependencias*** entre las diversas tareas, dado que muchas de éstas se deben realizar en ***paralelo*** y ***el producto de una puede tener gran efecto en las otras que hayan sido desarrolladas o estén en actual desarrollo***.

Además, Pressman (2002) indica que estimar los esfuerzos requeridos y los plazos para lograr una tarea, no es una actividad fácil de realizar.

Es así como, una planificación adecuada requiere que:

- Todas las tareas aparezcan en la red.
- El tiempo, el esfuerzo y los recursos sean asignados objetivamente o basados en la experiencia.
- Las relaciones entre tareas estén correctamente indicadas.
- Los hitos se establezcan pensando en la realización del seguimiento del progreso.

Existen diversas razones por las cuales se retrasan los proyectos de software, las causas más comunes se deben a:

- Fechas límites poco realistas, generalmente impuestas por personal externo al equipo desarrollador.
- Cambios en los requisitos del cliente que no han sido reflejados en la planificación temporal.

- Subestimación de la cantidad de esfuerzos o recursos necesarios para concretar las tareas.

- Riesgos mal analizados y gestionados.

- Dificultades técnicas y humanas no previstas.

- Falta de comunicación.

- Falta de reconocimiento de los retrasos, lo que se traduce en no tomar las medidas a tiempo para corregir los problemas.

Putman y Myers (1992) plantearon una ecuación altamente no lineal entre el tiempo para concretar un proyecto y el esfuerzo humano aplicado. Un análisis de esta ecuación nos indica que los costos del proyecto se reducen cuando se disminuye la cantidad de personas involucradas en el desarrollo y se aumenta el periodo de elaboración.

Con respecto a la generalidad de las tareas, se sabe empíricamente que no hay un único conjunto de tareas que sean apropiadas para todos los proyectos sino que, como se ha planteado anteriormente, dependerán de las características propias del proyecto y del producto.

El ***grado de rigor*** con que se aplica el ***proceso de software*** también es un factor a considerar, dado a que si hay un *mayor grado de rigor* el conjunto de tareas aumentará en tamaño y complejidad. El ***grado de rigor*** es función de las características del proyecto como por ejemplo, que tan crítico es el producto.

Para realizar la planificación temporal de las tareas se pueden aplicar ciertas herramientas y técnicas específicas como: la ***técnica de revisión y evaluación de programas*** (*PERT*) y el ***método de la ruta crítica*** (*CPM*)

La planificación temporal del proyecto define las tareas e hitos que deben seguirse y controlarse en la medida que progresa el proyecto. El seguimiento generalmente se realiza mediante las siguientes acciones:

- Efectuando reuniones periódicas del estado del proyecto, informando el progreso y los problemas.

- Evaluando las revisiones hechas a lo largo del proceso de desarrollo.

- Determinando si se lograron los hitos del proyecto.

- Compara las fechas reales de inicio y las actualizadas, para el cumplimiento de las tareas.

- Revisando las versiones de los elementos que componen el software.

Se puede utilizar el seguimiento de los errores como una medida para evaluar el estado actual del proyecto. La ventaja de este enfoque radica en la cuantificación de problemas potenciales y las respuestas proactivas para superarlos (Pressman 2002).

2.1.1.5 Garantizar la calidad del software (SQA)

Es una ***actividad de protección*** que se aplica a lo largo de todo el ***proceso de software*** y engloba:

- Enfocarse en la gestión de la calidad.
- Los métodos y herramientas de ingeniería de software.
- Las revisiones técnicas formales (RTF) que se apliquen.
- La estrategia de pruebas multi-escalada.
- El control de la documentación y de los cambios realizados.
- Mientras sea posible, basar los procedimientos en estándares de desarrollo.
- Los mecanismos de medición y generación de informes.

Definir ***calidad*** para una *entidad intelectual*, como lo es el software, tiene un mayor grado de complejidad que determinar la ***calidad*** de un *objeto físico*, en el cual existen características físicamente mesurables que pueden compararse con estándares. En el caso del software se pueden distinguir dos *tipos de calidad*:

- ***Calidad de diseño***: Se produce cuando se definen tolerancias más estrictas y niveles más altos de rendimientos en los requisitos, especificaciones y diseño del sistema.

- ***Calidad de concordancia***: Consiste en el grado de cumplimiento de las especificaciones de diseño durante su realización. Por ejemplo, si la implementación del sistema cumple el diseño y los objetivos de los requisitos y el rendimiento, entonces la *calidad de concordancia* es alta.

Glass (1998) indica que la ***calidad*** es una parte importante de la ***satisfacción del cliente***, siendo este último el objetivo principal. Para ejemplificar su teoría proporciona la siguiente relación intuitiva:

$$Satisfacción\ usuarios = satisface\ las\ necesidades + buena\ calidad + despacho\ dentro\ del\ presupuesto\ y\ plazos\ estimados$$

Figura 2- 1: Relación propuesta por Glass (1998) para señalar los factores que influyen en la satisfacción del cliente

Algunos conceptos sobre la *calidad de software* son:

- ***Control de calidad:*** Se produce a través de una serie de inspecciones, revisiones y pruebas a lo largo de todo el ***proceso de software*** para asegurar que cada producto cumpla los requisitos asignados. La combinación de *mediciones* y *feedback* permiten mejorar este proceso. Las actividades de ***control de calidad*** pueden ser manuales, automáticas o semi-automáticas.

- ***Garantía de la calidad:*** Proporciona la gestión necesaria para informar la calidad a través de datos, entregando una visión profunda e imparcial sobre el cumplimiento de los objetivos.

- ***Costo de la calidad:*** Incluye los costos de investigación para garantizar la calidad y los costos de actividades necesarias para obtener la calidad adecuada.

Los costos para reparar un defecto aumentan dramáticamente cuando avanzan desde prevención a detección, y también cuando cambian desde un fallo interno a uno externo. Boehm (1981) garantiza este fenómeno gracias a un análisis que realizó basándose en datos recopilados en proyectos reales. La siguiente figura refleja lo planteado por Boehm (1981):

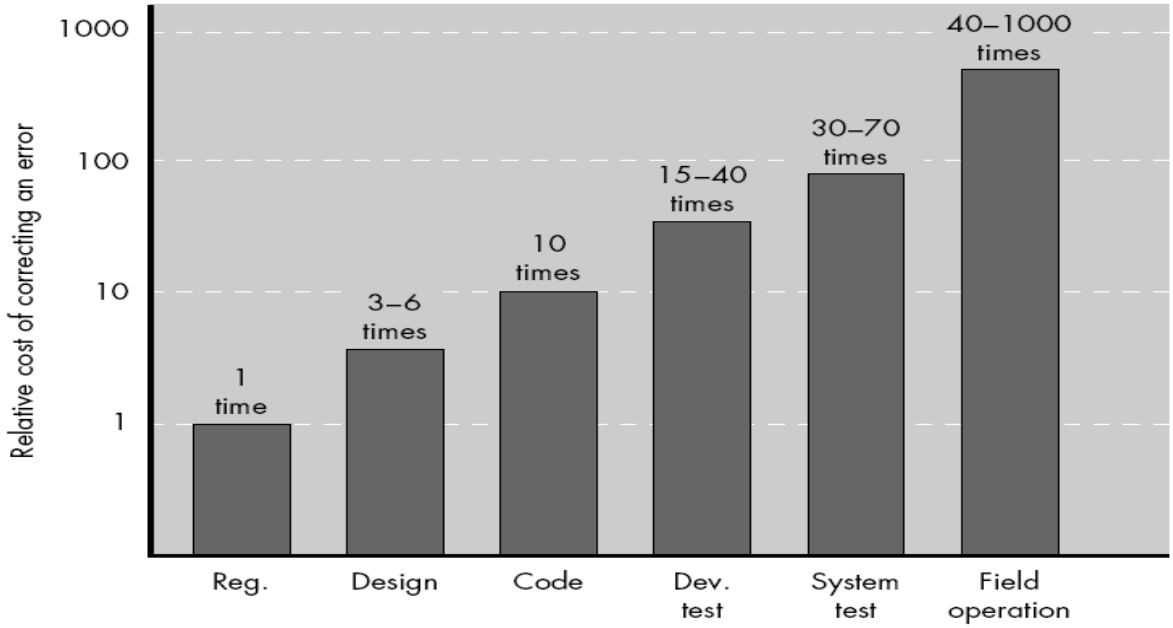

Figura 2- 2: Costo relativo de corregir un error según la fase del ciclo de desarrollo donde se repare. Boehm (1981)

Las actividades de ***SQA*** generalmente son realizadas por un grupo independiente de miembros. Algunas de estas tareas son:

- Establecer un ***plan de SQA*** para el proyecto, el que debe identificar:

  - Las auditorías y revisiones a realizar
  - Los estándares que se pueden aplicar al proyecto
  - Los procedimientos para informar y seguir los errores

- Los documentos producidos por el grupo ***SQA***
- La información de *feedback* que se entregará al resto del equipo

- Participar en la descripción del ***proceso de software*** del proyecto

- Revisar las actividades de *ingeniería de software* verificando que se ajustan al ***proceso de software*** definido

- Garantizar que las reasignaciones de trabajo y la documentación se realice de acuerdo a estándares

- Registrar e informar lo que no se ajusta a los requisitos

Con respecto a las actividades de revisiones diversos autores señalan que una ***revisión técnica formal*** (*RTF*) es el filtro más efectivo para mejorar la *calidad del software*. Los objetivos de las ***RTF***s son:

- Descubrir errores en la función, lógica e implementación del software.

- Verificar que el software logre los requisitos que están bajo revisión.

- Garantizar que se utilicen estándares predefinidos de representación.

- Lograr que el software se desarrolle de manera uniforme.

- Disciplinar la evolución del proyecto.

Generalmente las ***RTF***s se centran sólo en una parte específica del total del software, se realizan mediante reuniones donde se presentan ciertas características particulares. Al final de cada revisión se decide si se rechaza o acepta el producto de forma íntegra o parcial. Posteriormente se debe realizar un informe que señale lo atendido, quién hizo la revisión y las conclusiones acordadas.

La ***garantía de calidad estadística*** es una tendencia creciente en esta industria porque establece una calidad más cuantitativa. Lo que implica:

- Agrupar y clasificar la información sobre los defectos del software.
- Identificar la causa de cada defecto.
- Identificar los defectos críticos para poder corregirlos.

2.1.1.6 Gestión de Configuración del software (SCM)

La elaboración de software necesariamente involucra cambios (se conoce a este hecho como la Primera Ley de la Ingeniería de Sistemas), éstos pueden confundir a los *ingenieros de software* que trabajan en el proyecto. La confusión surge cuando las variaciones no se han:

- Analizado debidamente antes de realizarlas.
- Registrado antes de implementarlas.
- Informado a las personas que necesitan saberlo.
- Controlado de manera que mejoren la calidad y reduzcan los errores.

Esta práctica señala un conjunto de actividades de auto-corrección que se aplican durante el ***proceso de software*** para su seguimiento y control. Comienza cuando se inicia el proyecto y terminan cuando se deja de aplicar soporte al software.

Como los cambios pueden producirse en cualquier momento del *ciclo de desarrollo*, esta actividad sirve para:

- Identificarlos.
- Controlarlos.
- Garantizar que se implementen adecuadamente.
- Auditarlos e informarlos a todos los interesados.

En resumen, esta práctica procura controlar el caos que se produce en los *flujos de cambios*, como también pretende controlar el número creciente de ***elementos de configuración*** a medida que progresa el proyecto.

En la práctica, todas las ***tareas de ingeniería de software*** producen uno o más productos (también llamados *artefactos*). Cuando formalmente se revisó la versión de un producto y se llegó a consenso, entonces este se establece como una *versión oficial* o ***baseline***. Posteriormente los ***baselines*** serán los puntos de inicio para que los artefactos sigan evolucionando según las metas propuestas.

Algunas actividades involucradas en esta práctica son:

- ***Auditoría de la configuración***: Esta actividad complementa las *RTF*s al comprobar características que no han sido consideradas en otro tipo de revisiones. También permite plantear y responder las siguientes interrogantes:
    - ¿Se han hecho los cambios especificados?

- ¿Se han incorporado modificaciones adicionales?
- ¿Se han especificado las fechas de cambio y el autor?

- ***Informes de los estados de la configuración***: Esta actividad ayuda a mejorar la comunicación entre las personas involucradas al plantear y responder las siguientes interrogantes:

  - ¿Qué pasó?
  - ¿Quién lo hizo?
  - ¿Cuándo pasó?
  - ¿Qué más se vio afectado?

- ***Control de acceso***: Administra los derechos de los *ingenieros de software* para acceder y modificar los objetos de configuración.

- ***Control de sincronización***: Garantiza que los cambios en paralelo no se sobrescriban mutuamente.

Bach (1998) afirma que cuando se aplica escasamente esta práctica, generalmente se incide en problemas que afectan la viabilidad del proyecto. Sin embargo, aplicar mucho ***control de cambio*** también induce a otros problemas como incrementar los tiempos, esfuerzos y costos; por esto, se debe equilibrar su aplicación. También, se debe considerar que los problemas relacionados con la ***gestión del control de cambios*** disminuyen notablemente gracias a la utilización de herramientas especialmente diseñada para esta actividad.

En resumen, esta práctica corresponde a una *actividad de protección* que se aplica a todo el ***ciclo de desarrollo*** controlando, auditando e informando las modificaciones y los estados de los productos y tareas del proyecto.

Por consiguiente, ***el progreso de un proyecto también puede ser examinado a través de los informes de estado***. Por su parte Pressman (2002) indica que el ***control de cambios***, ***control de versiones*** y las ***auditorías de la configuración***, son un conjunto de actividades que ***garantizan la calidad del producto***.

2.1.2 Principios aplicados en los proyectos de software más exitosos

Los siguientes principios son los más observados por los miembros de ***IBM Rational*** en los proyectos más exitosos de la industria del software (Kroll y Maclsaac 2006), estos son:

- ***Adaptar el proceso***.
- ***Equilibrar las prioridades de los Stakeholders***.
- ***Colaboración a lo largo del equipo***.
- ***Demostrar valor iterativamente***.
- ***Elevar el nivel de abstracción***.
- ***Enfocarse continuamente en la calidad***.

A continuación se presenta una breve explicación de cada principio y las prácticas que las sustentan.

2.1.2.1 Demostrar valor iterativamente

Cada iteración debe entregar capacidades incrementales que hayan sido valoradas por los *stakeholders*. Esto faculta que los requisitos se adapten mejor a la planificación mediante *feedback* rápido y perfecciona la *gestión de riesgos* aumentando la *predictibilidad* del proyecto.

Beneficios:

- Reducción prematura del riesgo.
- Alta predictibilidad durante todo el proyecto.
- Promueve la confianza entre los *stakeholders*.

Patrones:

- Proveer en cada iteración un *valor incremental de uso* mediante *feedback*.
- Adaptar la planificación utilizando un proceso iterativo.
- Aceptar y gestionar los cambios.
- Atender anticipadamente los mayores riesgos técnicos, de negocio y de planificación.

Anti-patrones:

- Planificar en detalle todo el ***ciclo de desarrollo***.
- Garantizar el progreso del proyecto en sus dos primeros tercios, basándose en las *revisiones de especificaciones* y de *productos de trabajo*, en vez de dar mayor importancia a los resultados de las pruebas y a demostrar que el software funciona.

2.1.2.2 Enfocarse constantemente en la calidad

Mejorar continuamente la calidad requiere más que testear las condiciones de uso para validarlas. Más bien, incluye a todos los miembros involucrados en el *ciclo de desarrollo* para construir *procesos y productos de calidad*. Las *estrategias iterativas* se enfocan en las pruebas prematuras y construir automáticamente durante todo el *ciclo de desarrollo*, lo que reduce la cantidad de defectos, proporciona métricas de calidad basadas en hechos, y permite planificar y adaptar el producto objetivamente.

Beneficios:

- *Calidad alta*.
- Comprensión anticipada de la *calidad de los productos*.

Patrones:

- El equipo debe responsabilizarse por la calidad del producto.
- Realizar pruebas de forma prematura y de manera continua, en conjunto con una integración de capacidades demostrables.
- Construir incrementalmente pruebas automatizadas.

Anti-patrones:

- Guiarse en profundidad por todos los artefactos intermedios.
- Completar las pruebas unitarias de toda la aplicación antes de hacer pruebas de integración.

Los *procesos iterativos* se adaptan para conseguir la *calidad* cuando ofrecen diversas medidas y oportunidades de corrección. Para garantizar una *calidad alta* se requiere la participación de *todos los miembros del equipo* en todas las *etapas del ciclo de desarrollo*, es así que:

- ***El jefe de proyecto***: Debe garantizar la planificación de las pruebas y que estén los recursos necesarios para la construcción de los tests requeridos.

- ***Los analistas***: Deben responsabilizarse de asegurar que los requisitos estén sujetos a pruebas.

- ***Los desarrolladores***: Deben elaborar aplicaciones pensando en sus pruebas y deben responsabilizarse de las pruebas de su código.

- ***Los testers***: Deben guiar al resto del equipo para que comprendan las problemáticas de la *calidad del software*, para esto deben proporcionar *métricas objetivas de calidad* durante todo el proyecto.

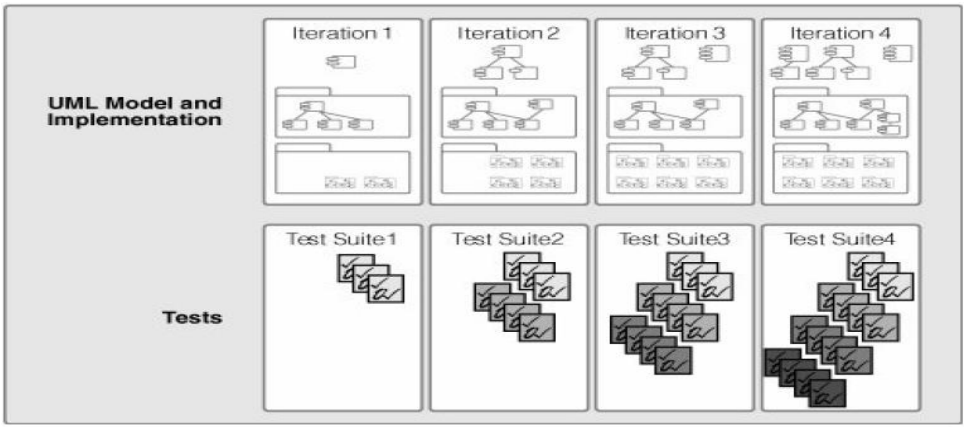

Figura 2- 3: Desarrollo de un sistema y la estructura de pruebas que deben ser aplicadas según el avance del proyecto basado en iteraciones. Extraída de Kroll y Maclsaac (2006)

A medida que se construye incrementalmente la aplicación, también se deben construir pruebas automáticas para detectar los defectos de forma prematura. Mientras se diseña el sistema se debe considerar cómo deberá ser testeado, así se toman las decisiones de diseño más adecuadas para mejorar la habilidad de automatizar las pruebas. Esto permite crear las directrices que guiarán a los tests desde la misma modelación del diseño. Según Kroll y Maclsaac (2003) estas medidas ahorran tiempo, entregan incentivos para las evaluaciones tempranas e incrementan la calidad de las pruebas minimizando el número de errores.

Una máxima es tratar de automatizar todas las ***pruebas unitarias*** y en menor medida las ***pruebas de aceptación***. Por último, escribir las pruebas antes de que el código sea elaborado también es una buena práctica.

2.1.2.3 Equilibrar las prioridades de los stakeholders

Ellos Siempre van a tener prioridades, tal como producir una solución lo más rápido y económicamente posible, versus satisfacer todos los requisitos del negocio. Por lo tanto, es necesario trabajar estrechamente con los *stakeholders* para garantizar que se atiendan sus preferencias y para priorizar adecuadamente el proyecto. También es necesario producir un balance adecuado entre *apalancar el valor existente* y construir software *a medida*, asumiendo que en algunos casos se pueden comprometer ciertos requisitos.

Beneficios:

- Alinear las aplicaciones con las necesidades del negocio y de los usuarios.
- Reducir el desarrollo personalizado.
- Optimizar el valor del negocio.

Patrones:

- Definir, entender y priorizar las necesidades del negocio y de los usuarios.
- Priorizar los proyectos y requisitos, como también hacer coincidir la necesidades con las capacidades del software.
- Entender cuales activos pueden ser apalancados.
- Balancear la reutilización de activos con las necesidades del negocio.

Anti-patrones:

- Conseguir requisitos completos y minuciosos antes del comienzo el proyecto.
- Documentar completamente los requisitos al inicio del proyecto.
- Guiar el proyecto hacia una solución personalizada.
- Diseñar un sistema que cumpla sólo con las necesidades de los *stakeholders* más demandantes.

2.1.2.4 Mejorar la colaboración entre los miembros del equipo

Al facilitar que los integrantes den lo mejor de sí mismos; integrando personas talentosas y con las habilidades correctas; superando las barreras que impidan una colaboración efectiva; y proporcionando ambientes adecuados para favorecer una colaboración significativa.

Beneficio:

- Mayor productividad del equipo.
- Una relación más directa entre las necesidades del negocio, el desarrollo de software y la operación de los sistemas.

Patrones:

- Motivar a las personas para que den lo mejor de ellos.
- Alentar la mutua colaboración.
- Proveer un ambiente efectivo para la colaboración.
- Integrar a los equipos de negocio, software y operaciones.

Anti-patrones:

- Fomentar largas jornadas de trabajo donde la responsabilidad del desarrollo depende de cada miembro y no hay control por parte del *jefe de proyecto*.
- Tener integrantes especializados y equipados con herramientas escasamente integrables limitando así la colaboración entre los diferentes miembros del equipo.

La mayoría de los sistemas complejos requieren la colaboración de un número importante de *stakeholders* que posean diversas habilidades. En los proyectos de mayor envergadura, generalmente, se sobrepasan las fronteras geográficas y horarias, incrementando la complejidad del ***proceso de desarrollo***. Por

su parte, las ***metodologías ágiles*** atienden la complejidad basándose en los elementos ***soft*** del *desarrollo de software*, es decir, en las ***personas*** y la ***colaboración***.

2.1.2.5 Elevar el nivel de abstracción

La complejidad es el principal enemigo de los proyectos exitosos. Ésta se reduce minimizando la cantidad de construcciones humanas como *código*, *estructura de datos*, *componentes*, *elementos de modelación* u otros; reutilizando activos como: *modelos del negocio*, *componentes*, *patrones* y *servicios*; apalancando herramientas, *frameworks y lenguajes*; automatizando las *pruebas unitarias;* administrando la complejidad de la *Gestión de Configuración; y* promoviendo la simplicidad al implementar primero los aspectos claves de arquitectura y manteniendo lo más limpio posible los modelos y el código.

Beneficios:

- Mejorar la productividad.
- Reducir la complejidad.

Patrones:

- Reutilizar activos existentes.
- Utilizar herramientas y lenguajes de alto nivel para reducir la cantidad de documentación producida.
- Enfocarse en la arquitectura.

Anti-patrones:

- Ir directamente desde los requisitos ambiguos y de alto nivel a un código de cliente artesanal.

La complejidad es una problemática en el desarrollo de software. Elevar el nivel de abstracción ayuda a reducir la complejidad y la cantidad de documentación requerida para el proyecto y facilita la comunicación. Se puede alcanzar este nivel de abstracción reutilizando artefactos, utilizando herramientas para una modelación de alto nivel, realizando ***refactoring*** y estabilizando la arquitectura tempranamente.

Una estrategia efectiva para reducir la complejidad consiste en reutilizar los activos existentes, como: ***componentes***, ***sistemas legados***, ***procesos de negocios***, ***patrones*** y/o software ***Open Source***.

La reutilización también se facilita al utilizar estándares abiertos, como: ***SOAP***, ***WSDL***, ***RAS***, ***UDDI*** y ***XML***.

Se pueden reutilizar los activos existentes gracias a la ***arquitectura orientada a servicios***, ya que los ***componentes*** o los ***sistemas legados*** pueden ***envolverse*** en ***servicios***, permitiendo de esta manera, que otros ***componentes*** o ***aplicaciones dinámicas*** accedan a sus capacidades a través de interfaces basadas en estándares; al margen de la plataforma y tecnología utilizada.

Otra forma de reducir la complejidad y mejorar la comunicación, consiste en apalancar herramientas de alto nivel, utilizar ***frameworks de software***, ***servidores de aplicaciones*** y lenguajes. Los ***frameworks de software*** reutilizan diseños y/o arquitecturas de software que atienden ciertos tipos de problemas; los ***servidores de aplicaciones*** facilitan la interacción entre las distintas aplicaciones; mientras que los lenguajes estándares como ***UML*** y ***EGL***, entregan las habilidades para una construcción veloz y de alto nivel, tal como un ***proceso de negocio*** y un ***componente de servicio***. Todas estas medidas facilitan la colaboración y ocultan los detalles innecesarios.

2.1.2.6 Adaptar el proceso

Más procesos no implica un mejor proyecto. Más bien, se deben adaptar a las necesidades específicas del proyecto, basándose en el tamaño, la complejidad, la necesidad de cumplimiento, entre otros. Además, se deben adaptar a cada *fase del ciclo de desarrollo*; por ejemplo, menos rigurosidad al principio del proyecto y más en las etapas finales. También, se debe procurar un mejoramiento continuo; por ejemplo, midiendo el logro de las metas al final de cada iteración.

Beneficios:

- Eficiente ***ciclo de desarrollo de software***.
- Comunicación de los riesgos abierta y honestamente.

Patrones:

- Tamaño adecuado de los procesos que necesita el proyecto, incluyendo el tamaño y la distribución del equipo, la complejidad de la aplicación y la necesidad de cumplimiento.
- Adaptar la rigurosidad del proceso a las ***fases del ciclo de desarrollo***.
- Mejorar el proceso continuamente.
- Equilibrar la planificación y estimaciones del proyecto versus su incertidumbre.

Anti-patrones:

- Ver positivamente el tener más procesos, más documentación y una planificación inicial más detallada.
- Insistir en estimaciones tempranas y considerarlas como ciertas.
- Utilizar la misma cantidad de procesos a través de todo el proyecto.

2.1.3 Prácticas seleccionadas

2.1.3.1 Aceptar y gestionar los cambios

En los sistemas físicos los cambios son difíciles y costosos de efectuar cuando ya se ha iniciado su construcción. En cambio, el software puede ser modificado fácilmente permitiendo su evolución a favor de las necesidades cambiantes de los *stakeholders*. Sin embargo, el cambio continuo representa un reto para el equipo desarrollador, porque los cambios introducen costos que necesitan ser gestionados. Esta práctica señala que se deben dar a conocer las situaciones que pueden conducir al caos, como también los cambios necesarios e importantes. Lo anterior se realiza proporcionándole información confiable al cliente sobre el impacto potencial de los cambios en costos y tiempos.

2.1.3.2 Apalancar apropiadamente la automatización de pruebas

En algunas ocasiones la automatización de pruebas puede ser perjudicial para la productividad y la calidad, porque se puede invertir demasiado tiempo automatizándolas o porque se pueden efectuar pruebas que no ayudan a encontrar los defectos más adecuados para el tipo de pruebas que se están realizando. Es decir, demasiado o poca automatización de pruebas puede tener implicancias negativas.

Algunos mitos sobre la ***automatización de pruebas*** son:

- Reduce el tiempo de los tests si es que estos se realizan manualmente.

- Elimina la necesidad de pruebas independientes.

- Substituye la estrategia global de tests para el software.

- Trabaja bien en todos los casos.

En algunas instancias la automatización es inapropiada porque proporciona un *bajo retorno de la inversión* o porque no satisface de buena forma una situación particular. En cambio, una apropiada automatización de pruebas ayuda a encontrar y prevenir problemas antes que se vuelvan graves.

Las áreas donde la automatización de pruebas se vuelve más beneficiosa se categorizan de la siguiente manera:

- ***Ambiente***: Automatizar el ambiente es la forma más útiles para este tipo de pruebas.

- ***Regresión***: Este concepto hace alusión a emplear una prueba automática más de una vez, lo que es particularmente útil para el desarrollo iterativo, como también es conveniente para capturar errores que hayan sido introducidos con posterioridad.

- ***Funcionamiento y estrés***: El desarrollo iterativo permite pruebas tempranas de funcionamiento y estrés.

- ***Interfaz de usuario***: Ayuda a capturar los defectos que los usuarios pueden ver.

2.1.3.3 Construir equipos de alto desempeño

Pressman (2002) señala tres características presentes en los equipos de alto rendimiento (*cohesionados*): ***Confianza***, ***Talento*** y ***Unión***.

- ***Confianza***: Los miembros del equipo deben confiar los unos con los otros.
- ***Talento***: La distribución de habilidades debe adecuarse al problema.
- ***Unión***: Se debe mantener al equipo unido de tal forma que se produzcan sinergias.

En cambio, Jackman (1998), define cinco factores que atentan contra la *cohesión de los equipos*, los que impiden lograr las sinergias esperadas. Los factores son:

- Atmósfera de trabajo frenética donde la energía y los objetivos no se centran en el desarrollo
- Procesos escasos o inadecuados
- Confusa definición de los roles, dado que produce falta de responsabilidad
- Continua y repetida exposición a los fracasos
- Fricción entre los miembros del equipo por problemas tecnológicos, de negocio o personales

Este mismo autor indica acciones que permiten controlar los factores *anti-cohesión*, estos son:

- Procurar que el equipo tenga toda la información necesaria para poder realizar su trabajo
- No modificar los objetivos ni las metas principales
- Informar tempranamente los conflictos que se presenten, para tomar las medidas pertinentes y solucionarlos a la brevedad
- Evitar que los desarrolladores pierdan la autoridad para controlar la situación
- Fomentar que todo el equipo tome parte en las decisiones técnicas y de progreso
- Garantizar que las características del software se ajusten al rigor del proceso elegido
- Permitir que el equipo desarrollador seleccione el proceso, responsabilizándose de elaborar un producto de calidad
- Precisar los *roles* y las *responsabilidades* del jefe de proyecto y del equipo desarrollador
- Definir una plan de contingencia cuando un miembro del equipo falle en el desarrollo
- Establecer técnicas de *feedback* y solución de problemas
- Establecer una actitud que considere las fallas de un integrante como un problema del equipo completo

Además, Pressman (2002) señala que en la cohesión de los equipos también influyen los diferentes rasgos de personalidad presentes en sus integrantes, sin embargo, no detalla una medida específica para tratar esta situación, sino recomienda tener presente este imponderable para atenderlo de forma reactiva según cada circunstancia.

Por su parte, Kroll y Maclsaac (2006) exponen que un equipo debe basarse en una ***colaboración efectiva para alcanzar los objetivos comunes***. Afirman igualmente, que construir *equipos de alto desempeño* es una tarea difícil que debe realizarse continuamente, ya que existe una fuerte problemática que radica en la interacción constante de personas que poseen diferentes responsabilidades, por ejemplo: *analistas*, *arquitectos*, *desarrolladores*, *testers*, *gestores de configuración* y *gestores del proyecto*. Además, la ***colaboración*** y ***comunicación*** se debilita por otros factores como: poseer integrantes en diferentes lugares geográficos, distintas culturas y objetivos personales disímiles. Todas estas variables dificultan la construcción de un equipo *cohesionado* que colabore efectivamente en el logro de las metas acordadas dentro de los parámetros establecidos.

Los principales consejos, señalados por estos autores, para construir un equipo *cohesionado* son:

- ***Inculcar valores adecuados***: En necesario contar con un conjunto de reglas explícitas e implícitas que deben ser continuamente comunicadas y utilizadas como parte importante del estándar de evaluación del equipo y sus miembros. Las reglas más utilizadas son:

    - Ética;
    - Honestidad;
    - Calidad;
    - Justicia;

- Lealtad;
- Integridad;
- Pensamiento innovador;
- Enfocarse en el éxito del cliente y las utilidades de la organización;
- Participar en la comunidad; y
- Trabajo en equipo.

- ***Proporcionar claridad organizacional***: Los miembros del equipo deben entender y estar de acuerdo con las respuestas de las siguientes preguntas:

  - ¿Cuál es la misión del equipo?
  - ¿Cuál es la visión de la aplicación que se entregará?
  - ¿Cómo se medirá el éxito del equipo?
  - ¿Quiénes son los *stakeholders* del proyecto?
  - ¿Cómo se medirá el éxito del proyecto?
  - ¿Quién es el responsable de cada aspecto?
  - ¿Qué procedimientos se deben seguir para realizar el trabajo?

- ***Emplear a la gente adecuada.***

  - ¿Los integrantes comparten los valores de la organización?
  - ¿Los miembros del equipo respetan y apoyan la incorporación de un nuevo integrante?
  - ¿El personal cree en el equipo y en el proyecto?

- ***Construir confianza***: La *confianza* consiste en pensar que los pares no desean hacernos daño. En cambio, generalmente, la *confianza* se mal interpreta como no cuestionar la calidad o exactitud del trabajo del otro.

- ***Crear equipos ganadores***: Esta clase de equipos atrae a personas talentosas y alienta el logro de los hitos. Algunos consejos para fomentar estos equipos son:

    - Una primera iteración exitosa.
    - Limitar las metas.
    - Implantar una actitud de "*tan sólo hazlo*".
    - Permitir que los miembros del equipo puedan asignar y estimar las tareas.
    - Tener una actitud positiva.

2.1.3.4 Establecer mecanismos eficaces de comunicación para mejorar la coordinación

Pressman (2002) indica que los problemas presentes en los proyectos de software, se pueden producir por las siguientes causas:

- ***Tamaño***: Trae complejidad, confusión y dificultades significativas para poder coordinar a los miembros del equipo.

- ***Incertidumbre***: Es usual y proporciona un continuo flujo de cambios que impacta al proyecto.

- ***Interoperabilidad***: La mayoría de las aplicaciones deben comunicarse o ajustarse a las restricciones de los sistemas anteriores.

Este mismo autor también manifiesta que estas causas se enfrentan ***estableciendo métodos efectivos de coordinación***, los que necesitan ***mecanismos eficaces de comunicación entre los miembros del equipo***.

2.1.3.5 Establecer reglas de comunicación

La ***comunicación formal*** se lleva a cabo mediante *documentos escritos*, *reuniones organizadas* y *otros canales de comunicación impersonales y no interactivos*. Por su parte, la ***comunicación informal*** es más personal, las ideas se comparten, la ayuda se produce en la medida que surjan los problemas y cuando se interactúa con otras personas.

Kraul y Streeter (1995) categorizaron las *técnicas de comunicación y coordinación* para estudiar su valor y utilización, las categorías fueron:

- ***Redes Interpersonales***: Corresponden a discusiones informales entre los miembros del equipo o con personas que no están formalmente en el proyecto, pero que pueden proveer experiencia o una visión útil.

- ***Procedimientos interpersonales y formales***: Centrados en las actividades de *garantía de la calidad*. Incluyen reuniones de *revisiones de estados* e *inspección de diseño y código*.

- ***Procedimientos interpersonales e informales***: Incluyen reuniones de grupo para la divulgación de información y resolución de problemas.

- ***Estrategias impersonales y formales*:** Incluyen los documentos, entregables, memorándums técnicos, hitos del proyecto, planificaciones, herramientas de control, solicitudes de cambio, información de seguimiento de errores e información almacenada.

- ***Comunicación electrónica*:** Corresponde a los correos electrónicos, boletines electrónicos de noticias y sistemas de video conferencia.

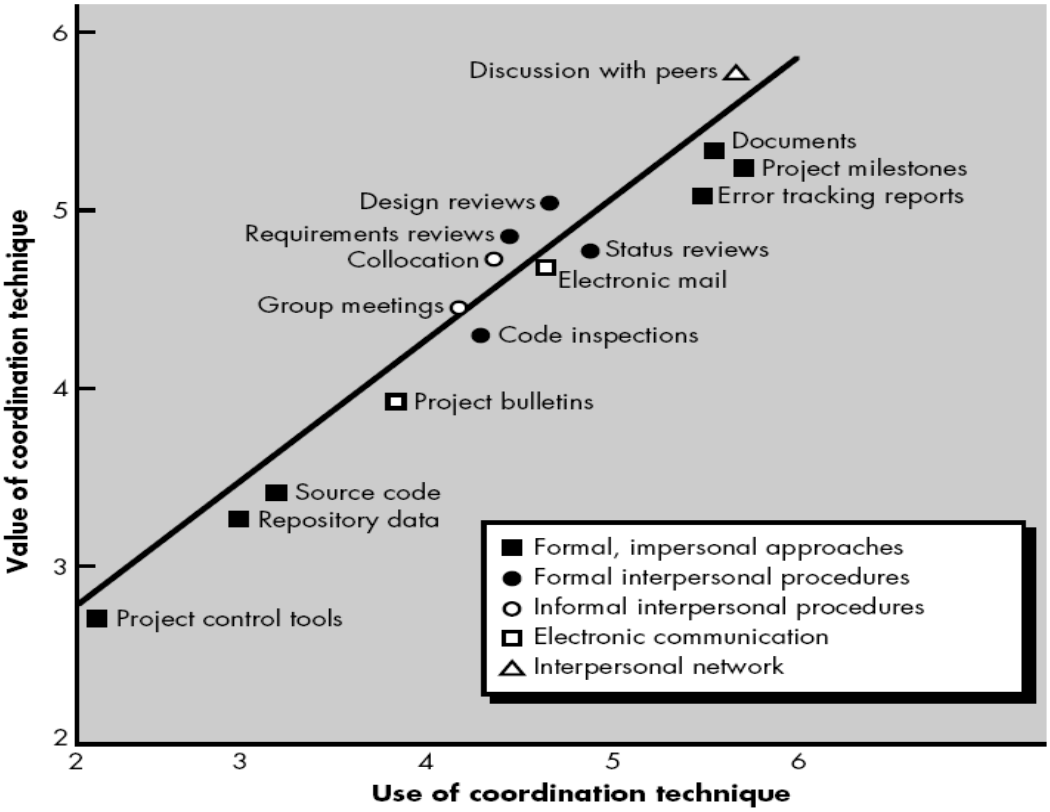

Figura 2- 4: Gráfico extraído del estudio hecho por Kraul y Streeter (1995), donde se muestra la valoración y el empleo de las técnicas de comunicación

La figura anterior muestra las *técnicas de comunicación* más valoradas por los miembros de los equipos desarrolladores, son aquellas que están por encima de la curva del gráfico. Ordenadas de mayor a menor valor, son:

- Discusión cara a cara con los pares (redes interpersonales).
- Revisiones de diseño (Procedimientos interpersonales y formales).
- Revisión de requisitos (Procedimientos interpersonales y formales).
- Colaboración (Procedimientos interpersonales e informales).
- Reuniones de grupo (Procedimientos interpersonales e informales).

Se desprende de esta investigación que las cinco técnicas más valoradas para la comunicación involucran la ***comunicación interpersonal***, siendo ésta más dificultosa en los ambientes de ***desarrollo distribuidos***. Por lo tanto, sustituir la comunicación interpersonal es una dificultad importante que debe ser atendida para lograr un exitoso ***desarrollo distribuido de software***.

Una forma de encarar esta dificultad se logra estableciendo reglas claras de comunicación para maximizar la efectividad y cohesión del equipo. Algunas reglas propuestas son:

- Estimular, cuando sea posible, la comunicación ***presencial***.
- Utilizar reuniones ***presenciales*** o telefónicas para resolver debates efectuados por correo electrónico.
- Promover la mensajería instantánea en los equipos distribuidos.
- Alentar reuniones que no estén relacionadas con el trabajo.
- Reunir en un lugar a todos los miembros del equipo que trabajen en una misma actividad o artefacto.
- Colocar a los miembros del equipo en las inmediaciones más cercanas.

2.1.3.6 Gestionar las versiones

Esta práctica es crítica para gestionar software complejo y sus cambios en el tiempo. El problema surge cuando diversos desarrollos involucran un amplio conjunto de personas y equipos, que trabajan en forma paralela o interdependiente sobre múltiples iteraciones, productos, plataformas y realizaciones. Esto facilita que se pierda el registro de lo que ha cambiado, por qué ha variado y cómo encajan las diferentes piezas, lo que generalmente conlleva un aumento de los costos, plazos y disminución de la calidad.

Esta práctica pretende controlar el caos producido en los proyectos complejos al proporcionar una visión clara que facilite la gestión de las versiones de: los *archivos*, *componentes* y *productos*. Al organizar el cambio de forma más manejable, entregable y dentro de un *ambiente estable*, para trabaja con mayor efectividad.

Según Kroll y Maclsaac (2006) la *gestión del cambio* se dificulta incrementalmente por las siguientes razones:

- Una estrategia iterativa de desarrollo conlleva componentes en constante evolución.

- Compartir los componentes a través de los productos, produce que un único componente pueda tener diversas variables que podrían evolucionar en paralelo.

- El incremento de la coexistencia de distintos elementos como los son los componentes dinámicos, arquitectura orientada a servicios y sistemas legados.

Según estos mismos autores un control de versiones efectivo requiere:

- Un repositorio seguro para almacenar las versiones.

- Equilibrar el control versus el acceso.

- Utilizar herramientas que faciliten el control de versiones.

- Proporcionar ***ambientes de trabajo*** (*Workspaces*) controlados, permitiendo que los desarrolladores realicen cambios que no impacten al proyecto.

- Posibilitar el desarrollo paralelo a través de ***Streams***. Los ***Streams*** son espacios lógicos de trabajo que contienen la versión de un archivo aceptado (***baseline***) y las actividades que describen lo que debe cambiar.

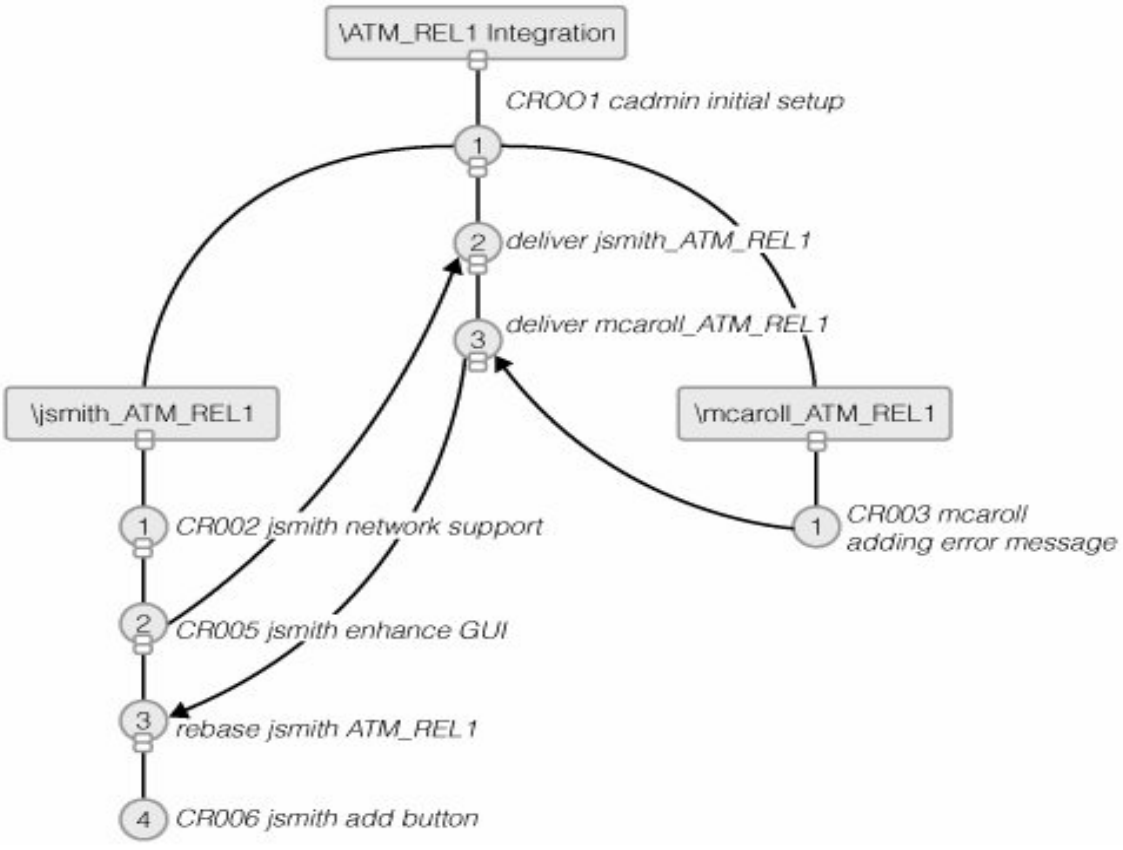

Figura 2- 5: Esquema de trabajo basado en Streams, actividades y baselines. Extraído de Kroll y Maclsaac (2006)

Explicación: *La figura muestra un* ***stream*** *de integración y* ***streams*** *de dos desarrolladores (Jsmith y Mcarroll). Jsmith completa los cambios CR002 y CR005, posterior a esto los entrega para integrarlos. Mientras que Mcarroll realiza de forma paralela el cambio CR003 (es un cambio adicional y no contradictorio con los cambios CR002 y CR005), posteriormente Mcarroll entrega el cambio y lo integra. Finalmente Jsmith realiza un nuevo* ***baseline*** *y continúa incorporando funcionalidades.*

2.1.3.7 Gestionar y analizar los riesgos

Los riesgos usualmente surgen de forma sutil, mediante pequeñas inestabilidades, por eso es importante monitorearlos, priorizarlos y evaluarlos constantemente.

Los pasos establecidos para su gestión son:

i. Identificarlos
ii. Evaluar su probabilidad de ocurrencia
iii. Estimar su impacto
iv. Establecer un plan de contingencia

Para estimar el impacto del riesgo se deben considerar los siguientes factores:

- ***Naturaleza***: Indica los problemas potenciales que pueden ocurrir si se produce el riesgo.

- ***Alcance***: Combina la severidad y la proporción del proyecto que se verá afectada.

- ***Temporización***: Considera cuándo puede ocurrir y cuánto tiempo pueden durar los efectos negativos.

Para a***nalizar los riesgos*** se deben considerar los siguientes aspectos:

- Evitar el riesgo
- Supervisar el riesgo
- Gestionar el riesgo
- Establecer planes de contingencia

Generalmente el ***análisis del riesgos*** requiere una cantidad significativa de esfuerzo, sin embargo, debido a la importancia que tiene esta práctica ciertos autores recomiendan enfocarse en el ***20% de los riesgos conocidos que sean más peligrosos***, porque ellos explicarían el 80% del riesgo total del proyecto si cumplen la regla 80/20.

2.1.3.8 Los desarrolladores deben probar su propio código

Diversos desarrolladores piensan que su trabajo sólo consiste en escribir el código mientras que el *equipo de pruebas* o ***testers***, debe encontrar los defectos. Empíricamente se ha demostrado que esta actitud entorpece la agilidad del ***proceso de desarrollo de software***, generalmente requiriendo esfuerzos adicionales (Kroll y Maclsaac 2006). En cambio, cuando los desarrolladores adoptan una prevención simple de defectos y técnicas prematuras para su detección, se producen mejoras inmediatas en la calidad del código y la productividad, porque así se evitan los defectos y permite que estos se reparen antes de que pasen a los ***testers***.

Las técnicas que implican esta práctica involucran ***técnicas defensivas de codificación*** y ***buenas prácticas de pruebas de desarrollo***, ambas ayudan a producir código en menos tiempo y con pocos defectos. Las técnicas principales según la ***fase del ciclo de desarrollo*** son:

- ***Antes de escribir el código***, considerar:

    - Diseño.
    - Instalación.
    - Uso.
    - Sistema Operativo.
    - Bases de datos.
    - Capacidad de prueba.

- ***Mientras se escribe el código***:

    - Técnicas defensivas de codificación.
    - Conocimiento pleno del cliente.
    - Programación en pares.
    - Inspección de código.

- ***Cuando se prueba el código*:**

    - Técnicas defensivas de testeo.
    - Código andamio (*scaffolding code*)[1],
    - Desarrollo guiado por pruebas (*test-driven development*).
    - Depurar (*debuggers*) a nivel de fuente.

[1] Código temporal y sin utilidad en la aplicación, se ocupa para asegurar que el código elaborado realiza las funciones que debe realizar.

En la medida que se encuentren los defectos, apenas se introduzcan, se produce un producto de mejor calidad y el proceso de elaboración es más eficiente. Con respecto a la adopción de estas prácticas Kroll y Maclsaac (2006) señalan que éstos son independientes de cualquier ***proceso de software*** y que su adopción es generalmente simple.

2.1.3.9 Medir el progreso objetivamente

Según Kroll y Maclsaac (2006), medir objetivamente el progreso de un proyecto de software es dificultoso por los siguientes desafíos:

- Las tareas varían ampliamente.
- Los eventos que no se han planificados usualmente retrasan los proyectos.
- Existe presión por informar un progreso positivo y estimaciones optimistas.
- Existen factores que compiten. Por ejemplo, menor tiempo de entrega sacrificando calidad o incrementando los costos.
- Las diferentes técnicas para medir el progreso tienen sus fortalezas y limitaciones.

Pese a lo anterior, la adecuada realización de esta práctica ayuda a gestionar los costos, plazos, la gente y las prioridades. Siendo una práctica fundamental para el *jefe de proyecto*, quien al desconocer objetivamente el progreso, no sabe realmente si el proyecto se dirige hacia el éxito o al fracaso. Un progreso escaso indica problemas que deben ser solucionados; por ejemplo, problemas de *personal* o de *procesos*.

Por otro lado, medir el progreso y realizar estimaciones son dos actividades estrechamente vinculadas; dado que afirmar que se lleva un 10% de progreso señala implícitamente que falta sólo un 90%. Sin embargo, realizar estimaciones sujetas a la realidad es dificultoso para estos proyectos.

Algunos métodos utilizados para medir el progreso objetivamente son:

- Medir el progreso a través de las funcionalidades de los usuarios, seguido de la documentación y el material que enseñe a utilizarlo.

- Basarse en la experiencia adquirida en tareas similares para ajustar las estimaciones de tareas futuras.

- No medir el progreso del proyecto sin funcionalidades, sino medir el progreso de una fase para estimar cuando concluirá ésta.

- Visualizar el progreso utilizando gráficas de avance para la iteración y el proyecto. El progreso de la actividades que generalmente se miden y visualizan son:

    - ***El progreso de las pruebas***: Las pruebas ejecutadas y pruebas pasadas, como también valorizar la efectividad de las pruebas al término de cada iteración.

    - ***El progreso de la reparación de defectos***: Los defectos deben ser clasificados según la severidad de los problemas que le puedan producir al usuario final.

2.1.3.10 Planificar, estimar y ejecutar el proyecto en iteraciones

Las aplicaciones de software son tan complejas que generalmente no es posible definir de forma secuencial y correcta los requisitos, arquitectura, codificación y pruebas. En cambio, cuando se divide el proyecto en una serie de iteraciones con plazos acotados, se pueden entregar funcionalidades incrementalmente. La ventaja de esta estrategia radica en que al final de cada iteración los *stakeholders* pueden evaluar y entregar *feedback* rápido y a tiempo, lo que permite solucionar problemas y realizar mejoras a un costo menor.

Una iteración debe abarcar todas las actividades necesarias para realizar una *aplicación estable del producto* o ***release***. Las fases mínimas requeridas para elaborar una funcionalidad son: ***Requisitos***, ***Análisis y Diseño***, ***Implementación***, y ***Testeo***.

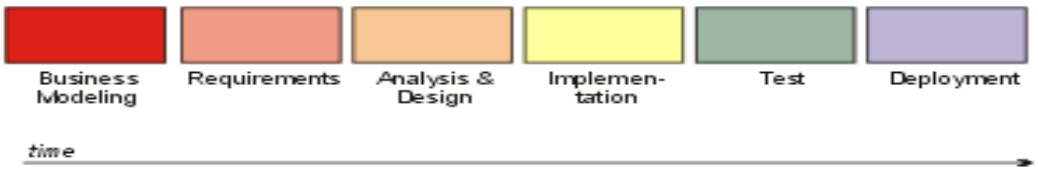

Figura 2- 6: Representación de las fases de una iteración. Extraído de RUP (2007)

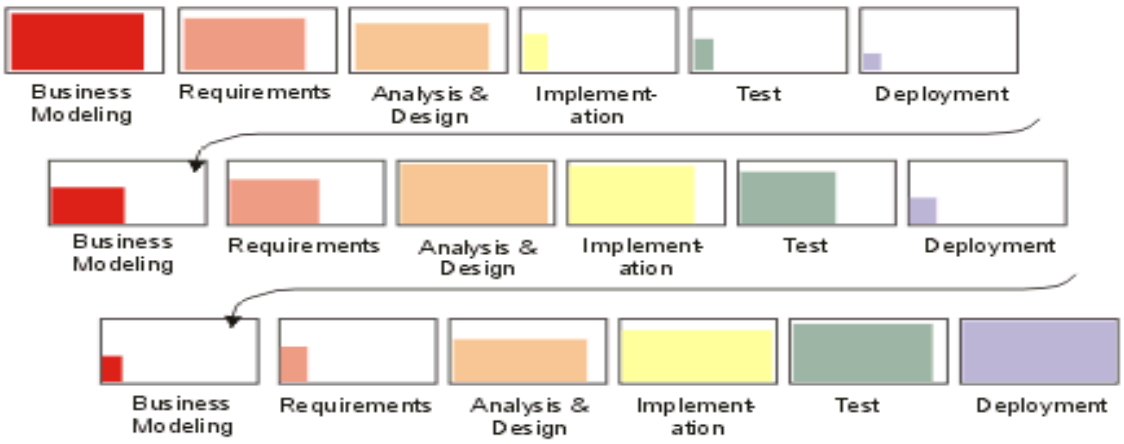

Figura 2- 7: Representación de un proyecto que se realizó en tres iteraciones. La porción de color de cada recuadro representa la intensidad de trabajo en cada fase. Extraído de RUP (2007)

Larman (2004) señala las ventajas de utilizar esta práctica:

- Incrementa la probabilidad de construir una aplicación que aborde las necesidades de los usuarios, ya que faculta la validación temprana de las funcionalidades claves y también permite la incorporación de nuevas cuando el proyecto está avanzado. Enfrentando así el problema señalado por *Standish Group International* (2003) al concluir que en diversos sistemas de información los usuarios nunca utilizaron el 45% de las funcionalidades y que el 19% sólo se empleó rara vez.

- Se ha comprobado empíricamente que la integración continua disminuye los costos y tiempos, dado que una única integración al final del proyecto produce retrabajo.

- Generalmente el riesgo es descubierto y encausado en las iteraciones iniciales.

- Se mejora la habilidad para trabajar efectivamente, por cuanto al producirse satisfactoriamente las primeras iteraciones se garantiza que se poseen las herramientas, habilidades y estructura organizacional adecuada para ejecutar efectivamente el proyecto.

- Los *stakeholders* pueden realizar cambios tácticos al producto.

- Es posible identificar porciones de código que se podrían reutilizar en proyectos actuales o futuros.

- A medida que se encuentran los errores, se van corrigiendo.

- El *proceso de desarrollo* puede mejorarse en la medida que avance el proyecto.

Por su parte, Pressman (2002) agrega que una de las medidas más significativas para impedir los atrasos, consiste en emplear un ***modelo de proceso incremental***, lo que permite establecer una cantidad de funcionalidad crítica en una fecha determinada para que posteriormente se vayan incorporando las funcionalidades restantes.

2.1.3.11 Priorizar los requisitos a implementar

Esto contribuye a entregar valor y disminuir los riesgos. La dificultad de esta práctica radica en decidir qué requisitos se deben desarrollar primero, la decisión adecuada ayudará a maximizar el valor para los *stakeholders* y el negocio. Al respecto, se deben equilibrar las necesidades técnicas y del negocio, teniendo presente que una decisión inadecuada puede guiar la entrega de funcionalidades que no serán utilizadas o identificará problemas tardíamente, lo que puede redundar en atrasos o el mismo fracaso del proyecto.

Es así como, esta práctica propone enfrentar el problema priorizando los requisitos una vez que se hayan comprendido y clasificado.

La clasificación de los requisitos en categorías ayuda a los desarrolladores y a los analistas a tener una visión más clara. Grady (1997) propone la clasificación "**FURPS+**" que está dada por ***Funcionalidad***, ***Usabilidad***, ***Fiabilidad*** (*Reliability*)**,** ***Ejecución*** (*Performance*) y ***Sustentabilidad*** (*Supportability*), mientras que el símbolo "+" hace referencia al ***diseño***, la ***implementación***, las ***interfaces***, el ***cuerpo*** y otras restricciones.

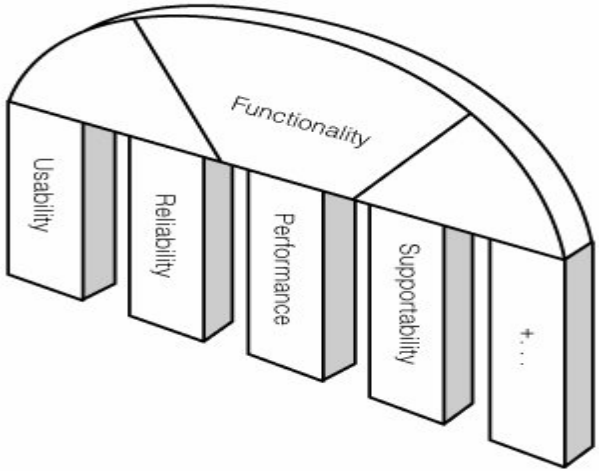

Figura 2- 8: Representa la clasificación "FURPS+" propuesta por Grady (1997)

Sin embargo, para aplicar esta práctica se deben:

- Entender las dinámicas del proyecto y de los *stakeholders*. Para esto, es necesario clasificar a los *stakeholders*, por ejemplo en:

  - ***Patrocinador del proyecto***, su función consiste en determinar los requisitos que tendrán mayor prioridad para satisfacer las necesidades del cliente.

  - ***Clientes***, tienen la función de satisfacer a los usuarios y son los que contratan los servicios.

  - ***Usuarios***, son quienes finalmente utilizarán el sistema.

  - ***Organización cliente.***

  - ***Organización desarrolladora***.

  - ***Otros Stakeholders*** como *instructores*, *instaladores*, *testers*, entre otros.

- Asignar *roles* para resguardar y apoyar las metas claves.

- Incrementalmente identificar y priorizar los entregables, para esto se debe proporcionar la menor cantidad de funcionalidades en el primer ***release*** utilizable y posteriormente se deben ir incorporando las nuevas funcionalidades en los siguientes ***release***.

- Identificar y clasificar los requisitos:

  - Identificando las funcionalidades claves.

  - Identificando los requisitos de soporte (*Supporting Requirements*), a veces llamados requisitos técnicos, no-funcionales, de calidad, o suplementales.

  - Estabilizando y clarificando los requisitos críticos.

  - Enfrentando los requisitos riesgosos.

  - Identificando y priorizando los escenarios.

  - Capturando los atributos de los requisitos, algunos atributos son:

    - Importancia para los *stakeholders*.
    - Esfuerzo estimado.
    - Impacto en la arquitectura.
    - Estabilidad.
    - Riesgo.

2.1.3.12 Utilizar la arquitectura de software para minimizar los requisitos de comunicación y colaboración

Esta práctica plantea que los proyectos deben organizarse en torno a la arquitectura de software para *facilitar la comunicación* y producir una *colaboración efectiva*. El problema se produce cuando el tamaño del proyecto es mayor, porque la comunicación entre los miembros se vuelve más compleja, lo que aumenta su costo y dificulta que los miembros conozcan la globalidad del proyecto. En consecuencia, se deben minimizar las *necesidades de comunicación* y para esto se plantea organizar al equipo en torno a la arquitectura porque permite reducir significativamente los ***canales de comunicación*** (Kroll y Maclsaac 2006).

Generalmente la comunicación ***presencial*** es la más efectiva, excepto en los proyectos donde existen diversos ***canales de comunicación***, los que crecen geométricamente según el tamaño del equipo. El número de ***canales de comunicación*** en un equipo de ***n*** individuos, está dado por la siguiente relación:

$$\textbf{Canales de comunicación} = \frac{n*(n-1)}{2}$$

Figura 2- 9: *Relación para obtener los calanes de comunicación*

Uno de los beneficios de una arquitectura de software robusta, consiste en poder dividir claramente las responsabilidades del sistema en *subsistemas bien definidos*, con *interfaces bien definidas*. De esta manera, al organizarse en función de la arquitectura se reduce el riesgo de trabajo duplicado.

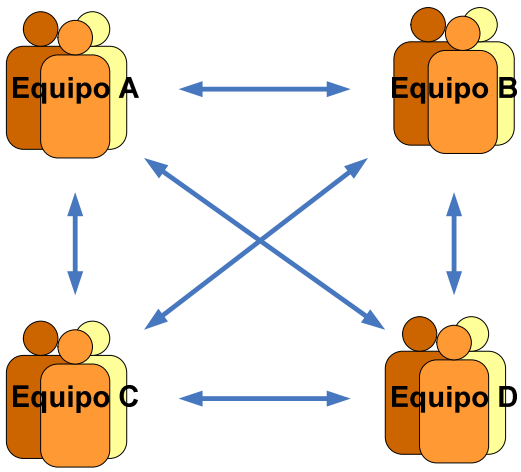

Figura 2- 10: Canales de comunicación entre los miembros del equipo cuando no se organizan en función de la arquitectura

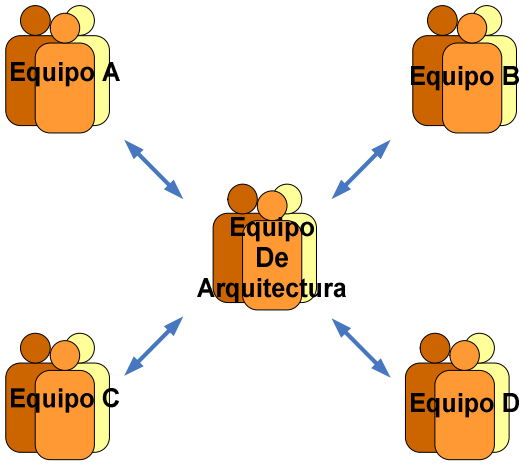

Figura 2- 11: Muestra los canales de comunicación cuando el equipo se organiza en función de la arquitectura

En cuanto a los problemas relativos a la interacción de los subsistemas, éstos deben resolverse por el equipo de arquitectos, quienes son los que proveen las interfaces entre estos.

Para su adecuada aplicación, los arquitectos deben trabajar con todo los integrantes del proyecto para garantizar que se detallen los requisitos fundamentales que afecten la *arquitectura*, *diseño*, *implementación* y *pruebas* (Ambler y Jefries 2002).

Para determinar la estructura del equipo desarrollador, por su parte, existen diversas formas para dividir una aplicación en subsistemas, por ejemplo; en torno a las funciones o a las características horizontales y verticales del negocio.

Es así como cada subsistema debe tener algún equipo responsable que comprenda las restricciones de la arquitectura, las interfaces entre subsistemas, las elecciones tecnológicas, los *patrones arquitectónicos* que deben incorporarse y los *requisitos arquitectónicos*. Por lo general, estos mismos equipos deben realizar cambios o ajustes arquitectónicos que afecten el comportamiento y/o la interfaz de los subsistemas, es por esto que los equipos encargados deben trabajar en conjunto con el *equipo de arquitectura del proyecto*, para garantizar que los cambios requeridos sean aceptables no sólo a nivel del subsistema sino también en su totalidad. Cuando los cambios son aceptados, deben comunicarse a todos los miembros del proyecto. Kroll y Maclsaac (2006) recalcan que el equipo de arquitectura y el equipo responsable por el subsistema no deben trabajar de forma aislada, sino mediante estrecha colaboración para garantizar una correcta elaboración.

En relación a promover la propiedad colectiva, donde cualquier miembro esté facultado para realizar cambios en el código o en la modelación, como se señala en *XP* (Beck y Andrew 2004) y *Agile Modeling* (Ambler y Jefries 2002), Kroll y Maclsaac (2006) expresan que esta práctica funciona sólo para proyectos pequeños de limitada complejidad, donde los integrantes pueden comunicarse fácilmente para entender que cambios han sido hechos y el porqué; o por equipos ligeramente mayores, pero que poseen miembros muy talentosos.

Para la mayoría de los proyectos y especialmente los mayores, es necesario asegurar que los responsables de los subsistemas serán notificados ante cualquier cambio, de otra manera se corre el riesgo de introducir problemas inesperados en el código. Al respecto se recomiendan que en los proyectos largos y/o complejos los miembros encargados de los subsistemas sean los que efectúen los cambios.

2.1.3.13 Utilizar un modelo de equipo Descentralizado Controlado

Existen diversos modos de organizar a un equipo desarrollador, ciertos autores afirman que la mejor forma es a través de ***sub-equipos***, donde a cada uno se les asignan una o más tareas funcionales. En este tipo de organización la coordinación se realiza mediante la interacción entre el *jefe del proyecto* y el *equipo desarrollador*.

Mantei (1981), a su vez, describe siete factores que deben considerarse al planificar el organigrama de los equipos desarrollares de software, estos son los siguientes:

- Dificultad del problema a resolver
- Tamaño del programa resultante, en líneas de código o puntos de función
- Tiempo en que trabajarán juntos los miembros del equipo y tiempo del ciclo de vida del proyecto
- Grado de descomposición de los problemas
- Calidad requerida y fiabilidad del sistema que se va a construir

- Rigidez de la fecha de entrega

- Grado de comunicación requerido para el proyecto

Un factor crítico en el *rendimiento* de los equipos es la *cantidad de comunicación requerida*, dado que en condiciones normales estos dos factores son inversamente proporcionales (Pressman 2002).

Es así como, Mantei (1981) sugiere tres modelos para organizar un equipo genérico, sin embargo, el más aceptado es el modelo ***Descentralizado Controlado (DC)***. En éste, existe un *jefe principal* y varios *jefes secundarios*. El principal se encarga de coordinar las tareas específicas, mientras que los secundarios se responsabilizan por la implementación de las *sub-tareas*. En este tipo de organización la resolución de problemas sigue siendo una actividad de grupo, sin embargo, el *jefe de proyecto* es el que reparte la implementación de las soluciones. La comunicación entre los *sub-equipos* es horizontal, mientras que la comunicación vertical se realiza de forma jerárquica.

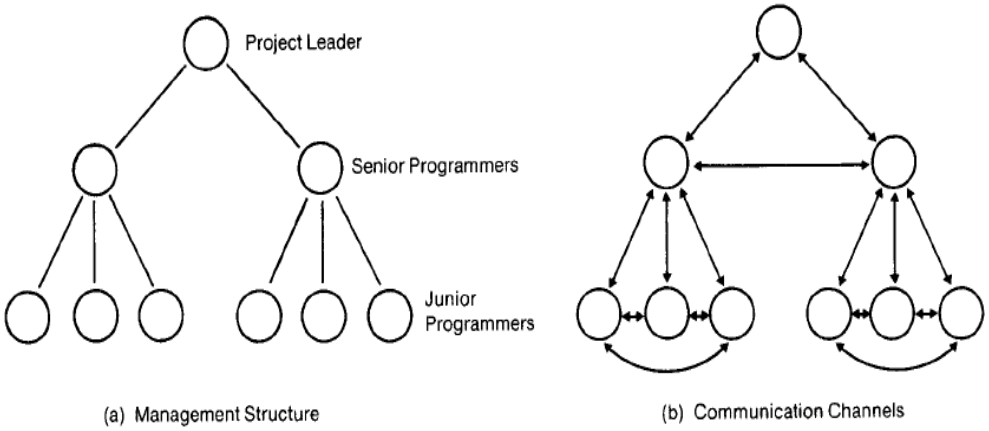

Figura 2- 12: Modelo de equipo Descentralizado Controlado propuesto por Mantei (1981)

2.1.3.14 Utilizar un paradigma organizacional Sincronizado

Por su parte, Constantine (1993) sugiere cuatro paradigmas de organización según el tipo de software que se elaborará, estos son: ***Cerrados***, ***Aleatorios***, ***Abiertos*** y ***Sincronizados***.

- ***Cerrados***: Presentan una jerarquía tradicional de autoridad (sólo comunicación vertical). Estos equipos trabajan adecuadamente cuando se elabora software similar a otros desarrollos anteriores.

- ***Aleatorios***: Presentan una estructura libre que depende de la iniciativa individual de los miembros del equipo. Es el más apto cuando se requiere innovación o avances tecnológicos.

- ***Abiertos***: El trabajo se desarrolla en colaboración y con alta comunicación, donde las decisiones se toman de manera consensuada. Es adecuado cuando se enfrentan problemas complejos.

- ***Sincronizados***: Se basan en dividir el problema de forma natural para minimizar la comunicación entre los miembros del equipo. Este paradigma es el más apto para los desarrollos distribuidos.

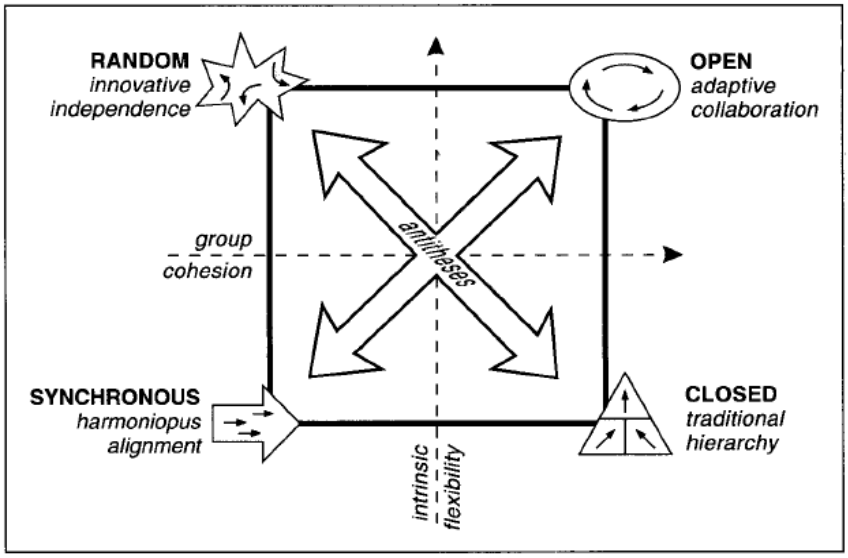

Figura 2- 13: Muestra la relación entre los distintos paradigmas organizacionales planteados por Constantine (1993)

2.1.3.15 Utilizar una adecuada cantidad y rigurosidad de procesos en el ciclo de vida del software

Elegir el tamaño adecuado de los procesos permite maximizar la productividad mientras se gana agilidad. Los proyectos que no actúan guiados por procesos son generalmente ineficientes. En cambio, los proyectos con muchos procesos pueden sufrir falta de creatividad y productividad, porque fuerzan la realización de artefactos y actividades que agregan escaso o nulo valor.

La estrategia planteada por Kroll y Maclsaac (2006) consiste en proveer directrices para justificar y elegir una adecuada cantidad de procesos, basándose en las necesidades del proyecto. Estas son:

- Enfocarse en el ***software ejecutable*** y no en los ***artefactos de soporte*** que no pertenezcan al proyecto, porque la mejor forma de mitigar el riesgo es mediante las pruebas que los *stakeholders* puedan efectuar al producto.
- Producir sólo artefactos que agreguen valor al equipo y la organización, y reducir su cantidad al cambiar el modo en que colaboran los integrantes del equipo, involucrando a los *stakeholders* adecuados para que orienten sobre que artefactos deben ser elaborados.

- Escalar los procesos a través de patrones como ***Eclipse Process Framework*** y ***Rational Method Componer***.

- Automatizar las tareas rutinarias.

- Mejorar los procesos continuamente a través de la valoración, porque al realizar un análisis retrospectivo se mejora la calidad del trabajo futuro.

Algunos factores que impactan en la cantidad adecuada de los procesos son:

Tabla 2-2: Factores que impactan la cantidad de procesos

Factores		
Tipo	**Menos disciplina**	**Mas disciplina**
Tamaño del proyecto	Pequeños	Grandes
Tamaño del equipo	Pocos integrantes	muchos integrantes
Desarrollo distribuido	equipo co-localizado	equipo distribuido
Requisitos regulatorios	Sin requisitos	Requisitos como (Sarbanes-Oxley, alimentación y drogadicción, administración, entre otros).
Requisitos contractuales	Sin requisitos	Relaciones contractuales
Cantidad de Stakeholders	Pocos	Muchos
Tipo de relación con los stakeholders	Simple	Compleja
Ciclo de vida de la aplicación	Corta	Larga
Fases del ciclo de vida	Fases iniciales	Fases finales

Por su parte Pressman (1999), define los siguientes criterios de selección para determinar el ***grado de rigor*** recomendado y establecer una ***red de tareas*** adecuadas al tipo de proyecto:

- Tamaño del proyecto.
- Número potencial de usuarios.
- Importancia de la misión.
- Antigüedad de la aplicación.
- Estabilidad de los requisitos.
- Facilidad de comunicación cliente/desarrollador.
- Madurez de la tecnología aplicable.
- Limitaciones del rendimiento.
- Características embebidas/no embebidas.
- Personal del proyecto.
- Factores de reingeniería.

2.1.3.16 Utilizar una arquitectura basada en componentes y servicios

Estas arquitecturas ayudan a construir software más fácil de comprender, cambiar y reutilizar, porque los ***componentes*** y los ***servicios*** son unidades relativamente independientes que se pueden ensamblar para construir sistemas mayores.

Algunos beneficios que proporcionan las arquitecturas basadas en ***componentes*** y ***servicios*** son:

- Facilitan la comprensión, porque definen una capacidad consistente y ocultan la complejidad interna.

- Facilitan el reemplazo, porque pueden ser sustituidos individualmente o modificados sin afectar a los demás elementos del sistema.

- Pueden ser reutilizados de más fácilmente, porque tienen la capacidad de ensamblarse con otros sistemas.

Según Kroll y Maclsaac (2006) el término ***componente*** ha sido mal utilizado dado que generalmente se ha denominado de esta manera a cualquier parte constituyente. Por su parte, RUP 7 define el término ***componente*** como una pieza de un sistema que no es trivial, cumple cabalmente una función determinada en una arquitectura bien definida, es escasamente dependiente y puede ser reemplazable (IBM Corporation 2005). Los ***componentes*** poseen puntos de conexión y se encargan de funciones específicas. Estas características permiten el reemplazo de una pieza antigua por una nueva, incluso si ésta hubiese sido construida por una organización distinta. Es así como un ***cliente del componente*** sólo dependerá de su interfaz para poder conectarse con el ***componente***.

Los ***servicios*** tienen diversas similitudes con los ***componentes***. Sin embargo, la principal diferencia radica en que las capacidades de los servicios se publican en un directorio para que los ***clientes de los servicios*** puedan identificar su interfaz. Esto permite una composición flexible de nuevas capacidades desde una holgada asociación de ***servicios***. En cambio, el término ***componente de servicio*** o ***service component*** señala la implementación de uno o más ***servicios***.

Finalmente, esta práctica entrega una visión general para identificar y especificar los ***componentes*** y ***servicios***. Las prácticas propuestas son:

- Identificarlos modelando globalmente el negocio y encapsulando sus reglas o procesos.

- Reconocerlos a partir de los activos existentes, porque pueden existir recursos reutilizables, por ejemplo bases de datos y sistemas existentes.

- Utilizar una ***arquitectura de capas***, lo que permite clasificar la funcionalidad en ***capas*** haciendo más fácil la identificación de los ***componentes***, al precisar las dependencias y responsabilidades de éstos. Este tipo de arquitectura habilita que las ***capas superiores*** utilicen ***servicios*** de las ***capas inferiores***. La separación de capas según los intereses facilita la identificación de los ***componentes*** y la posibilidad de efectuar cambios (IBM Corporation 2005). La ***arquitectura en capas*** también se aplica a la ***arquitectura orientada a servicios*** **(SOA)**. Aunque frecuentemente la discusión se enfoca en la interacción ***peer-to-peer*** dentro de las ***capas superiores***. Los ***servicios*** también pueden beneficiarse de las ***capas*** para la reutilización y reducción de dependencias (Greenfield et al. 2004; y Stojanovic y Dahanayake 2005).

- Extraer el factor común de comportamiento, porque diversas ***clases*** tienen requisitos similares que pueden ser factorizados y cambiados a otros ***componentes***. Por ejemplo, los mecanismos de almacenamiento continuo, comunicación entre procesos, seguridad, transacciones, informe de errores y conversión de formatos. Otro enfoque considera aislar los comportamientos que pueden ser reutilizados en sistemas futuros.

- Considerar cambios probables, aislando áreas potenciales para facilitar los cambios futuros.

- Distinguir ***especificación*** y ***realización***. La ***especificación*** sirve como un contrato que define todo lo que el cliente necesita saber para utilizar el ***componente***. En cambio, la ***realización*** es el diseño interno detallado para guiar la implementación. Lo mismo se aplica para los ***servicios***.

- Proveer el nivel correcto de interconexión, porque no todos los ***componentes*** se deben exhibir como ***servicios*** ya que la flexibilidad que proporcionan los ***servicios*** incrementan sus costos.

- Desarrollar ***componentes*** iterativamente, porque la reconstrucción de ***componentes*** una vez que hayan sido implementados definitivamente puede ser dificultoso y costoso. Se sugiere validar el diseño y las interfaces de los ***componentes*** al implementar escenarios básicos que confirmen la integración de éstos. Por otro lado, se pueden incorporar funcionalidades en cada iteración, además de refinar y expandir las interfaces. Al desarrollar los ***componentes*** de forma iterativa se pueden encontrar problemas, como por ejemplo, de asociación o ejecución. Así estos problemas se pueden corregir sin producir un gran impacto en el código; una de las técnicas sugeridas es el ***refactoring***.

2.2 En los proyectos distribuidos de software

En los capítulos anteriores se proporcionaron un conjunto de prácticas y técnicas para enfrentar los problemas que atañen al ***desarrollo tradicional de software***. En cambio, en este capítulo se plantean los principales problemas y requisitos en los ***desarrollos distribuidos de software***.

2.2.1 Problemas y requisitos del desarrollo distribuido de software

Los problemas presentes en el ***desarrollo tradicional*** se incrementan por las dificultades de ***comunicación***, ***coordinación*** y ***mecanismos de control***, que requieren los ***desarrollos distribuidos*** (Rafii 1995; Carmel 1999; Karolak 1999; Van Fenema y Kumar 2000; Espinosa y Carmel 2003; Herbsleb y Mockus 2003; Kotlarsky 2005).

Los problemas fundamentales del escenario distribuido surgen principalmente por tres características, estas son:

- ***Distancia***, porque reduce la intensidad de la comunicación y la posibilidad de reuniones presénciales para enfrentar los problemas que surjan (Kotlarsky 2005).

- ***Diferencias culturales***, porque produce malos entendidos y conflictos, como consecuencia de las diferencias de lenguaje, valores, suposiciones implícitas y hábitos de comunicación y trabajo (Baumard 1999).

- ***Diferencias horarias***, porque limita la posibilidad de una colaboración en tiempo real cuando los horarios de trabajo no se superponen (Kotlarsky 2005).

Los problemas más recurrentes en la literatura son:

- Fallas en los mecanismos tradicionales de control y coordinación (Carmel 1999; Van Fenema 2002; Cheng et al. 2004; Berenbach 2004; Bellagio y Milligan 2005):
    - Grado de dispersión del equipo de trabajo, estructura organizacional y malas prácticas de gestión (Berenbach 2004).
    - Dificultad para establecer una colaboración efectiva debido a la incompatibilidad en la infraestructura TIC y a las diferencias de habilidades y competencias de los miembros del equipo para trabajar con herramientas y tecnologías distintas (Van Fenema 2002; Sarker y Sahay 2004).
    - Falta de cohesión del equipo y falta de motivación para colaborar, producto del decaimiento de la moral y la confianza (Jarvenpaa y Leidner 1998; Carmel 1999; Karolak 1999).
    - Dificultades técnicas y tecnológicas (Bellagio y Milligan 2005).
- Pérdidas en la riqueza comunicacional (Carmel 1999; Van Fenema 2002; Bellagio y Milligan 2005):
    - Falta de comprensión de la contra parte (Orlikowski 2002).
    - Barreras de lenguaje y competencias diferentes (Saber y Sahay 2003).
    - Malos entendidos causados por diferencias culturales, diferentes estilos de conversación y distintas interpretaciones subjetivas (Battin et al 2001; Olson y Olson 2004).

- Falta de comunicación informal e interpersonal (Herbsleb y Grinter 1999; Van Fenema 2002).

- Actitud tímida y pasiva producida por la falta de una proximidad física (Berenbach 2004).

- Dificultades para trabajar en diferentes zonas horarias (Karolak 1999; Kobitzsch et al. 2001).

- Retrasos en el trabajo colaborativo entre los sitios remotos por respuestas poco claras y *feedback* insuficiente. Causado por las diferencias en los horarios de trabajo y por las distintas interpretaciones (Jarvenpaa y Leidner 1998; Herbsleb et al 2000).

Sutherland (2007) precisa un conjunto de categorías para encasillar los principales problemas presentes en los desarrollos distribuidos, las categorías son:

- ***Estratégicos***: Dificultades para mantener las mejores prácticas y apalancar los recursos disponibles, porque consumen demasiado tiempo.

- ***Comunicacionales***: Carencia de canales efectivos de comunicación.

- ***Culturales***: Conflictos en los comportamientos, procesos y tecnologías.

- ***Técnicos***: Incompatibilidad en los formatos de los datos, esquemas y estándares.

- ***Seguridad***: Dificultades para garantizar una transferencia electrónica confidencial y segura.

Mientras que Bellagio y Milligan (2005) platean los siguientes requisitos tecnológicos esenciales para logar un exitoso ***desarrollo distribuido de software***:

- *Controlar*, *medir* y *dirigir* lo que está sucediendo en el *proceso de desarrollo*.

- *Identificar* y *controlar* la *documentación*, *el código* y las *interfaces*.

- Recolectar datos que apoyen la gestión.

- Armar el sistema final utilizando *listas* que señalen las *partes* y sus *versiones*.

- Tener *acceso rápido y fácil para encontrar y extraer artefactos*, tanto del proyecto actual como de anteriores.

- *Evitar la pérdida y la falta de identificación de las versiones de los artefactos* mediante la utilización de un método efectivo para organizarlos y localizarlos.

- Los repositorios deben *tolerar las fallas*, *ser escalables*, *distribuibles* y *replicables*.

- *Control de acceso* para modificar o visualizar los artefactos.

- Registrar información que permita: la corrección rápida de errores, evitar la incorporación de nuevos errores, y prescindir de cierta comunicación presencial. Por lo cual se debe registrar la siguiente información:

    - ¿Qué fue modificado?
    - ¿Dónde se modificó?
    - ¿Quién produjo los cambios?
    - ¿Cuándo fueron hechos?

- ¿Por qué fueron hechos?

- Organizar los archivos y directorios para:

  - Reducir la complejidad de su manejo
  - Facilitar la identificación de su calidad
  - Facilitar el intercambio y la reutilización de componentes
  - Ayudar a mantener la arquitectura del software
  - Poder definirlos, controlarlos y gestionarlos jerárquicamente

- Manejar la velocidad de cambio de la elaboración del software.

- Reunir en un mismo lugar y tiempo las versiones adecuadas de los *componentes* para facilitar su conexión al construir la versión final del software.

- Reproducir los *releases* anteriores.

- Presentar informes de los avances de desarrollo.

- Registrar y rastrear las *solicitudes de cambio* para verificar su progreso.

- Priorizar y establecer fechas para materializar las *solicitudes de cambio*.

- Trazabilidad para vincular y hacer seguimiento de los *requisitos*, *casos de prueba*, *planificación* y *artefactos*.

- Tener un modelo de decisión para aceptar o rechazar las *solicitudes de cambio*.

- Implementar consistentemente los *cambios* para evitar que los integrantes incurran en errores.

- Facilitar el trabajo simultáneo y la recolección de información clave que agilice el proceso de integración.

- Garantizar la consistencia de la *configuración*, al margen del *ambiente de trabajo* y *construcción*.

- Vincular la gestión del proyecto con la *gestión de configuración* y la *gestión de solicitudes de cambio*.

- Proporcionar *ambientes de trabajo* estables.

- Sincronizar los cambios que realizan periódicamente los miembros del equipo.

- Posibilitar el trabajo aislado.

- Aumentar el nivel de abstracción de la comunicación desde archivos a ***actividades***, porque es una forma de comunicación más similar a la efectuada por las personas.

- Aislar los cambios disruptivos para no afectar la integridad del proyecto y de los otros miembros.

- Proporcionar información actualizada y acceso a los últimos artefactos elaborados en el proyecto.

- Poder modificar, integrar y fusionar los cambios realizados, en un tiempo apropiado y minimizando la sobrecarga de trabajo.

- Registrar, cómo se construyó el software y las versiones de los componentes que lo conforman.

- Mantener el control sin que influya la cantidad de cambios realizados en el software y productos de trabajo.

- El *jefe de proyecto* debe tener acceso rápido y actualizado de las actividades asignadas a cada integrante.

- Posibilitar el desarrollo paralelo y los cambios concurrentes de los artefactos.

Para Bellagio y Milligan (2005), estos requisitos son satisfechos por un conjunto de herramientas y procesos adecuados, los que permiten minimizar el riesgo inherente al ***desarrollo distribuido de software***.

2.2.2 Prácticas planteadas por empresas con experiencia en proyectos distribuidos

2.2.2.1 Prácticas sugeridas por Siemens Corporate Research

Para mitigar los problemas organizacionales y de gestión Berenbach (2004) sugiere las siguientes prácticas:

- Minimizar el grado de dispersión geográfica, dado que la probabilidad de fracaso está dada por la siguiente expresión probabilística:

$$P_{fd} = f(P_r, P_a, P_i) \ U \ P_{fc}$$

Figura 2- 14: Variables en la probabilidad de fracaso de un proyecto distribuido

Donde:

P_{fd} = Probabilidad de fracaso en los proyectos distribuidos.

P_{fc} = Probabilidad de fracaso en los proyectos co-localizados.

P_r = Probabilidad de fracaso cuando los Requisitos se realizan de forma distribuida.

P_a = Probabilidad de fracaso cuando el Análisis se efectúa en forma distribuida.

P_i = Probabilidad de fracaso cuando la Implementación se efectúa de forma distribuida.

Este planteamiento señala que el riesgo asociado a la *dispersión geográfica* del proyecto se maximiza cuando el proyecto está 100% distribuido, porque los integrantes no tienen la posibilidad de reunirse presencialmente lo que afecta la *comunicación*, *coordinación* y *control*.

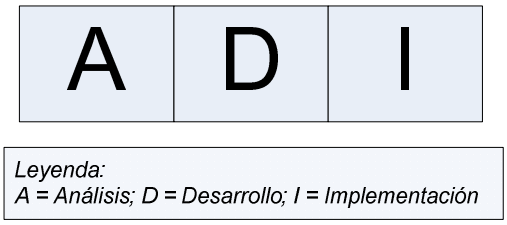

Figura 2- 15: Representación de un desarrollo sin dispersión, también llamado desarrollo co-localizado

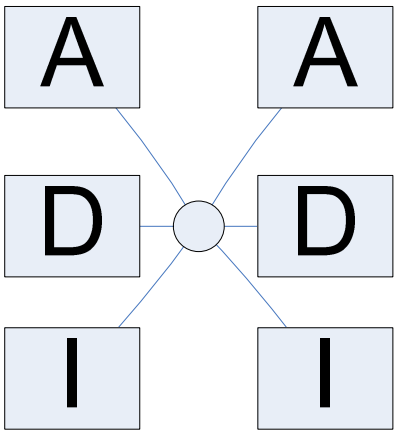

Figura 2- 16: Representación de un desarrollo 100% distribuido

- Mantener un liderazgo enérgico a nivel de jefe de proyecto y líder técnico.
- Construir una relación presencial antes de iniciar el proyecto.
- Establecer una línea clara de dirección.
- Definir consistentemente los roles y sus responsabilidades.
- Establecer un proceso de ***ingeniería de requisitos*** bien definido y comprendido.

- Realizar reuniones colaborativas frecuentemente, utilizando herramientas web.

- Establecer responsabilidades comunicacionales en cada sitio.

- Fijar políticas efectivas de ***Gestión del Cambio***.

2.2.2.2 Prácticas utilizadas por SirsiDynix

Por su parte, Sutherland (2007) señala un conjunto de prácticas denominadas ***Distributed Scrum***, según este autor éstas serían las responsables del éxito logrado en el proyecto ***Horizon 8.0*** elaborado por la empresa ***SirsiDynix*** (56 desarrolladores distribuidos entre EEUU, Canadá y Rusia). La productividad alcanzada en este proyecto fue de 15,3 ***puntos de función mensual por desarrollador*** y las prácticas que permitieron alcanzar dicha productividad fueron:

- Reuniones diarias entre el ***jefe de proyecto*** y los ***miembros de los sub-equipos remotos*** (mediante teleconferencia). Estas reuniones se denominaron *reuniones scrum* porque estaban basadas en las reuniones propuestas por la ***metodología Scrum***. Las reuniones tuvieron cuatro características principales: trataban un conjunto de *preguntas estándares,* en un periodo de tiempo reducido, a través de un *protocolo específico* e involucraban a todos los miembros que estaban ubicados remotamente. El *protocolo específico* consistió en enviar las respuestas por correo electrónico antes del inicio de cada reunión, lo que ayudó a derribar barreras culturales y diferencias de estilo. Mientras que las *preguntas estándares* fueron:

  - ¿Qué se logró ayer?
  - ¿Qué se realizará hoy?
  - ¿Qué impedimentos han surgido para lograr las metas?

- Cada sub-equipo realizó reuniones diarias, al comienzo de la jornada laboral, para coordinar las actividades periódicas.

- Los jefes de los sub-equipo tuvieron cada semana una reunión ***Scrum*** en un día determinado.

- El *cliente*, el *jefe de proyecto* y los *arquitectos* se situaron en una misma localidad.

- Los arquitectos tuvieron, una vez a la semana, reuniones ***Scrum*** presenciales.

- Para captar los nuevos requisitos y ajustar las funciones elaboradas el *jefe del proyecto* y los *analistas* tuvieron reuniones diarias con el cliente.

- Todos los *desarrolladores* tuvieron las mismas condiciones.

- Los equipos remotos integraron e implementaron internamente las prácticas de las metodologías ***XP*** y ***Scrum***, los más destacadas fueron:

    - Programación en parejas (*Pair programming*)
    - Refactorización (*Refactoring*)
    - Construcción continua (*Continuous Integration*)

- Para desarrollar las funcionalidades, cada vez que se pudo, se reutilizó código funcional para adaptarlo a las necesidades del cliente.

- Todos los miembros del equipo, proveniente de cualquier sitio, pudieron trabajar sobre todas las tareas del proyecto.

- Los requisitos fueron captados a través de ***historias de uso***, algunos ampliamente detalladas y otras no. Las dudas surgidas se despejaron a través de *reuniones scrum*, mensajería instantánea y/o a través de e-mails.

- Cada dos semanas se realizaron reuniones de planificación, donde se solicitaron las ***historias de uso***, posteriormente éstas se dividían en tareas y sub-tareas de desarrollo. Se denominó a los periodos entre cada reunión como ***Sprint***, al final de cada uno de estos plazos debían estar implementadas las funcionalidades solicitadas, ya sea a modo de prototipo o de forma definitiva.

- Los desarrolladores utilizaron ***pruebas unitarias*** para comprobar su código; el cliente utilizó sondeos manuales; y los ***equipos de pruebas*** automáticas y manuales.

- Se utilizó una herramienta para producir construcciones automáticas a cada hora, hechas desde un repositorio común; después de cada construcción el sistema envió un e-mail a los desarrolladores informando sobre este proceso. Sin embargo, sólo se solicitaron construcciones estables al fin de cada ***Sprint***.

- Se utilizó una herramienta especial para sustentar el repositorio y se proveyó de una efectiva ingeniería concurrente. Primero se utilizó la herramienta Open Source ***CVS***, sin embargo por su falta de soporte distribuido se emigró a la herramienta propietaria ***Perforce.***

- Se *midió el progreso del proyecto a través del seguimiento funcionalidades, tareas y actividades*. Para esto se utilizó la herramienta de gestión de proyectos ***Jira***. Esta herramienta le brindó a cada integrante una visión en tiempo real del estado de cada ***Sprint***.

2.2.2.3 Prácticas utilizadas por SAP, TCS y LeCroy

En cambio, Kotlarsky (2005) investigó las prácticas de gestión aplicadas por un conjunto de empresas, tanto las que tuvieron éxito en sus proyectos de ***desarrollo distribuido*** (*SAP, TCS y LeCroy*), como las que fracasaron en dichos emprendimientos (*BAAN*). Los resultados de esta investigación señalan un conjunto de 22 prácticas agrupadas en 5 factores claves de éxito, los que son: ***coordinación entre los sitios***; ***herramientas y tecnologías apropiadas***; ***vínculos sociales***; ***intercambio de conocimiento***; y ***gestión de componentes***.

2.2.2.3.1 Mejorar la coordinación y comunicación entre los sitios

- Incrementar la percepción sobre lo que ocurre en el proyecto, la compañía y los sitios remotos. Esta medida reduce la posibilidad de malos entendidos, conflictos y fallas de coordinación.

- Realizar una división de trabajo eficiente. Estas empresas han demostrado que una repartición de trabajo basada en las capacidades, aumenta la probabilidad de éxito cuando hay fuertes relaciones laborales y experiencia. En cambio, si dichas características son escasas se recomienda una división de trabajo basada en las funcionalidades. Este autor, recomienda, así mismo, considerar el impacto que produce cambiar las asignaciones de tareas establecidas, porque influye negativamente en la motivación de los integrantes del equipo, pudiendo afectar el éxito del proyecto.

- Facilitar la flexibilidad laboral, porque permite la superposición de los diferentes horarios laborales incrementando la comunicación en tiempo real.

- Facilitar el seguimiento de bugs y el desarrollo de tareas. Esta práctica se considera críticas para lograr proyectos exitosos.

- Flexibilizar la gestión del proyecto y establecer objetivos claros, realizando una planificación semanal flexible que permita la adaptación dinámica de las tareas diarias.

- Establecer comunicación sistemática, fijando reglas de estilo y frecuencia, reduciendo el riesgo de malos entendidos y conflictos, además relacionándose positivamente con una comunicación efectiva y la satisfacción personal de los integrantes.

2.2.2.3.2 Utilizar herramientas y tecnologías que apoyen la accesibilidad, colaboración y ejecución de los proyectos

- Herramientas para el desarrollo de software, que provean las siguiente capacidades:

    - Gestión automática para garantizar las interdependencias entre los componentes y los archivos relacionados, y una veloz actualización de los cambios (al menos cuatro veces al día).

    - Realización de pruebas automáticas a los componentes.

    - Estandarización de herramientas y procesos para los diferentes sitios. Lo anterior mediante: la utilización de herramientas similares; duplicando al cliente en cada sitio; y estandarizando los procesos de desarrollo.

- Centralizar el acceso a las herramientas a través de: la Web, bases de datos replicadas, ambientes de desarrollo individuales y repositorios centrales.

- Crear y distribuir tutoriales para mejorar la utilización de las herramientas y procesos.

- Según las necesidades del proyecto, desarrollar y/o adaptar las herramientas.

- Infraestructura TI, ésta incrementa la eficiencia y efectividad de la colaboración, y a su vez, aumenta la probabilidad de éxito del proyecto. Los requisitos de la infraestructura TI son:

  - Acceso rápido a la red de trabajo.

  - Recursos compartidos como bases de datos, servidores y repositorios del proyecto.

  - Acceso a las bases de datos y a las herramientas a través de la web.

  - Conectividad fácil y rápida entre los sitios remotos, utilizando herramientas colaborativas.

- Tecnologías colaborativas que permitan incrementar la efectividad de la comunicación y la satisfacción personal. Las empresas utilizaron las siguientes herramientas colaborativas:

  - Chat Online: para asuntos urgentes y/o puntuales.

- Teléfono y teleconferencias: para asuntos urgentes; actualizar la información del estado del proyecto; ayudar a la corrección de bugs; y para resolver malos entendidos y conflictos.

- Aplicaciones compartidas: para ayudar a reparar los bugs y proporcionar intercambio de conocimiento.

- Videoconferencias: para revisar el diseño y efectuar reuniones virtuales.

- E-mail: para comunicar el estado de las tareas de poca prioridad, aclarar asuntos y enviar código fuente y requisitos.

- Intranet: para colocar documentos internos.

2.2.2.3.3 Promover vínculos sociales entre los integrantes

- Construir relaciones sociales, facilita la buena comprensión entre los interlocutores y establece confianzas. Esta práctica se presenta en los proyectos exitosos, no así en los proyectos que han fracasado. Para construir buenas relaciones sociales se recomiendan los encuentros presenciales informales o formales (por ejemplo: capacitaciones o reuniones fuera del horario laboral).

- Incrementar la accesibilidad entre los miembros de los sub-equipos, para conocer la disponibilidad de los otros integrantes (día y hora) y determinar a quién contactar según una situación puntual.

- Crear y mantener una atmósfera de equipo, evita el aislamiento de los sitios remotos. Se relaciona directamente con la satisfacción del personal y su motivación a colaborar, reduciendo los conflictos y las fallas de coordinación.

- Facilitar y permitir la interacción de los miembros, porque se relaciona positivamente con una mejor comprensión e incremento de la confianza.

- Facilitar el intercambio de personal entre los sitios, para reducir las brechas culturales entre los integrantes.

2.2.2.3.4 Promover el intercambio de conocimiento

- Originar, documentar y divulgar conocimiento propio de cada equipo y promover las capacidades de cada miembro, se relaciona positivamente con los proyectos exitosos.

- Ampliar el conocimiento que se posee de los sub-equipos, porque aumenta la colaboración, comunicación y satisfacción de los integrantes. Esta práctica se aplica al instruir sobre la cultura regional y organizacional de los sub-equipos.

- Utilizar jefes de proyecto con amplia experiencia, incrementa la probabilidad de poseer una mejor conciencia de lo que acontece en los sitios remotos. Se relaciona positivamente con la efectividad y eficiencia del trabajo en equipo.

- Capacitar a los miembros del equipo y constantemente ir incorporando nuevas tecnologías, dado que motiva al personal, mantiene competitiva a la organización y facilita, mejora y agiliza la labor de los miembros.

2.2.2.3.5 Promover y facilitar la gestión de componentes

- Diseñar los ***componentes*** para que sean reutilizados. Esto incrementa el costo del primer producto, sin embargo, reduce significativamente el costo y los tiempos de los productos futuros que reutilicen parcial o totalmente los componentes.

- Invertir en proyectos de investigación y desarrollo. Se relaciona positivamente con la reutilización de componentes en productos futuros.

- Facilitar la reutilización, identificando las oportunidades que permitan emplear nuevamente los componentes y el conocimiento elaborado.

2.2.2.4 Prácticas sugeridas por IBM Rational

En este mismo capítulo se plantearon una serie de ***requisitos*** indicados por Bellagio y Milligan (2005) para la elaboración exitosa de los ***proyectos distribuidos de software***. Estos autores también señalan las medidas que satisfacen dichos requisitos, proponiendo las características necesarias que deben poseer las ***herramientas*** y algunos ***procesos de desarrollo***.

Es así como proponen controlar la evolución de los proyectos mediante herramientas de ***Software Configuration Management*** (***SCM***). Tal como se señaló en el segundo capítulo, esta disciplina de la *ingeniería en software* comprende las herramientas y las técnicas (*procesos* y *metodologías*) necesarias para que una organización gestione el cambio de sus ***activos de software***.

Según IEEE "***Standard for Software Configuration Management Plans***", la ***SCM*** constituye una buena práctica para todos los proyectos de software, tanto para las

fases de desarrollo, la construcción rápida de prototipos, o una mantención permanente. Ésta incrementa la fiabilidad y calidad del software al:

- Proveer una estructura que soporte todas las ***fases del ciclo de desarrollo*** al identificar y controlar la documentación, código, interfaces y bases de datos.

- Apoyar la metodología de desarrollo y mantención seleccionada, que se ha ajustado a los requisitos, estándares, políticas, organización y filosofía de gestión.

- Producir gestión e información de los productos, al relacionarlos con versiones oficiales denominadas ***baselines***, controlar los cambios, las pruebas, los ***releases***, las auditorias, entre otros.

Bellagio y Milligan (2005) sostienen que las partes de un sistema de software deben ser identificadas y deben tener un número de versión, además necesitan interfaces compatibles (algunas versiones pueden tener diferentes interfaces). Por tal motivo, se necesita una "***lista***" que contenga todas las partes involucradas, especificando que versión de cada parte se debe utilizar para armar el sistema final. Es así como, en el ambiente dinámico y cambiante del desarrollo de software, la disciplina de ***SCM*** permite que todos los ***componentes de un sistema de software*** sean ubicados en el mismo lugar y al mismo tiempo, permitiendo que se unan como sea requerido.

Por su parte ***IBM Rational*** (2005) señala las mejores prácticas para un adecuada ***SCM***, estas son:

- Identificar y almacenar los artefactos en repositorios seguros.

- Controlar y auditar los cambios de los artefactos.

- Organizar las versiones de los artefactos dentro de las versiones de los componentes.
- Crear baselines en cada hito del proyecto.
- Registrar y rastrear las solicitudes de cambio.
- Organizar e integrar un conjunto consistente de versiones utilizando actividades.
- Mantener espacios de trabajo de forma estable y consistente.
- Soportar el cambio concurrente de artefactos y componentes.
- Integrar temprana y frecuentemente.
- Garantizar la reproducción de las construcciones de software.

Para una mejor comprensión de estas prácticas y para resaltar su particular importancia en los ***proyectos de desarrollo distribuido***, se detallarán cada una de ellas:

2.2.2.4.1 Identificar y almacenar los artefactos en repositorios seguros

Para que sean fácil y rápidamente reconocidos y encontrados deben incorporarse a una herramienta adecuada de ***SCM***, ya que en caso de perderse o no identificarse alguna versión de un artefacto, puede fracasar el proyecto porque se obstaculiza la entrega del sistema, o puede disminuir su calidad, porque se pueden ensamblar partes incorrectas. Los artefactos que deben gestionarse por estas herramientas se relacionan con:

- La gestión y el diseño del sistema

- La planificación del proyecto
- Los modelos de diseño
- Archivos fuentes
- Librerías
- Ejecutables
- Los mecanismos utilizados para construirlos

2.2.2.4.2 Controlar y auditar el cambio en los artefactos

Esta práctica guarda relación con la capacidad de registrar información de auditoría, la que debe incluir:

- El control de los permisos para modificar los artefactos
- El registro de la siguiente información cuando se produce un cambio:
  - ¿Qué cambió?
  - ¿Dónde cambió?
  - ¿Cómo se produjo el cambio?
  - ¿Por qué se produjo el cambió?
  - ¿Quién lo realizó?

La ***información de auditoría*** tiene diversos beneficios, uno de ellos es que permite corregir fácilmente los errores cuando se han introducido al sistema.

2.2.2.4.3 Organizar las versiones de los artefactos dentro de las versiones de los componentes

Cuando hay gran cantidad de archivos y directorios, es necesario agruparlos en objetos que representen una granularidad mayor con el propósito de facilitar su gestión y organización. Para esto, se reúnen los archivos y directorios en componentes únicos de ***SCM*** que físicamente implementen un componente lógico del sistema. Una estrategia basada en componentes de ***SCM*** mejora el nivel de comunicación y reduce los errores cuando dos o más proyectos intercambian componentes, ocurre por las siguientes circunstancias:

- Los componentes reducen la complejidad y hacen más manejables los problemas, ya que se asemejan más a la comunicación humana al utilizar un nivel mayor de abstracción.
- Se facilita la identificación de la calidad de un componente mediante sus ***baselines***, en lugar de utilizar una gran cantidad de archivos individuales.

- Al ejemplificar un objeto de componente físico en una herramienta, se ayuda a institucionalizar el intercambio y su reutilización, ya que esta actividad es prácticamente imposible cuando no se pueden determinar las versiones de los archivos que constituyen un componente determinado.

- Al mapear los componentes lógicos de arquitectura a componentes físicos de ***SCM*** (implementables), se gana la habilidad de construir y probar piezas individuales de arquitectura, lo que produce código de mejor calidad e interfaces más limpias.

2.2.2.4.4 Organizar los componentes en subsistemas

La implementación de esta práctica permite que los sistemas complejos sean definidos, controlados y gestionados jerárquicamente. Otra ventaja es que los subsistemas pueden ser reutilizados en futuros proyectos si son controlados, gestionados y publicados independientemente.

2.2.2.4.5 Crear baselines en los hitos del proyecto

Para registrar todas las versiones de los artefactos y componentes que constituyen el sistema o subsistema, en un tiempo determinado del proyecto (por ejemplo en cada hito o entre éstos). Las principales razones para crear ***baselines*** son:

- ***Reproducibilidad***: Capacidad para "retroceder en el tiempo" y reproducir un ***release*** determinado.

- ***Trazabilidad***: Vincula los requisitos, la planificación del proyecto, los casos de prueba y los artefactos de software.

- ***Presentación de informes***: Registra el contenido de los ***baseline*** para poder compararlos, lo que ayuda a remover errores y a la realización de consultas.

Estas tres condiciones son necesarias para resolver problemas de procesos, permiten reparar defectos en los ***releases***, facilitan la aplicación de las ***normas ISO-9000*** y las ***auditorias SEI***, y también ayudan a garantizar que el diseño satisfaga los requisitos, que el código implemente el diseño y que se utilicen los componentes y las versiones correctas para construir los ejecutables.

2.2.2.4.6 Registrar y realizar seguimiento a las solicitudes de cambio

Las solicitudes surgen por diversos motivos, por ejemplo cuando se encuentran defectos (ya sea por el equipo desarrollador o los usuarios), o producto de una nueva idea producida interna o externamente. El registro de las solicitudes ayuda a medir indirectamente el progreso del proyecto y el desempeño de los integrantes. Además, ayuda a la gestión de los proyectos al priorizar y establecer fechas de inclusión de los cambios en algún ***release***.

2.2.2.4.7 Organizar e integrar las versiones utilizando actividades

Se denomina ***actividad*** al vínculo existente entre la versión modificada y el detalle de los motivos para realizar el cambio, representando una ***unidad de trabajo*** en ejecución. Existen diferentes tipos de ***actividades***, por ejemplo: un defecto, solicitud de mejora, requisitos, etc. Las ***actividades*** presentan diferentes granularidades permitiendo que puedan ser parte de una ***actividad mayor***. Una estrategia basada en ***actividades*** combina las disciplinas de ***gestión de configuración, gestión de solicitud de cambios y gestión del proyecto.*** En cambio, las ***unidades de trabajo*** vinculan las ***solicitudes de cambio*** y los ***procesos de implementación***.

El problema ocurre cuando depende del desarrollador mantener un registro de las versiones de los archivos que implementan los cambios lógicos y consistentes; además de él depende garantizar que dichos cambios sean integrados como una unidad. Esto resulta en un proceso tedioso, manual y propenso a errores (especialmente cuando se trabaja en varios cambios a la vez). Los errores pueden ser de construcción (los que producen pérdidas de tiempo) o de Runtime (no pudiendo ser reproducidos en los ambientes de trabajo de los desarrolladores). Las herramientas ***SCM*** deben proveer las capacidades para que los integrantes del equipo registren las ***solicitudes de cambio*** y los

defectos en los que trabajan. Además esta información se debe utilizar para registrar los archivos o directorios que han cambiado, con lo cual se agiliza el proceso de integración y garantiza la configuración de la construcción.

Las herramientas de ***SCM*** que permiten la gestión a través de ***actividades*** aumentan el nivel de abstracción desde trabajar con archivos y versiones a trabajar con ***actividades***. Para que haya trabajo eficiente, las ***actividades*** se deben presentar en las interfaces de los usuarios de forma permanente y actualizada.

Los beneficios de una herramienta ***SCM*** basada en ***actividades,*** son:

- ***Cambios consistentes***: Eliminando diversos problemas de integración y construcción, al integrar un cambio lógico único, reduciendo los errores causados por el olvido de los desarrolladores. También, garantizan que las pruebas se ejecuten en las versiones que serán integradas.

- ***Siguen una lógica semejante a la humana***: Las personas generalmente piensan en características, solicitudes de mejoras o defectos, en vez de enfocarse en los detalles como los archivos. Además, al disminuir el nivel de especificación se reduce la información que deben procesar los integrantes del equipo.

- ***Se vinculan naturalmente con la gestión de solicitud de cambios***: Permitiendo informes precisos sobre los defectos que fueron arreglados y los cambios producidos entre los ***baselines***.

- ***Se vinculan naturalmente con la gestión de proyectos***: Debido a que los gestores no sólo se interesan en lo que ha cambiando, sino también en los ***estados de los cambios***, quién fue asignado para su realización y cuánto esfuerzo se estima para

implementarlo. Apoyando la ***automatización del control*** y los ***mecanismos de coordinación y comunicación***.

- ***Facilitan la comunicación***: Al permitir la rápida confección de informes y al promover indicadores de los cambios realizados. Facilitando una comunicación más natural entre los involucrados en el proyecto.

- ***Agilizan la revisión de código***: Los miembros del equipo puedan comparar automáticamente la versión predecesora con la última, facilitando y haciendo menos propensa a errores las revisiones.

- ***Agilizan las pruebas***: Cuando se utilizan los informes de pruebas anteriores de ***baselines*** para revisar sólo los nuevos cambios.

2.2.2.4.8 Mantener espacios de trabajo estables y consistentes

El desarrollador requiere herramientas y automatización para formar y mantener un ambiente de trabajo que lo aísle de los cambios disruptivos que se produzcan en otros lugares del proyecto, a su vez, el proyecto debe protegerse de los cambios perjudiciales que se produzcan en los ***espacios de trabajo*** de los desarrolladores. Cobrando particular importancia cuando se produce una sincronización periódica de los cambios producidos por los otros miembros del equipo. Permite maximizar la productibilidad de los desarrolladores, porque sin un lugar de trabajo estable y consistente, éstos pierden tiempo en problemas que han sido introducidos por otros integrantes. A su vez, un modelo consistente de ***espacios de trabajo*** proporciona al actualizarse, un conjunto de versiones conocidas que han sido construidas y probadas.

2.2.2.4.9 Apoyar los cambios concurrentes tanto de artefactos como de componentes

Idealmente, sólo una persona debería cambiar los archivos, o sólo un equipo debería trabajar en un componente. Sin embargo, esto no siempre es eficiente o práctico, por lo tanto las herramientas ***SCM*** deben apoyar las modificaciones concurrentes de un mismo archivo y deben integrar en un tiempo apropiado los cambios que se hicieron en paralelo.

2.2.2.4.10 Integrar temprana y frecuentemente

Durante la integración se descubren problemas de interfaces y malos entendidos del diseño. Estos problemas generalmente tienen un impacto mayor en los plazos siendo necesario descubrirlos tempranamente. El ***plan de integración*** debe realizarse en forma prematura en el ***ciclo de desarrollo***. Además, se tiene que integrar continuamente a medida que avance el proyecto. Los gestores y los integradores del proyecto deben tener una forma válida y automática de asegurar que los desarrolladores se mantengan al tanto de los cambios en ejecución. Se debe establecer un balance entre integración prematura y su frecuencia, para no afectar la productividad.

2.2.2.4.11 Garantizar la reproducción de las construcciones del software

Generalmente se necesita establecer cómo se construyó y qué hay dentro del software. Lo que sirve para depurar los problemas y reproducir la misma construcción. Esta práctica señala que se deben establecer procedimientos manuales y automáticos para registrar información como la siguiente:

- Quién hizo la construcción
- Qué hay dentro de cada ejecutable
- En cuál máquina se construyó
- Qué versión y sistema operativo se utilizó para realizar la construcción
- Cuáles líneas de comando se usaron en la construcción

Esta información es útil porque en algunas ocasiones se pueden introducir ***bugs*** tan sólo cambiando el *switch optimizador del compilador*. Esta práctica también ayuda realizar futuras mantenciones al sistema.

2.2.2.5 Prácticas sugeridas por ThoughtWorks

Las prácticas propuestas por esta empresa son el resultado de cuatro años de experiencia (desde comienzos de 2002 hasta comienzos de 2006) en proyectos desarrollados en la India, América del Norte y Europa (Fowler 2006). Enfocándose principalmente en utilizar prácticas ágiles en los proyectos distribuidos. Las prácticas propuestas son:

- Usar integración continua para evitar los problemas
- Intercambio de embajadores entre los sitios remotos
- Utilizar visitas presenciales para construir confianza

- No subestimar los cambios culturales
- Utilizar wikis para almacenar información común
- Utilizar Teste Scripts para ayudar a entender los requisitos
- Utilizar frecuentemente construcciones para obtener feedback de las funcionalidades
- Realizar reuniones cortas y frecuentas para conocer el estado de las tareas y actividades
- Realizar iteraciones cortas (de una o dos semanas), que vayan entregando funcionalidades incrementalmente
- Utilizar una planificación de iteración adaptada a los sitios remotos, según la restricciones propias de cada sitio (por ejemplo a las diferencias horarias y culturales)
- Realizar reparación de defectos para dar a conocer el código
- Separar los equipos por funcionalidades y no por actividades
- Utilizar más documentación, que en los proyectos ágiles co-localizados, para mejorar la comunicación, coordinación y control
- Implementar múltiples formas de comunicación desde el inicio del proyecto

2.2.3 Prácticas seleccionadas

1 Minimizar el grado de dispersión geográfica

2 Establecer un proceso de ingeniería de requisitos bien definido y comprendido

3 Realizar reuniones colaborativas frecuentemente, fijando reglas de estilo y frecuencia:

- Reuniones Scrum diarias entre el jefe de proyecto y los miembros remotos, utilizando protocolos de comunicación, por ejemplo enviando las respuestas por mail antes de la reunión
- Reuniones diarias en cada sub-equipo
- Reuniones Scrum semanalmente entre los arquitectos del proyecto
- Reuniones diarias (o lo más frecuentemente posible) que involucren al jefe de proyecto, los analistas y al cliente
- Cada dos semanas realizar reuniones de planificación donde se deben solicitar los requisitos

4 Mejorar la coordinación y comunicación entre los sitios

- Establecer responsabilidades comunicacionales en cada sitio
- Incrementar la percepción sobre lo que ocurre en el proyecto, la compañía y los sitios remotos

5 Utilizar herramientas y tecnologías que apoyen la accesibilidad, colaboración y ejecución de los proyectos

- Utilizar una herramienta para realizar construcciones automáticas varias veces al día
- Utilizar una herramienta especial para sustentar una efectiva ingeniería concurrente que mida el progreso del proyecto mediante el seguimiento de funcionalidades, tareas y actividades
- Estandarización de herramientas y procesos para los diferentes sitios
- Centralizar el acceso a las herramientas mediante la web, bases de datos replicadas, ambientes de desarrollo individuales y repositorios centrales
- Rápido acceso a la red de trabajo
- Compartir recursos como bases de datos, servidores y repositorios del proyecto
- Conectividad fácil y rápida entre los sitios remotos
- Tecnologías colaborativas que permitan incrementar la efectividad y satisfacción personal

6 Promover vínculos sociales entre los integrantes

- Construir relaciones sociales para facilitar la buena comprensión entre los interlocutores y establecer confianza

- Incrementar la accesibilidad entre los miembros de los sub-equipos, para conocer la disponibilidad de los otros integrantes (día y hora)

- Crear y mantener una atmosfera de equipo, evitando el aislamiento de los sitios remotos

- Facilitar y permitir la integración entre los miembros

- Facilitar el intercambio de personal entre los sitios para reducir las brechas culturales

7 Promover el intercambio de conocimiento

- Originar, documentar y divulgar conocimiento propio de cada equipo y promover las capacidades de cada miembro

- Ampliar el conocimiento que se posee de los sub-equipos

- Capacitar a los miembros del equipo y constantemente ir incorporando nuevas tecnologías

8 Promover y facilitar la gestión de componentes

- Diseñar los componentes para que sean reutilizados

- Invertir en proyectos de investigación y desarrollo

- Facilitar la reutilización, identificando las oportunidades que permitan emplear nuevamente los componentes y el conocimiento elaborado

9 Prácticas básicas de gestión de la configuración

- Identificar y almacenar los artefactos en repositorios seguros
- Controlar y auditar los cambios de los artefactos
- Mantener ambientes de trabajo de forma estable y consistentemente
- Soportar el cambio concurrente de artefactos y componentes
- Garantizar la reproducción de las construcciones de software

10 Integrar temprana y frecuentemente (práctica avanzada de gestión de la configuración)

11 Registrar y rastrear las solicitudes de cambio (práctica avanzada de gestión de la configuración)

12 Organizar e integrar un conjunto consistente de versiones utilizando actividades (práctica avanzada de gestión de la configuración)

13 Realizar iteraciones cortas (de una o dos semanas), que vayan entregando funcionalidades incrementalmente

3. PRIORIZACIÓN DE PRÁCTICAS

3.1 Criterio de priorización para la implementación de las prácticas

El propósito de establecer criterios de priorización para implementar gradualmente las prácticas y requisitos tecnológicos, consiste en minimizar los riesgos de las empresas elaboradoras de software que deseen pasar desde un estado básico a uno optimizado. Sin embargo, el primer desafío consiste en definir los que denominados estado básico y optimizado. Para esto, se escogió como criterio fundamental que las empresas desarrollan software por un afán de lucro. Por ende, la finalidad de pasar desde un estado básico a uno optimizado es maximizar la función de utilidad de la empresa.

$$Utilidades = Ingresos - Costos$$

Figura 3- 1: Relación simplificada para determinar las utilidades de una empresa

En consecuencia, el motivo de la propuesta consiste en proveer una guía, para que las organizaciones elaboradoras de software, puedan implementar de forma paulatina las prácticas y requisitos tecnológicos, que les permitan desarrollar controladamente la madurez de la empresa, permitiéndoles fijar así, metas para transitar desde un estado básico de ejecución a uno más alineado con los objetivos del negocio.

Para establecer los criterios de priorización que permitan la categorización de las empresas elaboradoras de software, primero se determinarán:

- las dimensiones, que definen los estados posibles
- los estados, para categorizar a las empresas
- las transiciones, que delimitan las rutas posibles entre los estados
- las causas, que precisan los motivos para transitar desde un estado a otro

La unidad de negocio básica para una empresa que desarrolla software a medida, consiste en la ejecución exitosa de los proyectos de software, sin embargo, diversas variables se deben atender para que pueda ocurrir esto.

En función de lo anterior, se seleccionó como primera dimensión los *costos esperados*, porque una forma de maximizar la función de utilidad de una organización, consiste en minimizar sus costos (suponiendo que los precios de venta y los productos comercializados están acotados).

La segunda dimensión que se seleccionó tiene relación directa con la satisfacción del cliente, pudiéndose así, maximizar el complemento de la función de utilidad de una empresa, es decir, maximizar los ingresos (precio del producto y volumen de producción). Suponiendo que se satisfacen las necesidades de los usuarios, mediante los requisitos del software, quedan dos variables que inciden en el éxito de estos proyectos, estas son costo y plazo para elaborar el software. Por su parte, estas dos variables se relacionan directamente con uno de los principales problemas de los proyectos de software, el cual es realizar estimaciones correctas, dado que es dificultoso determinar efectivamente el costo y plazo para desarrollar el producto. Por lo tanto, se considerará como segunda dimensión de categorización el *grado de certeza para establecer las estimaciones de costo y tiempo para ejecutar el proyecto.* Estas dos variables están estrechamente relacionadas, porque el costo del producto será directamente proporcional al tiempo en que se ejecute el proyecto. Por ende, la meta de esta dimensión consistirá en poder maximizar el grado de certeza para determinar las estimaciones de costo y tiempo, es decir, minimizar la variabilidad y maximizar la exactitud. La dimensión antes declarada repercute directamente en la confianza y satisfacción de los clientes, tanto para nuevos como antiguos; lo que a su vez se puede manifestar en un mayor precio y/o volumen de venta.

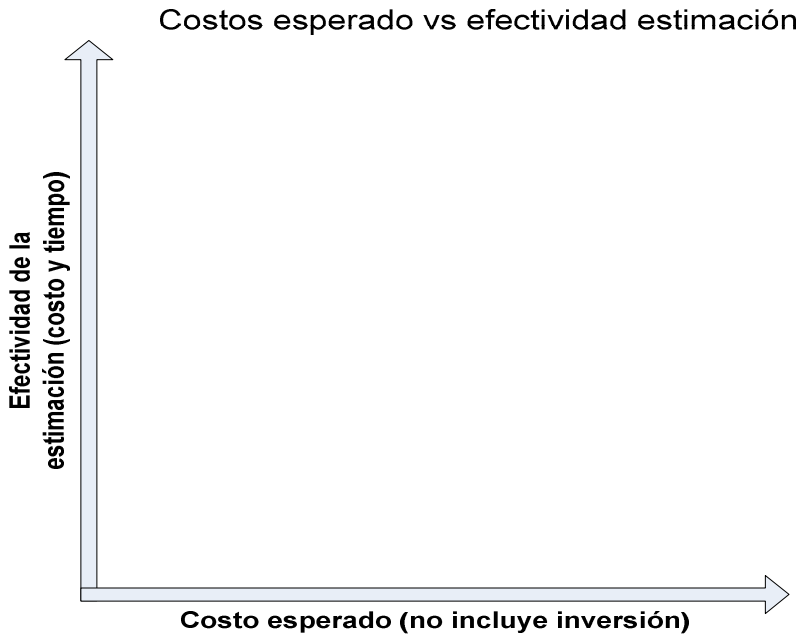

Figura 3- 2: Dimensiones que se utilizarán para definir los estados posibles que categorizarán a las empresas

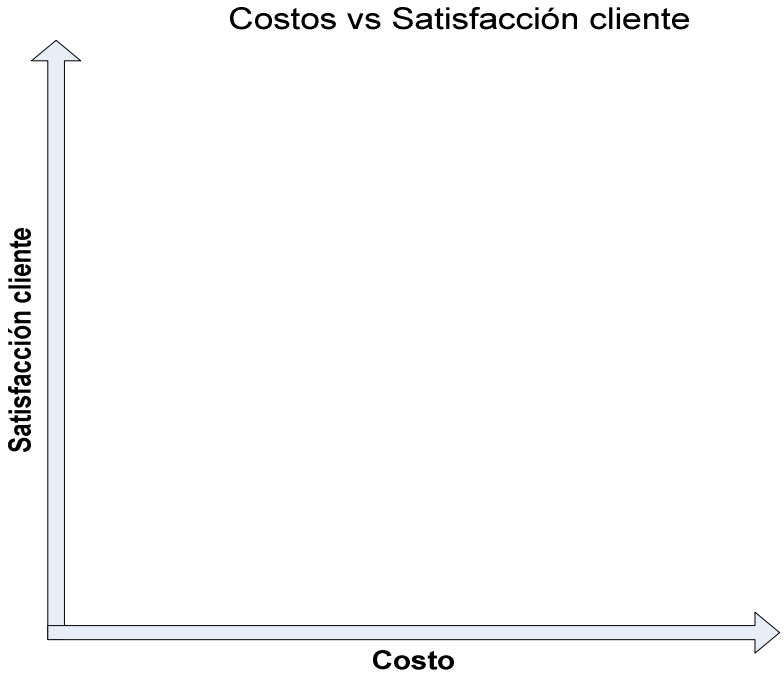

Figura 3- 3: Dimensiones genéricas que pueden ser utilizadas para categorizar a las empresas, independiente de su industria

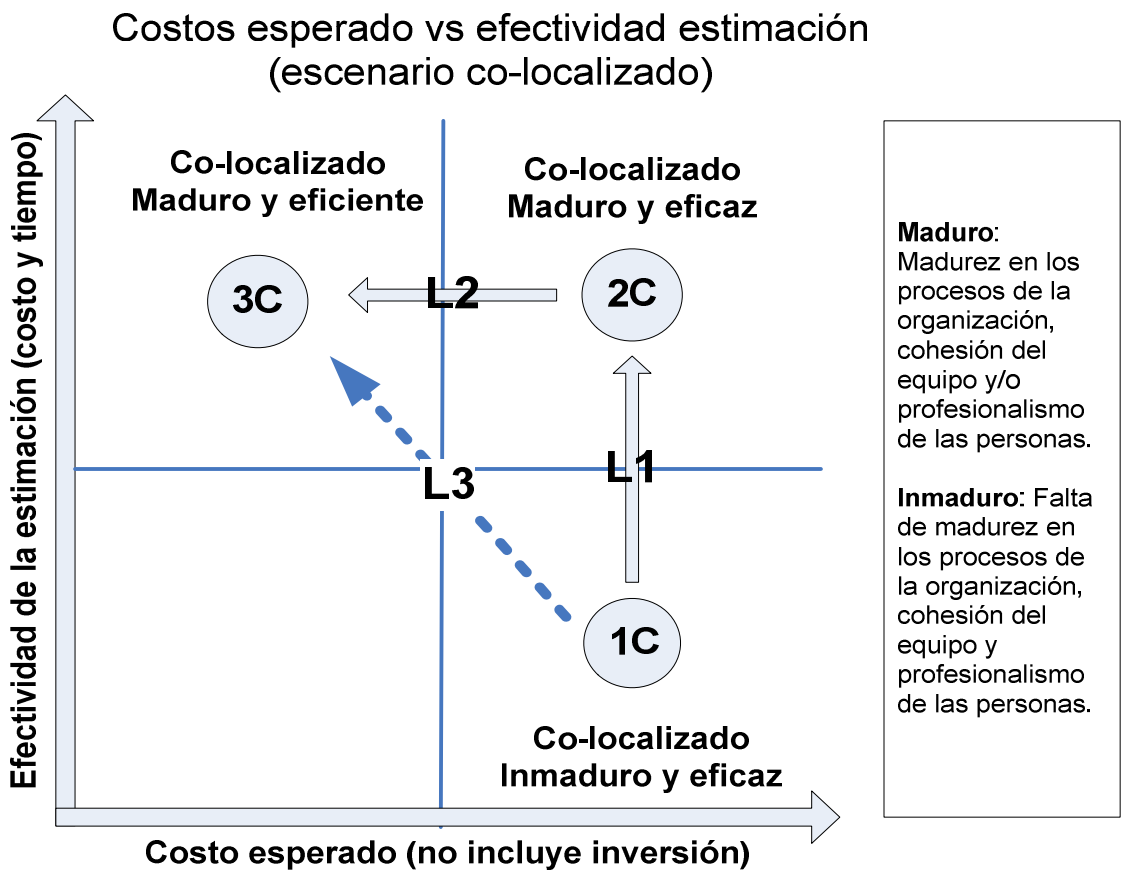

Figura 3- 4: Representación de los estados para categorizar a las empresas desarrolladoras co-localizadas y transiciones posibles para pasar desde un estado inferior a uno superior

Como se aprecia en el gráfico, el estado correspondiente al *nivel 1 Co-localizado (**1C**)*, es el más básico, definiéndose como el estado en el cual las organizaciones consiguen producir el producto, pero no satisfacen todas las necesidades del cliente, ya que no logran estimaciones efectivas para determinar el costo y tiempo en el que se efectuará el proyecto (este estado se asemeja, en ciertos aspectos, al nivel de madurez 1 de CMMI). Esta situación se manifiesta cuando los proyectos sobrepasan el presupuesto y los plazos determinados; o cuando concluyen antes de lo previsto. Las dos circunstancias son perjudiciales para las organizaciones desarrolladoras, ya que podrían repercutir en sus ingresos futuros, dado que en ambos casos los clientes pueden perder la confianza en su proveedor.

El estado correspondiente al *nivel 2 Co-localizado* (***2C***), ha sido pensado para categorizar a las organizaciones que satisfacen todas las necesidades del cliente y los usuarios (este estado se asemeja, en ciertos aspectos, a una madurez CMMI de nivel mayor o igual a dos), ya que produce un producto que atiende los requisitos del negocio del cliente, y lo realiza dentro de los plazos y costos definidos. No sólo es posible lograr este estado mediante la madurez de los procesos de la organización, sino también, a través de equipos de desarrollo bien cohesionados (por ejemplo al implementar ciertas prácticas proporcionadas por TPS o Scrum) y/o gracias a la madurez profesional alcanzada por los integrantes del equipo (por ejemplo al implementar ciertas prácticas proporcionadas por PSP o XP). Las condiciones para transitar desde un estado ***1C*** a uno ***2C*** (representado por ***L1***) dependerán de ciertos factores que influyan en el proyecto, por ejemplo: tamaño (horas-hombre), madurez de la tecnologías que se utilizarán, estabilidad de los requisitos, gravedad de los requisitos, entre otros. Es así como, en la medida que se incremente el nivel de complejidad, se requerirán mayores compromisos en los procesos organizacionales, cohesión del equipo y madurez de los profesionales.

El tercer estado corresponde al *nivel 3 Co-localizado* (***3C***), ha sido pensado en las organizaciones qué, además de satisfacer todas las necesidades del cliente y los usuarios, también logran alinearse con las necesidades de su propio negocio, es decir, consiguen disminuir su costos de elaboración. Este nivel se logra gracias a diferentes estrategias que agilizan el proceso de desarrollo (disminuyendo el ciclo de vida de producción), las que pueden ser implementadas conjunta o aisladamente, las estrategias identificadas son: incorporar nuevos principios, prácticas y técnicas que agilizan la elaboración; utilizar tecnología apropiadas que apoyen los procesos de desarrollo; aumentar el grado de cohesión de los integrantes; y/o integrar a personas que posean mayor grado de experiencia o habilidades para desarrollar.

Se definieron las transiciones ***L2*** y ***L3*** para pasar desde un estado inferior (***1C*** o ***2C***) a otro superior (***3C***). La ruta lógica para lograr el estado ***3C*** corresponde a transitar por ***L1*** y posteriormente por ***L2***. Sin embargo, se ha definido la ruta ***L3*** como un atajo factible, siendo posible gracias a la incorporación de tecnologías adecuadas y producto de un importante factor humano (compromiso, entusiasmo y entrenamiento), ejemplo de esto es el caso de la empresa Link (Villalón 2007).

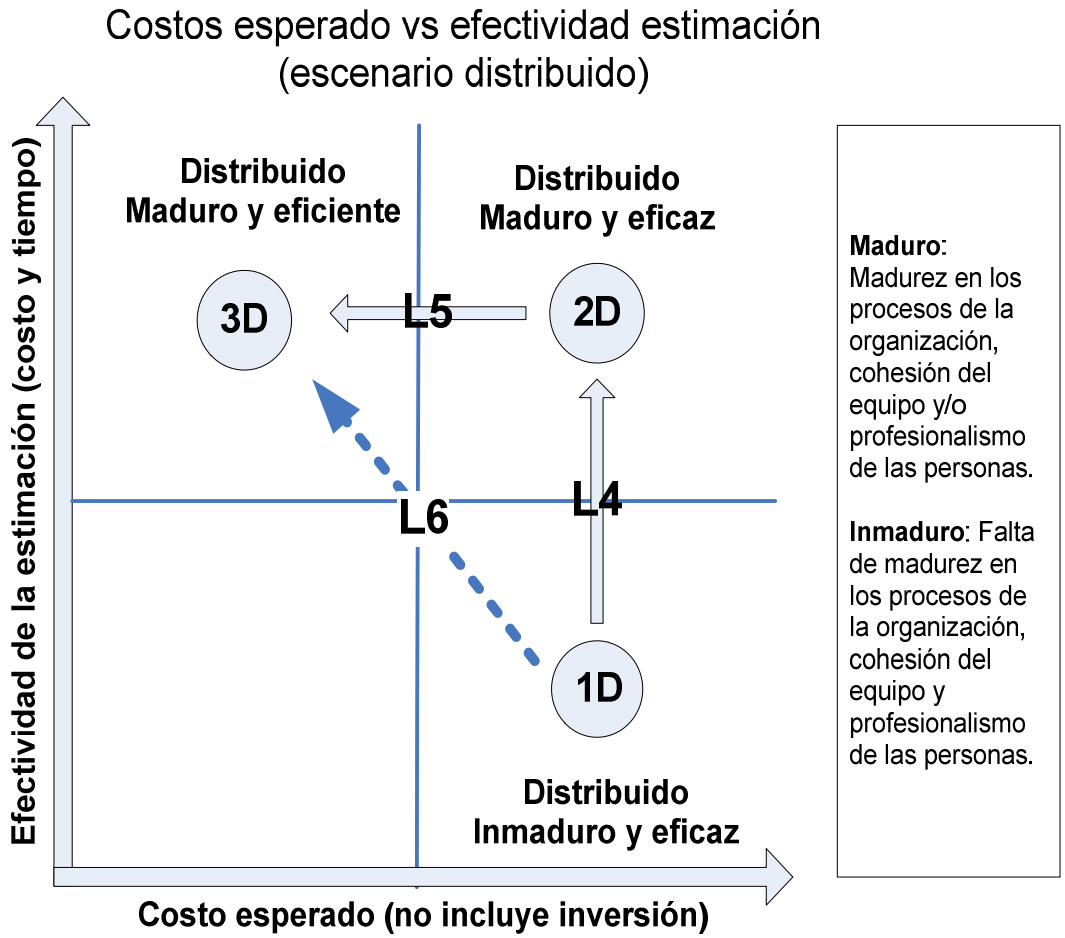

Figura 3- 5: Representación de los estados para catalogar a las empresas que desarrollan distribuídamente, y las transiciones posibles para pasar desde un estado inferior a uno superior

Para categorizar a las empresas distribuidas se empleó la misma lógica aplicada en el escenario co-localizado, dado el supuesto que los estados y transiciones posibles son condiciones propias de la industria del software, más que una condición propia del escenario. Sin embargo, la diferencia radica en las prácticas que se deben aplicar para poder pasar desde un estado a otro, ya que en este escenario existen barreras adicionales que dificultan el normal desarrollo de los proyectos distribuidos, por ende, es altamente probable que los cambios de estado sean más complejos, es decir, tomen más esfuerzo y tiempo.

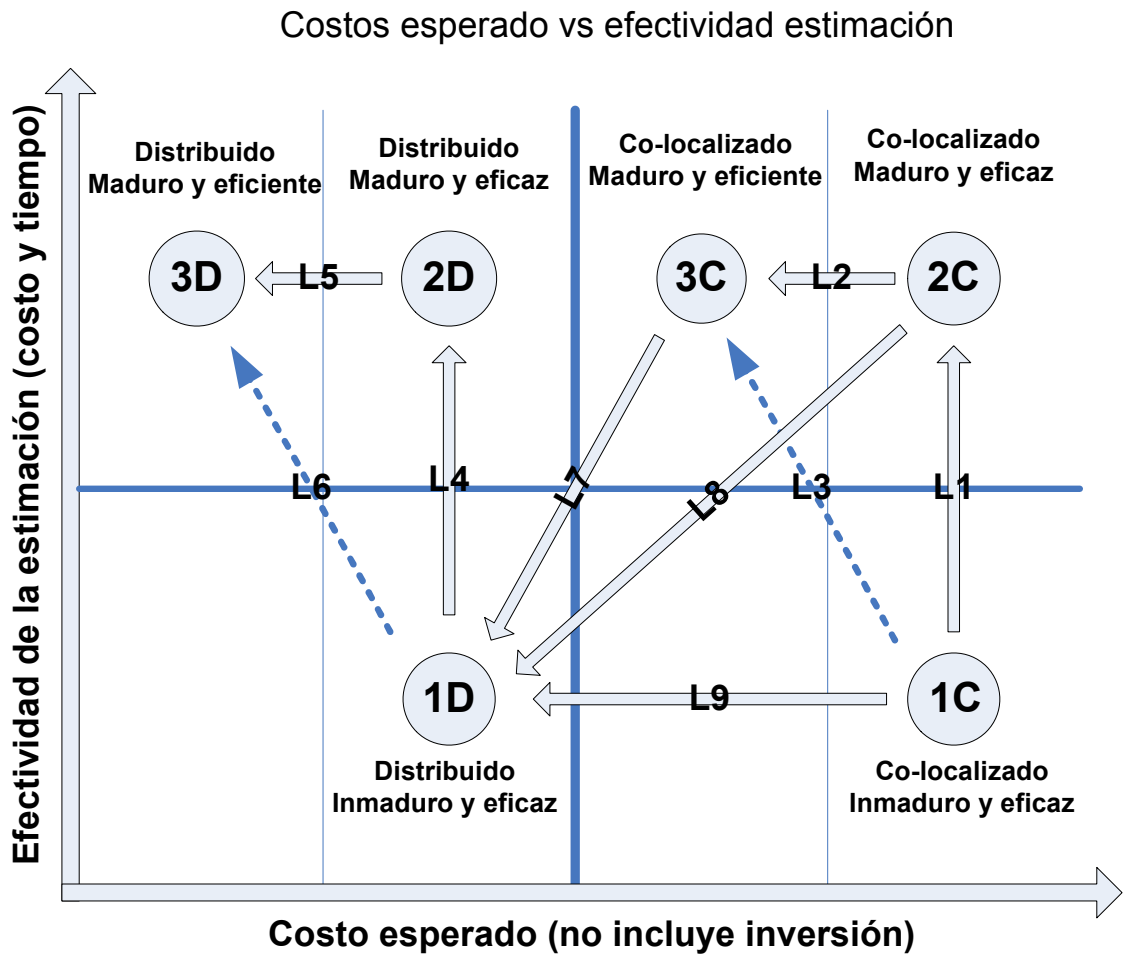

Figura 3- 6: Rutas posibles para pasar desde un escenario co-localizado a uno distribuido

Para pasar desde un escenario co-localizado a uno distribuido, las organizaciones deben atender nuevas problemáticas concernientes al *outsourcing* de procesos. Como referencia se ha utilizado la propuesta hecha por el Centro de Calificación de los Servicios de Tecnología de la Información (ITSqc) de la universidad de Carnegie Mellon. Esta institución ha identificado 23 problemáticas que deben ser atendidas para establecer una relación adecuada entre la organización cliente y proveedora, incluso ha creado un marco de trabajo denominado eSourcing Capability Model (eSCM), el cual consta de 84 prácticas agrupadas en 5 niveles de madurez. Por tal motivo, se puede inferir que es baja la probabilidad de éxito cuando se desea pasar desde un estado Co-localizados a un estado ***2D*** o ***3D***. Por tal motivo, se plantea que para transitar desde cualquier estado co-localizado a uno distribuido, primero se debe pasar por ***1D.*** Es así como, se plantean las rutas ***L7***, ***L8*** y ***L9***; las cuales, para transitarlas, se deben adaptar los procesos de la organización, entrenar al personal e incorporar tecnología que pueda facilitar la comunicación, coordinación y control entre los equipos remotos. La diferencia entre las rutas radica, principalmente, en el grado de riesgo asociado; por ejemplo, la ruta ***L7*** minimiza la probabilidad de fracaso del proyecto, ya que la organización presenta un alto grado de madurez y optimización (en procesos apoyados por tecnologías, cohesión del equipo y profesionalismo de sus integrantes), mientras que la ruta ***L9*** tiene asociada la mayor probabilidad de fracaso, porque el equipo desarrollador sólo posee un respaldo mínimo para lograr el éxito del proyecto.

Se identificó otra ventaja asociada al tránsito por ***L7***, consiste en lograr más rápidamente el estado óptimo (***3D***). Dicha afirmación se basa en que para transitar desde un estado ***2D*** a uno ***3D***, también se deben implementar las prácticas utilizadas en la transición ***L2*** (desde ***2C*** a ***3C***), esto se debe a que las prácticas propuestas son acumulativas. Por ende, cuando se pasa desde ***3C*** a ***1D***, es más probable que el salto a los siguientes estados superiores sea más veloz porque los miembros de la organización ya tienen experiencia estimando efectivamente, como también ya han optimizado parcialmente el proceso de desarrollo.

3.2 Tipo de proyectos más competitivos para el escenario distribuido

Este capítulo tiene la finalidad de realizar un análisis para seleccionar las características que agregan mayor valor a los proyectos distribuidos de software que se realicen en países emergentes como Chile.

- El desarrollo distribuido de software tiene la ventaja potencial de poder captar demanda geográficamente distante. Por tal motivo, las empresas de software chilenas que puedan aplicar un exitoso desarrollo distribuido, pueden satisfacer las necesidades de clientes que posean alto poder adquisitivo. Un ejemplo es poder satisfacer a empresas de países desarrollados.

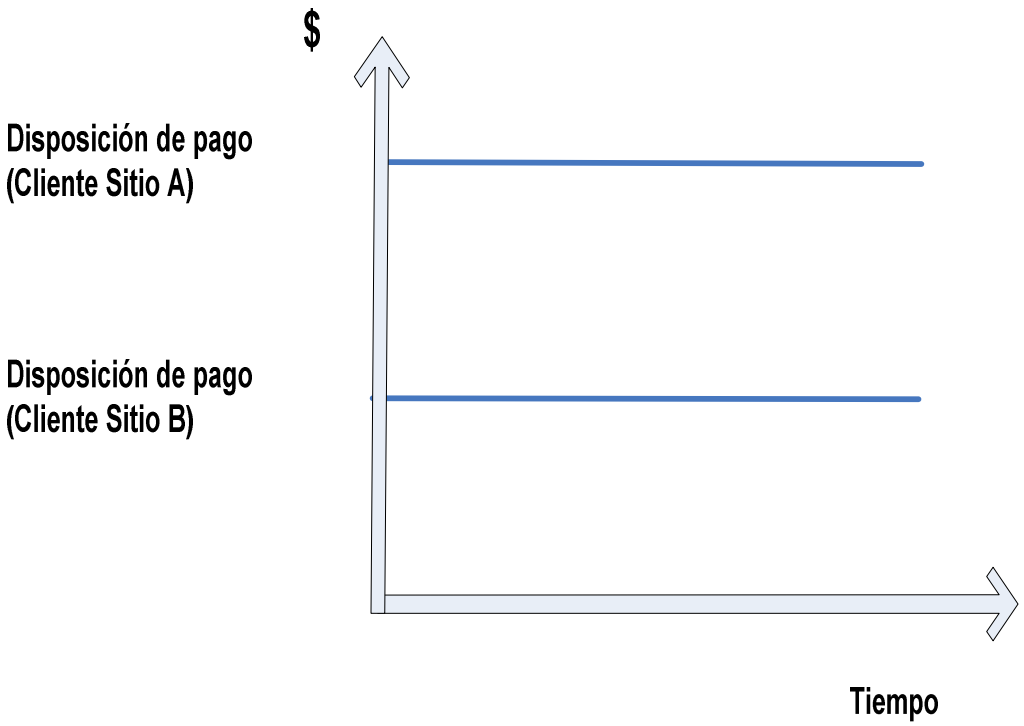

Figura 3- 7: Representación de las diferencias en la disposición a pagar de dos clientes que desean adquirir un mismo producto, pero que pertenecen a distintas economías (geográficamente remotas)

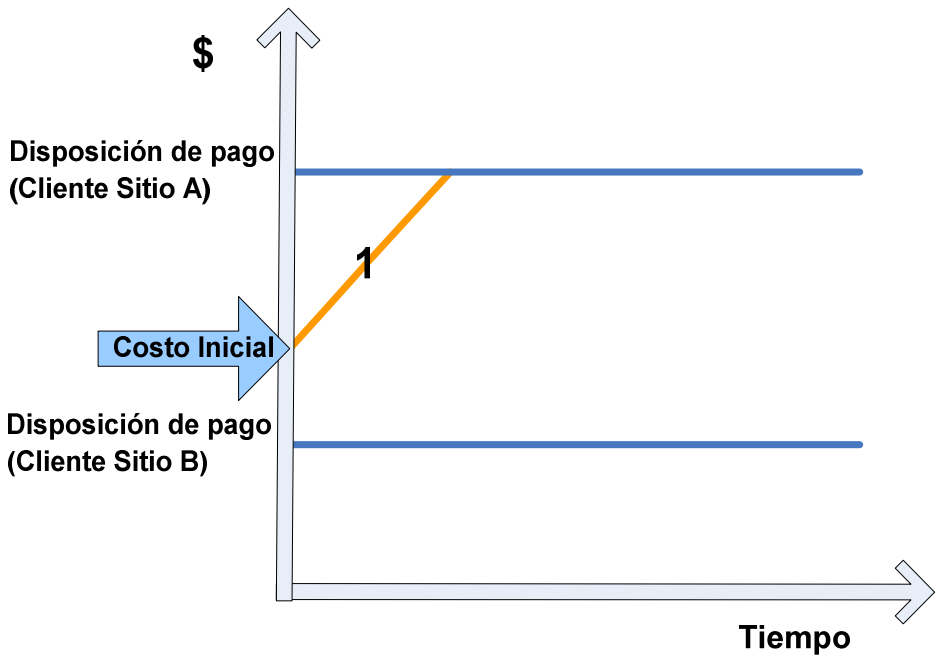

Figura 3- 8: Representación de la función de costo en el tiempo, de un proyecto elaborado en proximidad geográfica con el cliente (co-localizados en el sitio A)

En la figura se puede apreciar que los costos del proyecto se incrementan proporcionalmente a medida que transcurre el tiempo para concretarlo. Dicha curva de costo está compuesta por el costo inicial (dependerá de los cotos concernientes a la puesta en marcha del proyecto, relacionados directamente con la infraestructura, costos administrativos, entre otros), y un factor variable (que dependerá fundamentalmente de la cantidad de personal requerido, su sueldo mensual y el tiempo necesario para efectuar el proyecto).

$$Costos_{proyecto} = Costo_{inicial} + p * q * t$$

Figura 3- 9: Ecuación lineal que muestra el comportamiento de la curva de costo.

Donde:

q = Es la cantidad de personal requerido
p= Es un promedio de los sueldos
t = Es el tiempo necesario para elaborar el proyecto

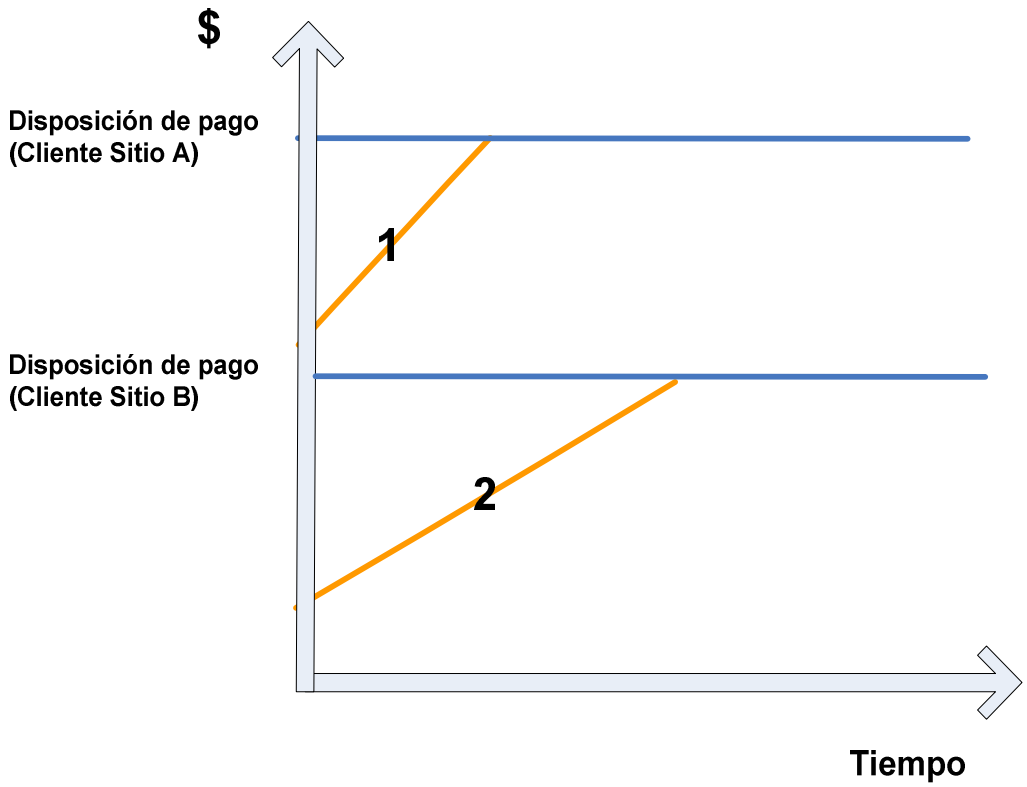

Figura 3- 10: Representación de dos clientes ubicados en localidades distintas (con sus respectivas disposiciones de pago), y a dos empresas también ubicadas en localidades distintas

Curva 1: representa la función de costos del proyecto co-localizado en el sitio A

Curva 2: representa la función de costos del proyecto co-localizado en el sitio B

En la figura se puede distinguir que las dos empresas presentan curvas de costos distintas, producto de la brecha en los costos iniciales y por la diferencia en los sueldos que se pagan en cada localidad. También se puede apreciar que la pendiente de la curva de costos, en la empresa ubicada en el sitio B es menos pronunciada porque el costo marginal de poseer un empleado adicional es menor en B que en A. Como comparación Azher (2007) señala que el salario anual de un programador en los EEUU es de US$ 55.000, mientras que en India es de US$ 15.000.

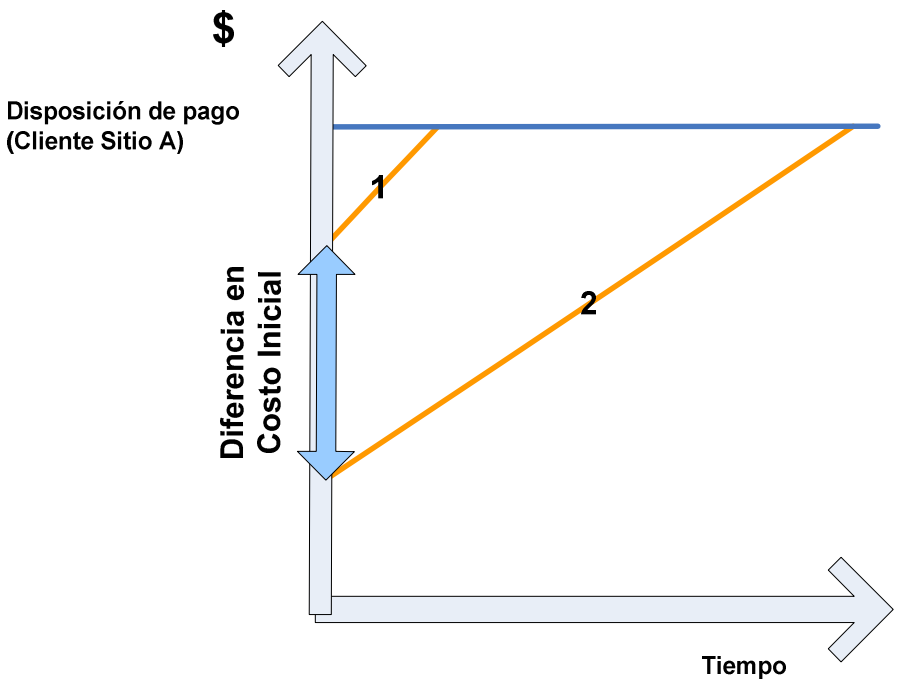

Figura 3- 11: Representa a las dos empresas, que están ubicadas en distintos países, compitiendo por un mismo cliente

La figura anterior muestra a dos proyectos (ubicados en A y B), que elaboran un mismo software, pero que poseen curvas de costo distintas. El proyecto elaborado en el sitio B, corresponde a un desarrollo y venta remota (100% ubicado en el sitio B), ejemplo de este modelo son las empresas que venden software por internet (como ciertos antivirus). Este modelo de negocio no es usualmente utilizado, principalmente, porque conlleva problemas de piratería y producto de la creciente competencia del software que posee licencia de libre utilización (por ejemplo *GNU General Public License*). Pese a las problemáticas señaladas, el modelo de negocio de desarrollo y venta remota, nuevamente ha empezado a utilizarse gracias al surgimiento de dos estrategias, las que son: Software como un Servicio (*SaaS*) y Arquitectura Orientada a Servicios (SOA). Lo anterior producto a que estas estrategias permiten las ventas remotamente, sin la necesidad de intercambiar software. Porque se cobra por el servicio brindado, evitándose así, los problemas de piratería.

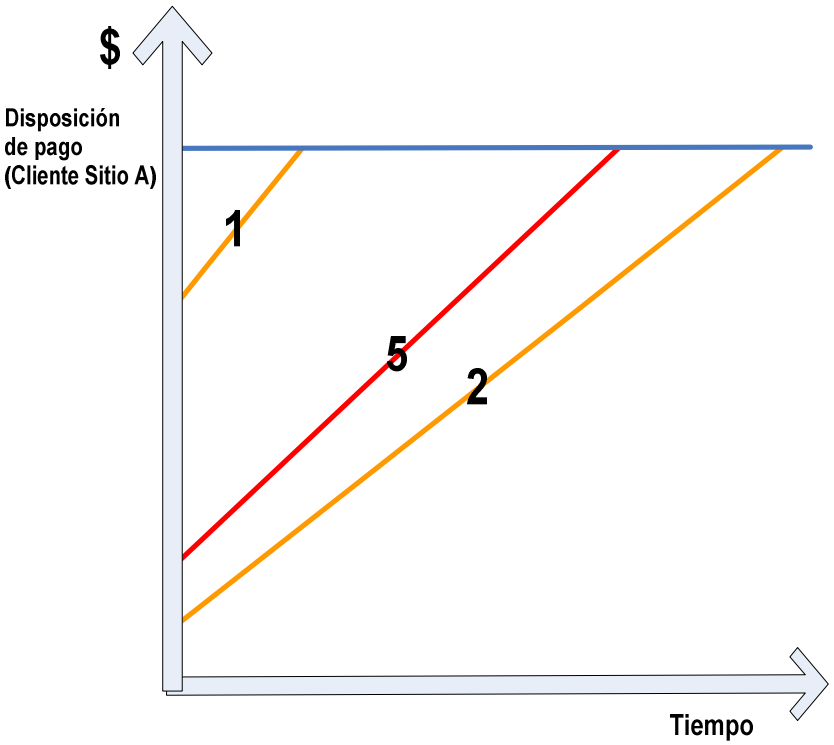

Figura 3- 12: Curva de costos (línea 5) para un proyecto distribuido entre los dos países

La línea 5 de la figura anterior, muestra la curva de costo de un proyecto distribuido que considera la variable salario como el principal factor.

$$Costo_{proyecto\ distribuido\ (A\ y\ B)} = \propto * Costo_{sitio\ A} + (1-\propto) * Costo_{sitio\ B}$$

$$Donde\ 0 \leq \propto \leq 1\ dependiendo\ del\ \%\ del\ personal\ en\ cada\ sitio$$

Figura 3- 13: ponderación del desarrollo co-localizado en el sitio A y del desarrollo co-localizado en el sitio B

Se puede apreciar que la pendiente resultante y los costos iniciales son una ponderación del desarrollo co-localizado en el sitio A y del desarrollo co-localizado en el sitio B.

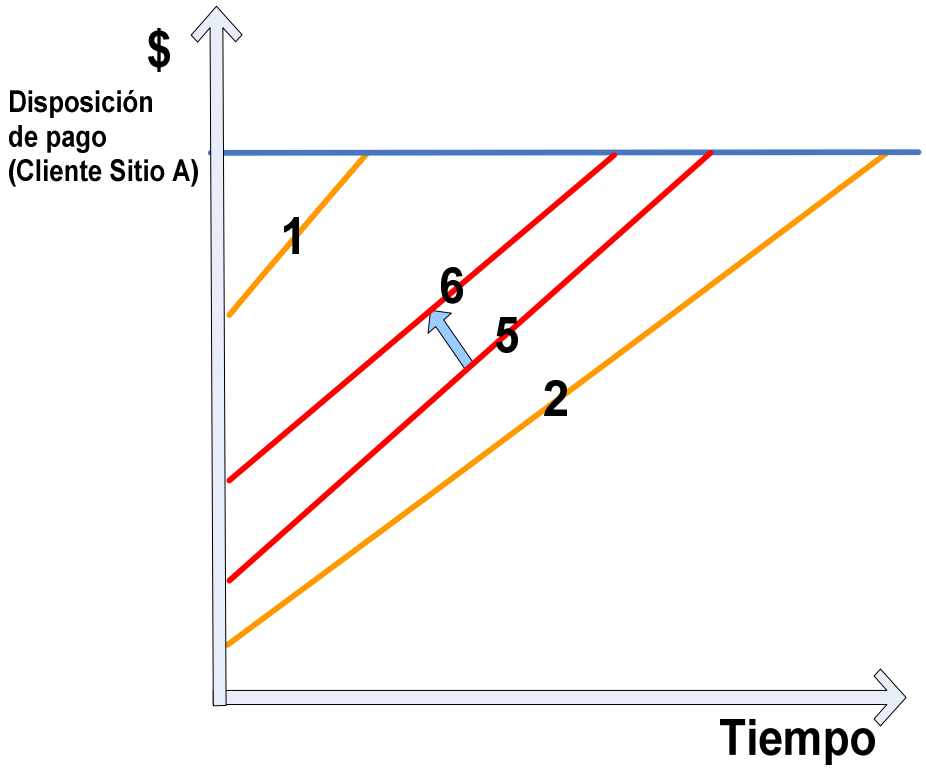

Figura 3- 14: Alteración de la curva de costos (para un proyecto distribuido entre A y B), al considerar los costos adicionales que deben encarar los proyectos dispersos geográficamente, para que puedan ser exitosos

El grado de dispersión geográfica (cantidad de sub-equipos remotos), es un factor que incrementa los costos de producción. Esto es lo planteado por Kroll y Maclsaac (2006) al señalar que aumenta la complejidad de un proyecto cuando se incrementan los posibles canales de comunicación, ya que para lograr una mínima coordinación y control (que permita lograr el éxito del proyecto), es necesario que haya una comunicación fluida entre los sub-equipos remotos.

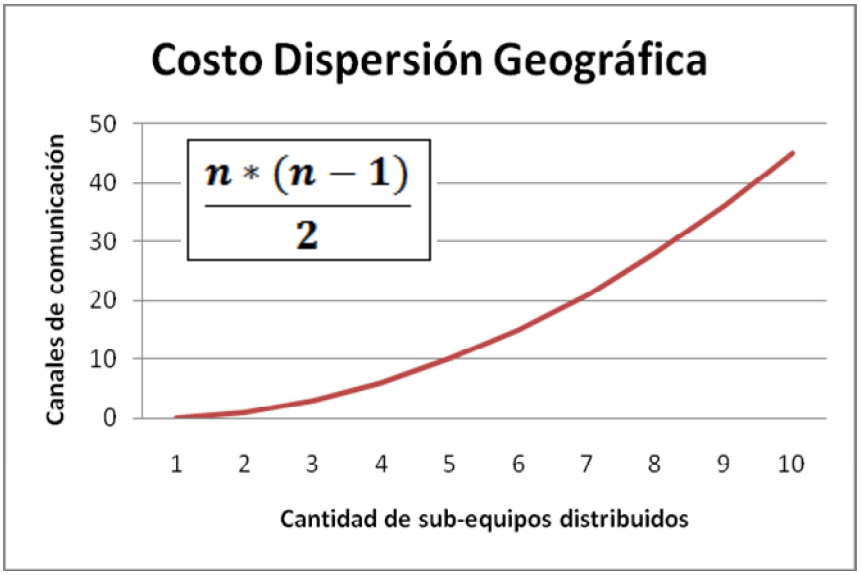

Figura 3- 15: Representación del incremento en costos del proyecto, mediante los canales de comunicación requeridos, para una determinada cantidad de sub-equipos

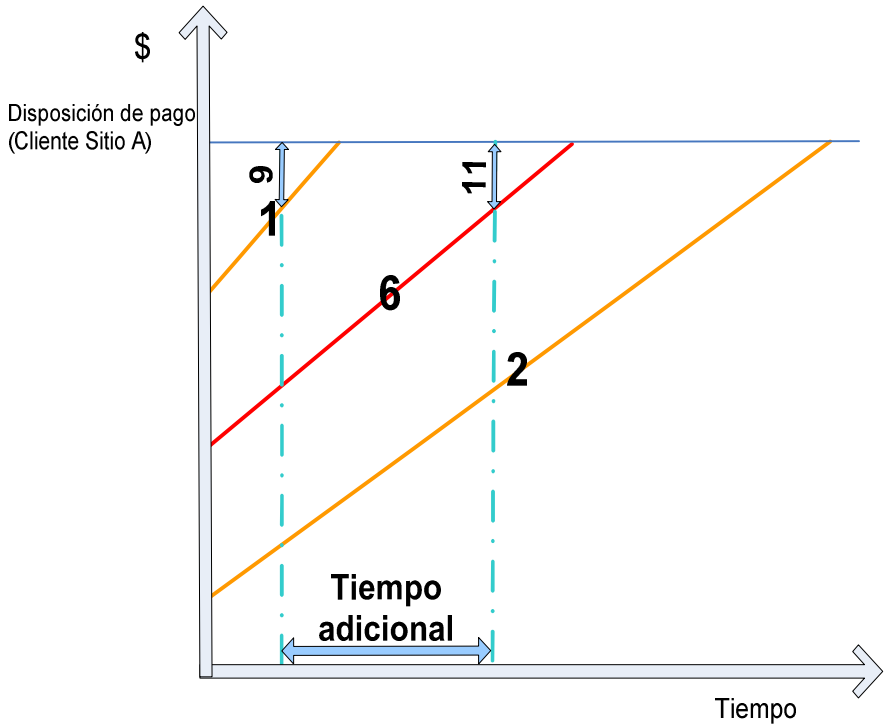

Figura 3- 16: Tiempo adicional que poseen los proyecto distribuidos (para producir una aplicación que proporcione igual rentabilidad), versus la limitación en tiempo y costos que poseen los desarrollos co-localizados en el mismo sitio que el cliente

El tiempo adicional es una ventaja competitiva que poseen los proyectos más económicos, porque ciertos proyectos requieren mayores plazos para poder satisfacer las necesidades de los clientes y usuarios. Ejemplo de esto son los proyectos con clientes inmaduros.

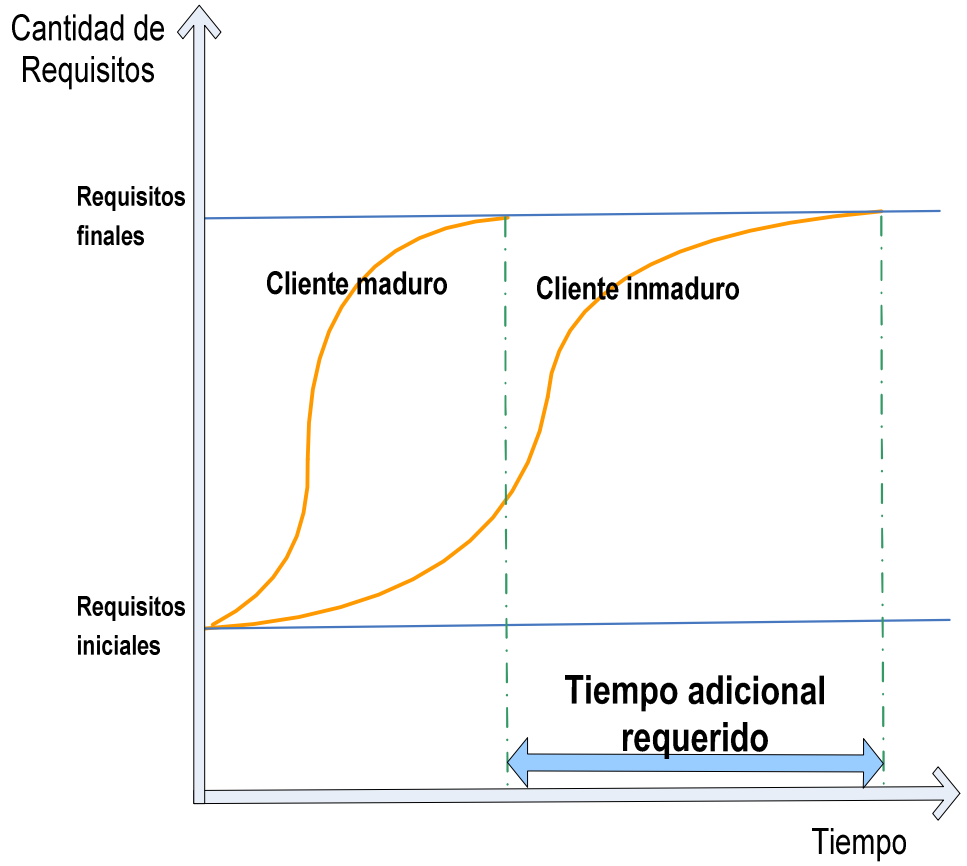

Figura 3- 17: Diferencia en el tiempo requerido por clientes inmaduros para estabilizar los requisitos, en comparación con los clientes maduros

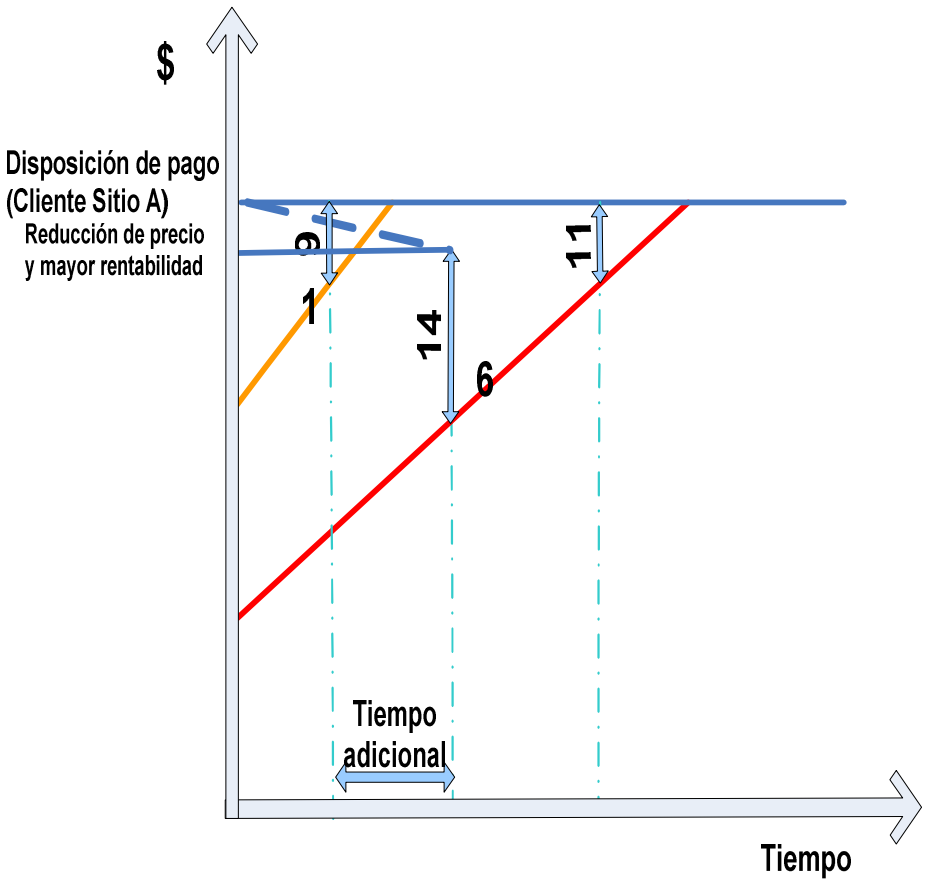

Figura 3- 18: Situación favorable para el proyecto distribuido, donde se puede reducir el precio del producto y se alcanza una rentabilidad mayor. La desventaja para el equipo distribuido está en el tiempo adicional requerido para lograr el producto

De los análisis antes planteados, se puede concluir que existen ciertas características que favorecen a los proyectos distribuidos, estas son:

- Cuando se producen diferencias entre los costos iniciales y marginales, consecuencia de la utilización de personal altamente calificado que debe elaborar una aplicación intensa en conocimientos específicos. Un ejemplo de esta situación es cuando se produce *Knowledge Process Outsourcing* (KPO) al emplear a personas con alta capacidad técnica en países emergentes, dado a que su contratación es más económica que en los países con mercados más desarrollados.

- Cuando se necesiten elaborar proyectos con plazos mayores. Por ejemplo, los proyectos que se enfocan en un nicho de mercado donde los clientes son inmaduros o existen requisitos cambiantes.

- Cuando se elaboren proyectos intensos en horas-hombre, dado que se incrementa la diferencia entre los costos iniciales y marginales.

- Cuando se utiliza una estrategia de elaborar Software como un Servicio (SaaS) o se venden servicios producto de utilizar una arquitectura (SOA), porque se minimiza el problema de la piratería. Estas estrategias está siendo paulatinamente adoptadas por las empresas, siendo una tendencia creciente. Un ejemplo de este modelo de negocio es el servicio ofrecido por la empresa Google, denominado Google Apps.

En síntesis los proyectos más apropiados, económicamente, para aplicar un desarrollo distribuido son:

- Intensos en conocimiento especializado
- Intensos en horas-hombre
- Con requisitos cambiantes

3.3 Priorización de prácticas y requisitos tecnológicos

Tabla 3-1: Priorización de prácticas para los niveles co-localizados

Nivel	# Práctica	General	Requisitos cambiantes	Intenso en RRHH
1C	1	Aceptar y gestionar los cambios	6	7
	2	Gestionar las versiones	9	11
	3	Gestionar y analizar los riesgos	10	
	4	Planificar, ejecutar y estimar el proyecto en iteraciones		
	5	Utilizar un modelo de equipo Descentralizado Controlado		
2C	6	Construir equipos de alto desempeño	15	13
	7	Establecer mecanismos eficaces de comunicación para mejorar la coordinación		14
	8	Los desarrolladores deben probar su código		16
	9	Medir el progreso objetivamente		
	10	Priorizar los requisitos a implementar		
	11	Utilizar la arquitectura de software para minimizar los requisitos de comunicación y colaboración		
3C	12	Apalancar apropiadamente la automatización de pruebas		
	13	Establecer reglas de comunicación		
	14	Utilizar un paradigma organizacional sincronizado		
	15	Utilizar una adecuada cantidad y rigurosidad de procesos		
	16	Utilizar una arquitectura basada en componentes		

Tabla 3-2: Priorización de prácticas para los niveles distribuidos

Nivel	# Práctica	Práctica aplicable a todos los proyectos	Requisitos cambiantes	Intenso en RRHH
1D	**1**	Adicionar prácticas 1C	**9**	**14**
	2	Establecer un proceso de ingeniería de requisitos bien definido y comprendido	**10**	
	3	Minimizar el grado de dispersión geográfica	**13**	
	4	Prácticas básicas de gestión de la configuración		
	5	Realizar iteraciones cortas (de una o dos semanas), que vayan entregando funcionalidades incrementalmente		
	6	Utilizar herramientas y tecnologías que apoyen la accesibilidad, colaboración y ejecución de los proyectos		
2D	**7**	Adicionar prácticas 2C	**14**	
	8	Integrar temprana y frecuentemente (práctica avanzada de gestión de la configuración)	**16**	
	9	Mejorar la coordinación y comunicación entre los sitios		
	10	Promover el intercambio de conocimiento		
	11	Promover vínculos sociales entre los integrantes		
	12	Realizar reuniones colaborativas frecuentemente, fijando reglas de estilo y frecuencia		
3D	**13**	Adicionar prácticas 3C		
	14	Organizar e integrar un conjunto consistente de versiones utilizando actividades (práctica avanzada de gestión de la configuración)		
	15	Promover y facilitar la gestión de componentes		
	16	Registrar y rastrear las solicitudes de cambio (práctica avanzada de gestión de la configuración)		

Tabla 3-3: Priorización de requisitos tecnológicos

Nivel	# Práctica	General	Requisitos cambiantes	Intenso en RRHH
1D	**1**	Armar el sistema final utilizando listas que señalen las partes y sus versiones	**12**	**13**
	2	Evitar la pérdida y la falta de identificación de las versiones de los artefactos, mediante un método efectivo para organizarlos y localizarlos	**14**	
	3	Garantizar la consistencia de la configuración, al margen del ambiente de trabajo y construcción	**16**	
	4	Implementar los cambios consistentemente, para evitar que los integrantes incurran en errores	**17**	
	5	Mantener el control de la elaboración sin que influya la cantidad de cambios realizados en el software y productos de trabajo		
	6	Poder reproducir los release anteriores		
	7	Posibilitar el desarrollo paralelo y los cambios concurrentes de los artefactos		
	8	Proporcionar ambientes de trabajo estables para aislar, a los otros miembros, de los cambios disruptivos		
	9	Registrar información de auditoría		
	10	Registrar, cómo se construyó el software y las versiones de los componentes que lo conforman		
	11	Reunir en un mismo lugar y tiempo las versiones adecuadas de los componentes para facilitar su conexión al construir la versión final del software		
2D	**12**	Control de acceso para modificar o visualizar los artefactos	**20**	**22**
	13	Controlar, medir y dirigir lo que está sucediendo en el proceso de desarrollo	**24**	**23**
	14	Modificar, integrar y fusionar los cambios realizados en un tiempo apropiado y minimizando la sobrecarga de trabajo para que se cumplan los tiempos estimados	**25**	
	15	Presentar informes de los avances de desarrollo		
	16	Proporcionar información actualizada y acceso rápido a los últimos artefactos elaborados en el proyecto		
	17	Recolectar datos que apoyen la gestión		

	18	Trazabilidad para vincular y hacer seguimiento de los requisitos, casos de prueba, planificación y artefactos		
3D	**19**	Aumentar el nivel de abstracción de la comunicación desde archivos a actividades		
	20	El jefe de proyecto debe tener acceso rápido y actualizado de las actividades asignadas a cada integrante		
	21	Facilitar el trabajo simultáneo y la recolección de información clave que agilice el proceso de integración		
	22	Los repositorios deben tolerar las fallas, ser escalables, distribuibles y replicables		
	23	Organizar los archivos y directorios para facilitar su identificación (para los proyectos en curso como en anteriores)		
	24	Registrar y rastrear las solicitudes de cambio para verificar su progreso		
	25	Tener acceso rápido y fácil para encontrar y extraer artefactos, tanto del proyecto actual como de anteriores		
	26	Vincular la gestión del proyecto con la gestión de configuración y la gestión de solicitudes de cambio		

4. METODOLOGÍA PARA LOGRAR Y MANTENERSE EN EL ESTADO ÓPTIMO (3D)

4.1 Marco teórico de las metodologías de apoyo

4.1.1 Modelo para coordinar a los equipos remotos (Múltiples Clientes/Proveedores)

Bellagio y Milligan (2005) presentan este modelo para incrementar la probabilidad de éxito en los proyectos distribuidos. Para esto, los *equipos remotos* (sub-equipos) que comparten *componentes* deben hacerlo a través de una relación ***cliente/proveedor***, es decir, el *cliente* no tiene facultades para modificar los artefactos elaborados por el *proveedor*, sino que debe solicitar o proponer que se efectúen cambios. La ventaja de este modelo radica en minimizar las dependencias entre los distintos *equipos remotos*.

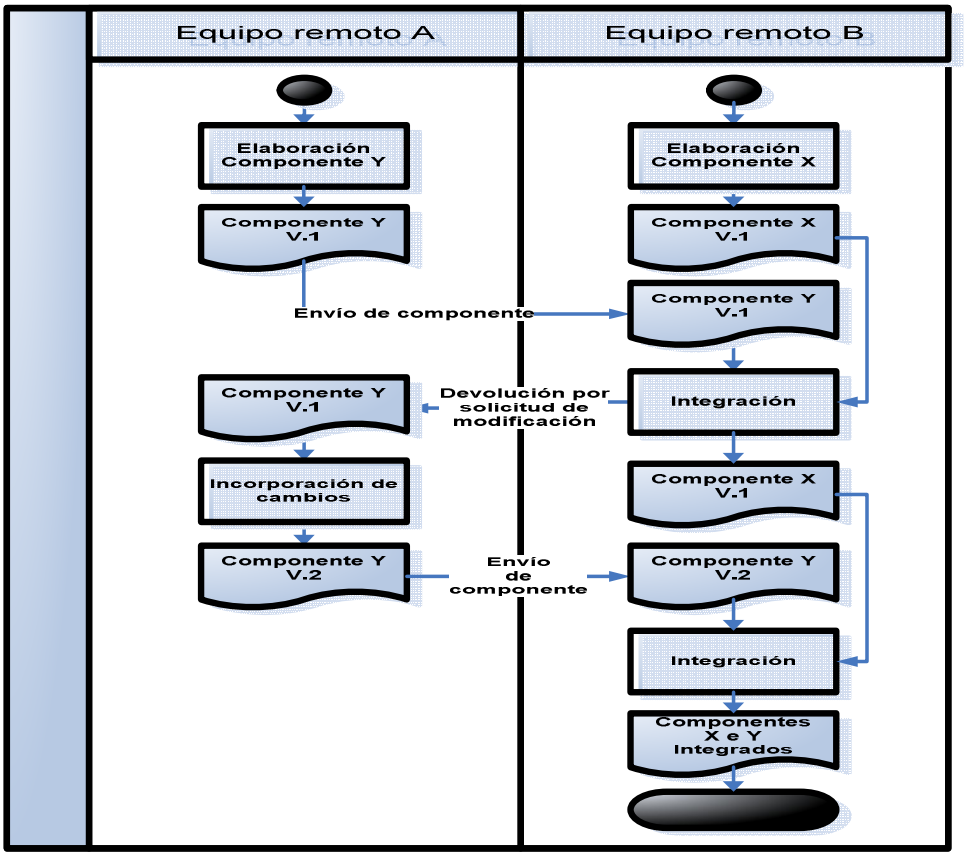

Figura 4- 1: Representa un proceso según el modelo múltiples clientes/proveedores

Para que este modelo pueda aplicarse efectivamente, se debe utilizar una arquitectura de software que pueda dividir eficazmente las funciones a desarrollar, minimizando así las necesidades de *comunicación*, *coordinación* y *control*; por ejemplo, se podría utilizar una arquitectura basada en ***componentes*** o ***servicios***, los que deben asignarse a los distintos *equipos remotos*. Finalmente, un único *sub-equipo* debe responsabilizarse de consumir e integrar todos los ***componentes*** para producir el sistema final.

La dificultad de este modelo radica en definir una adecuada arquitectura de software para que pueda evolucionar (***interfaces*** y ***componentes***). Adicionalmente, se deben considerar y superar las siguientes dificultades para mitigar parte del riesgo existente en las arquitecturas mayores:

- Capacidad para identificar una versión de un *componente*
- Poder enviar los *componentes* desde un equipo a otro de forma confiable y eficiente
- Garantizar la integración temprana y regular entre los *componentes*

Para facilitar la comunicación entre los *equipos remotos,* la *planificación del proyecto* debe incluir los siguientes puntos claves en cada hito:

- *Baselines* de los *componentes* que deben ser entregados en cada iteración
- Las dependencias del proyecto
- Las funcionalidades que deben exhibirse

Finalmente, para proporcionar una arquitectura adecuada a este modelo, Bellagio y Milligan (2005) hacen las siguientes recomendaciones:

- Establecer un equipo de arquitectura transversal al proyecto

- Tener un jefe de arquitectura con la autoridad para decidir los debates arquitectónicos.

- Poseer un jefe de proyecto global, que se responsabilice de la gestión integral del proyecto, el que debe preocuparse por resolver los problemas de carácter personal, organizacional, cultural, de procesos, entre otros.

4.1.2 Modelo de proceso espiral

Propuesto por Boehm en 1988, aún es utilizado por diversas empresas y ha servido como base para plantear nuevos modelos de procesos (por ejemplo Microsoft Solutions Framework). El *modelo espiral*, corresponde a un proceso evolutivo que proporciona el potencial para el rápido desarrollo de versiones incrementales, mediante la elaboración de prototipos que aumentan gradualmente las funcionalidades solicitadas por el cliente. Éste, se divide en *regiones de tareas* que guían el *proceso de desarrollo de software*. Las seis regiones más utilizadas son:

- ***Comunicación y coordinación***: Compuesta por tareas diseñadas para establecer una mejor comunicación y coordinación entre el cliente y el equipo desarrollador.

- ***Planificación***: Compuesta por tareas que se relacionan con la estimación de recursos, plazos y otra información relacionada con el proyecto.

- ***Análisis de riesgo***: Compuestas por tareas que evalúan los riesgos técnicos y de gestión.

- ***Ingeniería***: Compuestas por tareas diseñadas para construir una o más representaciones de la aplicación.

- ***Construcción y Release***: Compuestas por tareas diseñadas para construir, probar, instalar y proporcionar soporte al usuario.

- ***Evaluación del sistema***: Compuestas por tareas diseñadas para testear el sistema y obtener *feedback* del cliente, según su evaluación del producto.

Figura 4- 2: Modelo de Proceso Espiral. Obtenido de Pressman (2002)

4.1.3 Modelo de proceso espiral basado en componentes

El *modelo de proceso espiral basado en componentes* aplica el mismo principio que el *modelo de proceso espiral*, su diferencia radica en emplear un *paradigma orientado a objetos para* encapsular los datos y algoritmos.

Como se mencionó en los capítulos anteriores, se propone un *paradigma basado en componentes,* fundamentalmente, por los diversos beneficios que presenta cuando se implementa un desarrollo distribuido, por ejemplo: disminuye los riesgos porque facilita la utilización del modelo *Múltiples Clientes/proveedores* y permite la disminución de las interdependencias entre los *equipos remotos*.

4.1.4 Gestión de solicitudes de cambio

Es el registro, seguimiento y reporte de *solicitudes* hechas por algún *stakeholder* para pedir que se modifique: el sistema de software, un artefacto o algún proceso (RUP 1999). Algunas disciplinas relacionadas con esta área son: ***gestión de requisitos***, ***solicitud de pruebas***, ***gestión de release*** y ***gestión del proyecto***.

Una solicitud efectiva informa el origen y el impacto del problema, la solución propuesta y su costo relativo (ya sea monetario, en horas-hombre, entre otros). Éstas, generalmente se subdividen en ***solicitudes de mejoras*** y ***defectos***. Las primeras especifican una nueva característica o un cambio en el diseño del comportamiento del sistema. En cambio, los ***defectos*** son anomalías o desperfectos informados para que sean reparados.

Cuando se desea implementar un ***proceso de gestión de solicitudes de cambio,*** se deben tomar ciertas decisiones como: los *stakeholders* involucrados, las solicitudes que

se rastrearán, los datos que se recopilarán para realizar los seguimientos y la forma de rastrear dichas solicitudes.

Para algunas organizaciones con características específicas, las *solicitudes* pueden ser más complejas y multicapas (diferente granularidad). Por ejemplo, se puede definir un tipo de *solicitud* que represente un requisito de un cliente externo y, a su vez, dicha *solicitud* puede generar nuevos requisitos a nivel de ingeniería.

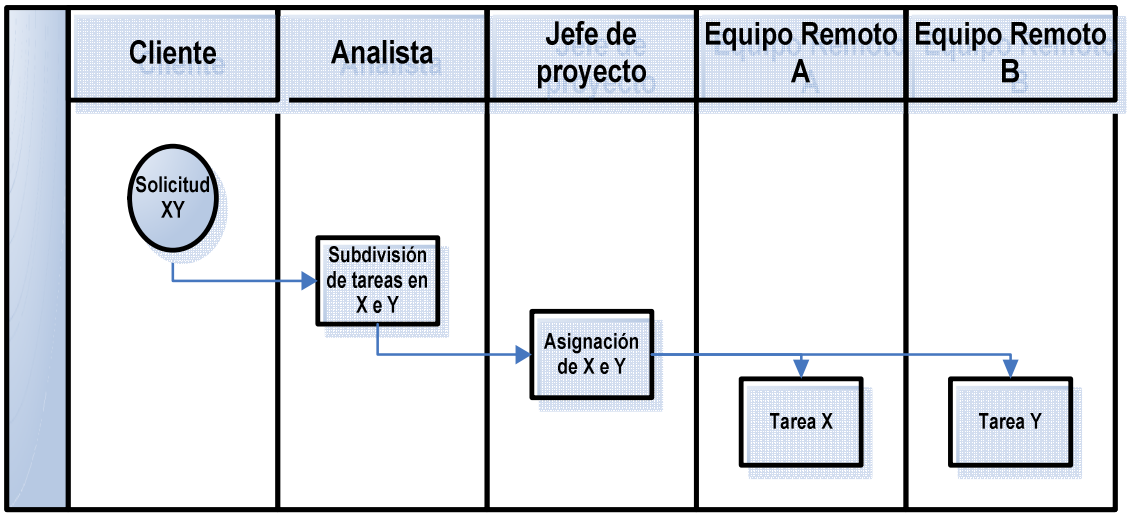

Figura 4- 3: Ejemplificación de la subdivisión de solicitudes en tareas con granularidad menor

Una vez que se identifiquen las *solicitudes* que se *rastrearán*, el siguiente paso es definir la información que se *registrará* a través del ***ciclo de vida de la solicitud***.

Por otra parte, los procesos utilizados para monitorear las *solicitudes* varían ampliamente, por ejemplo, se pueden utilizar ciertos tipos de herramientas como las de ***Change Requirement Management*** (***CRM***). Éstas, generalmente utilizan un modelo de transición basado en *estados posibles*.

Por ende, para plantear un *proceso de gestión de solicitudes de cambio*, se deben definir: los ***estados*** posibles, los ***motivos para crear una solicitud***, las ***transiciones*** entre los estados y los ***criterios*** para saltar desde un estado a otro.

Los ***tipos de solicitudes*** y los ***modelos de estado*** varían ampliamente dependiendo del tipo de proyecto y organización. Sin embargo, las herramientas de ***CRM*** generalmente incluyen ciertos estados como los siguientes:

- ***Presentación***: Es el registro de la solicitud, generalmente difiere del origen y del tipo de información recopilada. Los datos claves que usualmente se capturan son:

    - Importancia de la solicitud para el cliente.
    - La mayor cantidad de detalles posibles.
    - Identificación del solicitante original, así se posibilita que los ingenieros puedan hacer preguntas aclaratorias.

    Los defectos, por su parte, también surgen de un amplio conjunto de fuentes, los datos claves que deben registrarse son:

    - Cómo fue descubierto el defecto
    - Cómo reproducirlo
    - Su gravedad
    - Quién lo descubrió

- ***Evaluación***: La solicitud es evaluada, categorizada y priorizada. En este paso se revisan las nuevas solicitudes y se toman las determinaciones necesarias. Por ejemplo, los ingenieros evalúan los defectos, en cambio, las *solicitudes de mejoras* son priorizadas.

- ***Decisión***: Se basan en la información recopilada en el ***estado evaluación*** para indicar si se debe implementar el cambio en un ***release*** determinado o debe ser pospuesto o rechazado. En esta etapa también se señala el orden de implementación. Para esto, se deben considerar los siguientes factores:

  - ***Fase del ciclo de desarrollo de software***: Al comienzo se deben maximizar los cambios para ajustar los requisitos del cliente, en cambio, al final del ***ciclo de desarrollo*** se deben realizar sólo los cambios necesarios para finiquitar el proyecto, porque en esta etapa los cambios son disruptivos, es decir, contienen alto riesgo y pueden hacer que fracase el proyecto, en cuanto a sobrepasar los tiempos y costos presupuestados.

  - ***Tamaño del esfuerzo de desarrollo***: Las organizaciones de mayor tamaño deben poseer un proceso formal de aprobación de los cambios, por ejemplo incluyendo plantillas o tablas para realizar la evaluación.

- ***Implementación***: Se crean o se efectúan los cambios para implementar la solicitud. Se debe tener presente actualizar la documentación del sistema, para que los cambios sean considerados.

- ***Verificación***: En esta etapa, toma lugar el testeo y la revisión de la documentación. Las pruebas de las *solicitudes de mejoras* confirman que los cambios satisfacen las necesidades. En cambio, las pruebas de los defectos ratifican que estos han sido corregidos en un release o construcción oficial.

- ***Finalización***: Se cierra la solicitud y se notifica al demandante que se satisfizo la petición o que se decidió no satisfacerla.

4.1.5 El modelo concurrente

Las principales ventajas de un *modelo de desarrollo concurrente* son permitir la ***Gestión de Solicitudes*** y el ***trabajo basado en actividades***. Es así como se mejora la visibilidad, coordinación, comunicación y control del proyecto, lo que incrementa su probabilidad de éxito.

Este *modelo de procesos* se representa mediante *actividades*, *estados* asociados a dichas actividades, *rutas* entre estados*, motivos de creación y causas para transitar* desde un estado a otro. Es similar a la disciplina de *solicitudes de cambio*, la diferencia radica en que este modelo tiene un rango de acción más amplío, pudiendo incorporar más elementos que puedan ser gestionados de igual forma que las *solicitudes de cambio*. Por ejemplo, ciertas actividades que usualmente se gestionan, se relacionan con:

- Las solicitudes de cambio
- Los defectos
- Los requisitos de los usuarios y el cliente
- Las problemáticas que deben ser resueltas
- Los riegos que deben ser mitigados

Como se ha señalado, es común que ésta metodología sea adaptada a las necesidades propias de los proyecto y las organizaciones involucradas (desarrolladora y cliente). Adaptar esta metodología involucra considerar: los campos, los motivos de creación, los estados posibles, las causas para saltar desde un estado a otro y sus rutas.

Figura 4- 4: Estados, flujos y causas de las transiciones de una actividad según una metodología de desarrollo concurrente. Extraído de Microsoft Solutions Framework (MSF) versión 4 CMMI

La figura anterior representa el esquema de una *actividad* en un *modelo de desarrollo concurrente*, donde dicha *actividad* puede situarse en cualquiera de los *estados* planteados (*Propuesto*, *Activo*, *Resuelto* y *Cerrado*), las *transiciones* desde un *estado* a otro se representan por las flechas que indican el flujo que debe seguir la *actividad* al momento de cambiar de *estado*. Además, el *modelo de desarrollo concurrente* define una serie de acontecimientos que gatillan las transiciones de un *estado* a otro; por ejemplo existen dos causas para pasar desde el *estado Propuesto* a *Activo*, las que son *Aprobado* e *Investigar*.

Aplicar este modelo faculta la recolección de datos que, principalmente, sirven para apoyar una gestión más efectiva en esta clase de proyectos. Esto, finalmente se traduce en una mejor estimación de los plazos y costos, como también minimiza el caos producido por una escasa comunicación, coordinación y control, disminuyendo los costos globales. Es así como, se alinean con los proyectos clasificados 3D.

Figura 4- 5: Relaciones entre un modelo concurrente, las métricas del proyecto y la Gestión de la Configuración. Extraído de RUP 7 (2005)

Algunos gráficos que se pueden obtener mediante la información proporcionada por esta metodología son:

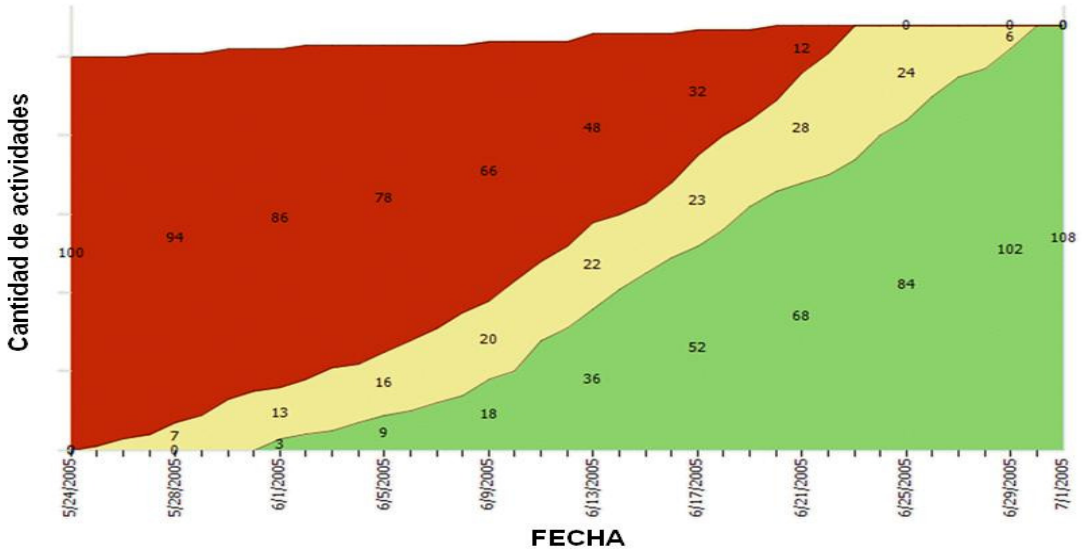

Figura 4- 6: Gráfico de la evolución de las actividades en el tiempo para determinar cuánto trabajo queda por realizar

Rojo= actividades que no se han concretado (estado Activo)
Amarillo= actividades deben aprobarse para que sean concretadas (estado Resuelto)
Verde= actividades que se han concretado (estado Cerrado)

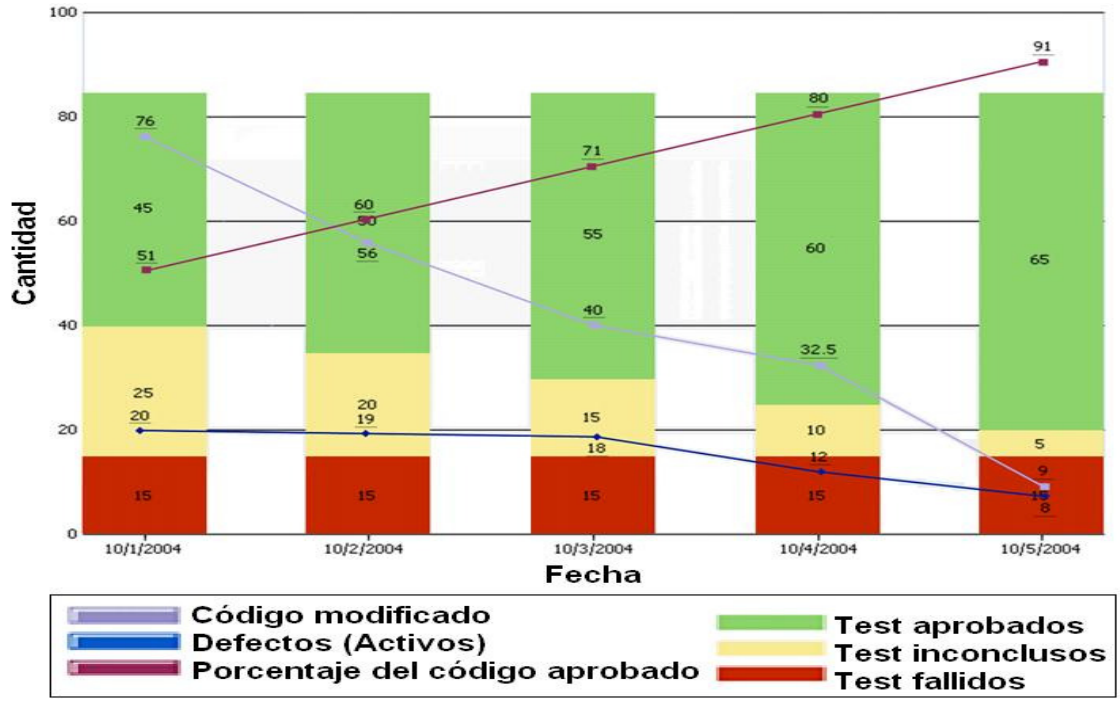

Figura 4- 7: relaciones entre: código modificado, defectos por resolver, porcentaje de código aprobado y los test (aprobados, inconclusos y fallidos)

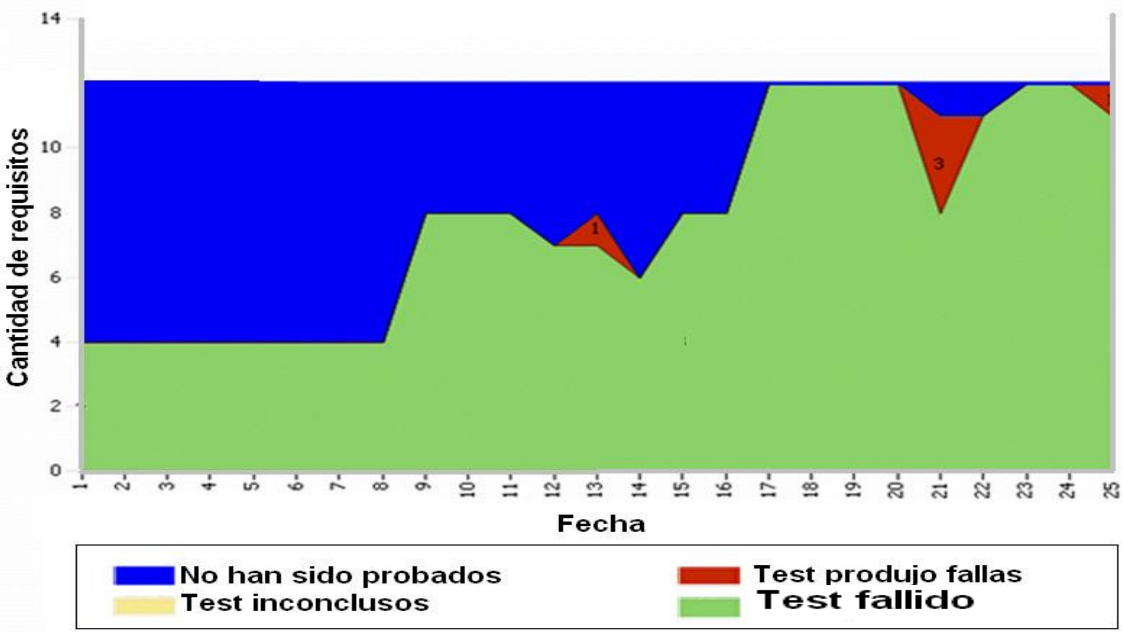

Figura 4- 8: Evolución de la aprobación de los requisitos

4.2 Diseño de una metodología concurrente adaptada al escenario distribuido

Considerando la información antes expuesta, se determinó un conjunto de *actividades* aplicables a un *desarrollo distribuido de software* (independiente del tipo de proyecto). Las *actividades* tienen la función de *registrar*, *comunicar* y *realizar seguimiento* a las *tareas* involucradas en el proceso de elaboración de un sistema de software, su finalidad es poder mejorar la *comunicación, coordinación* y *control*. Además, se identificó cierta información que usualmente es modificada y actualizada durante gran parte del ciclo de vida del desarrollo. La divulgación y accesibilidad de dicha información puede facilitarse cuando se incorpora a una herramienta que gestione las *actividades* de un *modelo de desarrollo concurrente*. Por ende, la proposición de una metodología de estas características adaptada al escenario distribuido también considera la incorporación de plantillas de documentos para estandarizar y protocolizar la comunicación y el registro de información.

Las *actividades* propuestas y sus definiciones son:

- ***Defecto*** (reparar defecto): Comunica que existió o existe un problema en la aplicación, su objetivo es crear un informe estandarizado y preciso del defecto para comprender su impacto. Los campos de esta *actividad* se diseñaron para reproducir el error y así facilitar su solución.

- ***Problemática*** (atender problemática): Comunica un evento que bloqueó o bloquea el trabajo, está diseñado para comunicar acciones correctivas que resuelvan dicho problema.

- ***Requisito*** (incorporar requisito): Comunica una característica que debe poseer la aplicación para atender los requisitos del cliente.

- ***Revisión Técnica Formal*** (realizar revisión y comunicar resultados): Actividad que tiene como principales funciones, coordinar una Revisión Técnica Formal y comunicar sus resultados. Ha sido diseñada para incluir información que comprueba el cumplimiento de estándares, permitiendo garantizar un producto de mejor calidad.

- ***Riesgo*** (mitigar riesgo): Comunica una condición o evento que podría tener un efecto negativo en el desarrollo del proyecto. Tiene como función comunicarlo para que sea gestionado.

- ***Solicitud de Cambio*** (incorporar cambios solicitados): Comunica una *solicitud de cambio* propuesta por un stakeholder para que se incorpore en: el sistema de software, un proceso de desarrollo o un artefacto determinado.

- ***Tarea*** (realizar tarea): Es una actividad genérica que comunica la necesidad de realizar cierto trabajo.

- ***Test*** (realizar test y comunicar sus resultados): Es una actividad que asigna la realización de una prueba e informa los resultados obtenidos, tanto de forma estándar como de forma detallada.

A continuación se propone, más detalladamente, el diseño de un modelo de desarrollo concurrente para lograr y mantenerse en el estado optimizado 3D. Para cada una de las actividades, la propuesta define:

- La información que debe almacenarse
- Los motivos para crear una actividad
- Los estados potenciales
- Las causas para saltar de un estado a otro
- Las rutas posibles entre estados.

Los campos propuestos, que se aplican a todas las actividades son:

Tabla 4-1: Campos propuestos aplicables a todas las actividades

Campos comunes para todas las actividades	
Id:	Número de identificación de la actividad
Título de la actividad:	Nombre para identificar a la actividad
Asignado a:	Persona (s) a las que se les asigna la actividad
Tipo:	El tipo de actividad (Defecto, Tarea, Test, etc)
Información histórica:	Información histórica adicional
Iteración relacionada:	Iteración donde se creó la actividad
Estado actual:	Por ejemplo: Propuesto, Abierto, Resuelto o Cerrado
Motivos del estado:	Responde la pregunta ¿Por qué no está cerrado?
Fecha de cambio de estado:	Responde la pregunta ¿Cuándo se modificó?
Modificado por:	Responde la pregunta ¿Quién lo modificó?
Fecha de creación:	Para registrar la cantidad de días demorados para cerrarlo
Creado por:	Responde ¿Quién identificó la necesidad? y ¿Quién posee más información del motivo?
Descripción:	Es una breve descripción para explicar mejor el contexto de la actividad
Problema:	Responde ¿Existen problemas adicionales para resolver esta actividad?
Criterio de cierre:	Responde ¿Qué debe cumplirse para poder cerrar la actividad?
Fecha de cierre:	Responde ¿Cuándo fue cerrada la actividad?
Cerrado por:	Responde ¿Quién cerró la actividad y es responsable por dicha acción?
Prioridad:	Responde comparativamente ¿Qué tan relevante es la actividad para el éxito del proyecto?
Estimación fecha de cierre (inicial):	Responde ¿Cuándo podría cerrarse? (según la primera estimación hecha)
Estimación horas hombre (inicial):	Responde ¿Cuánto es fuerzo es necesario para atenderla? (según la primera estimación hecha)
Fecha comienzo (inicial):	Responde ¿Cuándo se empezará a trabajar en esta actividad? (según la primera estimación hecha)
Posible fecha de cierre (actual):	Responde ¿Cuándo podría cerrarse? (según la última estimación hecha)
Estimación horas hombre (actual):	Responde ¿Cuánto es fuerzo es necesario para atenderla? (según la última estimación hecha)
Fecha comienzo (actual):	Responde ¿Cuándo se empezará a trabajar en la tarea? (según la última estimación hecha)
Fecha límite:	Responde ¿Plazo máximo para concretar la tarea?
Porcentaje completado:	Responde ¿Cuánto se lleva? y ¿Cuánto falta para terminar? (en porcentaje)
Equipo responsable:	Responde ¿Cuál es el equipo responsable? (por ejemplo “Sitio A” o “Sito B”)
Ubicación archivo relacionado:	Responde ¿Dónde se puede encontrar información adicional?
Versión archivo relacionado:	Responde ¿A qué versión corresponde?

Detalles específicos para una Actividad “Defecto”:

Tabla 4-2: Detalles específicos para una actividad “Defecto”

Motivo para crear la actividad	Información adicional estándar
Se descubrió error de interfaz	Descubierto por los usuarios
Se descubrió error de almacenamiento	Descubierto por los testers
Se descubrió error de ambiente	Descubierto por los desarrolladores
Se descubrió error de construcción	Descubierto por los arquitectos
Se descubrió error de conversión de formato	Descubierto por otro rol
Se descubrió error de estrés	
Se descubrió error de funcionamiento	
Se descubrió error de implementación	
Se descubrió error de integración	
Se descubrió error de interfaz usuario	
Se descubrió error de lógica	
Se descubrió error de regresión	
Se descubrió error de seguridad	
Se descubrió error de transacción	
Se descubrió error de Runtime	
Se descubrió otro tipo de error	

Tabla 4-3: Detalle sobre la información que almacena en una actividad Defecto

Campos particulares actividad Defecto	
Gravedad:	Cuantifica el impacto negativo en el éxito del proyecto
Disciplina relacionada:	Por ejemplo (Requisitos, Diseño, Implementación, Pruebas, etc.)
ID de la prueba que encontró el error:	Identifica la prueba que encontró el error
Tipo de prueba que encontró el error:	Detalla sobre el tipo de error
Nombre de la prueba que encontró el error:	Información para poder repetir el error
Ruta para acceder a la prueba:	Información para poder repetir el error
Release donde se encontró:	Versión del release donde se descubrió el error
Síntomas del error:	Información que ayuda a identificar el problema
Pasos para reproducirlo:	Información para poder repetir el error e identificar el problema
Corrección propuesta:	Si se propone información adicional que ayude a la corrección
Entorno donde se encontró:	Detalles para reproducir e identificar el error
Causa principal:	Responde ¿Por qué es necesario repararlo?
Cómo se encontró:	Información para poder repetir el error e identificar el problema

Tabla 4-4: Estados potenciales, causas de transición entre estados y rutas posibles, para una actividad Defecto

	Estado		Causa transición			
	Inicial	Final				
1	Propuesto	Activo	Aceptado	Investigar		
2	Propuesto	Resuelto				
3	Propuesto	Cerrado	Pospuesto	Rechazado	Duplicado	
4	Activo	Propuesto	Investigación terminada			
5	Activo	Resuelto	Corregido	Corregido por diseño		
6	Activo	Cerrado	Propuesto	Duplicado	Ningún error	No se puede reproducir
7	Resuelto	Propuesto				
8	Resuelto	Activo	Sin corregir			
9	Resuelto	Cerrado	Confirmada corrección			
10	Cerrado	Propuesto				
11	Cerrado	Activo	Cerrado por error	Regresión		
12	Cerrado	Resuelto				

Detalles específicos para una actividad “Problemática”:

Tabla 4-5: Causas para crear una actividad Problemática e información adicional

Motivo para crear la actividad	Información adicional estándar
Desviación de las especificaciones	Relacionado con el tamaño del proyecto
Documentación imprecisa o incompleta	Relacionado con el equipo
Se detectó error en la lógica del diseño	Relacionado con un rol determinado
Se detectó error en la representación de los datos	
Se detectó error de transcripción desde el diseño al lenguaje de programación	
Especificaciones incompletas	
Inconsistente y/o ambigua interfaz humano/computador	
Incumplimiento en los estándares de programación	
Interfaces inconsistentes	
Mala interpretación de la comunicación con el cliente	
Pruebas incompletas o erróneas	
Otro motivo	

Tabla 4-6: Detalle sobre la información que almacena en una actividad Problemática

Campos particulares actividad Problemática	
Release en el cual se resolvió el problema:	Responde ¿Dónde y Cómo se puede verificar su solución?
Iteración donde se resolvió el problema:	Responde ¿Cuándo se resolvió el problema?
Identificación de las causas:	Responde ¿Cuáles son las causas posibles? (para evitar uno nuevo)
Soluciones posibles:	Responde ¿Cómo se podría solucionar?
Impacto en el éxito del proyecto:	Responde ¿Cuáles pueden ser sus consecuencias?
Plan de acción correctiva:	Responde ¿Cómo se enfrentará el problema?
Puede agravarse:	Responde ¿Es un problema que puede empeorar? ¿Cómo podría empeorar?

Tabla 4-7: Estados potenciales, causas de transición entre estados y rutas posibles, para una actividad Problemática

	Estado		Causa transición		
	Inicial	**Final**			
1	**Propuesto**	**Activo**	Aceptado	Investigar	
2	**Propuesto**	**Resuelto**			
3	**Propuesto**	**Cerrado**	No es un problema		
4	**Activo**	**Propuesto**	Investigación terminada		
5	**Activo**	**Resuelto**	Resuelto		
6	**Activo**	**Cerrado**	Superado por otro evento		
7	**Resuelto**	**Propuesto**			
8	**Resuelto**	**Activo**	Reprocesar		
9	**Resuelto**	**Cerrado**	Resolución aceptada		
10	**Cerrado**	**Propuesto**			
11	**Cerrado**	**Activo**	Cerrado por error	Se ha vuelto a abrir	ha ocurrido nuevamente
12	**Cerrado**	**Resuelto**			

Detalles específicos para una actividad “Requisito”:

Tabla 4-8: Causas para crear una actividad Requisito e información adicional estándar

Motivo para crear la actividad	**Información adicional estándar**
Definir Actor	Requisito clave
Definir requisito Arquitectónico	Requisito riesgoso
Definir Caso de Uso	
Definir Escenario	
Definir Historia de Uso	
Definir requisito Individual	
Definir requisito SQA	
Otro motivo	

Tabla 4-9: Detalle sobre la información que almacena en una actividad Requisito

Campos particulares actividad Requisito	
Jerarquía:	Responde ¿Qué requisitos deben atenderse antes de éste?
Release:	Responde ¿Dónde se puede apreciar su implementación?
Evaluación del impacto:	Responde ¿Qué impacto puede tener en el éxito del proyecto?
Prueba de aceptación del usuario:	Responde ¿Cómo medirá la aceptación el usuario?
Pruebas de garantía:	Responde ¿Cómo se medirá la garantía de la funcionalidad?
Expertos en la materia:	Responde ¿Se necesita un experto? ¿Qué características debe poseer?, ¿Quién es el experto? ¿Cómo ubicarlo?

Tabla 4-10: Estados potenciales, causas de transición entre estados y rutas posibles, para una actividad Requisito

	Estado		Causa transición		
	Inicial	Final			
1	Propuesto	Activo	Aceptado	Investigar	
2	Propuesto	Resuelto			
3	Propuesto	Cerrado	Rechazado		
4	Activo	Propuesto	Investigación terminada	Pospuesto	
5	Activo	Resuelto	Código completado y prueba sistema superada		
6	Activo	Cerrado	Dividido	Abandonado	Fuera de ámbito
7	Resuelto	Propuesto			
8	Resuelto	Activo	Prueba o validación no superada		
9	Resuelto	Cerrado	Prueba o validación superada		
10	Cerrado	Propuesto			
11	Cerrado	Activo	Cerrado por error	reintroducido en ámbito	
12	Cerrado	Resuelto			

Detalles específicos para una actividad “Revisión Técnica Formal”:

Tabla 4-11: Causas para crear una actividad Revisión Técnica Formal

Motivo para crear una actividad Revisión Técnica Formal
Se descubrió errores en la función
Se descubrieron errores en la implementación
Se descubrieron errores lógicos
Se garantiza la utilización de estándares
Se verifica que se lograron los requisitos
Otro motivo

Tabla 4-12: Detalle sobre la información que almacena en una actividad Revisión Técnica Formal

Campos particulares actividad Revisión Técnica Formal	
Propósito revisión:	Responde ¿Qué se medirá? y ¿Cómo se medirá?
Elementos revisados:	Responde ¿Lo que se revisó?
Feedback:	Entrega comentarios sobre la reunión que permitan hacer mejoras
Fecha y lugar reunión:	Responde ¿Cuándo y dónde se realizó la reunión?
Personas involucradas:	Responde ¿Quiénes deben participar? Y ¿Quiénes participaron?

Tabla 4-13: Estados potenciales, causas de transición entre estados y rutas posibles, para una actividad Requisito

	Estado		**Causa transición**	
	Inicial	**Final**		
1	**Propuesto**	**Activo**		
2	**Propuesto**	**Resuelto**		
3	**Propuesto**	**Cerrado**		
4	**Activo**	**Propuesto**		
5	**Activo**	**Resuelto**	Aceptado con cambios principales	Aceptado con cambios secundarios
6	**Activo**	**Cerrado**	Aceptado	
7	**Resuelto**	**Propuesto**		
8	**Resuelto**	**Activo**	Cambios principales completados	Cambios secundarios completados
9	**Resuelto**	**Cerrado**	Cambios secundarios completados	
10	**Cerrado**	**Propuesto**		
11	**Cerrado**	**Activo**	Cerrado por error	
12	**Cerrado**	**Resuelto**		

Detalles específicos para una actividad “Riesgo”:

Tabla 4-14: Causas para crear una actividad Riesgo

Motivo para crear la actividad
Causado por Características cliente
Causado por una imprecisa definición del proyecto
Causado por la naturaleza del negocio
Causado por la naturales del producto
Causado por la naturaleza del proyecto
Causado por el ambiente de desarrollo
Causado por la falta de experiencia y/o tamaño del equipo
Causado por factores técnico
Causo por inmadurez de la tecnología
Otra causa

Tabla 4-15: Detalle sobre la información que almacena una actividad Riesgo

Campos particulares actividad Riesgo	
Jerarquía:	Responde ¿Cuáles riesgo deben ser atendidos antes y después?
Gravedad:	Responde ¿Puede impedir el éxito del proyecto? ¿Cómo?
Iteración donde se encontró:	Responde ¿Cuándo y dónde se encontró?
¿Bloqueado o atendido?:	Responde si se mitigó en parte o totalidad
Gatilladores del riesgo:	Responde ¿Qué evento transforma al riesgo en un problema?
Plan para mitigarlo o impedirlo:	Responde ¿Qué acciones se realizarán?
Estimación horas hombre:	Responde ¿Qué esfuerzo se requiere para atenderlo?
Impactos posibles:	Responde ¿Qué consecuencias puede causar? Y ¿A quién afecta?
Duración impactos:	Responde ¿Cuánto tiempo pueden durar los efectos negativos?
Probabilidad de ocurrencia:	Responde ¿Qué tan probable es que ocurra?

Tabla 4-16: Estados potenciales, causas de transición entre estados y rutas posibles, para una actividad Requisito

	Estado		Causa transición		
	Inicial	Final			
1	Propuesto	Activo	Reducción desencadenada		
2	Propuesto	Resuelto			
3	Propuesto	Cerrado	No es un riesgo	Aceptado	
4	Activo	Propuesto			
5	Activo	Resuelto	Acción de reducción completada		
6	Activo	Cerrado	Superada por eventos	Eliminado	No es un riego
7	Resuelto	Propuesto			
8	Resuelto	Activo	Mitigación insatisfactoria		
9	Resuelto	Cerrado	Mitigado		
10	Cerrado	Propuesto			
11	Cerrado	Activo	Cerrado por error		
12	Cerrado	Resuelto			

Detalles específicos para una actividad “Solicitud de Cambio”:

Tabla 4-17: Causas para crear una actividad Solicitud de Cambio e información adicional estándar

Motivo para crear la actividad	Información adicional estándar
Para adaptar el sistema	Afecta la arquitectura
Para modificar la planificación	Afecta la experiencia del usuario
Para cambiar ciertas herramienta	Afecta a los test
Para modificar ciertos procesos	Afecta el desarrollo
Para modificar la definición de un rol	
Para prevenir problemas o riesgos	
Para mejoras el sistema actual	
Otro motivo	

Tabla 4-18: Estados potenciales, causas de transición entre estados y rutas posibles, para una actividad Solicitud de Cambio

	Estado		Causa transición	
	Inicial	Final		
1	Propuesto	Activo	Aceptado	Investigar
2	Propuesto	Resuelto		
3	Propuesto	Cerrado	Rechazado	
4	Activo	Propuesto	Investigación terminada	
5	Activo	Resuelto	Código completado y sistema aprobado	
6	Activo	Cerrado	Abandonado	fuera de ámbito
7	Resuelto	Propuesto		
8	Resuelto	Activo	validación no superada	
9	Resuelto	Cerrado	Revisión o prueba superada	
10	Cerrado	Propuesto		
11	Cerrado	Activo	Cerrado por error	
12	Cerrado	Resuelto		

Detalles específicos para una actividad "Tarea":

Tabla 4-19: Detalle sobre la información que almacena únicamente una actividad Tarea

Campos particulares actividad Tarea	
Prioridad:	Responde ¿Qué tan importante es la realización de la tarea para el éxito del proyecto?
Disciplina relacionada:	Por ejemplo (Requisitos, Diseño, Implementación, Pruebas, etc.)
Release:	Responde ¿Dónde se puede apreciar su implementación?
Iteración:	Responde ¿Cuándo estará cerrada esta tarea?
¿Requiere revisión?:	si es que fuera necesario responde ¿Quién debe revisarla?
¿Requiere comprobación?:	Si es que fuera necesario responde ¿Quién debe comprobarla?
Jerarquía de la tarea:	Responde ¿Qué tareas deben ser realizadas previamente? Y ¿A cuáles afectará?

Tabla 4-20: Estados potenciales, causas de transición entre estados y rutas posibles, para una actividad Solicitud de Cambio

	Estado		Causa transición				
	Inicial	Final					
1	Propuesto	Activo	Aceptado	Investigar			
2	Propuesto	Resuelto					
3	Propuesto	Cerrado	Rechazado				
4	Activo	Propuesto	Investigación terminada				
5	Activo	Resuelto	Completado	Requiere revisión o prueba			
6	Activo	Cerrado	Aplazado	Cortada	Cancelada	Superada por eventos	Completada, no requiere revisión o prueba
7	Resuelto	Propuesto					
8	Resuelto	Activo	Revisión o prueba no superada				
9	Resuelto	Cerrado	Revisión o prueba superada				
10	Cerrado	Propuesto					
11	Cerrado	Activo	Cerrado por error	Reactivada			
12	Cerrado	Resuelto					

Detalles específicos para una actividad “Test”:

Tabla 4-21: Causas para crear una actividad Test

Motivo para crear la actividad Test
Realizar prueba de aceptación
Realizar prueba de ambiente
Realizar prueba de funcionamiento y estrés
Realzar prueba de Integración
Realizar prueba de Interfaz de usuario
Realizar prueba de regresión
Realizar otro tipo de prueba

Tabla 4-22: Detalle sobre la información que almacena únicamente una actividad Test

Campos particulares actividad Test	
Jerarquía:	Responde ¿Cuáles test se deben realizar antes?
Criterios de aceptación:	Responde ¿Qué comportamiento o resultados se espera?
Expertos en la materia:	Si se requieren los comentarios o la aprobación de una persona con conocimientos especiales en la materia

Tabla 4-23: Estados potenciales, causas de transición entre estados y rutas posibles, para una actividad Test

	Estado		**Causa transición**	
	Inicial	**Final**		
1	**Propuesto**	**Activo**		
2	**Propuesto**	**Resuelto**		
3	**Propuesto**	**Cerrado**		
4	**Activo**	**Propuesto**		
5	**Activo**	**Resuelto**	Aceptado con cambios principales	Aceptado con cambios secundarios
6	**Activo**	**Cerrado**	Aceptado	
7	**Resuelto**	**Propuesto**		
8	**Resuelto**	**Activo**	Cambios principales completados	Cambios secundarios completados
9	**Resuelto**	**Cerrado**	Cambios secundarios completados	
10	**Cerrado**	**Propuesto**		
11	**Cerrado**	**Activo**	Cerrado por error	
12	**Cerrado**	**Resuelto**		

Como se ha mencionado, utilizar *actividades* permite un lenguaje común que facilita la *comunicación*, *coordinación* y *control*, incrementando la probabilidad de éxito en los proyectos distribuidos. Basándose en esta idea, se proponen las siguientes plantillas de documentos para que sean gestionadas como *actividades* mediante una herramienta que implemente y automatice un modelo de elaboración concurrente. Las plantillas de documentos propuestas son:

- ***Activo de software***: Son plantillas para facilitar la utilización y reutilización de *activos de software* (por ejemplo: archivos fuentes, ejecutables, librerías, componentes, servicios, subsistemas, procesos de negocios, elementos de modelación, especificaciones, manuales y otros tipos de documentos).

- ***Ámbito del software***: Plantilla que involucra ciertos aspectos del proyecto como: control, datos a procesar, funciones, rendimiento, restricciones, entre otros.

- ***Claridad organizacional***: Plantilla que define aspectos del proyecto como: misión del equipo; visión de la aplicación; cómo se medirá el éxito del equipo y proyecto; *stakeholders*; procedimientos para realizar el trabajo, entre otros.

- ***Planificación de la iteración***: Plantilla para definir y registra los aspectos claves que debe abordar las iteraciones.

- ***Planificación del proyecto***: Plantilla para definir y registrar los aspectos claves que debe abordar una *planificación*.

- ***Reunión***: Plantilla que brinda información para realizar y registra los resultados de las reuniones.

- ***Rol***: Plantilla para definir las responsabilidades, funciones, tareas y otros aspectos que detallen las acciones de un rol. Por ejemplo, algunos roles son: analista de negocio, jefe de producto, arquitecto de infraestructura, arquitecto de software, desarrollador, experto en la materia, gestor de configuración, jefe de arquitectura, jefe de desarrollo, jefe de proyecto, jefe de pruebas, tester, ingeniero de release & build, entre otros.

Tabla 4-24: Campos comunes para los documentos

Campos comunes para todos los documentos	
Id:	Número de identificación del documento
Asignado a:	Persona (s) a las que se les asigna su elaboración
Tipo de documento:	Por ejemplo: Claridad organizacional, Reunión, Rol, etc
Información histórica:	Información histórica adicional
Modificado por:	Responde la pregunta ¿Quién lo modificó?
Fecha de creación:	Para registrar la cantidad de días demorados para cerrarlo
Descripción:	Es una breve descripción para explicar mejor el contexto de la actividad
Problema:	Responde ¿Existen problemas adicionales para realizar el documento?
Fecha comienzo:	Responde ¿Cuándo se empezará a trabajar en el documento?
Fecha límite:	Responde ¿Plazo máximo para concretar el documento?
Equipo responsable:	Responde ¿Cuál es el equipo responsable? (por ejemplo “Sitio A” o “Sito B”)
Ubicación archivo relacionado:	Responde ¿Dónde se puede encontrar información adicional?

Información que registran las plantillas de documento Activo de Software:

Tabla 4-25: Planilla de documento Activo de Software

Campos particulares documento Activo de Software
ID del activo
Tipo de activo
¿Para qué sirve?
¿Qué problema resuelve?
¿Resuelve un problema recurrente?
¿Es independiente?
¿Es portátil?
¿Es flexible?
¿Es extensible?
¿Puede ser reutilizado en otros contextos?
¿En qué contextos puede ser reutilizado?
¿Cuándo debe re-utilizarse?
¿Cómo debe re-utilizarse?
¿Cuál es su ubicación?

Información que registra la plantilla de documento Ámbito del Software:

Tabla 4-26: Planilla de documento Ámbito del Software

Campos particulares documento Ámbito del Software
Cantidad potencial de usuarios
Ciclo de vida de la aplicación
Control
Datos a procesar
Estabilidad de los requisitos
Funciones
Interfaces
Madurez de la tecnología aplicable
Personal del proyecto
Rendimiento
Restricciones
Tamaño del proyecto
Otro

Información que registra la plantilla de documento Claridad Organizacional:

Tabla 4-27: Planilla de documento Claridad Organizacional

Campos particulares documento Claridad Organizacional
Misión del equipo
Modo de medición del éxito del equipo
Modo de medición del éxito del proyecto
Reglas de comunicación
Reglas de control
Reglas de coordinación
Stakeholders del proyecto
Valores
Visión de la aplicación
Otra

Información que registra la plantilla de documento Planificación de la Iteración:

Tabla 2-28: Plantilla de documento Planificación Iteración

Campos particulares documento Planificación Iteración
ID Iteración
Tareas básicas de la iteración
Tareas secundarias de la iteración
Expertos que se utilizarán en la iteración
Grado de rigor de los procesos
Cantidad de puntos de función elaborados
Cantidad de puntos de función que deben elaborarse en la iteración (inicial)
Cantidad de puntos de función que deben elaborarse en la iteración (actual)
Cantidad de requisitos elaborados
Cantidad de requisitos que deben elaborarse en la iteración (inicial)
Cantidad de requisitos que deben elaborarse en la iteración (actual)
Cantidad de líneas de código fuente que posee el proyecto
Documentos que se deben elaborar o actualizar
Documentos elaborados
Fecha fin iteración (inicial)
Fecha fin iteración (actual)
Días de retraso (actual)

Información que registra la plantilla de documento Planificación del proyecto

Tabla 4-29: Plantilla de documento Planificación del Proyecto

Campos particulares documento Planificación del Proyecto
Cliente que ha solicitado el producto
Características del producto
Personal que realizará la elaboración del software
Funciones que deben entregarse al final del proyecto
Fecha propuesta para entregar las funciones (inicial)
Fecha propuesta para entregar las funciones (actual)
Funciones que deben entregarse al final de la iteración actual
Fecha propuesta para entregar las funciones de la iteración actual (inicial)
Fecha propuesta para entregar las funciones de la iteración actual (actual)
Herramientas que se utilizarán
Estimación dinero requerido por el proyecto (inicial)
Fecha de estimación dinero requerido por el proyecto (inicial)
Estimación dinero requerido por el proyecto (actual)
Fecha de estimación dinero requerido por el proyecto (actual)
Estimación esfuerzo requerido por el proyecto (horas/hombre)(inicial)
Fecha estimación esfuerzo requerido por el proyecto (horas/hombre)(inicial)
Estimación esfuerzo requerido por el proyecto (horas/hombre)(Actual)
Fecha estimación esfuerzo requerido por el proyecto (horas/hombre)(Actual)
Estimación tiempo requerido por el proyecto (inicial)
Fecha de estimación tiempo requerido por el proyecto (inicial)
Estimación tiempo requerido por el proyecto (actual)
Fecha de estimación tiempo requerido por el proyecto (actual)
Recursos requerido por el proyecto (inicial)
Fecha de estimación recursos requerido por el proyecto (inicial)
Recursos requeridos por el proyecto (actual)
Fecha de estimación recursos requerido por el proyecto (actual)
Estimación tiempo requerido por la iteración en curso (inicial)
Fecha de estimación tiempo requerido por la iteración en curso (inicial)
Estimación tiempo requerido por la iteración en curso (actual)
Fecha de estimación tiempo requerido por la iteración en curso (actual)

Información que registra la plantilla de documento Reunión

Tabla 4-30: Plantilla de documento Reunión

Campos particulares actividad Reunión
¿Reunión presencial o remota?
¿Qué se logró ayer? (Scrum)
¿Qué se realizará hoy? (Scrum)
¿Qué impedimentos han surgido para lograr las metas? (Scrum)
Metas de la reunión
Temas que se tocaran
Quienes deberán asistir
Quienes asistieron
Que se concluyó

Información que registra la plantilla de documento Rol

Tabla 4-31: Plantilla de documento Rol

Campos particulares documento Rol
ID Rol
Persona asignada al Rol
Metas y objetivos
Valores
Responsabilidades y deberes
Decisiones que puede tomar
Decisiones que no puede tomar
Supervisores
Subordinados
Forma de entregar feedback
Forma en que resolverá los problemas
Plan de contingencia cuando surjan los problemas
Actividades primarias
Información requerida para realizar las actividades primarias
Actividades secundarias
Información requerida para realizar las actividades secundarias
Productos de trabajo esperados
Información requerida para realizar los productos de trabajo
Plantillas productos de trabajo
Capacidades requeridas
Experiencia requerida
Conocimiento requerido
Ubicación

4.3 Automatización de la metodología mediante herramientas de apoyo

Las funcionalidades que deben poseer las herramientas de SCM para apoyar un desarrollo distribuido de software son:

- ***Repositorio de SCM***: Para almacenar aplicaciones y archivos del sistema, debe ser tolerante a las fallas, escalable, distribuible y replicable.

- ***Modelo de Check-Out/Check-In*** para:

  - Proveer copias adecuadas de los archivos.

  - No afectar la estabilidad del sistema o al trabajo de los integrantes del equipo.

  - Garantizar que no se puedan realizar ***baseline*** si es que otro miembro está trabajando en él y tiene una prioridad mayor para realizar los cambios.

  - Registrar información sobre quién hizo los cambios y cuando se realizaron.

  - Describir lo que cambió y dar una explicación del porqué.

- ***Espacios de Trabajo*** (*Workspace*): consiste en una copia de los archivos y las direcciones correctas desde el repositorio, se utilizan para ejecutar tareas específicas. Los *espacios de trabajo* evitan los siguientes problemas:

  - Errores y tiempos perdidos por inconsistencias
  - Largas integraciones en los plazos límites al final del ***ciclo de desarrollo***

  También reducen la cantidad de ***Scripting*** requerido y los errores producidos por la creación y actualización manual de los ***espacios de trabajo***.

- ***Gestión de Construcción*** (*Build Management*): minimiza el tiempo de construcción, conserva espacio de disco y ayuda a mantener y auditar el proceso de construcción. El sistema de ***SCM*** tiene que proveer funcionalidades de auditoría que permitan realizar revisiones post-construcción y garantice que se ha utilizado la configuración correcta.

- ***Gestión de las realizaciones*** (*Release Management*): Incorpora aspectos de la gestión de ***espacios de trabajo*** y ***gestión de construcción***. Es clave para crear directorios correctamente estructurados, poder extraer los archivos y las versiones correctas, y para reconstruir cualquier ***release***. Esta funcionalidad debe evitar:

  - Las dificultades para desarrollar los ***scripting*** necesarios
  - Las diferencias en los algoritmos utilizados
  - Los problemas para almacenar las estructuras de los directorios
  - Los problemas para capturar otros aspectos del ambiente de construcción, tal como la versión del compilador que fue utilizado

- ***Cambios concurrentes***: Para remover cuellos de botella, las herramientas ***SCM*** tienen que ser capaces de soportar los cambios simultáneos de dos o más desarrolladores. Además, tiene que poseer la capacidad de fusionar los cambios en los tiempos adecuados.

- ***Desarrollo paralelo***: Para minimizar o eliminar los cambios en serie porque estos aumentan el tiempo de elaboración del producto, también permite la incorporación de nuevas funcionalidades o la reparación de defectos en los ***releases*** finales. Se logra al tener la capacidad de fusionar los cambios hechos en paralelo. Una de las técnicas que lo permite es el ***branching***, donde las versiones de los artefactos se organizan mediante ramificaciones. Algunos beneficios proporcionados por el ***desarrollo paralelo*** y el ***branching*** es que facilitan las siguientes acciones:

- Desarrollo de prototipos
- Funcionalidades *a pedido del cliente*
- Utilización de diferentes plataformas y puertos
- Publicar en serie los ***releases*** principales
- Parches
- Rutas de lotes y/o paquetes de servicios
- Reparaciones de emergencia
- Aislamiento de tareas individuales
- Soporta las versiones de promoción
- Aislamiento de espacios de trabajo individuales

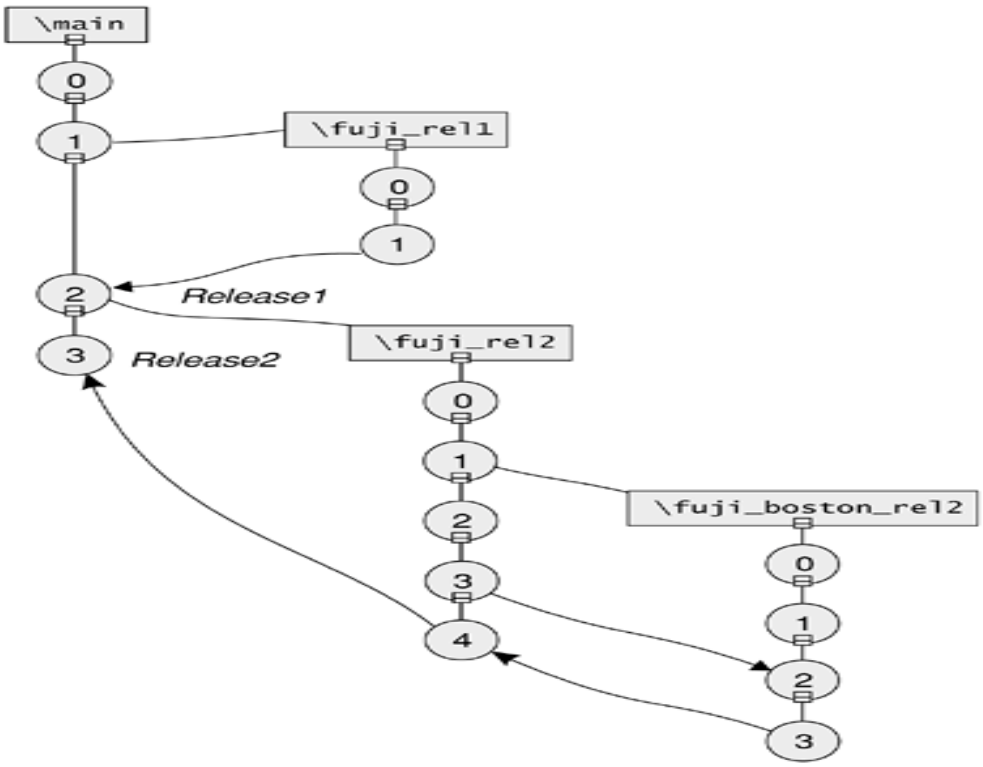

Figura 4- 9: Representación de la técnica de branching o ramificaciones. Extraído de Bellagio y Milligan (2005)

- ***Configuración de gestión basada en actividades:*** Se requiriere que la herramienta rastreen el trabajo necesario para implementar una actividad específica, disminuir la complejidad y garantizar que se ejecuten las tareas asignadas.

- ***Gestión del cambio*** (*Change Management*): Apoya la comunicación y reduce la complejidad. Las capacidades claves que deben soportar estas herramientas con respecto a este punto son:

  - ***Desarrollo distribuido de software***: Para facilitar la comunicación, coordinación y el control remoto.

  - ***Desarrollo basado en componentes***: Para identificar las versiones de los componentes y poder ensamblarlos consistentemente, lo que permite la creación de versiones estable del sistema.

  - ***Gestión de la configuración basada en actividades***: Para rastrear el trabajo que implementa una actividad específica, lo que disminuye la complejidad y garantizar que se ejecuten las tareas asignadas.

  - ***Búsquedas*** (*Queries*): Para facilitar las búsquedas mediante subconjunto de registros.

  - ***Informes*** (*Reports*): Debe proveer una forma de recolección de datos y debe proporcionar la capacidad para darles un formato que facilite la impresión, exportación y/o publicación a través de algún sitio web.

  - ***Gráficos*** (*Charts*): Para facilitar la interpretación se deben representar los datos mediantes gráficos. Por ejemplos como:

- *Gráficos de distribución*: Utilizados para categorizar datos y mostrar cómo se distribuyen los datos a lo largo de las distintas categorías.

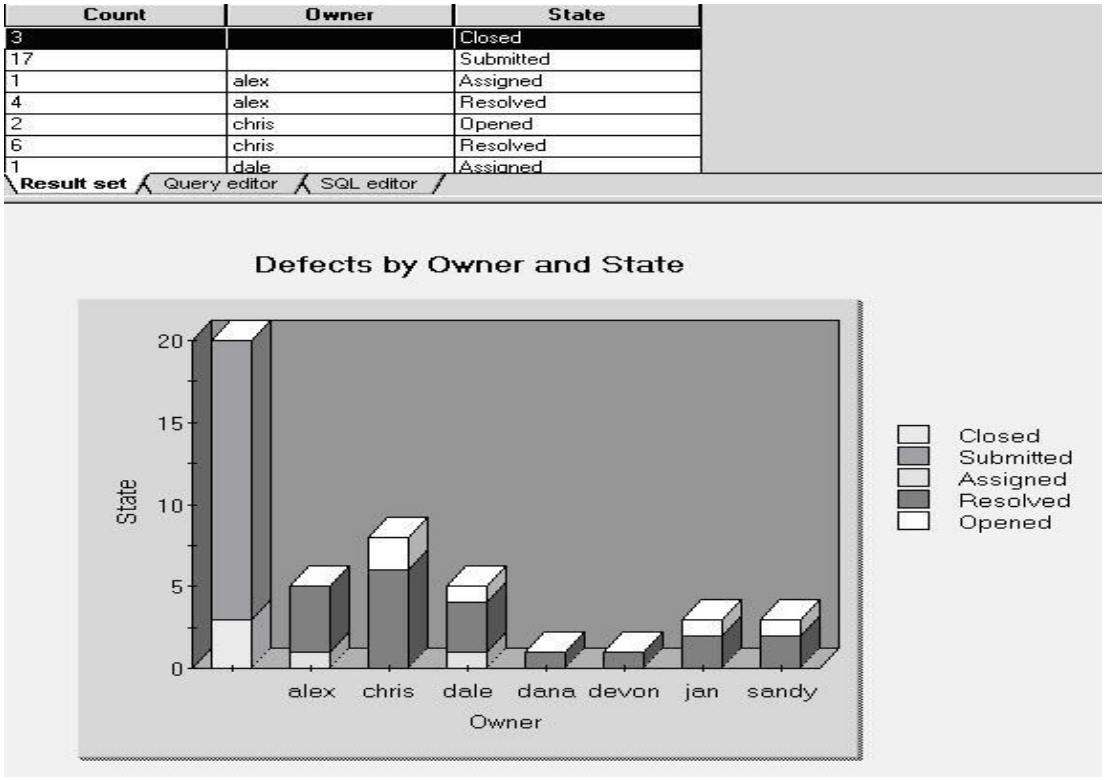

Figura 4- 10: Representación de un gráfico de distribución. Extraído de Bellagio y Milligan (2005)

- Gráficos de tendencias: Utilizados para desplegar cómo se gestiona las solicitudes de cambios a lo largo del tiempo.

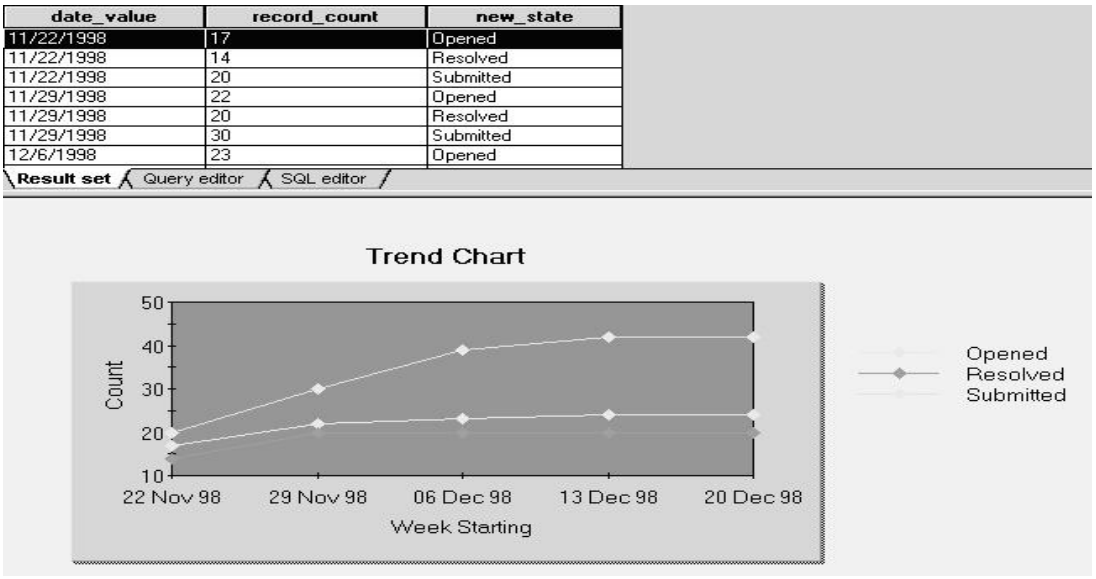

Figura 4- 11: Representación de un gráfico de tendencias. Extraído de Bellagio y Milligan (2005)

- *Gráficos de maduración*: Utilizados para mostrar cuanto tiempo permanecen las solicitudes en cada estado.

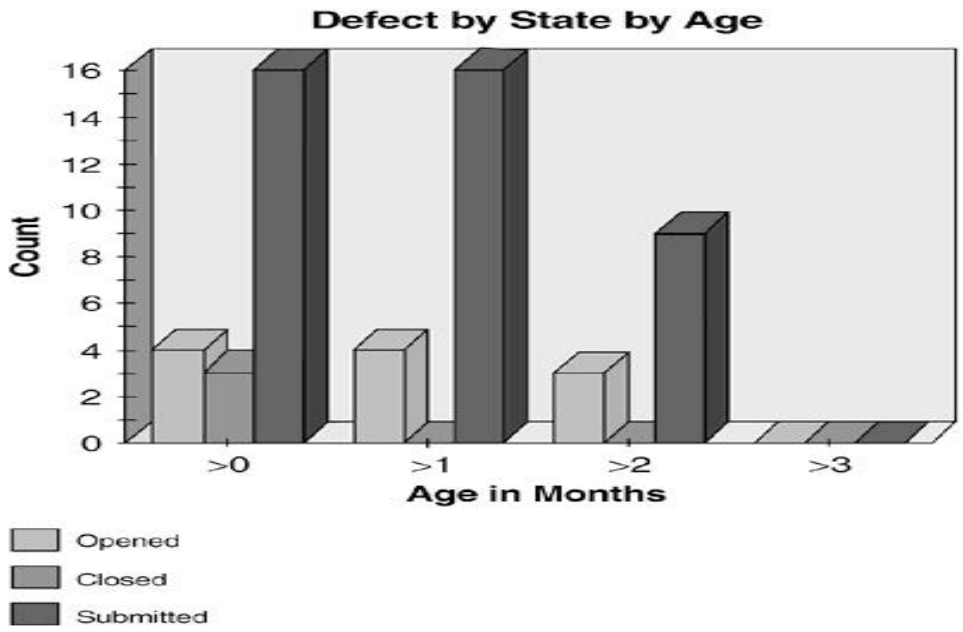

Figura 4- 12: Representación de un gráfico de maduración. Extraído de Bellagio y Milligan (2005)

4.4 Selección de una herramienta que automatice la metodología concurrente

La implementación de un modelo de elaboración concurrente, puede realizarse mediante herramientas*: desarrolladas en casa*, *propietarias,* de *acceso libre* (por ejemplo con licencia *GNL*), o las combinaciones posibles entre las tres categorías anteriores.

La mayoría de las herramientas son personalizables según las necesidades de desarrollo. Algunos ejemplos de herramientas propietarias son *Team Foundation Server* (*Microsoft*) y *ClearCase & ClearQuest* (*IBM Rational*). Mientras que algunas herramientas de *acceso libre* son *CVS* y *Subversion*.

Las principales diferencias entre dichas herramientas son sus costos y las funcionalidades que presentan, siendo las herramientas propietarias de elevado costo, pero generalmente entregando una mayor gama de funcionalidades. Pese a lo anterior, existe una quinta categoría, ya que también es posible obtener gran parte de las funcionalidades de una herramienta propietaria mediante *Software como un Servicio* (*SaaS*), un ejemplo de esto son los servicios ofrecidos por la compañía *Bright Work* (www.brightwork.com), la que permite contratar gran parte de las funcionalidades de *Team Foundation Server* mediante *servicios web.*

Utilizar herramientas *SaaS* reduce los costos, dado que se requiere una menor capacitación del personal, reduce la infraestructura TI y disminuye la cantidad de personal dedicado a mantener la infraestructura de apoyo. Además, en algunos casos, también permite la utilización de un *ambiente integrado de desarrollo* (IDE) que sea *Open Source* (por ejemplo *Eclipse*), lo que reduce los costos en licencias.

Aunque económicamente pareciera ser una mejor opción utilizar herramientas *SaaS*, es difícil generalizar, dado que el mercado está en constante evolución (las herramientas de *acceso libre* constantemente han incrementado sus funcionalidades), como también dependerá de las necesidades propias de cada proyecto.

5. VISIÓN AMPLIA DE LA PROPUESTA Y MODELACIÓN DE UN CASO DETERMINADO

5.1 Visión amplia de la propuesta (marco de trabajo)

A continuación se presenta una figura que resume la propuesta hasta este capítulo de la tesis

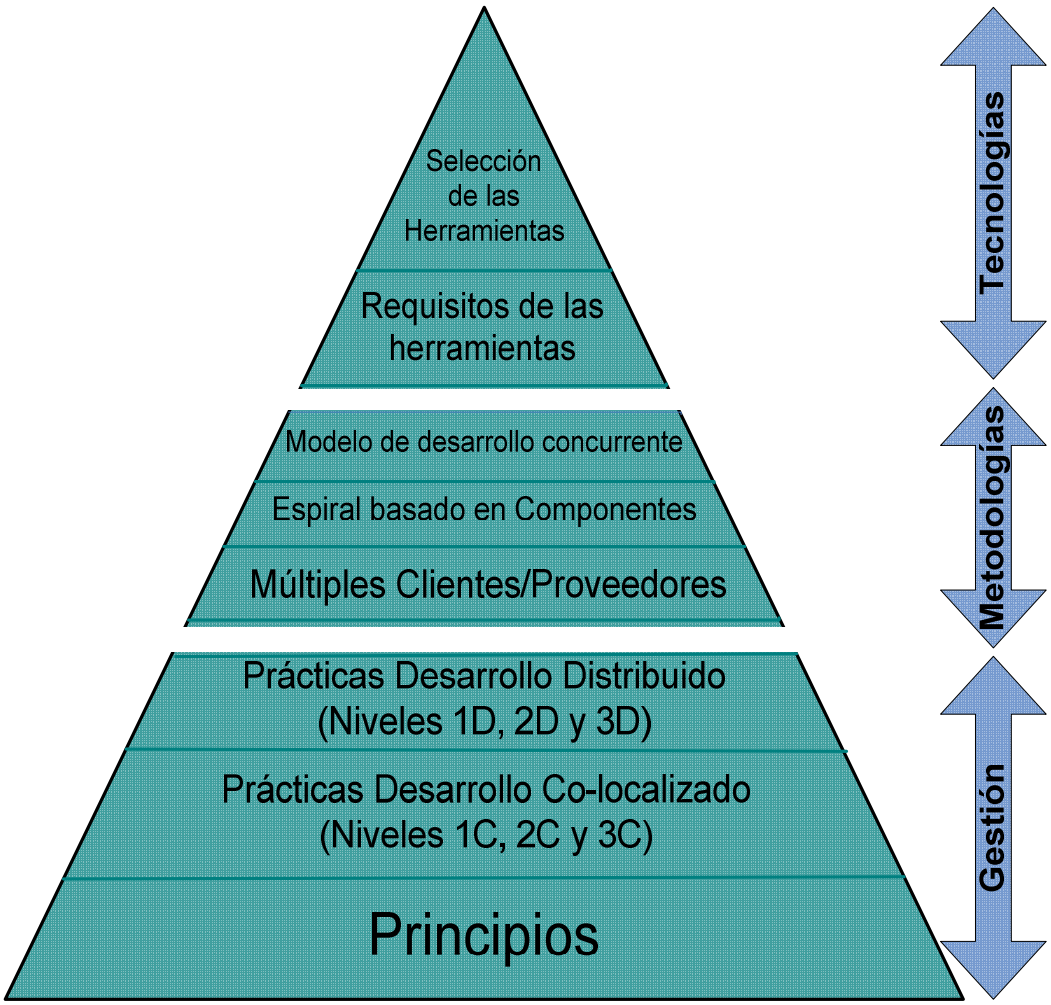

Figura 5- 1: Marco de trabajo propuesto para aumentar la probabilidad de éxito en los proyectos distribuidos. Desde lo más general a lo más particular

La siguiente figura muestra como se relacionan las metodologías propuestas.

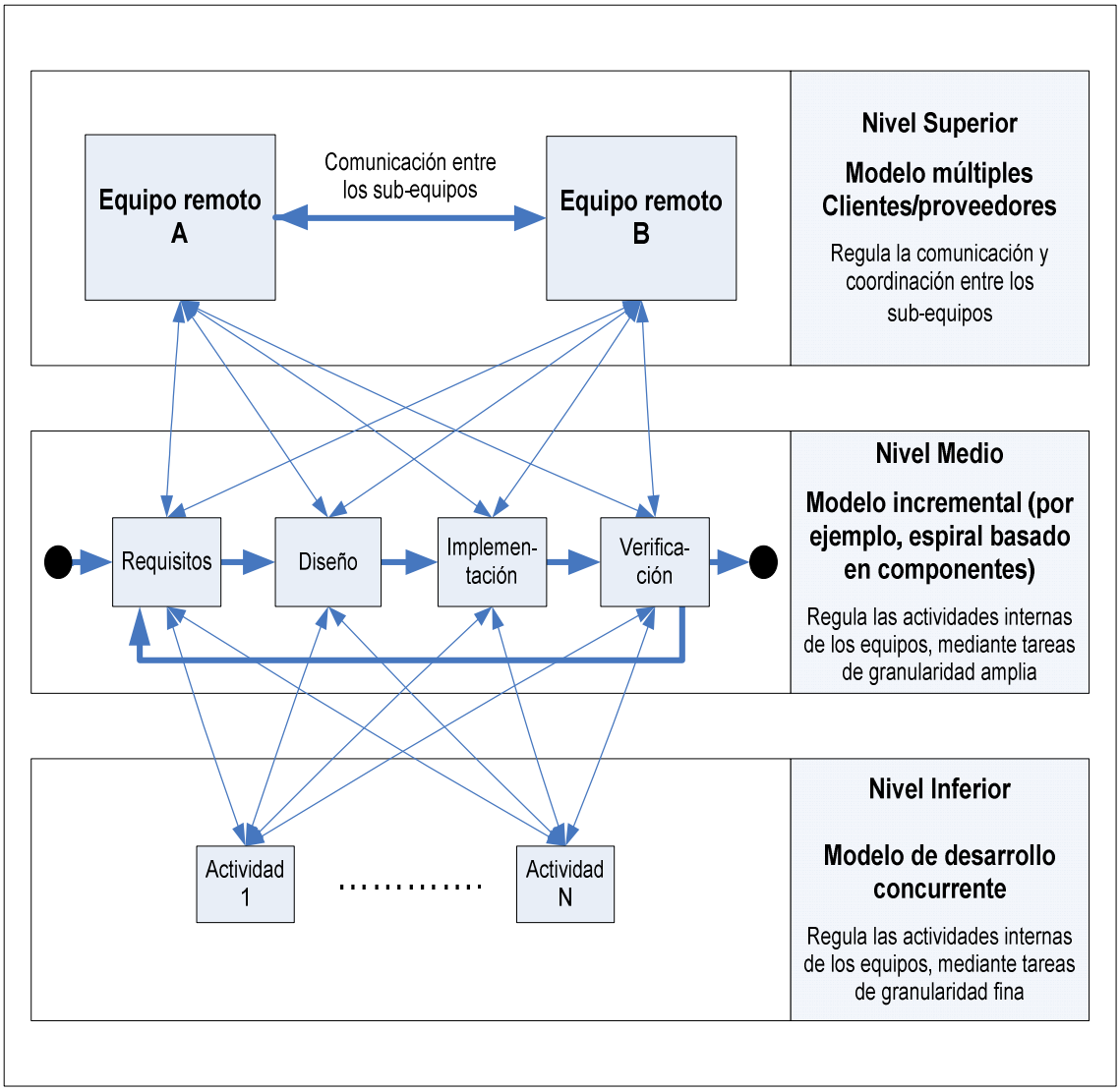

Figura 5- 2: Relación entre las metodologías propuestas

Aunque parezca que implementar las tres metodologías es una tarea demasiado compleja, la evidencia empírica muestra que la implementación de un modelo de desarrollo concurrente se hace en conjunto con la implementación de una metodología más amplia (por ejemplo modelo espiral). Un caso particular es la metodología Microsoft Solution Framework 4, dado que es una metodología que incluye tanto un modelo de desarrollo concurrente como un modelo más amplio.

5.2 Modelación de un desarrollo distribuido que utiliza el marco de trabajo propuesto

La primera variable que se consideró para realizar la modelación consistió en determinar el *grado de dispersión geográfica*, teniendo presente que esta variable permite disminuir considerablemente el riesgo de los proyecto distribuidos. También se consideró definir correctamente los roles involucrados en cada sitio.

El equipo que se propone esta compuesto por tres sub-equipos localizados en tres sitios: *Miami (EEUU)*, *Santiago (Chile)* y *Montevideo (Uruguay)*. Las localidades se han seleccionado según las siguientes características:

- ***Miami (EEUU)***: Donde se ubica el cliente.

- ***Santiago (Chile)***: Donde existen las mejores condiciones, dentro de Latinoamérica, para establecer una *plataforma de servicios* (exportación de intangibles).

- ***Montevideo (Uruguay)***: Donde existe personal altamente calificado, con experiencia, buena reputación y a costos altamente competitivos.

El equipo desarrollador se organiza según el modelo *Descentralizado Controlado* planteado en el segundo capítulo. El organigrama propuesto es el siguiente:

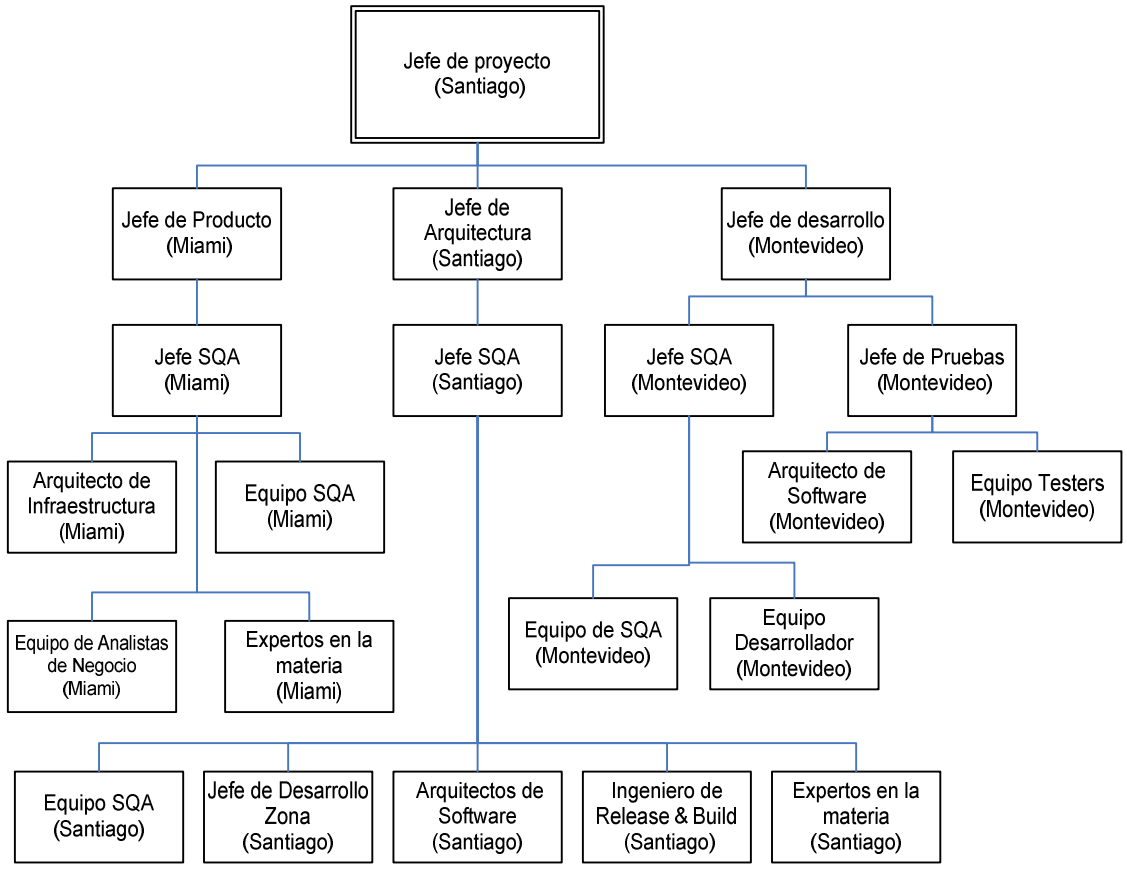

Figura 5- 3: Organigrama del equipo propuesto

Los roles involucrados y sus definiciones son:

- ***Jefe de proyecto:*** Es el responsable de crear la visión del producto, facilitar la comunicación dentro del equipo, controlar todas las fases del ciclo de vida del proyecto, planear y programar tareas (incluyendo planes de iteración, control, reducción de riesgos y elaborando informes sobre la salud del proyecto). Su objetivo principal es garantizar la adecuada elaboración del software, ajustándose al tiempo y presupuesto acordado. La estimación de los plazos y esfuerzos para realizar una tarea o actividad se debe hacer consultándole a los otros miembros del equipo.

- ***Jefe de producto***: Es el intermediario ante el *cliente*, tiene responsabilidad sobre los requisitos de la aplicación, debe garantizar la obtención de la visión del producto mediante los requisitos y las pruebas de aceptación; debe mostrar que la aplicación se ajusta al planeamiento estratégico de la *organización cliente*, debe garantizar que el proyecto se mantenga dentro del presupuesto.

- ***Jefe SQA y equipo SQA***: Actuará como representante de los usuarios y clientes, intermediando directamente con los arquitectos, desarrolladores, testers y otros roles.

- ***Analista de negocios***: Trabaja estrechamente con los usuarios, arquitectos y expertos en la materia para trabajar en la redacción de requisitos (actores, casos de uso, escenarios, historias de uso, requisitos SQA, entre otros).

- ***Arquitecto de infraestructura***: Lleva a cabo la función de especialista en topología de implementación tanto de equipos físicos como de servidores virtuales y de los servicios que se ejecutan en ellos, trabaja con herramienta de diseñado de implementación y colabora con los *arquitectos de software* para coordinar los diseños de la aplicación y del sistema en función de la configuración de implementación prevista.

- ***Jefe de arquitectura de Software y equipo de arquitectura de software***: Responsable de mantener la integridad de la arquitectura del producto y garantizar su éxito mediante el diseño de bases que permitan su materialización, incluyendo la definición de la estructura organizativa de la aplicación y la estructura física de su implementación. El objetivo del arquitecto de software es reducir la complejidad, reducir los efectos de acoplamiento y regresión e incrementar la cohesión de los componentes mediante la división del sistema en partes que puedan elaborarse y testearse independientemente. La arquitectura resultante es relevante porque no sólo dicta cómo se generará el sistema, sino que establece características fundamentales de uso, fiabilidad y mantención.

- ***Ingeniero de Release & Build***: Es un especialista cuyo trabajo consiste en integrar el código fuente y producir construcciones y *release*. Este rol debe ejecutar los *release* y desarrollar secuencias de comandos para su automatización y producir informes.

- ***Jefe de desarrollo y equipo de desarrollo***: Responsables principalmente de codificar el producto. También pueden ayudar a especificar los requisitos y trabajar en actividades de arquitectura, dependiendo de la fase en que se encuentre el proyecto.

- ***Jefe de pruebas y equipo de pruebas***: El objetivo principal del jefe de pruebas es descubrir y comunicar los problemas que influyen de forma negativa en el producto, comprendiendo el contexto del proyecto y ayudando a los demás a tomar decisiones en función de ese contexto. Las funciones del equipo de pruebas son describir los errores e detallar los pasos necesarios para que sea más fácil entenderlos y recrearlos. Este equipo debe participar en la definición de los niveles de calidad del producto. El objetivo de las pruebas es comprobar que las funciones conocidas se desempeñen correctamente y descubrir nuevos problemas que se presenten en el producto.

Para aumentar la cohesión del equipo, se propone incrementar la confianza promoviendo las reuniones presenciales antes y durante la realización del proyecto.

Para utilizar una adecuada cantidad y rigurosidad de procesos, durante el ciclo de vida del desarrollo, se propone la utilización de *herramientas mentoras* como ***RUP*** u ***OpenUP*** o *guías de procesos* como ***MSF Ágil*** o ***MSF CMMI***. Lo anterior pensando en que éstas dirigen los *flujos de trabajo* de las tareas y se basan en una amplia experiencia adquirida en una vasta cantidad de proyectos de software exitosos.

Para seleccionar al personal que trabajará en el equipo desarrollador se recomienda utilizar personas que tengan cierto nivel de experiencia comprobada, ya sea porque está certificada por alguna institución de prestigio o porque han trabajado en proyectos de similares características. En consecuencia, se recomienda tener un departamento de RRHH bien definido y constituido, en el cual su principal misión será reclutar y mantener a personas idóneas en los cargos y roles del proyecto.

Con respecto a la figura tributaria y legal del equipo desarrollador, se recomienda la constitución de empresas en cada localidad, dado que esta opción permite que se descuenten impuestos (como el IVA para el caso de Chile), le entrega estabilidad laboral al personal y permite que el flujo de activos intangibles esté regulado por un marco legal. Sin embargo, la posibilidad de no constituir tantas empresas también es posible a raíz de las nuevas herramientas de TI que permiten el trabajo remoto (por ejemplo se pueden hacer transacciones internacionales para pagar las remuneraciones gracias a nuevos medios de pago on-line como *paypal*). Otra ventaja que tiene esta opción es que en algunos países (por ejemplo Chile) aceptan la utilización de facturas y boletas extranjeras para descontar impuestos sobre los resultados operacionales nacionales. Por ende, aunque la opción de no conformar tantas empresas parece ser una alternativa más económica, es una decisión que debe estudiarse en cada caso ya que, generalmente, para que no se produzca rotación de personal los empleados de cualquier tipo de empresa prefieren tener una situación laboral estable que esté regulada bajo leyes que establezcan sus derechos y deberes.

A continuación se presentan modelaciones para los *flujos de trabajo* de las *actividades concurrentes principales*, considerando la estructura organizacional propuesta para el proyecto modelado:

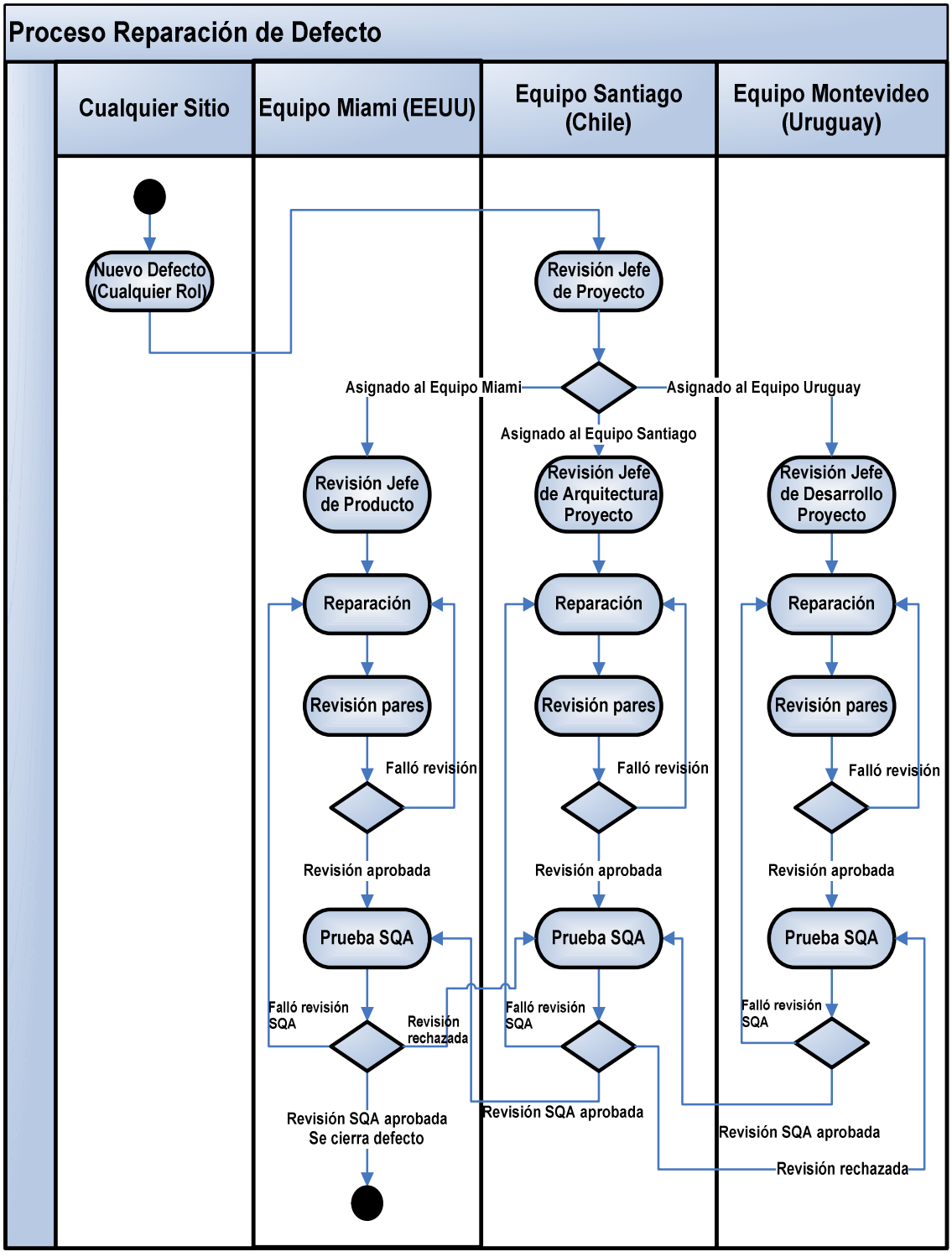

Figura 5- 4: Flujo de trabajo para la actividad Defecto, considerando la estructura organizacional del proyecto

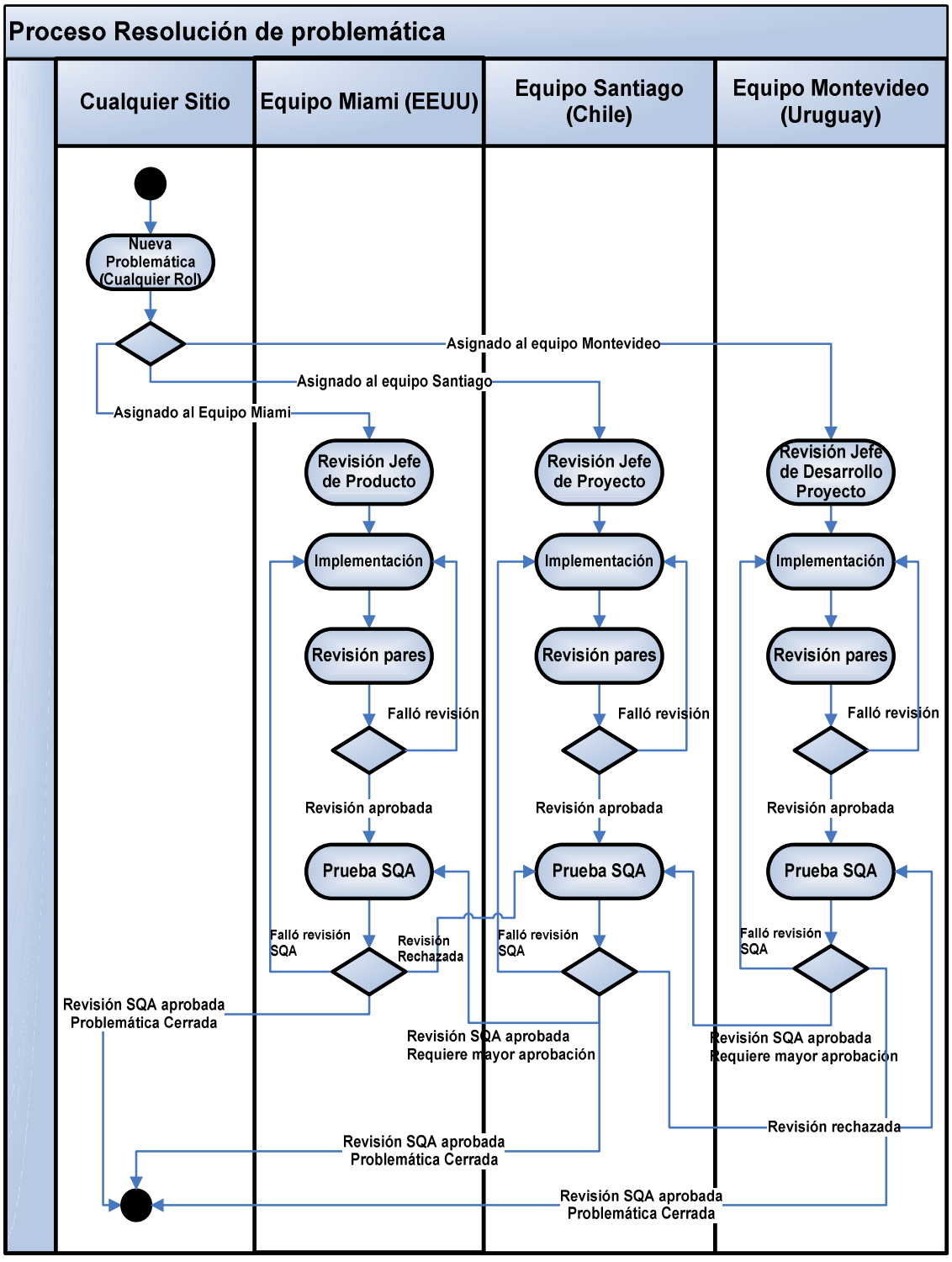

Figura 5- 5: Flujo de trabajo para la actividad Problemática, considerando la estructura organizacional del proyecto

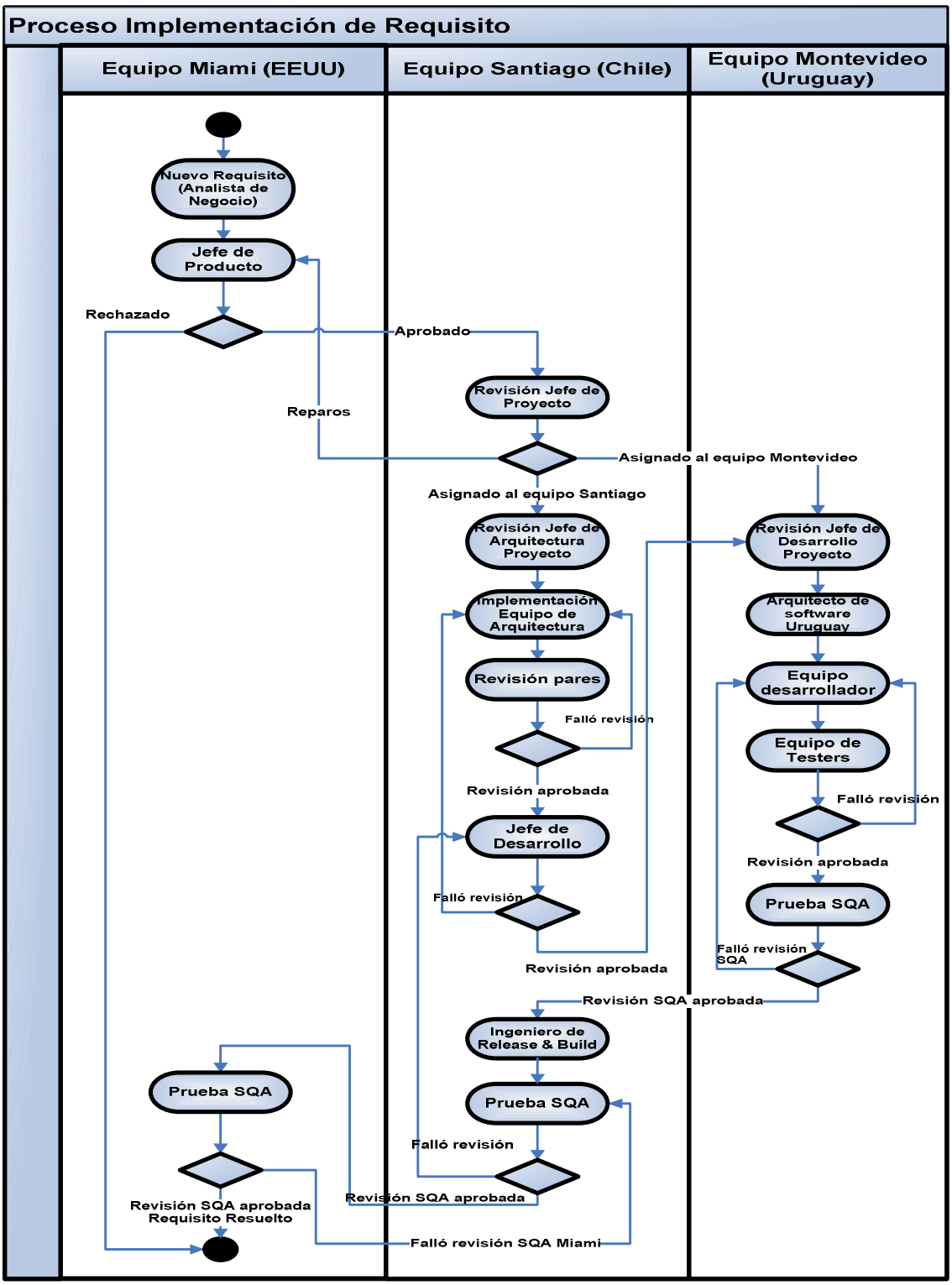

Figura 5- 6: Flujo de trabajo para la actividad Requisito, considerando la estructura organizacional del proyecto

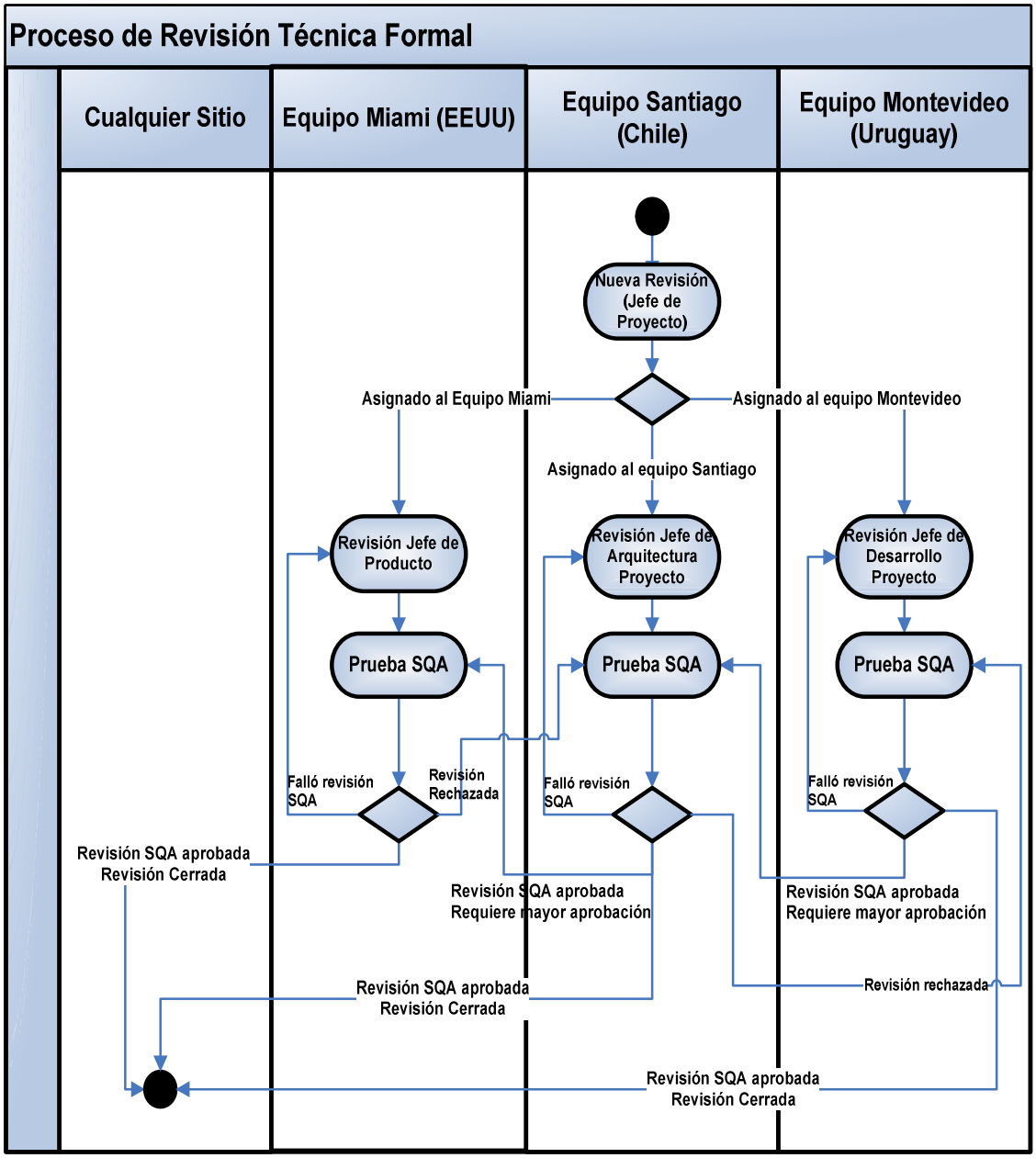

Figura 5- 7: Flujo de trabajo para la actividad Revisión Técnica Formal, considerando la estructura organizacional del proyecto

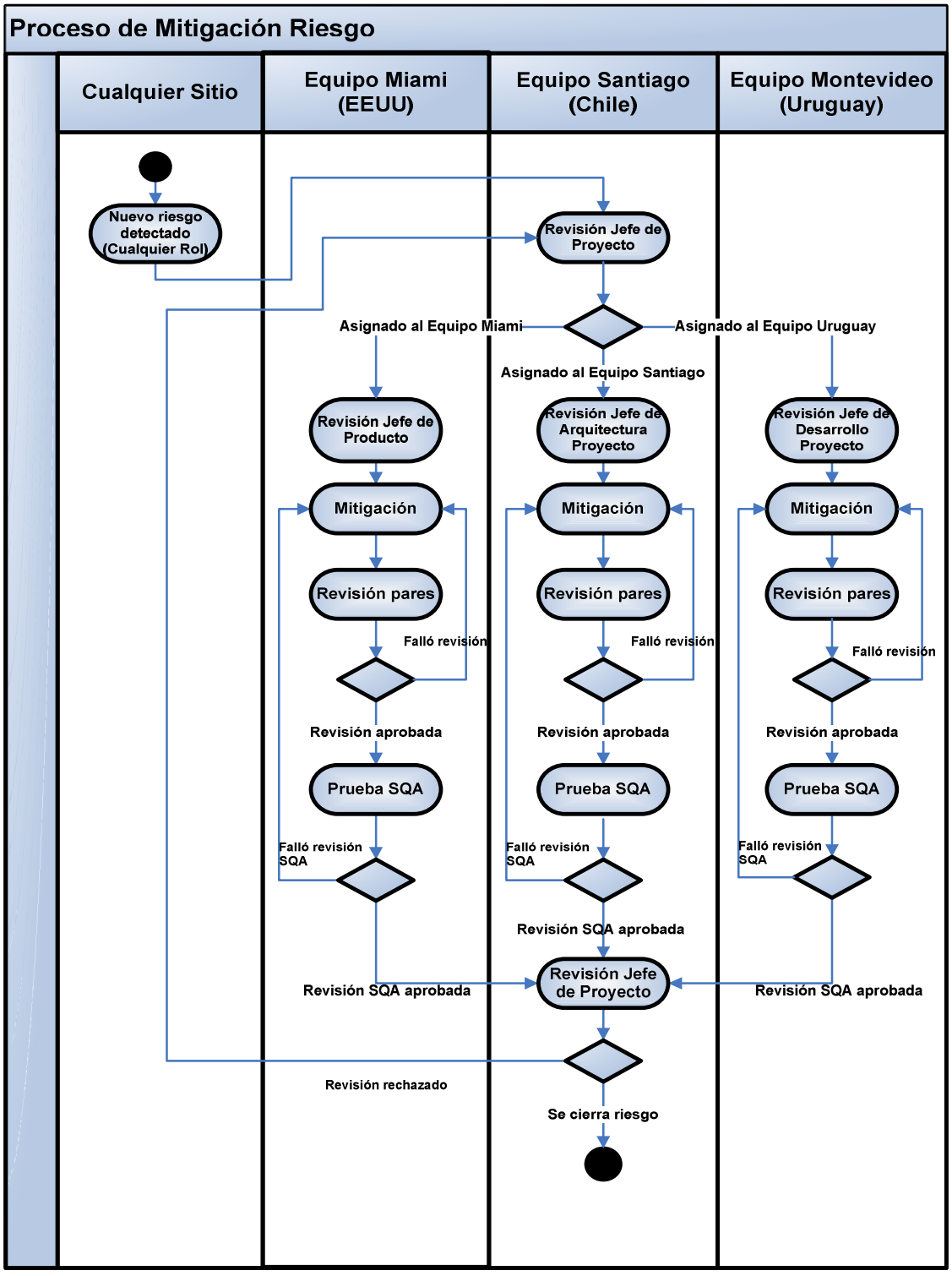

Figura 5- 8: Flujo de trabajo para la actividad Riesgo, considerando la estructura organizacional del proyecto

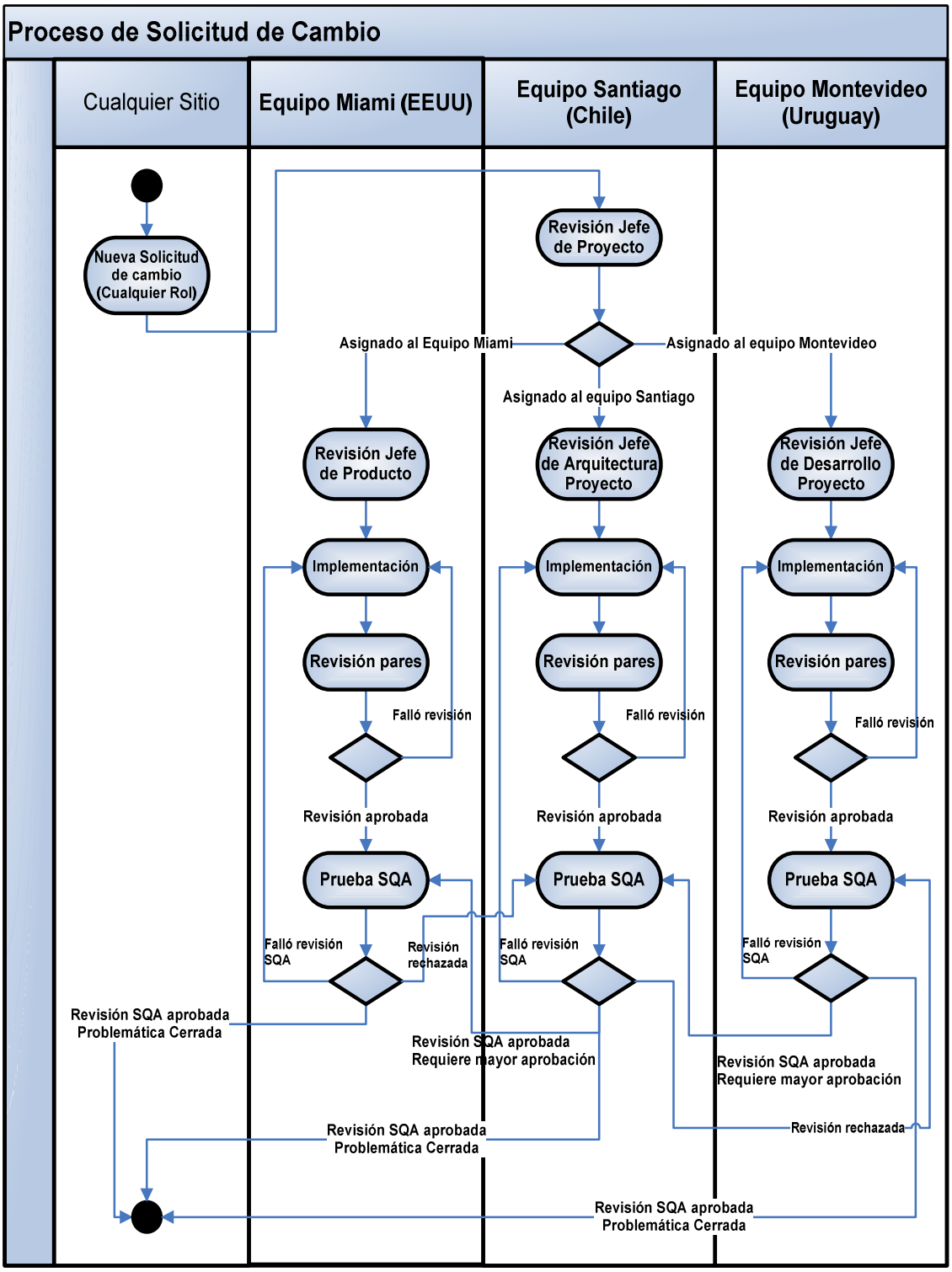

Figura 5- 9: Flujo de trabajo para la actividad Solicitud de Cambio, considerando la estructura organizacional del proyecto

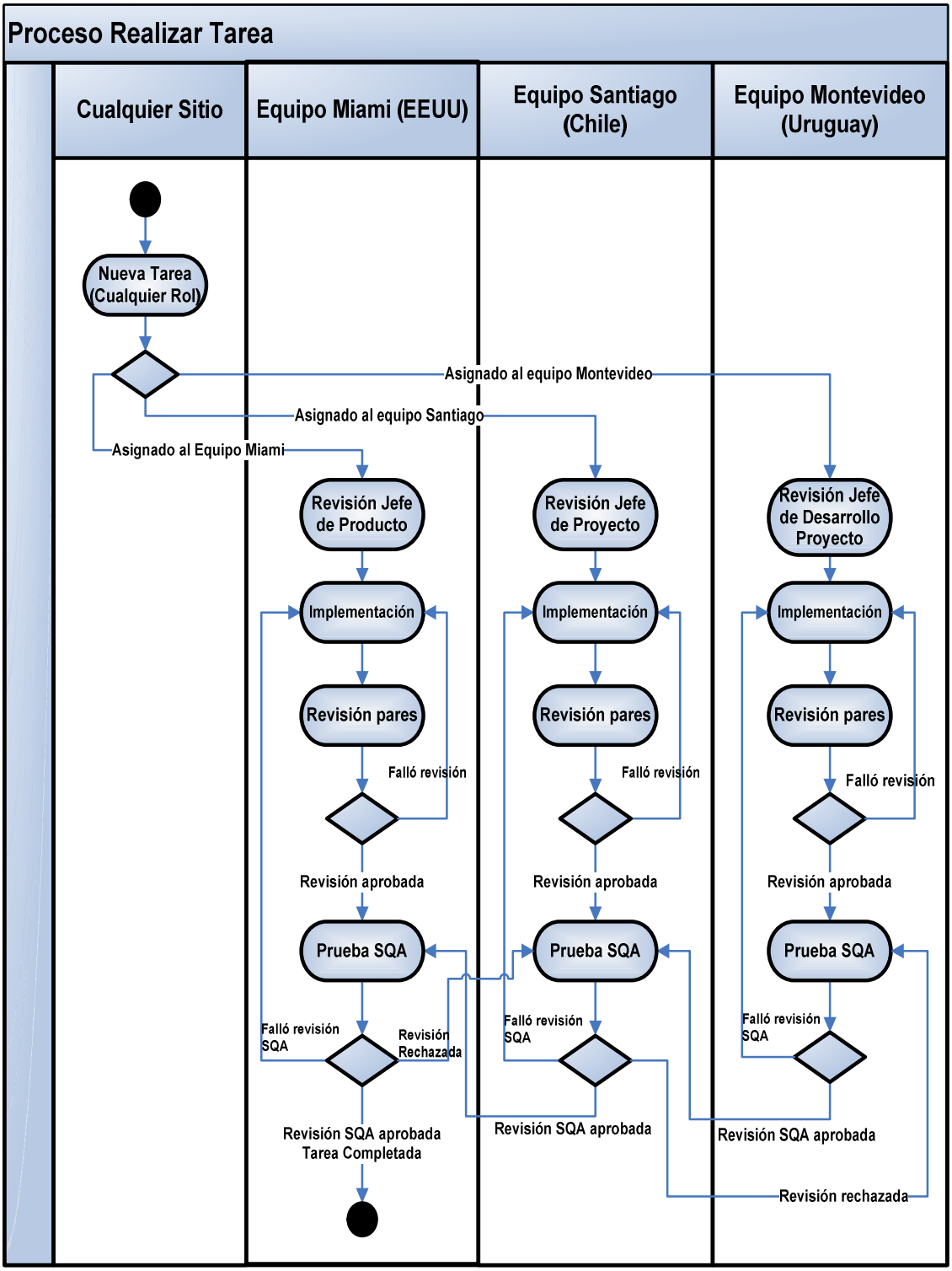

Figura 5- 10: Flujo de trabajo para la actividad Tarea, considerando la estructura organizacional del proyecto

6. CONCLUSIONES

A lo largo de toda la investigación se ha mostrado indirectamente que el *desarrollo distribuido de software* puede ser exitoso si se toman en cuenta todas las condiciones críticas necesarias para poder efectuarlo. Dichas medidas deben considerar las prácticas, principios, metodologías, proceso, herramientas correctas y criterios para adoptar paulatinamente las prácticas.

Por otro lado, a lo largo de toda la investigación se han hecho diversas propuestas que, adecuadamente aplicadas, deberían facultar a una empresa mediana a emprender exitosamente este nuevo modelo de negocio. Sin embargo, hay que considerar que cada proyecto y empresa tiene sus propios requisitos y dificultades, por ende, al igual que la elaboración de componentes lógicos, desplegar una empresa que desarrolle software es una labor que involucra una gran cantidad de tareas interrelacionadas con alto grado de incertidumbre, lo que impide la aplicación de una *receta única* o un conjunto de tareas bien definidas y estructuradas. Es así como, a lo largo de esta investigación se ha demostrado que para aplicar un desarrollo de software exitoso, ya sea distribuido o tradicional, se necesita conducir su elaboración basándose en directrices bien definidas, una analogía ampliamente utilizada en la literatura menciona que la elaboración de software se asemeja más a guiar una embarcación en un mar tormentoso que a la construcción de un componente físico como es el caso de un puente. Lo anterior se debe a diversas causas, como por ejemplo al gravitante peso que tiene el factor humano, la incertidumbre, el cambio y los costos en ingeniería (en comparación con la elaboración estándar de componentes físicos).

A pesar de que la *ingeniería de software* es una nueva e insipiente rama de las ingenierías, orientada a los componentes lógicos y no físicos, y a su diversidad de corrientes que intentan dar solución a ciertas problemáticas, la presente tesis ha dado una respuesta afirmativa a la hipótesis planteada:

Es posible tratar los principales problemas atribuibles al desarrollo distribuido de software aplicando una metodología de elaboración concurrente, adaptada especialmente para este fin y que considere el diseño de nuevas actividades, incluyendo sus posibles estados, flujos y causas de creación y transición.

Como también se han logrado los tres objetivos específicos planteados al inicio de la investigación:

a) Plantear un conjunto de ***prácticas y principios para facilitar la gestión*** de los *proyectos distribuidos de software*.

b) Proponer y diseñar una serie de ***metodologías para facilitar la ejecución*** de estos *proyectos*.

c) Señalar las ***herramientas que puedan sustentar las prácticas y las metodologías propuestas***.

7. TRABAJO FUTURO

Se propone como trabajo futuro probar empíricamente, a través de un *caso de prueba*, parte de las prácticas, principios y metodologías propuestas en esta investigación, según los niveles propuestos de adopción.

Una posible línea investigativa que continúe el tema abordado en este estudio debe poner especial énfasis en implementar, total o parcialmente, los modelos de elaboración planteados (por ejemplo: *modelo espiral basado en componentes*, *modelo concurrente* y *modelo Múltiples Clientes/Proveedores*), como también implementar las prácticas propuestas y , si es posible, utilizar una herramienta que pueda apoyar al equipo distribuido.

Dicho *caso de prueba* se puede abordar de la misma forma como se planteó la modelación realizada para exponer un desarrollo distribuido en tres localidades (Chile, EEUU y Uruguay), considerando que se pueden relajar las restricciones geográficas, disminuyendo así la distancia entre los sitios remotos, es decir, en vez de utilizar localidades distribuidas en entre países, se puede utilizar un equipo que se distribuya dentro de un mismo país, región o ciudad, sin embargo, al desarrollar esta experiencia se debe considerar la principal restricción presente en los desarrollos distribuidos, la que consiste en tener pocas o nulas posibilidades de reuniones presencialmente entre parte de los integrantes del equipo.

8. BIBLIOGRAFIA

Alarcón, R.A. (2004). *Awareness Semántico en Apoyo de Grupos de Trabajo Colaborativo Virtuales*. Disertación doctoral no publicada, Pontificia Universidad Católica de Chile, Santiago, Chile.

Ambler, S. (2002). *Bridging the Distance: Dispersed and distributed development teams may span the globe, but they can be linked through the agile movements´s main principle: communication*. Recuperado el 18 de junio de 2007, del Sitio web Dr. Dobb´s Journal: http://www.ddj.com

Ambler, S. y Jefries, R. (2002). *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. Hoboken, NJ, EE.UU.: John Wiley & Sons.

Azher, J. (2007). *Global Outsourcing with Microsoft Visual Studio 2005 Team System.* Boston, MA, EE.UU.: Thomson Learning Inc.

Bach, J. (1998). The Highs and Lows of Change Control. *Computer*, 31, 8, 113-115.

Bauer, F. (1972). *Software Engineering*. Amsterdam, Holland.: North Holland Publishing.

Beck, K. y Andres, C. (2004). *Extreme Programming Explained: Embrace Change, Second Edition*. Boston, EE.UU.: Addison-Wesley.

Bellagio, D. E. & Milligan, T. J. (2005) *Software Configuration Management Strategies and IBM Rational ClearCase: A Practical Introduction*. **(**2a. ed.). Upper Saddle River, NY, EE.UU.: IBM Press.

Berenbach, B. (2006, mayo). Impact of Organizational Structure on Distributed Requirements Engineering Processes: Lessons Learned. Documento presentado en, First International Workshop on Global Software Development for the Practitioner, Shangai, China.

Boehm, B. (1981). *Software Engineering Economics*. Upper Saddle River, NJ, EE.UU: Prentice-Hall.

Carmel, E. (2003). Taxonomy of New Software Exporting Nations. *The electronic Journal on Information Systems in Developing Countries,* 13, 2, 1-6.

Carmel, E. & Agarwal, R. (2002). The Maturation of Offshore Sourcing of Information Technology Work. *MIS Quarterly Executive*. 1, 2, 1-19.

Carmel, E. (1999). *Global Software Teams: Collaborating Across Borders and Time Zones*. Upper Saddle River, NJ, EE.UU.: Prentice-Hall PTR.

Crowston, K. y Kammerer, E. (1998). Coordination and collective mind in software requirements development. *IBM Systems Journal.* 37**,** 2, 227-245.

Constantine, L. (1993). Work Organization: Paradigms for Project Management and Organization. *CACM*, 36, 10, 34-43.

Costa, Y. P. (2005). *Desenvolvimento Distribuído de Software: Em busca de uma metodologia*. Recuperado el 29 de junio de 2007, del sitio web del Departamento de Sistemas y Computación de la Universidad Federal de Campina Grande de Brasil: http://www.dsc.ufcg.edu.br/~garcia/cursos/dglobal_software/Textos_Sem/

Damian, D. E., y Zowghi, D. (2003, enero). An insight into the interplay between culture, conflict and distance in globally distributed requirements negotiations. Ponencia presentada en la *36th Annual Hawaii International Conference on System Sciences*, Hawaii, EEUU

Damian, D. (2002, mayo). The Study of Requirements Engineering in Global Software Development: as Challenging as Important. Ponencia presentada en el *Workshop on Global Software Development, part of the International Conference on Software Engineering (ICSE)*, Orlando, Florida, USA.

DeMarco, T. y Lister, T. (2003). Risk Management during Requirements. *IEEE Software*. 20,5, 99-101

FriedlNet and Partners (2007), *Development of Global Software Industry Annual Report 2006-2007*. Recuperado de http://www.friedlnet.com/product_info.php?language=es&products_id=5061&osCsid

Glass, R. (1998). Defining Quality intuitively. *IEEE Software*.15, 3, 103-104 y 107.

Grady, R. (1997). *Successful Software Process Improvement*. Englewood Cliffs, NJ, EE:UU: Prentice Hall.

Greenfield, J. et al. (2004). *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Hoboken, NJ. EE.UU.: John Wiley & Sons.

Gottesdiener, E. (2003). Team Retrospectives for Better Iterative Assessment. Recuperado el 24, de Julio de 2007, de http://www.128.ibm.com/developerworks/rational/library/content/RationalEdge/apr03/TeamRetrospectives_TheRationalEdge_Apr2003.pdf .

Guckenheimer, S. y Perez, J. (2006). *Software Engineering with Microsoft Visual Team System*. Boston, MA, EE.UU.: Pearson Education Inc.

Higuera, R.., Dorofee, A.., Walker, J. y Williams, R. (1994). *Team Risk Management: A new Model for Customer-Supplier Relationships*. Pittsburgh, PA, EE.UU.: Software Engineering Institute.

Hundhausen, R. (2006). *Working with Microsoft Visual Studio 2005 Team System*. Redmond, WA, EE.UU.: Microsoft Press.

IBM Corporation. (2005). Rational Unified Process. (7.0.1.E). [Programa de Computador]. Westchester, NY, EE.UU.: Autor.

Jackman, M. (1998). Homeopathic Remedies for Team Toxicity. *IEEE Software*, 15, 4, 43-45.

Jacobson, I., Griss, M. y Jonsson, P. (1997). *Software Reuse: Architecture, Process, and Organization for Business Success*. Reading, MA, EE.UU.: Addison-Wesley.
Karolak, D. (1999). *Global Software Development: Managing Virtual Teams and Environments*. Piscataway, NJ, EE.UU.: IEEE Computer Society.

Kotlarsky, J. (2005). *Management of Globally Distributed Component-Based Software Development Projects.* Disertación doctoral no publicada, Erasmus University, Rótterdam, Holanda.

Kraul, R. y Streeter, L. (1995). Coordination in Software Development. *CACM*, 38, 3, 69-81.

Kroll, P. y Maclsaac, B. (2006). *Agility and Discipline Made Easy: Practices from OpenUP and RUP*. Boston, MA, EE.UU.: Pearson Education Inc.

Larman, C. (2004). *Agile and iterative development: a manager's guide*. Bosto, MA, EE.UU.: Addison-Wesley.

Larraguibel, G. (2007, abril). *Chile y la Nueva Industria de Offshoring ¿Cómo aprovechamos la oportunidad?*. Ponencia presentada en seminario Chile y la Nueva Industria de Offshoring ¿Cómo aprovechamos la oportunidad?, Santiago, Chile.

Levinson, J. y Nelson, D. (2006). *Pro Visual Studio 2005 Team System.* New Cork City, NY, EE.UU.: Board.

Massachussets Institute of Technology. (2005). *Perspectives on Free and Open Source Software*. Cambridge, England: The MIT Press.

Mantei, M. (1981). The Effect of Programming Team Development. *CACM*, 24, 3, 106-13.

Microsoft Corporation. (2006). MSF for Agile Software Development. (4.1.0). [Programa de Computador]. Redmond, WA, EE.UU.: Autor.

Microsoft Corporation. (2005). MSF for Agile Software Development. (4). [Programa de Computador]. Redmond, WA, EE.UU.: Autor.

Microsoft Corporation. (2006). MSF for CMMI Process Improvement. (4.1.0). [Programa de Computador]. Redmond, WA, EE.UU.: Autor.

Microsoft Corporation. (2005). MSF for CMMI Process Improvement. (4). [Programa de Computador]. Redmond, WA, EE.UU.: Autor.

Park, R., Goethert, W. y Florac, W. (1996). *Goal Driven Software Measurement A Guidebook*. Pittsburgh, PA, EE.UU.: Software Engineering Institute.

Pressman, R. (2002). *Ingeniería del Software. Un enfoque práctico* (5a. ed.). Madrid, España: The McGraw-Hill Companies.

Pressman, R. (1999). *Adaptable Precess Model*. Boca Raton, FL, EE.UU: Pressman & Associates.

Procópio, F. (2005). *Desenvolvimento Global de Software Distribuído*. Recuperado el 29 de junio de 2007, del sitio web del Departamento de Sistemas y Computación de la Universidad Federal de Campina Grande de Brasil: http://www.dsc.ufcg.edu.br/~garcia/cursos/dglobal_software/Textos_Sem/

Putman, L. y Myers, W. (1992). *Measures for Excellence*. Upper Saddle River, NJ, EE.UU: Prentice Hall PTR.

Reel, J. (1999). Critical Success Factors in Software Projects. *IEEE Software,* 16, 3, 18-23.

Ragland, B. (1995). Measure, Metric, or Indicator: What's the Difference?. *Crosstalk,* 8, 3, 29-30.

Sarker, S. y Sahay, S. (2004). *Implications of space and time for distributed work: an interpretive study of US–Norwegian systems development teams.* 13, 2-20.

Stojanovic, Z. y Dahanayake, A. (2005). *Service-Oriented Software System Engineering: Challenges and Practices*. Hershey, PA, EE.UU.: Idea Group Publishing.

Sutherland, J., Viktorov, A., Blount, J., y Puntikov, N. (2007, enero). Distributed Scrum: Agile Project Management with Outsourced Development Teams. Ponencia presentada en la *40th Annual Hawaii International Conference on System Sciences,* Hawaii, EEUU.

The Standish Group International (2003). *Chaos Chronicle v3.0*. Recuperado el 18 de Julio de 2007 de http://www.standishgroup.com/press/article.php?id=2.

Tschang, T. (2001). *The Basic Characteristics of Skills and Organizational Capabilities in the Indian Software Industry.* Tokio, Japón: Asian Development Bank Institute.

Wikipedia. (2007). Software engineering. Recuperado el 30 de Junio de 2007, de http://en.wikipedia.org/wiki/Software_engineering.