

Apuntes del curso de
**Introducción a la Ingeniería
de Software**

Facundo Spira

Índice

1. Introducción	9
1.1. Ingeniería de Software	9
1.2. Importancia de la Ingeniería de Software	9
1.3. Actividades del Proceso de Software	9
1.4. Problemas generales que afectan el software	9
1.5. Diversidad en ingeniería de software	10
1.6. Tipos de aplicaciones	10
1.7. Ética en la ingeniería de software	10
2. Procesos de Software	11
2.1. Descripción del proceso de software	11
2.2. Procesos basados en planes y procesos ágiles	11
2.3. Modelos de procesos	11
2.3.1. Modelo en cascada	11
2.3.2. Desarrollo incremental	12
2.3.3. Integración y Configuración	13
2.4. Actividades del Proceso de Desarrollo de Software	13
2.4.1. Especificación de software	13
2.4.2. Diseño e implementación de software	14
2.4.3. Validación de software	15
2.4.4. Evolución de software	15
2.5. Hacer frente al cambio	16
2.5.1. Reducir costos del retrabajo	16
2.5.2. Prototipado de software	16
2.5.3. Entregas incrementales	17
2.6. Mejora de procesos	17
2.6.1. Enfoques para la mejora	18
2.6.2. Ciclo de mejora de procesos	18
2.6.3. Actividades de la mejora de procesos	18
2.6.4. Medición del proceso	18
3. Desarrollo de Software Ágil	19
3.1. Características	19
3.2. Manifiesto ágil	19
3.3. Principios de los métodos ágiles	19
3.4. Aplicabilidad de las metodologías ágiles	19
3.5. Técnicas de desarrollo ágil	20
3.5.1. Extreme Programming (XP)	20
3.5.2. Prácticas de XP	20
3.5.3. Historias de usuario	21
3.5.4. Refactorización	21
3.5.5. Desarrollo dirigido por pruebas	21
3.5.6. Programación por pares	22
3.6. Gestión de proyectos ágiles	23
3.6.1. SCRUM	23
3.6.2. Elementos de SCRUM	23
3.6.3. Ciclos de sprints en SCRUM	24
3.6.4. Beneficios de SCRUM	24
3.7. Escalando métodos ágiles	24
3.7.1. Problemas al escalar	25
3.7.2. Problemas con el mantenimiento ágil	25
3.7.3. Enfoques dirigidos por planes y ágiles	25

3.7.4. Métodos ágiles para sistemas grandes	26
3.7.5. IBM Agile Scaling Model (ASM)	26
3.7.6. Multi-team Scrum	26
3.8. MUM (Modularizado, Unificado y Medible)	27
3.8.1. Principales características	27
3.8.2. Beneficios del enfoque iterativo	27
3.8.3. Dimensión del tiempo	27
3.8.4. Dimensión del Modelo	28
3.8.5. Disciplinas	28
3.8.6. Otros roles	29
3.8.7. Roles y combinación	30
4. Ingeniería de Requisitos	31
4.1. Abstracción de requisitos	31
4.1.1. Tipos de requisitos	31
4.1.2. Tipos de información de requisitos	31
4.2. Requisitos funcionales y no funcionales	32
4.2.1. Requisitos funcionales	32
4.2.2. Requisitos no funcionales	33
4.3. Problemas en los requisitos	33
4.4. Procesos de la ingeniería de requisitos	34
4.5. Obtención de requisitos	34
4.5.1. Dificultades comunes en el proceso	34
4.5.2. Actividades del proceso	34
4.6. Técnicas de obtención de requisitos	35
4.6.1. Entrevistas	35
4.6.2. Investigar antecedentes	36
4.6.3. Workshops	36
4.6.4. Focus groups	36
4.6.5. Prototipado	36
4.6.6. Observaciones - Etnografías	38
4.6.7. Cuestionarios / Encuestas	38
4.6.8. Análisis de las interfaces del sistema	38
4.6.9. Análisis de la interfaz de usuario	38
4.6.10. Análisis de documentación	39
4.6.11. Tormenta de ideas	39
4.6.12. Historias y escenarios	39
4.6.13. Historias de usuario	39
4.6.14. Modelado de Procesos	39
4.6.15. Casos de uso	39
4.7. Selección de técnicas a usar	42
4.8. Análisis de requisitos	42
4.9. Especificación de requisitos	42
4.9.1. Formas de escribir una especificación de requisitos	42
4.9.2. Características deseables según Wiegers	44
4.9.3. Guía para la escritura de requisitos	44
4.10. Ingeniería de Requisitos en Procesos Ágiles	45
4.10.1. Historias de Usuario (HU)	45
4.10.2. Redacción de HU (Cards)	45
4.10.3. Redacción de HU (Confirmación)	46
4.11. Modelado del sistema	47
4.11.1. Perspectiva del sistema	47
4.11.2. Modelos de contexto	47
4.11.3. Modelos de interacción	47

4.11.4. Modelos estructurales	48
4.11.5. Modelos de comportamiento	48
4.11.6. Ingeniería dirigida por modelos (MDE)	49
4.11.7. Otros modelos para requisitos - Wiegers	49
4.12. Documento de requisitos de software	52
4.12.1. Usuarios de un documento de requisitos	52
4.12.2. Estructura de un documento de requisitos	52
4.13. Priorización de requisitos	53
4.14. Validación de requisitos	53
4.14.1. Proceso	54
4.14.2. Chequeo de requisitos	54
4.14.3. Validación de requisitos no funcionales	54
4.14.4. Técnicas de validación de requisitos	54
4.14.5. ¿Cómo validar requisitos?	54
4.14.6. Validar utilizando criterios de aceptación	55
4.15. Gestión de requisitos	55
4.15.1. Cambios en los requisitos	56
4.15.2. Planificación de la gestión de requisitos	56
4.15.3. Gestión de cambios en los requisitos	56
4.15.4. Cambios de requisitos en ágiles	56
5. Diseño de Software	57
5.1. Principios de Diseño	57
5.1.1. Abstracción	57
5.1.2. Acoplamiento y Cohesión	58
5.1.3. Descomposición y modularización	58
5.1.4. Encapsulación y ocultamiento de información	58
5.1.5. Separación de la interfaz y la implementación	58
5.1.6. Suficiencia, integridad y primitivismo	58
5.1.7. Separación de intereses	58
5.1.8. Dividir y conquistar	58
5.1.9. Reuso	58
5.2. Problema malvado	59
5.3. Arquitectura de Software	60
5.3.1. Diseño arquitectónico	60
5.3.2. Ventajas de explicitar la arquitectura	60
5.3.3. Decisiones en el diseño de arquitectura	60
5.3.4. Reutilización de arquitectura	61
5.3.5. ¿Qué afecta y determina la arquitectura?	61
5.3.6. Conflictos entre soluciones	61
5.4. Patrones arquitectónicos	61
5.4.1. Arquitectura en capas	62
5.4.2. Arquitectura de repositorio	62
5.4.3. Arquitectura cliente servidor	63
5.4.4. Arquitectura de tubería y filtro	63
5.5. Arquitecturas de aplicación	64
5.5.1. Sistemas de procesamiento de transacciones	64
5.5.2. Sistemas de información	64
5.5.3. Sistemas de procesamiento de lenguajes	64
5.6. Vistas arquitectónicas	65
5.6.1. Modelo 4+1	65
5.6.2. Tipos de vistas	65
5.6.3. Diagrama de componentes	65
5.6.4. Diagrama de despliegue	66

5.7.	Ingeniería de Software basada en componentes	67
5.7.1.	Pilares de CBSE	67
5.7.2.	CBSE y principios de diseño	67
5.7.3.	Estándares de componentes	67
5.7.4.	Componentes	67
5.7.5.	Interfaces de los componentes	67
5.7.6.	Acceso y modelos	68
5.7.7.	Elementos de un modelo	68
5.7.8.	Soporte del middleware	68
5.7.9.	Desarrollo para el reuso	68
5.7.10.	Composición de componentes	69
5.8.	Ingeniería de software distribuido	70
5.8.1.	Aspectos de diseño	70
5.8.2.	Modelos de interacción	71
5.8.3.	Middleware	71
5.8.4.	Computación cliente-servidor	71
5.8.5.	Patrones arquitectónicos para sistemas distribuidos	72
5.8.6.	Software como Servicio (SaaS)	73
6.	Construcción de Software	74
6.1.	Fundamentos de la Construcción de Software	74
6.2.	Gestión de la Construcción	74
6.3.	Consideraciones prácticas	74
6.4.	Tecnologías de Construcción	75
6.5.	Herramientas de Construcción	77
6.6.	Estándares de Programación	77
6.7.	Reutilización de código	78
6.8.	Documentación del código	78
6.9.	Programación en Pares	78
6.10.	Debug	78
7.	Verificación y Validación	79
7.1.	Terminología e Introducción	79
7.1.1.	Definición de Verificación y Validación	79
7.1.2.	Principios de la verificación y validación	80
7.1.3.	Algunos defectos	80
7.1.4.	Clasificación de defectos	80
7.2.	Verificación y Validación en el ciclo de vida del Software	81
7.2.1.	Proceso en V	81
7.2.2.	Niveles de prueba	81
7.3.	Pruebas de componentes	82
7.3.1.	Estrategias de pruebas	82
7.3.2.	Módulos y componentes	82
7.3.3.	Drivers y Stubs	82
7.4.	Pruebas de integración	83
7.4.1.	Estrategia de pruebas	83
7.5.	Pruebas de sistema	84
7.6.	Pruebas de aceptación	84
7.6.1.	Tipos de pruebas de aceptación	84
7.7.	Pruebas de nuevas versiones del sistema	85
7.8.	Pruebas de regresión	85
7.9.	Tipos de pruebas y técnicas de generación de casos de prueba	85
7.9.1.	Tipos genéricos de pruebas	85
7.9.2.	Técnicas de verificación	85

7.10. Pruebas estáticas	85
7.10.1. Análisis de código	85
7.10.2. Análisis estático	87
7.11. Pruebas dinámicas	87
7.11.1. Caja negra vs Caja blanca	87
7.11.2. Técnicas de caja negra	88
7.11.3. Caja negra - Particiones en clases de equivalencia	88
7.11.4. Caja negra - Análisis de valores límites	88
7.11.5. Caja negra - Pruebas de transición de estados	89
7.11.6. Caja negra - Pruebas basadas en la lógica	89
7.11.7. Caja negra - Pruebas basadas en casos de uso	89
7.11.8. Técnicas de caja blanca	90
7.11.9. Técnicas basadas en la experiencia	91
7.12. Comparación de técnicas de pruebas	92
7.12.1. Proceso para un módulo	92
7.12.2. Comparación de técnicas	92
7.13. Pruebas no funcionales	92
7.13.1. Tipos de pruebas no funcionales	93
7.14. Gestión de las pruebas (Test management)	93
7.14.1. Roles en las pruebas	93
7.14.2. Planificar la verificación	94
8. Liberación de software	95
8.1. Instalación y configuración	95
8.2. Adopción (o conversión)	95
8.3. Entrenamiento y apoyo en el uso	96
8.3.1. Entrenamiento para cubrir perfiles y necesidades	96
8.3.2. Soporte para el entrenamiento	96
8.3.3. Revisión del entrenamiento	96
8.3.4. Apoyo durante el uso del sistema	97
9. Evolución de Software	98
9.1. Modelos de evolución	98
9.1.1. Modelo en espiral de desarrollo y evolución	98
9.1.2. Modelo de Evolución y Servicio	98
9.2. Procesos de Evolución	99
9.2.1. Cambios urgentes	100
9.3. Evolución en métodos ágiles	100
9.3.1. Escenarios de transferencia	100
9.4. Sistemas heredados	100
9.4.1. Gestión de sistemas heredados	101
9.5. Evaluación del valor para el negocio	102
9.5.1. Evaluación de la calidad del sistema	102
9.6. Mantenimiento de software	102
9.6.1. Tipos de mantenimiento	102
9.6.2. Costos de mantenimiento	103
9.6.3. Predicción de mantenimiento	103
9.6.4. Métricas de complejidad	104
9.6.5. Métricas de proceso	104
9.7. Reingeniería de software	104
9.7.1. Ventajas de la reingeniería	104
9.7.2. Proceso de reingeniería	104
9.7.3. Enfoques de la reingeniería	105
9.8. Refactorización	105

9.8.1. Refactorización y reingeniería	105
9.9. Bad Smells	105
10. Gestión de Proyectos	107
10.1. Introducción	107
10.2. Director de proyecto	107
10.3. Ciclo de vida de un proyecto	108
10.4. Planificación	108
10.5. Planificación - Alcance	108
10.5.1. EDT/WBS	109
10.6. Planificación - Actividades	109
10.7. Planificación - Estimaciones	109
10.7.1. Principios de estimación	110
10.7.2. Factores que influyen en las estimaciones	110
10.7.3. Estimación del tamaño del producto	110
10.7.4. Técnicas de estimación	111
10.7.5. Técnicas - Analogía	111
10.7.6. Técnicas - Juicio de expertos	112
10.7.7. Técnicas - Delphi	112
10.7.8. Técnicas - Algoritmos	113
10.7.9. Técnicas - COCOMO II	113
10.8. Cronograma	113
10.8.1. Grafo de precedencias - Camino crítico	113
10.9. Cronograma	114
10.9.1. Técnicas para comprimir	114
10.10. Riesgos	115
10.10.1. Categorización	115
10.10.2. Identificación de riesgos	115
10.10.3. Análisis cualitativo de riesgos	116
10.10.4. Planificar la respuesta a los riesgos	117
10.10.5. Monitoreo de riesgos	118
10.11. Seguimiento	118
10.11.1. Enfoque de valor ganado	119
10.12. Equipo	120
10.12.1. Motivación	120
10.12.2. Comunicación	121
10.12.3. Etapas del desarrollo de un equipo	122
10.12.4. Gestión de conflictos	122
10.13. Desarrollo ágil	123
10.14. Desarrollo ágil - SCRUM	123
11. Gestión de la configuración de Software	125
11.1. Actividades de la gestión de la configuración	125
11.2. Glosario de términos	125
11.3. Gestión de versiones	126
11.3.1. Control de versiones	126
11.3.2. Gestión de versiones	126
11.4. Armado del sistema	127
11.4.1. Integración continua	127
11.4.2. Puntos claves	127
11.5. Gestión de cambios	128
11.5.1. Proceso de gestión de cambios	128
11.5.2. Gestión de cambios y metodologías ágiles	129
11.6. Gestión de liberaciones	129

11.6.1. Consideraciones al planificar una liberación	129
11.6.2. Que puede incluir una versión a liberar?	129
11.6.3. Otras consideraciones	129
11.6.4. Software as Service (SaaS)	129
11.6.5. Puntos clave	130
11.7. Identificación, Alcance, Formalismo y Ambientes	130
11.7.1. Identificación	130
11.7.2. Alcance y formalismo	130
11.7.3. Gestión de ambientes	130

1. Introducción

Atributos esenciales de un buen software:

- Mantenibilidad
Debe poder evolucionar para satisfacer las necesidades cambiantes de los clientes.
- Confiabilidad y seguridad
No debe causar daños físicos o económicos en caso de falla del sistema. Los usuarios maliciosos no deberían poder acceder o dañar el sistema.
- Eficiencia
No debe hacer un derroche de recursos del sistema, como la memoria y los ciclos del procesador, incluye la capacidad de respuesta, el tiempo de procesamiento, etc.
- Aceptabilidad
Debe ser aceptable para el tipo de usuarios para los que está diseñado.

1.1. Ingeniería de Software

Es una disciplina de la ingeniería que se ocupa de todos los aspectos de la producción de software desde las primeras etapas de la especificación del sistema hasta el mantenimiento del sistema una vez que se ha puesto en uso.

Es una ingeniería ya que utiliza teorías y métodos teniendo en cuenta las limitaciones organizacionales y financieras.

Los aspectos de la producción de software hace referencia a la gestión de proyectos y desarrollo de herramientas, métodos, etc. Involucra compromisos.

1.2. Importancia de la Ingeniería de Software

- Confianza
Los individuos y la sociedad confían en los sistemas avanzados de software.
- Costos
A largo plazo, es más barato utilizar métodos y técnicas de ingeniería de software. Para la mayoría de los tipos de sistema, la mayoría de los costos consisten en cambiar el software después de ponerlo en producción.

1.3. Actividades del Proceso de Software

- Especificación del software
Los clientes y los ingenieros definen el software que se va a producir y las limitaciones de su funcionamiento.
- Desarrollo de software
Se diseña y desarrolla el software.
- Validación de software
Se verifica el software para garantizar que es lo que requiere el cliente.
- Evolución del software
Se modifica el software para reflejar los requisitos cambiantes de los clientes y del mercado.

1.4. Problemas generales que afectan el software

- Heterogeneidad
Los sistemas de software corren en distintos tipos de computadoras, dispositivos móviles, etc.

- Negocios y cambio social
Los negocios y la sociedad están cambiando increíblemente rápido.
- Seguridad y confianza
Es esencial que podamos confiar en el software que usamos.
- Escala
Amplia gama de escalas, desde sistemas embebidos en dispositivos portátiles hasta sistemas en la nube que sirven a una comunidad global.

1.5. Diversidad en ingeniería de software

- La ingeniería de software es un enfoque sistemático para la producción de software que tiene en cuenta los costos prácticos, el cronograma y los problemas de confiabilidad, así como las necesidades de los clientes y productos de software.
- Uno de los factores más importantes es el tipo de aplicación.
- Aunque los límites de los tipos son borrosos ayuda a determinar las técnicas y métodos a utilizar.
- Algunos principios fundamentales se aplican a todos los tipos:
 - un proceso de desarrollo manejado y entendido.
 - confiabilidad y rendimiento
 - comprender y gestionar la especificación del software.
 - reutilizar el software que ya se ha desarrollado.

1.6. Tipos de aplicaciones

- Aplicaciones independientes: incluyen toda la funcionalidad necesaria y no necesitan una red.
- Aplicaciones interactivas basadas en transacciones: se ejecutan en una computadora remota.
- Sistemas de control integrados: administran dispositivos de hardware.
- Sistemas de procesamiento por lotes: diseñados para procesar datos en grandes lotes.
- Sistemas de entretenimiento
- Sistemas para modelados y simulación: desarrollados para modelar procesos o situaciones físicas.
- Sistemas de recolección de datos: recopilan datos de su entorno utilizando sensores para su procesamiento.
- Sistemas de sistemas: están compuestos de otros sistemas de software.

1.7. Ética en la ingeniería de software

En la ingeniería de software implica responsabilidades más amplias que simplemente la aplicación de habilidades técnicas. Los ingenieros de software deben comportarse de una manera honesta y éticamente responsable. Algunas cuestiones de responsabilidad son:

- Confidencialidad: respetar la confidencialidad de sus empleadores.
- Competencia: no aceptar trabajos que están fuera de su competencia.
- Derechos de propiedad intelectual: conocer las leyes sobre el uso de la propiedad intelectual y licencias.
- Mal uso de la computadora: no usar sus habilidades técnicas para usar mal las computadoras de otras personas.

2. Procesos de Software

Existen diferentes modelos de proceso, pero todos ellos incluyen actividades de:

- Especificación: definir que es lo que el sistema debe hacer.
- Diseño e implementación: definir la organización del sistema e implementarlo.
- Validación: comprobar que el sistema construido es lo que el cliente necesita.
- Evolución: realizar cambios en el sistema como respuesta a los cambios en las necesidades del cliente.

2.1. Descripción del proceso de software

Cuando se describe un proceso, generalmente hablamos de actividades de dicho proceso y el orden en el cual se deberían realizar.

La descripción de un proceso además puede incluir:

- Productos, los cuales son producidos por las actividades.
- Roles, los cuales reflejan las responsabilidades de las personas involucradas en el proceso.
- Pre y Post condiciones, las cuales deben cumplirse antes y después de que una actividad se haya realizado o un producto se haya desarrollado.

2.2. Procesos basados en planes y procesos ágiles

Procesos dirigidos por planes

Son aquellos en los que todas las actividades del proceso son planificadas en detalle y el avance y el progreso se mide contra dicho plan.

Procesos ágiles

Aquellos en los que la planificación es incremental y es fácil de cambiar, a modo de reflejar los cambios en los requerimientos del cliente.

En la práctica la mayoría de los procesos incluyen elementos de ambos enfoques. No existen procesos correctos o incorrectos.

2.3. Modelos de procesos

Un modelo de proceso es una representación abstracta de un proceso, bajo un cierto punto de vista o perspectiva.

2.3.1. Modelo en cascada

Presenta el proceso de desarrollo de software como una secuencia de pasos, donde uno desemboca en otro. Debido a esto es que se conoce como modelo en cascada. Es un ejemplo de un proceso dirigido por planes, ya que en teoría se planifica todas las actividades del proceso antes de comenzar con el desarrollo.

Las etapas del modelo en cascada son las siguientes:

▪ Definición de requisitos

Se definen los objetivos, servicios y restricciones del sistema mediante consultas con usuarios del sistema. Luego se definen en detalle y se utiliza como especificación del sistema.

▪ Diseño del software del sistema

Se define una arquitectura del sistema y diseño del software.

- **Implementación y pruebas unitarias**

En esta etapa se desarrolla el software como un set de programas o unidades. El test unitario se encarga de verificar que cada unidad cumpla con su especificación.

- **Integración y prueba del sistema**

Se integran las unidades o programas implementados como un sistema completo. Se testeá que cumpla con todos los requisitos. Luego se entrega al cliente.

- **Operación y mantenimiento**

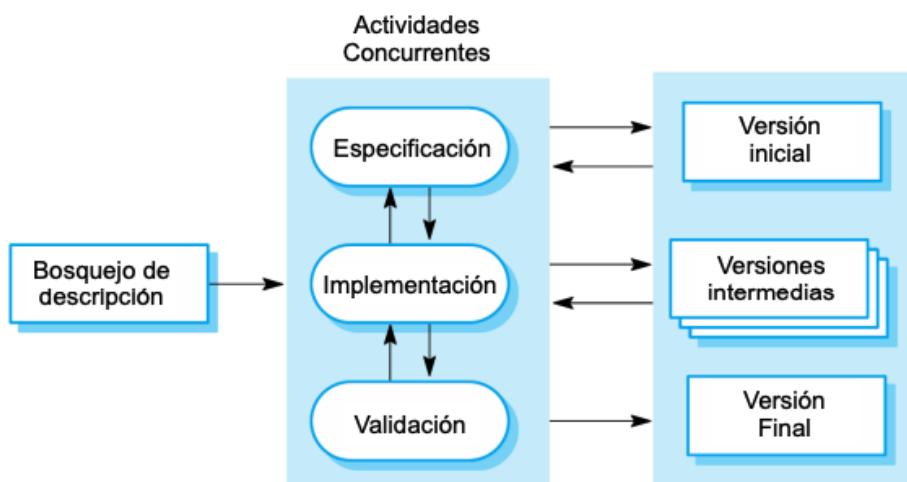
Se comienza a utilizar el sistema desarrollado. El mantenimiento involucra corregir errores que no fueron descubiertos en etapas tempranas, mejorar la implementación de unidades y mejorar los servicios del sistema a medida que se descubren nuevos requerimientos.

Problemas del modelo en cascada

- Dificultad en responder a los cambios de requerimientos del cliente. Este modelo sería apropiado únicamente si los requerimientos son bien claros y estables desde el inicio.
- Entrega tardía de valor para el cliente. Hay un alto riesgo de que el producto no cumpla con sus expectativas o se adapte a sus necesidades.

2.3.2. Desarrollo incremental

Se basa en la idea de desarrollar una implementación inicial, recibir feedback de usuarios y evolucionar el software a través de varias fases hasta que se cumplan los requisitos del sistema.



Beneficios

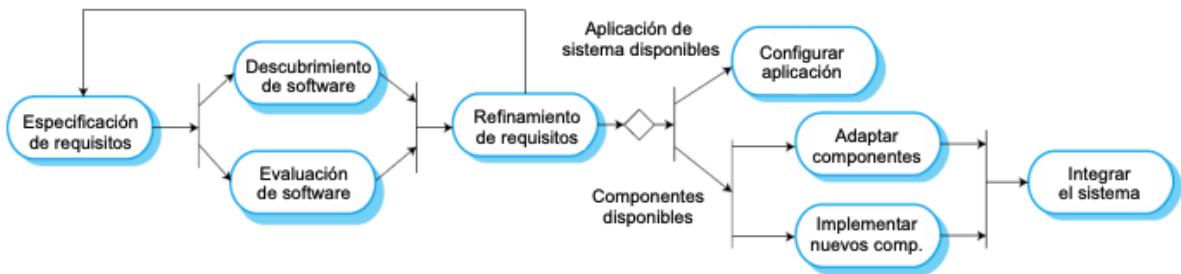
- Se reduce el costo de acomodar los cambios en los requisitos. El re-trabajo es mucho menor que en el modelo en cascada.
- Es más fácil obtener feedback del cliente sobre la parte del desarrollo terminada. El cliente puede ver el avance del proyecto y realizar comentarios en las demostraciones.
- Se pueden realizar entregas más rápidas de software que puede ser útil para el cliente. El cliente puede usar el software y ganar valor de forma más temprana que en el modelo en cascada.

Problemas

- Es difícil mantener la trazabilidad de documentos entre versiones. No es eficiente producir documentación que refleje cada versión del sistema.
- La estructura del sistema tiende a degradarse con cada incremento. Los cambios regulares tienen a corromper la estructura del sistema. Tiempo y dinero son invertidos en la refactorización del sistema. A medida que pasa el tiempo incorporar cambios se torna difícil y costoso.

2.3.3. Integración y Configuración

Se basa en la reutilización de componentes de software los cuales son integrados en el sistema a construir.



Tenemos las siguientes etapas:

- Especificación de los requisitos: se proponen los requisitos iniciales del sistema. No tienen porque ser muy detallados pero deberían incluir una breve descripción de los requisitos esenciales del sistema.
- Descubrimiento y evaluación de software: se realiza una búsqueda de componentes que provean las funcionalidades requeridas.
- Refinamiento de requisitos: se refinan los requisitos utilizando información sobre los componentes que se descubrieron.
- Configuración de la aplicación
- Adaptación de componentes e integración

Beneficios

- Se reducen costos y riesgos ya que menos software se desarrolla desde cero.
- Desarrollo y entrega del producto más rápida.

Problemas

- Puede que el sistema no cumpla con las necesidades reales del cliente, o que las cumpla parcialmente.
- Se pierde control sobre la evolución de los elementos reutilizados en el sistema, ya que son desarrollados por otros equipos externos.

2.4. Actividades del Proceso de Desarrollo de Software

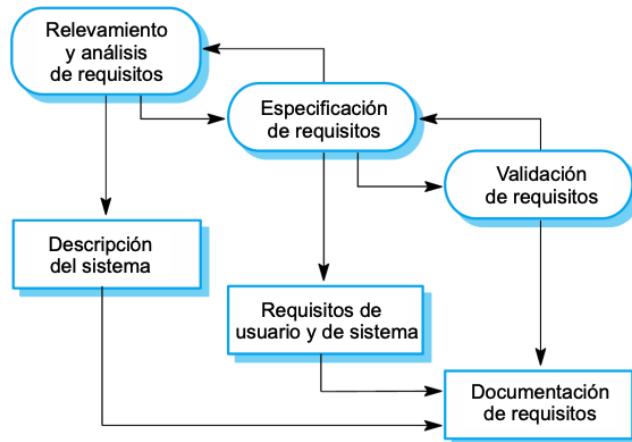
Las cuatro actividades de proceso básicas (especificación, desarrollo, validación y evolución) son organizadas de forma diferente en los diferentes procesos de desarrollo.

2.4.1. Especificación de software

Es el proceso de establecer qué servicios o funcionalidades son requeridos y las restricciones acerca de la operación y el desarrollo.

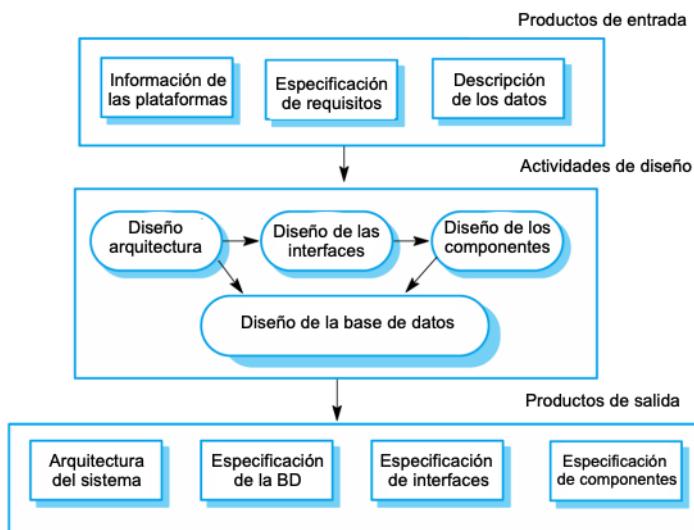
El proceso de ingeniería de requisitos intenta producir un documento que especifique el sistema, satisfaciendo todas las necesidades. Esto generalmente se presenta en dos niveles de detalle. Los usuarios finales y clientes necesitan una descripción general (high-level) de los requisitos, mientras que los desarrolladores necesitan una especificación más detallada. En este proceso podemos distinguir tres etapas:

- Relevamiento y análisis de requisitos: ¿Qué es lo que el cliente y los interesados requieren y esperan del sistema?
- Especificación de requisitos: definir los requisitos en detalle.
- Validación de requisitos: comprobar la validez de los requisitos.



2.4.2. Diseño e implementación de software

Es el proceso de convertir la especificación del sistema en un sistema ejecutable. El diseño de software se encarga del diseño de la estructura que satisface su especificación. Por otro lado, la implementación se encarga de traducir dicha estructura en un programa ejecutable. Esto también incluye el proceso de debugging (encontrar errores en los programas y corregirlos). Cabe observar que en un proceso ágil, estas dos actividades se realizan de forma intercalada.



Las actividades de diseño incluyen:

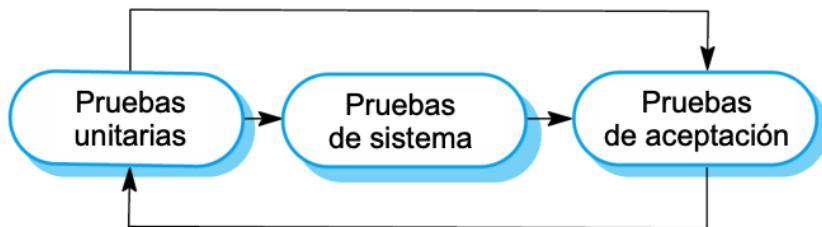
- **Diseño de arquitectura:** identificar todas las estructuras del sistema, los principales componentes, sus relaciones y cómo están distribuidos.
- **Diseño de la base de datos:** diseñar la estructura del sistema de datos y cómo éstos serán representados en la base de datos.
- **Diseño de interfaz de usuario:** se definen las interfaces entre los componentes del sistema.
- **Selección y diseño de componentes:** se buscan componentes reutilizables. En caso de que no estén disponibles, se diseña como deberían operar para luego ser diseñados.

2.4.3. Validación de software

La validación y verificación pretende mostrar que el sistema se ajusta a su especificación y cumple con los requisitos del cliente. Involucra los procesos de comprobación, revisión y pruebas de sistema. Las pruebas de sistema involucran la ejecución del sistema con casos de prueba, los cuales se derivan de la especificación del mismo e intentan simular un procesamiento real del sistema.

Las pruebas de software son unas de las actividades más comúnmente usadas en las actividades de V&V

Etapas del testing

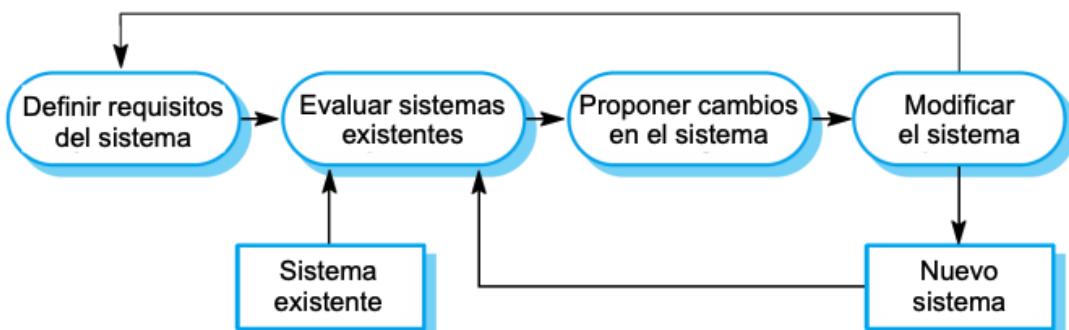


- **Pruebas unitarias:** unidades o componentes de software son probados de forma independiente. Pueden ser funciones, objetos o grupos agrupados de forma coherente. El programador es el que mejor conoce estos componentes y por tanto es el más indicado para realizar este tipo de pruebas.
- **Pruebas de sistema:** implica probar el sistema como un todo. Se preocupa en encontrar errores entre la interacción de los componentes.
- **Pruebas de aceptación:** se prueba el sistema con datos del cliente para comprobar que el sistema cumple con sus necesidades.

2.4.4. Evolución de software

El software puede cambiar en cualquier momento debido a cambios en las circunstancias de negocio, cambios en los requisitos, etc. Por tanto el software debe evolucionar o de lo contrario se vuelve obsoleto.

Históricamente ha existido una división entre la etapa de desarrollo y mantenimiento. Esta distinción es cada vez más irrelevante ya que casi no existen sistemas que sean completamente nuevos. En lugar de verlo como dos procesos independientes, es más realista pensar a la ingeniería de software como un proceso evolutivo donde el software va cambiando constantemente a lo largo de su vida en respuesta a cambios y requisitos y necesidades del cliente.



2.5. Hacer frente al cambio

El cambio es inevitable en todos los grandes proyectos de software. Los cambios en el negocio conducen a cambios en los requisitos de los sistemas. Emergen nuevas tecnologías las cuales posibilitan mejoras en la implementación. Cambios en las plataformas requieren cambios en sus aplicaciones.

Los cambios conducen al retrabajo, por tanto el costo de un cambio incluye tanto el costo del retrabajo como el costo de la implementación de esa nueva funcionalidad.

2.5.1. Reducir costos del retrabajo

Hay dos métodos

- **Anticipación al cambio:** incluye actividades que pueden anticipar los posibles cambios antes de que el retrabajo requerido sea muy grande. Por ejemplo se puede desarrollar un prototipo del sistema que muestre algunas funcionalidades claves a los clientes para su validación.
- **Tolerancia al cambio:** el proceso es diseñado de forma tal que los cambios pueden ser acomodados con un costo relativamente bajo. Esto normalmente involucra desarrollo incremental.

2.5.2. Prototipado de software

Es una versión temprana/inicial de un sistema, el cual se utiliza para demostrar conceptos, probar opciones de diseño y encontrar problemas y sus posibles soluciones. Es esencial que el prototipo se desarrolle rápidamente y de forma iterativa para que los costos sean controlados y se pueda experimentar con el mismo en etapas tempranas del proceso de software.

Un prototipo puede usarse en:

- En el proceso de ingeniería de requisitos, para relevar y validar requisitos.
- En el proceso de diseño, para explorar opciones y desarrollar la interfaz de usuario.
- En el proceso de pruebas, para ejecutar pruebas *back-to-back*.

Beneficios del prototipado

- Mejora la usabilidad del sistema.
- Obtiene una mirada más certera de las reales necesidades del cliente.
- Mejora la calidad del diseño.
- Reduce el esfuerzo de retrabajo.

No siempre tenemos estos beneficios. Puede suceder que el costo del prototipo termine siendo mayor que si se hubiese comenzado con el sistema final en una primera instancia. Sobretodo si el prototipo es descartable.

Desarrollo del prototipo

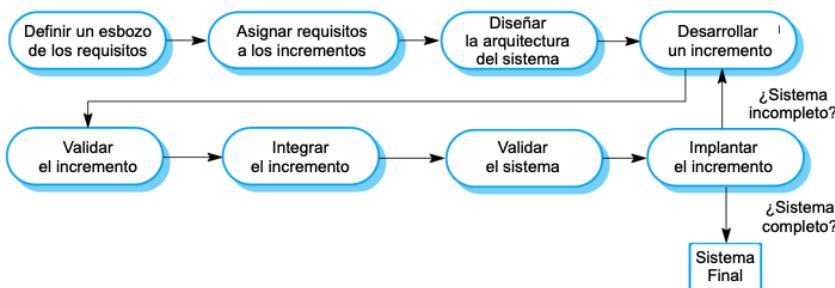
- Puede ser basado en lenguajes o herramientas de prototipado rápido.
- Puede incluirse dejar de lado alguna funcionalidad
 - Debe enfocarse en las áreas del producto que aún no se entienden por completo.
 - No es necesario que incluya recuperación y chequeo de errores.
 - Generalmente se enfocan más en los requisitos funcionales que en los no-funcionales, como ser confiabilidad o seguridad

Prototipos descartables Los prototipos deben ser descartados si no conforman una buena base para el sistema en producción

- Puede que sea imposible (o muy costoso) ajustarlo para que cumpla con requerimientos no funcionales.
- Muchas veces no se documentan.
- Si los cambios son muy rápidos y frecuentes, pueden degradar la estructura del prototipo fácilmente.
- Los prototipos generalmente no cumplen con los estándares de calidad organizacionales.

2.5.3. Entregas incrementales

El desarrollo se partitiona en incrementos, en donde cada incremento entrega parte de la funcionalidad requerida. Los requerimientos de usuario son priorizados y los más importantes se incluyen en incrementos tempranamente.



Ventajas

- Disponibilidad temprana de funcionalidades de valor para el cliente.
- Las entregas tempranas actúan de prototipos que ayudan a validar y relevar requisitos para próximos incrementos.
- Las funcionalidades/servicios de más alta prioridad, al ser desarrollados en los primeros incrementos, son los que reciben más testing.
- Se mantienen los beneficios del desarrollo incremental en que debería ser relativamente sencillo incorporar cambios al sistema.

Problemas

- La mayoría de los sistemas requieren un conjunto básico de facilidades que son usadas por las diferentes partes del sistema. Si los requerimientos no están definidos en detalle, es difícil identificar ese conjunto de facilidades que serán necesarias para todos los incrementos.
- La esencia de los procesos iterativos es que la especificación se desarrolla en conjunto con el software. Esto muchas veces genera conflicto con los modelos de adquisición de varias organizaciones, en donde la especificación completa del sistema es parte del contrato del desarrollo del mismo.

2.6. Mejora de procesos

Muchas empresas de software han recurrido a la mejora de procesos como una forma de mejorar la calidad del software, reducir los costos y acelerar sus procesos de desarrollo. La mejora de procesos significa entender los procesos existentes y cambiar dichos procesos para mejorar la calidad del producto y/o reducir costos y tiempos de desarrollo.

2.6.1. Enfoques para la mejora

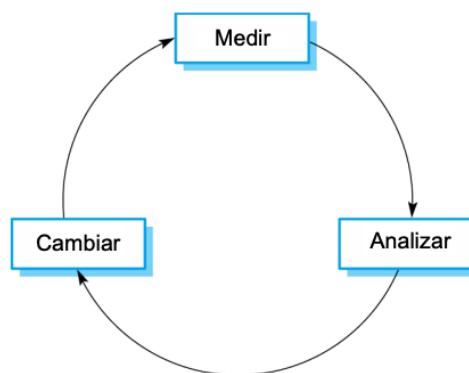
- **Enfoque de madurez de proyectos**

Se centra en mejorar procesos y gestión de proyectos introduciendo buenas prácticas de ingeniería de software. El nivel de madurez del proceso refleja el grado en el cual han sido adoptadas las buenas prácticas dentro del proceso de desarrollo de software.

- **Enfoque ágil**

Se centra en el desarrollo iterativo y en la reducción de los costos generales del proceso de desarrollo. La principal característica de los métodos ágiles es la entrega rápida de funcionalidad y la rápida respuesta a los cambios de los requisitos del cliente.

2.6.2. Ciclo de mejora de procesos



2.6.3. Actividades de la mejora de procesos

- **Medición del proceso**

Medir uno o más atributos del proceso/producto de software. Dichas medidas formarán el baseline que ayudará a decidir si las mejoras implantadas han sido efectivas.

- **Análisis del proceso**

El proceso actual es evaluado, en donde se identifican las debilidades y cuellos de botella. Los modelos de procesos describen el proceso que va a ser desarrollado.

- **Cambios en el proceso**

En base al análisis, cambios son propuestos para abordar algunas de las debilidades identificadas. Estos son introducidos y el ciclo se reanuda para recopilar información acerca de la efectividad de dichos cambios.

2.6.4. Medición del proceso

- Se deben recopilar datos cuantitativos (si es posible). Muchas veces es difícil saber qué medir mientras las organizaciones no tengan claramente definidos los estándares de procesos. En lo posible el proceso debe estar definido antes de realizar las mediciones.
- Las mediciones en el proceso deben usarse para asegurar las mejoras en el mismo. Esto no significa que las medidas deban guiar las mejoras. Las mejoras deben ser guiadas por los objetivos organizacionales.

Métricas de proceso

- **Tiempo** que lleva completar actividades del proceso.
- **Recursos** requeridos para las actividades o procesos.
- Cantidad de ocurrencias en un **evento particular**

3. Desarrollo de Software Ágil

Hoy en día uno de los más importantes requisitos para los sistemas de software es el desarrollo y entrega *rápida* de software. El software debe de evolucionar rápidamente para reflejar los cambios en las necesidades de negocio. Surgen a finales de los años 90 cuyo objetivo se enfoca en reducir radicalmente el tiempo de entrega de software funcionando.

3.1. Características

- Las actividades de especificación, diseño e implementación están intercaladas.
- El sistema es desarrollado a través de una serie de versiones en donde el cliente está involucrado en la especificación y evaluación de cada versión.
- Se realizan entregas frecuentes de nuevas versiones para evaluación.
- El desarrollo de software es soportado por el uso extensivo de herramientas, por ejemplo herramientas de testing automatizado.
- Documentación mínima. Se enfoca en que el código funcione.

3.2. Manifiesto ágil

“Estamos descubriendo formas mejores de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros. A través de este trabajo hemos aprendido a valorar:

- Individuos e interacciones sobre procesos y herramientas.
- Software funcionando sobre documentación extensiva.
- Colaboración con el cliente sobre negociación contractual.
- Respuesta ante el cambio sobre seguir un plan.

Esto es, aunque valoramos los elementos de la derecha, valoramos más los de la izquierda”

3.3. Principios de los métodos ágiles

Principio	Descripción
Involucramiento del cliente	Los clientes deben de estar estrechamente involucrados a lo largo del proceso de desarrollo. Su rol es proveer y priorizar nuevos requisitos del sistema y evaluar el resultado de las iteraciones.
Desarrollo incremental	El software es desarrollado en incrementos, en donde el cliente especifica que requisitos deben ser incluidos en cada incremento.
Personas antes que los procesos	Las habilidades del equipo de desarrollo deben ser reconocidas y explotadas. Se debe permitir que los miembros del equipo desarollen sus propias formas de trabajar, sin procesos prescriptivos.
Aceptar el cambio	Esperar que los requisitos cambien y por ende, modificar el sistema para acomodar dichos cambios.
Mantener la simplicidad	Centrarse en la simplicidad tanto del proceso de desarrollo como en el producto que se está desarrollando. Siempre que sea posible, trabajar activamente en eliminar la complejidad del sistema.

3.4. Aplicabilidad de las metodologías ágiles

- Desarrollo de productos de pequeño o mediano porte.
- Desarrollo de sistemas donde hay un claro compromiso del cliente en participar del proceso de desarrollo de software, en donde hay pocas reglas y no hay regulaciones externas que afecten al software.

3.5. Técnicas de desarrollo ágil

3.5.1. Extreme Programming (XP)

Uno de los métodos ágiles con gran influencia, desarrollado a fines de los años 90, el cual introduce varias técnicas de desarrollo ágil. Toma en extremo el enfoque del desarrollo iterativo:

- Nuevas versiones pueden generarse varias veces por día (integración continua).
- Los incrementos son entregados a los clientes cada dos semanas.
- Todas las pruebas deben ser ejecutadas para cada versión, y esta es aceptada solo si todas las pruebas se ejecutan satisfactoriamente.

3.5.2. Prácticas de XP

Principio	Descripción
Planificación incremental	Los requisitos que serán incluidos en una liberación serán determinados por el tiempo disponible y su prioridad relativa. Los desarrolladores dividirán la especificación del requisito en “tareas” de desarrollo.
Liberaciones pequeñas	Se debe desarrollar primero el conjunto de funcionalidades que sea “mínimo y útil” y que dé valor de negocio. Las liberaciones del sistema deben realizarse con frecuencia, agregando funcionalidad a la primer versión liberada.
Diseño simple	Diseño suficiente para cumplir los requisitos actuales y no más.
Desarrollo dirigido por pruebas (TDD)	Se utiliza un framework de pruebas unitarias automatizadas para generar las pruebas de una nueva funcionalidad, antes que la funcionalidad esté implementada.
Refactorización	Se espera que todos los desarrolladores realicen refactorización al código de forma continua, tan pronto como se encuentren posibles mejoras al mismo. Esto mantiene el código simple y mantenable.
Programación por pares	Los desarrolladores trabajan de a pares, revisándose el trabajo mutuamente y proporcionando soporte para realizar siempre un buen trabajo.
Propiedad colectiva	Los “pares” de desarrolladores trabajan en todas las áreas del sistema, por lo que no se generan islas de experiencia. Además, los desarrolladores toman responsabilidad por todo el código. Cualquiera puede cambiar cualquier cosa
Integración continua	No bien una tarea es terminada, se debe integrar con todo el sistema. Luego de cada integración, todas las pruebas unitarias deben ser ejecutadas con resultados exitosos.
Ritmo sostenible	Gran cantidad de horas extras son consideradas inaceptables. El efecto neto a menudo reduce la calidad del código y a mediano plazo la productividad.
Cliente	Una cantidad representativa de los usuarios del sistema (o el cliente) deben estar disponible para el equipo de XP. En un proceso XP, el cliente es un miembro del equipo de desarrollo y es el responsable de brindar al equipo los requerimientos del sistema a ser implementados.

En la práctica la aplicación de Extreme Programming como fue originalmente propuesto ha demostrado ser más difícil que lo anticipado. El aporte más significante de método fue el conjunto de prácticas ágiles que introdujo a la comunidad. Entre ellas tenemos:

- Historias de usuario.
- Refactorización.
- Desarrollo dirigido por pruebas.
- Programación por pares.

3.5.3. Historias de usuario

Una historia de usuario es un escenario de uso de un usuario del sistema. En la mayor medida de lo posible, el cliente trabaja de forma cercana al equipo de desarrollo y discute estos escenarios con otros miembros del equipo. Entre todos desarrollan una “tarjeta de historia” (story card) que describe brevemente una historia que encapsula las necesidades del cliente. El equipo de desarrollo luego intenta implementar dicho escenario en una versión futura del software.

Una vez que las tarjetas de historia son desarrolladas, el equipo de desarrollo se encarga de dividirla en tareas, y estima el esfuerzo que llevaría cada una de ellas. Esto usualmente involucra discusiones con el cliente para refinar los requisitos. El cliente luego prioriza las historias para implementar, eligiendo aquellas de las que puede obtener valor inmediatamente. Si surge algún cambio sobre un requerimiento ya implementado, se desarrolla una nueva tarjeta de historia, y el cliente le asigna una cierta prioridad respecto a otras tareas.

El principal problema con las historias de usuario es la completitud. Es difícil juzgar si se desarrollaron suficientes para cubrir todos los requisitos esenciales. Es también difícil juzgar si una historia da un panoramaadero de una actividad. Usuarios experimentados están usualmente tan familiarizados con su trabajo que dejan cosas sin describir.

3.5.4. Refactorización

Los desarrolladores de XP sugirieron que el código que se desarrolla debe de ser constantemente refactorizado. Refactorizar significa que el equipo de programación busque mejoras para realizar al software y las implemente de forma inmediata. Cuando un equipo ve código que puede mejorarse, realizan esas mejoras aún en situaciones donde no hay una necesidad inmediata de hacerlo.

Un problema del desarrollo incremental es que los cambios suelen degradar la estructura de software. Además, nuevos cambios son cada vez más difíciles de introducir. Refactorizar mejora la estructura y legibilidad del software, y de este modo evita el deterioro que ocurre naturalmente cuando el software cambia.

En la teoría, cuando la refactorización es parte del proceso de desarrollo, el software debe ser siempre fácil de entender y debe cambiar cuando nuevos requisitos son presentados. Sin embargo, en la práctica, esto no es siempre el caso. Muchas veces las presiones en el desarrollo llevan a que la refactorización se postergue para otro momento ya que se utiliza el tiempo de desarrollo en la implementación de nuevas funcionalidades.

3.5.5. Desarrollo dirigido por pruebas

Extreme Programming desarrolló un nuevo enfoque del testing para resolver las dificultades de testear sin una especificación. El testing es automatizado y central en el proceso de desarrollo. No se puede continuar con el desarrollo hasta que todos los tests sean ejecutados exitosamente. Las claves del testing en XP son:

- Desarrollo dirigido por pruebas
- Desarrollo de test incrementales a partir de escenarios.
- Involucramiento del usuario en el desarrollo y validación de los tests.
- Uso de frameworks de automatización de tests.

En lugar de escribir código y luego escribir tests para eso, primero se escriben los tests y luego el código. De esta forma se pueden correr los tests a medida que se desarrolla el código y descubrir problemas durante el desarrollo.

En el desarrollo dirigido por pruebas los desarrolladores deben de entender en profundidad la especificación para poder escribir las pruebas para el sistema. Esto significa que las ambigüedades e incertidumbres deben de ser aclaradas antes de pasar a la implementación. Por otro lado evita el problema de “test-lag” que sucede cuando el equipo de desarrollo trabaja más rápido que el tester.

El enfoque orientado por pruebas de XP asume que las historias de usuario fueron desarrolladas y divididas en un set de “task cards”. Cada una de estas tareas genera una o más unidades de test (unit tests) que chequean la implementación que se describe en dicha tarea. El rol del cliente en el proceso de testing es ayudar a desarrollar tests de aceptación para las historias que se implementarán en una release futura.

La automatización de tests es fundamental para un enfoque dirigido por pruebas. Los tests se escriben como componentes ejecutables antes de implementar la tarea. Estos deben ser independientes, deben simular el ingreso de los inputs correspondientes, y debe chequear que las salidas cumplan la especificación. Como las pruebas están automatizadas, siempre hay un set de pruebas que se puede ejecutar de forma rápida. De esta forma, cuando se agrega una nueva funcionalidad, se ejecutan las pruebas y es posible detectar problemas que el nuevo código introdujo de forma inmediata.

Tenemos ciertos problemas, sobretodo para asegurarnos que la cobertura de los tests es completa.

- Programadores prefieren programar antes que testear, y muchas veces se toman atajos al escribir los tests.
- Algunos tests pueden ser difíciles de escribir de forma incremental.

3.5.6. Programación por pares

Otra práctica innovadora que fue introducida por XP, es que los programadores trabajen en pares para desarrollar software. Esto es, se sientan en la misma computadora a programar. Sin embargo, no siempre trabajan con los mismos pares, sino que estos van rotando para que todos trabajen con todos.

Tenemos las siguientes ventajas:

- Apoya la idea de la propiedad y responsabilidad colectiva del sistema.
- Actúa como un proceso informal de review de código, ya que cada linea de código es vista por al menos dos personas.
- Incentiva la refactorización para mejorar la estructura del software.

Se podría pensar que la programación por pares va a ser menos eficiente que dos programadores por su cuenta. Estudios formales tienen resultados mixtos:

- Utilizando estudiantes como voluntarios se observó que la productividad trabajando de a pares era semejante a la de dos personas trabajando independientemente. Además, se evitaron más cantidad de errores lo cual a su vez redujo el retrabajo.
- Estudios con programadores con más experiencia no replicaron estos resultados. Se vio una significante pérdida de productividad. Hubieron algunos beneficios de calidad, pero no compensaban la perdida de productividad observada.

3.6. Gestión de proyectos ágiles

La principal responsabilidad de los líderes o jefes de proyecto es gestionar el proyecto de forma tal que el software sea liberado en fecha y dentro del presupuesto planificado. El enfoque estándar para un líder de proyecto es el dirigido por planes donde se traza un plan para el proyecto, el cuál muestra que va a ser entregado, cuándo va a ser entregado y quiénes van a trabajar en dichas entregas. Sin embargo, la gestión de proyectos ágiles requiere un enfoque ligeramente distinto, el cual se adapte al desarrollo incremental y a las prácticas usadas en los métodos ágiles.

3.6.1. SCRUM

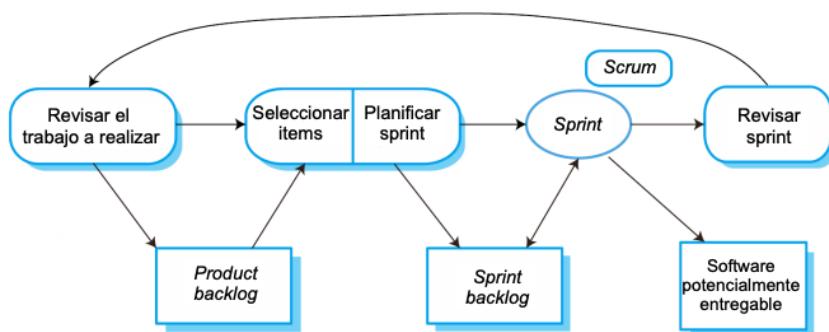
Scrum es un método ágil que se enfoca en la gestión del desarrollo iterativo más que en las prácticas ágiles específicas. Comprende tres fases:

- **La fase inicial:** se genera un esquema de la planificación en donde se establecen los objetivos generales del proyecto y el diseño de la arquitectura de software.
- **Ciclos de sprints:** la fase inicial es seguida de una serie de iteraciones, en donde se desarrolla un incremento del sistema en cada una.
- **Fase de clausura:** en esta fase se cierra el proyecto, se completa la documentación requerida y se evalúan las lecciones aprendidas del mismo.

3.6.2. Elementos de SCRUM

Término	Definición
Scrum	
Equipo de desarrollo	Un grupo auto-organizado de desarrolladores de no más de 7 personas. Son responsables de desarrollar el software y otros documentos esenciales del proyecto.
Incremento del producto	El incremento de software que se entrega en un sprint. La idea es que debe ser “potencialmente entregable”, lo cual significa que está en un estado terminado, donde no hay más trabajo (como testing) para hacer, necesario para incorporarlo al producto final. En la práctica esto no es siempre alcanzable.
Product Backlog	Es una lista de cosas para hacer que el equipo Scrum debe realizar. Pueden ser definiciones de features, requerimientos de software, historias de usuario o descripciones de tareas suplementarias que se deben realizar como ser definición de arquitectura o documentación.
Product Owner	Es una persona (o grupo pequeño) cuyo trabajo es identificar features o requerimientos, priorizarlos y continuamente revisar el product backlog para asegurarse que el proyecto cumple las necesidades del negocio. El Product Owner puede ser el cliente, pero también puede ser el “product manager” en una empresa de software, o un representante de las partes interesadas.
Scrum	Una reunión diaria del equipo Scrum, en la que se evalúa el progreso y se prioriza el trabajo a hacer en el día. Idealmente debe ser una reunión cara a cara y de corta duración (no más de 15 minutos).
Scrum Master	El Scrum Master es el responsable de asegurarse que el proceso Scrum es seguido, y guía al equipo para un uso efectivo de Scrum. Es responsable de comunicarse con el resto de la compañía y de asegurarse que el equipo Scrum no está desviado por una interferencia externa. Cabe aclarar que no es lo mismo que un “project manager”, aunque muchas veces es difícil de distinguir.
Sprint	Una iteración de desarrollo. Usualmente los sprints duran entre 2 y 4 semanas.
Velocity	Es un estimado de cuánto esfuerzo del backlog el equipo puede cubrir en un sprint. Entender la velocity del equipo ayuda a estimar qué puede ser cubierto en un sprint y provee una base para medir la mejora en la performance del equipo.

3.6.3. Ciclos de sprints en SCRUM



El punto de comienzo de un ciclo de Sprint Scrum es el Product Backlog. Allí se encuentran las tareas para hacer, las cuales pueden estar especificadas en distintos niveles de detalle. Es responsabilidad del Product Owner asegurarse que el nivel de detalle en la especificación de una tarea es apropiada para el trabajo a hacer. Luego de seleccionar los items que formarán parte del sprint, se da por comenzado el mismo. Este dura entre 2 a 4 semanas. Los sprints nunca se extienden si queda trabajo sin terminar, sino que estas tareas vuelven al product backlog. Todo el equipo Scrum se encarga de priorizar las tareas y estimar el tiempo requerido para terminarlas. Esto puede hacerse por ejemplo por medio de Story Points. Durante el sprint, todo el equipo tiene una reunión diaria (también llamada Scrum), donde se evalúa el progreso y se priorizan las tareas a realizar en el día. Además cada integrante menciona en qué trabajó y en qué trabajará en ese día. De esta forma todo el equipo sabe en qué está trabajando cada integrante.

Al final del sprint se realiza una reunión de evaluación. Esta reunión tiene dos propósitos. En primer lugar, es una forma de mejorar el proceso. El equipo evalúa la forma en la que estuvo trabajando y qué se pudo haber hecho mejor. Por otro lado, provee input al product backlog para evaluarlo, lo cual precede el comienzo del siguiente sprint.

Cabe aclarar que el “Scrum master” es un facilitador, quien concilia las reuniones diarias, gestiona el backlog de trabajo a ser realizado, registra las decisiones tomadas, mide el progreso del proyecto contra el backlog y se comunica con clientes y gerencias externas al proyecto.

3.6.4. Beneficios de SCRUM

- El producto es dividido en partes fácilmente entendibles y gestionables.
- Los requisitos inestables no trancan el progreso/
- Todo el equipo tiene visibilidad de todo y consecuentemente se mejora la comunicación entre los integrantes.
- Los clientes reciben a tiempo las entregas de los incrementos y se gana feedback en cómo el producto funciona (y debería funcionar).
- Se establece una confianza entre el cliente y el equipo, en donde se crea una cultura positiva en la cual cada uno espera que el proyecto tenga éxito.

3.7. Escalando métodos ágiles

Los métodos ágiles han probado ser efectivos para ciertos tipos de proyectos (productos pequeños/medianos, equipos reducidos y ubicados en el mismo lugar físico, con ciertas habilidades, etc.).

Escalar hacia arriba refiere a la utilización de métodos ágiles para el desarrollo de grandes sistemas de software, los cuales no pueden ser desarrollados por equipos reducidos. Por otro lado, escalar hacia afuera refiere a como los métodos ágiles pueden ser introducidos a lo largo de una

gran organización con muchos años de experiencia en el desarrollo de software.

Cuando se escalan los métodos ágiles es importante mantener una planificación flexible, liberaciones frecuentes y buena comunicación con el equipo.

3.7.1. Problemas al escalar

- La informalidad de los métodos ágiles es incompatible con la definición de “contrato” comúnmente usada en grandes compañías.
- Los métodos ágiles son más apropiados para el desarrollo de nuevos productos más que para el mantenimiento de los mismos. La mayor parte de los costos en las grandes compañías refieren a mantener sus sistemas de software ya existentes.
- Los métodos ágiles están diseñados para pequeños equipos que están ubicados en el mismo lugar físico. Grandes compañías desarrollan software con equipos distribuidos a lo largo de todo el mundo.

3.7.2. Problemas con el mantenimiento ágil

Problemas clásicos son los siguientes:

- Falta de documentación del producto.
- Mantener a los clientes involucrados en el proceso de desarrollo.
- Mantener la continuidad del equipo de desarrollo.

EL desarrollo ágil confía en que el equipo de desarrollo sabe y entiende que es lo que se debe hacer. Para sistemas de “larga vida” un gran problema es que los desarrolladores originales no siempre trabajan en el mantenimiento.

3.7.3. Enfoques dirigidos por planes y ágiles

Debemos hacernos preguntas para ver cual enfoque es más indicado para cada caso.

Respecto al sistema

- ¿Qué tan grande es el sistema a ser implementado?
Los métodos ágiles son más adecuados para cuando el sistema puede ser desarrollado por un pequeño equipo que puede comunicarse de forma informal.
- ¿De qué tipo es el sistema a ser implementado?
Si el sistemas requiere mucho análisis previo es más adecuado utilizar un enfoque dirigido por planes.
- ¿Cuál es el tiempo de vida esperado del sistema?
Sistemas de larga vida pueden requerir más documentación que es más difícil de mantener con un enfoque ágil.
- ¿Está el sistema regido por alguna regulación externa en particular
Si el sistema necesita ser aprobado por un ente externo, luego podría ser necesario producir documentación rigurosa.

Respecto al equipo

- ¿Qué tan buenos son los programadores y diseñadores del equipo?
Muchas veces se discute que un enfoque ágil requiere de un equipo con mayor experiencia.
- ¿Cómo está organizado el equipo?
Si el equipo está distribuido o tiene una parte terciarizada, deberá proveer documentación para que exista una comunicación fluida entre los equipos de desarrollo.

- ¿Qué tecnologías hay disponibles?

Métodos ágiles muchas veces se apoyan en buenas herramientas para llevar un registro de la evolución del diseño.

Respecto a la organización

- ¿Es importante tener los requisitos y el diseño detallados antes de implementar?

Si este es el caso, muy probablemente sea más útil un enfoque dirigido por planes para la ingeniería de requisitos. Pero se podría usar un enfoque ágil para el desarrollo.

- ¿Es factible obtener feedback de cliente con entregas rápidas?

- ¿Hay problemas culturales que pueden afectar el desarrollo del sistema?

3.7.4. Métodos ágiles para sistemas grandes

Los métodos ágiles deben de evolucionar para utilizarse en el desarrollo de software a gran escala.

Estos sistemas son más complejos debido a 6 factores:

- Grandes sistemas son usualmente sistemas de sistemas.
- Grandes sistemas incluyen interacciones con otros sistemas ya existentes.
- Como muchos sistemas son integrados para crear un sistema, una parte del desarrollo se enfoca en la configuración del sistema, en lugar de desarrollo de código.
- Grandes sistemas y su proceso de desarrollo son muchas veces limitados por reglas y regulaciones externas, limitando la forma en la que pueden ser desarrolladas.
- Grandes sistemas tienen un tiempo de desarrollo muy largo.
- Grandes sistemas generalmente tienen un gran número de partes interesadas, muchas veces con perspectivas y objetivos distintos.

3.7.5. IBM Agile Scaling Model (ASM)

- Un enfoque incremental para la ingeniería de requisitos es imposible. Se debe realizar un trabajo inicial para identificar las distintas partes del sistema y qué equipo trabajará en cada parte.
- No puede haber un único PO. Se deben involucrar distintas personas para diferentes partes del equipo las cuales deben de estar comunicadas.
- No es posible enfocarse únicamente en código. Es necesario preocuparse por diseño y documentación.
- Se deben diseñar y usar mecanismos para la comunicación entre distintos equipos.
- La integración continua es prácticamente imposible. Sin embargo, es esencial mantener una salida de versiones frecuentes.

3.7.6. Multi-team Scrum

- Replicación de roles: Cada equipo tiene un PO y Scrum Master. Puede haber un chief PO o Scrum Master para todo el proyecto.
- Arquitectos de software: Cada equipo elige a un arquitecto de software los cuales colaboran en el diseño y evolucionan la arquitectura del sistema.
- Alineación de releases: la fecha de las releases de cada equipo deben de estar alineadas para que se pueda producir un incremento completo y demostrable del sistema.
- Scrum de Scrum: Hay una Scrum de Scrum diaria donde representantes de cada equipo discuten el progreso, identifican problemas y planean el trabajo a futuro.

3.8. MUM (Modularizado, Unificado y Medible)

3.8.1. Principales características

- Iterativo incremental: desarrolla el sistema en refinamientos sucesivos incrementando la solución definida. Se mitigan riesgos en cada iteración.
- Basado en casos de uso: se capturan los requerimientos mediante casos de uso que guían el diseño, implementación y verificación del software en desarrollo.
- Centrado en la arquitectura: la arquitectura prioriza los casos de uso más significativos y especifica la estructura del sistema. Se da una definición temprana del esqueleto base de la solución definida. Esto se hace para evitar el refactoring.
- Modelado en UML.

3.8.2. Beneficios del enfoque iterativo

- Los riesgos son identificados y mitigados. Ayuda a mitigar problemas en integración, problemas en construir el sistema equivocado, y problemas en la arquitectura.
- Permite planificar el cambio en la próxima iteración
- Alto nivel de reutilización.
- El equipo aprende a lo largo del proyecto.
- El producto logra una mejor calidad global.

3.8.3. Dimensión del tiempo

El desarrollo se divide en 4 fases: Inicial, Elaboración, Construcción y Transición. Cada fase se divide en iteraciones y tiene objetivos definidos que se alcanzan según entregables que se deben obtener.

■ Fase inicial

Identificar requerimientos y riesgos. Estimar recursos. Evaluar capacidad de hacer el proyecto.

■ Fase de Elaboración

Dominio del problema. Prototipo de arquitectura. Plan del proyecto. Eliminar riesgos.

■ Fase de Construcción

Construcción del software.

■ Fase de Transición

Instalar el producto en el entorno del cliente. Verificar que todo funcione. Se llega a la versión final. Se corren las pruebas de aceptación.

Reuniones de equipo

- Evaluar el cumplimiento del plan de la iteración pasada.
- Identificar causas de desvíos y problemas.
- Analizar los riesgos que enfrenta el proyecto.
- Definir acciones a tomar.
- Ajustar el plan de la nueva iteración.

Plan de iteración

- Cada actividad se identifica por un código, el que sirve para registrar el trabajo realizado en la planilla de Registro de Actividad.

- Se debe indicar además la descripción, las fechas de inicio y fin planificadas, el esfuerzo (en horas/persona) planificado, los participantes previstos y el responsable.
- Las actividades son Las que surgen del modelo de proceso (agrupadas de forma adecuada para la planificación y el control) y actividades necesarias que no figuran en el modelo (estudio del dominio, instalación de herramientas, etc.)

3.8.4. Dimensión del Modelo

Cuatro elementos para mostrar quién está haciendo qué, de qué forma (cómo) y en qué momento (cuándo).

- **Disciplinas:** describen cuándo, agrupan actividades en forma lógica
- **Roles:** describen quién, responsabilidades
- **Actividades:** describen cómo, las acciones
- **Entregables:** describen qué, artefactos

3.8.5. Disciplinas

- Requerimientos
 - Establecer y mantener un acuerdo con los clientes e involucrados sobre qué debe hacer el sistema.
 - Proporcionar a los desarrolladores un mejor entendimiento de los requerimientos del sistema.
 - Definir el alcance del sistema.
 - Proporcionar las bases para la planificación del contenido técnico de las iteraciones.
 - Proporcionar las bases para estimar costo y tiempo para desarrollar el sistema.
 - Definir la interfaz de usuario del sistema, enfocando en las necesidades y objetivos de los usuarios
 - Para lograr estos objetivos, es importante entender la definición y alcance del problema que trata de resolver este sistema.
- Diseño
 - Transformar los requerimientos en el diseño de lo que debe ser el sistema.
 - Desarrollar una arquitectura robusta para el sistema.
 - Adaptar el diseño para que se corresponda con el entorno de implementación, diseñando de forma que la implementación tenga buena performance.
 - Para lograr estos objetivos es necesario conocer el alcance del sistema, las herramientas de desarrollo y los recursos disponibles
- Implementación
 - Definir la organización del código en términos de implementación de subsistemas organizados en capas.
 - Implementar clases y objetos en términos de componentes (archivos fuentes, binarios, ejecutables y otros).
 - Verificar los componentes desarrollados como unidades.
 - Integrar los resultados producidos por cada implementador (o por equipos), en un único sistema ejecutable.
 - En esta disciplina se implementan los distintos subsistemas y luego se integran para formar un sistema

- Verificación
 - Verificar la interacción entre componentes.
 - Verificar la correcta integración de todos los componentes del sistema.
 - Verificar que todos los requerimientos han sido correctamente implementados.
 - Identificar y asegurar que los defectos sean corregidos antes de la liberación de la versión del producto de software.
- Implantación
 - Puede hacerse una instalación personalizada, ofrecimiento del producto empaquetado, acceso al software por internet.
 - Si bien la mayoría de las actividades de Implantación se realizan en la Fase Transición algunas comienzan en fases anteriores, como por ejemplo la Planificación de la Implantación

3.8.6. Otros roles

- Gestión de proyecto
 - Proveer un ambiente para manejar proyectos de software intensivos.
 - Proveer una guía práctica para la planificación, gestión de recursos humanos, ejecución y monitoreo de los proyectos.
 - Proveer un ambiente para la Gestión de riesgos.
 - Realizar un seguimiento del avance del proyecto.
 - Realizar estimaciones y mediciones: de esfuerzo (en horas de trabajo), tamaño del producto en desarrollo y factibilidad del proyecto.
- Gestión de configuración (SCM)
 - Identificar los elementos del proyecto que deben estar bajo configuración
 - Restringir los cambios a dichos elementos
 - Auditarse los cambios a estos elementos
 - Definir y gestionar la configuración de estos elementos
- Gestión de calidad (SQA)
 - Identificar las propiedades de calidad
 - Realizar el seguimiento y control de la calidad del sistema en desarrollo.
 - Realizar el seguimiento y control de la calidad del proceso, las métricas y los procedimientos seguidos en el proyecto.
 - Indicar qué acciones correctivas deben tomarse en caso de encontrar inconsistencias o incumplimientos de la calidad.
 - Asegurar que el sistema desarrollado cumple con determinadas propiedades de calidad preestablecidas.
- Comunicación
 - Definir los métodos de comunicación y herramientas utilizadas para la comunicación entre los integrantes del equipo.
 - Definir los métodos de comunicación y herramientas utilizadas para la comunicación de los integrantes del equipo con entidades externas al equipo (Director de proyecto, docentes, Cliente, usuarios, etc...).
 - Convocar reuniones informativas donde se traten temas de interés para los participantes.

- Elaborar documentos informativos sobre temas de interés para los integrantes del equipo.
- Realizar el seguimiento de la Satisfacción del Cliente.
- Formación y Entrenamiento
 - Proporcionar al equipo de trabajo la formación necesaria así como también el entrenamiento y la capacitación para lograr una correcta comprensión del modelo y sus actividades.

3.8.7. Roles y combinación

El modelo propone tomar roles y combinación de roles. Los roles son los siguientes:

- Analista
- Arquitecto
- Especialista Técnico
- Implementador
- Responsable de Verificación
- Administrador
- Responsable de SQA.
- Responsable de SCM
- Documentador de Usuario
- Asistente de verificación

4. Ingeniería de Requisitos

Los requisitos de un sistema son descripciones de los servicios que el sistema debería proveer, y las restricciones en su operación. Estos requisitos reflejan la necesidad de clientes en un sistema que sirva un cierto propósito. El proceso de descubrir, analizar, documentar y chequear esos servicios y restricciones es llamado ingeniería de requisitos (RE - Requirements Engineering).

Los costos en los errores de requisitos van aumentando a medida que pasamos en las etapas del proceso. Por tanto, será fundamental tratar de detectar la mayor cantidad de errores en la etapa de la ingeniería de requisitos.

4.1. Abstracción de requisitos

La forma de un requisito puede variar desde una oración con un alto nivel de abstracción, hasta una especificación funcional matemática.

- Puede ser la base de una oferta para un contrato. En ese caso debe estar abierto a la interpretación.
- Puede ser la base para un contrato. En ese caso debe ser definido detalladamente.

4.1.1. Tipos de requisitos

- **Requisitos de usuario**

Son declaraciones en lenguaje natural, además de diagramas de los servicios que proporciona el sistema y sus limitaciones operativas. Escritos para los clientes.

- **Requisitos del sistema**

Un documento estructurado que establece descripciones detalladas de las funciones, servicios y restricciones operacionales del sistema. Define exactamente lo que se debe implementar. Escritos para quienes desean saber como se afectará el proceso del negocio, o para personas que forman parte de la implementación del sistema.

Los interesados del sistema (system stakeholders) incluye a cualquier persona que se vea afectada por el sistema de alguna forma, o que tenga un interés legítimo en el mismo.

- Clientes y usuarios - requisitos adecuados a sus necesidades.
- Diseñadores - comprenderlos para lograr diseño que los satisfaga.
- Supervisores del contrato - sugieren hitos de control, cronogramas.
- Gerentes del negocio - entienden el impacto en la organización.
- Verificadores - comprenderlos para poder verificar si el sistema satisface.

4.1.2. Tipos de información de requisitos

- **Requisitos del dominio**

Se derivan del dominio de aplicación del sistema y no de las necesidades específicas de los usuarios.

- **Requisitos del negocio**

Un objetivo de alto nivel de una organización que desarrolla un producto o de un cliente que lo compra.

- **Regla de negocio**

Una política, guía, estándar o regulación que define o restringe algún aspecto del negocio.

- **Requisito de interfaz externa**

Una descripción de una conexión entre un sistema de software y un usuario, otro sistema de software o un dispositivo de hardware.

- **Característica (feature)**

Una o más capacidades relacionadas de formas lógicas que proveen valor al usuario y son descriptas como un conjunto de requisitos funcionales. Por ejemplo los favoritos de un navegador.

- **Requisito funcional**

Una descripción de lo que el sistema debe hacer bajo condiciones específicas.

- **Requisitos no funcionales**

Una descripción de una propiedad o característica que un sistema debe poseer o una restricción que deba respetar.

- **Atributo de calidad**

Un tipo de requisito no funcional que describe una característica de servicio o desempeño de un producto.

4.2. Requisitos funcionales y no funcionales

4.2.1. Requisitos funcionales

Son declaraciones de los servicios que el sistema debería proveer, cómo el sistema debería reaccionar a entradas particulares y cómo debería comportarse en situaciones particulares. En algunos casos pueden indicar explícitamente lo que el sistema NO debería hacer.

Estos requisitos dependen en el tipo de software que se está desarrollando, la cantidad de usuarios en el sistema, y el enfoque tomado por la organización cuando escribió los requerimientos.

- Cuando son expresados como **requisitos de usuario** deben ser escritos en lenguaje natural para que usuarios y gerentes puedan entenderlo.
- Cuando son expresados como **requisitos del sistema**, expanden los requisitos de usuario y son escritos para desarrolladores del sistema. Deben describir las funciones del sistema, entradas y salidas, y excepciones en detalle.

Los requisitos funcionales pueden variar entre lo que debe hacer el software (literalmente), hasta algo mucho más específico y describir cómo se hace el trabajo localmente en una organización.

Falta de precisión en los requisitos

Falta de precisión en los requisitos puede llevar a disputas entre el cliente y los desarrolladores. Es natural para un desarrollador interpretar un requisito ambiguo de la forma que simplifique su implementación. Sin embargo, muchas veces esto no era lo que quiere el cliente. Luego se deben especificar nuevos requisitos y realizar cambios al sistema. Esto retrasa las entregas e incrementa los costos.

Compleitud y consistencia Idealmente la especificación de requisitos funcionales de un sistema debe ser tanto completa como consistente. Completa significa que todos los servicios e información requerida por los usuarios está definida. Consistente significa que los requisitos no deben contradecirse.

Sin embargo, en la práctica, solo es posible alcanzar tanto completitud como consistencia para pequeños sistemas. Para sistemas grandes es sencillo cometer errores y omitir detalles cuando se escribe la especificación. Además en sistemas grandes con muchos interesados muchas veces sucede que varios interesados tienen necesidades distintas y muchas veces inconsistentes. Estas inconsistencias muchas veces no son obvias cuando se escriben los requisitos, y son descubiertas luego de hacer un análisis en profundidad, o durante la etapa de desarrollo.

4.2.2. Requisitos no funcionales

Los requisitos no funcionales, como el nombre sugiere, son requisitos que no se preocupan directamente por los servicios específicos del sistema. Usualmente especifican o restringen características del sistema como un todo. Por ejemplo confiabilidad, tiempo de respuesta, capacidad de dispositivos de entrada/salida, representaciones del sistema.

Los requisitos no funcionales son muchas veces más críticos que los funcionales. Cuando un requisito funcional no cumple las necesidades del cliente, este puede encontrar alternativas para lograr el mismo objetivo. Sin embargo, cuando no se cumple un requisito no funcional, el sistema puede resultar inútil.

Implementación de requisitos no funcionales

En general es posible identificar los componentes que implementan un requisito funcional específico. Es mucho más difícil identificar los componentes que implementan un requisito no funcional. Esto se debe principalmente a dos razones:

- Los requisitos no funcionales pueden afectar a la arquitectura general del sistema en lugar de a componentes individuales.
- Un único requisito no funcional puede generar varios requisitos funcionales que definen servicios requeridos para el sistema.

Tipos de requisitos no funcionales

- **Requisitos de producto:** Especifican el comportamiento del producto. Por ejemplo la velocidad de ejecución, confiabilidad, etc.
- **Requisitos organizativos:** Son consecuencia de las políticas y procedimientos de la organización, por ejemplo estándares de procesos utilizados, requisitos de implementación, etc.
- **Requisitos externos:** Surgen de factores externos al sistema y su proceso de desarrollo, por ejemplo requisitos de interoperabilidad, requisitos legislativos, etc.

Requisitos no funcionales verificables

Los requisitos no funcionales pueden ser muy difíciles de declarar de forma precisa y los requisitos imprecisos pueden ser difíciles de verificar.

- Objetivos: una intención general del usuario como la facilidad de uso.
- Requisito no funcional verificable: una declaración con alguna medida que pueda ser probada objetivamente.

Los objetivos son útiles para los desarrolladores ya que transmiten las intenciones de los usuarios al sistema. Cuando sea posible, se deberían escribir los requisitos no funcionales de forma cuantitativa para que pueda ser testeado. En la práctica muchas veces es complicado traducir objetivos en requisitos verificables.

4.3. Problemas en los requisitos

La mayor consecuencia es el retrabajo (en etapas más avanzadas del desarrollo o después de liberar). Ejemplos de estos problemas son:

- Poco involucramiento de los usuarios.
- Planes inadecuados - utilizar requisitos muy vagos para crear planes.
- Recortes en los requisitos del usuario.
- Requisitos ambiguos.
- Gold plating: requisitos que creemos que el usuario va a amar. Cosas que no nos piden pero hacemos porque nos parecen buenas.
- No identificar correctamente a los usuarios correctos.

4.4. Procesos de la ingeniería de requisitos

Varían ampliamente dependiendo del dominio de la aplicación, personas involucradas y la organización que desarrolla los requisitos. Sin embargo hay una serie de actividades genéricas que son comunes a todos los procesos:

- **Estudio de factibilidad.**

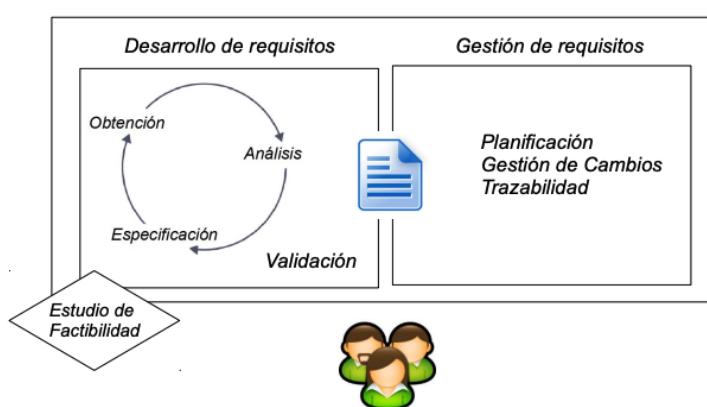
Tiene como objetivo averiguar si vale la pena implementar el sistema y si es posible implementarlo dadas las restricciones existentes.

- **Relevamiento y análisis de requisitos.**

- **Validación de requisitos.**

- **Gestión de requisitos.**

En la práctica la ingeniería de requisitos es una actividad iterativa en la cual se intercalan los procesos.



4.5. Obtención de requisitos

4.5.1. Dificultades comunes en el proceso

- Los stakeholders no saben lo que realmente quieren.
- Los stakeholders expresan los requisitos en sus propios términos.
- Diferentes stakeholders pueden tener conflictos con sus requisitos.
- Factores organizacionales y políticos pueden influir en los requisitos del sistema.
- Los requisitos cambian durante el proceso de análisis. Pueden surgir nuevos stakeholders y el entorno empresarial puede cambiar.

4.5.2. Actividades del proceso

- **Descubrimiento y entendimiento de requisitos**

Interactuar con los stakeholders para recolectar sus requisitos

- **Clasificación y organización de los requisitos**

Los requisitos que están relacionados se ponen en un mismo grupo y luego se organizan en subgrupos coherentes.

- **Asignación de prioridades y negociación**

Priorizar requisitos y resolver los requisitos en conflictos.

- **Especificación de requisitos**

Los requisitos se documentan y se ingresan a la próxima ronda del espiral.

4.6. Técnicas de obtención de requisitos

Las fuentes de requisitos pueden ser metas, conocimiento del dominio, los propios stakeholders, reglas de negocio, ambiente operacional, ambiente organizacional, etc. Hay muchas técnicas para obtenerlos, las cuales profundizaremos a continuación.

4.6.1. Entrevistas

La manera más obvia de averiguar que es lo que necesitan los usuarios es preguntarles. En las entrevistas, el equipo de ingeniería de requisitos le hacen preguntas a los stakeholders sobre el sistema que usan actualmente y el que se va a desarrollar. Las entrevistas son usadas para:

- Entender el problema de negocio.
- Entender el ambiente de operación.
- Evitar omisión de requisitos.
- Mejorar las relaciones con el cliente.

Tienen tanto ventajas como desventajas. Como ventajas se destaca que están orientadas a las personas, es interactivo y flexible y que puede ser rico en contenido. Sin embargo, son costosas y dependen de habilidades interpersonales.

Podemos distinguir dos tipos de entrevistas:

- Entrevistas cerradas basadas en una lista de preguntas pre-determinadas.
- Entrevistas abiertas donde varios temas son explorados con los stakeholders.

En la práctica las entrevistas son mayormente una mezcla de ambos tipos. Son buenas para conseguir una comprensión global de que quieren los stakeholders y como podrían interactuar con el sistema. Sin embargo, no son buenas para entender los requisitos de dominio debido a dos razones: en primer lugar, los ingenieros de requisitos no pueden entender la terminología específica del dominio. En segundo lugar, para algunas personas algunos de los conocimientos de dominio son tan familiares que les resulta difícil explicarlos o piensan que no vale la pena hacerlo.

Para que la entrevista sea efectiva se debe

- Tener la mente abierta, evitar ideas preconcebidas acerca de los requisitos y estar dispuesto a escuchar a los stakeholders.
- Utilizando una pregunta como trampolín se consiguen discusiones con el entrevistado para lograr una propuesta de requisitos o trabajar juntos en un prototipo del sistema.

Las siguientes son posibles preguntas que pueden hacerse en una entrevista:

- ¿Dónde se inicia el proceso?
- ¿Cómo se va a utilizar la funcionalidad?
- ¿Qué es necesario que cumpla la funcionalidad?
- ¿A quién le puedo preguntar para aprender más sobre esto?
- ¿En qué casos se puede utilizar la funcionalidad? ¿En qué casos no?
- ¿Hay otras maneras de realizar lo mismo?
- ¿La funcionalidad cubre todas las necesidades de negocio relacionadas?

Se plantean las siguientes recomendaciones para que la entrevista sea efectiva:

- Establecer una buena relación: presentarse, revisar agenda, recordar los objetivos de la sesión y responder dudas.
- Mantener el foco de la discusión.
- Preparar preguntas y maquetas previamente.
- Sugerir ideas.
- Escuchar activamente, mostrar paciencia, dar feedback y consultar puntos confusos.

4.6.2. Investigar antecedentes

Es una buena forma de comenzar un proyecto. Incluye estudio, muestreo, visitas, etc. Tiene como ventajas que ahorra el tiempo de otros, prepara para otros enfoques y puede llevarse a cabo fuera de la organización. Sin embargo la perspectiva puede ser limitada, puede obtenerse datos desactualizados y puede ser demasiado genérico.

4.6.3. Workshops

Son reuniones estructuradas en las cuales un selecto grupo de interesados y expertos trabajan en conjunto para definir, crear, refinar y acordar documentos y modelos que representan los requisitos del usuario. En general es necesario el rol de facilitador el cual explica las reglas y los objetivos al comienzo y luego mantiene la dinámica de trabajo y el foco en los objetivos. Debe también mantener la motivación de los participantes.

Se recomienda mantener grupos pequeños, utilizar tiempos fijos para discutir cada tema, manejar una lista de temas que surjan para tratarlos después y no perder el foco. Este tipo de técnica tiene como ventaja que es muy efectiva para resolver desacuerdos. Sin embargo son costosos, tanto en tiempo como en recursos.

4.6.4. Focus groups

Son grupos de usuario que participan en una actividad de obtención de requisitos para generar contribuciones e ideas sobre los requisitos funcionales y de calidad de un producto. Es necesario contar con un rol de facilitador así como hacer una buena selección de los participantes del grupo.

Tiene como ventajas que son útiles para explorar actitudes, impresiones, preferencias y necesidades de los usuarios y que son en particular valiosas cuando no hay fácil acceso a los usuarios finales. Sin embargo no hay que esperar resultados cuantitativos sino información subjetiva que sirve para evaluar y priorizar requisitos.

4.6.5. Prototipado

Es una implementación parcial que permite a los desarrolladores y usuarios entender mejor los requisitos, cuales son necesarios y cuales deseables, acotar riesgos, etc.

If a picture is worth 1000 words, then a prototype is worth a 1000 meetings

Alcance

- **Mock-ups / Bosquejos**

Prototipos horizontales. Se enfocan en porciones de la interfaz de usuario. Permiten explorar comportamientos específicos del producto. No realizan ningún trabajo útil, solo lucen como si lo hicieran.

- **Pruebas de concepto**

Prototipos verticales. Implementan una porción funcional de la aplicación. Permiten resolver incertidumbres sobre la factibilidad de la arquitectura propuesta u otros riesgos técnicos. Funcionan como el sistema real porque toca todos los niveles de la implementación.

Uso futuro

- **Desechables**

Sirven para responder preguntas, resolver incertidumbres y mejorar la calidad de los requisitos. Conviene hacerlos lo más rápido y baratos posibles resistiendo la tentación de hacerlos más elaborados de lo necesario. Hay que tener mucho cuidado con la calidad al incorporar algo de un prototipo desecharable al producto final. Ejemplos de esto son los wireframes.

- **Evolutivos**

Proveen una base arquitectónica sólida para desarrollar el producto de forma incremental mientras los requisitos se vuelvan más claros con el tiempo. Las metodologías ágiles proveen un ejemplo de prototipación evolutiva. Los equipos ágiles construyen el producto a lo largo de una serie de iteraciones, utilizando feedback para ajustar los siguientes ciclos de desarrollo. Es importante que desde un principio sean construidos con calidad en el código y deben ser diseñados para contemplar un rápido crecimiento y mejora frecuente. Son una buena elección para aplicaciones que sabemos que crecerán a lo largo del tiempo.

Forma

- **Prototipos en papel**

Son prototipos de baja fidelidad. Permiten de forma barata, rápida y de baja tecnología explorar cómo va a lucir una parte del producto. La idea es explorar posibles alternativas de comportamiento y estética del producto sin perderse mucho en los detalles. Facilitan una rápida interacción y se utilizan para refinar los requisitos antes de diseñar la interfaz de usuario. Ejemplos de esto son los storyboards.

- **Prototipos electrónicos**

Son prototipos de alta fidelidad. Se realizan con herramientas digitales de prototipado y simulación. Mediante la interacción con una versión muy similar en estética o comportamiento, provee un mecanismo muy valioso para clarificar los requisitos y estudiar el comportamiento de los usuarios.

Una vez de entrega un prototipo es importante evaluarlo. **La evaluación de prototipos** está relacionada a las pruebas de usabilidad. Se aprende más observando a los usuarios trabajando con el prototipo que preguntando qué piensan de él. Para esto, hay que validar los prototipos con un público adecuado. Para mejorar la evaluación se pueden usar guías. Sin embargo hay que evitar guiar demasiado y dejar que cada usuario lo utilice como normalmente lo haría. En base a esto se documenta para ser utilizado en futuras actividades.

Los siguientes son posibles riesgos que presentan los prototipos:

- Que el cliente piense que el producto ya está pronto, o que quiera una versión para probarlo fuera de la instancia de evaluación.
- Perderse en los detalles
- Generar expectativas irreales con respecto al producto final.
- Invertir demasiado esfuerzo en los prototipos.

Para tener éxito con los prototipos es importante

- Incluirlos en el plan de trabajo.
- Establecer los objetivos previo a su construcción.
- Planificar la construcción de varios. A veces es necesario hacer más de una versión de un prototipo.
- Crear prototipos desecharables tan rápido y barato como sea posible.
- No incluir muchos detalles como validaciones o manejo de errores en prototipos descartables.
- No prototipar requisitos que no se entendieron todavía.
- Usar datos cercanos a la realidad.
- No esperar realizar prototipos en vez de escribir requisitos.

4.6.6. Observaciones - Etnografías

Implican observar a los usuarios mientras realizan sus actividades. Pueden ser silenciosas o interactivas. En general los usuarios no son muy precisos al describir sus tareas. Esto puede ser porque son tareas complejas y difíciles de explicar. En otros casos, están tan familiarizados que omiten detalles de forma inconsciente.

Tiene como ventajas que es confiable, muy rico y desarrolla empatía. Sin embargo tiene como desventaja que es muy costoso, puede provocar el efecto Hawthorne (las personas actúan diferente cuando se las está observando) y hay que tener cuidado al generalizar.

4.6.7. Cuestionarios / Encuestas

Son una manera de estudiar grandes grupos de usuarios para entender sus necesidades. Antes de usar este enfoque debemos determinar la información que se precisa, determinar el enfoque más adecuado (abierto, cerrado, combinado, múltiple opción, valor en escala, orden relativo), desarrollar un cuestionario, probarlo con un perfil típico y analizar resultados de las pruebas. Su principal uso es para validar y obtener datos estadísticos sobre preferencias.

El mayor desafío es preparar preguntas bien escritas. Se dan las siguientes recomendaciones:

- Proveer opciones para todas las posibles respuestas.
- Hacer que las opciones sean mutuo-excluyentes.
- Utilizar preguntas cerradas para análisis estadístico y abiertas para recolectar ideas o necesidades nuevas.
- Siempre probar el cuestionario antes de usarlo.
- No incluir demasiadas preguntas.

Tiene como ventaja que es económico de escalar, es conveniente para quién la contesta y las respuestas son anónimas. Sin embargo tiene como desventaja que es menos rico (no se puede indagar en una respuesta), tiene problemas por no respuesta y tiene cierto esfuerzo de desarrollo.

4.6.8. Análisis de las interfaces del sistema

Implica examinar los otros sistemas con los que se conecta el sistema. Revela requisitos funcionales relativas al intercambio de datos y servicios del sistema. Para cada sistema que se deba comunicar con el nuestro se identifican las funcionalidades que nos puedan generar requisitos. Esos requisitos pueden describir los datos a pasar a otros sistemas, los datos a recibir de otros sistemas y las reglas sobre los datos (por ejemplo criterios de validación). Es útil para revisar las validaciones de la información a comunicar o recibir.

4.6.9. Análisis de la interfaz de usuario

Implica estudiar sistemas existentes para determinar requisitos de usuario y funcionales. Lo mejor es utilizar sistemas existentes, pero si no se pueden utilizar screenshots (por ejemplo de manuales de usuario).

Puede ayudar a identificar una lista completa de pantallas y a descubrir características potenciales del nuevo sistema. Se utiliza para identificar pasos comunes de los usuarios al realizar tareas así como para crear borradores de casos de uso. No hay que asumir que una funcionalidad es necesaria porque se encuentra en un sistema existente. O mantener la interfaz de usuario parecida o que se comporte de forma similar a la estudiada.

4.6.10. Análisis de documentación

Contempla examinar toda la documentación existente en busca de requisitos potenciales del software. La documentación más útil incluye especificación de requisitos, procesos de negocio, lecciones aprendidas y manuales de usuario de aplicaciones existentes o similares. Es una forma rápida de introducirse rápidamente en un nuevo dominio o en un sistema existente.

4.6.11. Tormenta de ideas

Ayuda a la participación de todos los involucrados. Tiene como reglas que no se permite criticar ni debatir y se busca generar tantas ideas como sea posible, mutar y combinar ideas.

Tiene una fase de generación donde los principales stakeholders se reúnen y se establecen objetivos. Se pide que cada uno escriba sus ideas.

Luego tiene una fase de reducción donde se leen las ideas y se establece si es válida. Se agrupan ideas y se hacen las definiciones necesarias. Opcionalmente se priorizan.

4.6.12. Historias y escenarios

Las historias y los escenarios describen cómo se puede utilizar el sistema para una tarea particular. Describen qué hacen las personas, qué información usan y producen y qué sistemas utilizan en ese proceso. Las historias se escriben como un texto narrativo que describen a alto nivel el uso del sistema, los escenarios en general tienen información estructurada como por ejemplo:

- Una descripción de la situación de partida.
- Una descripción del flujo normal de los eventos
- Una descripción de lo que puede salir mal.
- Información sobre otras actividades concurrentes.
- Una descripción del estado cuando finaliza el escenario.

4.6.13. Historias de usuario

Refieren a descripciones cortas y de alto nivel de las funcionalidades expresadas en los términos del cliente. Una historia de usuario típica tiene la forma: “Como <rol>, quiero <objetivo> para <beneficio>”. Pretenden contener justo la información necesaria para que los desarrolladores puedan producir una estimación razonable del esfuerzo para su implementación. La idea es evitar perder demasiado tiempo relevando detalles de requisitos que luego cambian demasiado o son desestimados.

4.6.14. Modelado de Procesos

Permiten entender el trabajo con múltiples pasos, roles o departamentos. Es iniciado por un evento e incluyen actividades manuales, automáticas o combinaciones de ambos tipos.

Pueden volverse complejos y pesados si no se estructuran con cuidado. Los procesos complejos se pueden descomponer para ayudar su entendimiento.

4.6.15. Casos de uso

Un caso de uso describe una secuencia de interacciones entre un sistema y un actor externo que resultan en un resultado de valor para el actor. Cada escenario comprende una instancia de uso del sistema. Un caso de uso contempla un conjunto de escenarios relacionados. Son independientes del método de diseño y del lenguaje que se utilice para la implementación.

Un conjunto de casos de uso debe describir todas las posibles interacciones con el sistema. Se identifica cada tipo de interacción entre los distintos actores y el sistema y se les da un nombre.

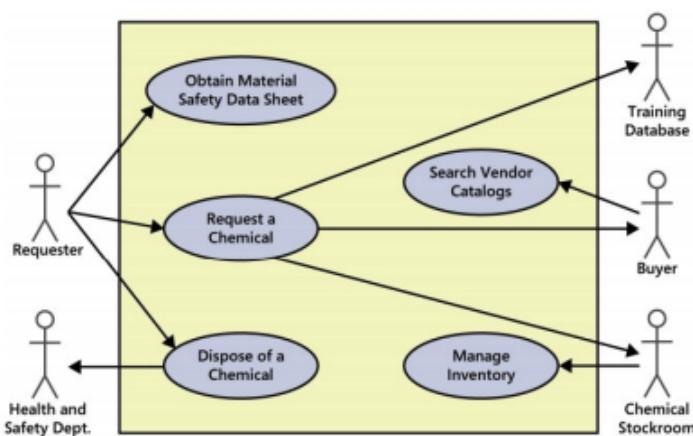
Tiene como ventaja que permiten ver fácilmente la interacción del sistema con los demás actores y que son muy útiles cuando la implementación va a ser OO. Tiene como desventaja que no son muy útiles para describir sistemas sin usuarios y no modelan bien requisitos no funcionales y restricciones de diseño.

Actores

Un actor es una persona (o quizás otro sistema) que interactúa con el sistema para realizar un caso de uso. Observar que nuestro sistema NO es un actor.

Diagrama de Casos de Uso

El diagrama de casos de uso provee una representación de alto nivel de los requisitos de usuario.



Elementos esenciales de un Caso de Uso

- Identificador único
- Breve descripción
Es una descripción breve, a alto nivel, del caso de uso y del flujo de acciones que comprende.
- Precondiciones
Describen condiciones que debe cumplir el sistema para que se pueda iniciar el caso de uso.
- Poscondiciones
Las poscondiciones describen el estado del sistema luego de la ejecución exitosa del caso de uso.
- Una lista enumerada de pasos que muestran la secuencia de interacciones entre el actor y el sistema.

Flujo normal y alternativo

El flujo principal muestra la secuencia de pasos más común que se lleva a cabo para la ejecución del caso de uso. Además del flujo principal, un caso de uso puede tener varios flujos alternativos. Un flujo alternativo describe un escenario alternativo al principal (casos particulares y manejo de excepciones/errores).

Casos de uso expandidos

Usualmente se crea una primera versión del caso de uso que no contempla los flujos normales y alternativos, sino que incluye únicamente descripción, precondiciones y poscondiciones. A este estilo de escritura se le llama **alto nivel**. En estos casos a la versión que incluye los flujos normales y alternativos se le llama **caso de uso expandido**.

Relaciones entre casos de uso

Inclusión: Permiten definir casos de uso que “ejecutan” otros casos de uso. Al caso de uso que incluye a otro se le llama caso base y al otro se le denomina caso incluido. Se puede incluir otros casos de uso tanto en el flujo principal como en algún flujo alternativo.

Es como ejecutar la totalidad del caso de uso incluido y luego continuar en el punto siguiente del caso base. Sólo se continúa la ejecución del caso base si el caso incluido se ejecutó con éxito.

1. El usuario ingresa al sitio.
2. <<incluye>> “Autentificar”
3. El sistema le ofrece un menú de servicios.
4. Usuario: indica al sistema que desea realizar una transferencia
5. Sistema: lista las cuentas del usuario
6. ...

Extensión: La relación de extensión refiere a un fragmento de un caso de uso que extiende, agrega comportamiento, a otro caso de uso. Se utilizan para describir escenarios alternativos complejos. Una extensión sólo se ejecuta cuando se cumple una condición particular en un punto específico del caso de uso a extender. A ese punto se le llama punto de extensión.

Estrategias para identificar Casos de Uso

- Identificar primero a los actores, luego a los procesos que serán soportados por el sistema y por último definir casos de uso para las actividades en las cuales interactúan los actores y el sistema.
- Crear escenarios específicos para ilustrar cada proceso de negocio, luego generalizar en casos de uso e identificar actores.
- Identificar los eventos externos a los cuales el sistema debe responder, luego relacionar esos eventos a los actores participantes y casos de uso específicos.
- Identificar que entidades de datos necesitan casos de uso de creación, lectura, actualización, borrado u otras operaciones de manipulación.

Consideraciones

- No tratar de forzar a que todos los requisitos sean contemplados por los casos de uso. En algunos casos no aporta demasiado valor hacerlo
- Ser concretos y mantenerse dentro de los límites de los casos de uso (precondiciones y poscondiciones).
- Evitar escribir demasiados casos de uso. Evaluar si son todos necesarios y si no corresponden a distintos escenarios de un mismo caso de uso.
- No crear casos de uso muy complejos. En particular, no incluir demasiados chequeos de errores y validaciones comunes.
- Los casos de uso no tienen porque desarrollarse por completo de una vez. En particular, no utilizar demasiado tiempo en detallar casos de uso que serán utilizados como insumo para otras actividades que comienzan dentro de meses o años.
- No se deberían incluir en los flujos detalles de la interfaz gráfica. Pero pueden agregarse secciones, por ejemplo, para incluir borradores de pantallas.
- No se debería incluir información de diseño o desarrollo. Así tampoco información sobre la definición de los datos.
- No escribir casos de uso que no sean entendidos por los usuarios.

4.7. Selección de técnicas a usar

En general es necesario usar más de una técnica y la elección está muy relacionada al tipo de proyecto. Sugerencias (Wiegers)

	Interviews	Workshops	Focus groups	Observations	Questionnaires	System interface analysis	User interface analysis	Document analysis
Mass-market software	x		x	x				
Internal corporate software	x	x	x	x	x		x	
Replacing existing system	x	x		x	x	x	x	
Enhancing existing system	x	x			x	x	x	
New application	x	x			x			
Packaged software implementation	x	x		x	x	x		
Embedded systems	x	x			x		x	
Geographically distributed stakeholders	x	x		x				

4.8. Análisis de requisitos

El análisis se da naturalmente junto a las actividades de obtener y especificar los requisitos. Conceptualmente implica:

- Detectar y resolver conflictos entre los requisitos.
- Descubrir las fronteras del software y cómo debe interactuar con los ambientes organizacional y operacional.
- Elaborar los requisitos del sistema para derivar los requisitos del software.

Las actividades del análisis de requisitos incluye:

- **Clasificación de requisitos**
Clasificar de acuerdo a categorías. Priorizar, ver alcance y estabilidad de los requisitos.
- **Modelo conceptual**
Realizar modelos resulta clave para el análisis de requisitos. Ayudan a entender la situación en la cual ocurren los problemas o necesidades, así como representar soluciones.
- **Diseño de la arquitectura y asignación de requisitos**
- **Negociación de requisitos**
Resolución de conflictos. Priorización.
- **Análisis formal**
Muy recomendado para ciertos dominios de aplicación.

4.9. Especificación de requisitos

Es el proceso de escribir los requisitos de usuario y del sistema en un documento de requisitos. Los requisitos de usuario deben ser entendidos por los usuarios finales y los clientes que no tienen formación técnica. Los requisitos de sistema son más detallados y pueden incluir más información técnica. Se debe intentar que sean lo más completos posibles.

4.9.1. Formas de escribir una especificación de requisitos

- **Lenguaje natural**
Los requisitos son escritos usando frases numeradas escritas en lenguaje natural. Cada frase debe expresar un requisito. Se utiliza para escribir los requisitos porque es expresivo, intuitivo y universal. Esto significa que los requisitos pueden ser entendidos por los usuarios y los clientes.

Guía para escribir requisitos

- Crear un formato estándar y usarlo para todos los requisitos.
- Usar un lenguaje de manera consistente
- Utilizar texto resaltado (subrayar, negrita, colores) para identificar las partes claves de los requisitos.
- Evitar el uso de jerga informática.
- Incluir una explicación lógica de porqué es necesario un requisito.

Problemas con el lenguaje natural

- Falta de claridad: es difícil la precisión sin hacer que el documento se vuelva difícil de leer.
- Confusión de requisitos: los requisitos funcionales y no funcionales tienden a mezclarse.
- Requisitos mezclados: varios requisitos diferentes pueden ser expresados juntos.

■ **Lenguaje natural estructurado**

Es una forma de escribir requisitos donde estos se escriben en una forma estándar en lugar de texto libre. De esta forma, se mantiene la mayor parte de la expresividad y entendimiento del lenguaje natural, pero se asegura uniformidad en la especificación. A su vez se reduce la variabilidad en la especificación, y los requisitos se organizan de forma más efectiva.

Posibles secciones

1. Descripción de la función o entidad.
2. Descripción de las entradas y de donde vienen.
3. Descripción de las salidas y a donde van.
4. Información sobre los datos necesarios para el cálculo y otras entidades usadas.
5. Descripción de las acciones a tomar.
6. Pre y post condiciones en caso de ser necesario.
7. Los efectos secundarios de las operaciones.

Muchas veces, sigue siendo difícil escribir los requerimientos de una forma clara y sin ambigüedades, particularmente cuando se deben especificar cálculos complejos. Para enfrentar este problema se puede agregar información adicional para complementar el lenguaje natural, por ejemplo utilizando tablas o modelos gráficos. Las tablas son particularmente útiles

■ **Lenguajes de descripción de diseño**

Este enfoque usa un lenguaje como un lenguaje de programación pero con características más abstractas para especificar los requisitos mediante la definición de un modelo operativo del sistema. Ahora este enfoque es raramente usado aunque puede ser útil para las especificaciones de interfaz.

■ **Notaciones gráficas**

Modelos gráficos complementados por anotaciones de texto son usados para definir los requisitos funcionales para el sistema. Comúnmente se utilizan diagramas UML de casos de uso y diagramas de secuencia.

■ **Especificaciones matemáticas**

Estas notaciones son basadas en conceptos matemáticos tales como máquinas de estado finitos o conjuntos. Aunque estas especificaciones pueden reducir la ambigüedad la mayoría de los clientes no entienden la especificación formal. Ellos no pueden chequear que se representa lo que ellos quieren y se resisten a aceptarlo como un contrato del sistema.

4.9.2. Características deseables según Wiegers

Características deseables de un requisito según Wiegers

- **Completo:** contiene toda la información para entenderlo.
- **Correcto:** describe de forma precisa una necesidad y de forma clara la funcionalidad a desarrollar para cumplirla.
- **Factible:** es posible implementarlo dada las capacidades y limitaciones del sistema y su ambiente operacional.
- **Necesario:** tiene valor para todo el negocio.
- **Priorizado:** fue priorizado según su importancia utilizando perspectiva de todos los interesados.
- **No ambiguo:** no admite múltiples interpelaciones.
- **Verificable:** se pueden realizar pruebas para determinar si se encuentra presente en un producto de software o no.

Características deseables de un conjunto de requisitos según Wiegers

- **Completo:** no hay información necesaria o de requisitos ausente.
- **Consistente:** no hay requisitos en conflicto con otros.
- **Modificable:** es posible reescribir un requisito.
- **Trazable:** un requisito puede ser rastreado hacia atrás a su origen y hacia adelante a elementos de diseño, código implementado y pruebas que verifican su implementación.

4.9.3. Guía para la escritura de requisitos

▪ **Perspectiva del sistema o del usuario**

- Sistema: [precondición opcional] [evento disparador opcional] el sistema debe [respuesta esperada del sistema]
- Usuario El [nombre del actor o clase de usuario] debe ser capaz de [hacer algo] [por algún motivo] [condiciones, tiempo de respuestas]

▪ **Estilo de escritura**

- Tratar de incluir la frase clave al principio de cada requisito, y luego los detalles accesorios.
- Evitar utilizar voces activas y pasivas de forma intercalada. Es más, es mejor usar la voz activa.
- No utilizar múltiples términos para el mismo concepto.
- Escribir frases completas y utilizar correctas gramática, ortografía y puntuación.
- Mantener las oraciones y los párrafos cortos y directos.
- Evitar escribir largos párrafos con más de un requisito.

▪ **Nivel de detalle**

No todos los requisitos tienen que tener el mismo nivel de detalle. Aunque se recomienda mantener el mismo nivel en requisitos relacionados.

Se debería incluir más detalle sí:

- El trabajo será hecho para un cliente externo.
- El desarrollo o las pruebas serán tercerizados.
- Los miembros del equipo del proyecto están dispersos.
- Las pruebas del sistema se basan en requisitos.

- Son necesarias estimaciones precisas.
- La trazabilidad de los requisitos es importante.

Se puede incluir menos detalle cuando:

- Los clientes están muy involucrados.
- Los desarrolladores tienen mucha experiencia en el dominio.
- Se dispone de precedentes.
- Se usará una solución empaquetada.

■ Evitar la ambigüedad

- Utilizar términos de forma consistente y como están definidos en el glosario.
- Evitar adverbios (generalmente, rápidamente, etc).
- Tratar de escribir todos los requisitos de forma positiva

■ Evitar incompletitud

- Incluir operaciones simétricas (guardar borrador/recuperar)
- Revisar en búsqueda de excepciones que falten.

4.10. Ingeniería de Requisitos en Procesos Ágiles

Generalmente los requisitos o características del producto se especifican como Historia de Usuario.

4.10.1. Historias de Usuario (HU)

Una HU describe una funcionalidad que por si misma aporta valor al usuario. Es una representación de un requisito, escrito de manera simple, sencilla y fácil de entender. Son una forma rápida de administrar los requisitos cambiantes de un proyecto.

Una HU se compone de tres elementos:

- Card (Ficha): Descripción breve de la historia de usuario, utilizada como recordatorio y para planificar.
- Conversación: Comunicación cara a cara que intercambia no solo información sino también pensamientos, opiniones y sentimientos.
- Confirmación: Detalles de la historia de usuario para que el equipo sepa lo que tienen que construir y lo que la contra-parte espera. Se conoce como Criterios de Aceptación.

4.10.2. Redacción de HU (Cards)

Mike Cohn sugiere redactar las HU siguiendo el siguiente formato:

Como [rol] quiero [funcionalidad] para [beneficio]

Tiene como beneficios que está en primera persona, son fáciles de priorizar y permite al equipo de desarrollo plantear alternativas que cumplan con el mismo propósito cuando el costo de la funcionalidad sea alto o su construcción sea no viable.

Cuando una historia de usuario es demasiado grande se la denomina **épica**. Las clasificamos en una de las siguientes categorías:

- Historia de usuario compuesta: contiene múltiples historias de usuario.
- Historia de usuario compleja: no se puede separar fácilmente en un conjunto de historias de usuario. Si es compleja debido a que existe una incertidumbre asociada a la misma, se puede dividir en dos historias: una para investigar y otra para desarrollar la funcionalidad.

4.10.3. Redacción de HU (Confirmación)

La confirmación se realiza mediante criterios de aceptación, los cuales ayudan a entender mejor cómo se espera que el producto se comporte frente a ciertas situaciones. Ayudan a resolver ciertas expectativas y permiten que el desarrollo sea más fluido, claro y con menor incertidumbre para el equipo.

Por ejemplo si tenemos la siguiente HU: “*Como cliente habitual quiero ver un listado de locales de comida con delivery para evaluar las distintas propuestas*”. Los siguientes son ejemplos de CA:

- Los locales de comida con delivery están ordenados según valoración (mejor valoración primero) y la ubicación que se encuentra el usuario.
- Para cada local de comida con delivery se muestra: nombre, foto, calificación, distancia con respecto a la ubicación del usuario y tiempo estimado de entrega.
- Si el usuario selecciona un local de comida con delivery, se despliegan las comidas que vende el local. De cada una de las comidas se muestra nombre, precio y una breve descripción.

De esta forma los criterios de aceptación aportan más contexto y sirven de guía para el desarrollo de los tests de aceptación. Deben cubrir tanto los casos comunes como los alternativos. La calidad de un criterio de aceptación eficaz se define bajo el método SMART Specific, Measurable, Achievable, Relevant, Time-bound (Temporalmente limitado).

Generalmente escribimos los criterios de aceptación utilizando una de las siguientes técnicas:

- **Técnicas de comportamiento:** Dada una condición, cuando ocurre un evento o acción, entonces sucederá una consecuencia. Así se consigue una estructura consistente que se trasladará a los tests automáticos.
- **Técnicas de escenarios:** Suele definir el escenario normal o usual y un escenario alternativo de la funcionalidad en cuestión, y debe describir cómo el usuario ejecutaría o intentaría ejecutar diferentes pasos en dichos trayectos. La forma más utilizada de escribirlos es mediante la sintaxis de gherkin la cual tiene la siguiente forma:

Dado que [Contexto] y adicionalmente [Contexto] cuando [Evento], entonces [Resultado/Comportamiento Esperado]

Se recomienda que toda HU cumpla con 6 características que se pueden recordar bajo la regla mnemotécnica INVEST

- Independiente: se debe tener cuidado para evitar dependencias entre HUs.
- Negociable: los detalles de una historia de usuario se negociarán en una conversación entre el cliente y el equipo de desarrollo.
- Valiosa: valiosa para los clientes o usuarios del software.
- Estimable: para esto se debe evitar que las HU sean demasiado grandes. Puede influir en esto la falta de conocimiento funcional o la falta de conocimiento técnico.
- Small (Pequeña): cada una debe ser pequeña en esfuerzo.
- Testable (Verificable): una historia necesita poder probarse para poder ser confirmada.

Requisitos no funcionales

Newkirk y Martin recomiendan la práctica de anotar una historia con la palabra CONSTRAINT para cualquier historia que debe ser obedecida en lugar de implementada directamente.

Historias de Usuario tipo SPIKE

Refieren a tareas de investigación durante una iteración, por ejemplo: investigar, diseñar, explorar, comprender mejor un requisito, etc. Suelen ser de dos tipos: pueden ser técnicos (se utilizan para verificar la viabilidad) o funcionales (se utilizan para analizar funcionalidad, su riesgo y complejidad).

4.11. Modelado del sistema

El modelado del sistema es el proceso de desarrollar modelos abstractos del sistema. Con cada modelo se presentan diferentes visiones o perspectivas. Ayuda a los analistas a entender las funcionalidades del sistema y facilita la comunicación con los clientes. En la actualidad es una actividad que se basa generalmente en la notación UML.

4.11.1. Perspectiva del sistema

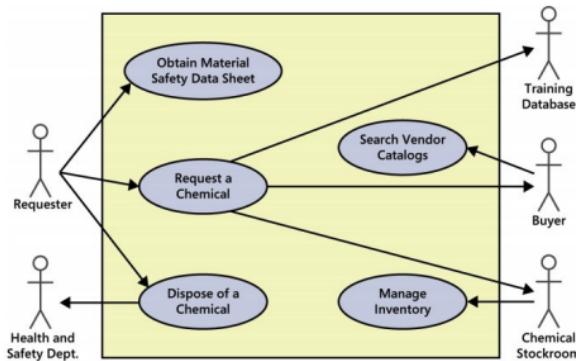
- **Perspectiva externa:** se modela el contexto o entorno del sistema.
- **Perspectiva de la interacción:** se modelan las interacciones entre un sistema y su entorno o entre un sistema y sus componentes.
- **Perspectiva estructural:** se modela la organización del sistema o la estructura de los datos que se procesan por el sistema.
- **Perspectiva del comportamiento:** se modela el comportamiento dinámico del sistema y la forma en que responde a eventos.

4.11.2. Modelos de contexto

Son usados para ilustrar el contexto operacional del sistema. Muestran lo que está fuera de los límites del sistema. Temas sociales y organizacionales pueden afectar la decisión sobre donde situar los límites del sistema.

Límites del sistema

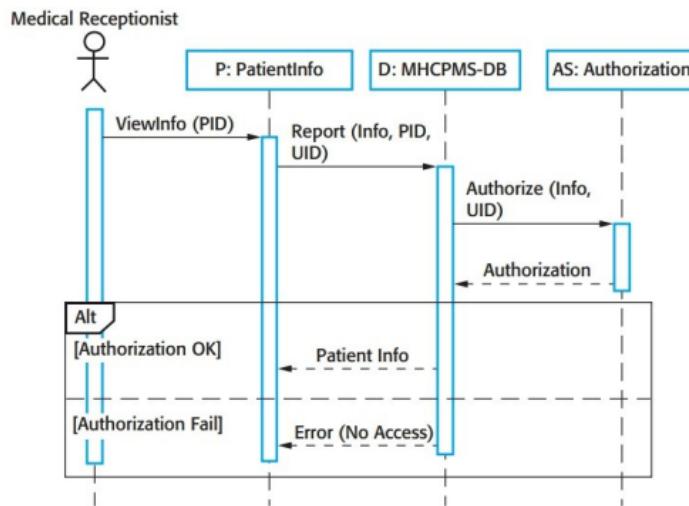
Definen que está adentro y que está afuera del sistema. La definición de los límites del sistema tiene un profundo efecto sobre los requisitos del sistema. Como modelo de contexto podemos utilizar un diagrama de casos de uso que incluye la frontera.



4.11.3. Modelos de interacción

Todo sistema involucra interacción: interacción con usuarios, con otros sistemas o entre sus componentes. Modelar la interacción con los usuarios ayuda a identificar los requisitos de usuario. Modelar la interacción con otros sistemas permite explorar posibles problemas de comunicación. Modelar la interacción entre los componentes ayuda a entender si la estructura propuesta del sistema es apropiada para los requisitos y atributos de calidad establecidos.

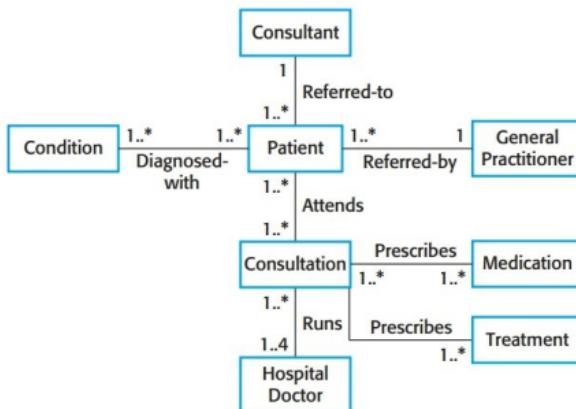
En UML podemos modelar la interacción de un sistema utilizando Diagramas de Secuencia. Estos diagramas son usados para modelar las interacciones entre los actores y los objetos dentro de un sistema. Muestra la secuencia de interacciones que tienen lugar durante un caso de uso particular o una instancia de caso de uso. Los objetos y actores involucrados son enumerados a lo largo de la parte superior del diagrama con una línea vertical de puntos a partir desde allí hacia abajo. Las interacciones entre objetos son indicadas mediante flechas con anotaciones.



4.11.4. Modelos estructurales

Disponen la organización del sistema en términos de los componentes que lo componen y sus relaciones. Pueden ser modelos estáticos, los cuales muestran la estructura del sistema diseñado o modelos dinámicos los cuales muestran la organización del sistema cuando se está ejecutando. Se pueden crear modelos estructurales del sistema cuando se está discutiendo y diseñando la arquitectura del sistema.

Un ejemplo de modelo estructural son los Diagramas de clases. Estos diagramas muestran las clases de un sistema y sus asociaciones. Una clase puede ser considerada como una definición general de un tipo de objeto del sistema. Una asociación se representa como un enlace entre dos o más clases. Cuando se desarrollan modelos durante fases tempranas los objetos representan algo del mundo real (modelo conceptual).



4.11.5. Modelos de comportamiento

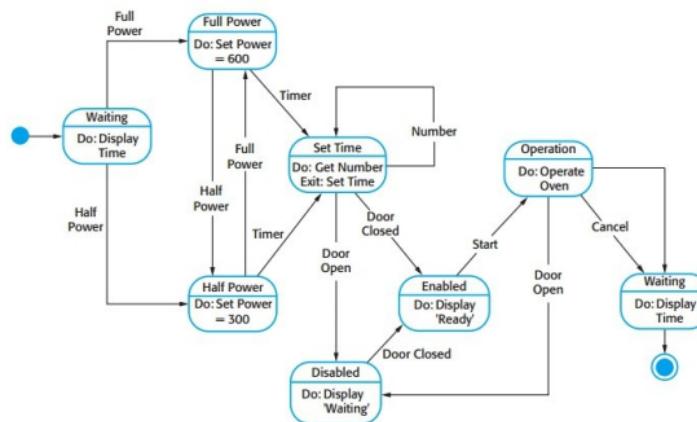
Modelan el comportamiento dinámico de un sistema en ejecución. Muestran qué sucede o qué debe suceder cuando un sistema responde a los estímulos de su entorno. Se puede pensar en dos tipos de estímulos:

- **Modelado dirigido por datos (data-driven)**

Muestran la secuencia de acciones involucradas en procesar los datos de entrada y generar una salida asociada. Los modelos de flujo de datos permiten realizar el seguimiento y documentar cómo los datos asociados a un proceso particular se mueven a través del sistema. UML no soporta de forma natural los modelos de flujo de datos pero pueden utilizarse diagramas de secuencia.

- Modelado dirigido por eventos (event-driven)

Muestran cómo un sistema responde a eventos externos e internos. Se basan en la suposición de que un sistema tiene un número finito de estados y que eventos (estímulos) pueden causar una transacción de un estado a otro. En UML se pueden utilizar diagramas de estado



4.11.6. Ingeniería dirigida por modelos (MDE)

Es un enfoque para el desarrollo de software donde los modelos (y no los programas) son los principales resultados del proceso de desarrollo. Los programas que se ejecutan en una plataforma de hardware/software se generan automáticamente a partir de los modelos. La MDE se encuentra todavía en una etapa temprana de desarrollo y no está claro si tendrá o no un efecto significativo en la práctica de la ingeniería de software.

4.11.7. Otros modelos para requisitos - Wiegers

- Los modelos visuales de requisitos ayudan a identificar requisitos faltantes, extraños o inconsistentes.
 - Los modelos sirven tanto para elaborar y explorar requisitos así como para diseñar soluciones de software.
 - No hay que asumir que los clientes saben cómo interpretar los modelos de análisis.
 - En general no es posible modelar todo el sistema, así que lo recomendable es enfocarse en las porciones del sistema más riesgosas.

Diagrama de flujo de datos

Provee una visión de cómo se mueven los datos a través de un sistema.

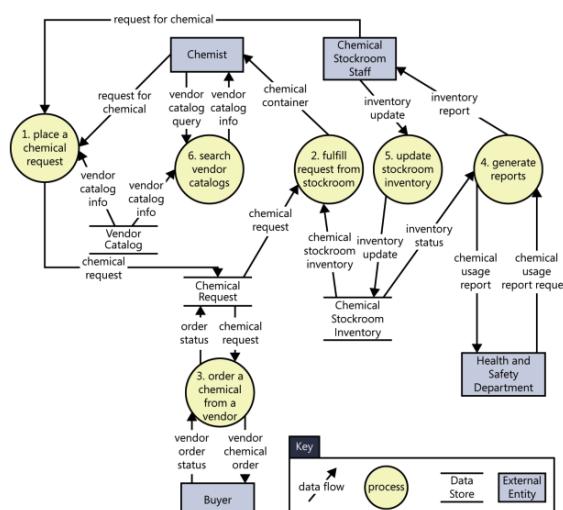


Diagrama de andariveles

Proveen una manera de representar los pasos involucrados en un proceso o las operaciones de un nuevo sistema de software. Los carriles representan actores u otros sistemas que ejecutan un paso del proceso. Es uno de los modelos más simples de entender por los clientes.

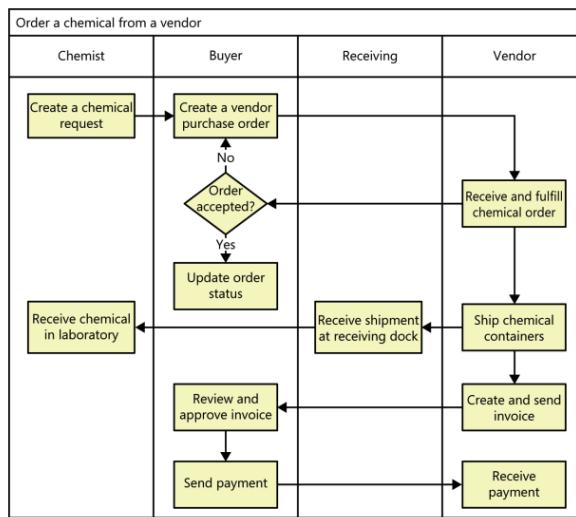
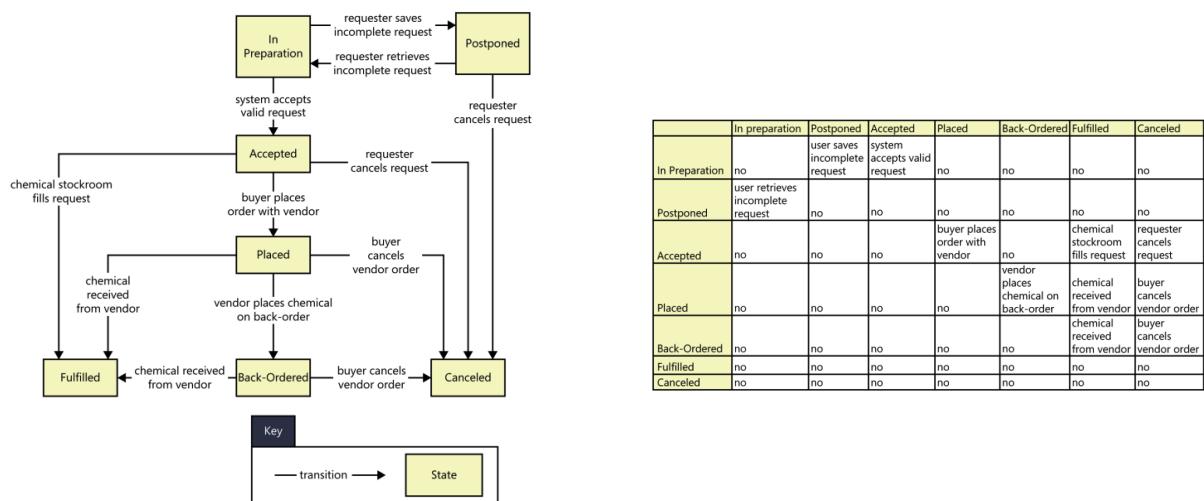


Diagrama de estados y transiciones y tablas de estados

Proveen una representación concisa, completa y no ambigua de los estados de un objeto o sistema. Los diagramas de estado contienen:

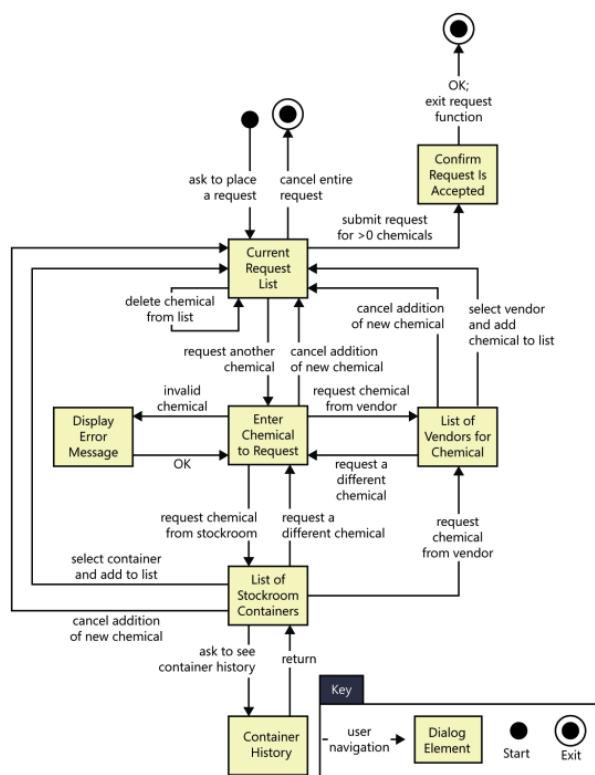
- Los posibles estados se representan como rectángulos.
- Las transiciones o los cambios de estado permitidos se representan como flechas entre dos rectángulos.
- Los eventos o condiciones que causan los cambios de estado se muestran como etiquetas de texto sobre las flechas de transición.

Por otro lado las tablas de estados muestran todas las posibles transiciones entre los estados en forma de una matriz.



Mapa de dialogo

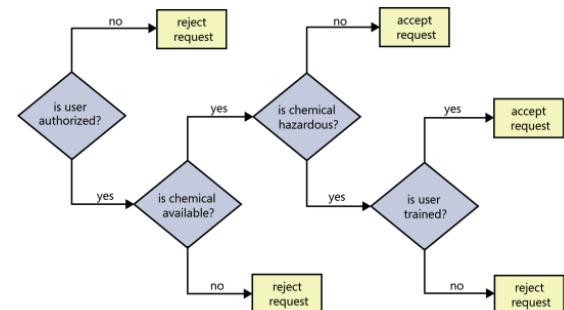
Un mapa de diálogo representa el diseño de la interfaz de usuario con un nivel alto de abstracción. La técnica permite modelar la interfaz de usuario en la forma de un diagrama de estados y transiciones.



Tablas y árboles de decisión

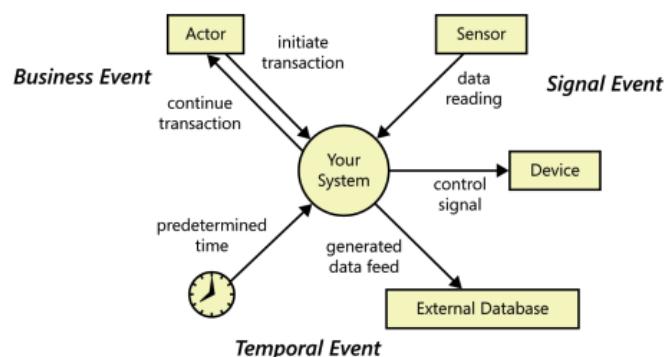
Las tablas y árboles de decisión son dos técnicas alternativas para representar qué debería hacer el sistema ante decisiones o lógica compleja. Una tabla de decisión lista los valores para todos los factores que influencian el comportamiento del sistema e indican la acción esperada del sistema en respuesta a cada combinación de factores.

Requirement Number	1	2	3	4	5
Condition	1	2	3	4	5
User is authorized	F	T	T	T	T
Chemical is available	—	F	T	T	T
Chemical is hazardous	—	—	F	T	T
Requester is trained	—	—	—	F	T
Action					
Accept request			X		X
Reject request	X	X		X	



Tablas de evento-respuesta

Permiten representar las respuestas del sistema a los posibles eventos que puedan ocurrir. Un evento es un cambio o actividad que tiene lugar en el ambiente del usuario y que estimula una respuesta del sistema.



4.12. Documento de requisitos de software

El documento de requisitos de software (SRS) es una declaración oficial de lo qué se requiere de los desarrolladores del sistema. Puede incluir una definición de los requisitos del usuario y una especificación de los requisitos del sistema. No es un documento de diseño. Se debe indicar lo que el sistema debe hacer y no la forma en la que lo debe hacer. Metodologías ágiles prefieren, en lugar de construir un documento formal, recolectar requisitos de forma separada. Por ejemplo: las historias de usuario en Extreme Programming (XP).

4.12.1. Usuarios de un documento de requisitos

- **Clientes del sistema:** Especifican los requisitos y los leen para chequear que cumplan sus necesidades. Los clientes especifican cambios en los requisitos.
- **Managers:** Usan el documento de requisitos para planear el proceso de desarrollo.
- **Ingenieros del sistema:** Usan el documento para entender que sistema deben desarrollar.
- **Ingenieros de testing:** Usan el documento para desarrollar pruebas de validación para el sistema.
- **Ingenieros de mantenimiento del sistema:** Usan el documento para entender el sistema y la relación entre sus partes.

4.12.2. Estructura de un documento de requisitos

- **Prefacio:** Definir los lectores esperados y describir el historial de versiones, incluyendo una justificación para la creación de una nueva versión y resumen de los cambios realizados en cada versión.
- **Introducción:** Describir las necesidades y las funciones del sistema y explicar como va a funcionar con otros sistemas. También debe describir como el sistema se ajusta a los objetivos del negocio o a los planes estratégicos de la organización.
- **Glosario:** Definir los términos técnicos usados en el documento. No se deben hacer suposiciones acerca de la experiencia o de los conocimientos del lector.
- **Definición de requisitos de usuario:** Aquí se describen los servicios prestados para el usuario. También pueden describirse los requisitos no funcionales del sistema. Esta descripción puede usar el lenguaje natural, diagramas u otra notación entendible por los clientes. Las normas de producto y proceso que deben seguir deben ser especificadas.
- **Arquitectura del sistema:** Este capítulo debe presentar una visión de alto nivel de la arquitectura del sistema, mostrando la distribución de las funciones sobre los módulos del sistema.
- **Especificación de los requisitos del sistema:** Describir con más detalle los requisitos funcionales y no funcionales, así como interfaces con otros sistemas.
- **Los modelos del sistema:** Aquí se puede incluir modelos gráficos del sistema que muestren las relaciones entre los componentes del sistema y la relación con su entorno.
- **Evolución del sistema:** Describir los supuestos fundamentales en los cuales está basado el sistema y cualquier cambio previsto debido a la evolución del hardware, cambios en las necesidades de los usuarios, etc. Esta sección es útil para los diseñadores del sistema ya que puede ayudar a evitar que las decisiones de diseño restrinjan posibles cambios futuros.
- **Apéndices:** Debería proporcionar información detallada y específica relacionada con la aplicación que se está desarrollando, por ejemplo hardware y descripción de la base de datos.
- **Índices:** Se pueden incluir varios índices. Entre ellos puede ser un índice alfabético, índice de diagramas, índice de funciones, etc.

4.13. Priorización de requisitos

Todo proyecto que tenga recursos limitados necesita definir prioridades en los requisitos que va a incluir en el producto de software. La priorización ayuda a entregar el mayor valor de negocio tan rápido como sea posible con las restricciones existentes. A veces los clientes no quieren priorizar requisitos pensando que no se hará lo que sea de prioridad baja. Los programadores por su lado no suelen priorizar porque eso daría la impresión de que no son capaces de realizar todos los requisitos.

Se debe elegir un nivel de abstracción apropiado (casos de uso, historias de usuario, requisitos funcionales o quizás flujos dentro de un caso de uso). Se deben tener en cuenta:

- Las necesidades de los clientes.
- La importancia relativa de los requisitos para los clientes.
- Las relaciones de precedencia entre los requisitos.
- Requisitos que deban ser implementados en grupo.
- El costo de satisfacer cada requisito.

Un estudio muestra que cerca de dos tercios de las características de los sistemas de software se usan raramente o nunca (The Standish Group 2009).

Es importante destacar que si terminamos el proceso de priorización con todos los requisitos con la misma prioridad entonces no priorizamos nada. Hay que evitar dos casos de priorización:

- **Por decibeles:** los que gritan más definen las prioridades.
- **Por amenazas:** los que tienen más poder definen las prioridades

Técnicas de priorización de requisitos

- Adentro y afuera: se nombra cada requisito y se decide si queda en el alcance o no.
- Comparación por pares y ranking: se hace un ranking general de todos los requisitos mediante comparación de pares.
- Escala de tres niveles: se utilizan dos dimensiones.



- 4 niveles (MoSCoW): debe, debería, podría, no.
- \$100: asignar dinero disponible y luego “invertirlo” en los requisitos a priorizar.
- Priorización basada en valores, costos y riesgos.

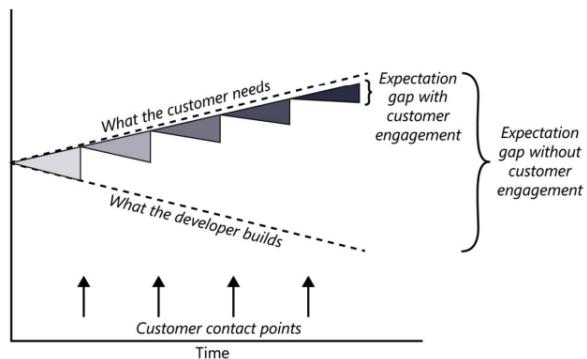
4.14. Validación de requisitos

Es el proceso por el cual se determina si los requisitos relevados son consistentes con las necesidades del cliente. Es un proceso muy importante ya que los errores en requisitos pueden llevar a costos de re-trabajo cuando los errores son descubiertos en el desarrollo o una vez el sistema ya está funcionando.

- **Verificación:** determina si el producto de alguna actividad de desarrollo cumple los requisitos (hacer las cosas bien).
- **Validación:** evalúa si un producto satisface las necesidades del cliente (hacer la cosa correcta).

4.14.1. Proceso

- Planificar quién (qué stakeholder) va a validar qué (artefacto) cómo (técnica).
- Ejecutar
- Registrar - reporte de validación / firma.



4.14.2. Chequeo de requisitos

- **Validez:** ¿El sistema provee las funciones que mejor soportan las necesidades del cliente?
- **Consistencia:** ¿Hay algún requisito en conflicto?
- **Compleitud:** ¿Están incluidas todas las funciones requeridas por el cliente?
- **Realismo:** ¿Los requisitos pueden ser implementados con el presupuesto y la tecnología disponible?
- **Verificabilidad:** ¿Los requisitos pueden ser chequeados?

4.14.3. Validación de requisitos no funcionales

Son difíciles de validar. Se deben expresar de manera cuantitativa utilizando métricas que se puedan probar de forma objetiva. Si bien esto es lo ideal, para los usuarios es difícil especificarlos de forma cuantitativa.

4.14.4. Técnicas de validación de requisitos

- Revisión de requisitos: análisis manual y sistemático de los requisitos.
- Prototipado: uso de modelos ejecutables del sistema para chequear los requisitos.
- Generación de casos de prueba: desarrollo de pruebas para los requisitos que pueden ser testeados.

4.14.5. ¿Cómo validar requisitos?

- Manuales
 - Lectura por parte del cliente.
 - Recorridas. Útiles con muchos stakeholders que no lo leerían de otra manera.
 - Entrevistas.

- Chequeo manual de referencias cruzadas.
 - Instancias de validación formal. Puede ser:
 - Revisiones: stakeholders revisan por separado y se reúnen para discutir problemas.
 - Inspecciones formales: se definen roles y reglas.
 - Listas de comprobación.
 - Escenarios.
 - Generación de casos de prueba.
- Automatizadas
 - Chequeo automático de referencias cruzadas.
 - Ejecución de modelos para verificar funciones y relaciones.
 - Construcción de prototipos.
 - Simulaciones.

Ejemplo checklist de defectos en requisitos

Completeness

- Do the requirements address all known customer or system needs?
- Is any needed information missing? If so, is it identified as TBD?
- Have algorithms intrinsic to the functional requirements been defined?
- Are all external hardware, software, and communication interfaces defined?
- Is the expected behavior documented for all anticipated error conditions?
- Do the requirements provide an adequate basis for design and test?
- Is the implementation priority of each requirement included?
- Is each requirement in scope for the project, release, or iteration?

Correctness

- Do any requirements conflict with or duplicate other requirements?
- Is each requirement written in clear, concise, unambiguous, grammatically correct language?
- Is each requirement verifiable by testing, demonstration, review, or analysis?
- Are any specified error messages clear and meaningful?
- Are all requirements actually requirements, not solutions or constraints?
- Are the requirements technically feasible and implementable within known constraints?

4.14.6. Validar utilizando criterios de aceptación

El cliente es el que tendrá la opinión final sobre un producto. La idea es entonces establecer, de forma complementaria a la definición de los requisitos, los criterios con los cuales se aceptarán. Los criterios de aceptación no son pruebas funcionales o unitarias; sino condiciones que debe cumplir el sistema. Las pruebas son más completas y prueba todos los flujos funcionales, excepciones, límites, etc.

Por ejemplo si tenemos un requerimiento que es “Registrar oferta laboral”, podemos tener el siguiente criterio de aceptación:

Permite que una empresa pueda registrar una oferta laboral basándose en las capacidades y competencias que maneja la carrera de computación.

4.15. Gestión de requisitos

La gestión de requisitos implica gestionar los cambios de los requisitos durante el proceso de ingeniería de requisitos y el desarrollo del sistema. Nuevos requisitos surgen cuando un sistema está siendo desarrollado y después de haber entrado en uso. Es necesario mantener un rastreo (trazabilidad) de los requisitos y mantener sus referencias para facilitar el análisis de impacto ante posibles cambios. Para esto se recomienda establecer un proceso formal de control de cambios.

4.15.1. Cambios en los requisitos

El entorno empresarial y técnico del sistema siempre cambia después de la instalación (nuevo hardware, nuevas interfaces con otros sistemas, cambian las prioridades del negocio, nuevas legislaciones, etc). Las personas que pagan por un sistema y los usuarios de ese sistema casi nunca son las mismas personas. Los grandes sistemas tienen una diversa comunidad de usuarios, algunos de los cuales tienen diferentes requisitos y pueden existir conflictos en las prioridades.

4.15.2. Planificación de la gestión de requisitos

Establece en detalle del nivel de gestión de requisitos que es requerido. Tenemos varias decisiones de gestión de requisitos:

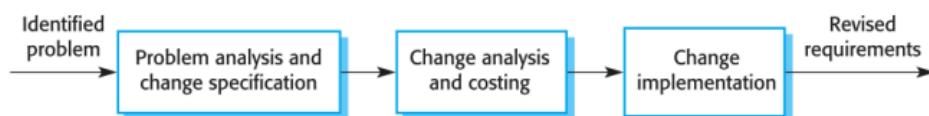
- Identificación de requisitos: cada requisito debe ser identificado de modo que pueda hacerse una referencia cruzada con otros.
- Proceso de gestión de cambios: es el conjunto de actividades que evalúan el impacto y el costo de los cambios.
- Políticas de trazabilidad: estas políticas definen cómo registrar las relaciones entre los requisitos y el sistema diseñado.
- Soporte de herramientas: las herramientas que se utilizarán.

4.15.3. Gestión de cambios en los requisitos

Implica lo siguiente:

- Decidir si un cambio en los requisitos debe ser aceptado.
- Problemas de análisis y cambios en la especificación: se analiza para chequear si es válido.
- Cambiar el análisis y calcular costos: se evalúa el efecto (información de trazabilidad y el conocimiento general del sistema). Se toma la decisión de si se debe proceder o no.
- Se implementa el cambio.

Esto puede resumirse con la siguiente gráfica



4.15.4. Cambios de requisitos en ágiles

Cuando un usuario propone un cambio de requisitos, este cambio no pasa por un proceso formal de gestión de cambios. El usuario debe priorizar ese cambio y, si es de alta prioridad, debe decidir qué características del sistema que se planificaron para la próxima iteración se deben eliminar para que se implemente el cambio.

Es importante destacar que en los sistemas con múltiples partes interesadas, los cambios beneficiarán a algunas partes interesadas y no a otras. Para esto se sugiere tener una autoridad independiente, que puede equilibrar las necesidades de todos los interesados.

5. Diseño de Software

El diseño de software es una actividad creativa donde se identifican los componentes del software y sus relaciones, con base en los requerimientos de un cliente. Algunas veces hay una etapa de diseño separada y este último se modela y documenta. En otras ocasiones, un diseño se halla en la mente del programador o se bosqueja en hojas de papel. El diseño trata sobre cómo resolver un problema, de modo que siempre existe un proceso de diseño.

El diseño de software se define como “el proceso de definición de la arquitectura, componentes, interfaces y otras características de un sistema o componente, y el resultado de ese proceso”

- Como proceso es una actividad del ciclo de vida en la cual se analizan requisitos de software para producir una descripción interna del software que servirá como base para su construcción.
- Como resultado describe la arquitectura del software (cómo se descompone y se organiza en componentes), y las interfaces entre esos componentes.

Consiste en **dos** actividades que se encuentran entre el análisis de requisitos del software y la construcción del software:

- **Diseño arquitectónico de software (o diseño de alto nivel)**
Desarrolla la estructura y la organización de alto nivel del software e identifica los diversos componentes.
- **Diseño detallado del software**
Especifica cada componente con detalles suficientes para facilitar su construcción.

Diseño correcto

Decimos que el diseño de un sistema es correcto si un sistema construido de acuerdo a ese diseño satisface los requerimientos del sistema. Sin embargo, el objetivo del diseño no es encontrar el diseño correcto sino encontrar el mejor diseño posible dentro de las limitaciones impuestas por requerimientos, ambiente, etc.

Un diseño debe de ser

- Verificable
- Completo (implementa toda la especificación)
- Rastreable: se puede rastrear hacia los requerimientos que diseña.
- Eficiente
- Simple

Crear un diseño simple y eficiente de un sistema grande puede ser una tarea extremadamente compleja ya que es sumamente creativa y no puede reducirse a una serie de pasos a seguir. Sin embargo podemos tener guías. Si se logra manejar la complejidad se reducen costos de diseño y se reduce la posibilidad de introducir defectos durante el diseño.

5.1. Principios de Diseño

5.1.1. Abstracción

La abstracción es “una vista de un objeto que se enfoca en la información relevante para un propósito particular e ignora el resto de la información”. Permite al diseñador considerar un componente sin preocuparse por los detalles de implementación (por ej. describiendo comportamiento exterior sin describir los detalles internos).

5.1.2. Acoplamiento y Cohesión

El acoplamiento es una medida de la interdependencia entre módulos en un programa informático. Más acoplamiento implica que será más difícil comprender y modificar. Depende de cuánta información se necesita para entender un módulo, y qué tan compleja y explícita es esa información.

La cohesión es una medida de la fuerza de asociación de los elementos dentro de un módulo. Más cohesión implica que es más fácil de comprender y modificar.

5.1.3. Descomposición y modularización

Descomponer y modularizar significa que el software grande se divide en una serie de componentes nombrados más pequeños que tienen interfaces bien definidas que describen las interacciones de los componentes. Usualmente el objetivo es colocar diferentes funcionalidades y responsabilidades en diferentes componentes. Decimos que un sistema se considera modular si consiste de componentes que se pueden implementar separadamente y el cambio en un componente tiene mínimos impactos en otros componentes.

5.1.4. Encapsulación y ocultamiento de información

Significa agrupar y empaquetar los detalles internos de una abstracción y hacer que esos detalles sean inaccesibles para las entidades externas.

5.1.5. Separación de la interfaz y la implementación

La separación de la interfaz y la implementación implica la definición de un componente especificando una interfaz pública (conocida por los clientes) que es independiente de los detalles de cómo se realiza el componente.

5.1.6. Suficiencia, integridad y primitivismo

Lograr la suficiencia y la integridad significa garantizar que un componente de software capture todas las características importantes de una abstracción y nada más. Primitividad significa que el diseño debe basarse en patrones fáciles de implementar.

5.1.7. Separación de intereses

Permite enfrentarse a los distintos aspectos individuales de un problema de forma de concentrarse en cada uno por separado. Por ejemplo: según el tiempo (ciclo de vida), cualidades (correctitud y eficiencia).

5.1.8. Dividir y conquistar

Debido a la complejidad de los grandes problemas y las limitaciones de la mente humana estos no se pueden atacar como una unidad monolítica. Aplicar dividir y conquistar implica dividir en piezas que pueden ser “conquistadas” por separado. Las piezas están relacionadas. Juntas forman el sistema. Existe comunicación y cooperación entre ellas. Esto agrega complejidad, que surge de la propia partición. Cuando el costo de particionar sumado la complejidad agregada supera los ahorros logrados por particionar se debe detener el proceso.

5.1.9. Reuso

Utilizar nuevamente algo (o construir pensado en). Hay varios niveles.

- Nivel de abstracción - Se utiliza el conocimiento de abstracciones exitosas en el diseño del software.
- Nivel del objeto - Se reutilizan objetos disponibles en lugar de escribir el código nuevamente

- Nivel de componente - Se reutilizan colecciones de objetos
- Nivel del sistema - Se reutilizan los sistemas de aplicación enteros.

Tiene como beneficios:

- Mayor confianza
El software reutilizado ha sido probado y testeado en producción, debería ser más confiable que el software nuevo.
- Reducción del riesgo del proceso
El costo de software existente es conocido, reduce el margen de error en la estimación del costo del proyecto.
- El uso eficaz de los especialistas
Especialistas de software pueden desarrollar software reutilizable que encapsule su conocimiento.
- Conformidad de los estándares
Algunos estándares tales como los estándares de interfaz de usuario pueden ser implementados como un conjunto de componentes reutilizables.
- Desarrollo acelerado
Puede acelerar la velocidad con que se pone el sistema en producción.

Tiene como problemas

- Aumento de los costos de mantenimiento
Si el código fuente de un sistema de software o componente no está disponible, los costos de mantenimiento pueden ser cada vez más altos porque los elementos reutilizados en el sistema pueden llegar a ser cada vez más incompatibles con los cambios del sistema.
- “Crear, mantener y usar” o “encontrar, comprender y adaptar” componentes reutilizables
Desarrollar componentes reutilizables y asegurarse de que los desarrolladores de software puedan usar la librería puede ser costoso.
- Síndrome de no inventado aquí
Algunos ingenieros de software prefieren reescribir los componentes porque creen que ellos lo pueden mejorar. Esto tiene que ver en parte con la confianza y en parte con el hecho de que escribir software original es visto como un desafío mayor que la reutilización de software de otros.

5.2. Problema malvado

Un problema malvado se caracteriza como un problema tal que una solución para uno de sus aspectos simplemente cambia el problema. Esto trae consigo varias implicancias

- No hay una formulación definitiva de un problema malvado.
- Los problemas malvados no tienen una regla de detención.
- Sus soluciones no son verdaderas o falsas, sino buenas o malas.
- No hay una prueba inmediata ni definitiva de una solución.
- Cada solución a un problema malvado es una “operación de una sola vez”, porque no hay oportunidad de aprender por ensayo y error, cada intento cuenta de manera significativa.
- No tienen un conjunto enumerable (o exhaustivamente describable) de soluciones potenciales.
- Cada problema malvado es esencialmente único.
- Cada problema malvado puede considerarse como un síntoma de otro problema.

5.3. Arquitectura de Software

Debemos pasar de la especificación de requerimientos al sistema instalado y funcionando. Para esto pasamos por varias etapas:

- Arquitectura de software
- Diseño detallado
- Implementación
- Verificación
- Liberación

5.3.1. Diseño arquitectónico

Cómo debe organizarse un sistema y cómo tiene que diseñarse la estructura global de ese sistema. Tiene como salida un modelo arquitectónico que describe la forma en que se organiza el sistema como un conjunto de componentes en comunicación. Usualmente no resulta exitoso el desarrollo incremental de arquitecturas.

Es una etapa temprana del proceso de diseño del sistema. Representa la asociación entre la especificación y los procesos de diseño. Comprende la identificación de los principales componentes del sistema y sus relaciones. A menudo se lleva a cabo con algunas actividades en paralelo de especificación y otras disciplinas.

Podemos separar la arquitectura en dos tipos

- **Arquitectura en alto nivel**

Se refiere a la arquitectura de sistemas empresariales complejos que incluyen otros sistemas, programas y componentes de programas.

- **Arquitectura detallada**

Tiene que ver con la arquitectura de los programas individuales. En este nivel nos preocu-pamos de que los programas individuales sean descompuestos en componentes.

5.3.2. Ventajas de explicitar la arquitectura

- Nos podemos comunicar mejor. Tener una descripción de la arquitectura mejora la comuni-cación.
- Permite hacer un mejor análisis de requisitos funcionales y no funcionales, así como poder negociar los requisitos.
- Permite la reutilización a gran escala.

5.3.3. Decisiones en el diseño de arquitectura

Es un proceso creativo, por lo que está formado por decisiones en vez de actividades. Podemos tomar medidas para guiar esas decisiones. Algunas preguntas guía son las siguientes:

- ¿Existe una arquitectura genérica que se pueda usar?
- ¿Cómo va a ser distribuido el sistema?
- ¿Qué patrones o estilos son apropiados?
- ¿Se descompondrá en módulos?
- ¿Cómo va a ser evaluado el diseño de la arquitectura?
- ¿Cómo se documentará la arquitectura del sistema?

5.3.4. Reutilización de arquitectura

Generalmente los sistemas que pertenecen al mismo dominio tienen arquitecturas similares que reflejan los conceptos del dominio. La arquitectura de un sistema puede ser diseñada en torno a uno o mas patrones de arquitectura o “estilos” (capturan la esencia de una arquitectura y pueden ser instanciados de diferentes maneras).

5.3.5. ¿Qué afecta y determina la arquitectura?

En general la arquitectura afecta a todos los requisitos no funcionales. Algunos puntos en los que dependerá son:

- Rendimiento
- Seguridad
- Protección
- Disponibilidad
- Mantenibilidad

5.3.6. Conflictos entre soluciones

Muchas veces ocurre que hay conflictos entre las soluciones. En estos casos se puede recurrir a soluciones de compromiso (reuniones en donde las partes se ponen de acuerdo), o utilizar diferentes estilos para diferentes partes.

Evaluar un diseño arquitectónico es difícil porque la verdadera prueba es qué tan bien el sistema cubre sus requisitos funcionales y no funcionales cuando están en uso.

5.4. Patrones arquitectónicos

Un patrón es una descripción abstracta de buena práctica, que se ensayó y puso a prueba en diferentes sistemas y entornos. Describe una organización de sistema que ha tenido éxito en sistemas previos. Incluye información sobre cuándo es y cuándo no es adecuado usar dicho patrón, así como las fortalezas y debilidades del patrón.

Tiene como propósito compartir una solución probada, ampliamente aplicable a un problema particular de diseño. El patrón se presenta en una forma estándar que permite que sea fácilmente reutilizado. Un patrón se compone de 5 piezas importantes: Nombre, Contexto, Problema, Solución, Consecuencias (positivas y negativas).

Beneficios

Permite seleccionar una solución entendible y probada a ciertos problemas, definiendo los principios organizativos del sistema. Al basar la arquitectura en estilos que son conocidos se facilita comunicar las características importantes de la misma.

Formas de usar patrones

- Utilizar un patrón como solución para el diseño del sistema.
- Utilizar un patrón como base para una adaptación. Algún estilo soluciona parcialmente los problemas.
- Patrón como inspiración para una solución relacionada. Ningún estilo sirve, pero algunos ayudan a entender mejor los problemas.
- Patrón como motivación para un nuevo patrón. Ningún estilo sirve, además encontramos una solución general al problema.

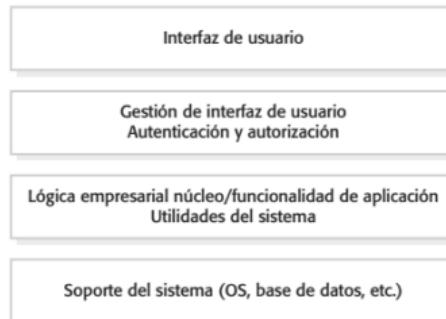
5.4.1. Arquitectura en capas

Descripción: Organiza el sistema en capas con funcionalidad relacionada con cada capa. Una capa da servicios a la capa de encima.

Cuándo usar: Nuevas facilidades encima de sistemas existentes. Desarrollo disperso en varios equipos, y cada uno es responsable de una capa de funcionalidad. Requisito de seguridad multi-nivel.

Ventajas: Permite la sustitución de capas en tanto se conserve la interfaz. Aumentar la confiabilidad, en cada capa pueden incluirse facilidades redundantes (por ejemplo, autenticación).

Desventajas: Es difícil ofrecer una separación limpia entre capas. El rendimiento suele ser un problema.



- Soporta el desarrollo incremental
- También permite implementaciones multi-plataformas.

5.4.2. Arquitectura de repositorio

Descripción: Todos los datos en un sistema se gestionan en un repositorio central, accesible a todos los componentes del sistema. Los componentes no interactúan directamente.

Cuándo usar: Cuando hay grandes volúmenes de información o para sistemas dirigidos por datos.

Ventajas: Los componentes no necesitan conocer la existencia de otros componentes. Los cambios se propagan hacia todos los componentes. Gestión consistente y centralizada de los datos.

Desventajas: Problemas en el repositorio afectan a todo el sistema. Posibles ineficiencias en la comunicación. Posibles dificultades al distribuir el repositorio.



Una alternativa es la arquitectura de pizarrón donde los componentes se activan cuando hay datos en particular. Hay una especie de suscripción a distintos tipo de contenidos.

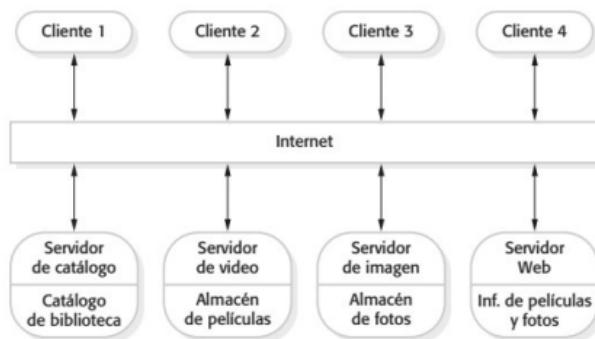
5.4.3. Arquitectura cliente servidor

Descripción: La funcionalidad del sistema se organiza en servicios, y cada servicio lo entrega un servidor independiente. Los clientes son usuarios de dichos servicios.

Cuándo usar: Se usa cuando, desde varias ubicaciones, se tiene que ingresar a los datos en una base de datos compartida. También cuando la carga es variable (replicación de servidores).

Ventajas: Los servidores se pueden distribuir a través de una red. La funcionalidad general estaría disponible a todos los clientes, no necesita implementarse en todos los servicios.

Desventajas: Cada servicio son puntos de falla. El rendimiento depende de la red, así como del sistema. Posibles problemas administrativos cuando los servidores son propiedad de diferentes organizaciones.



- El modelo lógico de servicios independientes que opera en servidores separados puede implementarse en una sola computadora.
- El beneficio importante es la separación e independencia.

5.4.4. Arquitectura de tubería y filtro

Descripción: Cada componente de procesamiento (filtro) es discreto y realiza un tipo de transformación de datos. Los datos fluyen de un componente a otro para su procesamiento.

Cuándo usar: Se suele utilizar en aplicaciones de procesamiento de datos, donde las entradas se procesan en etapas separadas para generar salidas relacionadas.

Ventajas: Fácil de entender y soporta reutilización. El estilo del flujo de trabajo coincide con la estructura de muchos procesos empresariales. La evolución al agregar transformaciones es directa. Permite concurrencia.

Desventajas: El formato debe acordarse y respetarse. Esto aumenta la carga del sistema, y puede significar que sea imposible reutilizar transformaciones funcionales que usen otros formatos.



- Usada para procesamiento automático de datos.
- Difícil de usar para sistemas interactivos.

5.5. Arquitecturas de aplicación

Encapsulan las principales características de una clase de sistemas. Uso de modelos de arquitecturas de aplicación:

1. Como punto de partida para el diseño
2. Como lista de verificación del diseño
3. Como una forma de organizar el trabajo
4. Como un medio para valorar componentes a reusar
5. Como un vocabulario para hablar sobre tipos de aplicaciones

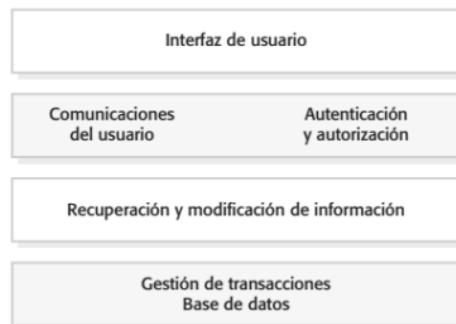
5.5.1. Sistemas de procesamiento de transacciones

Procesan peticiones del usuario mediante la información o una base de datos. Una transacción de base de datos es una secuencia de operaciones que se trata como una sola unidad. En general, son sistemas interactivos donde los usuarios hacen peticiones asíncronas de servicios. Pueden organizarse como una arquitectura tubería y filtro.

5.5.2. Sistemas de información

Todos los sistemas que incluyen interacción con una base de datos compartida se consideran sistemas de información basados en transacciones. Un sistema de información permite acceso controlado a una gran base de información, tales como un catálogo de biblioteca, un horario de vuelos o los registros de pacientes en un hospital. Cada vez más, los sistemas de información son sistemas basados en la Web, cuyo acceso es mediante un navegador Web.

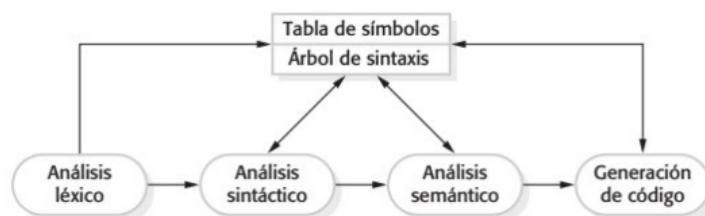
Un modelo muy general para estos sistemas se basa en capas.



5.5.3. Sistemas de procesamiento de lenguajes

Convierten un lenguaje natural o artificial en otra representación del lenguaje y, para lenguajes de programación, también pueden ejecutar el código resultante.

Para compiladores (entornos más batch) puede utilizarse una composición de un repositorio con tubería y filtro.

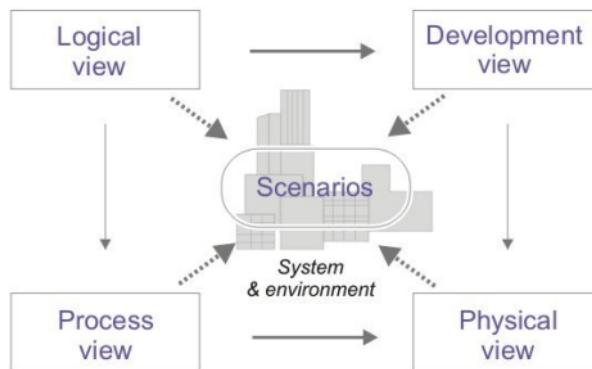


Cuando es más interactivo puede ser más efectivo utilizar un repositorio.

5.6. Vistas arquitectónicas

Refiere a que vistas o perspectivas son útiles para diseñar y documentar una arquitectura.

5.6.1. Modelo 4+1



- **Vista lógica:** abstracciones claves en el sistema como objetos o clases de objetos.
- **Vista del proceso:** interacción de procesos en tiempo de ejecución.
- **Vista de desarrollo:** como el software se descompone para el desarrollo.
- **Vista física:** hardware del sistema y como los componentes de software son distribuidos.
- Relacionados con los casos de uso o escenarios (+1)

5.6.2. Tipos de vistas

Respecto a las “notaciones” hay diferentes opiniones.

- Cuando se usa UML se hace informalmente
- Algunos autores proponen notaciones rápidas e informales
- Otros lenguajes basados en componentes y conectores

Se sugiere desarrollar las vistas que sean útiles para la comunicación sin preocuparse por completitud a no ser que sea un sistema crítico.

Tenemos los siguientes tipos de vistas:

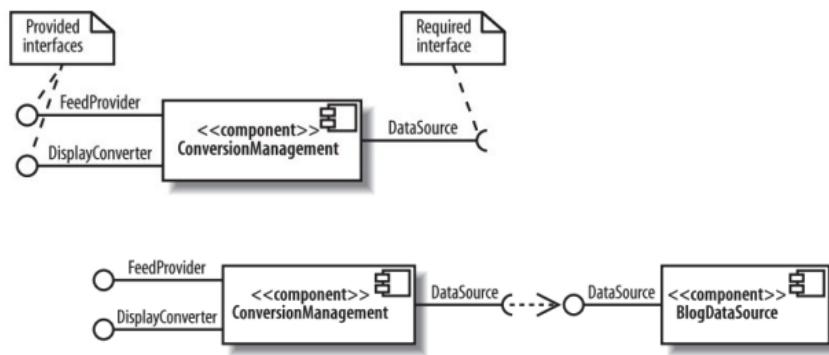
- Vista de casos de uso/escenarios ->casos de uso
- Vista de procesos ->andariveles, diagramas de actividad, bpmn
- Vista lógica ->clases, secuencia, actividad
- Vista de desarrollo ->diagrama de componentes
- Vista física ->diagrama de despliegue (deploy)

5.6.3. Diagrama de componentes

Un componente es una parte encapsulada, reusable y reemplazable del software (bloques de construcción). Pueden ser de tamaño chico (clase) hasta grande (subsistema).

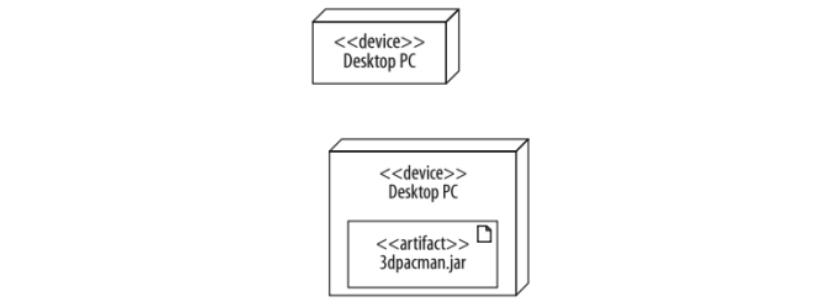


Se puede indicar interfaces requeridas y provistas.

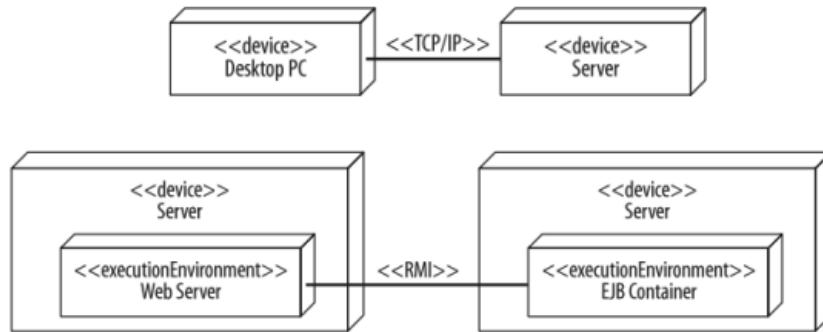


5.6.4. Diagrama de despliegue

Muestra cómo el software se asigna al hardware y cómo se comunican las distintas piezas. Cada nodo representa una pieza de hardware. Dentro de cada nodo se pueden colocar artefactos de software los cuales ejecutarán en esa pieza de software. Saliendo un poco de UML es frecuente colocar componentes (grandes) en los nodos para mostrar cómo será desplegado el software.



Se muestra la comunicación entre nodos con una línea sólida agregando un estereotipo indicando la forma de comunicación (ej, protocolo).



¿Cuándo usar diagramas de despliegue?

Los diagramas de despliegue son útiles en todas las etapas del proceso de diseño. Por ejemplo en una etapa temprana, si bien es probable que no se haya decidido que hardware usar, se puede querer comunicar características del sistema como las siguientes:

- La arquitectura incluye un servidor web, servidor, base de dato.
- Los clientes pueden acceder a la aplicación a través del navegador o a través de una interfaz gráfica más rica.
- El servidor web está protegido por un firewall.

Luego, los diagramas de despliegue son muy útiles en etapas tardías de desarrollo, ya que permiten ver una vista detallada de como es la implementación del sistema.

5.7. Ingeniería de Software basada en componentes

La ingeniería de software basada en componentes (CBSE) es un enfoque para el desarrollo de software que se basa en la reutilización de entidades llamadas “componentes de software”. Surgió del fracaso del desarrollo orientado a objetos para apoyar la reutilización efectiva. Las clases de objetos individuales son demasiado detalladas y específicas. Los componentes son más abstractos que las clases de objetos y pueden considerarse proveedores de servicios independientes. Pueden existir como entidades independientes.

5.7.1. Pilares de CBSE

- Componentes independientes especificados por sus interfaces.
- Estándares de componentes para facilitar la integración de componentes.
- Middleware que proporciona soporte para la interoperabilidad de los componentes.
- Un proceso de desarrollo que está orientado a la reutilización.

5.7.2. CBSE y principios de diseño

- Los componentes son independientes, así que no deben interferir entre sí.
- Las implementaciones de componentes están ocultas.
- La comunicación es a través de interfaces bien definidas.
- Un componente puede ser reemplazado por otro si se mantiene su interfaz.
- Las infraestructuras de componentes ofrecen una gama de servicios estándar.

5.7.3. Estándares de componentes

Los estándares deben establecerse para que los componentes se puedan comunicar entre sí e interoperar. Desafortunadamente, se establecieron varios estándares de componentes competidores. Estos estándares múltiples han dificultado la adopción de CBSE. Es imposible que los componentes desarrollados usando diferentes enfoques trabajen juntos. Para esto surge como alternativa la “ingeniería de software orientada a servicios”.

5.7.4. Componentes

Los componentes proporcionan un servicio sin importar dónde se está ejecutando el componente o su lenguaje de programación. Un componente es una entidad ejecutable independiente que puede estar compuesta por uno o más objetos ejecutables. La interfaz del componente se publica y todas las interacciones se realizan a través de la interfaz publicada.

Tiene como características que es estandarizado, independiente, componible, implementable y documentado.

5.7.5. Interfaces de los componentes

- Interfaces provistas: Definen los servicios que ofrece el componente.
- Interfaces requeridas: Definen qué servicios deben ofrecer otros componentes para que este componente opere correctamente. Esto no compromete la independencia o la capacidad de despliegue de un componente porque la interfaz “requiere” no define cómo se deben proporcionar estos servicios.

5.7.6. Acceso y modelos

Se accede a los componentes mediante llamadas a procedimientos remotos (RPC). Cada componente tiene un identificador único (generalmente una URL) y puede referenciarse desde cualquier computadora en red. Un modelo de componentes es una definición de estándares para la implementación, documentación y despliegue de componentes. Ejemplos: EJB (Enterprise Java Beans), COM+ (modelo .NET). El modelo de componente especifica cómo se deben definir las interfaces y los elementos que deberían incluirse en una definición de interfaz.

5.7.7. Elementos de un modelo

- **Interfaces**

El modelo especifica cómo se deben definir las interfaces y los elementos, como los nombres de operación, los parámetros y las excepciones, que deben incluirse en la definición de la interfaz.

- **Uso**

Para que los componentes se distribuyan y accedan de forma remota, deben tener un nombre único o un identificador asociado a ellos. Esto tiene que ser globalmente único.

- **Despliegue**

El modelo incluye una especificación de cómo los componentes se deben empaquetar para su despliegue como entidades ejecutables independientes.

5.7.8. Soporte del middleware

Los modelos de componentes son la base del middleware que proporciona soporte para la ejecución de componentes. Las implementaciones del modelo de componentes proporcionan:

- Servicios de plataforma que permiten que los componentes escritos según el modelo se comuniquen.
- Servicios de soporte que son servicios independientes de la aplicación utilizados por diferentes componentes.

Para utilizar los servicios proporcionados por un modelo, los componentes se implementan en un contenedor. Este es un conjunto de interfaces utilizadas para acceder a las implementaciones del servicio.

5.7.9. Desarrollo para el reuso

Es el desarrollo de componentes o servicios que se reutilizarán/ Los componentes desarrollados para una aplicación específica generalmente deben generalizarse para que sean reutilizables.

La reutilización de componentes debería

- Reflejar abstracciones de dominio estables.
- Ocultar la representación del estado.
- Ser lo más independiente posible.
- Publicar excepciones a través de la interfaz del componente.

Hay una compensación entre la reutilización y la usabilidad. Cuanto más general es la interfaz, mayor es la capacidad de reutilización, pero es más compleja y, por lo tanto, menos utilizable.

Se deben de realizar los siguientes cambios para que un componente o servicio se pueda reutilizar:

- Eliminar los métodos específicos de la aplicación.
- Cambiar los nombres para hacerlos generales.

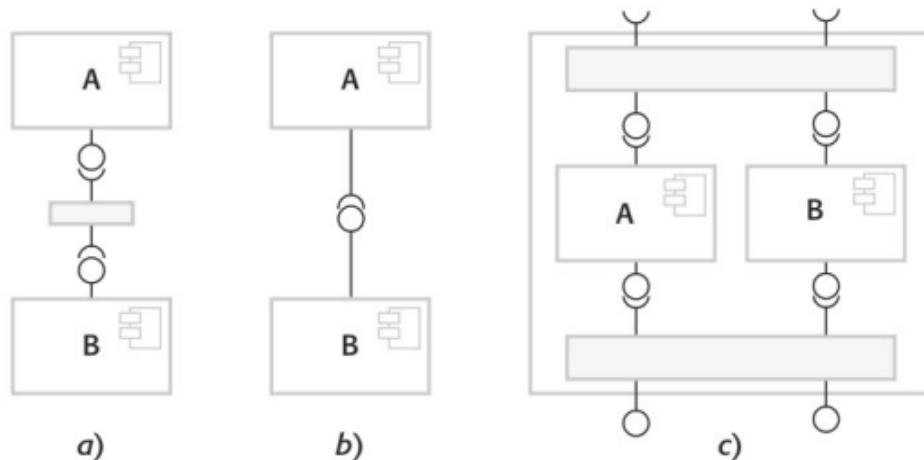
- Agregar métodos para ampliar la cobertura.
- Hacer que el manejo de excepciones sea consistente.
- Agregar una interfaz de configuración para la adaptación del componente.
- Integrar los componentes necesarios para reducir las dependencias.

CBSE con proceso de reutilización tiene que encontrar e integrar componentes reutilizables. Al reutilizar los componentes, es esencial negociar entre los requisitos ideales y los servicios realmente proporcionados por los componentes disponibles. Esto involucra:

- Desarrollar requisitos de esquema;
- Búsqueda de componentes y luego modificación de requisitos según la funcionalidad disponible.
- Buscando de nuevo para encontrar si hay mejores componentes que cumplan con los requisitos revisados.
- Componiendo componentes para crear el sistema.

5.7.10. Composición de componentes

- Secuencial: los componentes se ejecutan en secuencia.
- Jerárquica: un componente llama a los servicios de otro.
- Aditiva: las interfaces de dos componentes se juntan para crear un nuevo componente.



Problemas de adaptabilidad se solucionan con adaptadores (wrappers). Para resolver posibles conflictos se puede pensar en las siguientes preguntas:

1. ¿Qué composición es más efectiva para entregar los requerimientos funcionales del sistema?
2. ¿Qué composición facilitará la adaptación del componente cuando cambien los requerimientos?
3. ¿Cuáles serán las propiedades emergentes del sistema compuesto? (rendimiento y confiabilidad).

Una buena idea es utilizar el principio de separación de intereses donde cada componente tiene un papel claramente definido y que los roles no se traslapen.

5.8. Ingeniería de software distribuido

Un sistema distribuido es una colección de computadoras independientes que aparece para el usuario como un único sistema coherente. El procesamiento de información se distribuye en varias computadoras en lugar de limitarse a una sola máquina. Tiene como características:

- Compartir recursos: compartir recursos de hardware y software.
- Apertura: uso de equipos y software de diferentes proveedores.
- Concurrencia: procesamiento concurrente para mejorar rendimiento.
- Escalabilidad: mayor rendimiento al agregar nuevos recursos.
- Tolerancia a fallas: la capacidad de continuar en funcionamiento después de que se ha producido un error.

Los sistemas distribuidos son más complejos que los sistemas que se ejecutan en un solo procesador. La complejidad surge porque las diferentes partes del sistema se manejan de manera independiente como lo hace la red. No hay una sola autoridad a cargo del sistema, por lo que el control descendente es imposible.

5.8.1. Aspectos de diseño

- Transparencia
¿En qué medida debería aparecer el sistema distribuido al usuario como un sistema único? Idealmente queremos transparencia pero en la práctica es casi imposible por la gestión independiente y las demoras en la red. A veces es mejor advertir a los usuarios. Recursos direccionados de forma lógica y no física.
- Apertura
¿Debe diseñarse un sistema utilizando protocolos estándar que respalden la interoperabilidad? Desarrollo basado en estándares generalmente aceptados. Implica desarrollo independiente en cualquier lenguaje.
- Escalabilidad
¿Cómo se puede construir el sistema para que sea escalable? La escalabilidad es la capacidad para ofrecer un servicio de alta calidad a medida que aumentan las demandas sobre el sistema.
 - Tamaño (agregar más recursos)
 - Distribución (dispersar geográficamente los componentes)
 - Capacidad de administración
 - Ampliar (scaling-up): sistema más poderoso
 - Escalar (scaling-out): más instancias del sistema.
- Seguridad
¿Cómo se pueden definir e implementar políticas de seguridad utilizables? La cantidad de formas en que el sistema puede ser atacado aumenta significativamente, en comparación con los sistemas centralizados. Al estar distribuido se tienen más vulnerabilidades. También sucede que es más difícil de manejar puesto que diferentes organizaciones pueden poseer partes del sistema.
- Calidad del servicio
¿Cómo se especifica la calidad del servicio? La calidad de servicio (QoS) refleja la capacidad del sistema de entregar sus servicios de manera confiable y con un tiempo de respuesta y rendimiento aceptable para sus usuarios. Es importante cuando el sistema maneja datos de tiempo crítico tales como transmisiones de sonido o vídeo.

- Gestión de fallas

¿Cómo se pueden detectar, contener y reparar las fallas del sistema? En un sistema distribuido, es inevitable que se produzcan fallas, por lo que el sistema debe diseñarse para ser resistente a estas fallas. Si un componente del sistema ha fallado tratar de continuar entregando tantos servicios como sea posible y si se puede recuperarse.

5.8.2. Modelos de interacción

- Interacción procesal

Una computadora llama a un servicio conocido ofrecido por otra computadora y espera una respuesta. Implementación RPC (remote procedure calls), solicitudes como si el otro fuera un componente local (solución middleware).

- Interacción basada en mensajes

La computadora que envía, envía información sobre lo que se requiere a otra computadora. No hay necesidad de esperar una respuesta. Se crean mensajes que se mandan a través del middleware. En este enfoque no es necesario conocer al otro.

5.8.3. Middleware

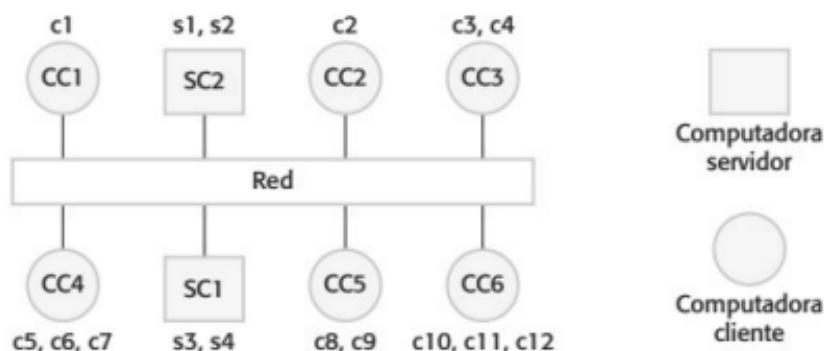
Los componentes en un sistema distribuido pueden implementarse en diferentes lenguajes de programación y pueden ejecutarse en tipos de procesador completamente diferentes. Los modelos de datos, la representación de información y los protocolos de comunicación pueden ser diferentes. Middleware es un software que puede administrar estas diversas partes y garantizar que puedan comunicarse e intercambiar datos.

Se debe tener **soporte de interacción** para coordinar las interacciones entre los diferentes componentes de un sistema. No es necesario que los componentes conozcan las ubicaciones físicas de otros componentes.

Por otro lado, la **provisión de servicios comunes** proporciona implementaciones reutilizables de servicios que pueden ser requeridas por varios componentes en el sistema distribuido. Los componentes pueden interactuar fácilmente y proporcionar servicios al usuario de manera consistente.

5.8.4. Computación cliente-servidor

Los sistemas distribuidos a los que se accede a través de Internet normalmente están organizados como sistemas cliente-servidor. La computadora remota proporciona servicios, como el acceso a páginas web, que están disponibles para clientes externos.



5.8.5. Patrones arquitectónicos para sistemas distribuidos

Maestro-esclavo

Se usan comúnmente en sistemas en tiempo real. El proceso “maestro” generalmente es responsable del cálculo, la coordinación y las comunicaciones, y controla los procesos “esclavos”. Los procesos “esclavos” están dedicados a acciones específicas, como la adquisición de datos de una matriz de sensores.

Cliente-servidor de dos niveles

El sistema se implementa como un solo servidor lógico más un número indefinido de clientes que usan ese servidor. Distinguimos:

- **Cliente ligero:** la capa de presentación se implementa en el cliente y todas las demás capas (gestión de datos, procesamiento de aplicaciones y base de datos) se implementan en un servidor. Es simple de manejar por clientes, soporta manejo de sistemas legados, fuerte carga en red y servidor.
- **Cliente pesado:** parte o todo el procesamiento de la aplicación se lleva a cabo en el cliente. La administración de datos y las funciones de la base de datos se implementan en el servidor. Es adecuado cuando sabemos capacidades de los clientes y se pueden utilizar, más difícil de gestionar y se debe instalar el software en los clientes.

Sin embargo, las fronteras son borrosas. Por ejemplo Javascript ha permitido clientes más pesados sin gestión adicional, pocos cliente finos actualmente.

Cliente-servidor multinivel

Las diferentes capas del sistema, a saber, presentación, administración de datos, procesamiento de aplicaciones y base de datos, son procesos separados que pueden ejecutarse en diferentes procesadores. Tiene como ventaja que son más escalables, ya que podemos separar información y generar redundancia fácilmente. Se usan cuando hay varias bases de datos (se agrega servidor de integración). El procesamiento puede distribuirse entre la lógica de la aplicación y los servidores de gestión de datos lo cual genera una respuesta más rápida.

Componentes distribuidos

No hay distinción entre clientes y servidores. Cada entidad distribuible es un componente que proporciona servicios a otros componentes y recibe servicios de otros componentes. La comunicación de componentes se realiza a través de un sistema de middleware.

Ventajas

- Permite al diseñador retrasar las decisiones sobre dónde y cómo se deben proporcionar los servicios.
- Es una arquitectura muy abierta que permite agregar nuevos recursos según sea necesario.
- El sistema es flexible y escalable.
- Se puede reconfigurar el sistema dinámicamente con migración de componentes a través de la red. Esto puede ser importante cuando hay patrones de fluctuaciones en la demanda de los servicios.

Desventajas

- Es posible reconfigurar el sistema de forma dinámica según sea necesario.
- Son más complejos de diseñar que los sistemas cliente-servidor.
- El middleware estandarizado nunca ha sido aceptado por la comunidad.

Las arquitecturas orientadas a servicios están reemplazando las arquitecturas de componentes distribuidos en muchas situaciones.

Entre pares (peer-to-peer)

Son sistemas descentralizados donde los cálculos pueden ser llevados a cabo por cualquier nodo en la red. El sistema general está diseñado para aprovechar la capacidad de cómputo y el almacenamiento de una gran cantidad de computadoras en red. La mayoría de los sistemas P2P han sido sistemas personales, pero el uso comercial de esta tecnología es cada vez mayor.

Son útiles de usar cuando se tiene un sistema de cómputo intensivo y podemos distribuir procesamiento, sistema de intercambio de información sin necesidad de gestión de información centralizada. La arquitectura es descentralizada. (todos pares, más redundante, más robusto) o semicentralizadas (uno o más servidores, menos carga de red).

5.8.6. Software como Servicio (SaaS)

El software como servicio (SaaS) implica alojar el software de forma remota y proporcionar acceso a él a través de Internet. Se implementa en un servidor (o más comúnmente en varios servidores) y se accede a él a través de un navegador web. No se implementa en una PC local. El software es propiedad y está administrado por un proveedor de software, en lugar de las organizaciones que usan el software. Los usuarios pueden pagar por el software de acuerdo con la cantidad de uso que hacen de él o mediante una suscripción anual o mensual.

Diferencias entre SaaS y SOA

- SaaS es una forma de proporcionar funcionalidad en un servidor remoto, con acceso de clientes mediante un navegador Web. El servidor conserva los datos y el estado del usuario durante una sesión de interacción. Por lo regular, las transacciones son largas.
- SOA (Arquitecturas orientadas a servicios) es un enfoque a la estructuración de un sistema de software como un conjunto de servicios independientes, sin estado. Éstos pueden proporcionarse mediante múltiples proveedores y distribuirse. Por lo general, las transacciones son transacciones cortas.

En definitiva, SaaS es una forma de entregar funcionalidad de aplicación a los usuarios, mientras que SOA es una tecnología de implementación para sistemas de aplicación.

6. Construcción de Software

Refiere a la creación detallada del software mediante una combinación de codificación, verificación, pruebas unitarias, pruebas de integración y debugging.

6.1. Fundamentos de la Construcción de Software

- **Minimización de la complejidad**

El software tiene que ser entendido por personas. Además reducir la complejidad es crítico para las pruebas. Esto se puede lograr mediante la escritura de código simple y legible en lugar de inteligente, uso de estándares, diseño modular.

- **Anticipación al cambio**

Construir sabiendo que lo que hacemos puede cambiar.

- **Construcción para la verificación**

Esto permite encontrar las faltas fácilmente en el software.

- **Reuso**

Se logra en distintos niveles: abstracción / objeto / componente / sistema.

Se puede construir PARA el reuso, o construir CON reuso.

- **Estándares en la construcción**

Utilización de estándares tanto internos como externos.

6.2. Gestión de la Construcción

La construcción depende del modelo del ciclo de vida utilizado. Esto implica la elección del método de construcción, el orden en el cual se crean los componentes, se integran, se gestiona la calidad de la construcción y la asignación a los distintos programadores.

6.3. Consideraciones prácticas

- **El diseño en la construcción**

Cuando se programa pueden surgir ocasiones en las que hay que hacer cambios en el diseño.

Esto hay que manejarlo con cuidado, y actualizar los documentos que sean necesarios para no perder el rastro al diseño original.

- **Lenguajes de construcción**

Tenemos lenguajes de configuración, lenguajes de caja de herramientas (toolkit), lenguajes de programación, notaciones lingüísticas, notaciones formales y notaciones visuales.

- **Codificación**

- Técnicas para crear código fuente, convenciones de código y plantillas de código fuente.
- Uso de clases, tipos enumerados, variables, etc.
- Uso de estructuras de control.
- Manejo de excepciones (errores).
- Prevención de rupturas de seguridad a nivel del código (por ej. índice fuera del array).
- Mecanismos de exclusión y manejo de recursos compartidos.
- Organización del código.
- Documentación del código.
- Puesta a punto del código (tuning).

- **Pruebas en la construcción**

Realización de pruebas unitarias y pruebas de integración. El propósito es reducir el intervalo entre que se introduce una falla en el código y su detección.

- **Construcción para el reuso**

Análisis de variabilidad y diseño.

■ Construcción con reuso

Involucra construir con el reuso de software existente.

■ Calidad en la construcción

- Pruebas unitarias y de integración.
- Desarrollo basado en pruebas.
- Uso de aserciones y programación defensiva.
- Debugging.
- Inspecciones.
- Revisiones técnicas.
- Análisis estático.

■ Integración

- Big bang
- Incremental

6.4. Tecnologías de Construcción

■ APIs

API significa Application Programming Interface. Corresponde a un conjunto de formas que son exportadas y disponibles para que los usuarios de una biblioteca o framework puedan escribir sus aplicaciones.

■ Cuestiones de ejecución OO

- Polimorfismo: es la habilidad de un lenguaje de soportar operaciones generales sin saber hasta tiempo de ejecución que tipos concretos de objetos incluirán.
- Reflection: es la habilidad de un programa de observar y modificar su propia estructura y comportamiento en tiempo de ejecución.

■ Tipos genéricos

Permiten definir un tipo o clase sin especificar los tipos de los objetos que utiliza.

■ Aserciones, Diseño por contrato y Programación defensiva

- Aserciones: es un predicado ejecutable que permite realizar chequeos en tiempo de ejecución. En general son quitadas para la versión del software que será puesta en producción.
- Diseño por contrato: pre y poscondiciones son incluidas para cada operación.
- Programación defensiva: la idea es proteger a las rutinas de entradas incorrectas.

■ Manejo de errores, Manejo de excepciones y Tolerancia a las Fallas

- La manera en que se manejan los errores afecta varios atributos de calidad (correctitud, robustez, etc). Existen muchas técnicas para realizar esto: aserciones, retornar valores neutros, logueo de errores, retornar códigos de error, “apagar el software”, etc.
- Excepciones: se utilizan para detectar y procesar errores o eventos excepcionales. La estructura básica utiliza: *throw* para lanzar una excepción y el bloque *try-catch* para su manejo.
- Tolerancia a las fallas: son un conjunto de técnicas que buscan aumentar la confiabilidad en el software mediante la detección de error y la recuperación.

■ Modelos ejecutables

Buscan abstraer los detalles de los lenguajes específicos de programación y las decisiones sobre la organización del software. Una especificación construida en un modelo ejecutable puede ser desplegada en varios entornos de ejecución sin realizar cambios. Ejemplos son xUML, BPMN, Model-driven Architecture.

■ Técnicas de construcción basadas en estados o dirigidas por tablas

- Programación basada en estados: es una tecnología de programación que utiliza máquinas de estados finitos para describir el comportamiento de los programas.
- Un método dirigido por tabla es un esquema en el cual se busca en una tabla información en lugar de utilizar sentencias lógicas (por ejemplo if-else).

■ Configuración en tiempo de ejecución e Internacionalización

- Configuración en tiempo de ejecución (runtime configuration): es una técnica para enlazar valores de las variables y seteos del programa en ejecución, utilizando, en general, archivos de configuración.
- Internacionalización: es la actividad técnica de preparar un programa para múltiples sitios. La actividad de localización corresponde a la adaptación para un lenguaje local específico.

■ Procesamiento de entradas basado en la gramática

Involucra el análisis sintáctico (parseo) de la entrada.

■ Primitivas de Concurrencia

Son primitivas que facilitan concurrencia y sincronización. Ejemplos de esto son Semáforos (variable protegida que provee control de acceso a un recurso en común), Monitores (data type con operaciones definidas por el programador ejecutadas con mutua-exclusión) y Mutex (primitiva de sincronización que garantiza el acceso exclusivo a un recurso compartido).

■ Middleware

Es una clasificación para todo software que provea servicios sobre la capa del sistema operativo y por debajo de la capa aplicación. Puede proveer contenedores en tiempo de ejecución para componentes de software para proveer pasaje de mensajes, persistencia y ubicación transparente a través de la red. El middleware orientado a mensajes moderno provee usualmente de un Enterprise Service Bus (ESB) que soporta la interacción orientada a servicios y la comunicación entre múltiples aplicaciones de software.

■ Cloud computing

Paradigma que permite ofrecer servicios de computación a través de una red, que usualmente es Internet.

- Software como servicio (SaaS): Una instancia del software que corre en la infraestructura del proveedor y sirve a múltiples organizaciones de clientes.
- Plataforma como servicio (PaaS): Es la encapsulación de una abstracción de un ambiente de desarrollo y el empaquetamiento de una serie de módulos o complementos que proporcionan, normalmente, una funcionalidad horizontal (persistencia de datos, autenticación, mensajería, etc.). En algunos modelos de servicio se ofrece la plataforma de desarrollo y las herramientas de programación por lo que puede desarrollar aplicaciones propias y controlar la aplicación, pero no controla la infraestructura.
- Infraestructura como servicio (IaaS): Es un medio de entregar almacenamiento básico y capacidades de cómputo como servicios estandarizados en la red. A veces también son llamados Hardware as a Service (HaaS). Un ejemplo de esto es AWS (Amazon Web Services).

■ Métodos de construcción de software distribuido

Se distinguen por cuestiones de paralelismo, comunicación y tolerancia a fallas.

■ Construcción de sistemas heterogéneos

Los sistemas heterogéneos consisten en una variedad de unidades computacionales especializadas de diferentes tipos. En general son independientes pero se comunican entre sí. Se plantean cuestiones relacionadas a los distintos lenguajes utilizados, así como de aspectos sobre la comunicación.

- **Análisis de desempeño (Performance) y Puesta a punto (Tuning)**
 - Análisis de desempeño: se estudia el comportamiento del programa en su ejecución.
 - Puesta a punto del código: involucra (en general) cambios menores para ajustar el desempeño del programa utilizando la información recolectada en el análisis de desempeño.
- **Estándares de plataforma**

Permiten que los programadores desarrollen aplicaciones portables que pueden ser ejecutadas en entornos compatibles sin realizar cambios. Por ej. J2EE.
- **Test-First Programming**

Test-First Programming o Test-driven development (TDD) es un estilo de programación en el cual se escriben las pruebas antes que el código.

6.5. Herramientas de Construcción

- **Entornos de desarrollo (IDEs)**

Proveen facilidades a los programadores para desarrollar software. La elección de un IDE puede afectar la eficiencia y la calidad de la construcción del software.
- **Herramientas de pruebas unitarias**

Permiten automatizar las pruebas unitarias. El programador especifica las entradas y salidas esperadas y la herramienta permite ejecutar un conjunto de pruebas indicando la salida obtenida.
- **GUI Builders**

Son herramientas que permiten el desarrollo y mantenimiento de interfaces de usuario gráficas. En general de un modo WYSIWYG (what you see is what you get – lo que ves es lo que obtienes).

6.6. Estándares de Programación

Un estándar o estilo de programación son convenciones y buenas prácticas para escribir código fuente en determinados lenguajes de programación.

- Fuerte dependencia del lenguaje de programación
- La mayoría del software es desarrollado/mantenido por equipos
- Aunque se trabaje en forma individual, definir un estilo de codificación ayuda a organizarse.
- Aportan a la “mantenibilidad” del Software
- Priorizar el código “legible”

Tenemos los siguientes estándares:

- Nomenclatura: como nombrar constantes, variables, tipos de datos, procedimientos, funciones, clases, módulos, controles, etc.
- Indentación de código: tener una convención de como se debe de indentar el código.
- Mensajes de alerta/error.
- Manejo de excepciones
- Estructuras de Control (bucles, iteraciones)
- Acceso a base de datos
- Diseño de menús y atajos por teclado
- Comentarios de código
- Espacios y líneas en blanco
- Operaciones de bloqueo deben incluir un timeout
- Operaciones que toman tiempo deben mostrar indicador de avance o estado procesando

6.7. Reutilización de código

Administración de una Base de Conocimiento o Biblioteca de Código. Tenemos dos roles:

- Productor: identificar que producir y definir características.
- Consumidor: identificar que puede servir y evaluar si efectivamente sirve.

6.8. Documentación del código

■ Documentación interna

Documentación, comentarios internos al código fuente. Es fundamental para el mantenimiento.

■ Documentación externa

Documentación técnica. Existen herramientas automáticas para generarla. Puede incluir el problema, algoritmos, datos e interfaces.

6.9. Programación en Pares

Es una práctica que reúne dos programadores a trabajar en forma conjunta. Uno tiene el rol de “conductor” (el que escribe) y el otro de “acompañante” (el que revisa).

Su éxito depende de la combinación de las habilidades y personalidades del par. La mayor crítica a esta práctica es que es difícil aseverar si afecta en forma positiva o negativa a la productividad.

6.10. Debug

If debugging is the process of removing software bugs, then programming must be the process of putting them in. (Edsger Dijkstra)

Determinar los pasos para reproducir un bug. Al encontrar un punto dónde sabemos que el sistema estaba bien y un lugar dónde falló tenemos un pedazo de código a revisar.

- Usar aserciones o revisar invariantes, por ej. imprimiendo mensajes (mejor aserciones o herramientas del IDE).
- Búsqueda binaria, para dividir el pedazo de código a la mitad.
- Arqueología de software

Luego análisis causal y aprender. Usar debuggers.

En el caso de que el bug no sea reproducible:

- Llevar registro del bug, contexto, consecuencias para buscar patrones en el futuro.
- Considerar agregar aserciones o logs para registrar más información.
- Si es un problema grave buscar tener un ambiente lo más parecido al cual dio problemas.

7. Verificación y Validación

7.1. Terminología e Introducción

Un **error humano** puede provocar un **defecto** o **falla** (interna), la cual puede provocar una **falla** (externa).

El software **falla cuando no hace lo requerido**, o cuando **hace algo que no debería**. Esto puede suceder porque:

- Las especificaciones no estipulan exactamente lo que el cliente precisa o quiere (requerimientos faltantes o incorrectos).
- Requerimientos no se pueden implementar.
- Defectos en el diseño.
- Defectos en el código.

Se busca detectar y corregir estos defectos antes de liberar el producto.

Es importante aclarar que testing no es debugging.

- Probar (testing) es una simple revisión del comportamiento del software mediante su ejecución.
- Debugging es la actividad de localizar el defecto en el código y corregirlo.

Algunos objetivos de la Verificación y Validación son:

- Ejecutar el programa para provocar fallas (una forma de encontrar defectos).
- Ejecutar el programa para medir su calidad.
- Ejecutar el programa para generar confianza.
- Analizar/revisar el programa o su documentación para detectar defectos (y prevenir fallas).

Otros conceptos importantes son los siguientes:

- Objetivo de prueba o tipo de prueba
- Técnica de prueba
- Objeto de prueba
- Nivel de prueba
- Perfil de persona que prueba
- Grado de prueba
- Bases de prueba
- **Caso de prueba:** Un caso de prueba contiene condiciones del test. En la mayoría de los casos esto son **precondiciones, entradas, y el resultado o comportamiento esperado del objeto de prueba.**

7.1.1. Definición de Verificación y Validación

- **Verificación:** ¿estamos construyendo el producto correctamente?
 - Busca comprobar que el sistema cumple con los requerimientos especificados (funcionales y no funcionales).
 - ¿El software está de acuerdo con su especificación?
 - Testing de defectos.

- **Validación:** ¿estamos construyendo el producto correcto?
 - Busca comprobar que el software hace lo que el usuario espera.
 - ¿El software cumple las expectativas del cliente?
 - Testing de validación.

7.1.2. Principios de la verificación y validación

1. El testing muestra la presencia de defectos, no la ausencia de ellos.
2. El testing exhaustivo no es posible.
3. Las actividades de verificación deberían comenzar lo antes posible.
4. Agrupamiento/aglomeración de defectos (defect clustering): los defectos se concentran en determinadas partes. Esto puede ser en las partes complejas, las partes core, etc. Conviene clasificar donde fue encontrado un defecto y si hay muchos en la misma parte considerar un refactor.
5. La paradoja del pesticida: si utilizo siempre la misma técnica para controlar, cada vez nos volvemos menos efectivos.
6. El testing es contexto-dependiente: puede pasar que en un ambiente funcione bien, pero en otro no lo haga. (“en mi maquina funciona”)
7. Decir que el software es útil porque “no se experimentan fallas” es una falacia. La utilidad del software viene dada por el usuario. Que el software no tenga fallas no quiere decir que sea útil para el cliente.

7.1.3. Algunos defectos

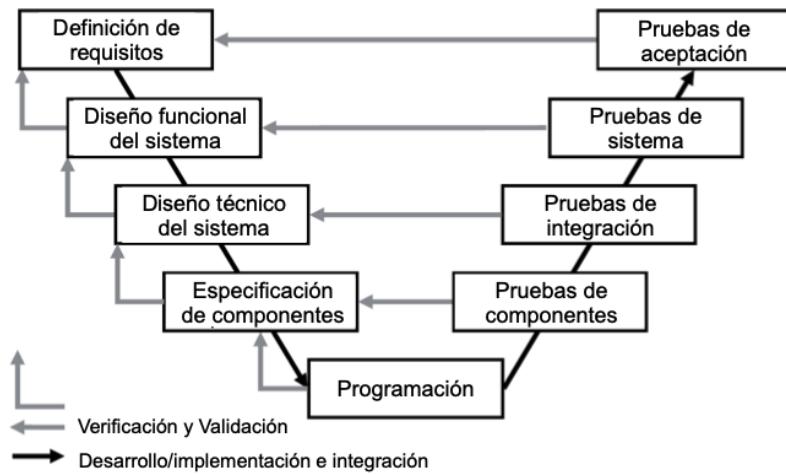
- en algoritmos
- de sintaxis
- de precisión y cálculo
- de documentación
- de estrés o sobrecarga
- de capacidad o de borde
- de sincronización o coordinación
- de capacidad de procesamiento o desempeño
- de recuperación
- de estándares y procedimientos
- relativos al hardware o software del sistema

7.1.4. Clasificación de defectos

- Categorizar y registrar los tipos de defectos
 - Guía para orientar la verificación: si conozco los tipos de defectos en que incurre el equipo de desarrollo me puedo ocupar de buscarlos expresamente.
 - Mejorar el proceso: si tengo identificada la fase en la cual se introducen muchos defectos me ocupo de mejorarla.
- Clasificación Ortogonal
 - Por ejemplo: tipo de defecto, fuente, impacto, disparador, severidad/criticidad.

7.2. Verificación y Validación en el ciclo de vida del Software

7.2.1. Proceso en V



El proceso en V muestra los distintos niveles de prueba. A medida que vamos avanzando en el desarrollo, ya podemos ir planificando distintos niveles de pruebas.

7.2.2. Niveles de prueba

- **De módulo, componente o unitaria:** verifica el funcionamiento de los componentes de acuerdo a su especificación
 - **De integración:** verifica que los grupos de componentes interactúan de la forma en la cual fue especificada acorde al diseño técnico del sistema.
 - **De sistema:** determina si el sistema integrado (como un todo) cumple con los requisitos especificados.
 - **De aceptación:** verifica que el sistema cumple con los requisitos del cliente de acuerdo a como los mismos fueron especificados en el contrato y/o el sistema cumple con las necesidades y expectativas del cliente.

Las pruebas unitarias son normalmente realizadas por el equipo de desarrollo. Generalmente lo hace la persona que lo implementó ya que tiene conocimiento detallado del módulo. Las pruebas de integración normalmente las realiza el equipo de desarrollo ya que se necesita conocimiento de las interfaces y funciones en general. El resto de las pruebas en general las realiza un equipo especializado. Para esto es necesario conocer los requerimientos y tener una visión global. Generalmente se necesita un equipo especializado debido a que:

- Maneja mejor las técnicas de pruebas.
 - Conoce los errores más comunes realizados por el equipo de programadores.
 - Conoce el modelo de negocio.
 - Problemas de psicología de pruebas
 - El autor de un programa tiende a cometer los mismos errores al probarlo.
 - Debido a que es “SU” programa inconscientemente tiende a hacer casos de prueba que no hagan fallar al mismo
 - Puede llegar a comparar mal el resultado esperado con el resultado obtenido debido al deseo de que el programa pase las pruebas

Es importante aclarar que esto depende el proceso y el equipo.

7.3. Pruebas de componentes

Son el primer nivel de prueba: verificación unitaria o de componentes.

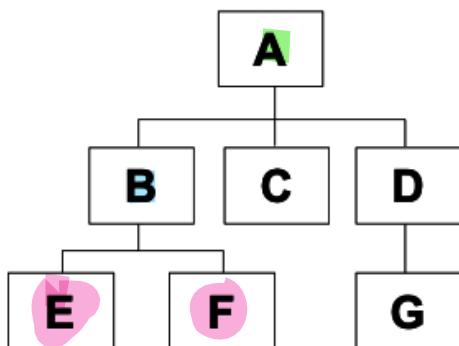
- **Objetos de prueba:** clases, módulos, scripts, etc.
- **Entorno de prueba:** los objetos de prueba vienen directamente del “escritorio del programador”, por lo tanto requiere un alto nivel de cooperación con desarrollo.
- **Objetivos de prueba:** comprobar que toda la funcionalidad del objeto de pruebas es completa, se desempeña y funciona adecuadamente de acuerdo a su especificación

Es importante probar la robustez de los componentes (tests negativos), la eficiencia (uso de recursos) y la mantenibilidad (estructura del código, modularidad, cumplimiento de estándares).

7.3.1. Estrategias de pruebas

- **Técnicas estáticas (analíticas):** analizar el producto (p.e. programa) para deducir su correcta operación
- **Técnicas dinámicas:** experimentar con el comportamiento de un producto para ver si el producto actúa como es esperado
 - **Testing de caja blanca:** basado en el código fuente del programa, el cual se utiliza para diseñar los casos de prueba.
 - **Testing de caja negra:** basado en la especificación del programa

7.3.2. Módulos y componentes



Si queremos probar el módulo B:

- El módulo B es usado por el módulo A, por lo que debo simular la llamada del módulo A al B (driver).
- El módulo B usa a los módulos E y F. Cuando llamo desde B a E o F debo simular la ejecución de estos módulos (stub).

7.3.3. Drivers y Stubs

Stub (simula la actividad del componente omitido)

- Es una pieza de código con los mismos parámetros de entrada y salida que el módulo faltante pero con una alta simplificación del comportamiento. Observar que de todas formas, tiene un costo de realización.
- Por ejemplo puede producir los resultados esperados leyendo de un archivo, pidiéndole de forma interactiva a un testeador humano, o no hacer nada en caso de que sea aceptable.
- Si el stub devuelve siempre los mismos valores al módulo que lo llama es probable que este módulo no sea testeado adecuadamente.

Driver

- Pieza de código que simula el uso (por otro módulo) del módulo que está siendo testeado. Es menos costoso de realizar que un stub.
- Puede leer los datos necesarios para llamar al módulo bajo test desde un archivo, GUI, etc.
- Normalmente es el que suministra los casos de prueba al módulo que está siendo testeado.

7.4. Pruebas de integración

Son el segundo nivel de prueba: testing de integración de unidades o componentes.

- **Objetos de prueba:** conjunto relacionados de clases, módulos, scripts, etc.
- **Entorno de prueba:** se utiliza también el entorno de desarrollo y los drivers y stubs ya disponibles (si fuera necesario).
- **Objetivos de prueba:** revelar problemas de interfaz, así como también conflictos entre los componentes integrados. Es importante probar el intercambio de datos y comunicaciones.

7.4.1. Estrategia de pruebas

¿En qué orden y de qué forma deberíamos probar la integración de los componentes? En la práctica, los diversos componentes van a estar disponibles en momentos diferentes. Una práctica común es realizar el test de integración a medida que los componentes están disponibles. El responsable de las pruebas debe elegir una estrategia de integración que optimice:

- Tiempo que se insume (ahorro del tiempo).
- Costo del entorno de pruebas.

Tenemos dos tipos de estrategias:

- No incremental
 - Big-Bang
- Incrementales
 - Bottom-Up (Ascendente)
 - Top-Down (Descendente)
 - Por disponibilidad (Ad hoc)
 - Integración Backbone (esqueleto)

El objetivo es lograr combinar módulos o componentes individuales para que trabajen correctamente de forma conjunta.

La elección está sujeta por varias restricciones como lo son:

- La arquitectura del sistema
- El plan de proyecto
- El plan de pruebas

La estrategia de pruebas puede ser un mix de varias alternativas distintas según aspectos de:

- Riesgo
- Disponibilidad
- Costo
- Otros...

7.5. Pruebas de sistema

Son el tercer nivel de prueba: testing de sistema, en donde se comprueba que el sistema cumple con los requisitos especificados.

- **Objetos de prueba:** el sistema integrado como un todo
- **Entorno de prueba:** lo más cercano posible al ambiente de producción. Un error común es utilizar el ambiente de producción.
- **Objetivos de prueba:** validar que el sistema completo cumple con la especificación de sus requisitos funcionales y no funcionales. Es importante verificar la documentación y omisión de requisitos.

Si no tenemos los requisitos documentados, se puede probar igual. Se debe buscar la forma de entender sobre el sistema para poder probarlo. Muchas empresas dedicadas al testing hacen esto.

7.6. Pruebas de aceptación

Son el cuarto nivel de prueba: testing de aceptación, en donde se comprueba que el sistema es adecuado al uso y necesidades del cliente (puede realizarse como parte de las pruebas de niveles inferiores o distribuido en varios niveles de pruebas).

- **Objetos de prueba:** el sistema (o parte de este) bajo la perspectiva del usuario/cliente.
- **Entorno de prueba:** entorno de producción.
- **Objetivos de prueba:** validar que construimos el producto “correcto”.
- **Bases de prueba:** cualquier documento que describa el sistema desde el punto de vista del usuario/cliente: casos de uso, procesos de negocio, user stories, etc.

7.6.1. Tipos de pruebas de aceptación

▪ Pruebas de aceptación contractuales

Si se construyó software a medida, el cliente realiza prueba de aceptación contractuales para determinar si el sistema construido está libre de deficiencias y si lo que se acordó en el contrato fue logrado y es aceptable. En casos de desarrollo de software interno, esto puede ser un contrato menos formal entre el departamento de usuarios y el departamento de tecnología.

▪ Pruebas de aceptación de usuario

Un aspecto a considerar es los tests para la aceptación del usuario. Se recomiendan si el cliente y el usuario son distintos. Diferentes grupos tienen usualmente completamente diferentes expectativas del sistema, por lo que es importante organizar pruebas de aceptación de usuario para cada uno de estos grupos. Si se encuentra una cantidad muy grande de problemas en estos tests, generalmente es muy tarde para solucionar el problema. Es por esto que se recomienda dejar a un selecto grupo de usuarios de cada grupo examinar prototipos en versiones tempranas del sistema.

▪ Pruebas de aceptación operacionales

Este tipo de pruebas asegura la aceptación del sistema por parte de administradores. Puede incluir pruebas de backup/ciclos de restauración, recuperación de desastres, manejo de usuarios y chequeos de vulnerabilidades de seguridad.

▪ Pruebas de campo (alpha/beta testing y dogfood tests)

Si el software debe correr en varios ambientes, es muy costoso o incluso imposible crear ambientes de prueba para cada uno de ellos durante las pruebas. En estos casos se puede optar por realizar pruebas de campo luego de las pruebas del sistema. El objetivo de estas pruebas es identificar influencias de los ambientes que no son enteramente conocidos, y eliminarlas de ser necesario.

7.7. Pruebas de nuevas versiones del sistema

El sistema ahora está en producción. Los problemas no terminan, sino que empiezan.

- Todo lo que vimos anteriormente en el principio del ciclo de vida del software.
- Luego de la instalación de la primera versión del software, el mismo puede ser usado por años o décadas, siendo cambiado, actualizado y extendido muchas veces.
- El software no se gasta con el uso, ni los defectos se originan por dicha causa. Sin embargo, los bug-fixing puede hacer que se vaya degradando.
- El mantenimiento puede ser correctivo, adaptativo o evolutivo.

7.8. Pruebas de regresión

Tanto en el mantenimiento como en procesos iterativos incrementales, es necesario realizar pruebas de regresión: verificar que los cambios o agregados realizados a la nueva versión no hayan degradado o “roto” el funcionamiento del sistema.

7.9. Tipos de pruebas y técnicas de generación de casos de prueba

7.9.1. Tipos genéricos de pruebas

- **Pruebas funcionales:** verifican las entradas/salidas del sistema. Las técnicas de caja negra (basadas en la especificación) son las más usadas. Son basadas en los requisitos y en los procesos de negocio.
- **Pruebas no funcionales:** verifican en qué grado se cumplen los requisitos no funcionales (confiabilidad, usabilidad y eficiencia). Se pueden usar las pruebas funcionales para recrear el escenario donde la característica no-funcional debe ser medida.
- **Pruebas de la estructura del software:** se basan en la estructura de los artefactos (código, requisitos, arquitectura). Las técnicas de caja blanca son las más usadas.
- **Pruebas relacionadas a los cambios:** Un ejemplo de estas son pruebas de regresión las cuales se usan para verificar que nuevas faltas no se introdujeron o se enmascararon con los cambios realizados.

7.9.2. Técnicas de verificación

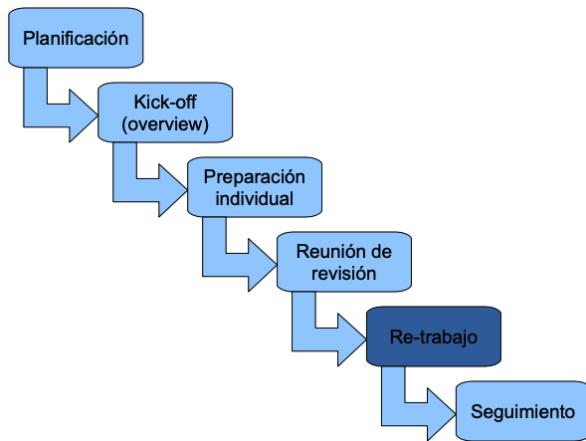
- **Técnicas estáticas (análisis):** analizar el producto para deducir su correcta operación. Por ejemplo *Análisis de código* y *Análisis estático*.
- **Técnicas dinámicas (pruebas):** experimentar con el comportamiento de un producto para ver si el producto actúa como es esperado. Por ejemplo *Caja negra*, *Caja blanca* y *Basadas en la experiencia*.

7.10. Pruebas estáticas

7.10.1. Análisis de código

También llamadas “Structured group evaluations”. En ellas se revisa el código buscando defectos. Se puede llevar a cabo en grupos. Las técnicas más conocidas son **recorridas e inspecciones**.

Llamaremos **revisión** al término común utilizado para las diferentes técnicas de análisis estático realizado por personas. Se revisa el código buscando problemas en algoritmos y otras faltas. La siguiente imagen muestra el proceso general de una revisión.



Algunos roles y responsabilidades en las revisiones son los siguientes:

- **Líder (manager):** selecciona los artefactos a ser revisados, asigna los recursos necesarios y selecciona el equipo de revisión (generalmente no participa de la reunión de revisión).
- **Moderador:** es el responsable de ejecutar la revisión (cada una de sus etapas). Organiza la discusión.
- **Autor:** presenta y explica el código (o documento).
- **Revisores:** expertos técnicos que participan de la revisión. → Devs
- **Secretario:** escribe el reporte de la reunión (problemas, acciones a tomar, decisiones y recomendaciones).

En relación al objeto que se examina, podemos distinguir dos tipos de revisiones:

- Revisión de **productos** generados a partir del proceso de desarrollo del software.
- Revisión del **proyecto o del proceso** de desarrollo en sí (management reviews).

A su vez, dentro de las revisiones de productos, podemos distinguir los siguientes tipos:

- **Recorridas (Walkthroughs)**
Es un método informal y manual. Su objetivo principal es descubrir defectos, problemas y ambigüedades. Otros objetivos son educar a la audiencia sobre el producto, aprendizaje mutuo, mejorar el producto, discutir formas alternativas de implementación, etc. En esta técnica se “recorre” el producto (o se simula su comportamiento en caso de ser código) en busca de defectos. Es bastante más informal que otros tipos de revisiones (inspecciones por ejemplo).
- **Inspecciones (inspections)**
Es uno de los métodos más formales de revisión. Sigue un proceso formal y prescriptivo. El objetivo es encontrar defectos y elementos poco claros, midiendo la calidad del objeto bajo revisión y mejorar la calidad del proceso de inspección y de desarrollo de software. Se examinan artefactos (no solo aplicables al código) buscando defectos comunes. Los revisores (inspectores) utilizan listas de comprobación (checklists). Estas listas dependen del lenguaje de programación y de la organización. Por ejemplo revisan el uso de variables no inicializadas y asignaciones de tipos no compatibles.
- **Revisiones técnicas**
Expertos técnicos hacen las revisiones. Son altamente formales. Requieren un gran esfuerzo de preparación.
- **Revisiones informales**
Es una versión ligera de una revisión. Sigue el procedimiento general en una forma simplificada. En la mayoría de los casos el autor inicia el proceso. La planificación es simplemente elegir quien formará parte de la revisión y pedirles su opinión. Usualmente no hay reunión o

intercambio entre lo que se descubre. En estos casos la revisión es simplemente una revisión cruzada entre uno o más colegas. Los resultados no tienen porque ser explícitamente documentados. Tipos de revisiones informales son programación entre pares, “buddy testing” y “code swapping”. Las revisiones informales son muy comunes y ampliamente aceptadas debido al mínimo esfuerzo y costo que se requiere.

Algunos datos interesantes respecto al análisis de código

- **Fagan:** Con Inspección se detectó el 67 % de las faltas detectadas. Al usar Inspección de Código se tuvieron 38 % menos fallas (durante los primeros 7 meses de operación) que usando recorridas.
- **Ackerman et. al:** Reportaron que 93 % de todas las faltas en la aplicación de negocios fueron detectadas a partir de inspecciones.
- **Jones:** Reportó que inspecciones de código permitieron detectar el 85 % del total de faltas detectadas.

7.10.2. Análisis estático

Cuyo objetivo es (al igual que las revisiones) encontrar defectos o partes propensas a defectos en los documentos. La diferencia está en que la revisión la hace una herramienta. Para esto el documento a ser analizado debe de seguir cierta estructura formal para poder ser chequeado por la herramienta. Decimos que es estático ya que no requiere la ejecución del código.

Podemos verlo como un complemento al compilador. Mayormente es un análisis sintáctico de:

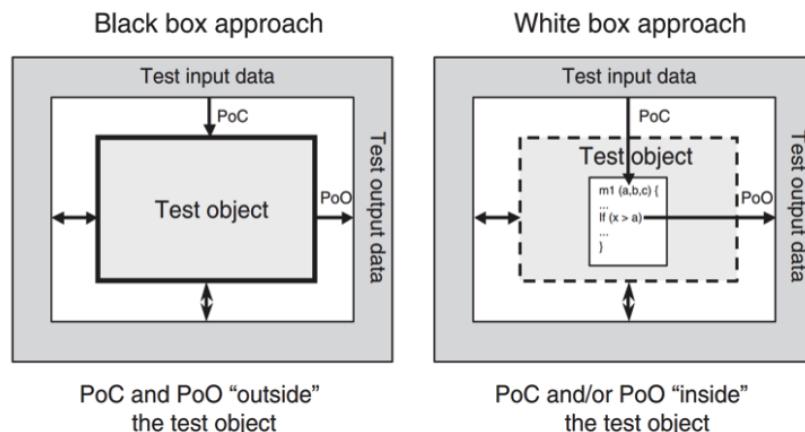
- Instrucciones bien formadas (violación de sintaxis)
- Desviación de estándares.
- Anomalías en el flujo de control o de datos.

7.11. Pruebas dinámicas

Permiten probar el sistema ejecutando el objeto de pruebas. Podemos ver tres categorías generales (refieren al diseño de los CP):

- Pruebas de caja negra (también llamadas pruebas basadas en la especificación).
- Pruebas de caja blanca (también llamadas pruebas estructurales).
- Pruebas basadas en la experiencia.

7.11.1. Caja negra vs Caja blanca



7.11.2. Técnicas de caja negra

La estructura inherente o diseño del objeto de prueba no es considerado para el diseño de los casos de prueba.

Algunas técnicas son las siguientes:

- Particiones en clases de equivalencia.
- Análisis de valores límites.
- Pruebas de transición de estados.
- Pruebas basadas en la lógica (grafos causa efecto, tablas de decisión y pairwise testing).
- Pruebas basadas en casos de uso.

7.11.3. Caja negra - Particiones en clases de equivalencia

Definimos una **clase de equivalencia** como un **conjunto** de entradas para las cuales suponemos que el software se comporta igual.

El proceso de partición de equivalencia tiene los siguientes pasos:

1. Identificar el dominio de entrada y de salida
2. Identificar las clases de equivalencia del dominio de entrada
3. Definir los casos de prueba
Un buen caso de prueba reduce significativamente el número de los otros casos de prueba. Además cubre un conjunto extenso de otros casos de prueba posibles.

Para identificar las clases de equivalencia:

- Cada condición de entrada separarla en 2 o más grupos.
- Identificar las clases válidas así como también las **clases inválidas**.
- Si se cree que ciertos **elementos** de una clase de equivalencia **no** son tratados de forma **idéntica** por el programa, dividir la clase de equivalencia en clases de equivalencia diferentes.

Proceso de definición de los casos de prueba

1. Asignar un **número único** a cada clase de equivalencia.
2. Hasta **cubrir todas las clases de equivalencia con casos de prueba**, escribir un nuevo caso de prueba que cubra tantas clases de equivalencia válidas, no cubiertas, como sea posible.
3. Escribir un caso de prueba para cubrir una y solo una clase de equivalencia para cada clase de equivalencia inválida (evita cubrimiento de errores por otro error).

7.11.4. Caja negra - Análisis de valores límites

La experiencia muestra que los casos de prueba que **exploran las condiciones límite** producen mejor resultado que aquellas que no lo hacen. Las condiciones límite son aquellas que se hallan "arriba" y "debajo" de los márgenes de las clases de equivalencia de entrada y de salida.

Las diferencias con partición de equivalencia son:

- Elegir casos tal que **los márgenes de las clases de equivalencia sean probados** (el límite y los adyacentes a ambos lados).
- Se debe **tener en cuenta** las clases de equivalencia de la salida (esto también se puede considerar un **particiones de equivalencia**).

Se debe **usar el ingenio** para encontrar condiciones límite.

7.11.5. Caja negra - Pruebas de transición de estados

En esta técnica se toma en consideración la historia de ejecución del sistema, se usan diagramas de estados para ilustrar la dependencia en la historia. El sistema u objeto de testeo comienza en un estado y puede luego cambiar de estado debido a distintos eventos que promueven la transición.

7.11.6. Caja negra - Pruebas basadas en la lógica

Son pruebas donde los valores de entrada son considerados todos por separado al generar casos de prueba. No se consideran explícitamente dependencias entre distintas entradas y sus efectos en la salida para el diseño de los tests.

7.11.7. Caja negra - Pruebas basadas en casos de uso

Seguimos los siguientes pasos para la generación:

1. Para cada caso de uso, generar el conjunto de todos los escenarios posibles.
Para hacer esto podemos ver los distintos caminos entre el flujo principal y alternativos.
2. Para cada escenario, identificar por lo menos un caso de prueba y las condiciones que lo harán ejecutable.
3. Para cada caso de prueba, identificar los datos de prueba a utilizar.

Ejemplo: Dado el siguiente caso de uso

Nombre:	Inscripción a cursos
Precondiciones: El estudiante se encuentra logueado en el sistema.	
Flujo Principal:	
1.El Estudiante selecciona la opción “Crear una agenda”. 2.El sistema devuelve la lista de cursos disponibles que ofrece el Sistema de Catálogo de Cursos y despliega una lista al estudiante. 3.El estudiante selecciona cuatro cursos primarios ofrecidos y 2 cursos alternativos de la lista de cursos disponibles. 4.El estudiante indica que la agenda está completa. 5.El sistema verifica que el estudiante tiene los pre-requisitos necesarios para la inscripción a cada curso seleccionado en la agenda. 6.El sistema despliega la agenda conteniendo los cursos ofrecidos y un mensaje de confirmación de que se creó la agenda de forma exitosa junto con un número de confirmación .	
Flujos Alternativos:	
G1. Salir sin confirmar agenda G1.1. El estudiante elige guardar la agenda de forma parcial antes de salir. G1.2. El sistema guarda la agenda. G1.3. Fin del CU. 2.a. Sistema de Catálogo de Cursos no disponible 2.a.1. El sistema despliega un mensaje de error. 2.a.2. Fin del CU. 2.b. Inscripciones cerradas 2.b.1. El sistema despliega un mensaje indicando que las inscripciones están cerradas. 2.b.2. Fin del CU. 6.a. El estudiante no completa los requisitos, Curso lleno o Conflicto en la Agenda 6.a.1.El sistema determina que el estudiante no cumple con los pre-requisitos, el curso está lleno o hay conflictos de agenda. 6.a.2. El sistema despliega un mensaje al estudiante indicando que debe seleccionar un curso distinto. 6.a.3. Sigue en el punto 3 del FP.	

Generamos los escenarios

Escenario	Flujo Inicial	Flujo Alternativo	Flujo Alternativo
1	FP		
2	FP	2A	
3	FP	2B	
4	FP	G1(2)	
5	FP	6A	G1(3)
6	FP	6A	
7	FP	G1(3)	

Identificamos los casos de prueba con las condiciones

Esc.	SCC disp.	Insc. ab	Est. c/r	Curso c/c	No conf. ag.	Conf.	Sal. s/g (2)
1	V	V	V	V	V	V	F
2	F	N/A	N/A	N/A	N/A	F	F
3	V	F	N/A	N/A	N/A	F	F
4	N/A	N/A	N/A	N/A	N/A	F	V
5C1	V	V	F	V	V	F	F
5C2	V	V	V	F	V	F	F
5C3	V	V	V	V	F	F	F
6C1	V	V	F	V	V	V	F
6C2	V	V	V	F	V	V	F
6C3	V	V	V	V	F	V	F
7	V	V	N/A	N/A	N/A	F	F

Para cada caso de prueba identificamos los datos de prueba

Se crea una tabla con los datos necesarios para probar cada uno de los casos de prueba identificados anteriormente.

7.11.8. Técnicas de caja blanca

La derivación de los casos de prueba se basa en la estructura inherente o diseño del objeto de prueba. Son también llamadas técnicas basadas en la estructura o en el código. Podemos diferenciar los siguientes tipos:

- **Basadas en el flujo de control del programa:** Expresan los cubrimientos del testing en términos del grafo de flujo de control del programa
- **Basadas en el flujo de datos del programa:** Expresan los cubrimientos del testing en términos de las asociaciones definición-uso del programa

El foco de las técnicas de caja blanca es ejecutar (cubrir) cada parte del código al menos una vez. El cubrimiento de ese código puede tener diferentes criterios:

- **Cubrimiento de sentencias**

Asegura que el conjunto de casos de pruebas (CCP) ejecuta al menos una vez cada instrucción del código.

- **Cubrimiento de decisión**

Cada decisión dentro del código toma al menos una vez el valor true y otra vez el valor false para el CCP.

- **Cubrimiento de condición**

Cada condición dentro de una decisión debe tomar al menos una vez el valor true y otra el valor false para el CCP.

- **Criterio de cubrimiento de decisión/condición**

Combinación de los dos criterios anteriores.

- **Criterio de condición múltiple**

Todas las combinaciones posibles de resultados de condición dentro de una decisión se ejecuten al menos una vez.

- **Criterio de caminos**

Se ejecutan al menos una vez todos los caminos posibles (combinaciones de trayectorias).

7.11.9. Técnicas basadas en la experiencia

No es un enfoque sistemático (como vimos anteriormente), sino que por el contrario, se basa fuertemente en la intuición, habilidades, conocimiento y experiencia del tester. Los casos de prueba se basan en datos históricos (errores/defectos del pasado) o en la intuición de dónde podrían estar en el futuro (error guessing). Si no existe documentación para utilizar como bases del testing (requisitos, casos de uso, etc.), como técnica puede utilizarse el testing exploratorio.

Testing Exploratorio

Se realizan las siguientes actividades en forma simultánea:

- Aprendizaje/exploración de la aplicación.
- Diseño de casos de prueba.
- Ejecución de casos de prueba.

Si bien no hay actividades explícitas de planificación, se puede utilizar el enfoque basado en sesiones. Para esto se debe definir el objetivo que queremos cumplir con la sesión de testing y limitar el tiempo dedicado a dicha sesión.

Los elementos del software bajo prueba son “explorados”.

- En una primera instancia se ejecutan unos pocos casos de prueba para obtener conocimiento acerca del software.
- Se deriva el comportamiento del software (qué es y cómo funciona, qué problemas de calidad podría tener y qué expectativas debería cubrir).
- En base a los resultados de sesiones anteriores se planifican nuevas sesiones utilizando una “hoja de ruta” para cada una.

Es una técnica altamente dependiente de las habilidades del tester. Es importante tener un pensamiento crítico, ser observadores y analíticos. Además se deberá tener o adquirir un conocimiento del negocio del software.

Preguntas importantes a realizarse en cada sesión de prueba:

- ¿Cuál es el objetivo?
- ¿Qué debe ser probado?
- ¿Qué método de prueba debo usar?
- ¿Qué tipo de problemas podría encontrar?

Consideraciones generales:

- Los resultados de los CP ejecutados influencian el diseño y ejecución de los CP que siguen a continuación
- Durante las pruebas se construye un modelo mental del SW
- Las pruebas se ejecutan contra ese “modelo” mental

7.12. Comparación de técnicas de pruebas

Factores que influencian que técnica de prueba utilizar:

- El tipo de objeto de prueba.
- Documentación formal y/o disponibilidad de herramientas.
- Cumplimiento de estándares.
- Experiencia de los testers.
- Necesidad/deseos del cliente.
- Evaluación de riesgos.

7.12.1. Proceso para un módulo

Jacobson sugiere ejecutar primero las pruebas de caja negra y cuando estas sean todas correctas completar con caja blanca. Esto se debe a que la caja blanca depende del código y cuando se detecten errores con caja negra el código cambia al corregir los errores detectados.

7.12.2. Comparación de técnicas

▪ Estáticas

Ventajas

- Efectivas en la detección temprana de defectos
- Sirven para verificar cualquier producto (requerimientos, diseño, código, casos de prueba, etc)
- Conclusiones de validez general

Desventajas

- Sujeto a los errores de nuestro razonamiento
- No se usan para validación (solo verificación)
- No consideran el hardware o el software de base

▪ Dinámicas

Ventajas

- Se considera el ambiente donde es usado el software (realista).
- Sirven tanto para verificar como para validar
- Sirven para probar otras características además de funcionalidad

Desventajas

- Está atado al contexto donde es ejecutado
- Su generalidad no es siempre clara (solo se aseguran los casos probados)
- Solo sirve para probar el software construido
- Normalmente se detecta un único error por prueba (los errores cubren a otros errores o el programa colapsa)

7.13. Pruebas no funcionales

Para este tipo de pruebas lo esencial es definir:

- Procedimientos de prueba.
- Criterios de aceptación.
- Características del ambiente.

7.13.1. Tipos de pruebas no funcionales

- **Prueba de carga:** como se comporta el sistema ante un aumento de la carga.
- **Prueba de performance (rendimiento):** se mide la velocidad de procesamiento y tiempos de respuesta para casos de uso particulares.
- **Prueba de volumen:** observación del comportamiento del sistema dependiendo la cantidad de datos.
- **Prueba de estrés:** observación del comportamiento del sistema cuando está sobrecargado.
- **Prueba de seguridad:** pruebas de seguridad antes accesos no autorizados o ataques de negación de servicio.
- **Prueba de confiabilidad:** pruebas sobre operaciones permanentes.
- **Prueba de robustez:** se mide la respuesta del sistema ante errores de operación, mala programación, etc. Se evalúa como se recupera de errores.
- **Prueba de compatibilidad y conversión de datos:** se examina la compatibilidad con sistemas existentes, importación/exportación de datos, etc.
- **Prueba de configuración:** se prueba el sistema con distintos sistemas operativos, lenguajes de interfaz de usuario, etc. (ej. back to back testing).
- **Prueba de facilidad de uso:** se examina la facilidad de aprender a usar el sistema, eficiencia en la operación y entendimiento de la salida del sistema.
- **Comprobación de documentación:** para el cumplimiento del comportamiento esperado.

7.14. Gestión de las pruebas (Test management)

Así como en el desarrollo, las actividades de testing necesitan ser organizadas. Las mismas deben ser coordinadas con las actividades de desarrollo.

Ya hablamos de los beneficios de que el testing sea independiente del desarrollo (sicología de las pruebas). Algunas opciones de organización serían:

- El equipo de desarrollo es el responsable de las pruebas, pero los desarrolladores se verifican mutuamente el código.
- Hay roles específicos de testers dentro del equipo de desarrollo.
- Uno o más equipos de testing dedicados al proyecto trabajan en conjunto con el equipo de desarrollo.
- Especialistas independientes en pruebas se encargan de algunas actividades específicas de testing (usabilidad, seguridad, estándares).
- Tercerización del testing en una organización separada.

7.14.1. Roles en las pruebas

- **Líder de testing (test manager):** planifica y da seguimiento a las actividades de verificación.
 - Selecciona y da soporte a las estrategias de pruebas a utilizar, procura y asigna los recursos a las mismas.
 - Identifica las métricas necesarias para dar seguimiento al progreso de las pruebas y controlar la calidad del producto de software.
- **Diseñador de pruebas (test analyst):** experto en los métodos/técnicas de prueba.
- **Automatizador de pruebas:** tiene conocimiento de testing así como también de programación y herramientas de automatización.

- **Administrador de pruebas:** expertos en instalar y operar el entorno de pruebas.
- **Probador (tester):** ejecuta y reporta los resultados de las pruebas.

7.14.2. Planificar la verificación

Las actividades de verificación se deben planificar en mayor o menor grado, dependiendo y siendo coherente con el enfoque del proceso de desarrollo. Esto permite que el personal técnico obtenga una visión global de las pruebas del sistema y ubique su propio trabajo en ese contexto.

La V&V es un **proceso caro** se requiere llevar una **planificación cuidadosa** para obtener el máximo provecho de las revisiones y las pruebas para controlar los costos del proceso de V&V. El proceso de planificación de V&V

- Pone en la **balanza los enfoques estático y dinámico** para la verificación y la validación.
- Utiliza **estándares y procedimientos para las revisiones y las pruebas de software**.
- Establece **listas de verificación para las inspecciones**.
- Define el **plan de pruebas de software**.

El mayor error cometido en la planificación de un proceso de prueba es **suponer que no se encontrarán fallas** al momento de realizar el cronograma. Esto lleva a que:

- Se subestima la cantidad de personal a asignar
- Se subestima el tiempo de calendario a asignar
- Entregamos el sistema fuera de fecha o ni siquiera entregamos

8. Liberación de software

Comprende todas las actividades que son necesarias, luego de que se ha construido y probado, para comenzar el uso de un software. No se trata solamente de desplegar el sistema, sino que incluye:

- Instalación y configuración.
- Adopción del software.
- Entrenamiento y apoyo en el uso.

8.1. Instalación y configuración

Se instala el software de forma que quede disponible y operativo. Su complejidad depende de:

- Tecnología utilizada
- Restricciones funcionales (por ejemplo temporales)
- Requisitos de disponibilidad

La facilidad de instalación afecta la liberación inicial y las sucesivas liberaciones durante el mantenimiento. Esta etapa muchas veces incluye:

- Migración o carga de datos iniciales
- Parametrización de procesos
- Carga de usuarios y asignación de permisos (o roles)

8.2. Adopción (o conversión)

Sustituir un sistema anterior por uno nuevo

- manual o automatizado
- carga inicial de
 - datos básicos
 - información histórica (calidad de datos)

También debemos definir una **estrategia de conversión**:

- **Big-bang**
En una fecha dada todos los módulos son instalados en la organización. Se debe planificar cuidadosamente
- **Paulatina**
Segmentación por módulo, por unidad de negocio, por localización. Tiene como desventaja que conviven varios sistemas. Tiene como ventaja que el ajuste de procedimientos es más sencillo.
- **Procesamiento en paralelo**
Dos sistemas funcionando a la vez, uno en producción y otro en prueba/control. Entrenamiento y validación en operación.
- **Estrategias híbridas**
Mezcla las estrategias vistas anteriormente. Por ejemplo Big-bang en un área de negocio y paulatina en el resto.

Finalmente, se debe tener un **plan de contingencia** para definir qué hacer en casos de que el nuevo sistema impida la operativa o tenga fallas importantes.

8.3. Entrenamiento y apoyo en el uso

8.3.1. Entrenamiento para cubrir perfiles y necesidades

Podemos distinguir dos grupos a entrenar:

- **Usuarios finales:** deben saber que hace el sistema y cómo usarlo.
- **Administradores y operadores:** funciones de soporte. Deben saber cómo funciona el sistema.

Podemos también distinguir diferentes necesidades de entrenamiento:

- Usuarios frecuentes o eventuales: si el sistema no se usa frecuentemente es fácil olvidarse como usarlo.
- Nuevos usuarios
- Usuarios experimentados que necesitan repasar
- Entrenamiento especializado (generalmente para funcionalidades nuevas).

8.3.2. Soporte para el entrenamiento

- **Documentos**

Se debe cuidar el tamaño y calidad, y la relación costo/beneficio de mantenerlos. Solo pocos usuarios lo leen y más al comienzo del uso.

- **Ayuda en línea**

- **Demostraciones, talleres y simulacros de uso**

Son más individualizados, se puede utilizar distintas dinámicas. Hay que definir cuándo hacerlos.

- **Usuarios expertos**

Pueden dar primer nivel de ayuda.

- **Documentación**

Facilita soporte y solución de problemas. Su importancia depende de cantidad de usuarios y su dispersión / diversidad.

Atributos de calidad son legibilidad, completitud, correctitud.

Se deben atender distintos roles:

- manual de usuario: propósito general y funcionalidades.
- manual de operadores: configuración hw+sw, acceso a usuarios, solución de problemas
- guía general del sistema: detalles que el cliente comprenda, configuración hw+sw, filosofía detrás del sistema
- guía para el desarrollador

8.3.3. Revisión del entrenamiento

Se debe evaluar el entrenamiento lo cual incluye:

- **Grado de uso del sistema**
- **Eficiencia en el uso**
- **Cumplimiento de objetivos**

El entrenamiento debe tomar en cuenta:

- características y preferencias personales
- estilos de trabajo
- presiones de la organización

8.3.4. Apoyo durante el uso del sistema

Una vez terminada la capacitación, se debe también proveer apoyo a lo largo del uso del software. Tenemos varias alternativas:

- Guía de mensajes de error.
- Guía rápida.
- Ayuda en línea (mesa de ayuda). Por ejemplo llamadas telefónicas, sms, whatsapp, etc.
- Proceso de gestión de incidentes – niveles de servicio.
- Funcionalidades para deshacer y procedimientos de contingencia.

9. Evolución de Software

El software evoluciona porque necesita

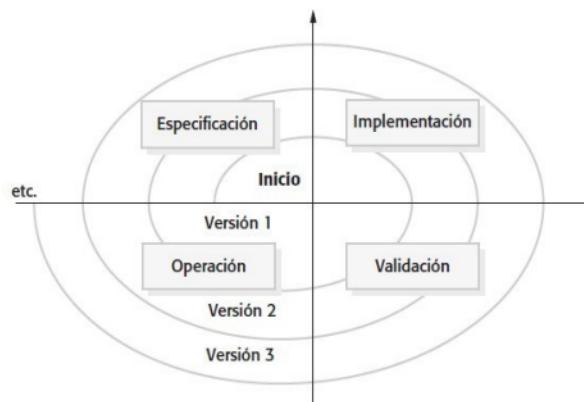
- Adaptarse al entorno operativo
- Adaptarse al usuario
- Mejorar en el desempeño
- Mejorar en funcionalidades
- Corrección de errores

El software requiere continua adaptación, mantenimiento y mejoras para seguir siendo útil y valioso para la organización.

Los sistemas de software representan activos críticos del negocio de las organizaciones. Para mantener el valor, los sistemas deben ser modificados y actualizados. La mayor parte del presupuesto está dedicada al cambio y a la evolución del software existente.

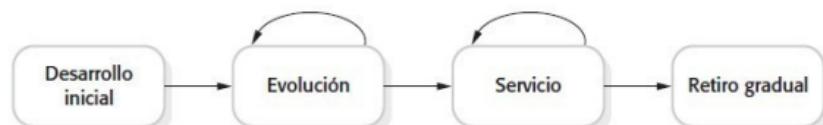
9.1. Modelos de evolución

9.1.1. Modelo en espiral de desarrollo y evolución



El tiempo entre iteraciones puede ser bastante corto. Cuando las transición de desarrollo a evolución está bien marcada, hablamos de mantenimiento de software.

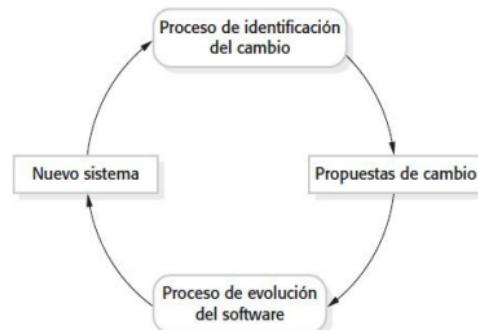
9.1.2. Modelo de Evolución y Servicio



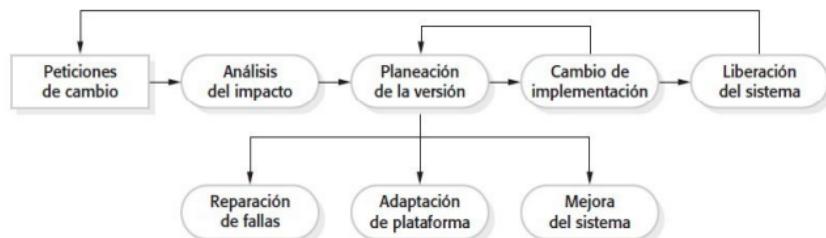
- **Evolución:** el sistema vive en un ciclo donde se encuentra en su uso operacional y evoluciona cuando los nuevos requisitos son propuestos e implementados en el sistema.
- **Servicio:** el software sigue siendo útil pero los únicos cambios realizados son para mantenerlo operativo. Por ejemplo corrección de errores y cambios producto de modificaciones en el entorno del software. No se agregan nuevas funcionalidades.
- **Retiro gradual:** el software puede ser utilizado todavía pero no se realizan más cambios sobre el mismo.

9.2. Procesos de Evolución

Las propuestas de cambio guían la evolución del software. Sugieren cambios a un sistema existente. Se basan en: requisitos preexistentes que no fueron realizados antes de la liberación, nuevos requisitos, errores reportados o nuevas ideas del equipo de desarrollo. Se deben identificar y relacionar los componentes que son afectados por un cambio, permitiendo así estimar el costo y el impacto del cambio.



La identificación de los cambios y la evolución continúan a lo largo del tiempo de vida del sistema.



■ Análisis de impacto

El costo y el impacto de los cambios se valoran para saber qué tanto resultará afectado el sistema y cuánto costaría. Cuando el equipo de mantenimiento es diferente al de desarrollo, la implementación incluye una comprensión del programa. Se planifica una nueva versión con los cambios aceptados.

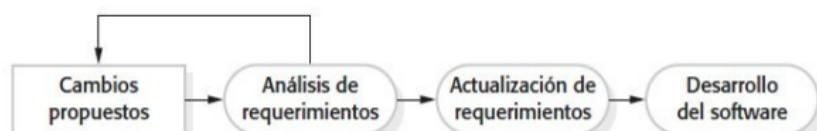
■ Planeación de la versión

La planificación considera todos los cambios propuestos. Se decide cuáles cambios implementar en la siguiente versión.

■ Liberación del sistema

Después de implementarse, se valida y se libera una nueva versión. Luego, el proceso se repite con un conjunto nuevo de cambios propuestos para la siguiente liberación.

Respecto a la implementación del cambio



Durante el proceso, se analizan los requerimientos y pueden surgir modificaciones a los cambios propuestos. En la primera etapa de la implementación del cambio puede implicar entender el programa, especialmente si los desarrolladores no son responsables de la implementación del sistema desarrollado originalmente. Durante esta etapa, se debe entender cómo el programa está estructurado, cuál es la forma en que ofrece funcionalidad y cómo el cambio propuesto puede afectar el programa.

9.2.1. Cambios urgentes

Los cambios urgentes pueden tener que ser implementados sin pasar por todas las etapas del proceso de ingeniería de software.

- Si es una falla grave, el sistema debe ser reparado para permitir que continúe la operación normal del sistema.
- Si cambios en el entorno del sistema tienen efectos inesperados.
- Si hay cambios en el negocio que requieren una rápida respuesta (ej.: la competencia lanza un producto).

Para realizar los cambios de manera rápida, se elige una solución rápida y viable. En general, no es la mejor solución. Existe el riesgo de no actualizar la documentación.

Se puede registrar un cambio no urgente para rehacerlo. Por ejemplo *Política del día después*. Es decir, todos los pasos que nos salteamos hoy los dejamos para mañana.

9.3. Evolución en métodos ágiles

Los métodos ágiles se basan en un desarrollo incremental, por lo cual la transición del desarrollo a la evolución es perfecta. La evolución es una simple continuación del proceso de desarrollo basado en frecuentes liberaciones del sistema.

Las pruebas de regresión automatizadas son muy valiosas cuando se realizan cambios en el sistema. Los cambios pueden ser expresados como historias de usuario adicionales.

9.3.1. Escenarios de transferencia

- Desarrollo con enfoque ágil, evolución con un enfoque tradicional
El equipo tradicional estará esperando que se le entregue documentación detallada que sirva como soporte durante la evolución.
- Desarrollo con enfoque tradicional, evolución con un enfoque ágil
Es más difícil. Podrían no existir pruebas automatizadas ya desarrolladas. El código puede no haber sido refactorizado y simplificado.

9.4. Sistemas heredados

En una organización, los sistemas se remplazan a medida que el negocio cambia. Sin embargo muchos viejos sistemas continúan siendo utilizados, e incluso, tienen un rol crítico en el negocio. Estos son llamados sistemas heredados.

- No son solo sistemas de software, abarcan hardware, software, librerías, software de soporte y procesos de negocio.
- Han tenido mantenimiento por un largo tiempo, por lo que su estructura puede estar degradado.
- Pueden depender de hardware antiguo.
- Es probable que no soporten nuevos procesos de negocio.

El mantenimiento de este tipo de sistema tiene dificultades y es costoso:

- Falta de habilidades o conocimiento de viejas tecnologías (recursos externos)
- Dificultades para entender el código debido a que fue modificado por muchas personas con diferentes estilos.
- Sistema degradado por muchos años de mantenimiento. Falta documentación o está desactualizada.

- Vulnerabilidades de seguridad
- Problemas para integrarse con sistemas construidos con tecnologías nuevas
- Ausencia de soporte oficial
- Hardware obsoleto y costoso de mantener
- Problemas a nivel de datos: duplicación y baja calidad.

Tomar la decisión de reemplazar un sistema heredado puede ser costoso y riesgoso:

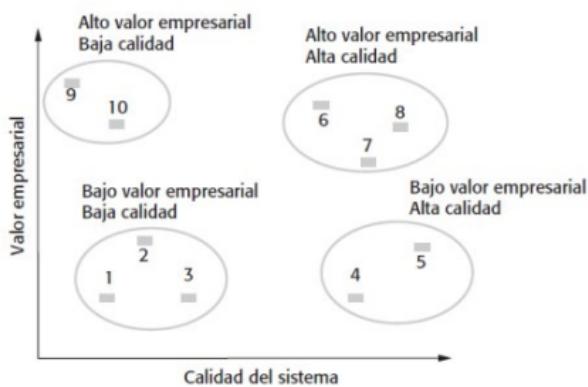
- No existe especificación completa del sistema.
- Los procesos de negocio seguramente tengan que ser modificados.
- Reglas de negocio pueden estar hardcodeadas sin documentación.
- El nuevo software trae riesgos consigo (tiempo y costo).

9.4.1. Gestión de sistemas heredados

Las organizaciones que se basan en sistemas heredados deben elegir la estrategia para la evolución del software.

- Desechar el sistema por completo y modificar los procesos de negocio de forma tal que no sea necesario.
- Continuar el mantenimiento del sistema
- Transformar el sistema por medio de la reingeniería y mejorar la mantenibilidad.
- Reemplazar el sistema con un sistema nuevo.

La estrategia elegida dependerá de la calidad del sistema y del valor del sistema para el negocio.



- 1,2,3: Estos sistemas deben ser desechados
- 4,5: Remplazar con componentes de terceros, desechar completamente o mantener.
- 6,7,8: Continuar con el sistema en operación y realizar un mantenimiento normal del sistema.
- 9,10: Estos hacen una importante contribución al negocio pero son caros de mantener. Debe ser reestructurado o remplazado por otro sistema comercial que esté disponible.

9.5. Evaluación del valor para el negocio

La evaluación debe de tomar en cuenta diferentes puntos de vista:

- Usuarios finales del sistema.
- Clientes de negocios.
- Gerentes de línea.
- Administradores.

Se puede entrevistar a diferentes interesados y cotejar resultados.

Se deberá tener en cuenta los distintos aspectos básicos en la valoración para un negocio.

- **El uso del sistema**

Si los sistemas son utilizados ocasionalmente o por un pequeño número de personas, el sistema puede tener un bajo valor para el negocio.

- **Los procesos de negocio que son soportados**

Un sistema puede tener un bajo valor para el negocio si fuerza a que se utilicen procesos de negocios ineficientes.

- **Confianza del sistema**

Si un sistema no es confiable y los problemas afectan directamente a los clientes del negocio, el sistema tiene un bajo valor para el negocio.

- **Las salidas del sistema**

Si el negocio dependerá de las salidas del sistema, entonces el sistema tiene un alto valor para el negocio.

9.5.1. Evaluación de la calidad del sistema

- Evaluación del proceso de negocio

¿En qué medida el proceso de negocio da soporte a los objetivos actuales del negocio?

- Evaluación del entorno

¿Cuán efectivo es el entorno del sistema y cuán costoso es su mantenimiento?

- Evaluación de la aplicación ¿Cuál es la calidad de la aplicación del sistema de software?

- Cantidad de solicitudes de cambio.

- Número de interfaces de usuario.

- Volumen de datos utilizados por el sistema

9.6. Mantenimiento de software

Es la modificación de un programa luego de que está siendo utilizado. El término se utiliza sobre todo para realizar cambios en el software personalizado. Normalmente el mantenimiento no genera grandes cambios en la arquitectura del sistema. Los cambios implican modificar componentes existentes y agregar componentes al sistema.

9.6.1. Tipos de mantenimiento

- **Mantenimiento para reparar defectos o vulnerabilidades del software (24 %)**

Los errores pueden haber sido introducidos en diferentes etapas, lo que influye en el costo de corrección.

- **Mantenimiento para adaptar el software a un entorno operativo diferente (19 %)**

Cambios en el sistema para que este opere en entornos diferentes a los de la implementación inicial.

- Mantenimiento para agregar o modificar funcionalidades al sistema (58 %)
Modificar el sistema para satisfacer nuevos requisitos.

Podemos categorizarlos de otra forma (Pfleeger, 2010)

- **Correctivo (21 %)**
Control del funcionamiento diario del sistema a través de la reparación de fallas.
- **Adaptativo (25 %)**
El sistema se modifica para adaptarse a cambios en el entorno.
- **Perfectivo (50 %)**
Mejorar funcionalidades existentes.
- **Preventivo (4 %)**
Prevenir que el desempeño del software se degrade.

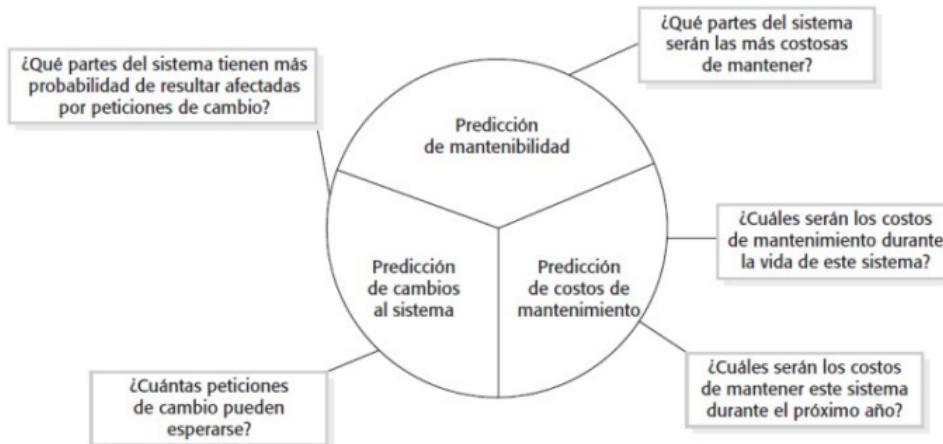
9.6.2. Costos de mantenimiento

Los costos de mantenimiento son generalmente mayores que los costos de desarrollo. Agregar funcionalidades nuevas es más costoso que haberlo hecho durante el desarrollo inicial.

Nos motiva más tener un problema nuevo, que mantener algo ya existente. Además, la tarea de mantenimiento del software no es popular. El mantenimiento corrompe la estructura del software y así se hace aún más difícil de mantener.

9.6.3. Predicción de mantenimiento

Se ocupa de evaluar qué partes del sistema pueden causar problemas y cuál serán los costos de mantenimiento. Los costos de mantenimiento dependen del **número de cambios**, y el **costo del cambio** depende de la **capacidad de mantenimiento**



Predicción del cambio Es la predicción de la cantidad de cambios requeridos y la comprensión de las relaciones entre el sistema y su entorno. Los sistemas fuertemente acoplados requieren cambios cada vez que cambia su entorno.

Los factores que influencian esta relación son:

- Cantidad y complejidad de las interfaces del sistema.
- Cantidad de requisitos del sistema que son volátiles de forma inherente.
- Los procesos de negocios donde el sistema es utilizado.

9.6.4. Métricas de complejidad

Realizar las predicciones de mantenimiento en base a la evaluación de la complejidad de los componentes del sistema. Estudios muestran que el mayor esfuerzo de mantenimiento se gasta en un número relativamente pequeño de componentes.

La complejidad depende de:

- La complejidad de la estructura de control.
- La complejidad de la estructura de datos
- Objetos, métodos y el tamaño de los módulos.

9.6.5. Métricas de proceso

Para evaluar la mantenibilidad también podemos utilizar métricas de proceso. Por ejemplo:

- Cantidad de solicitudes de mantenimiento correctivo.
- Promedio de tiempo para el análisis del impacto.
- Promedio de tiempo para implementar la solicitud de cambio.
- Cantidad de solicitudes de cambios excepcionales.

Si alguna de estas métricas incrementa, puede indicar que está disminuyendo la mantenibilidad.

9.7. Reingeniería de software

Es la reestructuración o la reescritura de parte o todo un sistema heredado sin cambiar su funcionalidad. Es aplicable cuando los sistemas necesitan un mantenimiento frecuente. Involucra agregar un esfuerzo para hacer que sea más fácil mantener el sistema.

9.7.1. Ventajas de la reingeniería

■ Reducción de riesgos

Hay un alto riesgo en el desarrollo de un nuevo software. Pueden haber problemas de desarrollo, problemas de equipo y problemas de especificación.

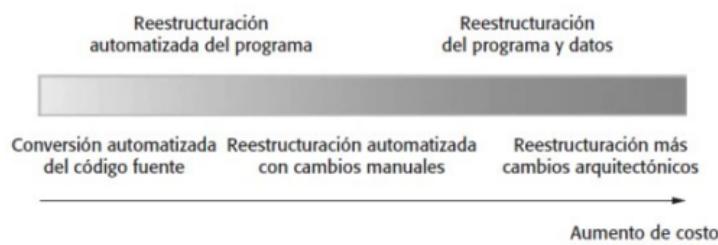
■ Reducción del costo

Generalmente los costos de la reingeniería son mucho menores que los costos de desarrollar un nuevo software.

9.7.2. Proceso de reingeniería



9.7.3. Enfoques de la reingeniería



Se tienen los siguientes factores de costo:

- La calidad del software a ser rediseñado.
- Las herramientas de soporte disponibles para la reingeniería.
- El grado de conversión de datos que se requiere.
- La disponibilidad de personal experto para la reingeniería. Esto puede ser un problema cuando los sistemas están basados en tecnología vieja u obsoleta.

9.8. Refactorización

La refactorización es el proceso de hacer mejoras a un programa para reducir la degradación del sistema al realizar cambios. Se puede pensar en la refactorización como un *mantenimiento preventivo* que reduce los problemas en cambios futuros.

- La refactorización trata de modificar un programa para mejorar su estructura, reducir su complejidad o hacerlo más fácil de entender.
- Cuando se refactoriza un programa no se deben agregar funcionalidades.
- Es inherente a las metodologías ágiles.

9.8.1. Refactorización y reingeniería

- La reingeniería toma lugar luego de que el sistema ha sido mantenido por algún tiempo y los costos se incrementan. Utiliza herramientas automatizadas para procesar y rediseñar sistemas heredados para crear un nuevo sistema más mantenable.
- La refactorización es un proceso continuo de mejora en todo el proceso de desarrollo y su evolución. Se tiene la intención de evitar que se incremente la degradación de la estructura y el código para que así no se dificulte el mantenimiento del sistema.

9.9. Bad Smells

Un *Bad smell* es una señal de que el código puede ser mejorado. Podemos ver los siguientes bad smells en el código:

- **Código duplicado**
El mismo código o muy similar es incluido en diferentes lugares del sistema. Estos pueden ser removidos e implementados como una única función o método que puede invocarse cuando se requiere.
- **Métodos largos**
Si el método es demasiado largo, debe ser rediseñado en métodos más cortos.
- **Enunciados Switch (case)**
Generalmente implica duplicación. El cambio depende del tipo de un valor. Estos enunciados pueden estar dispersos en el programa. Normalmente en los lenguajes de programación orientados a objetos se puede utilizar el polimorfismo para lograr el mismo comportamiento.

- **Aglomeración de datos**

La aglomeración de datos ocurre cuando el mismo grupo de ítems de datos aparecen en varios lugares del programa. Generalmente puede ser reemplazado por un objeto que encapsule todos estos datos.

- **Generalidad especulativa**

Esto ocurre cuando los programadores incluyen generalidad al programa en caso de que sea necesaria en el futuro. Por lo general puede ser removido de forma simple.

10. Gestión de Proyectos

10.1. Introducción

En primer lugar, un proyecto es un esfuerzo **temporal** que se lleva a cabo para crear un producto, servicio o resultado **único**. Analicemos esta definición:

- Decimos que es temporal porque tiene un inicio y un fin. Un proyecto puede terminar por muchas razones, por ejemplo:
 - Se cumplen los objetivos del proyecto (el caso feliz).
 - La necesidad que dio origen al proyecto ya no existe.
 - Decisión del cliente o patrocinador.
- Decimos que es único ya que a pesar de existir actividades que se repiten en algunas actividades del proyecto, esta repetición no altera las características únicas del proyecto.

La **gestión de proyectos** es la aplicación de conocimientos, habilidades, herramientas y técnicas a las actividades del proyecto para cumplir con los requisitos de aquél.

La gestión de proyectos de software es muy diferente a la gestión de proyectos tradicionales ya que:

- El producto es intangible
- Los grandes proyectos de SW, varían mucho de uno a otro
- Los procesos son distintos para cada organización

Los principales objetivos son:

- Entregar un sw que cumpla con las expectativas del cliente.
- Entregar un sw que cumpla con la calidad esperada.
- Entregar un producto de sw con el alcance acordado.
- Entregar el sw en el plazo acordado.
- Mantener los costos dentro del presupuesto acordado.
- Mantener un equipo funcionando adecuadamente.

10.2. Director de proyecto

De acuerdo a PMBOK (5ta edición): “El director del proyecto es la persona asignada por la organización ejecutora para liderar al equipo responsable de alcanzar los objetivos del proyecto.”

Para que el director de proyectos pueda liderar al equipo para lograr los objetivos, es necesario que:

- Cuente con habilidades en el área de desempeño del proyecto. Esto no quiere decir que sea un experto técnico pero debe tener noción del área.
- Cuente con las competencias de conocimiento, desempeño y personales. Conocimiento refiere a lo que el director de proyecto conozca sobre la dirección de proyectos, la formación que tenga. Respecto a desempeño, apunta a lo que el director de proyecto pueda lograr al aplicar los conocimientos en dirección de proyectos aplicándolos de forma adecuada. Y personal, refiere a cómo se comporta el director de proyectos. Esto abarca personalidad, actitudes y habilidades interpersonales.
- Es vital para el éxito del proyecto que el director de proyectos tenga formación y experiencia en: liderazgo, trabajo en equipo, negociación, manejo de conflictos, motivación entre otros.

10.3. Ciclo de vida de un proyecto

A continuación describimos 3 ciclos de vida de un proyecto

- Predictivo

También conocido como orientado a planes, refiere a los ciclos donde el alcance, tiempo y costo se definen lo más temprano posible en el proyecto. En general, este tipo de ciclo se utiliza cuando el producto a desarrollar se conoce muy bien. El objetivo primordial del equipo al inicio del proyecto será poder definir el alcance del producto y de proyecto, armar un plan para cumplir con este alcance y proceder a ejecutar. En este tipo de proyectos, los cambios de alcance, son revisados minuciosamente y deben pasar por un proceso formal antes de ser aceptados.

- Iterativo e incremental

El desarrollo de producto en estos ciclos se hace de forma iterativa y con incrementos graduales. En cada iteración, se construyen los entregables necesarios para cumplir los criterios de salida de la iteración o fase según corresponda. Esto ayuda a poder entregar valor en forma temprana al cliente y obtener feedback del mismo e ir adaptando al producto. Se opta por estos ciclos cuando el alcance cambia, es necesario reducir la complejidad total y cuando posibles entregas parciales generan valor.

- Adaptativos

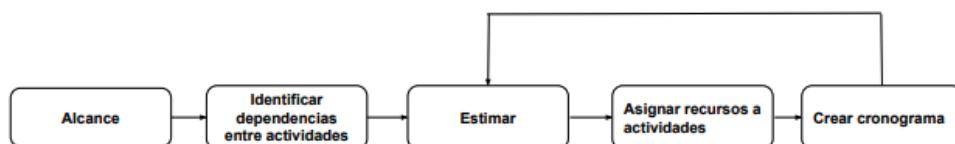
También conocidos como métodos ágiles, es un tipo de ciclo iterativo e incremental. En general, se opta por estos modelos cuando existen altos niveles de cambio y es posible la interacción durante todo el proyecto con los interesados.

10.4. Planificación

Planificamos en tres etapas del proyecto:

- En la propuesta o contrato.
- Al inicio del proyecto.
- En forma periódica durante el proyecto.

El proceso de planificación tiene las siguientes etapas:



10.5. Planificación - Alcance

- **Alcance del producto:** Características y funciones que describen un producto, servicio o resultado.
- **Alcance del proyecto:** Trabajo a realizar para entregar el producto, servicio o resultado con las funciones y características especificadas

Pasos a seguir:

1. Recopilar los requisitos.
2. Definir el alcance, qué se incluye y qué no.
3. Crear WBS (work break down structure) o EDT (estructura de desglose de trabajo).

10.5.1. EDT/WBS

Según PMBOK (5ta edición): “La EDT/WBS es una descomposición jerárquica del alcance total del trabajo a realizar por el equipo del proyecto para cumplir con los objetivos del proyecto y crear los entregables requeridos”

La creación de un EDT consiste en subdividir los entregables de proyecto en partes más pequeñas donde cada componente tiene su identificación. Esta tarea en general es realizada por los miembros claves del equipo.

El nivel más bajo de la EDT, se denomina paquete de trabajo. Un paquete de trabajo es el resultado de la ejecución de varias actividades, pero no es una actividad. Tener en cuenta que la EDT **NO incluye actividades**.

El nivel de detalle para los paquetes de trabajo varía en función del tamaño y la complejidad del proyecto. Cuanto mayor nivel de detalle incorporamos a la EDT, facilita la planificación, estimación, asignación de tareas, etc.

También se debe tener en cuenta que el esfuerzo de seguir refinando la EDT puede ser bastante grande, es necesario llegar a un equilibrio entre el detalle de la EDT y el esfuerzo que se dedica.

La EDT/WBS representa todo el trabajo necesario para realizar el producto y el proyecto, e incluye el trabajo de dirección del proyecto. Debe cumplir la *regla del 100 %* donde el total del trabajo de los niveles inferiores corresponde al acumulado de los niveles superiores y solo ese trabajo.

La EDT NO está basada en dependencias de secuencia o tiempo entre sus componentes.

10.6. Planificación - Actividades

Luego de realizado el EDT, debemos identificar las actividades

- Los paquetes de trabajo de la EDT se descomponen en actividades.
- Estas actividades van a poder ser estimadas, van a ser parte del cronograma y servirán para el seguimiento.

Luego es necesario identificar precedencia entre las actividades. Para esto se debe:

- Identificar las relaciones lógicas en las actividades
- Todas las actividades, salvo la primera y la última, se conectan a por lo menos una predecesora y una sucesora.

10.7. Planificación - Estimaciones

Estimar es predecir cuánto va a durar un proyecto o cuánto va a costar. Es algo difícil de hacer. Las estimaciones pueden ser inexactas debido a:

- Definición de requisitos poco precisa.
- Entornos desconocidos / tecnología de punta.
- Experiencia del equipo.

10.7.1. Principios de estimación

- Una estimación es una proyección de la experiencia del pasado hacia el futuro, ajustando según las diferencias entre el pasado y el futuro. para esto:
 - Es necesario contar con experiencia pasada
 - Es necesario saber algo del futuro
 - Es necesario saber cómo ajustar
- Todas las estimaciones están basadas en supuestos y restricciones.
- Los proyectos deben re-estimarse en forma periódica y aperiódica, cuando corresponda. Por ej: un cambio mayor de requerimientos, pérdida de personal clave, reducción del presupuesto, etc.

10.7.2. Factores que influyen en las estimaciones

- Tamaño del software.
- Tipo de SW.
- Factores del personal.
- Lenguaje de programación.

10.7.3. Estimación del tamaño del producto

La estimación de factores como: esfuerzo, costos, cronograma, recursos y calidad se basan en la estimación de atributos del producto, restricciones del proyecto, experiencias pasadas y factores de ajuste, siendo el tamaño, el atributo del producto más utilizado. Dependiendo de la naturaleza del producto, factores adicionales al tamaño, tales como la complejidad del producto, se incluyen como factores de ajuste en la mayoría de los modelos de estimación.

Las razones para utilizar el tamaño como factor principal en los métodos de estimación basados en factores del producto son:

- La relación entre el tamaño y factores del producto tales como esfuerzo y cronograma tiene un mayor impacto respecto a otros atributos del producto.
- La medición del tamaño puede realizarse de forma más objetiva que otros atributos del producto.
- Algunos tipos de medidas de tamaño sirven para estimar de forma más precisa a partir de los requerimientos.
- Datos como el tamaño, esfuerzo y cronograma de un proyecto terminado pueden ser guardados, como experiencias pasadas, para ser utilizados en la estimación de proyectos futuros.

LOCs (líneas de código)

Históricamente se han usado las líneas de código para medir el tamaño de un producto. Sin embargo existen algunos problemas relacionados a su uso:

- Es difícil estimar líneas de código al inicio de un proyecto.
- Es difícil relacionar cambios en los requerimientos con cambios en líneas de código.
- Calcular la productividad como líneas de código generadas puede producir muchas líneas de código de baja calidad.
- Métodos de desarrollo modernos, como la reutilización de componentes, hacen que la relación entre las líneas de código y atributos del proyecto ya no sean tan útiles y precisas.

PF (Puntos de función)

Los puntos de función se calculan contando el número de diferentes tipos de entradas, salidas, archivos internos, consultas a base de datos e interfaces en un sistema a estimar. El cálculo se basa en reglas objetivas de conteo y cada entrada única, salida, archivo interno, consulta a base de dato e interfaz se ponderan como simple, promedio o complejo.

Tiene como ventajas que:

- Se puede medir sin que exista el código, sólo a partir de requisitos o diseño
- Es independiente del lenguaje.

Tiene como desventajas que:

- Aplicación restringida a sistemas con uso intensivo de datos y poco peso algorítmico.
- Medir PF requiere esfuerzo, es complejo

10.7.4. Técnicas de estimación

Hay dos tipos de técnicas de estimación

- **Basadas en la Experiencia**

La estimación se realiza acorde a la experiencia y conocimiento del equipo en proyectos pasados.

- Analogía
- Juicio de expertos
- Delphi

- **Algoritmos**

Se aplican modelos matemáticos considerando atributos del producto como tamaño, experiencia, etc

- Cocomo

10.7.5. Técnicas - Analogía

Encontrar proyectos análogos para utilizarlos como referencia. Cuanto más análogo el proyecto hay mayor certeza. Se puede armar una tabla con proyectos pasados para identificar similitudes. Para poder utilizar esta técnica es conveniente tener datos de los proyectos anteriores, como por ejemplo:

- Tipo de producto
- Alcance de las actividades incluidas
- Tamaño del producto y la medida de tamaño utilizada
- Factores de ajuste utilizados (por ejemplo, complejidad del producto, nivel de habilidad de los desarrolladores)
- Modelo de desarrollo utilizado
- Herramientas de desarrollo utilizadas
- Productos entregables producidos
- Duración estimada y real del proyecto
- Esfuerzo estimado y real
- Costo estimado y real
- Densidades de defectos previas y posteriores a la liberación
- Problemas encontrados
- Lecciones aprendidas

10.7.6. Técnicas - Juicio de expertos

Implica solicitar a uno o varios expertos sus estimaciones respecto a atributos del proyecto, tales como esfuerzo, tiempo, habilidades requeridas en el equipo y factores de riesgo. Las estimaciones obtenidas por cada experto pueden incluir factores subjetivos, relacionados al conocimiento del equipo que trabajará en el desarrollo del proyecto, políticas internas o conflictos de la organización, entre otras. Los expertos pueden considerar que los requisitos son demasiado vagos o incompletos para poder realizar la estimación. Por otro lado, diferentes tipos de expertos pueden estimar distintos atributos del proyecto.

Esta técnica tiene como ventajas que

- Diferentes tipos de expertos pueden proporcionar estimaciones para diferentes tipos de componentes del producto e incluir factores subjetivos y políticos que típicamente no son guardados con los datos de los proyectos pasados.

Tiene como desventajas que

- Los expertos pueden ser excesivamente optimistas al estimar el tiempo y los recursos necesarios para ellos y no considerando que el equipo puede no tener otras habilidades.
- Su recuerdo de experiencias pasadas puede ser incorrecto o incompleto.

10.7.7. Técnicas - Delphi

El objetivo de esta técnica es obtener estimaciones de diferentes expertos donde cada experto recibe la misma información del producto y puede utilizar la técnica de estimación que deseé.

Se realiza de la siguiente forma:

- A cada experto se le solicita un estimado de los atributos del proyecto y una justificación para su estimación.
- Luego de que se realiza la estimación, un coordinador proporciona todas las estimaciones recolectadas de todos los expertos (sin incluir nombres) para que cada uno realice otra estimación. El intervalo entre cada estimación debe ser de uno o dos días para que cada experto tenga tiempo de revisar y reflexionar sobre la documentación provista de la ronda anterior.
- Generalmente las estimaciones convergen después de 3 o 4 rondas. Si las estimaciones convergen, entonces se realiza una reunión para confirmar las estimaciones y registrar posibles problemas o inquietudes ocurridas durante las mismas.
- Si las estimaciones no convergen, se realiza una reunión para explicar las diferencias y justificar las estimaciones. Si no se llega a un acuerdo en la reunión se pueden utilizar rangos de estimación calculados con funciones de probabilidad.

Variante: Wideband Delphi

El proceso Wideband Delphi es un proceso alternativo al Delphi, donde antes de que se lleven a cabo las rondas de estimaciones se realiza una reunión para discutir el proyecto, el cual permite tiempo para reflexionar y enviar estimaciones de forma anónima. Se mantienen reuniones luego de cada ronda de estimación para discutir suposiciones y justificaciones a las estimaciones. Una desventaja es que puede existir algún tipo de influencia entre los expertos.

Variante: Planning Poker

Otra variante es completar todo el proceso en una única reunión. En esta, cada ronda de estimación se realiza de forma secreta, discutiendo la estimación entre rondas. Generalmente la estimación y justificación de los expertos se muestran de forma anónima a todo el grupo para fomentar la discusión entre rondas de estimación.

10.7.8. Técnicas - Algoritmos

Se basan en fórmulas matemáticas. Por ejemplo:

$$E = S \times \text{Size}^B \times M$$

Donde:

- A: es un factor constante organizacional
- Size: medida del producto. Por ejemplo puntos de función
- B: es la complejidad, en general entre 1 y 1.5
- M: es un factor que considera proceso, atributos, equipo, etc.

Su uso puede sonar muy atractivo, pero tiene algunos problemas:

- Muy difícil de aplicar en etapas tempranas
- La estimación de B y M son subjetivas
- En general son complejos y difíciles de usar
- Es necesario calibrar a la propia historia organizacional. No todas las organizaciones recolectan suficientes datos para usarlos para calibrar.

10.7.9. Técnicas - COCOMO II

Se recolectan datos de varios proyectos de diferentes tamaños. Se analizan estos datos y se definen fórmulas. Estas fórmulas relacionan: tamaño de sistema, del producto y del equipo.

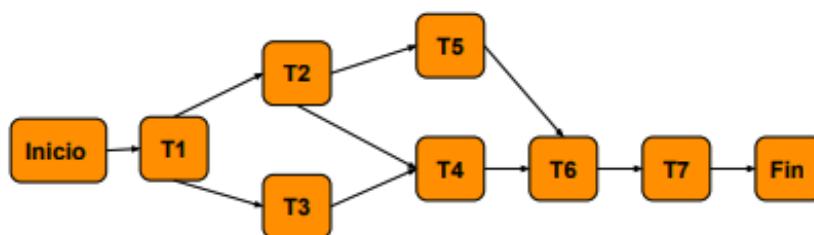
Tiene 4 sub-modelos:

- **Composición de aplicaciones:** fue introducido para apoyar el esfuerzo de estimación requerido para proyectos donde se utilizan componentes pre-existentes. Se basa en la estimación de puntos de aplicación (o puntos de objeto) los cuales tienen un cierto peso, divididos por un estimado de puntos de productividad.
- **Diseño temprano:** se utiliza este modelo en etapas tempranas del proyecto. El objetivo es hacer una rápida y aproximada estimación de costos. Son muy útiles para la exploración de opciones.
- **Reuso:** se usa para estimar el esfuerzo requerido para integrar código reutilizable.
- **Modelo post arquitectura:** es el modelo más detallado. Se usa cuando se tiene un diseño de arquitectura inicial del sistema. En esta etapa se debe poder hacer una estimación más precisa del tamaño del proyecto.

10.8. Cronograma

10.8.1. Grafo de precedencias - Camino crítico

Creamos un grafo que indique el orden en el que deben de realizarse las actividades.



Este grafo tiene varios caminos entre inicio y fin. Uno de estos caminos es el “camino crítico”. Este es el camino que si se retrasa, retrasa todo el proyecto. Es importante aclarar que:

- Es el camino más largo en duración
- Puede haber más de uno
- Puede cambiar durante el ciclo de vida
- No tiene relación con la importancia técnica de las actividades

Algunas definiciones:

- **Comienzo temprano (ES):** lo antes posible que puede comenzar una actividad respetando las precedencias y duraciones.
- **Fin Temprano (EF):** la fecha de fin si la actividad comienza lo antes posible y dura lo previsto.
- **Comienzo tardío (LS):** lo más tarde que puede comenzar la actividad sin afectar la duración del proyecto.
- **Fin tardío (FT):** lo más tarde que puede terminar la actividad sin afectar la duración del proyecto.
- **Holgura total:** cuánto se puede retrasar el comienzo de un actividad sin afectar la fecha de fin del proyecto.
- **Holgura libre:** cuánto se puede retrasar un actividad dentro de un camino sin retrasar la fecha de inicio temprana de cualquier actividad subsiguiente inmediata.

Podemos calcular el camino crítico como en IIO aplicando el algoritmo de Bellman.

10.9. Cronograma

Luego de definir la precedencia entre las tareas, podemos armar un cronograma de como se trabajará.

- Se ajustan fechas de inicio y/o fin cuando hay restricciones de recursos
- Se usa luego de determinar el camino crítico y cuando hay recursos:
 - Compartidos o críticos durante ciertos momentos
 - Disponibles en cantidades limitadas
 - Que se desean utilizar con un nivel constante de ocupación en un período de tiempo
- Es necesario cuando hay sobreasignación de recursos
- Puede cambiar el camino crítico, usualmente crece en tiempo.

Por ejemplo si tenemos que nivelar los recursos, puede suceder que el proyecto aumente la duración y haya un cambio en el camino crítico.

10.9.1. Técnicas para comprimir

Hay dos técnicas si necesitamos acortar el cronograma: Crashing (compresión) y Fasttracking (ejecución rápida).

- **Crashing**
Tiene como objetivo acortar el cronograma con el menor incremento de costo posible. Por ejemplo: horas extra, más recursos, pago adicional por acelerar la entrega, etc.
 - Sólo funciona para actividades del camino crítico.
 - No siempre es viable, hay que tener cuidado con el incremento de costos y riesgos.
- **Fast tracking**
Actividades o fases que en general se hace secuenciales, se hacen en paralelo. Puede generar retrabajo y aumento en riesgos. Solo funciona si la paralelización de tareas es viable.

10.10. Riesgos

Un riesgo es un evento o condición incierta que si se produce, tiene un efecto positivo o negativo sobre al menos un objetivo del proyecto, tales como el alcance, el cronograma, el costo y la calidad.

Un riesgo puede tener una o más causas y, de materializarse, uno o más impactos. Una causa puede ser un requisito especificado o potencial, un supuesto, una restricción o una condición que crea la posibilidad de consecuencias tanto negativas como positivas.

- Las organizaciones tienen distintos grados de tolerancia al riesgo
- Según la actitud hacia el riesgo, es la forma en que se responde. Algunos riesgos se aceptan o rechazan según el grado de recompensa que puede obtenerse.
- Riesgo positivo: oportunidad
- Riesgo negativo: amenaza
- Riesgo negativo materializado: problema

10.10.1. Categorización

- **De proyecto:** afectan cronograma, recursos. Ejemplo: perder un miembro experimentado del equipo.
- **De producto:** afectan calidad o performance. Ejemplo: falla en un componente clave del producto.
- **De negocio:** afectan el negocio de lo que se está construyendo. Ejemplo: un competidor introduce un nuevo producto.

10.10.2. Identificación de riesgos

De acuerdo al PMBOK (5ta edición): “Identificar los Riesgos es el proceso de determinar los riesgos que pueden afectar al proyecto y documentar sus características”. Es un proceso iterativo dado que los riesgos pueden evolucionar a lo largo del proyecto o también pueden surgir nuevos riesgos. El formato de las declaraciones de riesgos debe ser consistente para asegurar que cada riesgo se comprenda claramente y sin ambigüedades, a fin de poder llevar a cabo un análisis y un desarrollo de respuestas eficaces.

Herramientas y técnicas

■ **Revisiones a la Documentación**

Se pueden realizar revisiones estructuradas a la documentación del proyecto, incluidos los planes, los supuestos, los archivos de proyectos anteriores, los acuerdos, entre otros. La calidad y consistencia de la documentación pueden ser indicadores de riesgo en el proyecto.

■ **Técnicas de Recopilación de Información**

Ejemplos de esto son tormentas de ideas, técnicas Delphi, entrevistas, análisis de causa raíz.

■ **Análisis con lista de verificación**

Las listas de verificación están basadas en información histórica y conocimiento acumulado a partir de proyectos anteriores similares y de otras fuentes de información. Estas listas no son exhaustivas, por lo que no deberían ser utilizadas para evitar el esfuerzo de identificar los riesgos. Es conveniente revisar estas listas durante el cierre de un proyecto y así incorporar nuevas lecciones aprendidas a fin de mejorarla, para poder usarla en proyectos futuros.

■ **Análisis de Supuestos**

Cada proyecto y su plan se crean y desarrollan sobre la base de un conjunto de hipótesis, escenarios o supuestos. El análisis de supuestos explora la validez de los supuestos según se

aplican al proyecto. Identifica los riesgos del proyecto relacionados con el carácter inexacto, inestable, inconsistente o incompleto de los supuestos.

■ Técnicas de diagramación

Utilizamos técnicas de diagramación de riesgos, las cuales incluyen:

- Diagramas de causa y efecto: útiles para identificar las causas de los riesgos.
- Diagrama de flujo de procesos o sistemas: muestran cómo se relacionan entre sí los diferentes elementos de un sistema, y el mecanismo de causalidad.
- Diagramas de influencias: son representaciones gráficas de situaciones que muestran las influencias causales, la cronología de eventos y otras relaciones entre las variables y los resultados.

■ Análisis FODA

Esta técnica examina el proyecto desde cada uno de los aspectos FODA (fortalezas, oportunidades, debilidades y amenazas) para aumentar el espectro de riesgos identificados, incluidos los riesgos generados internamente. La técnica comienza con la identificación de las fortalezas y debilidades de la organización, centrándose ya sea en el proyecto, en la organización o en el negocio en general.

■ Juicio de expertos

Los expertos con la experiencia adecuada, adquirida en proyectos o áreas de negocio similares, pueden identificar los riesgos directamente. En este proceso se deben tener en cuenta los sesgos de los expertos.

10.10.3. Análisis cualitativo de riesgos

En el análisis se evalúa la prioridad de los riesgos identificados a través de la probabilidad relativa de ocurrencia, del impacto correspondiente sobre los objetivos del proyecto si los riesgos llegaran a presentarse, así como de otros factores, tales como el plazo de respuesta y la tolerancia al riesgo por parte de la organización, asociados con las restricciones del proyecto en términos de costo, cronograma, alcance y calidad. El proceso de análisis se lleva a cabo de manera regular a lo largo del ciclo de vida del proyecto.

Herramientas y técnicas

■ Evaluación de Probabilidad e Impacto de los Riesgos

La evaluación de la probabilidad de los riesgos estudia la probabilidad de ocurrencia de cada riesgo específico. La evaluación del impacto de los riesgos estudia el efecto potencial de los mismos sobre un objetivo del proyecto, tal como el cronograma, el costo, la calidad o el desempeño, incluidos tanto los efectos negativos en el caso de las amenazas, como los positivos, en el caso de las oportunidades. Para cada uno de los riesgos identificados, se evalúan la probabilidad y el impacto.

■ Matriz de Probabilidad e Impacto

Por lo general, la evaluación de la importancia de cada riesgo y de su prioridad de atención se efectúa utilizando una tabla de búsqueda o una matriz de probabilidad e impacto. Dicha matriz especifica las combinaciones de probabilidad e impacto que llevan a calificar los riesgos con una prioridad baja, moderada o alta. Dependiendo de las preferencias de la organización, se pueden utilizar términos descriptivos o valores numéricos. Cada riesgo se califica de acuerdo con su probabilidad de ocurrencia y con el impacto sobre un objetivo, en caso de que se materialice. La organización debe determinar qué combinaciones de probabilidad e impacto dan lugar a una clasificación de riesgo alto, riesgo moderado y riesgo bajo.

10.10.4. Planificar la respuesta a los riesgos

Existen varias estrategias de respuesta a los riesgos. Para cada riesgo, se debe seleccionar la estrategia o la combinación de estrategias con mayor probabilidad de eficacia.

Estrategias para riesgos negativos o amenazas

■ Evitar

Es una estrategia de respuesta a los riesgos según la cual el equipo del proyecto actúa para eliminar la amenaza o para proteger al proyecto de su impacto. Ejemplos de lo anterior son la ampliación del cronograma, el cambio de estrategia o la reducción del alcance. La estrategia de evasión más drástica consiste en anular por completo el proyecto. Algunos riesgos que surgen en etapas tempranas del proyecto se pueden evitar aclarando los requisitos, obteniendo información, mejorando la comunicación o adquiriendo experiencia.

■ Transferir

Es una estrategia de respuesta a los riesgos según la cual el equipo del proyecto traslada el impacto de una amenaza a un tercero, junto con la responsabilidad de la respuesta. La transferencia de un riesgo simplemente confiere a una tercera parte la responsabilidad de su gestión; no lo elimina. La transferencia no implica que se deje de ser el propietario del riesgo por el hecho de transferirlo a un proyecto posterior o a otra persona sin su conocimiento o consentimiento.

■ Mitigar

Es una estrategia de respuesta a los riesgos según la cual el equipo del proyecto actúa para reducir la probabilidad de ocurrencia o impacto de un riesgo. Implica reducir a un umbral aceptable la probabilidad y/o el impacto de un riesgo adverso. Adoptar acciones tempranas para reducir la probabilidad de ocurrencia de un riesgo y/o su impacto sobre el proyecto, a menudo es más eficaz que tratar de reparar el daño después de ocurrido el riesgo. Ejemplos de acciones de mitigación son adoptar procesos menos complejos, realizar más pruebas o seleccionar un proveedor más estable. La mitigación puede requerir el desarrollo de un prototipo para reducir el riesgo de pasar de un modelo a pequeña escala de un proceso o producto a uno de tamaño real. Cuando no es posible reducir la probabilidad, una respuesta de mitigación puede abordar el impacto del riesgo centrándose en los vínculos que determinan su severidad. Por ejemplo, incorporar redundancias en el diseño de un sistema puede permitir reducir el impacto causado por una falla del componente original.

■ Aceptar

Es una estrategia de respuesta a los riesgos según la cual el equipo del proyecto decide reconocer el riesgo y no tomar ninguna medida a menos que el riesgo se materialice. Esta estrategia se adopta cuando no es posible ni rentable abordar un riesgo específico de otra manera. Esta estrategia indica que el equipo del proyecto ha decidido no cambiar el plan para la dirección del proyecto para hacer frente a un riesgo, o no ha podido identificar ninguna otra estrategia de respuesta adecuada. Esta estrategia puede ser pasiva o activa. La aceptación pasiva no requiere ninguna acción, excepto documentar la estrategia dejando que el equipo del proyecto aborde los riesgos conforme se presentan, y revisar periódicamente la amenaza para asegurarse de que no cambie de manera significativa. La estrategia de aceptación activa más común consiste en establecer una reserva para contingencias, que incluya la cantidad de tiempo, dinero o recursos necesarios para manejar los riesgos.

Estrategias para riesgos positivos u oportunidades

■ Explorar

Esta estrategia se puede seleccionar para los riesgos con impactos positivos, cuando la organización desea asegurarse de que la oportunidad se haga realidad. Esta estrategia busca eliminar la incertidumbre asociada con un riesgo a la alza en particular, asegurando que la oportunidad definitivamente se concrete. Algunos ejemplos de respuestas de explotación

directa incluyen la asignación al proyecto de los recursos más talentosos de una organización para reducir el tiempo hasta la conclusión, o el uso de nuevas tecnologías o mejoras tecnológicas para reducir el costo y la duración requeridos para alcanzar los objetivos del proyecto.

■ **Mejorar**

Se utiliza para aumentar la probabilidad y/o los impactos positivos de una oportunidad. La identificación y maximización de las fuerzas impulsoras clave de estos riesgos de impacto positivo pueden incrementar su probabilidad de ocurrencia. Entre los ejemplos de mejorar las oportunidades se cuenta la adición de más recursos a una actividad para terminar más pronto.

■ **Compartir**

Implica asignar toda o parte de la propiedad de la oportunidad a un tercero mejor capacitado para capturar la oportunidad en beneficio del proyecto. Entre los ejemplos de acciones de compartir se cuentan la formación de asociaciones de riesgo conjunto, equipos, empresas con finalidades especiales o uniones temporales de empresas, que se pueden establecer con el propósito expreso de aprovechar la oportunidad, de modo que todas las partes se beneficien a partir de sus acciones.

■ **Aceptar**

Es estar dispuesto a aprovechar la oportunidad si se presenta, pero sin buscarla de manera activa.

Estrategia de respuesta a contingencias

Algunas estrategias de respuesta se diseñan para ser usadas únicamente si se producen determinados eventos. Para algunos riesgos, resulta apropiado para el equipo del proyecto elaborar un plan de respuesta que sólo se ejecutará bajo determinadas condiciones predefinidas, cuando se prevé que habrá suficientes señales de advertencia para implementar el plan.

Se deben definir y rastrear los eventos que disparan la respuesta para contingencias. Las respuestas a los riesgos identificadas mediante esta técnica se denominan a menudo planes de contingencia o planes de reserva, e incluyen los eventos desencadenantes identificados que ponen en marcha los planes.

10.10.5. Monitoreo de riesgos

Es el seguimiento periódico para detectar riesgos nuevos, riesgos que cambian o riesgos que se tornan obsoletos. Implica:

- Seguimiento de condiciones que disparan planes de contingencia
- Revisar la ejecución de respuesta a riesgos y efectividad
- Ejecución de planes de contingencia
- Modificación del plan

10.11. Seguimiento

Tenemos varias técnicas de medición para controlar el alcance de una tarea

■ **De fórmula fija**

- Tareas no comenzadas: 0
- Tareas comenzadas se les asigna un porcentaje fijo al final del primer período y el resto al complejarse. Por ejemplo 50/50, 25/75 o 0/100.
- Es apropiado para tareas cortas.

- **Hitos con peso**

Se les da un valor a cada hito. Es apropiado para tareas más largas, con entregables intermedios.

- **Porcentaje de completitud**

El responsable de la tarea estima el porcentaje completado de la tarea. Hay que tener cuidado con el síndrome del 90 %.

10.11.1. Enfoque de valor ganado

Modelo en el que unifican todas las actividades planificadas llevándolas a \$ por su costo planificado. Es posible controlar si se logró el avance previsto y si costó lo previsto. Se puede obtener: % de avance, días de atraso, desviación de costos.

Definiciones

- **Valor planificado (PV)**

Lo que tendría que tener hecho hoy, al valor que estimé. Por ejemplo, al día de hoy tenía planificado terminar un componente de 100 horas a 40 USD la hora, entonces $PV = 4000$

- **Valor ganado (EV)**

Lo que hice hasta ahora, al valor que estimé. Hasta el momento tengo el 80 % del componente desarrollado considerando el presupuesto acordado, entonces $EV = 0,8 \times 4000 = 3200$

- **Costo actual (AC)**

Lo que llevo gastado para el trabajo que hice. Llevo 90 horas dedicadas, entonces $AC = 3600$.

- **Varianza de costos (CV)**

Indica si estamos por arriba o debajo del presupuesto. Tenemos que $CV = EV - AC$. En nuestro ejemplo $CV = 3200 - 3600 = -400$.

- **Variación de cronograma (SV)**

Indica si estamos adelantados o atrasados. Tenemos que $SV = EV - PV$. En nuestro ejemplo $SV = 3200 - 4000 = -800$.

- **Cost performance index (CPI)**

Indica cuán eficiente estamos usando los recursos. Menor a 1 indica sobre costo. $CPI = \frac{EV}{AC}$. En nuestro ejemplo $CPI = \frac{3200}{3600} = 0,89$

- **Schedule performance index (SPI)**

Indica cuán eficiente estamos usando el tiempo. Menor a 1 indica atraso. $SPI = \frac{EV}{PV}$. En nuestro ejemplo: $SPI = \frac{3200}{4000} = 0,8$

- **¿Cuánto costará finalmente el proyecto?**

Calculamos como $EAC = \frac{BAC}{CPI}$ donde BAC es el presupuesto planificado. Para esto se supone variación típica y manteniendo el mismo índice de desempeño del costo (CPI) registrado.

- **¿Cuándo terminará el proyecto?**

Calculamos como $EAC(t) = \frac{FP}{SPI}$ donde FP es el final planificado. Para esto se supone variación típica y manteniendo el mismo índice de desempeño de cronograma (SPI) registrado.

10.12. Equipo

Un equipo es un grupo cohesivo, es decir, una unidad motivados por el éxito del equipo y no el personal. El tener un grupo cohesivo tiene los siguientes beneficios:

- Estándares propios
- Aprendizaje compartido
- Compartir conocimiento
- Mejora continua

Los siguientes factores afectan a un equipo:

- Miembros del equipo: se busca un balance entre habilidades técnicas y personalidades, como también un balance de personalidades.
- Organización
- Comunicaciones

Los siguientes son factores críticos que influyen en la relación:

- Consistencia: todos tratados de la misma forma, igualdad.
- Respetar la diferencia: diferentes skills, oportunidad de contribuir.
- Inclusión: sentimiento de igualdad, todas las opiniones son consideradas.
- Honestidad: transparencia respecto al estado del proyecto, conocimientos, ejemplo.

Queremos desarrollar un equipo por las siguientes razones:

- Mejorar el conocimiento y las habilidades de los miembros del equipo para aumentar su capacidad de completar los entregables a la vez que se disminuyen los costos, se reduce el cronograma y se mejora la calidad.
- Mejorar los sentimientos de confianza y cohesión entre los miembros del equipo a fin de elevar la moral, disminuir los conflictos y fomentar el trabajo en equipo.
- Crear una cultura de equipo dinámico y cohesivo

10.12.1. Motivación

Hay muchas teorías motivacionales, por ejemplo:

- Jerarquía de Maslow



- Teoría de Herzberg



- **Factores de higiene:** pueden destruir la motivación, pero mejorarlos, en la mayoría de los casos, no mejorará la motivación. Algunos ejemplos de factores de higiene son: Condiciones de trabajo, Salario, Vida personal, Relaciones en el trabajo, Seguridad.
- **Factores de motivación:** Lo que motiva a las personas es el trabajo en sí. Ejemplos: responsabilidad, auto-realización, reconocimiento,etc
- Teoría X y Teoría Y de McGregor
Presenta dos supuestos opuestos que hacen los gerentes en cuanto a la naturaleza humana.

Teoría X	Teoría Y
<ul style="list-style-type: none"> • No les gusta su trabajo y tratarán de evitarlo • No tienen ambición ni capacidad de resolver problemas o ser creativos • Prefieren ser constantemente dirigidos y evitan tomar responsabilidad e iniciativas • Están motivados por las dos necesidades más básicas de Maslow • Son egoístas, indiferentes a las necesidades de la organización y resisten el cambio 	<ul style="list-style-type: none"> • Cumplen altas expectativas si son apropiadamente motivados y si el clima de trabajo les da apoyo • Son creativos, imaginativos, ambiciosos y comprometidos a cumplir los objetivos organizacionales • Son autodisciplinados, pueden autodirigirse, desean responsabilidades y las aceptan. • Están motivados por las necesidades más altas según Maslow

De acuerdo Bass y Duntzman, existen tres tipos de personalidades respecto a la motivación

- **Orientado a tareas:** motivado por el trabajo en si. Los motiva los desafíos intelectuales.
- **Orientado a sí mismo:** buscan el éxito personal y reconocimiento. Tienen objetivos a largo plazo. Buscan el éxito en el trabajo para realizar sus objetivos.
- **Orientado a interacción:** motivados por trabajo con otros.

10.12.2. Comunicación

Barreras de codificación	Barreras de decodificación
Habilidades de comunicación Marco de referencia Credibilidad del emisor Personalidad, intereses Sensibilidad interpersonal Actitud, emoción, auto-interés Suposiciones sobre los receptores Relación preexistente con los receptores	Tendencia evaluatoria Preconceptos Habilidades de comunicación Personalidad, intereses Sensibilidad interpersonal Actitud, emoción, auto-interés Posición, status Suposiciones sobre el emisor Relación preexistente con el emisor Falta de retroalimentación responsable Escucha selectiva

- La cantidad de posibles canales de comunicación en un equipo de proyecto está dada por la fórmula $n.(n + 1)/2$ donde n es el número de integrantes.
- El impacto de un mensaje se divide de la siguiente forma:
 - 7 % palabras
 - 65 % expresión facial
 - 38 % tono de voz
- Las siguientes habilidades son útiles para mejorar la comunicación: hacer preguntas, parafrasear, resumir, hacer contacto visual, comprender la real intención del mensaje y escucha activa

10.12.3. Etapas del desarrollo de un equipo

Tuckman & Jensen plantean 5 etapas para el desarrollo de un equipo. Un equipo puede estancarse en una de las etapas o retroceder. En equipos que ya trabajan juntos se puede saltar alguna de las etapas. La duración de una etapa depende del tamaño y liderazgo del equipo. Estas 5 etapas son:

- **Formación**

Esta es la fase en que se reúne el equipo y se informa acerca del proyecto y de cuáles son sus roles y responsabilidades formales. En esta fase, los miembros del equipo tienden a actuar de manera independiente y no demasiado abierta

- **Turbulencia**

Durante esta fase, el equipo comienza a abordar el trabajo del proyecto, las decisiones técnicas y el enfoque de dirección del proyecto. Si los miembros del equipo no colaboran ni se muestran abiertos a ideas y perspectivas diferentes, el ambiente puede tornarse contraproducente.

- **Normalización**

En la fase de normalización, los miembros del equipo comienzan a trabajar conjuntamente y a ajustar sus hábitos y comportamientos para apoyar al equipo. Los miembros del equipo comienzan a confiar unos en otros.

- **Desempeño**

Los equipos que alcanzan la etapa de desempeño funcionan como una unidad bien organizada. Son interdependientes y afrontan los problemas con eficacia y sin complicaciones.

- **Disolución**

En la fase de disolución, el equipo completa el trabajo y se desliga del proyecto. Esto sucede normalmente cuando se libera al personal del proyecto, al estar completos los entregables o como parte de la ejecución del proceso Cerrar el Proyecto.

10.12.4. Gestión de conflictos

Los conflictos son naturales e inevitables. Algunas fuentes de conflictos son:

- Cronograma
- Choques de personalidad
- Costos
- Opiniones técnicas, etc.

Su resolución debe de centrarse en el problema, no en las personas. Debe centrarse en el presente, no en el pasado.

Técnicas de gestión de conflictos

- **Retirarse/Eludir:** Retirarse de situaciones de conflicto reales o potenciales. Es una táctica de retraso que “enfría” la situación.
- **Suavizar/Adaptarse:** Suavizar las diferencias y resaltar los puntos en común. No resuelve el problema pero mantiene un ambiente cordial.
- **Consensuar/Conciliar:** Buscar soluciones que aporten un cierto grado de satisfacción a todas las partes.
- **Forzar/Dirigir:** Imponer su propio punto de vista. Una de las partes gana y la otra pierde.
- **Colaborar/Resolver el problema:** Incorporar múltiples puntos de vista y visiones desde diferentes perspectivas; requiere una actitud colaboradora y un diálogo abierto que normalmente conduce al consenso y al compromiso.

10.13. Desarrollo ágil

Se entrega el software en incrementos. No se hace un plan exhaustivo, se va decidiendo según el avance y las prioridades del cliente.

El Manifiesto ágil (2001) plantea:

- Individuos e interacciones sobre proceso y herramientas
- SW funcionando sobre documentación extensiva
- Colaboración con el cliente sobre negociación contractual
- Respuesta ante el cambio sobre seguir el plan

Tiene además 12 principios, algunos de ellos son:

- Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor
- Aceptamos que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos Ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.
- Los proyectos se desarrollan en torno a individuos motivados. Hay que darles el entorno y el apoyo que necesitan, y confiables la ejecución del trabajo.
- Las mejores arquitecturas, requisitos y diseños emergen de equipos auto-organizados.

10.14. Desarrollo ágil - SCRUM

Un marco de trabajo mediante el cual las personas pueden hacer frente a problemas adaptativos complejos, mientras entregan, creativa y productivamente, productos del mayor valor posible.

- Valores

- **Foco:** en conjunto acotado de funcionalidades por vez.
- **Coraje:** apoyo como equipo para afrontar desafíos.
- **Apertura:** transparencia y discusión abierta.
- **Compromiso:** equipos comprometidos.
- **Respeto:** respeto mutuo y ayuda.

- Pilares

- Transparencia
- Inspección
- Adaptación

- Roles

- PO (Product Owner).
- Equipo de desarrollo
- SM (Scrum Master). “Líder, facilitador, provocador, detective y soplador de brasas”.
- El PM no tiene tantas responsabilidades. Se debe transformar su rol.

- Artefactos

- Product backlog
- Sprint backlog
- Incremento del producto

- Eventos SCRUM

- Sprint
- Sprint planning
- Scrum diario
- Revisión del sprint
- Retrospectiva
- Refinamiento del product backlog.

- MVP (Minimum Viable Product)

- Es la versión mínima de un producto tal que nos permite recolectar la mayor cantidad de información de nuestro mercado con el menor esfuerzo.
- Haciendo foco en características mínimas y necesarias para que el producto pueda lanzarse

Las historias de usuario en SCRUM deben cumplir la regla INVEST.

- **Independiente:** las historias pueden completarse en cualquier orden.
- **Negociable:** los detalles de la historia son cocreados por los programadores y los clientes durante el desarrollo.
- **Valiosa:** la funcionalidad es valiosa para los clientes o los usuarios del software.
- **Estimable:** los programadores pueden encontrar una estimación razonable para construir la historia.
- **Small/Pequeña:** las historias deberían construirse en poco tiempo, generalmente alrededor de "días/persona". Se tienen que poder construir muchas historias en una iteración.
- **Testeable:** se debe poder escribir pruebas que verifiquen que el software de la historia funcione adecuadamente.

Otros conceptos importantes:

- **EPIC:** agrupación de varias historias de usuario.
- **DOR:** definition of ready. Indica cuando una historia de usuario está lista para ingresar a un sprint.
- **DOD:** definition of done. Indica cuando una historia de usuario se considera como lista.
- **Estimaciones:** En cada iteración re-estimamos. Utilizamos “Planning poker” para estimar. Está basada en Delphi, normalmente se utiliza la sucesión Fibonacci como puntos de historia. Hay distintas posturas respecto a utilizar o no la velocity del equipo.
- **Release plan:** plan de sprints de acuerdo a la velocidad del equipo considerando prioridades.
- **Sprint Burn down chart:** sirve para que el equipo pueda ver día a día cómo va respecto al plan. Se puede usar para ver tendencia del sprint.

11. Gestión de la configuración de Software

Los sistemas de software están en un continuo cambio durante su desarrollo y su uso. La gestión de la configuración comprende políticas, procesos y herramientas para gestionar los cambios en los sistemas de software. Su propósito es evitar perder tiempo en modificar versiones incorrectas del sistema, entregar la versión incorrecta a los clientes o perder el código fuente de una versión del sistema o de un componente. También busca evitar el retrabajo.

11.1. Actividades de la gestión de la configuración

- **Gestión de versiones:** Llevar el registro de las múltiples versiones de los componentes del sistema y asegurarse que los cambios realizados en los componentes por diferentes desarrolladores no interfieran entre si.
- **Armado del sistema (building):** Es el proceso de ensamblar los componentes, datos y bibliotecas y luego compilar los mismos y generar un ejecutable.
- **Gestión de cambios:** Mantener registro de los requerimientos de cambios que realizan los clientes y desarrolladores, obtener el costo y evaluar el impacto de los mismos y decidir cuales pueden ser implementados. Seguimiento del proceso de realizar un cambio.
- **Gestión de la liberación:** Preparación del software para la liberación externa y llevar un registro de las diferentes versiones del sistema que han sido liberados para uso del cliente.

11.2. Glosario de términos

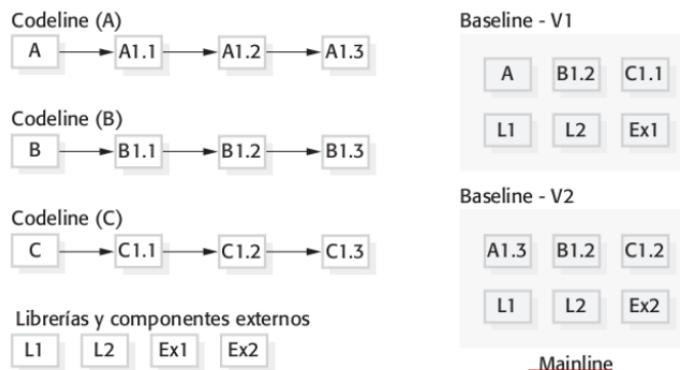
- **Ítem de configuración:** cualquier cosa relacionada al proyecto de software (diseño, código, datos de pruebas, documentos, etc.) que ha sido puesto bajo control de la configuración. En general hay varias versiones de un ítem de configuración. Tienen un identificador único.
- **Control de configuración:** es el proceso mediante el cual se asegura que se registran las distintas versiones de un sistema y sus componentes de modo de poder gestionar los cambios.
- **Versión:** una instancia de un ítem de configuración que difiere de alguna manera de otras instancias del mismo ítem.
- **Baseline (línea base):** un conjunto de versiones de ítems de configuración que han sido establecidos.
- **Codeline:** todos los ítems y versiones relacionados a un componente de software.
- **Mainline:** la secuencia de líneas base que representan diferentes versiones de un sistema.
- **Release (liberación):** una versión del sistema que ha sido liberada a los clientes para su uso.
- **Workspace (ambiente de trabajo):** un área de trabajo privada en la cual se pueden realizar modificaciones sin afectar a otros desarrolladores que pueden estar usando o modificando el software.
- **Branching:** la creación de una nueva codeline a partir de una versión en una codeline existente.
- **Merging:** la creación de una nueva codeline de un componente de software a partir de unir versiones de diferentes codelines.
- **Armado del sistema:** la creación de una versión ejecutable del sistema.

Toda la gestión de la configuración se basa en el **principio de inmutabilidad**, esto es, la información congelada no se puede modificar más. Esto implica la existencia de versiones y provee mecanismos y técnicas para que un equipo de personas puedan trabajar de forma coordinada.

11.3. Gestión de versiones

Es el proceso de realizar el seguimiento de las diferentes versiones de los componentes de software o ítems de configuración y los sistemas de los cuales forman parte. Se debe asegurar que los cambios realizados por distintos desarrolladores a estas versiones no interfieran entre sí. Puede verse como el proceso de gestión de líneas de código (codelines) y líneas base.

- Codeline: es una secuencia de versiones de código fuente donde se tienen las versiones más recientes en la secuencia derivadas de las versiones anteriores.
- Baseline: es una definición de un sistema específico. Especifica las versiones de los componentes que se incluyen en el sistema más una especificación de las= librerías usadas, archivos de configuración, etc. Permite recrear una versión específica de un sistema.



11.3.1. Control de versiones

Los sistemas de control de versiones identifican, almacenan y controlan el acceso a diferentes versiones de los componentes. Existen de dos tipos:

- **Sistema centralizado**

Existe un repositorio único que contiene todas las versiones de los componentes de software. En este modelo todas las funciones de control de versiones ocurren en un servidor compartido. Si dos desarrolladores tratan de cambiar un mismo archivo al mismo tiempo, y sin un mecanismo de acceso, uno podría sobreescribir el trabajo del otro. Se basan en File locking y Version Merging.

File locking se basa en la reserva de recursos de forma explícita cuando sabemos que vamos a modificarlos. Existen operaciones para reservar un archivo en modo escritura (check out) y liberarlo luego (check in).

- **Sistema distribuido**

Múltiples versiones del repositorio de componentes existen al mismo tiempo. Tienen un enfoque de manejo de versiones entre pares (peer-to-peer). En lugar de existir un repositorio en el cual sincronizan clientes, aquí las computadoras sincronizan entre sí. No hay una copia canónica del código. Las operaciones más comunes (commit, revertir cambios, ver historial) son más rápidas y hay menor riesgo de pérdida de datos.

11.3.2. Gestión de versiones

En general incluye:

- **Identificación de las versiones y de las liberaciones:** se les asigna un identificador único.
- **Gestión de almacenamiento:** para reducir el espacio de almacenamiento necesario para múltiples versiones, las herramientas en general proveen facilidades de gestión de almacenamiento.

- **Registro histórico de cambios:** todos los cambios realizados al código del sistema o de componentes son registrados y listados.
- **Desarrollo independiente:** el sistema de gestión de versiones mantiene un seguimiento de los componentes que fueron solicitados para edición y se asegura que las modificaciones realizadas a un componente por diferentes programadores no interfieran entre sí.
- **Soporte a proyectos:** el sistema de gestión de versiones mantiene un seguimiento de los componentes que fueron solicitados para edición y se asegura que las modificaciones realizadas a un componente por diferentes programadores no interfieran entre sí.

11.4. Armado del sistema

EL armado del sistema es el proceso de crear una versión completa y ejecutable del sistema mediante la compilación y el lindeo de sus componentes, librerías externas, archivos de configuración, etc. Involucra hacer check-out de versiones de componentes que se encuentran en el repositorio gestionado mediante el sistema de gestión de versiones. Es posible utilizar una herramienta que realice el armado.

Las herramientas pueden incluir las siguientes funcionalidades:

- Generación de scripts de armado (build scripts).
- Integración con sistemas de gestión de versión.
- Recompilaciones mínimas.
- Creación del ejecutable del sistema.
- Automatización de las pruebas.
- Reporte de los resultados.
- Generación de la documentación

11.4.1. Integración continua

- Muy utilizado en metodologías ágiles.
- Se recomienda complementar con pruebas automatizadas.
- Permiten enfrentar y resolver rápidamente posibles conflictos entre desarrolladores.
- Bastante malo para sistemas muy grandes o complejos. O cuando la plataforma objetivo es diferente a la plataforma de desarrollo.

Una alternativa es la integración frecuente

- Si no es posible utilizar integración continua se pueden utilizar builds diarios o frecuentes.
- Tiene beneficios no sólo en cuanto a encontrar conflictos sino también porque fomenta la calidad de las pruebas unitarias.

11.4.2. Puntos claves

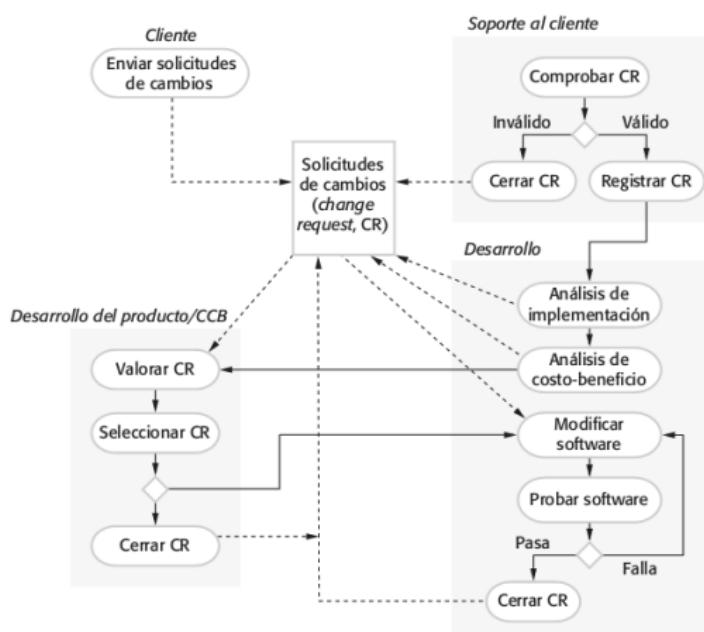
- La gestión de la configuración es la gestión de un sistema de software en evolución: al mantener un sistema, se crea un equipo de CM para asegurar que los cambios se incorporen al sistema de manera controlada y que se lleva registro de los cambios que se han implementado.
- Los principales procesos de gestión de configuración son la gestión de cambios, la gestión de versiones, el armado de sistemas y la gestión de liberaciones.
- La gestión de versiones implica el seguimiento de las diferentes versiones de componentes de software a medida que se realizan cambios en ellas.

- El armado de un sistema es el proceso de ensamblar los componentes del sistema en un programa ejecutable para operar en un sistema de computación determinado.
- El software debería ser rearmado de forma frecuente y probado inmediatamente luego de armar una nueva versión. Esto facilita la detección de defectos y problemas que pueden haberse introducido desde el armado anterior.

11.5. Gestión de cambios

Pretende asegurar que la evolución del sistema es un proceso gestionado y que se priorizan los cambios urgentes y beneficiosos. El proceso de gestión de cambios abarca analizar los costos y beneficios de las peticiones de cambio, aprobar los cambios e identificar los componentes que deben ser modificados.

11.5.1. Proceso de gestión de cambios



El comité de control de cambios debe analizar cada cambio teniendo en cuenta:

- Las consecuencias de no realizarlo.
- Los beneficios del cambio.
- El número de usuarios afectados por el cambio.
- Los costos de realizar el cambio.
- El ciclo de liberación del cambio.

Otros conceptos:

- Formulario de solicitud de cambio: formulario para solicitar un cambio en el sistema.
- Historial de cambios (changelog): documento que indica los cambios en cada versión del sistema.
- Control de cambios durante desarrollo: se usa un proceso de gestión de cambios más sencillo. Si el cambio afecta a un módulo independiente, decide el desarrollador. Si el cambio afecta a diferentes módulos, el arquitecto decide.
- Herramientas para el control de cambios: por ejemplo Bugzilla. Permiten reportar problemas para así automatizar el proceso.

11.5.2. Gestión de cambios y metodologías ágiles

- En algunas metodologías ágiles, el cliente está directamente involucrado en la gestión de los cambios.
- Considerar la propuesta de cambios a requerimientos, evaluar el impacto y decidir cuando la modificación tiene prioridad sobre las características planificadas para el siguiente incremento del sistema.
- Los cambios para mejorar el software son decididos por los programadores que trabajan en el sistema.
- Refactorizar, no es visto como una sobrecarga sino como una parte del proceso de desarrollo.

11.6. Gestión de liberaciones

Una liberación corresponde a una versión (del software) que queda liberada para su uso. Existen varios tipos de liberaciones: mayores, menores, fixes (parches), etc.

Cuando se produce una liberación se debe documentar para que pueda ser recreada en un futuro. Se debe conocer su objetivo, su composición y toda la información que se considere necesaria. Cabe destacar que liberar versiones es costoso.

11.6.1. Consideraciones al planificar una liberación

- Calidad técnica del sistema: creo versiones para reparar fallas o mejorar el sistema.
- Cambios en la plataformas.
- 5ta ley de Lehman: sugiere que si se agregan muchas funcionalidades a un sistema; también se introducen bugs a corregir en la siguiente versión.
- Competencia: si la competencia hace nuevas versiones, nosotros también debemos hacer para no quedarnos atrás.
- Requisitos del mercado.
- Propuestas del cliente.

11.6.2. Que puede incluir una versión a liberar?

- Código ejecutable
- Archivos de configuración
- Archivos de datos
- Un programa de instalación
- Documentación (en papel o electrónica)
- Packaging (empaquetado)

11.6.3. Otras consideraciones

- Requisitos de una liberación: indicar si una liberación necesita de otra anterior.
- Actualización por internet: de debe definir si será obligatorio o no, y si será gratuita o paga.

11.6.4. Software as Service (SaaS)

Simplifica la gestión de liberaciones y la instalación del sistema por parte de los clientes. El desarrollador del sistema es responsable de remplazar la versión actual con una nueva liberación. La nueva versión queda disponible para todos los clientes.

11.6.5. Puntos clave

- La gestión de cambios implica evaluar las propuestas de cambios de los clientes del sistema y otras partes interesadas y decidir si es rentable implementarlas en una nueva versión de un sistema.
- Las liberaciones del sistema incluyen el código ejecutable, datos, archivos de configuración y documentación.
- La gestión de liberaciones abarca tomar decisiones en cuanto a la fecha de la liberación, preparar toda la información a distribuir y documentar cada liberación.

11.7. Identificación, Alcance, Formalismo y Ambientes

11.7.1. Identificación

El propósito es determinar la metadata de un ítem de configuración, de modo de identificarlo de forma única y especificar sus relaciones con el mundo exterior y otros ítems de la configuración. Por ejemplo dado un sistema, se muestra el número de versión del mismo.

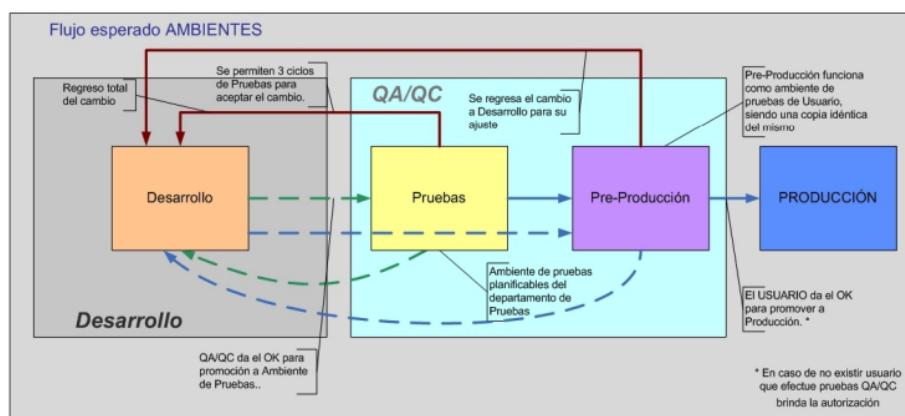
11.7.2. Alcance y formalismo

Los beneficios y costos de la Gestión de la Configuración están relacionados al alcance y al grado del formalismo.

- El alcance corresponde al número de objetos comprendidos en la gestión de la configuración.
- El grado del formalismo es el control con el cual se realizan las tareas de la gestión de la configuración.

Es imposible determinar cual alcance o grado de formalismo es el mejor, sino que la elección depende de los requerimientos y las posibilidades. En general hay un mínimo de inversión que permite evitar situaciones que serían muy costosas. Esto define un mínimo alcance que nos cubre para evitar desastres.

11.7.3. Gestión de ambientes



- **Desarrollo:** Ambiente destinado al desarrollo de aplicaciones. Controlado por el personal de desarrollo.
- **Prueba:** Ambiente aislado para la ejecución de pruebas de sistema, integrales y de retroalimentación. Este ambiente es de uso exclusivo para el área de Testing.
- **Pre-producción:** Ambiente de antecámara para la instalación de cambios listos al 100% para Producción y pruebas de sistema. Este ambiente está destinado principalmente a las pruebas de usuario.
- **Producción:** Ambiente productivo