Procesos de Software

Modelos de Procesos
Unidad 1

Modelo en Cascada

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, primera figura) 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) (segunda figura). Pero la realidad es otra y concibe que el mismo modelo pueda tener retroalimentación. Pero finalmente propone agregar varias actividades antes de analizar o especificar los requisitos del software (tercera figura).

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.

Secuencia de la propuesta de Royce

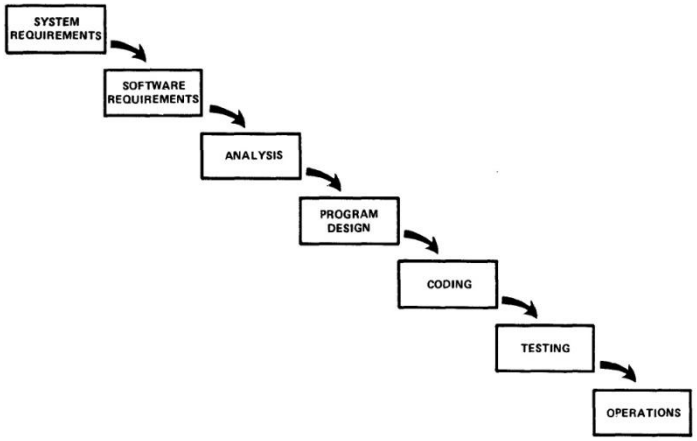


Figure 2. Implementation steps to develop a large computer program for delivery to a customer.

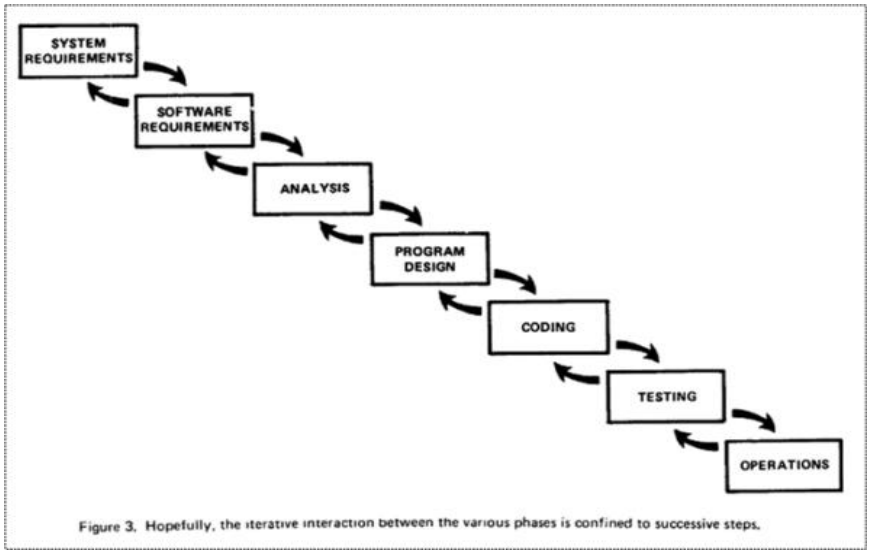
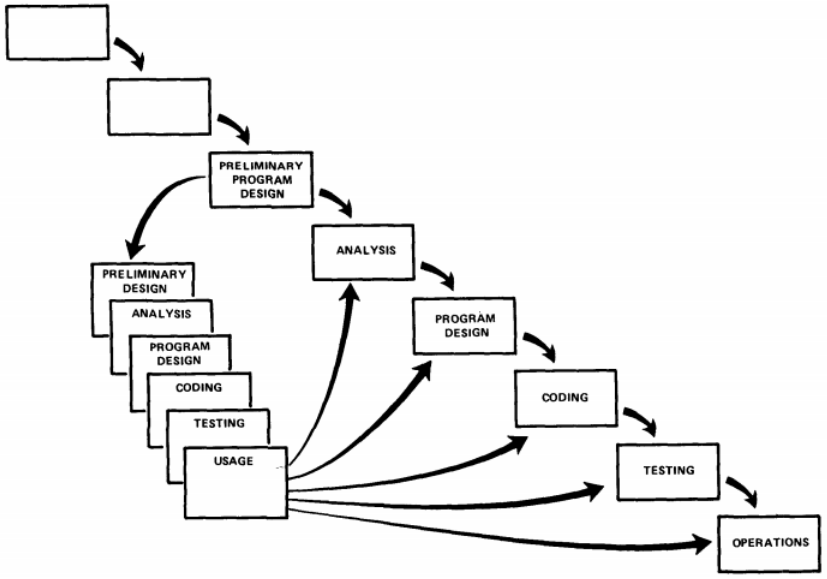


Figure 3. Hopefully, the iterative interaction between the various phases is confined to successive steps.



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 y Desventajas

- *Ventajas:*
- Fácil de administrar. El modelo da una buena visibilidad 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, 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 *Visibilidad del proceso*: capacidad de poder detectar los avances en el desarrollo.
- *Robustez*: capacidad de un sistema para adaptarse al cambio.

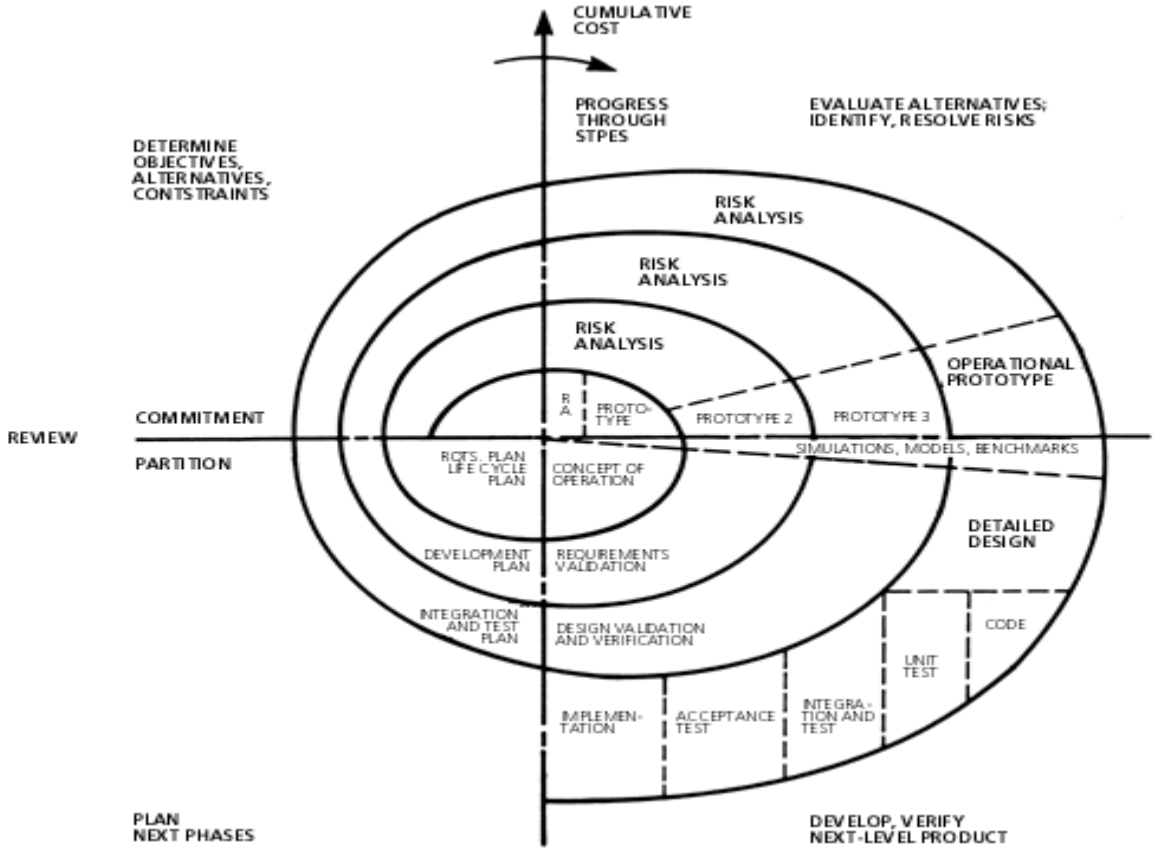
Modelo en Espiral

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.


CUMULATIVE COST
PROGRESS THROUGH STPES
EVALUATE ALTERNATIVES; IDENTIFY, RESOLVE RISKS
DETERMINE OBJECTIVES, ALTERNATIVES, CONTSTRAINTS
RISK ANALYSIS
RISK ANALYSIS
RISK ANALYSIS
OPERATIONAL PROTOTYPE
R A
PROTO-TYPE
PROTOTYPE 2
PROTOTYPE 3
COMMITMENT
REVIEW
PARTITION
SIMULATIONS, MODELS, BENCHMARKS
ROTS, PLAN LIFE CYCLE PLAN
CONCEPT OF OPERATION
DETAILED DESIGN
DEVELOPMENT PLAN
REQUIREMENTS VALIDATION
INTEGRATION AND TEST PLAN
DESIGN VALIDATION AND VERIFICATION
CODE
UNIT TEST
IMPLEMEN-TATION
ACCEPTANCE TEST
INTEGRA-TION AND TEST
PLAN NEXT PHASES
DEVELOP, VERIFY NEXT-LEVEL PRODUCT

Ventajas:

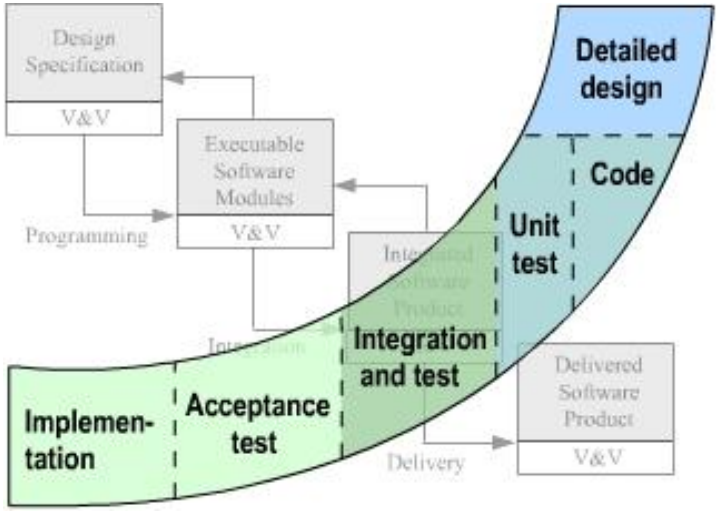
- Incorpora muchas de las ventajas de los otros ciclos de vida
- Conjuga la naturaleza iterativa de los prototipos con los aspectos controlados y sistemáticos del modelo clásico
- El modelo en espiral puede adaptarse y aplicarse a lo largo de la vida del software de computadora.
- Como el software evoluciona 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

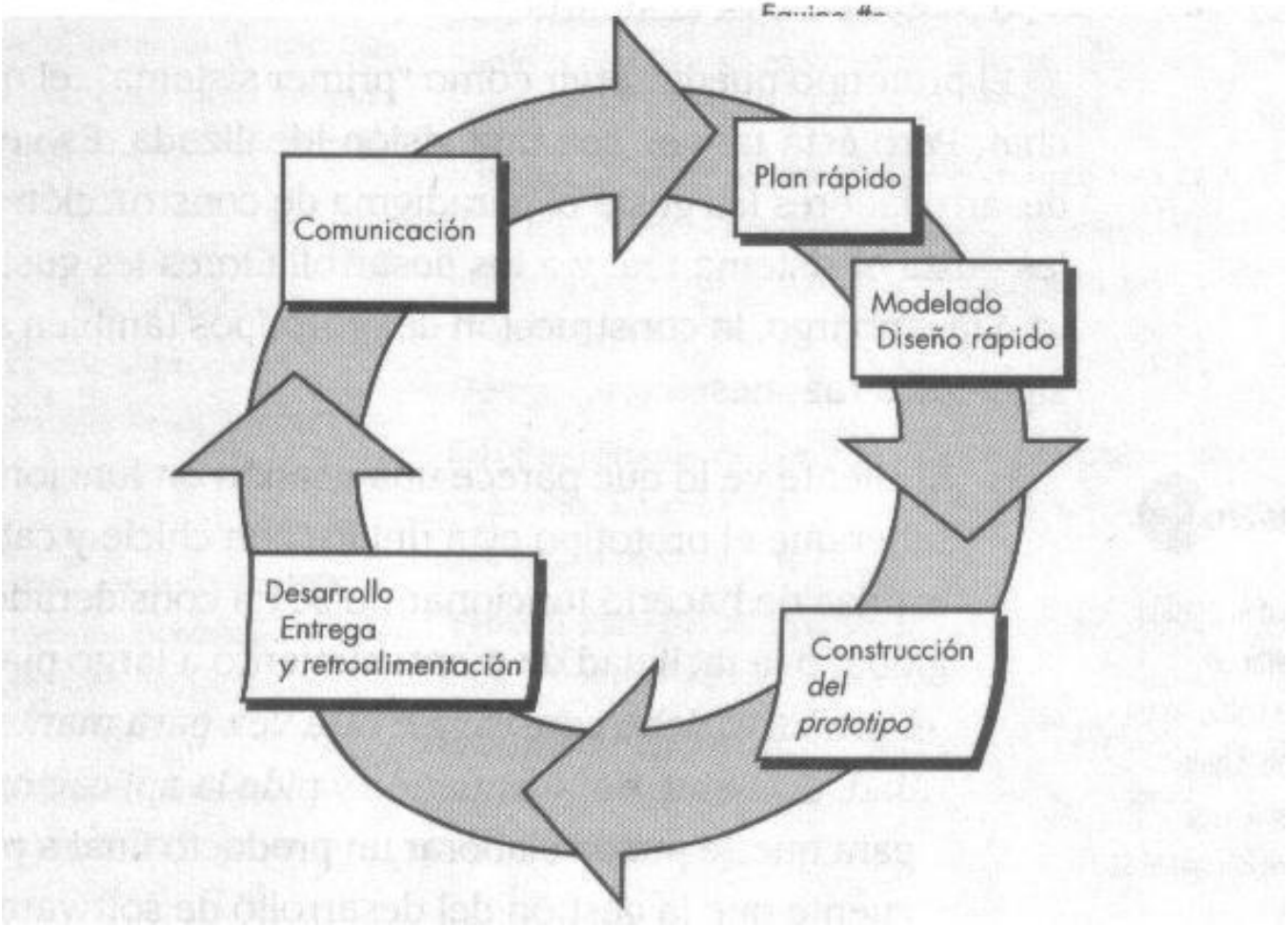
Relación entre el Cascada y el Espiral

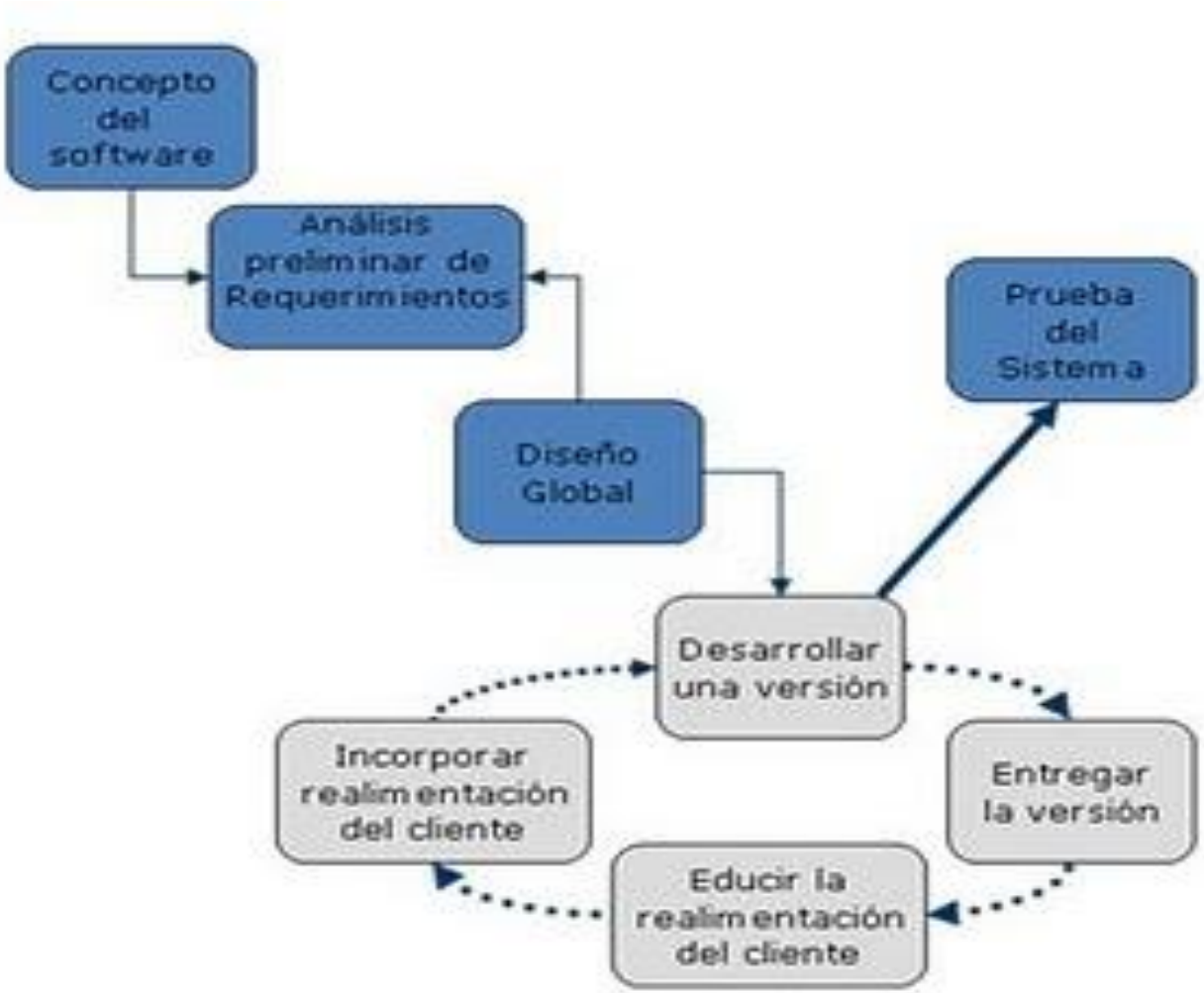
Waterfall Model	Spiral Model
Design Specifications	Detailed design
Programming	Code, Unit test
Integration	Integration and test
Delivery	Acceptance test, Implementation



Prototipado

- 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:
- 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.
- 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.


Comunicación
Plan rápido
Modelado
Diseño rápido
Construcción
del
prototipo
Desarrollo
Entrega
y retroalimentación


Concepto del software
Análisis preliminar de Requerimientos
Diseño Global
Desarrollar una versión
Prueba del Sistema
Incorporar realimentación del cliente
Entregar la versión
Educir la realimentación del cliente

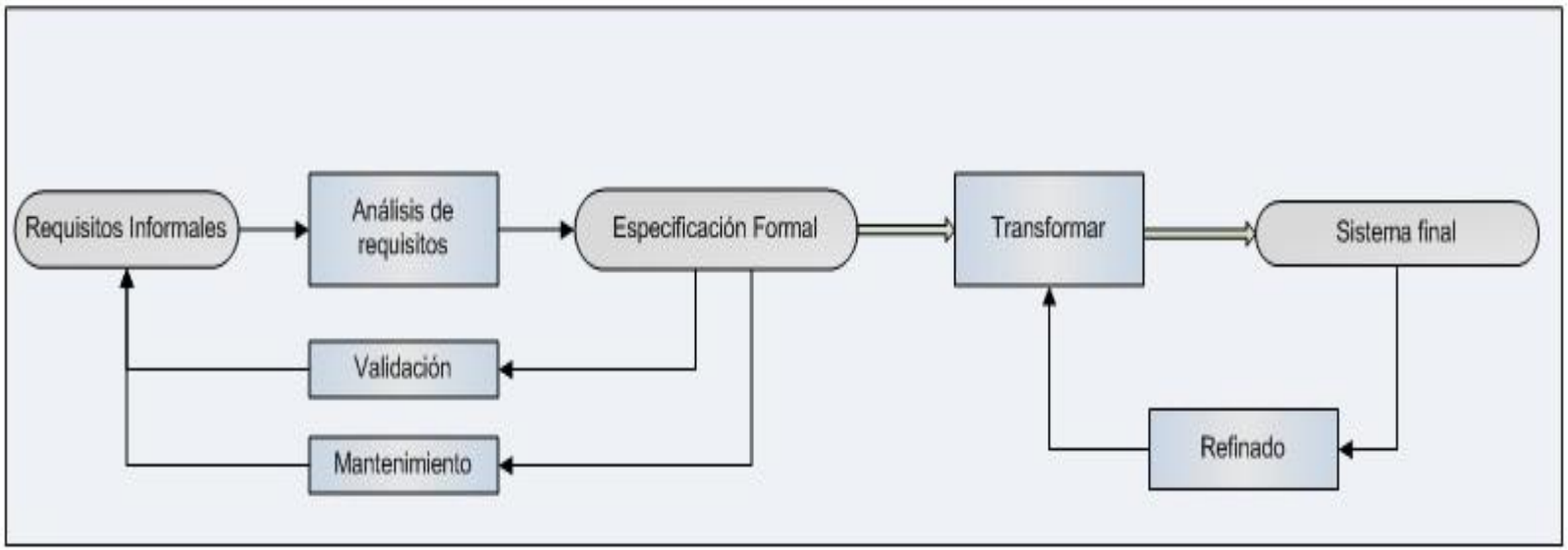
MODELO DE TRANSFORMACIÓN

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.


Requisitos Informales
Análisis de requisitos
Especificación Formal
Transformar
Sistema final
Validación
Mantenimiento
Refinado

Ventajas y Desventajas

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:

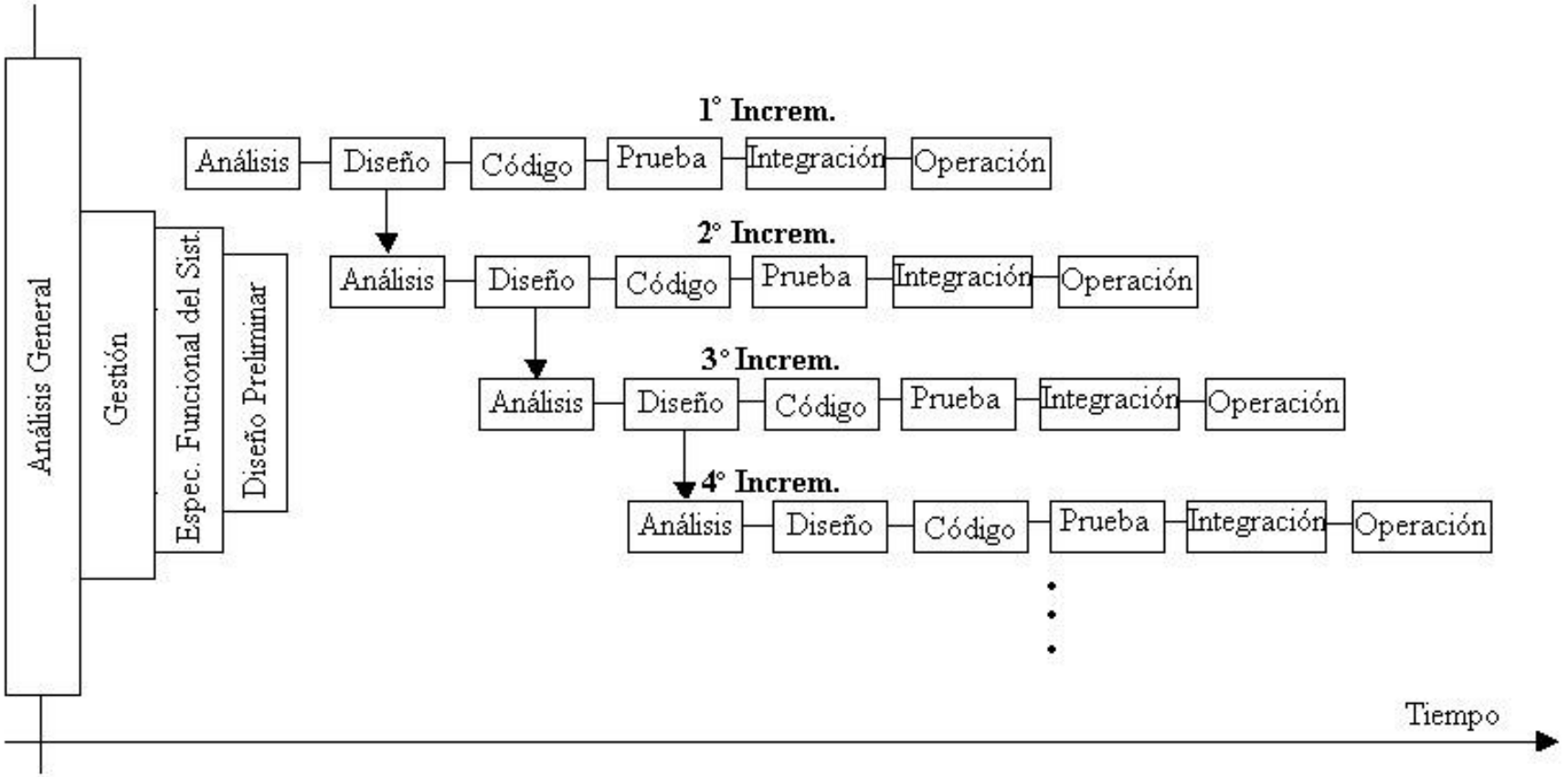
- · Se requiere mucho tiempo y esfuerzo para construir una especificación formal con precisión. Requiere tecnología específica y personal altamente especializada.
- · Dificultad de validación, pues las especificaciones formales son difíciles de entender por los clientes y usuarios.
- · Es difícil la 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

- 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


Análisis General
Gestión
Espec. Funcional del Sist.
Diseño Preliminar
1° Increm.
Análisis
Diseño
Código
Prueba
Integración
Operación
2° Increm.
Análisis
Diseño
Código
Prueba
Integración
Operación
3° Increm.
Análisis
Diseño
Código
Prueba
Integración
Operación
4° Increm.
Análisis
Diseño
Código
Prueba
Integración
Operación
Tiempo

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.

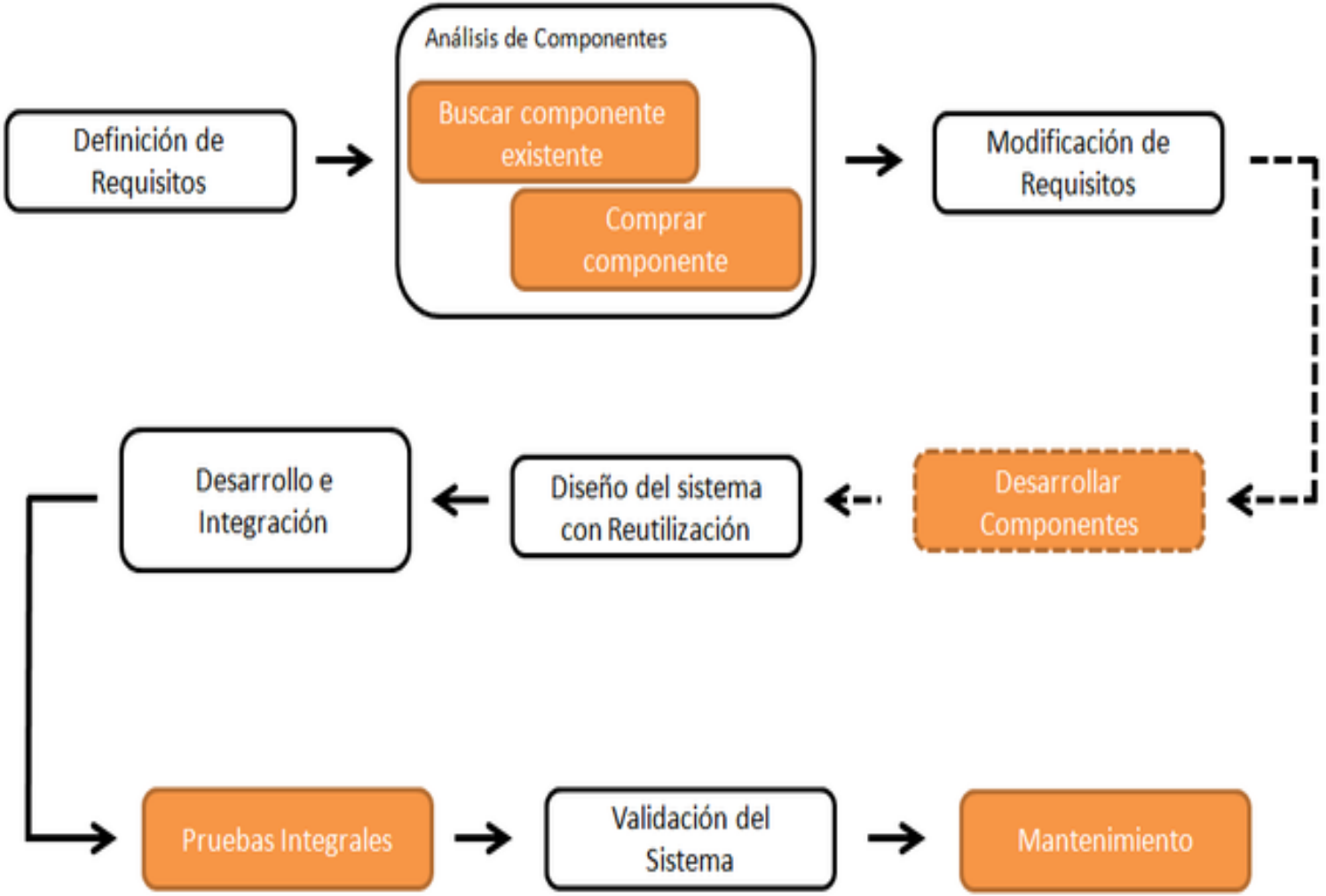
En la mayoría de los proyectos de software, existe algo de reutilización de software cuando se conocen diseños o códigos similares al requerido para el proyecto. Esta reutilización informal es independiente del proceso de desarrollo. Pero ha surgido un enfoque de desarrollo que se basa casi exclusivamente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

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.

COTS: Commercial Off-The-Shelf, producto de software comercial preprogramado parametrizable, usado para dominios específicos.


Reutilización de Software
Desarrollo de Software Basado en Líneas de Productos
Desarrollo de Software Basado en Componentes
Ingeniería de Dominio
Ingeniería de Aplicaciones
Desarrollo de Software para reutilización
Desarrollo de Software con reutilización

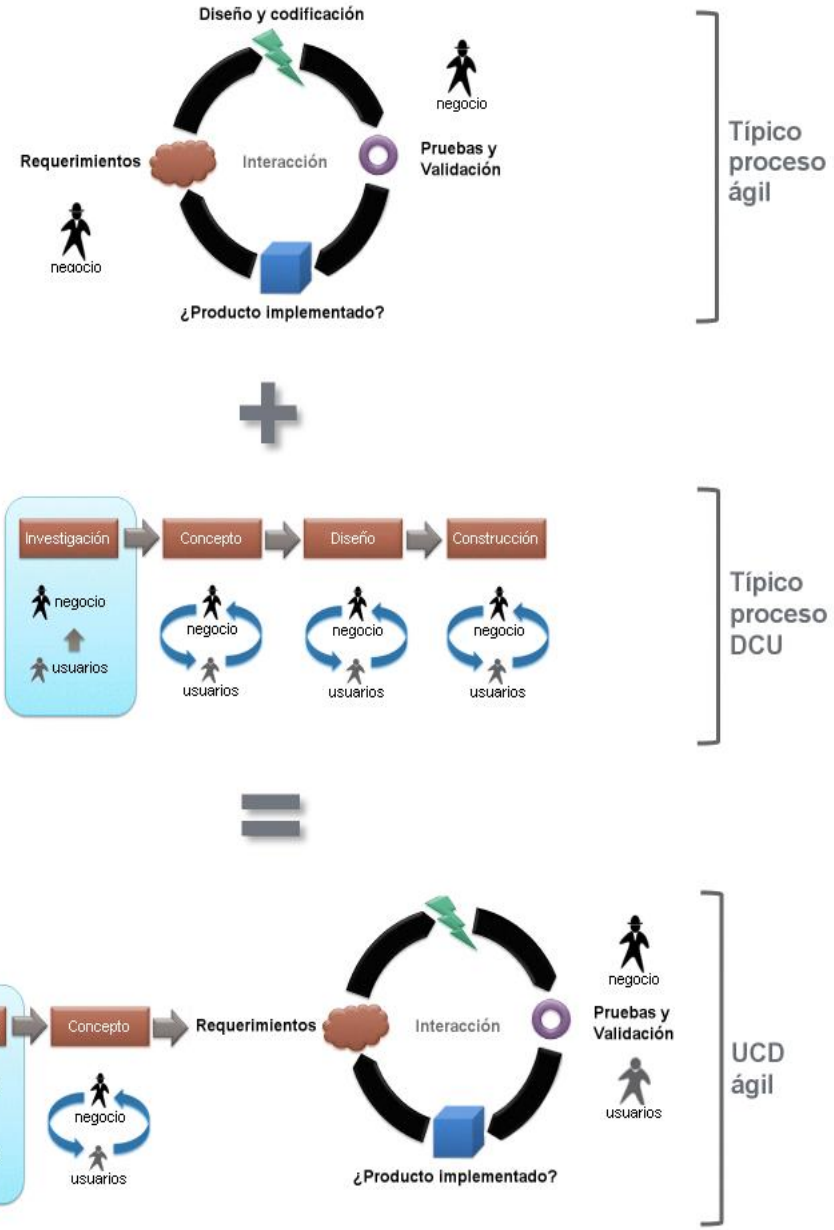

Definición de Requisitos
Análisis de Componentes
Buscar componente existente
Comprar componente
Modificación de Requisitos
Desarrollar Componentes
Diseño del sistema con Reutilización
Desarrollo e Integración
Pruebas Integrales
Validación del Sistema
Mantenimiento

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.


Diseño y codificación
negocio
Requerimientos
Interacción
Pruebas y Validación
negocio
¿Producto implementado?
Típico proceso ágil
Investigación
Concepto
Diseño
Construcción
negocio
usuarios
negocio
usuarios
negocio
usuarios
negocio
usuarios
Típico proceso DCU
Investigación
Concepto
Requerimientos
Interacción
negocio
Pruebas y Validación
negocio
usuarios
negocio
usuarios
usuarios
¿Producto implementado?
UCD ágil

Orientado a Aspectos

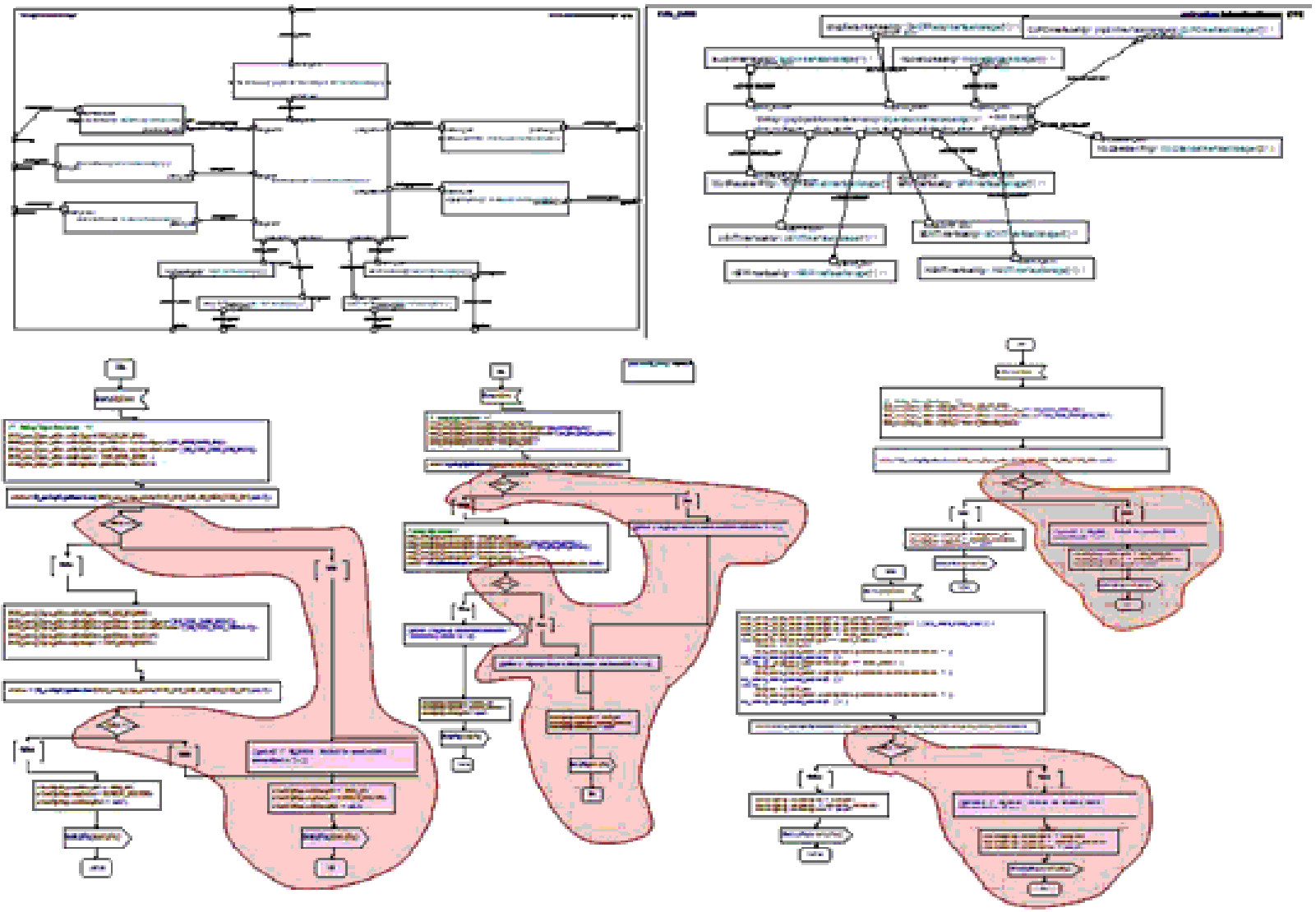
- Los conceptos de orientación a aspectos nacieron en 1996, cuando Gregor Kiczales propuso una nueva idea: La Programación Orientada a Aspectos (POA) o Aspect Oriented Programming en su denominación en inglés. A partir de entonces, la POA y otras técnicas y tecnologías, que se centraban en la modularización 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.



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].

En “Lectura secundaria” podrá encontrar más material para profundizar en estos modelos y otros.

Debe realizar los TPs indicados en esta Unidad.

Muchas gracias!

Recuerde que los docentes estarán conectados para responder dudas los jueves de 19 a 22hs. en el chat de Miel.