Cátedra de Proceso Software

**APUNTE DE MODELOS DE PROCESO**

Prof. Gladys Kaplan

Equipo docente:

Gabriel Blanco

Gladys Kaplan

Guillermo Hinidi

Rosa Sanabria

María Inés Bertin

Contenido

[MODELO CASCADA 4](#_Toc415829613)

[MODELO PRESCRIPTIVO 4](#_Toc415829614)

[MODELO EN ESPIRAL 7](#_Toc415829615)

[MODELO PRESCRIPTIVO 7](#_Toc415829616)

[TIPOS 10](#_Toc415829617)

[Estrategia de PROTOTIPADO 11](#_Toc415829618)

[MODELO PRESCRIPTIVO 11](#_Toc415829619)

[MODELO ESPIRAL CON REUTILIZACION 14](#_Toc415829620)

[MODELO ESPIRAL CON CASCADA Y PROTOTIPOS 14](#_Toc415829621)

[MODELO DE TRANSFORMACIÓN 15](#_Toc415829622)

[MODELO PRESCRIPTIVO 15](#_Toc415829623)

[Ventajas: 15](#_Toc415829624)

[Desventajas: 16](#_Toc415829625)

[ENFOQUE INCREMENTAL 16](#_Toc415829626)

[MODELO PRESCRIPTIVO 16](#_Toc415829627)

[BASADO EN REUTILIZACION 18](#_Toc415829628)

[THE PRISM – Modelo de Cambios 20](#_Toc415829629)

[MODELO DESCRIPTIVO 20](#_Toc415829630)

[ORIENTADO A ASPECTOS 21](#_Toc415829631)

[DESARROLLO ÁGIL 24](#_Toc415829632)

[PROCESO UNIFICADO 33](#_Toc415829633)

[MODELO PRESCRIPTIVO 33](#_Toc415829634)

[CONCLUSIONES 35](#_Toc415829635)

[Referencia Bibliográfica 37](#_Toc415829636)

Por su naturaleza, los modelos son simplificaciones; por lo tanto, un modelo de procesos de software es una simplificación o abstracción de un proceso real. Podemos definir un modelo de proceso del software como una representación abstracta de alto nivel de un proceso software. Cada modelo es una descripción de un proceso software que se presenta desde una perspectiva particular. Alternativamente, a veces se usan los términos ciclo de vida y Modelo de ciclo de vida. Cada modelo describe una sucesión de fases y un encadenamiento entre ellas. Según las fases y el modo en que se produzca este encadenamiento, tenemos diferentes modelos de proceso. Un modelo es más adecuado que otro para desarrollar un proyecto dependiendo de un conjunto de características de éste.

Los modelos de proceso de software se clasifican en **descriptivos** y **prescriptivo**. Los primeros suelen ser informativos o analíticos y en algunos casos se realizan una vez que el proceso fue realizado. Los modelos de proceso prescriptivos son aquellos que definen las actividades, acciones, tareas, hitos y artefactos para desarrollar un producto software de calidad y estas actividades pueden ser lineales, incrementales, evolutivas.

Un proceso del software es un conjunto de actividades que conducen a la creación de un producto software. Estas actividades pueden consistir en el desarrollo de software desde cero en un lenguaje de programación estándar como Java o C. Sin embargo, cada vez más, se desarrolla nuevo software ampliando y modificando los sistemas existentes y configurando e integrando software comercial o componentes del sistema.

Los procesos del software son complejos y, como todos los procesos intelectuales y creativos, dependen de las personas que toman decisiones y juicios. Debido a la necesidad de juzgar y crear, los intentos para automatizar estos procesos han tenido un éxito limitado. Las herramientas de ingeniería del software asistida por computadora (CASE) pueden ayudar a algunas actividades del proceso. Sin embargo, no existe posibilidad alguna, al menos en los próximos años, de una automatización mayor en el diseño creativo del software realizado por los ingenieros relacionados con el proceso del software.

Una razón por la cual la eficacia de las herramientas CASE está limitada se halla en la inmensa diversidad de procesos del software. No existe un proceso ideal, y muchas organizaciones han desarrollado su propio enfoque para el desarrollo del software. Los procesos han evolucionado para explotar las capacidades de las personas de una organización, así como las características específicas de los sistemas que se están desarrollando. Para algunos sistemas, como los sistemas críticos, se requiere un proceso de desarrollo muy estructurado. Para sistemas de negocio, con requisitos rápidamente cambiantes, un proceso flexible y ágil probablemente sea más efectivo.

Aunque no existe un proceso del software «ideal», en las organizaciones existen enfoques para mejorarlos. Los procesos pueden incluir técnicas anticuadas o no aprovecharse de las mejores prácticas en la ingeniería del software industrial. De hecho, muchas organizaciones aún no aprovechan los métodos de la ingeniería del software en el desarrollo de su software.

Los procesos del software se pueden mejorar por la estandarización del proceso donde la diversidad de los procesos del software en una organización sea reducida. Esto conduce a mejorar la comunicación y a una reducción del tiempo de formación, y hace la ayuda al proceso automatizado más económica. La estandarización también es un primer paso importante para introducir nuevos métodos, técnicas y buenas prácticas de ingeniería del software.

El presente apunte describe los principales modelos de proceso. Para una mejor comprensión se presenta una cronología de estos modelos y luego se describen cada uno.

BREVE CRONOLOGÏA DE LOS MODELOS DE PROCESO:

1970 Royce – El modelo Cascada

1980 Mills – Estrategia Incremental

1982 McCraken – Estrategia Iterativa

1983 Blazer – Modelo de Transformación

1984 Floyd – Estrategia con Prototipos

1988 Boehm – Modelo en Espiral

1988 Jones – Modelo basado en la Reutilización

1991 Madhavji – Modelo de cambios Prism

1996 Gregor Kiczales - Orientación a Aspectos

1999 Ivar Jacobson, Grady Booch y James Rumbaugh - RUP

2001 – Desarrollo Ágil

## MODELO CASCADA

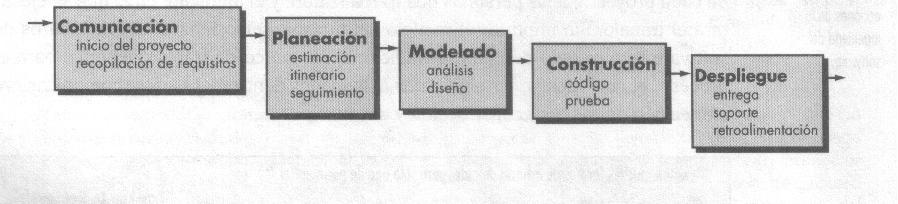
### MODELO PRESCRIPTIVO

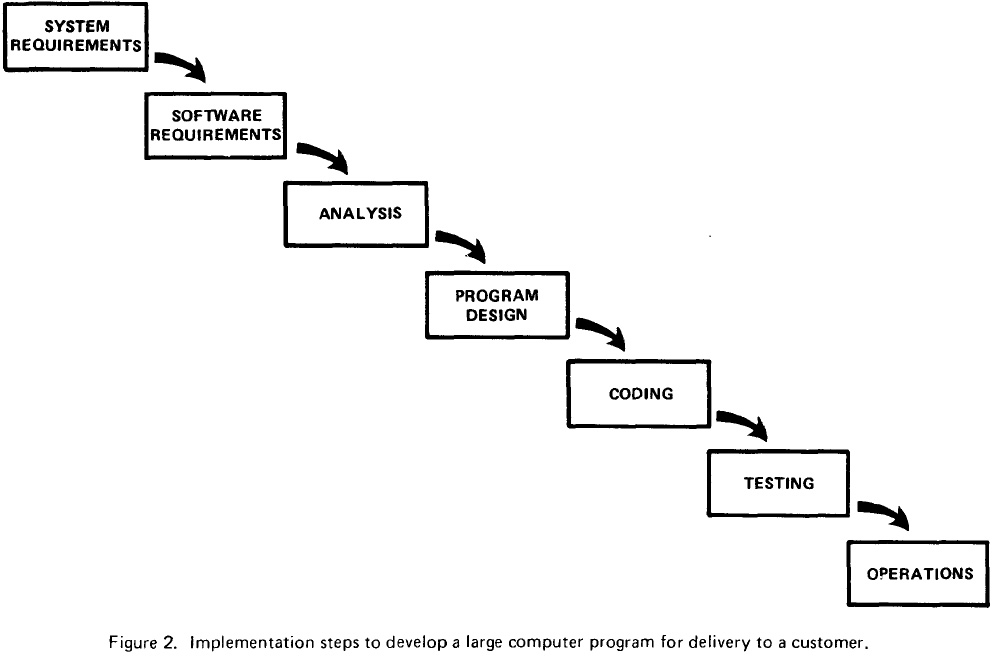
El conocido Modelo en Cascada nunca fue así denominado por su creador. Royce, en 1970, escribe un artículo [Royce 70] donde describe el modelo “non-working model” (modelo que no funciona) del cual se desprende luego el Modelo en Cascada debido a su proyección “secuencial lineal”. La propuesta ideal de Royce consiste en una secuencia de etapas (de aquí se desprende lo secuencial) las cuales hacen imposible el desarrollo si se está en constante cambio, por tal motivo propone no retornar a etapas anteriores (de aquí el se desprende el concepto de lineal). Pero la realidad es otra y concibe que el mismo modelo pueda tener retroalimentación.

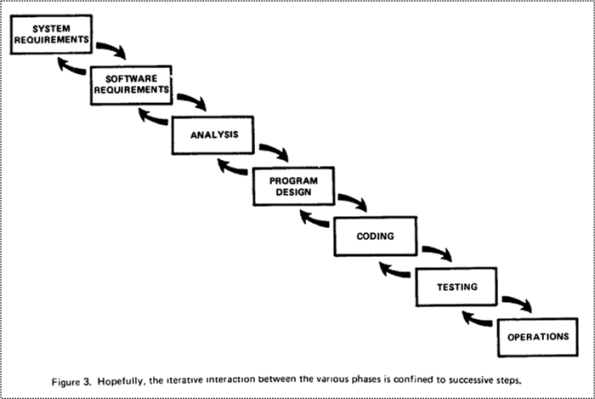
Este modelo considera a cada actividad del proceso como una fase autónoma que produce una salida o documento aprobado. La siguiente fase comienza cuando se ha finalizado con la anterior. Las fases se suceden en orden estrictamente secuencial.

Interesante pasaje del artículo original de Royce:

“Winston is aware that software designs are subject to wide interpretation even after previous agreement. So he suggested to involve customers regularly. For that, a dedicated person (we call her the product owner (PO)) is representing a wider audience of stakeholders. But other users are also invited to give feedback on a regular basis during so called sprint reviews or show & tells.”

****



****

*Ventajas:*

* Fácil de administrar. El modelo da una buena visibilidad[[1]](#footnote-2) del proceso, es decir, permite detectar fácilmente el avance en el desarrollo, debido a que cada actividad produce un artefacto, ya sea un documento, modelo o software.
* En cada fase están bien definidas las salidas a producir para avanzar a la siguiente etapa.
* El proceso de desarrollo es claro de entender por los clientes.
* La separación de análisis, diseño e implementación conduce a sistemas robustos[[2]](#footnote-3), lo que facilita el cambio posterior.

*Desventajas:*

* Modelo inflexible: presenta dificultades para hacer cambios entre etapas.
* Este modelo tiene una visión estática de la ingeniería de requisitos, ignora la volatilidad natural de los requisitos y cómo repercute ésta en las etapas del desarrollo. No está preparado para responder a cambios en los requisitos pues se deberían rehacer las fases. Los cambios deben ser ignorados o deben realizarse “por fuera del proceso”, lo que provoca que los documentos se vayan tornando obsoletos con el paso del tiempo.
* El congelamiento prematuro de los requisitos puede implicar que el sistema no haga lo que los usuarios desean.
* Los clientes y usuarios no participan en etapas posteriores a la especificación de requisitos sino hasta la prueba de aceptación. Por lo tanto, no hay un compromiso de los clientes y usuarios a lo largo de todo el ciclo de vida del software.
* Este modelo no trata al proceso de software como un proceso de resolución de problemas, pues presenta una visión de manufactura sobre el desarrollo de software, es decir, producir un artículo en particular y reproducirlo muchas veces. Este modelo no trata los avances y retrocesos normales hasta crear el producto final (el sistema de software).
* Errores y omisiones en los requisitos originales se descubren recién en las fases finales del ciclo.

## MODELO EN ESPIRAL

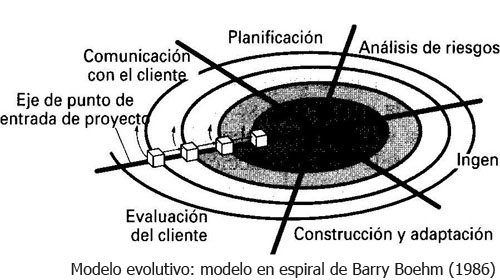
### MODELO PRESCRIPTIVO

El modelo evolutivo en espiral es la experiencia realizada por Boehm [Boehm 88] durante varios años de utilizar el modelo en cascada para proyectos de gobierno. O sea, que este modelo resulta de un refinamiento exhaustivo del modelo en cascada al que se le agregó el desarrollo incremental.

Cada ciclo del espiral comienza con la identificación de objetivos de la porción de producto que se va a elaborar, la implementación de la porción del producto (diseño, reuso, etc.) y sus restricciones (costo, tiempos, etc.).

Cada espira produce un producto software que puede ser una versión del software o un componente del mismo. Por lo tanto, el concepto de espiral se basa en la idea de volver a repetir todos los pasos por cada producto de software realizado. También puede darse que se realicen diferentes espirales en paralelo para desarrollar diferentes componentes o incrementos.

Es un modelo dirigido a los riesgos, y estos son los referidos a costos, tiempos, no comprensión de los requisitos para cada porción del software que se esté realizando. Este análisis de riesgos implica detectarlos, evaluarlos y prever caminos alternativos si es necesario.





La aplicación del modelo en cascada en la concepción del modelo en espiral:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Waterfall Model** | **Spiral Model** |  |  |  | | --- | --- | | Design Specifications | Detailed design | | Programming | Code, Unit test | | Integration | Integration and test | | Delivery | Acceptance test, Implementation | | spiral-water |

*Ventajas:*

* Incorpora muchas de las ventajas de los otros ciclos de vida
* Conjuga la naturaleza iterativa de los prototipos con los aspectos controlados y sistemáticos del modelo clásico
* El modelo en espiral puede adaptarse y aplicarse a lo largo de la vida del software de computadora.
* Como el software evoluciona a medida que progresa el proceso, el desarrollador y el cliente comprenden y reaccionan mejor ante riesgos en cada uno de los nivele evolutivos.
* El modelo en espiral permite a quien lo desarrolla aplicar el enfoque de construcción de prototipos en cualquier etapa de evolución del producto.
* El modelo en espiral demanda una consideración directa de los riesgos técnicos en todas las etapas del proyecto y si se aplica adecuadamente debe reducir los riesgos antes de que se conviertan en problemas.
* Proporciona el potencial para el desarrollo rápido de versiones incrementales
* Permite aplicar el enfoque de construcción de prototipos en cualquier momento para reducir riesgos

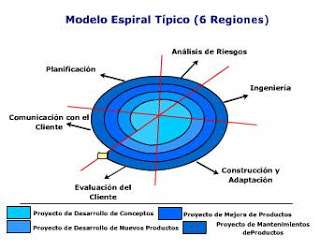
*Desventajas:*

* Solo resulta aplicable para proyectos de gran tamaño
* Supone una carga de trabajo adicional, no presente en otros ciclos de vida
* Requiere una considerable habilidad para la evaluación y resolución del riesgo, y se basa en esta habilidad para el éxito
* Si un riesgo importante no es descubierto y gestionado, indudablemente surgirán problemas
* Es bastante complicado de realizar y su complejidad puede incrementarse hasta hacerlo impracticable
* El modelo no se ha utilizado tanto como otros, por lo que tendrán que pasar años antes de que determine con certeza la eficacia de este modelo

### [TIPOS](http://modeloespiral.blogspot.com.ar/2009/08/tipos.html)

El modelo espiral tuvo dos modificaciones principales que son:

* Modelo Típico de Seis Regiones.
* Modelo WINWIN.





## Estrategia de PROTOTIPADO

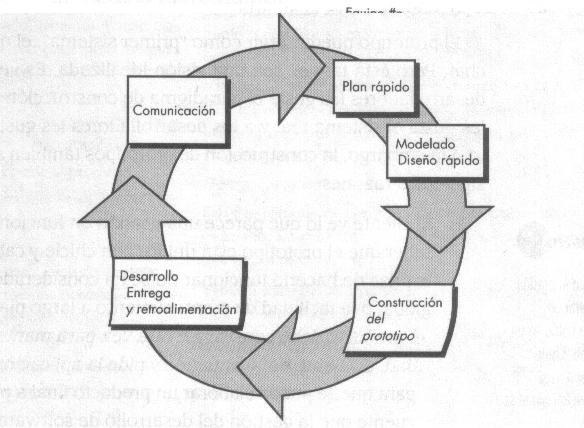
### MODELO PRESCRIPTIVO

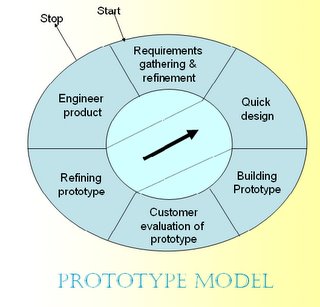
La estrategia de Prototipado [Floyd 84] consiste en construir un mecanismo para generar software que sea evaluado por el cliente en conjunto con el programador. Es importante recalcar que para Floyd esta estrategia no es en sí misma un método para desarrollar sistemas de software, más bien debe ser considerado como un procedimiento dentro de la construcción del software. La construcción de un prototipo suele ser muy adecuado al comienzo de la etapa de análisis, ya que el prototipo es el único medio a través del cual se pueden obtener de una manera más eficaz los requisitos.

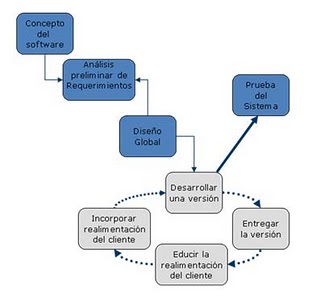
Según Floyd, existen tres enfoques para utilizar prototipos: el prototipo exploratorio, el prototipo experimental y el prototipo evolutivo.

El prototipo exploratorio se utiliza en etapas muy tempranas del desarrollo con el objetivo de clarificar o elicitar requisitos del software. El prototipo experimental sirve para simular aspectos o partes de un sistema de software para evaluar aspectos técnicos desconocidos. Existen dos tipos de Prototipo Evolutivo:

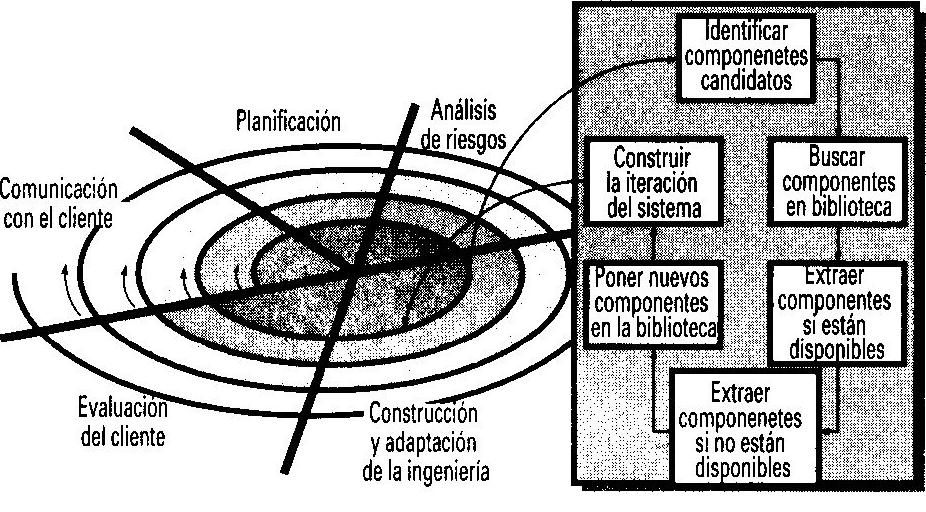
1. Desarrollo incremental (“slowly growing systems”). El sistema evoluciona gradualmente en incrementos parciales. No hay cambios en el diseño. Las necesidades del usuario son conocidas al principio y se realiza un diseño completo que se realiza en incrementos con un prototipo.
2. Desarrollo evolutivo. Mira al desarrollo como una secuencia de ciclos: re-diseño, re-implementación, re-evaluación. Producción del software en un ambiente dinámico y de cambios. Los requisitos no se terminan de conocer nunca y se van descubriendo constantemente. El sistema evoluciona de manera continua, por lo tanto nunca se distingue una etapa de mantenimiento.



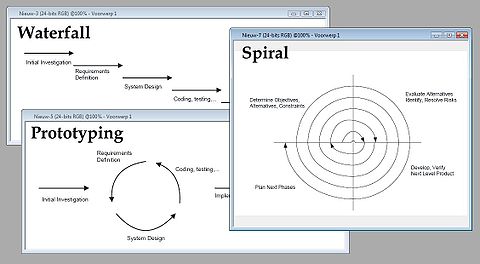




## MODELO ESPIRAL CON REUTILIZACION



## MODELO ESPIRAL CON CASCADA Y PROTOTIPOS



## MODELO DE TRANSFORMACIÓN

### MODELO PRESCRIPTIVO

Este modelo, propuesto por Robert Balzer en 1983 [Balzer 83], aplica una serie de transformaciones usando un soporte automatizado para convertir una especificación formal (modelo matemático) en un sistema implement able (ejecutable). Es decir, este paradigma intenta automatizar las etapas de diseño e implementación usando el concepto de transformación. También se denomina a este paradigma Síntesis Automática de Software.

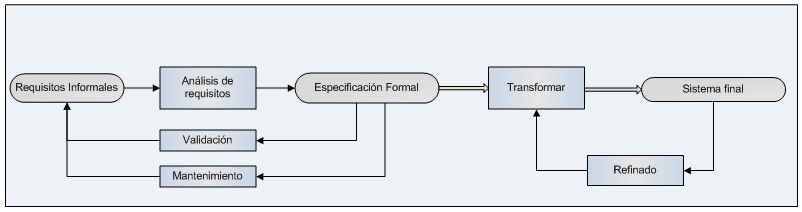
La especificación formal se convierte en forma sistemática en una representación más detallada del sistema, matemáticamente correcta. Cada paso agrega detalle hasta que la especificación formal se convierte en un programa equivalente. Como hay muchos caminos a seguir desde la especificación hasta el sistema final, la secuencia de transformaciones y su justificación se reflejan en un registro formal de desarrollo. Se utilizan técnicas de validación del modelo matemático, como la Simulación.

La especificación de requisitos se refina en una especificación formal detallada, expresada en notación matemática. Los procesos de diseño, implementación y prueba de unidades se reemplaza por un proceso de transformaciones donde la especificación formal se refina hasta llegar a un software.

Se puede citar como ejemplos de lenguajes de especificación formal:

VDM [Jones 80], CSP [Hoare 85], Larch [Guttag 85], Z [Spivey 89], B

[Wordsworth 96] entre otros.



### Ventajas:

La especificación formal representa los requisitos del sistema en una forma precisa y no ambigua.

Reduce la posibilidad de error por la eliminación de varios pasos de desarrollo. Si la especificación es correcta entonces se garantiza que el sistema final también es correcto. Por lo tanto, sólo es necesario validar la especificación formal.

Buena visibilidad: en cada transformación se genera un registro o documento.

Reduce el esfuerzo de mantenimiento, dado que los cambios se hacen sobre la especificación, la cual es más fácil de entender10 y modificar que el código.

Reduce o elimina el esfuerzo de verificación de la especificación formal.

### Desventajas:

Se requiere mucho tiempo y esfuerzo para construir una especificación formal con precisión. Requiere tecnología específica y personal altamente especializada.

Dificultad de validación, pues las especificaciones formales son difíciles de entender por los clientes y usuarios.

Es difícil la automatización del proceso de transformación.

Aunque intuitivamente parezca que, es tan probable o aún más probable que se cometan errores en el desarrollo de una especificación formal frente a la codificación de un programa, se ha observado en proyectos reales que las especificaciones formales tienen menos errores [Vienneau 93].

## ENFOQUE INCREMENTAL

### MODELO PRESCRIPTIVO

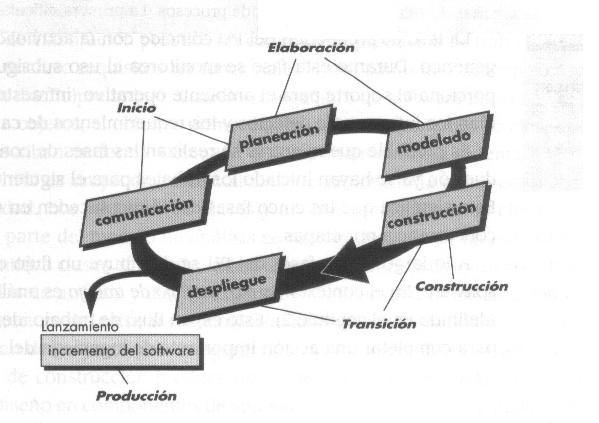
La propuesta del enfoque es diseñar sistemas que puedan entregarse por piezas.

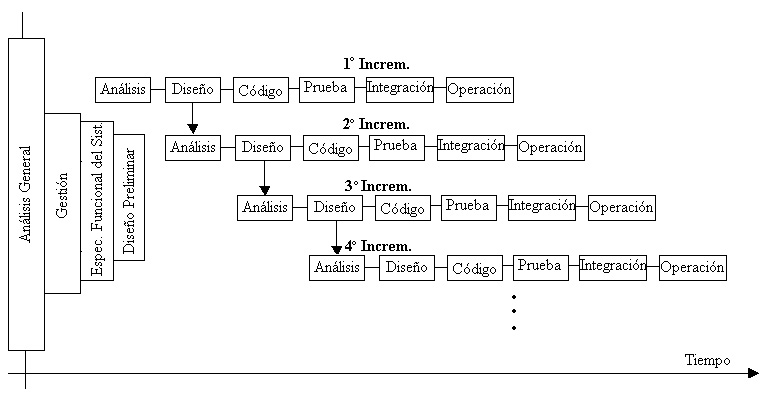
* A partir de la evaluación se planea el siguiente incremento y así sucesivamente
* Es interactivo por naturaleza
* Es útil cuando el personal no es suficiente para la implementación completa
* En lugar de entrega del sistema en una sola entrega, el desarrollo y la entrega están fracturados bajo incrementos, con cada incremento que entrega parte de la funcionalidad requerida.
* Los requerimientos del usuario se priorizan y los requerimientos de prioridad más altos son incluidos en los incrementos tempranos.
* Hechos de incrementos tempranos como un prototipo, ayudan a obtener requisitos para los incrementos más tardíos.
* Los usuarios no tiene que esperar.
* Pueden aumentar el coste debido a las pruebas.
* El desarrollo incremental es el proceso de construcción siempre incrementando subconjuntos de requerimientos del sistema.
* El modelo incremental presupone que el conjunto completo de requerimientos es conocido al comenzar
* Se evitan proyectos largos y se entrega “Algo de valor” a los usuarios con cierta frecuencia
* El usuario se involucra más
* Riesgos largos y complejos.
* Difícil de aplicar a sistemas transaccionales que tienden a ser integrados y a operar como un todo
* Requiere gestores experimentados
* Los errores en los requisitos se detectan tarde.

Bajo este modelo se entrega software “por partes funcionales más pequeñas”, pero reutilizables, llamadas incrementos. En general cada incremento se construye sobre aquel que ya fue entregado.

Beneficios:

* Construir un sistema pequeño es siempre menos riesgoso que construir un sistema grande.
* Al ir desarrollando parte de las funcionalidades, es más fácil determinar si los requerimientos planeados para los niveles subsiguientes son correctos.
* Si un error importante es realizado, sólo la última iteración necesita ser descartada.
* Reduciendo el tiempo de desarrollo de un sistema (en este caso en incremento del sistema) decrecen las probabilidades que esos requerimientos de usuarios puedan cambiar durante el desarrollo.
* Si un error importante es realizado, el incremento previo puede ser usado.  
  Los errores de desarrollo realizados en un incremento, pueden ser arreglados antes del comienzo del próximo incremento
* El resultado puede ser muy positivo





## BASADO EN REUTILIZACION

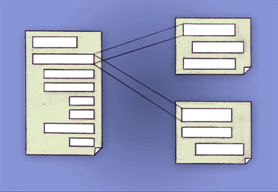
El Modelo basado en Reutilización [Jones 84] se centra en integrar un gran número de componentes de software existentes para construir un sistema, más que en desarrollarlos desde cero. El proceso de ingeniería de requisitos involucra una primera fase de definición de requisitos, seguida de un análisis de componentes existentes que cubran los requisitos especificados y posteriormente una fase de modificación de requisitos en función de los componentes disponibles. Si la modificación no es aceptable entonces se retorna a la fase de análisis de componentes.

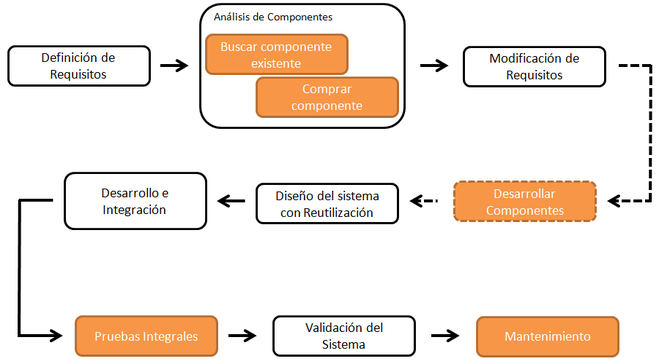
En la mayoría de los proyectos de software, existe algo de reutilización de software cuando se conocen diseños o códigos similares al requerido para el proyecto. Esta reutilización informal es independiente del proceso de desarrollo. Pero ha surgido un enfoque de desarrollo que se basa casi exclusivamente en la reutilización, integrando un gran número de componentes y de productos COTS[[3]](#footnote-4).

Cabe observar que el reuso de componentes no está limitado exclusivamente a componentes ejecutables, sino también al reuso de componentes de diseño e incluso reuso de requisitos.

Presenta una visibilidad moderada. Es importante contar con documentación de los componentes reutilizables.







## THE PRISM – Modelo de Cambios

## MODELO DESCRIPTIVO

(Texto de la Tesis “Un modelo del proceso de desarrollo de software guiado por la traceability” UNLP, 2006)

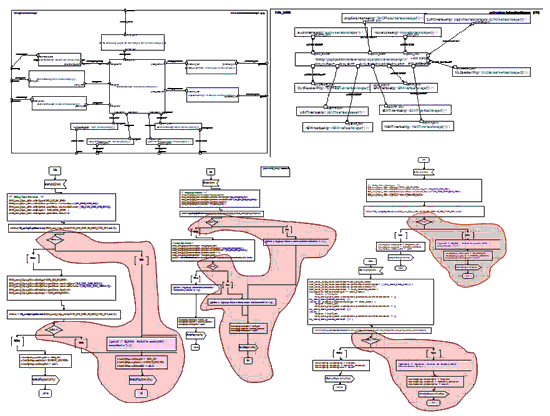
Con el fin de poder gestionar la problemática relacionada con el cambio en los proyectos de software en [Madhavji, 1991] [Madhavji, 1992] se define una descripción abstracta denominada Prism – Modelo del Cambio. Es importante destacar, que este modelo forma parte de un proyecto de investigación (Proyecto Prism), que estudia los distintos ambientes de desarrollo de software [Madhavji, 1991].

La gestión de los cambios en un proyecto de software involucra la posibilidad de identificar los diversos ítems afectados por éste, como así también la clasificación y registro de los datos relacionados con el cambio ocurrido [Madhavji, 1992]. Prism lleva a cabo todas estas actividades especificando un Modelo del Cambio, conjuntamente con dos elementos denominados Estructura de Dependencia y Estructura del Cambio, los cuales conforman la Infraestructura del Ambiente [Madhavji, 1991] [Madhavji, 1992]. El Modelo del Cambio representa los diferentes niveles de operación que se implementan en Prism, como así también la infraestructura encargada de gestionar las acciones que se producen debido a la implementación de un cambio ocurrido. Además se definen los procesos que se deben llevar a cabo durante el uso y evolución de los demás elementos que conforman el modelo Prism [Madhavji, 1992]. Este modelo es explícito (describe cuáles son los componentes importantes del mismo, conjuntamente con sus interacciones), incremental y continuo. Esta última característica indica que el modelo abarca todo el ciclo de vida del desarrollo. Como consecuencia de este hecho se mantiene el historial de los cambios producidos durante el desarrollo del proyecto. El modelo presenta dos niveles de operaciones. Aquellas que abarcan el uso de los elementos de Infraestructura del Ambiente constituyen el nivel normal, el cual se diferencia del metanivel de operaciones. Este metanivel utiliza la información generada en el nivel normal con el propósito de mejorar la infraestructura del ambiente [Madhavji, 1992]. La Estructura de Dependencia especifica todo lo necesario para poder describir los ítems de cambios y sus interrelaciones. Se logra de esta manera identificar cuáles son los ítems afectados debido a un determinado cambio [Madhavji, 1992]. En esta estructura también se definen los distintos niveles de abstracción (individuo, grupo, proyecto, organización, nación, comunidad) de los ítems de cambio, con el objetivo de poder categorizarlos. En cada uno de estos niveles se especifican los ítems básicos y el conjunto de relaciones, que se denomina sheet. Los ítems de cambio se clasifican en personas, políticas, leyes, procesos de desarrollo de software, recursos y resultados producidos. La Estructura del Cambio, en cambio, posee el objetivo de facilitar la clasificación, almacenamiento y análisis de los datos relacionados con el cambio producido [Madhavji, 1992] [Wieringa, 1995]. Ésta especifica los mecanismos necesarios para obtener información de retroalimentación relacionada con los distintos ítems de cambio [Madhavji, 1992]. La integración con la Estructura de Dependencia representa una de las características principales de esta estructura. Tal integración permite asociar los ítems de cambio de la estructura de dependencia (definidos en el componente sheet) con un conjunto de componentes de la estructura de cambio denominados CS-sheet (cada CS-sheet describe un cambio particular) [Madhavji, 1992]. En la Estructura de Cambio se especifica un framework para el cambio. El framework representa un esquema de clasificación que se especializa en capturar la información relacionada con un cambio específico. Éste debe estar descripto por un conjunto de propiedades. Tales propiedades abarcan, entre otras cuestiones, datos que especifican el tipo de cambio (correctivo, perfectivo), la ubicación del ítem en el ambiente, la justificación del cambio (origen de la decisión, ventajas y desventajas, razón), tamaño del cambio (impacto, costo), stakeholders responsables de la decisión, entre otras propiedades [Wieringa, 1995] [Madhavji, 1992].

## ORIENTADO A ASPECTOS

Los conceptos de orientación a aspectos nacieron en 1996, cuando Gregor Kiczales propuso una nueva idea: La Programación Orientada a Aspectos (POA) o Aspect Oriented Programming en su denominación en inglés. A partir de entonces, la POA y otras técnicas y tecnologías, que se centraban en la modularización del código que atraviesa varios componentes para realizar una cierta función (crosscutting code), se agruparon bajo el nombre de Advanced.

El enfoque orientado a aspectos define un mecanismo que ayuda a resolver problemas de codificación en los requisitos, los cuales son un código disperso (scattered) y diseminado (tangled), que no se resuelven fácilmente usando el enfoque orientado a objetos. Este mecanismo se enfoca principalmente en la separación de intereses (separation of concerns) de un sistema para obtener una mejor modularización.



El desarrollo de software orientado a aspectos (DSOA) se enfoca en crear una mejor abstracción modular del sistema. Incluye las siguientes fases:

- Captura de requisitos

- Análisis

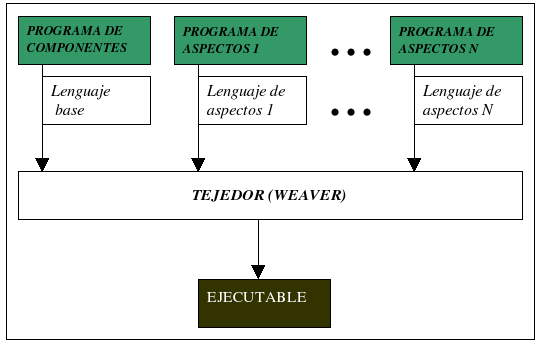
- Diseño

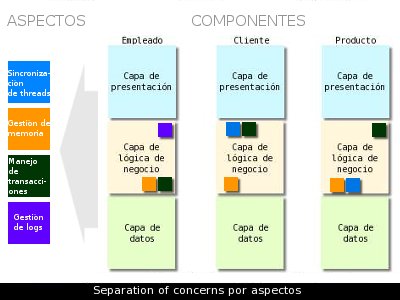
- Implementación

- Pruebas

La primera fase trata la separación de intereses tanto los funcionales como los no funcionales; los requisitos funcionales son modelados con casos de uso que representan la función básica del sistema y los requisitos no funcionales se representan con casos de uso de infraestructura. En el análisis y el diseño los casos de uso se representan en una estructura de composición que se identifica con el estereotipo <<use case slice>> y agrupa elementos de modelo que colaboran para lograr los requisitos del sistema tanto funcionales como no funcionales. En la implementación se genera el código de las clases y aspectos.

Por último en las pruebas se diseñan las pruebas tanto para los casos de uso de la aplicación como para los casos de uso slice.





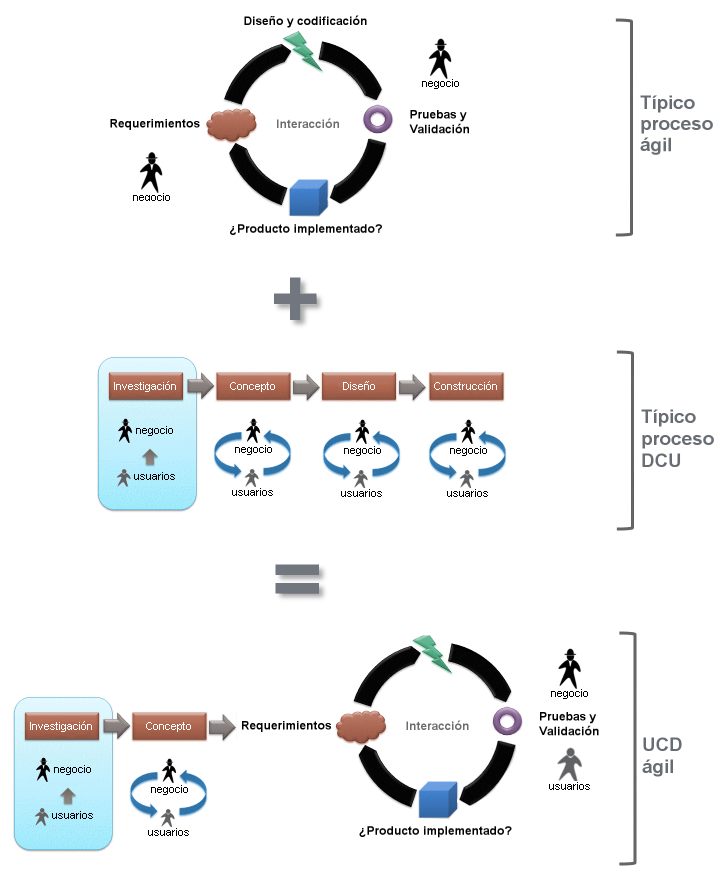
## DESARROLLO ÁGIL

En febrero de 2001, tras una reunión celebrada en Utah-EEUU, nace el término ágil aplicado al desarrollo de software. En esta reunión participan un grupo de 17 expertos de la industria del software, incluyendo algunos de los creadores o impulsores de metodologías de software. Su objetivo fue esbozar los valores y principios que deberían permitir a los equipos desarrollar software rápidamente y respondiendo a los cambios que puedan surgir a lo largo del proyecto.

Se pretendía ofrecer una alternativa a los procesos de desarrollo de software tradicionales, caracterizados por ser rígidos y dirigidos por la documentación que se genera en cada una de las actividades desarrolladas.

Tras esta reunión se creó The Agile Alliance3, una organización, sin ánimo de lucro, dedicada a promover los conceptos relacionados con el desarrollo ágil de software y ayudar a las organizaciones para que adopten dichos conceptos. El punto de partida es fue el Manifiesto Ágil, un documento que resume la filosofía ágil.





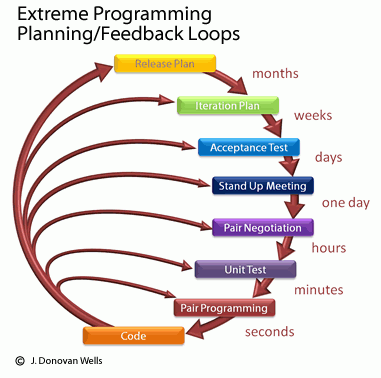


**XP (XTREME PROGRAMING)**

Las características fundamentales del método XP son:

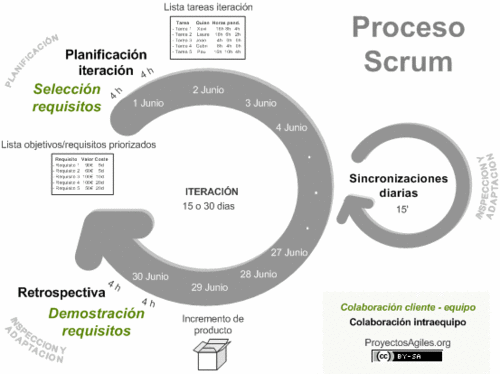
* **Desarrollo iterativo e incremental**: pequeñas mejoras, unas tras otras.
* [**Pruebas unitarias**](http://es.wikipedia.org/wiki/Prueba_unitaria)**continuas**, frecuentemente repetidas y automatizadas, incluyendo [pruebas de regresión](http://es.wikipedia.org/wiki/Pruebas_de_regresi%C3%B3n). Se aconseja escribir el código de la prueba antes de la codificación. Véase, por ejemplo, las herramientas de prueba [JUnit](http://es.wikipedia.org/wiki/JUnit) orientada a Java, DUnit orientada a Delphi y NUnit para la plataforma.NET. Estas dos últimas inspiradas en JUnit.
* [**Programación en parejas**](http://es.wikipedia.org/wiki/Programaci%C3%B3n_en_pareja): se recomienda que las tareas de desarrollo se lleven a cabo por dos personas en un mismo puesto. Se supone que la mayor calidad del código escrito de esta manera -el código es revisado y discutido mientras se escribe- es más importante que la posible pérdida de productividad inmediata.
* Frecuente **integración del equipo de programación con el cliente** o usuario. Se recomienda que un representante del cliente trabaje junto al equipo de desarrollo.
* **Corrección de todos los**[**errores**](http://es.wikipedia.org/wiki/Error_de_programaci%C3%B3n) antes de añadir nueva funcionalidad. Hacer entregas frecuentes.
* [**Refactorización**](http://es.wikipedia.org/wiki/Refactorizaci%C3%B3n)**del código**, es decir, reescribir ciertas partes del código para aumentar su legibilidad y mantenibilidad pero sin modificar su comportamiento. Las pruebas han de garantizar que en la refactorización no se ha introducido ningún fallo.
* **Propiedad del código compartida**: en vez de dividir la responsabilidad en el desarrollo de cada módulo en grupos de trabajo distintos, este método promueve el que todo el personal pueda corregir y extender cualquier parte del proyecto. Las frecuentes pruebas de regresión garantizan que los posibles errores serán detectados.
* **Simplicidad** en el código: es la mejor manera de que las cosas funcionen. Cuando todo funcione se podrá añadir funcionalidad si es necesario. La programación extrema apuesta que es más sencillo hacer algo simple y tener un poco de trabajo extra para cambiarlo si se requiere, que realizar algo complicado y quizás nunca utilizarlo.

La simplicidad y la comunicación son extraordinariamente complementarias. Con más comunicación resulta más fácil identificar qué se debe y qué no se debe hacer. Mientras más simple es el sistema, menos tendrá que comunicar sobre este, lo que lleva a una comunicación más completa, especialmente si se puede reducir el equipo de programadores.



**SCRUM**

|  |  |
| --- | --- |
| Beneficios de Scrum | Cómo se consiguen |
| **Gestión regular de las expectativas del cliente**  El cliente establece sus expectativas indicando el valor que le aporta cada requisito del proyecto y cuando espera que esté completado. | [**Lista de requisitos priorizada**](http://www.proyectosagiles.org/lista-requisitos-priorizada-product-backlog)  El cliente crea y gestiona la lista de requisitos del producto o proyecto, donde quedan reflejadas sus expectativas a nivel de requisitos, valor, coste y entregas. |
| El cliente comprueba de manera regular si se van cumpliendo sus expectativas, da feedback, ya desde el inicio del proyecto puede tomar decisiones informadas a partir de resultados objetivos y dirige estos resultados del proyecto, iteración a iteración, hacia su meta. Se ahorra esfuerzo y tiempo al evitar hipótesis. | [**Demostración de los resultados de proyecto en cada iteración**](http://www.proyectosagiles.org/demostracion-requisitos-sprint-review)  Al final de cada [iteración](http://www.proyectosagiles.org/desarrollo-iterativo-incremental) el equipo demuestra al cliente los requisitos que ha conseguido completar. Tras una inspección del resultado real del proyecto hasta ese momento, y considerando el esfuerzo que ha sido necesario para realizarlo, el cliente solicita los cambios que necesita y replanifica el proyecto. |
| **Resultados anticipados (“time to market”)**  El cliente puede empezar a utilizar los resultados más importantes del proyecto antes de que esté finalizado por completo.  Siguiendo la ley de Pareto (el 20% del esfuerzo proporciona el 80% del valor), el cliente puede empezar antes a recuperar su inversión (y/o autofinanciarse) comenzando a utilizar un producto al que sólo le faltan características poco relevantes, puede sacar al mercado un producto antes que su competidor, puede hacer frente a urgencias o nuevas peticiones de clientes, etc. | [**Priorización de requisitos por valor y coste**](http://www.proyectosagiles.org/priorizacion-requisitos-valor-coste)  Al inicio de cada iteración el cliente prioriza la lista de requisitos del producto o proyecto en función del valor que le aportan, su coste de desarrollo y los riesgos del proyecto, cambiando los requisitos previstos para reaccionar a cambios de contexto en el proyecto.  El progreso del proyecto se mide en función de los requisitos que el equipo completa en cada iteración. |
| **Flexibilidad y adaptación**  De manera regular el cliente redirige el proyecto en función de sus nuevas prioridades, de los cambios en el mercado, de los requisitos completados que le permiten entender mejor el producto, de la velocidad real de desarrollo, etc.  Al final de cada iteración el cliente puede aprovechar la parte de producto completada hasta ese momento para hacer pruebas de concepto con usuarios o consumidores y tomar decisiones en función del resultado obtenido. | [**Replanificación en el inicio de cada iteración**](http://www.proyectosagiles.org/replanificacion-proyecto)  Se asume que los cambios son parte natural del proyecto. Toda iteración comienza con una replanificación del proyecto. Esta replanificación no es traumática puesto que Scrum minimiza el número de objetivos/requisitos en que el equipo trabaja (WIP, Work In Progress) a los que caben en una iteración. Todavía no se ha hecho ningún esfuerzo en desarrollar los requisitos de las siguientes iteraciones.  El hecho los requisitos se completen en función del valor que aportan al cliente minimiza la probabilidad de que se produzcan grandes cambios en el transcurso del proyecto. |
| **Retorno de inversión (ROI)**  De manera regular, el cliente maximiza el ROI del proyecto. Cuando el beneficio pendiente de obtener es menor que el coste de desarrollo, el cliente puede finalizar el proyecto. | [**Priorización de requisitos por valor**](http://www.proyectosagiles.org/priorizacion-requisitos-valor-coste)  Cada iteración el cliente dispone de unos requisitos completados y replanifica el proyecto en función del valor que le aportan los requisitos pendientes respecto del coste de desarrollo que tienen. |
| **Mitigación de riesgos**  Desde la primera iteración el equipo tiene que gestionar los problemas que pueden aparecer en una entrega del proyecto. Al hacer patentes estos riesgos, es posible iniciar su mitigación de manera anticipada. "Si hay que equivocarse o fallar, mejor hacelo lo antes posible". El feedback temprano permite ahorrar esfuerzo y tiempo en errores técnicos.  La cantidad de riesgo a que se enfrenta el equipo está limitada a los requisitos que se puede desarrollar en una iteración. La complejidad y riesgos del proyecto se dividen de manera natural en iteraciones. | [**Desarrollo iterativo e incremental**](http://www.proyectosagiles.org/desarrollo-iterativo-incremental)  Un requisito se debe completar en una iteración. El equipo debe realizar todas las tareas necesarias para completarlo y que esté preparado para ser entregado al cliente con el esfuerzo mínimo necesario. De esta manera no se deja para el final del proyecto ninguna actividad arriesgada relacionada con la entrega de requisitos. |
| **Productividad y calidad**  De manera regular el equipo va mejorando y simplificando su forma de trabajar. | [**Mejora continua**](http://www.proyectosagiles.org/control-predictivo-control-empirico)  Cada iteración el equipo realiza una retrospectiva para analizar su manera de trabajar e identificar los obstáculos que le impiden avanzar al mejor ritmo posible. |
| Los miembros del equipo sincronizan su trabajo diariamente y se ayudan a resolver los problemas que pueden impedir conseguir el objetivo de la iteración. La comunicación y la adaptación a las diferentes necesidades entre los miembros del equipo son máximas (se van ajustando iteración a iteración), de manera que no se realizan tareas innecesarias y se evitan ineficiencias. | [**Comunicación diaria del equipo**](http://www.proyectosagiles.org/reunion-diaria-de-sincronizacion-scrum-daily-meeting)  Todo miembro del equipo conoce cómo el trabajo de los otros miembros impacta en el suyo y cuáles son las necesidades de los otros. |
| Las personas trabajan más enfocadas y de manera más eficiente cuando hay una fecha límite a corto plazo para entregar un resultado al que se han comprometido. La consciencia de esta limitación temporal favorece la priorización de las tareas y fuerza la toma de decisiones.  Las iteraciones (Sprints) son regulares y de un mes para facilitar la sincronización sistemática con otros equipos, con el resto de la empresa y con el cliente. | [**TimeBoxing**](http://www.proyectosagiles.org/timebox)  Cada actividad de Scrum siempre tiene la misma duración (1 mes, 4 horas, etc.), con lo que las personas aprenden lo que pueden conseguir en este tiempo, cómo organizarse, priorizar tareas y tomar decisiones. |
| El equipo minimiza su dependencia de personas externas para poder avanzar (depender de la disponibilidad de otros puede parar tareas). | [**Equipo multidisciplinar**](http://www.proyectosagiles.org/equipo-team)  El equipo está formado por todas las personas con las especialidades necesarias para llevar a cabo el proyecto. |
| La estimación de esfuerzo y la optimización de tareas para completar un requisito es mejor si la realizan las personas que van a desarrollar el requisito, dadas sus diferentes especializaciones, experiencias y puntos de vista. Asímismo, con iteraciones cortas la precisión de las estimaciones aumenta. | [**Estimación de esfuerzo conjunta**](http://www.proyectosagiles.org/planificacion-iteracion-sprint-planning)  En el inicio de la iteración los miembros del equipo estiman de manera conjunta el esfuerzo necesario para completar requisitos y sus tareas. |
| Las personas trabajan de manera más eficiente y con más calidad cuando ellas mismas se han comprometido a entregar un resultado en un momento determinado y deciden cómo hacerlo, no cuando se les ha asignado una tarea e indicado el tiempo necesario para realizarla. | [**Compromiso del equipo**](http://www.proyectosagiles.org/planificacion-iteracion-sprint-planning)  En el inicio de cada iteración el equipo selecciona los requisitos que se compromete a completar y entregar al final de la iteración (responabilidad). El propio equipo se organiza (autoridad) identificando las tareas necesarias, su esfuerzo y autoasignandose cada miembro las tareas que se compromete a realizar. |
| El equipo se evita caminar mucho tiempo por un camino equivocado que le obligue a realizar un gran esfuerzo para llegar al objetivo esperado  Se asegura la [calidad del producto](http://www.proyectosagiles.org/calidad-agilidad-cuarto-encuentro-agil-barcelona) de manera sistemática y objetiva, a nivel de satisfacción del cliente, requisitos listos para ser utilizados y calidad interna del producto. | [**Demostración de resultados preparados para ser utilizados**](http://www.proyectosagiles.org/demostracion-requisitos-sprint-review)**y velocidad sostenida**  Por un lado, al final de cada iteración el equipo demuestra al cliente los requisitos que ha conseguido completar, de manera que están completamente operativos. Por otro lado, para tener una velocidad de desarrollo sostenida, el equipo necesita desarrollar cada incremento de producto sin tener que revisitar aspectos mal resueltos en iteraciones anteriores. |
| **Alineamiento entre cliente y equipo**  Los resultados y esfuerzos del proyecto se miden en forma de objetivos y requisitos entregados al negocio. Todos los participantes en el proyecto conocen cuál es el objetivo a conseguir. El producto se enriquece con las aportaciones de todos. | **Cliente y equipo trabajando “en equipo”**  Cada iteración el equipo y el cliente trabajan juntos en la creación de los requisitos del proyecto (en la [estimación de la lista priorizada de requisitos del proyecto](http://www.proyectosagiles.org/replanificacion-proyecto)), en darles detalle (en la [reunión de planificación de la iteración](http://www.proyectosagiles.org/planificacion-iteracion-sprint-planning)) y en el análisis del resultado obtenido (en la[demostración de los requisitos completados](http://www.proyectosagiles.org/demostracion-requisitos-sprint-review)). |
| **Equipo motivado**  Las personas están más motivadas cuando pueden usar su creatividad para resolver problemas y cuando pueden decidir organizar su trabajo. | [**Equipo autogestionado**](http://www.proyectosagiles.org/potenciacion-equipo)  El equipo es quien se compromete a completar unos requisitos determinados en una iteración y quien mejor sabe cómo desarrollarlos. Por ello es el equipo quien se autoorganiza y quien planifica cómo trabajará en la iteración. |
| Las personas se sienten más satisfechas cuando pueden  mostrar los logros que consiguen. | [**Demostración**](http://www.proyectosagiles.org/demostracion-requisitos-sprint-review)  Cada iteración el equipo muestra al cliente los resultados que consigue. No está meses trabajando sin poder exhibir su obra. |



## PROCESO UNIFICADO

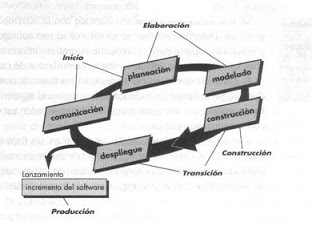
### MODELO PRESCRIPTIVO

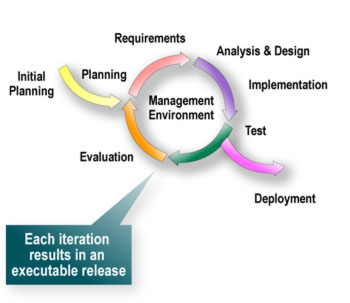
 Ivar Jacobson, Grady Booch y James Rumbaugh (1999)

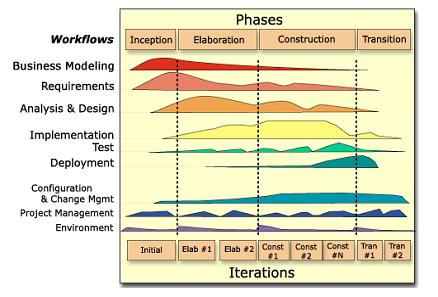
El Proceso Unificado propuesto por IBM, que incluye prácticas claves y aspectos relacionados a la planeación estratégica y administración de riesgos; y actualmente guían de forma natural el proceso de desarrollo de software complejo por lo que ha sido considerado como un estándar el desarrollo de software en las empresas.

El proceso unificado conocido como RUP, es un modelo de software que permite el desarrollo de software a gran escala, mediante un proceso continuo de pruebas y retroalimentación, garantizando el cumplimiento de ciertos estándares de calidad. Aunque con el inconveniente de generar mayor complejidad en los controles de administración del mismo. Sin embargo, los beneficios obtenidos recompensan el esfuerzo invertido en este aspecto.

El proceso de desarrollo constituye un marco metodológico que define en términos de metas estratégicas, objetivos, actividades y artefactos (documentación) requerido en cada fase de desarrollo. Esto permite enfocar esfuerzo de los recursos humanos en términos de habilidades, competencias y capacidades a asumir roles específicos con responsabilidades bien definidas.







## CONCLUSIONES

El problema crucial que atraviesan los métodos de la Ingeniería de Software es tratar con requisitos siempre cambiantes. La evolución de los errores y la evolución de los requisitos son dos características del desarrollo de software estrechamente vinculadas. Pues las causas de los cambios en el software no son necesariamente debido a cambios en los requisitos por cambios en el universo de discurso, sino muchas veces por reparación de errores o por una mejora en la comprensión de los requisitos.

Belady y Lehman [Belady 76] construyeron un modelo que representa la evolución de los errores en un sistema de software a lo largo del tiempo, en función de las versiones del software que se generan para corregir errores en los programas y actualizar / mejorar la funcionalidad del mismo. El punto mínimo de la curva representa la versión de software con la menor cantidad de errores, pasado el cual la curva asciende rápidamente, pues cualquier cambio a realizar en el software se torna cada vez más dificultoso debido a un paulatino y constante desmejoramiento en su arquitectura, lo cual a su vez facilita la gestación de nuevos errores. Los métodos de la Ingeniería de Software deberían tender a que la parte ascendente de la curva de Belady- Lehman posterior al punto mínimo sea lo más suave posible postergando el aumento de la entropía del sistema.

Por otro lado, Davis, Bersoff y Comer [Davis 88] definieron métricas para comparar los modelos de proceso de software, mostrando sus similitudes y diferencias. Extraídas de [Davis 88], donde se compara la satisfacción de las necesidades de los usuarios según los distintos modelos de ciclo de vida. Esta comparación parte de la premisa que las necesidades de los usuarios evolucionan con el tiempo. En un extremo se muestra la función que representa las necesidades de usuarios12 y en el otro extremo, cómo el modelo de cascada las cubre en cantidad de funcionalidad incluida y en tiempo.

El resto de los modelos están representados por funciones que se acercan más a la función de necesidades que el enfoque convencional. Es decir, estos modelos disminuyen:

i) el tiempo para satisfacer una necesidad desde su ocurrencia, y

ii) la distancia entre las funcionalidades incluidas en el sistema en un momento dado y el total de necesidades requeridas hasta ese momento.

Los modelos desarrollados con posterioridad al modelo convencional han mejorado de distintas formas los inconvenientes presentados por el modelo de cascada, cuyas consecuencias se manifestaron en los fracasos paradigmáticos de la crisis del software y posteriores. Pero además, se desprende que la evolución de los requisitos es prácticamente ignorada durante el desarrollo del software por cualquiera de los modelos. Es por ello que nuevos enfoques en el desarrollo de software deben apuntar a obtener resultados parciales del sistema de software que sean más adaptables a la continua evolución de los requisitos. Es decir, deben producir versiones de sistemas de software que atiendan la totalidad de las necesidades a un momento dado y que sean altamente flexibles para satisfacer rápidamente los próximos nuevos requisitos.

Los modelos de prototipado, operacional y transformación formal surgieron como solución al problema planteado por el modelo de cascada referido a la poca participación del usuario en el desarrollo. Por lo tanto, los tres modelos proponen la creación de un artefacto (prototipo, especificación operacional, especificación formal) tempranamente en el ciclo de vida, que pueda ser usado para comprender y validar los requisitos.

Cabe destacar por otro lado que los modelos que más soportan la evolución de los requisitos son aquellos que presentan procesos iterativos, como el modelo evolutivo, el incremental y el espiral, donde la especificación de los requisitos acompaña la implementación del software. Debido a lo cual, no se cuenta con un documento SRS completo final que sirva como contrato para el desarrollo del software. Lamentablemente, esto es una condición mandatoria en muchas organizaciones, principalmente gubernamentales, cuando el proveedor de software pertenece a una organización diferente de la del cliente y existe una relación contractual, formal o informal entre ellas.

Comparativamente, los modelos de prototipado, operacional y espiral, apuntan más a la validación de los requisitos, mediante la producción de un artefacto que les permita a los usuarios experimentar tempranamente.

En resumen, no existe un modelo de proceso ideal de desarrollo de software aplicable a cualquier tipo de sistema, en cualquier tipo de organización, y que garantice el mejor producto a un costo acorde. De ahí que surja una diversidad de métodos, que se basan en algunos de estos modelos o combinaciones de ellos, y que atienden algunos de los subprocesos involucrados en el desarrollo de software. Por ejemplo, el método de “Cuarto Limpio” [Mills 87] integra el modelo de transformación formal con el desarrollo incremental: en cada incremento se desarrolla y valida una especificación formal.

Es también el caso de muchas organizaciones que usan una combinación del modelo incremental con el modelo iterativo (denominada prácticas IID13), donde en cada versión se agregan funcionalidades y se mejoran funcionalidades existentes en la versión actual. Un ejemplo de método basado en desarrollo iterativo e incremental ampliamente difundido es el Rational Unified Process [Kruchten 04]. Recientemente se establecieron los “métodos ágiles” [Cockburn 02] cuando en febrero del 2001 se reunieron expertos en DSDM14, XP15, ADS16, FDD17 y otros, y formaron la “Agile Alliance”.

Estos métodos ágiles, basados en prácticas IID, sí tienen en cuenta la evolución de los requisitos, ya que en ellos todo evoluciona, pero no presentan un proceso dedicado a la definición de requisitos sino que éste es más bien informal.

Se debe tener presente que los desarrollos iterativos e incrementales, a pesar de su popularidad actual como piedra basal de los métodos ágiles, y de su puesta en conocimiento a fines de los 80 a través de reportes de resultados de proyectos y artículos científicos, realmente ya habían sido puestos en práctica y con éxito desde la década del 70 en pleno auge del desarrollo en cascada e inclusive se remontan a proyectos aislados desde fines de los 50 [Larman 03].

## Referencia Bibliográfica

|  |
| --- |
| [Balzer 83] Balzer, R., Cheatham, T.E., Green, C., “Software technology in the 1990s: using a new paradigm”, IEEE Computer, Noviembre 1983. |
| [Beck 00] Beck, K.. Extreme Programming Explained. Embrace Change, Pearson Education, 1999. Traducido al español como: Una explicación de la programación extrema. Aceptar el cambio, Addison Wesley, 2000. |
| [Belady 76] Belady, L.A., Lehman, M.M., “A Model of Large Program Development”, IBM Systems Journal, Vol.15, Nº3, 1976, pp. 225-252. |
| [Berry 02] Berry, D., “The Inevitable Pain of Software Development, Including of Extreme Programming, Caused by Requirements Volatility”, International Workshop on Time-Constrained Requirements Engineering (TCRE’02), Essen, Alemania, 2002, <http://www-di.inf.puc-rio.br/~julio/tcre-site/p2.pdf> accedido el 28-03-2005. |
| [Boehm 88] Boehm, B.W., “A Spiral Model of Software Development and Enhancement”, IEEE Computer, Vol.21, Nº 5, Mayo 1988, pp.61-72. |
| [Brooks 87] Brooks, F.P., "No Silver Bullet: Essence and Accidents of Software Engineering", IEEE Computer Society Press, Vol.20, Nº4, Abril 1987, pp.10-19. |
| [Coad 99] Coad P., Lefebvre E., De Luca J. Java Modeling In Color With UML: Enterprise Components and Process. Prentice Hall. 1999. |
| [Cockbun 01] Cockbun, A. Agile Software Development. Addison-Wesley. 2001. |
| [Cockburn 02] Cockburn, A., “Agile Software Development”, Addison-Wesley, 2002. |
| [Davis 88] Davis, A.M., Bersoff, E.H., Comer, E.R., “A Strategy for Comparing Alternative Software Development Life Cycle Models”, IEEE TSE, Vol.14, Nº10, Octubre 1988, pp.1453-1461. Reimpreso en “Software Requirements Engineering”, editores Richard H. Thayer y Merlin Dorfman, IEEE Computer Society Press, 2º edición, Los Alamitos, CA, 1997, pp.408-415. |
| [Floyd 84] Floyd, C., “A systematic look at prototyping”, en el libro Approaches to Prototyping, editor R. Buddle, Springer-Verlag, Berlín, 1984. |
| [Fowler 99] Fowler, M., Beck, K., Brant, J. Refactoring: Improving the Design of Existing Code. Addison-Wesley. 1999 |
| [GMoD 92] Germany Ministry of Defense, “V-Model: Software Lifecycle Process Model”, General Reprint Nº 250, 1992. |
| [Guttag 85] Guttag, J.V., Horning, J.J. et al., “The Larch family of specification languages”, IEEE Software, Vol.2, Nº 5, 1985, pp.24-36. |
| [Highsmith 10] Highsmith J., Orr K. Adaptive Software Development: A Collaborative Approach to Managing Complex Systems. Dorset House. 2000. |
| [Hoare 85] Hoare, C.A.R., “Communicating Sequential Processes”, Prentice-Hall, Londres, 1985. |
| [Jacobson 05] Jacobson I. and Ng P. W. Aspect-oriented Software Development with Use Cases. Addison Wesley Professional, 2005. Abrahamsson, P., Salo, O., Ronkainen, J., Warsta, J. Agile software development methods Review and analysis. VTT Publications. 2002. |
| [Jeffries 01] Jeffries, R., Anderson, A., Hendrickson, C. Extreme Programming Installed. Addison-Wesley. 2001 |
| [Jones 80] Jones, C.B., “Software Development – A Rigorous Approach”,Prentice-Hall, Londres, 1980. |
| [Jones 84] Jones, T.C., “Reusability in programming: A survey of the state of the art”, IEEE TSE, Vol.SE-10, Septiembre 1984, pp.488-494. |
| [Kruchten 04] Kruchten, P., “The Rational Unified Process: An Introduction”, Addison-Wesley, 3º edición, 2004. |
| [Larman 03] Larman, C. 2003. Agile and Iterative Development: A Manager’s Guide. AW. |
| [Larman 03] Larman, C. and Basili, V. Iterative and Incremental Development. IEEE Computer. Junio, 2003. |
| [Larman 03] Larman,C., Basili, V.R., “Iterative and Incremental Development: A Brief History”, IEEE Computer, ISSN: 0018-9162, Junio 2003, pp.47-56. |
| [Madhavji et. al , 1991] Madhavji. Nazim H. Schäfer Wilhelm. “Prism – Methodology and ProcessOriented Environment”. IEEE Transactions on Software Engineering, Vol 17. Nro 12 December 1991.  [Madhavji, 1991] Madhavji. Nazim H. “The Prism Model of Changes”. in Proc. 13th Int. Conf. on Software Pro. Los Alamitos, CA: IEEE Computer Soc. Press, 1991, pp 166-177.  [Madhavji, 1992] Madhavji. Nazim H. “Environment Evolution: The Prism Model of Changes”. IEEE Transactions on Software Engineering, Vol 18. Nro 5 May 1992.  [McCracken 82] McCracken, D.D., Jackson, M.A., “Life Cycle Concept Considered Harmful”, Software Engineering Notes, ACM, Abril 1982, pp.29-32. |
| [Mills 80] Mills, H.D., O’Neill, D., et al., “The management of software engineering”, IBM Systems Journal, Vol.24, Nº2, 1980, pp.414-477. |
| [Mills 87] Mills, H.D., Dyer, M., Linger, R.C., “Cleanroom Software Engineering”, IEEE Software, Vol.4, Nº5, Septiembre 1987, pp.19-25. |
| [Poppendieck 03] Poppendieck M., Poppendieck T. Lean Software Development: An Agile Toolkit for Software Development Managers. Addison Wesley. 2003. |
| [Royce 70] Royce, W.W., “Managing the Development of Large Software Systems: concepts and techniques”, IEEE WESCON, Los Angeles, California, Agosto 1970. |
| [Schwaber 01] Schwaber K., Beedle M., Martin R.C. Agile Software Development with SCRUM. Prentice Hall. 2001. |
| [Spivey 89] Spivey, J., “The Z Notation: A Reference Manual”, Prentice-Hall, Londres, 1989. |
| [Stapleton 87] Stapleton J. Dsdm Dynamic Systems Development Method: The Method in Practice. Addison-Wesley. 1997. |
| [Vienneau 93] Vienneau, R., “A Review of Formal Methods”, Kaman Science Corporation, 1993, pp.3-15 y 27-33. Reimpreso en “Software Requirements Engineering”, editores R.H. Thayer y M. Dorfman, IEEE Computer Society Press, 2º edición, Los Alamitos, CA, 1997, pp.324- 335. |
| [Wieringa, 1995] Wieringa, Roel. “An Introduction to Requirements Traceability”- Technical report, Faculty of Mathematics and Computer Science, Vrije Universiteit, Esprit Project 2RARE, November 1 (1995).  [Wordsworth 96] Wordsworth, J., “Software Engineering with B”, Wokinghman: Addison-Wesley, 1996. |

1. *Visibilidad del proceso*: capacidad de poder detectar los avances en el desarrollo. [↑](#footnote-ref-2)
2. *Robustez*: capacidad de un sistema para adaptarse al cambio. [↑](#footnote-ref-3)
3. COTS: Commercial Off-The-Shelf, producto de software comercial preprogramado parametrizable, usado para dominios específicos. [↑](#footnote-ref-4)