Compilación de apuntes sobre Conceptos Fundamentales de la Ingeniería de Software

Con una visión orientada a proyectos vinculados con las Ciencias Económicas

Lic. César Ariel Briano

2023

ISBN 978-987-88-9709-7

PROHIBIDA SU COMERCIALIZACION

SEGUNDA EDCIÓN

Rev. 23.06

Briano, César Ariel
Compilación de apuntes sobre conceptos fundamentales de la Ingeniería de Software : con una visión orientada a proyectos vinculados con las Ciencias Económicas / César Ariel Briano. - 2a ed mejorada. - Ciudad Autónoma de Buenos Aires : Cesar Ariel Briano, 2023.
Libro digital, PDF

Archivo Digital: descarga y online
ISBN 978-987-88-9709-7

1. Ingeniería de Software. I. Título.
CDD 004.071

Este libro está disponible gratis on-line en https://briano.com.ar/libro

Queda prohibida la venta y distribución de este libro, en formato físico o digital, por cualquier medio que implique algún tipo de pago o compensación, así como también cualquier forma de descarga que requiera registro por parte del usuario.

Prólogo a la Segunda Edición

Una premisa fundamental de un libro de texto utilizado, como material bibliográfico en materias de la Facultad, es que se mantenga actualizado con regularidad. Aproximadamente dos años después de la primera edición, esta nueva publicación busca cumplir con ese objetivo primordial.

Esta segunda edición del libro no debe considerarse como una ampliación de la primera. En realidad, se trata una revisión integral. Si bien no se han añadido apuntes ni contenido significativamente mayor, se han realizado actualizaciones y cambios en varios de los capítulos. Además, se ha llevado a cabo una revisión exhaustiva de todo el contenido, corrigiendo errores inadvertidos, mejorando la redacción, incorporando nuevos casos y ejemplos, y actualizando otros existentes.

La ingeniería de software es una disciplina en constante cambio y evolución. Es de esperar, entonces, que este libro siga evolucionando en el futuro, incorporando en sucesivas ediciones los nuevos tópicos que merezcan ser tratados.

Prólogo

Existe una amplia variedad de material publicado sobre Ingeniería de Software, incluyendo varios libros que se han convertido en best sellers y referencias globales en el tema. Sin embargo, seleccionar la bibliografía adecuada para algunas de las materias de la Licenciatura en Sistemas de Facultad de Ciencias Económicas de la UBA es siempre un desafío. Esto se debe a nuestro perfil particular, y a que la mayoría de los retos que enfrentarán nuestros profesionales en el desarrollo de software están relacionados con sistemas de gestión para organizaciones de diversos tipos. Lamentablemente, la bibliografía especializada a menudo tiene un alcance o profundidad que no siempre cubre nuestras necesidades específicas. Algunos textos son excesivamente técnicos, otros se enfocan en ejemplos que nos son ajenos, o están basados en la problemática de otros países

Es por esta razón que, durante varios años, he decidido armar una serie de apuntes complementarios a la bibliografía. Con el tiempo, estos apuntes se han ido actualizando y expandiendo para abarcar más temas relevantes. Finalmente, ha surgido la oportunidad de compilarlos y crear esta publicación.

Este libro recopila 11 apuntes que exploran las diferentes etapas del proceso de construcción de software. Cada uno de ellos aborda temáticas que a menudo presentan enfoques diferentes en la bibliografía de referencia. Además, se enfatiza la importancia de abordar cada tema desde una perspectiva local, considerando el contexto de las organizaciones y las ciencias económicas

Con esta compilación, se pretende cerrar la brecha con la bibliografía especializada, al ofrecer un enfoque más adaptado a los desafíos y necesidades que nuestros graduados enfrentarán en su desarrollo profesional. Estos apuntes recopilan, además, mi propia experiencia en el desarrollo y gestión de proyectos informáticos, con el objetivo de enriquecer la formación de los estudiantes en esta materia.

Es crucial resaltar que esta compilación no pretende ser un resumen de ningún libro. Ni siquiera abarca todos los aspectos relacionados con la ingeniería de software. Esta disciplina es mucho más amplia y compleja de lo que estos apuntes pueden abarcar. Es importante reconocer que el estudio de otras fuentes bibliográficas es necesario e irremplazable. La compilación de apuntes es solo una herramienta adicional, que puede enriquecer y complementar el estudio de la ingeniería de software, pero no debe considerarse como un sustituto de otras fuentes de conocimiento y referencia.

Índice

Apunte 1 Conceptos Fundamentales de Ingeniería de Software 8
1. Introducción 9
2. Un poco de historia 10
3. El software: Un producto ubicuo, indispensable y vital 11
4. La naturaleza del software 15
5. Los problemas al desarrollar software 18
6. El software como ventaja competitiva. 20
7. Ingeniería de Software 22
8. El proceso de desarrollo del software 23
9. Conclusiones 27
10. Bibliografía 27

Apunte 2 Conceptos Fundamentales del Desarrollo de Software Dirigido por un Plan 28
1. Introducción 29
2. Desarrollo Dirigido por Plan versus Metodologías Agiles 29
3. Desarrollo de sistemas Dirigidos por un Plan. El enfoque clásico 31
4. Una propuesta integral 32
5. Distintos problemas y sistemas, distintos modelos 35
6. Modelo “Lineal Secuencial” (“En Cascada” o “Ciclo de Vida”) 37
7. Modelo “Lineal Secuencial Iterativo o Incremental” 39
8. Modelo de “Prototipos” 40
9. Modelo en “Espiral” 42
10. Modelo de “Desarrollo Rápido de Aplicaciones” 44
11. Otros modelos 46
12. Elección de un modelo adecuado 47
13. Limitaciones 48
14. Bibliografía 48

Apunte 3 Conceptos Fundamentales del Desarrollo Ágil de Software 49
1. Introducción 50
2. Modelos Tradicionales versus Ágiles 51
3. El “Manifiesto Ágil” 52
   - Valores del manifiesto Ágil 52
   - Principios que rigen el desarrollo ágil 53
4. Programación Extrema 54
   - Valores de XP 54
   - Prácticas de XP 55
   - Roles en XP 56
   - Ciclo de desarrollo de la programación extrema 56
   - Las historias de usuario 57
   - Pruebas en XP 58
   - Programación en pares 58
   - Ventajas y desventajas de XP 59
5. Scrum 60
   - El proceso Scrum 60
   - Roles en Scrum 61
   - Las ceremonias del Scrum 62
   - El tablero Scrum 63
   - ¿Por qué utilizar Scrum? 65

Críticas a la metodología Scrum 66
Scrum, no solo para desarrollo de Software 66
6. Otras metodologías ágiles 67
Metodología Crystal 68
Método de Desarrollo de Sistemas Dinámicos 68
7. Ventajas, dificultades y limitaciones de las metodologías ágiles 69
Ventajas 69
Dificultades en la aplicación de los principios 70
Limitaciones 70
8. Casos de estudio 71
Caso A: Éxito en metodología Scrum en un banco 71
Caso B: Después de mucho tiempo y esfuerzo, se logró el objetivo. 72
Caso C: Scrum en la industria automotriz. 72
Caso D: Problemas en metodología Scrum en una institución pública 73
Caso E: Pérdida de tiempo y dinero en un banco. 74
9. Comentario final 75
10. Bibliografía y sitios consultados 75

Apunte 4 Ingeniería de Requerimientos: Claves para un desarrollo exitoso 76
1. Introducción 77
2. Los resultados no siempre son exitosos 77
3. Requerimientos contrapuestos 79
4. Tres conceptos clave sobre relevamiento 80
5. El proceso de obtención de requerimientos 81
6. Las situaciones habituales que dificultan el relevamiento 82
7. Otros problemas habitualmente no considerados 83
8. Escenarios hostiles 84
9. Algunos mitos sobre el relevamiento 85
10. Un enfoque alternativo 87
a. ¡Cuidado! Hay personas del otro lado. 88
b. Ingeniería de requerimientos versus relevamiento. 89
c. Aprovechemos el primer encuentro con el usuario. 90
11. Relevamiento y Metodologías Agiles 91
12. Conclusiones 92
13. Bibliografía 92

Apunte 5 Conceptos Fundamentales del Diseño de Software 93
1. Notas preliminares 94
2. Repaso 94
3. La etapa de diseño – Una mirada rápida 95
4. El proceso del diseño 97
5. El “arquitecto de software” 100
6. Diseño de la Interfaz de usuario 101
7. El valor económico del diseño 103
8. Bibliografía 105
ANEXO 1 – Patrones Arquitectónicos 106
ANEXO 2 – Patrones de diseño 108

Apunte 6 Conceptos Fundamentales de la Codificación de Software 118
1. Introducción 119
2. La etapa de codificación de software 119

3. ¿Qué tareas realiza un programador? 120
4. EL proceso de la programación 126
5. La elección del lenguaje de programación 127
6. Asistentes de Inteligencia Artificial 129
7. Referencias 130

Apunte 7 Conceptos Fundamentales de las Pruebas de Software 132
1. Introducción 133
2. Buscar que falle, no probar que funcione 133
3. La calidad y las pruebas 135
4. La prueba es la única actividad optativa en el proceso de desarrollo 136
5. El equipo de pruebas 138
6. Los casos de prueba 138
7. Los tipos de prueba 140
8. ¿Caja blanca o caja negra? 142
9. ¿Cuándo termina la prueba? 143
10. La depuración o debugging 145
11. Consideración final 146
12. Bibliografía 146

Apunte 8 Conceptos Fundamentales del Despliegue de Software 147
1. Introducción 148
2. Construir la versión final 148
3. El manejo de múltiples versiones 150
4. La capacitación de usuarios 152
5. Fechas de implementación 153
6. Migración de datos 156
7. Comenzando a operar 157
8. La resistencia al cambio 158
9. Bibliografía 159

Apunte 9 Conceptos Fundamentales del Mantenimiento de Software 160
1. Introducción 161
2. Definiciones 161
3. ¿Mantenimiento o Garantía? 162
4. Prueba versus Garantía 163
5. La dificultad en los contratos de mantenimiento 164
6. El mantenimiento ocurre al final, pero se piensa al inicio 165
7. La preservación de los datos 166
8. Mantenimiento y gestión de sistemas heredados 167
9. Costos del mantenimiento 167
10. Mantenimiento y Desarrollo Ágil 168
11. El dilema del mantenimiento infinito 169
12. ¿Todo el software requiere mantenimiento? 170
13. Bibliografía 171

Apunte 10 Conceptos Fundamentales de la Estimación, Costeo y Precio del Software .. 172
1. Introducción 173
2. Estimación de software 173
3. Restricciones 175
4. Costo del software 178
5. Presupuestación de software 179
6. Determinación del precio del software 180
7. Componentes ya desarrollados 181

8. Particularidades del software 182
9. Formas de venta o distribución del software. 183
10. Presupuesto y costos de un desarrollo ágil 184
11. Bibliografía 185

Apunte 11 Decisiones estratégicas antes de desarrollar software. 186
1. Introducción 187
2. El software no siempre es la solución 187
3. Desarrollar versus Comprar 190
4. “On Premises” versus “Cloud” 193
5. Tercerización (outsourcing) 194
6. Conclusión 195
7. Bibliografía 195

Licencia, acuerdo de uso y distribución 196

Apunte 1

Conceptos Fundamentales de Ingeniería de Software

1. Introducción

Es innegable que el desarrollo de software ha adquirido un papel central en nuestros días. Las organizaciones se ven en la necesidad de crear sistemas informáticos que les permitan interactuar con las nuevas generaciones de clientes y adaptarse a los nuevos modelos de negocio.

Si alguna vez faltaron ejemplos para respaldar esta afirmación, la pandemia dejó en claro, como nunca antes en la historia, la importancia del software. Durante el período de cuarentena, en el cual se nos impidió trabajar, estudiar, vender, comprar y relacionarnos de forma presencial, se hizo evidente que las operaciones a distancia, utilizando aplicaciones informáticas, eran posibles. Incluso, nos dimos cuenta de que ofrecían algunas ventajas en comparación con la situación anterior.

La pandemia demostró la capacidad de los sistemas para facilitar la continuidad de las actividades y permitir lo que comenzamos a llamar "la nueva normalidad". En la era de la virtualidad forzada, hemos comprendido aún más el papel crítico y fundamental del software, tanto en las empresas como en nuestra vida cotidiana. Si experimentábamos una interrupción en Zoom, los estudiantes se quedaban sin clases. Si WhatsApp dejaba de funcionar, perdíamos el contacto con nuestro entorno. Si el sitio web de una organización dejaba de operar, esta desaparecía literalmente del mundo por un tiempo. Si el campus virtual presentaba fallas, no podíamos llevar a cabo nuestras actividades académicas en la Facultad. La telemedicina, que en sus orígenes se implementó para reducir situaciones de contagio, hoy ha quedado como una práctica que hace más rápida y eficiente la atención primaria en los servicios de salud.

Todo esto conlleva una enorme responsabilidad para los desarrolladores de software. Cuando los sistemas dejan de funcionar, las organizaciones se ven afectadas en su operatividad y los perjuicios pueden ser enormes. Es crucial reconocer la importancia de garantizar la estabilidad y el correcto funcionamiento del software para todos los aspectos de nuestras vidas, desde el comercial, el educativo, el financiero, el de entretenimiento y hasta el de salud.

Lo peor de la pandemia quedó atrás, sin embargo, muchas de las transformaciones que esta nos impuso se han vuelto permanentes. Los dispositivos virtuales de enseñanza han llegado para quedarse. Los pagos digitales y por QR están desplazando al a las tarjetas y al efectivo. Preferimos los trámites virtuales, en lugar de hacer colas presenciales. Las compras en línea se han vuelto habituales. Además, las reuniones virtuales están adquiriendo cada vez más protagonismo en las organizaciones. Por supuesto, nada reemplaza un encuentro presencial con amigos, pero en situaciones en las que eso no es posible, una videollamada a veces es suficiente.

Aprendimos que internet, las aplicaciones, los teléfonos móviles y las computadoras son herramientas indispensables hacer más fácil nuestro día a día. Sin embargo, es importante tener en cuenta que para que todo esto sea posible, es imperativo que el software funcione de manera óptima y confiable.

Este trabajo tiene como objetivo profundizar en la importancia del software, tanto en el ámbito empresarial como en nuestra vida diaria. En este sentido, la aplicación de los principios de ingeniería se vuelve fundamental para construir aplicaciones de alta calidad, que satisfagan las crecientes necesidades, tanto de las organizaciones, como de los usuarios inmersos en el mundo digital. Estos principios nos ayudan a garantizar que las aplicaciones sean robustas, eficientes y confiables, y que funcionen sin fallos catastróficos o interrupciones.

2. Un poco de historia

En el Blog del Museo de Informática de la Universidad Politécnica de Valencia se encuentra la siguiente referencia sobre el origen de la Ingeniería de Software:

> *"El concepto de ingeniería del software surgió en 1968, tras una conferencia en Garmisch (Alemania) que tuvo como objetivo resolver los problemas de la crisis del software. El término crisis del software se usó desde finales de 1960 hasta mediados de 1980 para describir los frecuentes problemas que aparecían durante el proceso de desarrollo de nuevo software. Tras la aparición de nuevo hardware basado en circuitos integrados, comenzaron a desarrollarse sistemas y aplicaciones mucho más complejos, que hasta entonces no era posible construir puesto que el hardware disponible no lo permitía. Estos nuevos proyectos de desarrollo de software, en la mayoría de las ocasiones, no se terminaban a tiempo, lo cual también provocaba que el presupuesto final del software excediera de aquel que se había pactado. Algunos de estos proyectos eran tan críticos (sistemas de control de aeropuertos, equipos para medicina, etc.) que sus implicaciones iban más allá de las pérdidas millonarias que causaban. Además, en muchos casos el software no daba respuesta a las verdaderas necesidades del cliente o había que ser un usuario experto para poder utilizarlo, todo ello sumado a que el mantenimiento de los productos era complejo y muy costoso."*

En resumen, se reconoció que los métodos informales de desarrollo de software, que habían sido efectivos para proyectos pequeños, no eran adecuados para enfrentar los desafíos de mayor envergadura. Los proyectos, que solían ser llevados a cabo por grupos de programadores entusiastas y autodidactas, que experimentaban y aprendían informalmente, requerían ahora estandarización y profesionalización.

La necesidad de formalizar el desarrollo de software se hizo evidente ante la complejidad y la importancia que adquirieron los sistemas en diversos ámbitos Era evidente no podían depender de resultados aleatorios o de la habilidad de un programador excepcional, cuyo código resultaba no podía ser mantenido por otros colegas. La improvisación y la dependencia de habilidades individuales ya no eran suficientes para garantizar la calidad y la eficiencia de los sistemas. Se requería un enfoque disciplinado y estructurado, respaldado por procesos estandarizados y mejores prácticas. Era imprescindible formar profesionales especializados capaces de enfrentar la creciente complejidad de los sistemas modernos.

Como respuesta a estas demandas, era el momento de "crear" la Ingeniería de Software. Esta nueva disciplina se enfocaría en aplicar principios y técnicas de ingeniería para el desarrollo de software, abordando aspectos como el diseño, la implementación, la prueba y el mantenimiento de sistemas. La Ingeniería de Software busca establecer estándares, metodologías y herramientas que permitan gestionar la complejidad y asegurar la calidad de los sistemas desarrollados.

La profesionalización del desarrollo de software implicaba reconocer la importancia de la formación académica y la adquisición de conocimientos especializados en el campo de la ingeniería de software. Se requería contar con profesionales capacitados que comprendieran los fundamentos teóricos y prácticos de la disciplina, y que pudieran aplicarlos de manera sistemática en los proyectos.

En conclusión, el surgimiento de la necesidad de estandarizar y profesionalizar el desarrollo de software fue una respuesta a los desafíos cada vez mayores de los proyectos de mayor envergadura. La disciplina de la ingeniería de software surgió como un enfoque más estructurado y profesional para garantizar el éxito en el desarrollo de software a gran escala.

3. El software: Un producto ubicuo, indispensable y vital

Durante mucho tiempo, la mención del software estaba estrechamente ligada a las computadoras. Sin embargo, en la actualidad, esta asociación ha evolucionado considerablemente. Ahora, cuando escuchamos términos como Android o WhatsApp, inmediatamente los relacionamos con nuestros teléfonos móviles. El software se ha vuelto ubicuo[1]. No solo controla computadoras y teléfonos, sino también robots, automóviles, aviones, equipos médicos, juguetes, relojes, televisores, electrodomésticos, entre otros. Ya no nos sorprende encontrar versiones “smart”, “inteligentes” o “conectadas” de los productos que utilizamos a diario, ya que incorporan tanto hardware como software para potenciar sus funcionalidades.

El software forma parte, para bien o para mal, en nuestra vida cotidiana y ha cambiado el mundo tal y como lo conocíamos:

- Ha impulsado la creación de nuevos modelos de negocio revolucionarios, como Uber y las ventas por internet, que han cambiado la forma en que interactuamos y consumimos bienes y servicios.

- Ha permitido el desarrollo de nuevas tecnologías innovadoras, como la ingeniería genética, que han abierto nuevas posibilidades en campos como la medicina y la biotecnología.

- El software ha permitido la expansión de las tecnologías existentes de múltiples formas. Un ejemplo claro es la posibilidad de envío de mensajes en las redes de comunicación celular, lo cual sentó las bases para una gran cantidad de nuevos servicios y aplicaciones, como, por ejemplo, los mensajes instantáneos, las redes sociales, las aplicaciones de servicios al cliente y el marketing y la promoción.

- El software también ha contribuido a la obsolescencia de ciertas tecnologías, como las impresiones y los CDs de música, al ofrecer opciones digitales más convenientes y sostenibles. Por ejemplo, la distribución gratuita de este libro sería inviable si se requiriera su publicación en papel. En contraposición, el avance del software ha permitido la creación de plataformas digitales y servicios en línea que facilitan la distribución de contenido de forma instantánea y global. Esto ha llevado a un cambio significativo en la forma en que accedemos a la información y consumimos productos culturales, como libros, música y películas.

[1] El término ubicuo significa que está presente en todas partes, ampliamente distribuido y accesible en diferentes contextos. En el contexto del software, cuando se dice que es "ubicuo", significa que está presente en una amplia variedad de dispositivos y sistemas, abarcando desde computadoras y teléfonos hasta otros dispositivos electrónicos y objetos cotidianos.

- La inteligencia artificial potencia la investigación y amplía la capacidad humana en diversos campos, como el desarrollo de tratamientos médicos y vacunas para enfermedades como el COVID-19.

- El análisis eficiente de grandes volúmenes de datos proporciona un nivel de conocimiento superior, volviendo obsoletos ciertos paradigmas laborales, como las auditorías, que ya no requieren muestras debido a la capacidad de procesar fácilmente el lote completo de datos.

- Ha dejado de ser exclusivamente un producto empresarial y se ha convertido en un producto comercial de amplia disponibilidad, con la posibilidad de adquirirse tanto en supermercados, tiendas físicas y, actualmente, disponible para descargas en línea.

- Las empresas dedicadas al desarrollo de software han adquirido una influencia significativa y dominan la economía mundial. De hecho, de las 10 empresas más grandes del mundo[2], 6 de ellas desarrollan o basan su negocio principal en software y 2 más producen hardware que, obviamente, será operado por software.

- El software posee un valor dual: como producto en sí mismo y como herramienta fundamental para gestionar uno de los recursos más valiosos en la actualidad, que es la información.

- Internet, una red de hardware, ha transformado permanentemente nuestra vida gracias al software que la sustenta y potencia, brindándonos acceso instantáneo a información, comunicación global y nuevas formas de interacción social y comercial.

Los beneficios del software son innegables, pero es importante reconocer que las fallas en el software pueden tener consecuencias graves que comprometen la vida y el funcionamiento de diversas entidades y servicios. A continuación, presento algunos pocos ejemplos de las posibles consecuencias de las fallas en el software:

- En el ámbito empresarial, las fallas en el software pueden llevar a la pérdida de datos importantes, interrupción de servicios, problemas de seguridad y daño a la reputación de la empresa.

- En el sector bancario, las fallas en el software pueden resultar en errores en transacciones financieras, pérdida de dinero, exposición de información confidencial y vulnerabilidades en la seguridad financiera.

- En el gobierno, las fallas en el software pueden afectar la prestación de servicios esenciales a los ciudadanos, como la emisión de documentos, la gestión de registros y el funcionamiento de sistemas críticos.

[2] Según la Revista Forbes, a marzo del 2022, las 10 empresas más grandes del mundo son: 1) APPLE 2) MICROSOFT 3) SAUDI ARABIAN OIL 4) ALPHABET (Google) 5) AMAZON 6) TESLA 7) BERKSHIRE HATHAWAY 8) NVIDIA 9) META (Facebook, Instagram, WhatsApp) 10) TSMC (Taiwan Semiconductor Manufacturing Company)

- En las ciudades, las fallas en el software pueden tener impacto en el funcionamiento de la infraestructura urbana, como el suministro de agua, la gestión del tráfico y la seguridad pública.

- En la industria aeronáutica, las fallas en el software de los aviones pueden poner en peligro la seguridad de los pasajeros y la tripulación, lo que resalta la importancia de la calidad y fiabilidad del software utilizado en este ámbito.

- En los servicios de emergencia, las fallas en el software pueden afectar la capacidad de respuesta y coordinación en situaciones críticas, poniendo en riesgo la vida de las personas que necesitan ayuda urgente.

Y en este punto cabe entonces afirmar que **el desarrollo de software es una actividad crítica. Entregar software con fallas puede poner en riesgo y de modo directo, la salud, la seguridad, los derechos o los bienes de las personas.** Los profesionales que desarrollan software tienen una responsabilidad fundamental en asegurar que el software que crean cumpla con las mejores prácticas de desarrollo, siga estándares de calidad y sea seguro y confiable. Esto implica utilizar métodos y técnicas de desarrollo adecuadas, realizar pruebas exhaustivas, implementar medidas de seguridad y seguir principios éticos en su trabajo[3].

Pueden presentarse cientos de ejemplos para reforzar estar idea, pero estos tres casos son suficientes para entenderla:

El Boeing 787 podría pararse de repente por error software

Las tragedias aéreas, desgraciadamente, están sucediéndose con demasiada frecuencia en los últimos tiempos. Pero más allá de errores humanos o de maquinaria puntuales, nos llama estrepitosamente la atención que un avión se pueda parar en pleno vuelvo. Sí, tal cual, mientras vuelas felizmente, en un golpe de mala suerte, y por un error de software, un Boeing 787 podría detenerse. Y lo peor de todo es que el piloto no podría hacer nada para evitarlo.
Esto ocurre cuatro años después de que el Boeing 787 comenzase a operar de manera comercial, extendiendo así su innovación al terreno de la aviación comercial.
Aportando más detalles sobre el problema en cuestión, se ha conocido que, si los generadores eléctricos han permanecido encendidos de manera continua en los último ocho meses, podrían desencadenar este grave problema.
Este problema en el software del Boeing 787 ha sido detectado por la administración de la aviación federal (FAA), que ha elaborado un documento con todas las pruebas derivadas de su investigación. Ahora se ha trasladado el problema a los laboratorios, para realizar pruebas mucho más exhaustivas y poder poner una solución a este fallo.
Este ejemplo nos muestra como un software con fallas puede afectar directamente la seguridad de quienes vuelan en avión.

https://computerhoy.com/noticias/software/boeing-787-podria-pararse-repente-error-software-27797

[3] Los estados pueden y deben dictar normas que aseguren el ejercicio profesional regulado, en salvaguarda de las personas.

Una revisión invalida miles de estudios del cerebro.
Un fallo informático y malas prácticas generalizadas ponen en entredicho 15 años de investigaciones

La imagen por resonancia magnética funcional (fMRI, por sus siglas en inglés) es el método más extendido para estudiar el esfuerzo que realiza una región determinada del cerebro cuando se le asigna una tarea. La fMRI detecta qué zonas están reclamando más energía del flujo sanguíneo gracias al oxígeno que transporta. El resultado son esos mapas en 3D de la materia gris con unas zonas iluminadas. Y los científicos nos dicen: esa es la parte de tu cabeza que se activa cuando comes chocolate, cuando piensas en Trump, cuando ves películas tristes, etc.
Ahora, un equipo de científicos liderados por Anders Eklund ha destapado que muchas de esas zonas se pudieron iluminar por error, por un fallo del software y el escaso rigor de algunos colegas. En su estudio, publicado en PNAS, cogieron 500 imágenes del cerebro en reposo, las que se usan como punto de partida para ver si a partir de ahí el cerebro hace algo. Usaron los programas más comunes para realizar tres millones de lecturas de esos cerebros en reposo. Esperaban un 5% de falsos positivos y en algunos casos dieron hasta con un 70% de situaciones en las que el programa iluminaba una región en la que no pasaba nada, dependiendo de los parámetros mucho más exhaustivas y poder poner una solución a este fallo.
Estos programas dividen el cerebro humano en 100.000 voxels, que son como los píxeles de una foto en versión tridimensional. El software interpreta las indicaciones de la resonancia magnética e indica en cuáles habría actividad, a partir de un umbral que en muchos casos ha sido más laxo de lo que debiera, propiciando falsos positivos. Además, los autores de la revisión analizaron 241 estudios y descubrieron que en el 40% no se habían aplicado las correcciones de software necesarias para asegurarse, agravando el problema de los falsos positivos.
En este caso queda demostrado como un software con fallas puede afectar directamente nuestra salud.

https://elpais.com/elpais/2016/07/26/ciencia/1469532340_615895.html

El Banco Nación duplica créditos y débitos de sus cuentas

Por un error informático, aparentemente vinculado a la Red Link, las operaciones efectuadas por los clientes del Banco Nación durante el 20 y 21 de febrero de 2021 se duplicaron. De este modo, quien recibió transferencias en esos días, vio duplicar el dinero que le ingresó en la cuenta, mientras aquellos que realizaron pagos, sufrieron en el saldo el doble descuento del dinero.
Si bien el Banco solucionó rápidamente el inconveniente y lunes ya todo estaba normalizado, esto representó un gran problema para muchas personas que contaban con ese dinero para pasar el fin de semana. Se suman 2 agravantes: era un fin de semana de vacaciones, donde muchos podían necesitar pagar alojamiento, comida o entretenimiento, y además era un fin de semana cercano a fin de mes, donde los saldos suelen ser menores.
Este es un claro ejemplo de cómo la falla en un software afecta directamente a los bienes y derechos de las personas, que no contaron con su dinero para hacer vida normal.

https://www.infobae.com/economia/2021/02/21/una-falla-tecnica-produjo-duplicacion-de-debitos-en-cuentas-de-clientes-del-banco-nacion

4. La naturaleza del software

Vale citar en este punto, las consideraciones que Roger Pressman hace respecto de la naturaleza del software en su libro "*Ingeniería de Software, un enfoque práctico*". En dicha obra destaca interesantes diferencias del desarrollo de software con respecto a la fabricación de productos tradicionales. Estas diferencias explican, en cierto modo, el motivo por el cual los modos de producción tradicional no siempre pueden adaptarse al desarrollo de software:

1. **El software se desarrolla o modifica con el intelecto; no se manufactura en el sentido clásico.**

   Aunque existen algunas similitudes entre el desarrollo de software y la fabricación tradicional, como por ejemplo en el caso del hardware, es fundamental reconocer que estas dos actividades son fundamentalmente diferentes. Mientras que en la industria tradicional se adquieren materias primas y se utilizan procesos industriales para ensamblar y transformarlos en un producto final, el desarrollo de software no sigue este mismo proceso.

   Tanto en el desarrollo de software como en la fabricación tradicional, la búsqueda de alta calidad se logra a través de un buen diseño. Sin embargo, es importante destacar que la fase de manufactura del hardware introduce desafíos de calidad que no están presentes en el desarrollo de software. Por ejemplo, si un componente defectuoso o de baja calidad se utiliza en la fabricación de un producto físico, es probable que el producto final tenga que ser descartado. En cambio, en el desarrollo de software, los defectos o errores descubiertos durante la etapa de producción pueden ser corregidos sin que el producto final se vea afectado.

   Ambas actividades dependen de personas, máquinas y herramientas, pero la relación entre los individuos dedicados y el trabajo desarrollado es muy diferente. Las dos actividades requieren la construcción de un "producto", pero los enfoques son distintos. Mientras los productos tradicionales se construyen en fábricas, procesando insumos para convertirlos en el producto final, el software, casi en su totalidad, **es una producción intelectual**.

2. **El cálculo de costos es distinto y no siguen un esquema tradicional.**

   Por lo dicho anteriormente, la ecuación básica de costos (costo = materia prima + mano de obra + costos de fabricación) no es aplicable al desarrollo de software. En este caso, los costos principales se concentran, básicamente, en el costo de los recursos que se emplean para desarrollarlo y el tiempo durante el cual son utilizados esos recursos. **Los principales recursos que utiliza la industria del software son recursos humanos.**

   Además, en un esquema productivo tradicional, los gastos de fabricación se prorratean entre todas las unidades producidas. Cada unidad del producto tiene como costo el de las materias primas que lo componen, la mano de obra directa y el proporcional de gastos operativos.

En contraste, en el desarrollo de software, **el costo principal recae en la creación de la primera unidad**. Producir otras copias, ya sea una o cien mil, prácticamente no generan costos adicionales o son marginales. Por supuesto, cuando se trata de desarrollos comerciales, el costo de crear la primera unidad se distribuye entre todas las unidades que se planifican vender. Sin embargo, en el caso de sistemas construidos para un cliente específico, este último debe asumir todos los costos. En el caso de que el desarrollador decida vender nuevamente ese software en el futuro (suponiendo que no haya restricciones contractuales que lo impidan), esto sería en su mayoría ganancia, ya que generalmente existen algunos costos asociados a la personalización o comercialización de esa segunda unidad.

Para ejemplificar lo dicho: Desarrollar una versión de Windows sale varios de millones de dólares. Por supuesto Microsoft también espera vender millones de copias de ese sistema operativo. El costo total de se proporcionará para obtener el costo de cada copia. Si después se venden más copias de las estimadas, será todo ganancia. Si se venden menos, tendrán pérdidas. Por supuesto otras industrias tienen esquemas parecidos, como por ejemplo la farmacéutica donde no cargan a la primera vacuna todos los costos que tuvo desarrollarla.

3. **La mayor parte del software se construye para un uso individualizado, aunque la industria se mueve hacia la construcción basada en componentes.**

A medida que una disciplina evoluciona, surgen conjuntos de componentes estandarizados que simplifican la creación de productos. Los tornillos estándar y los circuitos integrados preconstruidos son solo dos ejemplos de los numerosos componentes estándar que los ingenieros mecánicos y eléctricos utilizan al diseñar nuevos productos. Estos componentes reutilizables permiten que los ingenieros se enfoquen en los aspectos verdaderamente innovadores de un diseño, es decir, en las partes que representan novedades. Ya no es necesario invertir tiempo en diseñar un tornillo, simplemente se especifica el uso de una pieza como el tornillo Philips M5 de 25 mm de largo y 2 mm de diámetro.

Aunque en el ámbito del software existen facilidades para reutilizar componentes en diferentes desarrollos mediante la copia y pegado de código, es común que se construya cada nuevo sistema desde cero. Esto se debe, en parte, a la rápida evolución de la industria, lo cual hace que los programas queden obsoletos con gran frecuencia.

Sin embargo, existen numerosas ventajas al diseñar e implementar código que pueda ser reutilizado en múltiples programas. Los modernos componentes reutilizables integran tanto los datos como el procesamiento que se les aplica, lo que permite a los ingenieros de software crear nuevas aplicaciones a partir de partes que pueden ser reutilizables. Por ejemplo, las interfaces interactivas de usuario actuales se construyen utilizando objetos y código previamente desarrollados, lo que permite la rápida creación de ventanas gráficas, menús desplegables y una amplia variedad de mecanismos de interacción. Las estructuras de datos y los detalles de procesamiento necesarios para construir la interfaz están contenidos en una biblioteca de componentes reutilizables específicamente diseñada para este propósito.

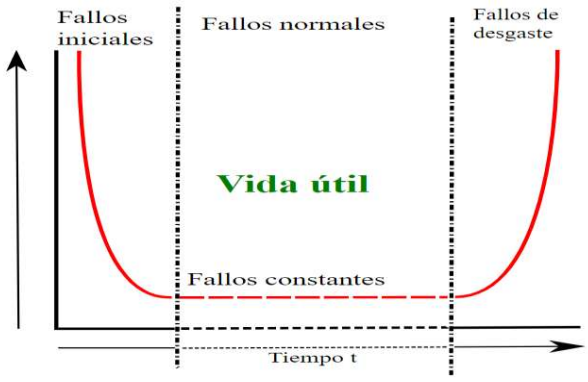
Es importante destacar, conforme lo analizado en el punto anterior, que los mayores costos que deben asumirse al construir componentes reutilizables[4], se ven recuperados con creces cuando estos componentes se incorporan en un segundo desarrollo.

También hay que mencionar que no estamos ante un proceso de reciclado o de reutilización de componentes usados. Por las características propias del software los componentes siempre funcionan a nuevo y hasta con la ventaja de tener pruebas o mejoras previas que hacen que sea quizá mejor que un desarrollo nuevo.

4. El software no se "desgasta".

En la siguiente figura se ilustra la tasa de falla del hardware como función del tiempo. La relación, que por su forma se la suele conocer como "*curva de la bañadera*", indica que el hardware presenta una tasa de fallas relativamente elevada en una etapa temprana de su vida (fallas que con frecuencia son atribuibles a defectos de diseño o manufactura). Luego defectos se corrigen y la tasa de fallas se mantiene en un nivel estable (muy bajo, generalmente) durante cierto tiempo. No obstante, conforme pasa el tiempo, la tasa de fallas aumenta de nuevo a medida que los componentes del hardware comienzan a sentir los efectos acumulativos de suciedad, vibración, abuso, temperaturas extremas y muchos otros inconvenientes ambientales. En pocas palabras, es el hardware comienza a desgastarse y a fallar.

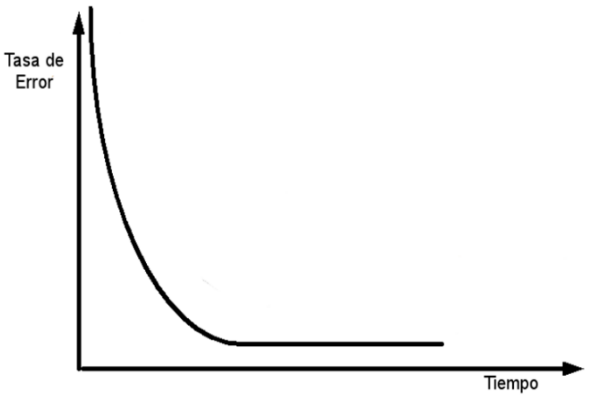
Sin embargo, no es el software el que se desgasta y suma problemas. A tal punto que, si reemplazamos el hardware por uno nuevo, el software seguirá funcionando igual que antes (o incluso mejor).



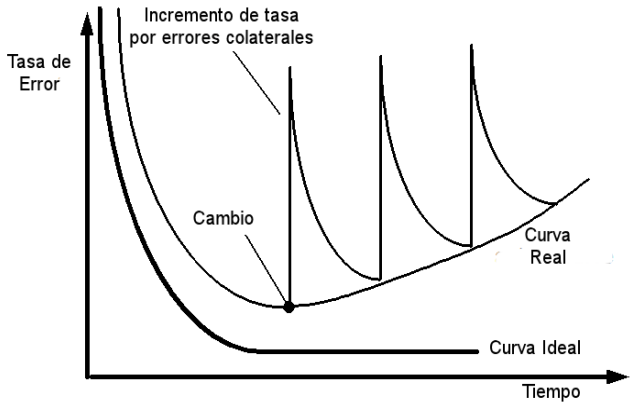
La curva de la siguiente figura muestra como los errores de programación, no detectados en la etapa de prueba, ocasionarán tasas elevadas de fallas al comienzo de la vida de un

[4] Construir un componente reutilizable implica considerar su capacidad de personalización, por ejemplo, mediante el uso de parámetros, de manera que pueda ser utilizado en diferentes partes del desarrollo o en futuras construcciones. Por ejemplo, en lugar de desarrollar un componente que muestre un mensaje de error específico, se puede crear uno genérico que reciba el mensaje como un parámetro. Este componente podrá ser reutilizado cada vez que se desee mostrar un mensaje en pantalla. Si bien desarrollar un componente genérico puede ser más complejo y costoso, los beneficios de su reutilización serán significativos y costos muy bajos en usos posteriores.

programa. Sin embargo, éstas se corrigen y la curva se aplana de modo permanente. El software continúa funcionando de este modo hasta que, finalmente deja de operar.



Claro que tampoco es cierto que el software no tenga cambios y agregados durante su vida útil. Con cada nueva versión, las tasa de "error inicial" vuelve a hacerse presente. La curva de la tasa de fallas adopta entonces la forma de la "curva idealizada" que se aprecia en la siguiente figura



La curva idealizada es una gran simplificación de los modelos reales de las fallas del software. Aun así, La implicación es clara: **el software no experimenta desgaste**, ¡pero sí se degrada! Cuando el software se somete a cambios y agregados constantes debido a circunstancias específicas, la curva de estabilidad no logra mantenerse. Cada nuevo cambio provoca que la curva se dispare nuevamente. Con el tiempo, el nivel mínimo de la tasa de fallas comienza a aumentar, lo que indica que **el software se está deteriorando como resultado de los constantes cambios**. En este punto, es importante considerar la conveniencia de reemplazar el software con otro que no requiera modificaciones permanentes.

5. Los problemas al desarrollar software

Hemos destacado previamente la importancia del software en la actualidad. Esta industria masiva se ha convertido en un factor dominante en las economías de los países industrializados. Siete de las diez empresas más grandes del mundo en términos de capitalización bursátil tienen al

software como su producto principal. Los equipos de especialistas en software, cada uno enfocado en una parte específica de la tecnología requerida para desarrollar aplicaciones complejas, han reemplazado al programador solitario de antaño, quien a menudo trabajaba en su garaje[5] con algún amigo.

A pesar de la evolución del desarrollo de software, los desafíos que enfrentan los programadores individuales en la actualidad, son similares a los que surgían al construir sistemas en el pasado:

1) **Se demora mucho tiempo en desarrollar software**. Habitualmente, más de lo previsto. No es una tarea mecánica con tiempos preestablecidos, y la complejidad del trabajo puede llevar a demoras significativas en la entrega de proyectos.

2) **Estimar los costos y plazos de entrega del software no es una tarea sencilla.** A menudo, surgen cambios y nuevas necesidades que no fueron contempladas inicialmente en el presupuesto y planificación original.

3) **Es difícil medir el avance del proyecto mientras se desarrolla o mantiene el software.** La naturaleza intangible del software y la interdependencia de sus componentes pueden dificultar la evaluación precisa del avance realizado.

4) **Los costos de desarrollo son altos, frecuentemente mucho mayores a los estimados inicialmente.** n ocasiones, es necesario presentar un presupuesto antes de contar con todos los datos necesarios para calcular los costos de manera completa y precisa (más adelante se profundizará este tema).

5) A pesar de realizar pruebas exhaustivas, **es imposible garantizar que el software entregado estará libre de errores**. Siempre existe la posibilidad de que se escapen algunos defectos o fallas en el producto final.

6) **Se gasta mucho tiempo y esfuerzo en mantenimiento**. Según los datos de la industria, entre el 60% y el 80% del esfuerzo dedicado al software ocurre después de entregarlo al cliente por primera vez.

7) **Los clientes tienen expectativas de tiempos cada vez más cortos.** Quieren que el software esté operativo en un plazo casi inmediato y no pueden esperar meses o años para obtener una solución informática completa.

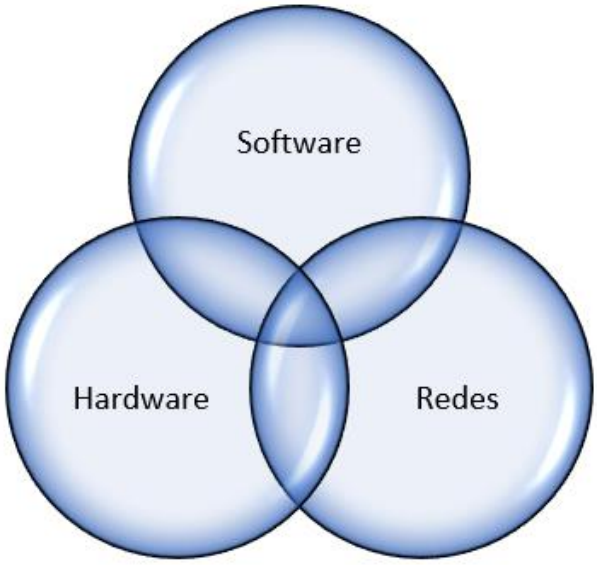
Es sorprendente que uno de los productos más destacados de nuestro siglo todavía enfrente numerosos desafíos sin resolver por completo. Surge entonces las preguntas: ¿Es posible desarrollar software de alta calidad y sin errores? ¿Existe la posibilidad de construir software seguro y evitar

[5] Muchas de las grandes compañías actuales comenzaron en los garajes de las casas de sus jóvenes fundadores. En 1975, Bill Gates y Paul Allen fundaron Microsoft en el garaje donde solían guardar el auto de la familia Gates. En 1976, Steve Jobs y Steve Wozniak iniciaron Apple en el garaje de la casa del padre de Jobs en Palo Alto, California. Al año siguiente, Larry Ellison, Bob Miner y Ed Oates fundaron Oracle también en un garaje en Santa Clara, California. Una década despues, en 1988, Google tuvo sus primeros pasos en el dormitorio del campus de Larry Page, pero pronto, junto con su socio Sergey Brin, se trasladaron al garaje que alquilaron a la señora Susan Wojcicki por 1700 dólares mensuales. Y más recientemente, en 1994, Jeff Bezos estableció la primera oficina de Amazon en el garaje de su casa en Seattle para vender sus libros on line.

fallos catastróficos? La respuesta es afirmativa, y la clave se encuentra en la aplicación de la disciplina de "**Ingeniería de Software**".

6. El software como ventaja competitiva.

Un sistema informático está compuesto por tres componentes básicos que interactúan entre sí (cuatro si consideramos al usuario). Es evidente que un software no puede funcionar sin estar respaldado por un hardware específico, al mismo tiempo que este hardware es inútil sin un software que lo administre. Las redes de comunicación se han convertido en el último componente esencial en integrarse, ya que en la actualidad resulta imposible concebir un software que funcione de manera aislada. En resumen, el hardware, el software y las redes de comunicación son los tres componentes principales de un sistema informático



Por supuesto, en cada proyecto informático, estos componentes tendrán un peso específico diferente. Algunos, por ejemplo, solo funcionarán con un hardware particular, y este se llevará buena parte de la inversión. En otros, el desarrollo será particularmente difícil, y el desafió mayor caerá sobre la construcción del software. Y por supuesto, habrá proyectos donde las redes de comunicaciones entre los dispositivos serán la parte central del mismo.

En 1985, Michael Porter publicó su libro "*Ventaja Competitiva: Creación y sostenimiento de un desempeño superior*", que introdujo un modelo conceptual llamado **Cadena de Valor** y proporcionó a las empresas nuevas herramientas de gestión. Este modelo plantea la idea fundamental de que cada actividad realizada en una empresa debe agregar más valor al producto final de lo que cuesta llevar a cabo dicha actividad. Basándose en esta cadena de valor, las empresas pueden obtener **ventajas competitivas**, lo que les permite adquirir una posición favorable en el mercado en comparación con sus competidores. Sin embargo, es importante tener en cuenta que esta ventaja competitiva no es estática ni permanente, sino que debe ser mantenida a lo largo del tiempo.

Las estrategias para crear y mantener una ventaja competitiva pueden centrarse en dos enfoques principales: la reducción de costos o la diferenciación del producto (o servicio). En el caso

de la reducción de costos, el objetivo es ofrecer un producto a un precio inferior al de los competidores. Por otro lado, la diferenciación implica ofrecer un producto, bien o servicio que sea más atractivo para los consumidores que las alternativas de la competencia.

En ambos casos, los sistemas informáticos desempeñan un papel fundamental. Estas herramientas tecnológicas permiten optimizar los procesos, mejorar la eficiencia operativa y facilitar la toma de decisiones estratégicas. Los sistemas informáticos ayudan a controlar los costos, automatizar tareas, gestionar la cadena de suministro y analizar datos relevantes para identificar oportunidades de diferenciación.

Pero… Cuál de los tres componentes es aquel que puede generar un factor diferencial y una ventaja competitiva. ¿Es el software?... ¿Es el hardware?... O ¿Es la red de comunicaciones?

¿Puede una empresa diferenciarse teniendo un hardware mejor o diferente que el de sus competidores? Difícil. Hoy el equipamiento es casi un "commodity"[6]. Se consigue fácilmente en el mercado y está disponible para cualquier empresa a un precio que puede pagar. Si bien es posible, es difícil pensar que una organización pueda montar una estructura de hardware que, por algún motivo, sea imposible de replicar por sus competidores. Es más, hoy en día es cada vez más factible que podamos reducir enormemente los costos de infraestructura reemplazando hardware y servidores por los servicios en la nube de Microsoft, Google o Amazon. Empresas más pequeñas, que antes no podían competir con las grandes porque no podían montar un centro de cómputos de envergadura, pueden hoy contratar enorme capacidad de almacenamiento y procesamiento en la nube, simplemente pagando cuotas mensuales por los servicios que consuman.

De igual modo es también poco probable que podamos montar una red de comunicaciones propia. Y aunque la construyamos, difícilmente logremos menores costos o servicios diferentes que los que nuestros competidores consiguen contratando los servicios de Personal o Telefónica.

¿Y qué pasa con el software? Ahí la cosa es distinta. En efecto puedo construir un software a medida y que agregue valor diferencial. Ese desarrollo es propio y no podrán tenerlo nuestros competidores. Y aun cuando contratemos software ya desarrollado, SAP[7], por ejemplo, las personalizaciones y diferentes implementaciones que hagamos nos darán ventajas por sobre el resto.

Para poner un ejemplo, tanto una empresa como sus competidores pueden acceder a los servicios de mapas y geolocalización de Google. Pero mientras una empresa solo lo utilizan para mostrar la ubicación de la sucursal más cercana al domicilio del cliente, otra puede mostrar en mapa el trayecto que va realizando el delivery del producto, un servicio de valor agregado que seguramente el comprador apreciará. Lo mismo ocurre con los bancos, muchos de ellos utilizan en mismo software de base, pero la implementación que hacen es diferente y el modo en el que se opera en sucursal y on-line es diferente.

[6] Se denomina **commodity** a todo bien que es producido en masa por el hombre o incluso del cual existen enormes cantidades disponibles en la naturaleza, y que por tanto tiene un valor o utilidad y un nivel de diferenciación o especialización muy escaso.

[7] SAP es una empresa alemana dedicada a desarrollas productos informáticos. Sus softwares de gestión empresarial son líderes a nivel mundial.

En definitiva, tener un mejor software que un competidor, permitirá dar un mejor servicio, reducir costos y/o aportarle al producto un valor agregado a la experiencia del cliente. No caben dudas que, hoy en día, un **buen software también permite generar ventajas competitivas y alimenta la cadena de valor de la empresa**.

Y claro, esto sirve por la positiva pero también por la negativa. Si el software falla, si no se puede hacer una venta porque el sistema este caído, si es difícil de utilizar, si no puede accederse desde dispositivos móviles, si tiene una interfaz confusa, si tiene fallas de seguridad, si queda obsoleto, si tiene menos funcionalidad que el de los competidores, no solo no se obtendrán ventajas, sino que adicionalmente habrá seguramente impacto muy malo para los negocios. Es necesario construir software de calidad. Se necesita ingeniería de software.

7. Ingeniería de Software

Las casas no se construyen simplemente apilando ladrillos con cemento. Si un obrero quiere levantar un edificio simplemente levantando paredes y pisos, está claro que, cuando llegue a cierta altura, hay altas chances de que colapse por no aguantar su propio peso.

La construcción de casas requiere un enfoque meticuloso y planificado. Antes de iniciar la construcción, es necesario realizar cálculos precisos, elaborar planos detallados y establecer cimientos sólidos. La estructura, compuesta por vigas y columnas adecuadas, debe ser construida siguiendo procesos de ingeniería y utilizando materiales de resistencia apropiada.

Además, existen procesos establecidos que ordenan las tareas de construcción. Por ejemplo, los vidrios se instalan al final para evitar daños durante la obra, mientras que los caños se colocan previamente a revocar y pintar las paredes. Es importante entender que la construcción de edificios no se limita a apilar ladrillos, sino que implica seguir rigurosos procesos de ingeniería.

En resumen, la construcción de casas involucra una serie de consideraciones técnicas y pasos organizados que garantizan la calidad y seguridad de la obra. Desde los cálculos iniciales hasta la correcta secuencia de tareas, se aplican principios de ingeniería para lograr una construcción exitosa y duradera.

Lo mismo debería pasar con el software. No se puede construir tirando líneas de código en una computadora. Para que el software no colapse en su primer uso, debe utilizarse lo que genéricamente se denomina Ingeniería de Software. **¿Qué es entonces la ingeniería de Software?** Estas son algunas de las definiciones más conocidas:

> "*Es el establecimiento y uso de principios solidos de la ingeniería para obtener económicamente un software confiable y que funcione de modo eficiente en máquinas reales*". **[Bauer, 1972]**

> "*Es el estudio de los principios y metodologías para desarrollo y mantenimiento de sistemas de software*". **[Zelkovitz, 1978]**

"*Es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo operación (funcionamiento) y mantenimiento del software: es decir, la aplicación de ingeniería al software*". **[IEEE, 1993]**

"*Es una disciplina de la ingeniería que comprende todos los aspectos de la producción de software desde las etapas iniciales de la especificación del sistema hasta el mantenimiento de este después que se utiliza*". **[Sommerville, 2004]**

"*Es una disciplina que integra el proceso, los métodos, y las herramientas para el desarrollo de software de computadora*". **[Pressman, 2005]**

Vale también en este considerar algunos aspectos claves, destacados por Ian Sommerville en su libro "Ingeniería de Software":

- La ingeniería de software es una disciplina de ingeniería que se interesa por todos los aspectos de la producción de software, no solamente por la etapa de construcción.

- El proceso de software incluye todas las actividades que intervienen en el desarrollo de software, incluso las actividades de alto nivel de especificación, desarrollo, validación y evolución.

- El software no es solo un programa o conjunto de programas, sino que también incluye documentación.

- Las ideas fundamentales de la ingeniería de software son aplicables a todos los tipos de sistemas de software. Dichos fundamentos incluyen procesos de administración de software, confiabilidad y seguridad del software, ingeniería de requerimientos y reutilización de software.

- Los ingenieros de software tienen responsabilidades con la profesión de ingeniería y tambien con la sociedad. No deben preocuparse únicamente por temas técnicos sino también contemplar los temas éticos vinculados con el desarrollo.

- Las sociedades profesionales publican usualmente códigos de conducta que establecen los estándares de comportamiento esperados de sus miembros.

8. El proceso de desarrollo del software

El proceso de desarrollo de software "*es aquel en que las necesidades del usuario son traducidas en requerimientos de software, estos requerimientos transformados en diseño y el diseño implementado en código, el código es probado, documentado y certificado para su uso operativo*". [Jacobson 1998].

Inicialmente y de forma general (este esquema se profundizará y mejorará más adelante), el desarrollo de software sigue una serie de etapas interconectadas que abarcan desde el contacto inicial con el cliente hasta la implementación del software. En algunos casos, pueden surgir nuevas versiones del software, lo que implica repetir este proceso. La siguiente representación gráfica ilustra las etapas y su interrelación:

1. **Definición Inicial:**
   En esta etapa se define globalmente el proyecto y los resultados esperados. Se hacen los acuerdos básicos, incluso un primer presupuesto inicial, que permiten comenzar con el proyecto.

2. **Análisis de los requerimientos y su viabilidad:**
   Iniciado el proyecto, se procede a recopilar, examinar y obtener todos requerimientos del cliente. También será importante detectar restricciones. En esta etapa debo establecer si es proyecto es viable de desarrollar.

3. **Diseño general y detallado:**
   A partir de los requerimientos recopilados, se procede al diseño de la aplicación informática que satisfaga las necesidades del cliente. Para ello, se plantean en primer lugar los requisitos generales de la arquitectura de la aplicación, seguidos por el detalle de cada uno de sus componentes. Este proceso requiere una precisión tal que permita su posterior construcción, siendo equivalente al plano que guía la edificación de un edificio.

4. **Codificación:**
   En esta etapa los componentes diseñados a nivel teórico se programan utilizando lenguaje de programación.

5. Pruebas:

Una vez programado, será necesario verificar que el software no contenga errores, y que cumple con los requerimientos originalmente solicitados por el usuario.

6. Implementación:

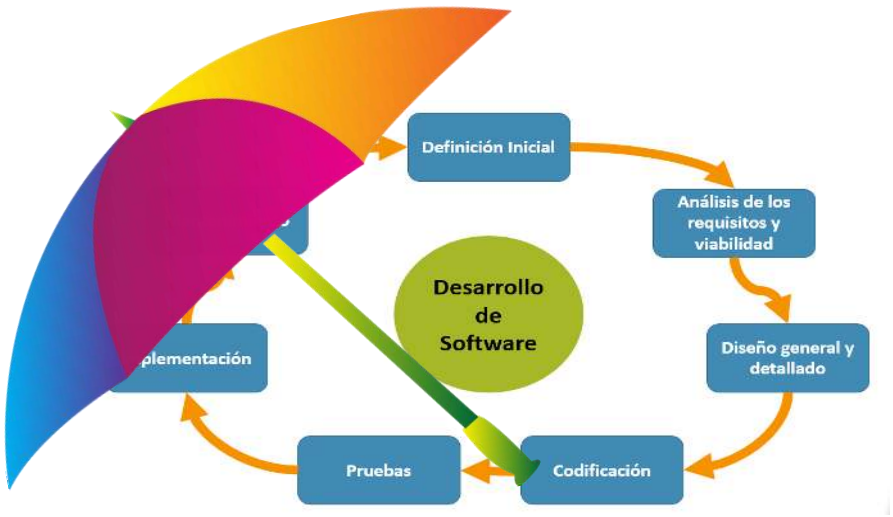
Finalmente, el software se instala en las computadoras del usuario y el sistema comienza a funcionar. Debe además asegurarse que quienes usarán el software reciban la capacitación necesaria para utilizarlo correctamente.

7. Evolución:

Una vez que el software comienza a prestar servicio, seguramente requerirá cambios, ya sea para corregir eventualmente algún error no detectado, o para incorporar nuevas funcionalidades.

Pero… ¿Bastan estas etapas para asegurar un producto de calidad? ¿Bastan para mejorar los procesos futuros de desarrollo? La respuesta es que no bastan. El responsable del proyecto de software tendrá que realizar una serie de actividades que protejan la calidad del software. Estas tareas se deben realizar en forma concomitante con las mencionadas anteriormente.

Roger Pressman identifica una serie 8 actividades que denomina "**Sombrilla**" o "**Protectoras de la calidad**". A diferencia de las etapas del proceso, estas actividades no se realizan de modo secuencial, sino que están presentes a lo largo de todo el desarrollo. Se inician al comenzar el proceso y funcionan como una sobrilla que protege a las etapas del proceso asegurando su calidad.



Las actividades sombrilla planteadas por Pressman son las siguientes:

- **Seguimiento y control del proyecto de software:** permite que el equipo de software evalúe el progreso, comparándolo con el plan del proyecto, y que pueda tomar cualquier acción necesaria corregir desvíos que se detecten.

- **Gestión del riesgo:** es necesaria una permanente evaluación de los riesgos que puedan afectar el resultado del proyecto o la calidad del producto; así como también establecer las acciones que deben realizarse para minimizar su probabilidad de ocurrencia o su impacto.

- **Aseguramiento de la calidad del software:** se establecen normas, protocolos y estándares a cumplir a los fines de cumplir a los fines de que el producto final sea un producto de aceptable calidad.

- **Revisiones técnicas:** evalúa los productos del trabajo de la ingeniería de software a fin de descubrir y eliminar errores antes de que se propaguen a la siguiente actividad.

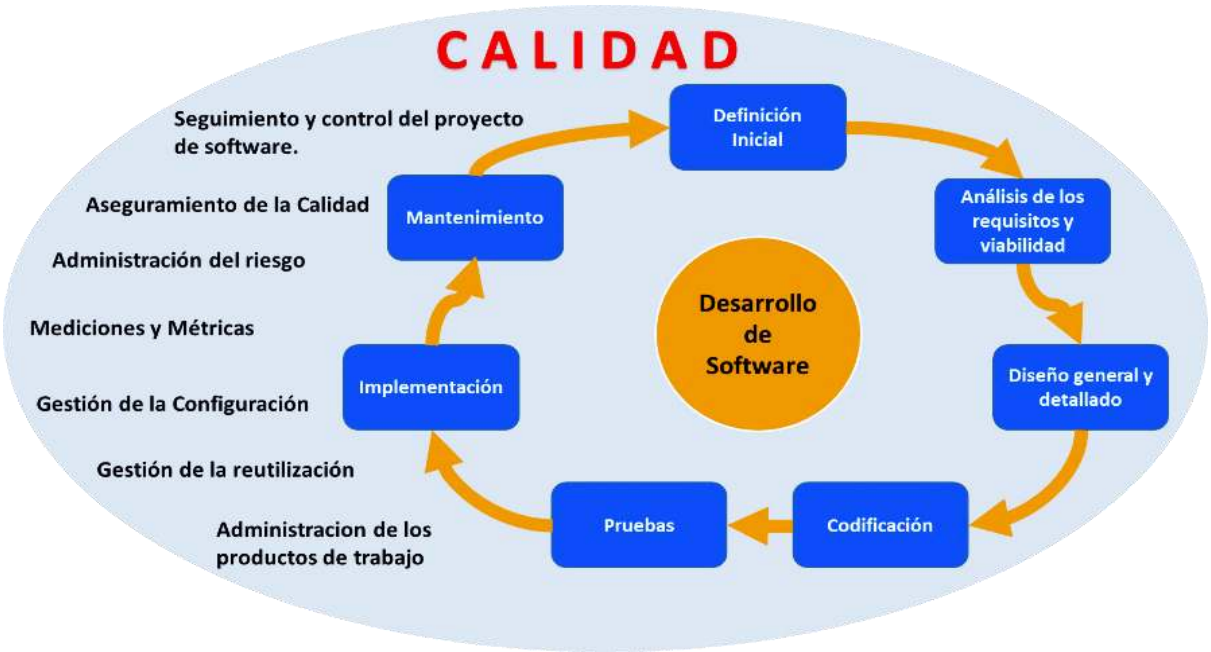
- **Mediciones y Métricas:** define y reúne mediciones para ayudar al equipo a conocer el estado del proyecto, detectar eventuales necesidades de aplicar alguna corrección, o servir al proceso de mejora continua.

- **Gestión de la configuración del software:** administra los efectos del cambio a lo largo del proceso del software.

- **Administración de la reutilización:** define criterios para volver a usar el producto del trabajo (incluso los componentes del software) y establece mecanismos para obtener, desde el inicio, componentes reutilizables en futuros proyectos.

- **Preparación y producción del producto del trabajo:** agrupa las actividades requeridas para crear productos del trabajo, tales como modelos, documentos, registros, formatos y listas.

En la siguiente imagen, puede verse de modo completo el proceso de desarrollo de software, con las etapas de desarrollo propiamente dichas, enmarcadas en un esquema de calidad, y con diversas actividades protectoras que se realizan en forma concomitante para asegurar un software de calidad.



9. Conclusiones

Entre 1985 y 1987, se produjeron una serie de incidentes trágicos relacionados con el tratamiento de pacientes oncológicos utilizando la máquina de radioterapia Therac-25. Esta máquina, que tenía un valor de un millón de dólares, presentaba fallas que resultaron en graves consecuencias para los pacientes. Los accidentes se producían cuando se activaba el haz de electrones de alta potencia en lugar del haz de baja potencia, sin tener colocada la placa difusora correspondiente. Aunque el software de la máquina detectaba el error, no impedía que el paciente recibiera una dosis de radiación potencialmente letal. En consecuencia, los usuarios eran expuestos a una radiación beta aproximadamente cien veces mayor que la dosis esperada. Varios pacientes sufrieron quemaduras y, lamentablemente, seis de ellos fallecieron como resultado de estos incidentes.

A partir de casos como el mencionado, se comenzó a reconocer la importancia crítica del software como un componente vital para las organizaciones y las personas. Surgió la necesidad de buscar diversas soluciones para desarrollar software de calidad y libre de errores. Al igual que en otras disciplinas, no existe una solución única que abarque todos los desafíos del campo de la ingeniería del software, el cual es extremadamente complejo y diverso. Sin embargo, a través de la incorporación constante de prácticas y enfoques, se ha logrado desarrollar aplicaciones de calidad aceptable que protegen a las organizaciones y a las personas, preservando su salud, seguridad, derechos y bienes. Estas prácticas, en conjunto, constituyen la base de la ingeniería del software y continúan evolucionando hasta el día de hoy.

10. Bibliografía

IAN SOMMERVILLE: “Software Engineering”. 10ma edición. 2016. Pearson Education.
ROGER PRESSMAN: Ingeniería del Software. 7ta Edición. 2008. Ed. McGraw-Hill.
MICHAEL PORTER: Ventaja Competitiva. 2da Edición. Grupo Editorial Patria. 1987
SHARI LAWRENCE PFLEEGER: Software Engineering: Theory and Practice. 4th Edition. 2009 Pearson.
Universitat Politécnica de Valencia: Blog de Historia de la Informática, Museo de Informática, https://histinf.blogs.upv.es/2010/12/28/ingenieria-del-software/

Apunte 2

Conceptos Fundamentales del Desarrollo de Software Dirigido por un Plan

1. Introducción

Muchas profesiones cuentan con protocolos o guías que brindan directrices para abordar su arte, ciencia o técnica. Un pintor sabe que, antes de comenzar a pintar su obra, debe preparar adecuadamente el lienzo siguiendo la técnica de pintura que utilizará. Un médico que realiza una visita domiciliaria sabe que primero debe tomar la temperatura del paciente, auscultarlo y examinar su garganta antes de realizar un diagnóstico. Con base en este diagnóstico inicial, el médico puede optar por realizar estudios de laboratorio adicionales, recetar medicamentos o incluso tomar la decisión de hospitalizar al paciente si es necesario.

La construcción de software no es tan simple como sentarse y comenzar a programar. Al igual que en la construcción de edificios, un proyecto de desarrollo de software requiere seguir una serie de tareas interrelacionadas en un orden específico y cumplir ciertas pautas para asegurar el éxito del proyecto y la calidad del software resultante.

Para brindar orientación a los desarrolladores de software, se han propuesto diversos modelos de desarrollo que constituyen un marco teórico a seguir. Estos modelos han sido utilizados y probados en innumerables proyectos, y aunque su uso no es obligatorio, se recomienda seguirlos para lograr el éxito. Existen numerosos modelos entre los cuales elegir (incluso más de los que se abarcan en este libro). Es probable que alguno de ellos se adapte mejor al tipo de sistema a construir, al tipo de organización en la que se implementará e incluso al contexto específico.

2. Desarrollo Dirigido por Plan versus Metodologías Agiles

Somerville hace referencia en su libro “Ingeniería de Software” a la posibilidad de encarar proyectos de software basados en un plan previo (plan-driven). El autor se refiere con este concepto al paradigma tradicional de desarrollo de sistemas. La idea central de este paradigma es que el desarrollo comienza analizando cuáles son las necesidades del cliente y, en base a estas, elaborar un plan detallado para desarrollar el software. Este plan será el que vaya guiando una serie de etapas sucesivas, hasta que finalmente el sistema queda funcionando.

En el enfoque tradicional o dirigido por un plan, una de las premisas clave es que una vez establecido dicho plan, no se permiten cambios significativos. Contar con un plan tiene sus ventajas, ya que permite presupuestar costos y estimar tiempos. El desarrollo se vuelve más predecible, con etapas secuenciales previamente definidas. Los requisitos planteados y acordados al inicio del proyecto serán los que se reflejen en la versión final del software.

Claro que en estas ventajas también están las principales críticas: los proyectos de software rara vez son tan lineales, hay poca interacción con el cliente, los errores u omisiones de una etapa se arrastran con facilidad a etapas posteriores, los requerimientos deben obtenerse al comienzo sin posibilidad de cambio, es difícil aplicarlo en proyectos grandes o complejos.

Si bien estos condicionantes son importantes, tampoco es cierto que todas las organizaciones se muevan en escenarios cambiantes. Por el contrario, algunas utilizas procesos que llevan años funcionando del mismo modo y con eficacia. Para este tipo de entidades, los modelos de desarrollo dirigidos por un plan resultan sin duda la mejor alternativa.

En el otro extremo, en la actualidad, muchas organizaciones se comportan de un modo diferente. Sus procesos, y por ende sus sistemas, están permanente cambio y adaptación a un contexto cambiante. Los sistemas están en permanente cambio y evolución. Para estos casos, los modelos basados en un plan no son eficaces y allí es donde las actuales metodologías ágiles representan de un modo más natural el desarrollo de software. [8]

Si bien en cierto que en la actualidad muchas organizaciones prefieren utilizar metodologías ágiles de desarrollo, también siguen existiendo múltiples escenarios donde los enfoques tradicionales siguen siendo más efectivos. Incluso Somerville destaca que "la mayoría de los proyectos de software incluyen y combinan prácticas del enfoque ágil y del basado en un plan".

No es el objetivo central de este trabajo comparar una y otra metodología de construcción, pero si es interesante analizar una serie una serie de preguntas que se hace Somerville a la hora de optar por una u otra:

1. ***¿Es importante y sobre todo posible tener una especificación y un diseño muy detallados antes de dirigirse a la implementación?*** *Siendo así, probablemente usted tenga que usar un enfoque basado en un plan*.

2. ***¿Es práctica una estrategia de entrega incremental? ¿Le sirve al cliente utilizar un software con funcionalidades acotadas? ¿La retroalimentación del cliente es importante?*** *De ser el caso, considere el uso de métodos ágiles.*

3. ***¿Qué tan grande es el sistema que se desarrollará?*** *Los métodos ágiles son más efectivos cuando el sistema logra diseñarse con un pequeño equipo asignado que se comunique de manera informal. Esto sería imposible para los grandes sistemas que precisan equipos de desarrollo más amplios, de manera que tal vez se utilice un enfoque basado en un plan.*

4. ***¿Qué tipo de sistema se desarrollará?*** *Los sistemas que demandan mucho análisis antes de la implementación (por ejemplo, sistemas que responden a una normativa compleja o que se vinculan con dispositivos o maquinarias), por lo general necesitan un diseño bastante detallado antes de comenzar el desarrollo. En tales circunstancias, quizá sea mejor un enfoque basado en un plan.*

5. ***¿Cómo está organizado el equipo de desarrollo?*** *Si el equipo de desarrollo está distribuido, o si parte del desarrollo se subcontrata, entonces tal vez se requiera elaborar documentos de diseño para comunicarse a través de los equipos de desarrollo. Quizá se necesite planear por adelantado cuáles son.*

6. ***¿Existen problemas culturales que afecten el desarrollo del sistema****? Las organizaciones de tradicionales presentan una cultura de desarrollo basada en un plan, pues es una norma en ingeniería. Esto requiere comúnmente una amplia documentación de diseño, propia de los desarrollos tradicionales, en vez del conocimiento informal que se utiliza en los procesos ágiles.*

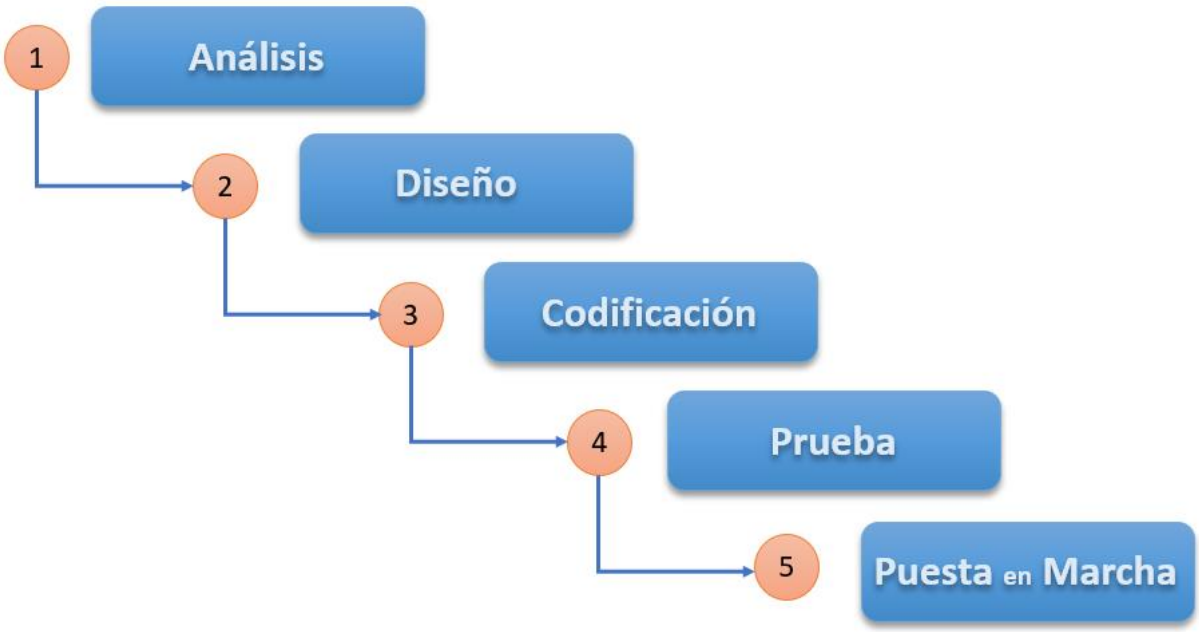
[8] El apunte "Conceptos Fundamentales del Desarrollo Ágil de Software", amplia este tema.

7. ***¿El sistema está sujeto a regulación externa?*** *Si un regulador externo tiene que aprobar el sistema (por ejemplo, la Agencia de Aviación Federal [FAA] estadounidense aprueba el software que es crítico para la operación de una aeronave), entonces, tal vez se le requerirá documentación detallada como parte del sistema de seguridad.*

Está claro que, finalmente, para **el usuario lo importante es que el software cumpla con sus requerimientos, independientemente de con que estrategia fue desarrollado**. En la práctica, muchas compañías que afirman haber usado métodos ágiles, adoptaron algunas habilidades ágiles y las integraron con sus procesos dirigidos por un plan.

3. Desarrollo de sistemas Dirigidos por un Plan. El enfoque clásico

La teoría de construcción de sistemas planteó, desde sus comienzos, que, para lograr el éxito del proyecto de software, deben desarrollarse una serie de actividades sucesivas. Este primer marco teórico planteado es conocido como **modelo lineal, modelo clásico, o secuencial**. Como su nombre lo sugiere, las etapas se encadenan. Una vez concluida una etapa, se pasa a la siguiente:



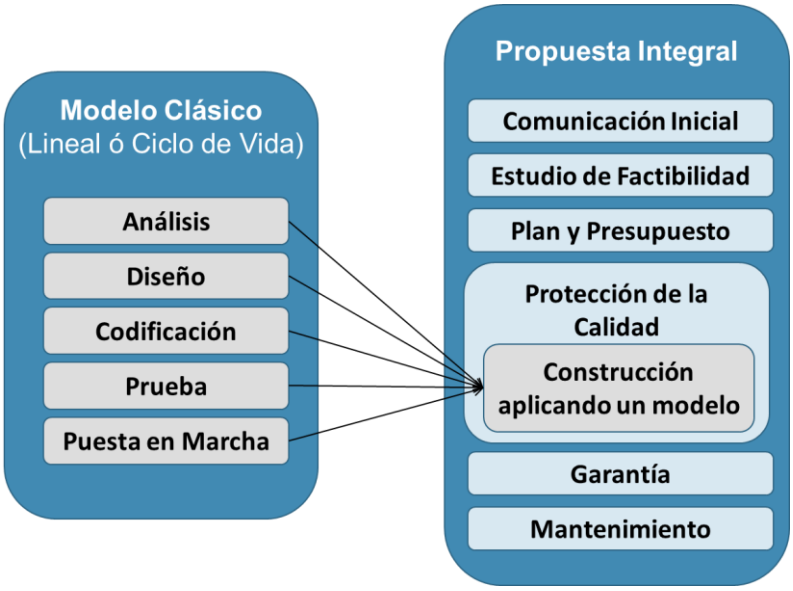
Este modelo presenta algunas variantes en cuanto su concepción. Algunos autores incluyen otras etapas, como la de relevamiento o la de mantenimiento (más adelante en el libro se desarrollará este tema) o las nombran de modo diferente (por ejemplo “implementación” o “implantación” en lugar de “puesta en marcha”). Sin embargo, la esencia es similar en todos: Un modelo de etapas secuenciales las cuales deben ser desarrolladas una tras otra hasta arribar al producto final ya instalado. Solo cuando se finaliza una etapa puede pasarse a la etapa siguiente.

Queda fuera de alcance de este trabajo el análisis exhaustivo de cada una de estas etapas. Sin embargo, es necesaria una breve descripción de cada una de ellas.

Etapa	Descripción
Análisis	Los analistas llevan a cabo un relevamiento exhaustivo de la organización para identificar sus necesidades y los problemas que deben abordarse mediante un sistema informático. El resultado de este proceso es la elaboración de un documento que, siguiendo reglas y convenciones específicas, capture todos los requerimientos de los usuarios. Asimismo, se deben identificar y tener en cuenta las restricciones y limitaciones pertinentes.
Diseño	Se elabora el diseño de una solución informática que aborde los problemas identificados, teniendo en cuenta los requerimientos obtenidos en la fase anterior y respetando las restricciones establecidas. Se genera una documentación detallada y técnica que permita a los programadores construir el código de manera efectiva.
Codificación	Los programadores llevan a cabo la etapa de codificación siguiendo las especificaciones del diseño, donde construyen el código del software empleando un lenguaje de programación determinado.
Prueba	La etapa de prueba se encarga de identificar posibles errores en el sistema antes de su puesta en uso, con el objetivo de reportarlos y corregirlos.
Puesta en marcha	Se realiza la instalación del programa en las computadoras del cliente y se proporciona capacitación a los usuarios para que puedan utilizar eficientemente el nuevo software.

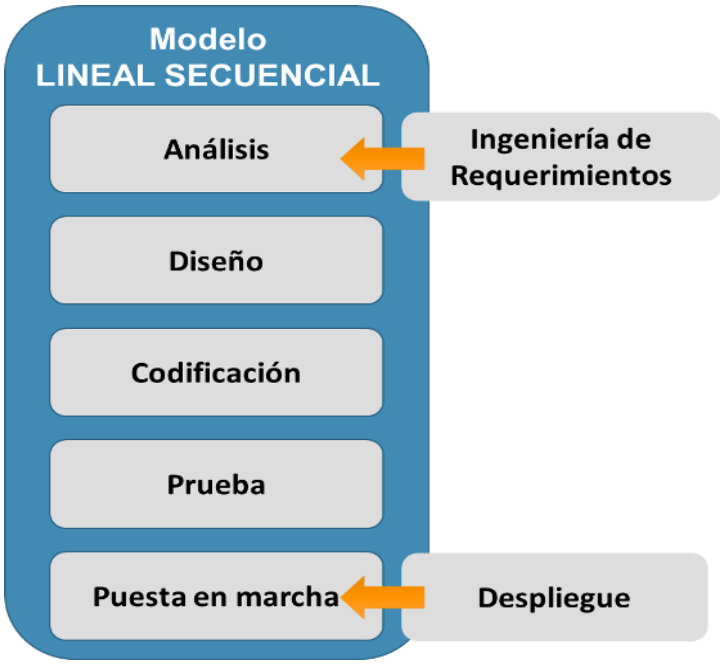
4. Una propuesta integral

Antes de adentrarnos en el análisis de los diversos modelos utilizados en la construcción de software, es importante destacar que el modelo clásico que se presenta no engloba todas las etapas existentes en un proyecto de desarrollo de aplicaciones informáticas. Por ejemplo, no es correcto afirmar que el desarrollo de software comienza directamente con la etapa de análisis, ni que estas cinco etapas son suficientes para garantizar la calidad del software resultante. El modelo tradicional podría ser mejorado al agregar algunas etapas y redefinir otras. Al combinar estas etapas con los aspectos de calidad que se han discutido en la sección anterior, la siguiente figura ilustra una propuesta integral y mejorada del modelo genérico para el desarrollo de software basado en un plan:



Etapa	Descripción
Comunicación inicial	Durante esta etapa inicial se establece el primer contacto entre el desarrollador y el cliente. En esta fase se acuerdan los lineamientos fundamentales del proyecto, se presentan de manera global los desafíos, los requerimientos principales y las restricciones existentes.
Estudio de Factibilidad	No todos los desafíos pueden ser abordados exclusivamente a través de un sistema informático o mediante un enfoque tradicional de desarrollo de software. En ocasiones, pueden surgir propuestas o enfoques alternativos. Además, las condiciones específicas del entorno, del cliente y del desarrollador, así como el presupuesto y los plazos disponibles, pueden condicionar la viabilidad de llevar a cabo un sistema. Por lo tanto, es fundamental evaluar la factibilidad del proyecto antes de decidir continuar adelante con el proyecto.
Plan y Presupuesto	Es necesario planificar el proyecto de desarrollo en su totalidad. Habrá que estimar recursos: **Humanos,** incluyendo la contratación y/o asignación temporal de personal); **de software** (licencias, entornos de desarrollo, librerías de software, componentes reutilizables, etc.) y **de entorno** (hardware, comunicaciones, espacio de trabajo, etc.). También, y en función del tipo de software a desarrollar, deberá definirse cuál será el modelo de desarrollo de software que se utilizará, los tiempos que se estima llevará cada una de las etapas y la forma en la que se desarrollarán cada una de las actividades del proyecto. Por último, en esta etapa también será necesario elaborar un primer presupuesto económico para presentarle al cliente. Por lo general no será posible presupuestar de antemano la totalidad del sistema, en cuyo caso podrá hacerse un presupuesto parcial que alcance solo las próximas etapas[9].
Construcción de la aplicación utilizando un modelo	Los modelos de desarrollo de software abarcan las etapas de **análisis, diseño, codificación, prueba y puesta en marcha**. Cada una de estas actividades tendrá particularidades y agregados según las características de cada modelo y de los sistemas a desarrollar.
Garantía	Aunque esta etapa no siempre se lleva a cabo de manera explícita y a veces se reduce a una mera declaración en la propuesta comercial, es evidente que el compromiso del desarrollador de software es que funcione de acuerdo con las especificaciones. Por lo tanto, es fundamental incluir una etapa dedicada a la corrección de errores que pueda presentar el software. El costo asociado a solucionar los posibles problemas que surjan en esta etapa debe ser considerado como parte de los costos generales del proyecto.[10]
Mantenimiento	Tanto los sistemas informáticos como las organizaciones en las que operan son dinámicos y están sujetos a cambios. Después de la instalación de un sistema, es probable que se requieran ajustes o mejoras. Esto puede implicar la incorporación de nuevas solicitudes por parte de los clientes, la adición de funcionalidades adicionales, la mejora de las existentes, la corrección de errores o la adaptación a nuevas tecnologías y cambios en el entorno. Sin embargo, es importante destacar que en algunos casos esta etapa puede no llevarse a cabo.

Una segunda propuesta consiste en ajustar los nombres de dos etapas clásicas del proceso. Sin embargo, más allá de ser solo un cambio de nombre, implica encontrar una descripción más precisa de las actividades llevadas a cabo en dichas etapas.



El término "**Ingeniería de Requerimientos**" transmite una idea más completa de esta etapa del proceso. Los requerimientos deben ser exhaustivos, abarcando tanto aquellos que surgen de entrevistas y relevamientos con los usuarios, como aquellos que requieren obtener información de diversas fuentes. Por ejemplo, es importante obtener detalles específicos, como el color institucional o los logos de la compañía. Estos elementos deben ser preguntados de manera precisa para asegurarse de incluirlos en los requerimientos.

De igual manera, el término "**Puesta en Marcha**" puede dar la impresión de que una vez que el sistema se encuentra en funcionamiento, la tarea se da por concluida. Sin embargo, en esta etapa hay muchas más actividades que deben llevarse a cabo, como la capacitación de los usuarios. En este sentido, el término "Despliegue" resulta más apropiado, ya que abarca de manera más amplia todas las acciones necesarias para poner en marcha el sistema, incluyendo tanto la instalación y configuración técnica como la formación y adaptación de los usuarios al nuevo sistema.

Más adelante en el libro se verán con mayor detalle y profundidad las etapas de ingeniería de relevamiento y despliegue.

[9] En el apunte de “Estimación y Costeo” se analizan las particularidades de la presupuestación y la dificultad de hacerlo con exactitud en el momento inicial.

[10] En el apunte referido al Mantenimiento de Software” se describen y explican las diferencias conceptuales de la etapa de garantía propuesta, con la de Mantenimiento, etapa con la que suele confundirse.

5. Distintos problemas y sistemas, distintos modelos

Es cierto, contar con un plan en el desarrollo de software tiene sus ventajas, ya que permite estimar plazos, presupuestos y garantizar que el sistema cumpla con su propósito. Esta aproximación se asemeja a la forma en que se construyen edificios u otras estructuras físicas. Sin embargo, los sistemas de software son diferentes entre sí, incluso si son similares en su naturaleza.

Cada organización tiene su propia cultura, procesos y formas de utilizar o implementar los sistemas. Por lo tanto, resulta difícil pensar que un único modelo de construcción, como el enfoque clásico o secuencial, pueda adaptarse a todos los tipos de software y a las necesidades específicas de cada organización. Es necesario evaluar las características del proyecto, las necesidades de la organización y considerar enfoques flexibles que permitan ajustarse a los cambios y lograr mejores resultados.

Lo primero que debe mencionarse es que existen varios factores que pueden complicar la construcción de un sistema y, con esto, desviarlo de sus estimaciones iniciales. Entre estos factores podemos mencionar:

- **El tamaño del software a desarrollar**:

  Aun cuando la funcionalidad o la complejidad técnica de un software sean parecidas, no es lo mismo construir un software chico que uno grande. Tampoco es lo mismo hacerlo para una pequeña empresa que para una de mayor tamaño.

  Un software más grande presupone más líneas de código, más usuarios, más requerimientos para relevar, una mayor cantidad de recursos asignados, un mayor despliegue en la implementación, una mayor capacitación, más horas de prueba, más documentación, etc.

  Por otro lado, los sistemas más pequeños, en general, tienen asignado un menor presupuesto. En estos casos el desarrollo tendrá que ser lo más sencillo posible.

  En resumen: Es más difícil desarrollar un software de gran tamaño que uno pequeño. Es más difícil mantener controlado un proyecto grande que otro más chico.

- **La complejidad para obtener requerimientos[11]:**

  Los sistemas buscan resolver problemas organizacionales. Y para esto es necesario conocer cuáles son estos problemas y cuáles son los requerimientos que tienen los diferentes usuarios e involucrados. Tambien debe tenerse en cuenta eventuales restricciones para resolverlos. Sin embargo, estos requerimientos no siempre son fáciles de obtener.

  Muchas veces la comunicación entre los especialistas en informática y los clientes no es tarea sencilla. Los usuarios no conocen de sistemas. Pueden no tener en claro sus

[11] Este tema es abordado con mayor profundidad en apunte “Claves de un Relevamiento Exitoso”

necesidades o no saberlas expresarlas correctamente. Los analistas, por su lado, suelen tener dificultades en comprender los términos propios de la organización. Los procesos más complejos agregan dificultades adicionales.

Tampoco puede asegurarse que lo que el analista entendió es exactamente lo que el usuario le quiso explicar. No es fácil, por ejemplo, describir verbalmente el comportamiento deseado una pantalla con múltiples botones.

También pueden ocultarse cosas, a veces por una simple omisión, otras veces deliberadamente. No debe olvidarse que el usuario puede asumir diversas actitudes frente a la instalación de un nuevo software. A veces puede incluso no querer implementarlo, por temor a ser desplazado, o para conservar cierto poder dentro de la organización.

Incluso debe considerarse que los usuarios no siempre tienen el tiempo suficiente para dedicarle a los analistas ya que además de reunirse por el nuevo sistema, tienen que realizar sus tareas habituales. Incluso, existe la posibilidad de que estos requerimientos no puedan ser explicitados al inicio o que cambien conforme va avanzando el proyecto.

Es importante entonces asegurar que los requerimientos sean bien comprendidos y que coincidan con las necesidades reales de la organización.

- **La complejidad técnica:**

  Realizar un software también implica desafíos técnicos, a veces mayores a los habituales Muchas veces debe desarrollarse software para un nuevo sistema operativo; o con nuevos lenguajes de programación; o vinculados con dispositivos de hardware especial (equipos portables, celulares, máquinas industriales, etc.)

  Incluso hay casos en los cuales el cliente fija determinados requerimientos tecnológicos que pueden resultar desafíos adicionales para el desarrollador, que quizá no tiene la suficiente experiencia en esa tecnología.

  Las cuestiones técnicas representan entonces un factor de complejidad adicional que debe ser tenido en cuenta.

- **El tiempo disponible:**

  Se supone que una de las cosas que se estiman y presupuestan al realizar un software es cuánto tiempo se tardará en el desarrollo.

  Sin embargo, muchas veces las organizaciones requieren tener disponible el software para una fecha específica (por ejemplo, para el lanzamiento de un nuevo producto, el cambio de período fiscal, la necesidad adelantarse a la competencia, la apertura de un nuevo local, etc.) En estos casos los plazos de desarrollos son parte del requerimiento e imponen importantes restricciones al proyecto.

- **El riesgo:**

  Cada nuevo proyecto representa un desafío y conlleva ciertos riesgos inherentes. De hecho, las situaciones que se mencionaron anteriormente presentan diferentes niveles de riesgo que pueden afectar los planes y los resultados esperados.

  Sin embargo, también existen otros riesgos específicos del proyecto que requieren una atención especial. Estas situaciones pueden requerir la redefinición e incluso la posible cancelación del proyecto. Por tanto, es crucial prestar una especial atención a estos riesgos concretos.

  La lista de posibles riesgos es larga, pero como ejemplo puede mencionarse: pérdida de apoyo político, cambios de personal, cambios en el presupuesto, cambios en la legislación, cambios en el mercado, lanzamiento de nuevo hardware, aparición de nuevo software o de un nuevo sistema operativo, cambios en sistemas de la competencia, ventas o fusiones de la empresa…

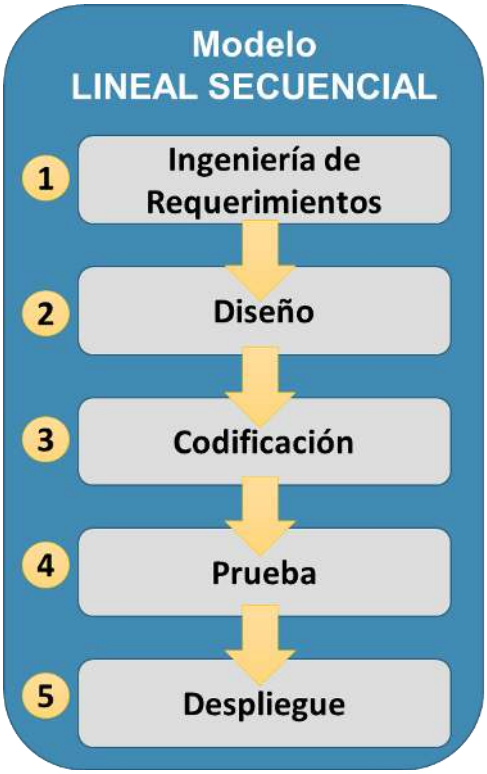
En resumen: Existen entonces diversos problemas para tener en cuenta a la hora de desarrollar software. No existe un modelo de desarrollo que pueda resolver eficazmente todos los temas planteados. Se analizarán, entonces, no uno sino varios modelos de desarrollo que podrán utilizarse conforme el escenario y los problemas que deban enfrentarse.

Es importante reconocer que el costo también puede ser un factor limitante al considerar el desarrollo de software, en adición a las posibles limitaciones de tiempo mencionadas anteriormente. Sin embargo, el presupuesto del cliente no debería ser el único factor determinante al elegir un modelo de desarrollo. Es cierto que algunos modelos pueden requerir más recursos y, por lo tanto, ser más costosos inicialmente. No obstante, es fundamental tener en cuenta otros factores al evaluar el costo total de un sistema. En algunos casos, un modelo inicialmente más costoso puede generar ahorros significativos a largo plazo. Por ejemplo, un modelo que permita una mejor comprensión de los requerimientos puede evitar reprogramaciones y costos adicionales. Por lo tanto, es crucial considerar tanto el presupuesto del cliente como los beneficios a largo plazo al tomar decisiones sobre el modelo de desarrollo de software. Aún con presupuestos limitados debe elegirse el modelo más adecuado y no necesariamente el más barato[12].

6. Modelo “Lineal Secuencial” (“En Cascada” o “Ciclo de Vida”)

Se han mencionado algunas de las desventajas y críticas del modelo lineal secuencial. No obstante, es importante destacar que, para sistemas simples y pequeños, optar por un modelo simple sigue siendo una excelente elección. Este sistema consta de 5 etapas, las cuales comparten características muy similares al modelo clásico.

[12] Este tema es abordado con mayor profundidad en apunte de “Conceptos Fundamentales de la Estimación, Costeo y Precio del Software”



Etapa	Descripción
Ingeniería de requerimientos	La etapa de ingeniería de requerimientos coincide con la fase de análisis, pero no se limita únicamente a recolectar información de los usuarios. El término "ingeniería de requerimientos" implica adoptar una actitud proactiva para identificar todos los problemas y obtener los requerimientos de manera exhaustiva y precisa, incluso aquellos que los usuarios pueden ocultar o desconocer. Además de esta interacción, es necesario analizar la documentación disponible, la normativa vigente y los procesos internos de la organización.
Diseño	Esta etapa representa la primera actividad técnica del proceso. Aquí se crea una solución que aborda los problemas identificados y se tienen en cuenta los requerimientos de los usuarios. El resultado es un modelo que servirá como base para la construcción futura del software. En el diseño se definen en detalle las estructuras de datos, las arquitecturas, las interfaces y los componentes necesarios para la programación del sistema.
Codificación	Los programadores utilizan las especificaciones del diseño como base para generar el código del software, empleando un lenguaje de programación. En esta etapa, se construye el programa en sí, siguiendo las pautas y las directrices establecidas.
Prueba	Frecuentemente se describe esta etapa como la prueba del software en busca de errores. Sin embargo, el objetivo no es probar que el sistema funcione correctamente, sino más bien identificar fallos y debilidades. Por lo tanto, se somete al programa a diversas condiciones extremas con el propósito de descubrir errores que no fueron detectados durante la etapa de codificación. El enfoque se centra en atacar y poner a prueba el software en busca de situaciones que puedan generar fallos.

Despliegue	En esta etapa, también conocida como "implementación", "puesta en marcha" o "implantación" según distintos autores, el software previamente probado se instala en el entorno productivo del cliente. Se llevan a cabo todas las tareas adicionales necesarias para asegurar que el sistema esté plenamente operativo, lo que incluye la instalación de servidores y redes, la configuración del hardware y software de los equipos de trabajo, la entrega de documentación y manuales, así como el suministro de cualquier otro material necesario para el funcionamiento del sistema. Un aspecto crucial de esta etapa es la capacitación de todos los usuarios involucrados, así como la adaptación de los circuitos administrativos a la nueva operativa basada en el sistema informático.

Ventajas	Desventajas
• Es un modelo simple, sencillo y probado. Por supuesto, es mucho mejor utilizar este modelo que no utilizar ninguno. • Para proyectos sencillos, es el modelo menos costoso y el más rápido.	• Los proyectos rara vez siguen el flujo secuencial. • Los errores de las primeras etapas se arrastran con facilidad a las etapas posteriores. Muchos errores solo se descubren al final cuando se entrega el sistema. • Pueden producirse estados de bloqueos por la secuencialidad de las etapas. • Es difícil que el cliente especifique todos los requerimientos al inicio • No prevé la posibilidad de introducir cambios en el medio del proyecto. • El cliente no ve el proyecto hasta que el software está terminado.

7. Modelo “Lineal Secuencial Iterativo o Incremental”

Puede ocurrir que, por razones de tamaño, tiempo u operativas; sea conveniente construir y entregar el sistema en partes. De este modo puede construirse una primera versión con algunas funcionalidades. Una vez implementada esta versión, se vuelve a emplear el modelo lineal secuencial para producir una nueva versión más completa.

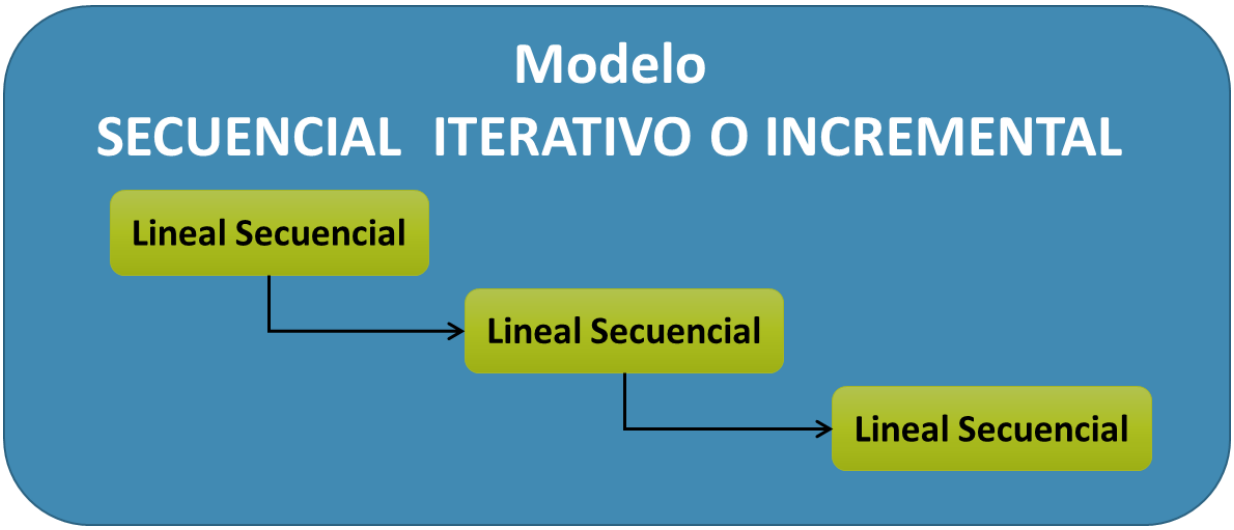
Las etapas de este modelo obviamente no difieren del modelo Lineal Secuencial ya que, en definitiva, son sucesivas iteraciones de aquel modelo.

Es importante diferenciar este modelo de la división modular de un sistema. En cada iteración **se entrega un sistema completo y operable, no módulos sueltos**. El cliente puede usar el sistema en entornos productivos, aun cuando a este no tenga completas todas sus funcionalidades.

Tampoco debe confundirse con Desarrollo Ágil. En este modelo, los componentes de la primera versión están todos planeados y definidos desde el comienzo. Las sucesivas versiones también siguen las mismas premisas: se planifica al inicio cuál será el contenido completo de la nueva versión. En contraste, en el desarrollo ágil, las mejoras se van incorporando de manera

incremental y periódica (generalmente cada 2 o 3 semanas). No existe una planificación inicial exhaustiva, sino que los requerimientos emergen a medida que avanzan y se priorizan para su desarrollo.

Como ejemplo puede mencionarse un sistema para un comercio minorista. Una primera versión podría contener solamente funciones para emitir las facturas y controlar la caja. Una segunda vuelta podría permitir registrar clientes y manejar la cuenta corriente de los mismos. Un tercer incremento habilitaría la posibilidad de registrar las compras y los proveedores y finalmente una última versión podría permitir un control de stock automatizado.



Ventajas	Desventajas
• Las entregas parciales reducen la complejidad del proyecto y mejoran las estimaciones. • El usuario recibe rápidamente una primera versión con sus primeras necesidades ya cubiertas. • Requiere menos personal concurrente y los recursos se asignan mejor en el tiempo. • Las implementaciones parciales son más sencillas que una implementación total.	• Sigue siendo un esquema secuencial, aunque esto sea menos grave que en el modelo anterior. • Los errores de las primeras etapas se siguen arrastran con facilidad a las siguientes. • Pueden producirse estados de bloqueos por la secuencialidad de las etapas, aunque menores porque se supone que las etapas son más acotadas.

8. Modelo de "Prototipos"

Ya se mencionó la dificultad existente para obtener requerimientos y para asegurar que los analistas entendieron lo mismo que los usuarios quisieron explicar. También la existencia de procesos técnicos complejos sobre los que se tienen dudas si se pueden realizar.

Es habitual que, al construir una casa, los arquitectos construyan maquetas y diseños simulados para que el cliente pueda tener una mejor idea de cómo será su hogar. De igual modo,

los ingenieros de software pueden construir un prototipo de lo que será el producto final. Este prototipo es una versión acotada del software, construida solamente a los fines de poder interactuar con el usuario y poder tener una mejor visión de que es lo que se está planificando hacer. Seguramente no todas las funciones, pantallas u opciones del software serán incluidas en el prototipo, que podrá ser parcial e incluir solo aquellas funciones más representativas.

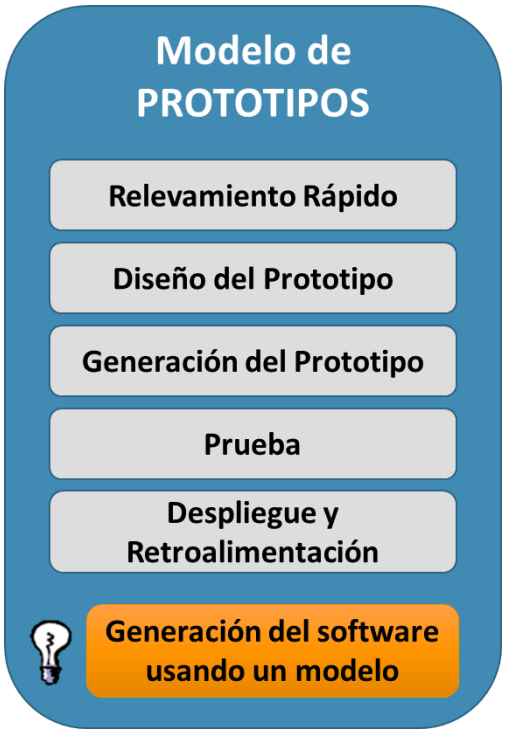
El desarrollador también podrá usar el prototipo para probar algún algoritmo determinado o alguna funcionalidad compleja del software. Por ejemplo, si se construye un software para interactuar y controlar algún dispositivo móvil, podría desarrollarse un prototipo para comprobar que en efecto puede accederse al dispositivo y que se pueden realizar las operaciones necesarias para controlarlo.

Lo habitual es que un prototipo sea un software simple, tambien puede usarse diseño de pantallas o imágenes gráficas del producto como tal. No se busca que el cliente pueda utilizar funcionalmente el prototipo, solo que pueda comprender y visualizar cómo funcionará la aplicación final.

El prototipo, incluso, puede tener varias versiones, cambios y ajustes. Podría trabajarse sobre el prototipo hasta que se finalmente acuerdan los requerimientos. También puede refinarse cuantas veces sea necesario hasta que efectivamente se logran resolverlos temas técnicos. Recién cuando se logra el acuerdo final sobre el prototipo se avanza en la construcción del software.

Es importante destacar que, una vez que el prototipo es aprobado y que los requerimientos han quedado finalmente claros**, este debe desecharse para iniciar la construcción del software real.** Para ello, podrá utilizarse cualquiera de los modelos de desarrollo, según las necesidades específicas del proyecto**.**

Las etapas del modelo de prototipos son similares a las del modelo lineal secuencial, aunque en este caso, no se busca la construcción de un producto final del software, sino solamente de una versión de prueba.



Etapa	Descripción
Relevamiento Rápido	Esta etapa se realiza un relevamiento general y luego de relevan con más detalle aquellas opciones que serán incluidas en el prototipo. Este relevamiento no se hace de un modo tan exhaustivo ya que no tiene como objetivo obtener todos los requerimientos, sino que, por el contrario, es una herramienta para mejorar ayudar a obtenerlos
Diseño del prototipo	Se diseña el prototipo, detallando el modo en el que se presentará y cuál será el alcance de cada una de las opciones de este.
Generación del prototipo	Se construye el prototipo. Puede utilizarse el mismo lenguaje de programación que luego se utilizará para programar el sistema, usar alguna herramienta específica de desarrollo de prototipos, o incluso mostrar ejemplos gráficos sin programación.
Prueba	Si el prototipo es un software operable, el mismo deberá ser probado para buscar posibles errores.
Despliegue y retroalimentación	**No se entrega el prototipo al cliente**, sino que el desarrollador lo ejecuta y analiza con él. A partir de este análisis podría modificarse el prototipo, construirse uno nuevo o, si ya quedó todo claro para ambos, comenzar con el desarrollo de la aplicación final.

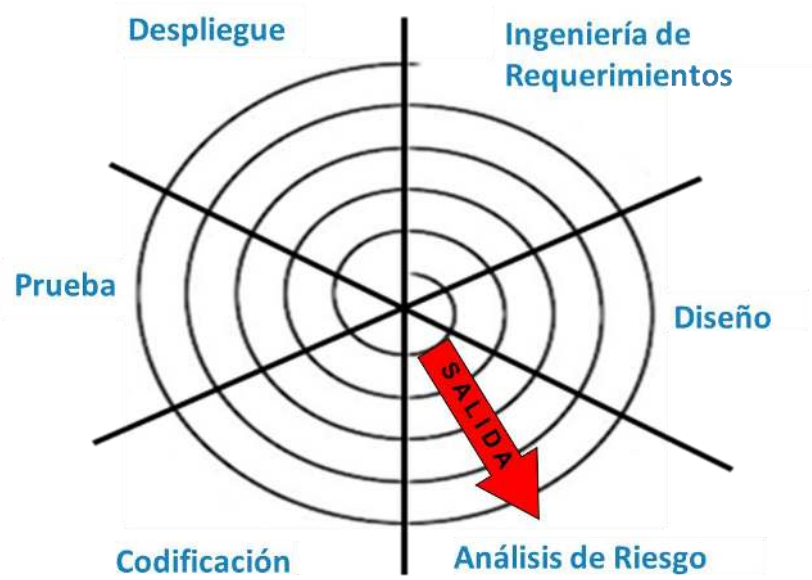
Ventajas	Desventajas
• Permite una mejor definición y comprensión de los requerimientos iniciales. • Permite un testeo de aquellas funciones técnicamente complejas. • Reduce la incertidumbre porque el cliente ve un modelo real de lo que será el futuro sistema.	• Si es un software, el cliente puede tentarse de utilizar el prototipo. Se debe tener en claro que el prototipo es construido a los solos efectos de mejorar el entendimiento entre usuario y desarrollador. **No es un producto que pueda utilizarse** bajo ningún punto de vista como una primera versión operativa. • El desarrollador puede tentarse de ajustar el prototipo y entregarlo como producto final, sin cumplir con ninguna pauta de calidad o seguridad en el desarrollo. • Algunas de las funcionalidades presentes en el prototipo podrían no ser tenidas en cuenta en la versión final, o quizá no sean posibles de construir en el entorno real de desarrollo.

9. Modelo en "Espiral"

Ya se ha destacado que, en escenarios riesgosos, no todos los proyectos que se inician terminan en un software que se implementa en forma exitosa. Boehm (1988) propuso un marco del proceso de software dirigido por el riesgo. Utilizó para representarlo gráficamente espira donde, cada ciclo, representa una fase de desarrollo[13].

[13] Puede consultarse punto 2.3.3 del libro "Ingeniería de Software" de Ian Sommerville Ed. 9, para una profundización del "Modelo en espiral de Boehm"

Se analiza en este apunte una versión resumida del Modelo en Espiral, presentada en libro "*Ingeniería de Software*" de *Roger Pressman*, y que resulta una buena elección situaciones de riesgo e incertidumbre. El gráfico del modelo es el siguiente:



Las primeras de estas iteraciones de la espiral podrían corresponder a una primera versión o bien al desarrollo de un prototipo. Luego aparecerán, de modo similar al modelo lineal secuencial iterativo, diferentes versiones operativas y más completas del software hasta que, finalmente, se completa el software.

La característica distintiva y saliente del modelo consiste en que agrega a las etapas habituales un análisis de riesgo. En cada una de estas iteraciones se evalúan los riesgos. A partir de dicho análisis, el proyecto puede ser modificado, acotado, puesto en pausa o, incluso, abortado sí las condiciones de riesgo evaluado así lo imponen.

Etapa	Descripción
Ingeniería de requerimientos	Como se explicó, en este modelo el sistema se va construyendo en etapas. Para ello, se establecen los requerimientos y necesidades generales de la organización y se definen que componentes se incluirán en la primera versión del software. La ingeniería de requerimientos se circunscribirá entonces a estas primeras funcionalidades. Se buscan determinar cuáles son los posibles riesgos que pueden determinar la no continuidad del proyecto.
Diseño	Se diseñan los componentes de la primera etapa del software.
Análisis de riesgo[14]	**Es la etapa distintiva del modelo**. En ella se analizan los riesgos que pueden afectar la continuidad del proyecto y se evalúa la conveniencia o no de continuar. A diferencia de otros modelos, esta etapa actúa como vía de escape. El desarrollo puede abortarse para no gastar recursos en proyectos inviables

[14] No debe confundirse esta etapa con la **Gestión de Riesgos** que se realiza para proteger la calidad del sistema. En este caso son riesgos que, de producirse, no tiene sentido continuar con desarrollo del sistema. Por ejemplo, un resultado adverso en las elecciones o la venta de la compañía.

Codificación	Se realiza la codificación con las primeras funcionalidades. Se producirá un software completamente funcional que luego se irá completando cuando se abarquen nuevas funcionalidades.
Prueba	Se prueba esta primera versión, buscando fallos y debilidad.
Despliegue	Se pone en funcionamiento la primera versión del sistema, la cual comprende las funcionalidades acordadas para esta etapa inicial. El cliente podrá utilizar el sistema en su totalidad. A medida que avancen las etapas posteriores del modelo, se irán añadiendo nuevas funcionalidades de forma gradual.

Ventajas	Desventajas
• Las entregas sucesivas y el análisis de riesgo lo vuelven un modelo apto para desarrollos en entornos riesgosos y con mucha incertidumbre. • Como otros modelos incrementales, puede utilizarse durante todo el ciclo de vida útil de un sistema, produciendo sucesivas versiones mejoradas del producto inicial.	• No es simple medir los riesgos, mucho menos cuando de ellos depende o no la continuidad del proyecto. Si el análisis no es preciso, podrían abortarse proyectos finalmente viables o, por el contrario, continuar aun cuando el riesgo finalmente se presente. • No siempre el cliente entiende que, a pesar de sus etapas y mejoras sucesivas, es un modelo controlado y que tiene con un final previamente planificado. • Puede resultar más caro que otros modelos ya que realizar un análisis de riesgos implica asumir costos, aunque también puede permitir ahorros si se tomas las medidas correctivas para mitigar el impacto del riesgo detectado.

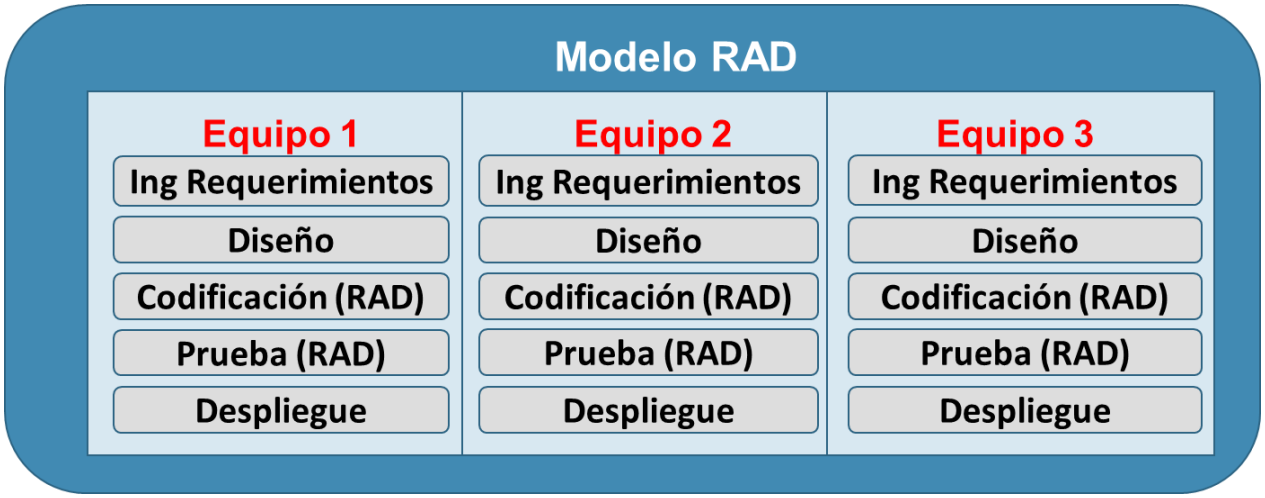
10. Modelo de "Desarrollo Rápido de Aplicaciones"

Los actuales entornos avanzados de desarrollo de aplicaciones proveen a los desarrolladores de poderosas herramientas que permiten generar código con gran rapidez y exactitud. El desarrollo visual; la generación automatizada de código en múltiples lenguajes; los asistentes; los componentes reusables o predefinidos; las herramientas colaborativas y las herramientas de prueba automatizada, entre otras, permiten generar aplicaciones robustas en espacios cortos de tiempo.

La utilización de estas herramientas da entonces origen a un modelo de desarrollo rápido de aplicaciones también conocido como RAD, por su sigla en inglés (Rapid Application Development). El propio nombre del modelo hace referencia la posibilidad de generar aplicaciones con mayor velocidad que los desarrollos tradicionales[15].

[15] Nuevamente no debe confundirse con desarrollos ágiles. También en este caso la planificación ocurre en el momento inicial. Incluso se puede pensar en un desarrollo rápido de una única versión de una aplicación, sin que esta tenga incrementos, cambios, ni mejoras posteriores.

Para agilizar los tiempos de desarrollo el software se divide en partes, para que puedan ser desarrolladas por varios equipos que trabajan en forma concurrente. Esta división, más la reutilización de componentes y las herramientas RAD permiten construir piezas de software completo y de calidad en tiempos que van de 30 a 90 días. La permanente comunicación con el cliente y entre los equipos de trabajo se vuelve imprescindible para el éxito final.



Etapa	Descripción
Ingeniería de requerimientos	Se relevan los requerimientos y necesidades de la organización. Se prevé la división del sistema para que sea desarrollado por los distintos equipos simultáneos.
Diseño	Se diseña en función de las herramientas RAD y usando los asistentes de esta. Se analizan los componentes que pueden ser reutilizados y aquellos que será necesario programar.
Codificación	**Se utiliza la herramienta RAD** para volcar en ella las características que tendrá cada uno de los componentes del sistema. El código se genera mediante un proceso semiautomático, pudiendo incluso utilizarse más de un lenguaje y/o generar para más de un entorno operativo.
Prueba	Además de la prueba tradicional, se utilizan las herramientas de prueba automatizadas provistas en la propia herramienta RAD para reducir los tiempos de la etapa.
Despliegue	Se procede a instalar el sistema en las computadoras del cliente, teniendo en cuenta las consideraciones generales de esta etapa.

Ventajas	Desventajas
• Los tiempos de desarrollo se acortan significativamente. • También se reducen los tiempos de prueba y los errores no detectados.	• El cliente debe también comprometerse con los plazos cortos de desarrollo y esto implica asignar mayor personal al proyecto y con una mayor carga horaria.

• Pueden construirse sistemas portables o de fácil migración entre diferentes entornos y sistemas operativos.	• No todos los proyectos pueden dividirse, ni son aptos de ser desarrollados con herramientas RAD • El código generado en forma automática suele tener menor performance y consumir más recursos. • Se complica su implementación en proyectos grandes porque requiere mucho personal en forma concurrente.

11. Otros modelos

Si bien los modelos descriptos anteriormente son los más utilizados en desarrollos vinculados a la administración y a las ciencias económicas, desde luego no agotan la totalidad de los modelos existentes. El análisis y desarrollo de otros modelos escapa a los alcances del presente trabajo, pero cabe de todos modos hacer una rápida mención de alguno de ellos, destacando algunas de sus características principales:

- **Desarrollo basado en componentes**: Este modelo centra su desarrollo en la reutilización de software y en el ensamble de componentes existentes. Si bien la reutilización no es exclusiva de este modelo, puede utilizarse un marco de trabajo especialmente orientado a trabajar reutilizando múltiples componentes preexistentes.
  Muchas empresas de desarrollo utilizan este modelo para software comercial: Poseen una aplicación parcialmente desarrollada, con las opciones genéricas ya programadas y la posibilidad de ajustar, mediante parametrización, los requerimientos específicos del cliente. Las ventajas de este tipo de desarrollo son[16]:

  - Implementación rápida de un sistema fiable.
  - Es posible analizar el software antes de implementarlo y ver si es adecuado para las necesidades de la organización.
  - Ahorro de costos.
  - Tiempos de desarrollo más cortos y con menos riesgo.
  - Se facilita la actualización, ya que el desarrollador suele actualizar las opciones generales para todos sus clientes.
  - Posibilidad de utilizar software como servicios. (Software en la nube)

- **Métodos formales**: Este modelo intenta hallar modelos matemáticos formales para asegurar un desarrollo de software libre de errores. Aun cuando difícilmente puedan encontrarse modelos matemáticos para todo tipo de software, puede ser interesante considerarse esta formalización cuando se requiera software con alto grado de precisión, por ejemplo, el desarrollado para el diagnóstico médico o para el manejo de vehículos

[16] Puede consultarse punto 16.6 del libro "Ingeniería de Software" de Ian Sommerville Ed. 9, Reutilización de productos COTS, para una mayor profundización de este tema

autónomos, aviones o grandes maquinarias. No es un modelo utilizado para el desarrollo de software comercial.

12. Elección de un modelo adecuado

Es importante tener en cuenta que no existe un modelo único que sea adecuado para todos los tipos de desarrollos, por lo tanto, no es posible realizar una recomendación definitiva sobre qué modelo utilizar. La elección del modelo adecuado depende de los factores destacados anteriormente, pero tambien de particularidades del proyecto, el tipo de cliente, la experiencia del equipo de desarrollo y su forma de trabajar. Todos estos elementos pueden influir en la elección final del modelo más apropiado para cada caso.

Además, es importante tener en cuenta que estos modelos no deben considerarse como caminos rígidos que se deben seguir de manera obligatoria. Un proyecto puede requerir el uso de características de varios modelos en su desarrollo. Por ejemplo, podría realizarse un prototipo y luego utilizar una herramienta RAD. O realizar un análisis de riesgo en alguna de las iteraciones del modelo secuencial incremental. Incluso puede utilizarse alguno de los modelos basados por un plan para construir la primera versión, y luego continuar con metodologías agiles para la etapa de evolución.

Incluso es factible que el desarrollador experimentado o la empresa de desarrollo armen su propio modelo, con las características que mejor se adapten a su forma de trabajar, al tipo de sistemas que realicen o al tipo de clientes para los cuales desarrollen.

De cualquier manera, es posible dar algunas recomendaciones genéricas, comparando los problemas habituales a los cuales se enfrentan los desarrollos, con las características propias de cada modelo:

Problema para resolver	Modelos recomendados
El tamaño	• Por su simplicidad, el modelo **lineal secuencial** presenta ventajas para sistemas de pequeño tamaño. • En el otro extremo, para proyectos de gran tamaño, la división y la construcción por etapas suele ser una buena estrategia. En este sentido, **los modelos "evolutivos"**, es decir aquellos que permiten construir versiones sucesivas e incrementales del software, parecen una buena elección para enfrentar este tipo de proyectos.
Complejidad de los requerimientos	• La elaboración de un **prototipo** ayuda en la comprensión de requerimientos. • Los **modelos evolutivos,** ya que evitan la necesidad de que todos los requerimientos deban ser explicitados al comienzo del relevamiento.
Complejidad técnica	• Puede utilizarse un **prototipo** para probar aquellas funciones técnicamente más complejas. • El modelo **en espiral** permite contemplar y monitorear riesgos técnicos conforme se avance con el desarrollo.

Tiempos disponibles	• El modelo de **desarrollo rápido** de aplicaciones permite reducir los tiempos de generación de aplicaciones.
Entornos riesgosos	• El **modelo en espiral** contempla situaciones de riesgo y la evaluación de impacto sobre el sistema.

13. Limitaciones

Si bien a lo largo del apunte se han mencionados diferentes modelos que se adecúan a diversas situaciones, existen escenarios, particularmente aquellos sujetos a cambios continuos, en los cuáles estos modelos se vuelven ineficaces. Entre las limitaciones de estos modelos podemos mencionar:

- Todos los requerimientos deben plantearse de antemano y no permiten incorporar cambios, ni aquellos producidos por la organización ni aquellos surgidos por el contexto.
- Los cronogramas son definidos de antemano, impidiendo que el usuario cambie las prioridades.
- Existe escasa participación del usuario en el proceso de desarrollo. El sistema finalmente entregado no siempre cumple sus expectativas y requiere una capacitación importante.
- El ciclo de desarrollo, aun en aquellos modelos evolutivos, resulta demasiado extenso para determinado tipo de organizaciones que se enfrentan a escenarios de cambio permanente.
- La documentación y los procesos formales no siempre son bienvenidos por todas las organizaciones, aunque definitivamente es un punto favorable para aquellas que deben cumplir con normativa externa, certificaciones, o que exijan tener documentados todos sus procesos y sus aplicaciones informáticas.

14. Bibliografía

IAN SOMMERVILLE: “Software Engineering”. 10ma edición. 2016. Pearson Education.
ROGER PRESSMAN: “Ingeniería del Software”. 7ta Edición. 2010. Ed. McGraw-Hill.

Apunte 3

Conceptos Fundamentales del Desarrollo Ágil de Software

1. Introducción

En la actualidad, muchas organizaciones operan en entornos altamente dinámicos que experimentan cambios constantes. Para mantenerse competitivas, estas organizaciones deben ser flexibles y capaces de adaptarse rápidamente a nuevos desafíos y oportunidades. Esto implica que tanto los procesos de producción de bienes y servicios como la estrategia de comercialización deben evolucionar y mejorarse de manera frecuente. En este contexto, resulta claro que el desarrollo de software también debe estar en sintonía con estos procesos cambiantes.

En situaciones como estas, es una ilusión pensar en una planificación inicial que se mantenga estable durante largos periodos de tiempo, ya sea meses o incluso años, que pueden abarcar un proceso de desarrollo. Surge entonces la necesidad de explorar nuevos modelos que permitan llevar a cabo proyectos de software exitosos en este tipo de escenarios.

En lugar de seguir un enfoque rígido y predecible, es necesario adoptar metodologías ágiles que fomenten la flexibilidad, la adaptabilidad y la entrega temprana de valor. Los métodos ágiles permiten responder de manera rápida a los cambios y ajustar el enfoque a medida que se van adquiriendo nuevos conocimientos y se comprenden mejor las necesidades del cliente.

Asimismo, es fundamental adoptar una mentalidad de mejora continua y aprendizaje. A través de ciclos iterativos y feedback constante, se puede adaptar el desarrollo de software a medida que se avanza y se obtienen nuevos aprendizajes. Esto permite maximizar el valor entregado al cliente y mantenerse en sintonía con los cambios del entorno.

Los modelos de desarrollo ágil se han desarrollado como una respuesta a los desafíos mencionados, permitiendo la producción rápida de software útil y de calidad en períodos de tiempo más cortos. Estos modelos se basan en la idea de que el software no se desarrolla como una única entidad completa, sino como una serie de incrementos que agregan gradualmente nuevas funcionalidades a una aplicación que ya está en funcionamiento, pero también en constante evolución.

El desarrollo ágil se lleva a cabo mediante equipos reducidos de profesionales altamente capacitados, que comparten inicialmente la visión general del producto o servicio que se pretende implementar. A partir de esta visión, se van realizando pequeños incrementos de acuerdo con las prioridades establecidas por el cliente. Estos incrementos se ejecutan en ciclos breves de desarrollo conocidos como iteraciones o sprints.

Cada iteración del ciclo de vida del desarrollo ágil incluye una serie de etapas, tales como planificación, análisis de requerimientos, diseño, codificación, revisión, implementación y documentación. Sin embargo, es importante destacar que estas tareas no se llevan a cabo con la misma formalidad que se requiere en los modelos tradicionales.

En lugar de seguir rigurosamente una secuencia lineal y rígida, los equipos ágiles adoptan un enfoque más flexible y adaptable. Las tareas se llevan a cabo de manera colaborativa y se prioriza la comunicación y la entrega temprana de valor. Las iteraciones permiten obtener rápidamente resultados tangibles y retroalimentación del cliente, lo que facilita la adaptación y mejora continua del producto.

Los requerimientos del cliente se recopilan y se les asignan prioridades, quedando en espera. En cada iteración, no es necesario que se agregue una gran cantidad de funcionalidades para justificar lo que en los enfoques tradicionales de desarrollo evolutivo podría considerarse una nueva versión. En cambio, se seleccionan las funcionalidades que pueden ser programadas e implementadas dentro de los cortos plazos del sprint. El objetivo es tener un producto constantemente operativo y en continua mejora.

Al concluir cada iteración, el equipo revisa nuevamente los requerimientos pendientes y toma decisiones sobre qué funcionalidades se incluirán en el siguiente sprint. Este ciclo iterativo se repite hasta que se determine que el producto no necesita más evolución. Durante esta evaluación periódica de los requerimientos, se tienen en cuenta factores como la prioridad de los elementos pendientes, la retroalimentación del cliente y otros criterios relevantes. El equipo busca maximizar el valor entregado en cada sprint y asegurarse de que el producto siga evolucionando de manera efectiva.

2. Modelos Tradicionales versus Ágiles

Es frecuente escuchar que los modelos tradicionales son considerados lentos, pesados y burocráticos, mientras que los modelos ágiles son percibidos como rápidos y livianos. Esta afirmación es válida en cierta medida, ya que los modelos tradicionales pueden resultar ineficientes en entornos que experimentan cambios constantes. Sin embargo, es importante señalar que no todas las organizaciones operan en escenarios dinámicos y que no todas las empresas tienen procesos internos que se orienten hacia la flexibilidad.

Existen situaciones en las que los desarrollos de software requieren un análisis exhaustivo del sistema y la entrega de un producto completo. Por ejemplo, los sistemas críticos que gestionan maquinaria o vehículos autónomos, así como los sistemas bancarios[17] que necesitan garantizar altos niveles de seguridad. En estos casos, es necesario realizar un análisis detallado y meticuloso del sistema, teniendo en cuenta los riesgos y las regulaciones aplicables.

Si bien los modelos ágiles son altamente efectivos en contextos cambiantes y permiten una entrega rápida y continua de software, no son la solución ideal en todos los escenarios. Cada enfoque tiene sus ventajas y desventajas, y es importante seleccionar el modelo más adecuado según las necesidades y características específicas de cada proyecto.

Podríamos entonces afirmar que ambos modelos tienen diferente utilidad y aplicación conforme el tipo de organización, el tipo de sistema y el escenario en los que se aplique. En todo caso los problemas pueden aparecer si se trata de usar metodologías tradicionales en escenarios cambiantes

[17] Actualmente, los bancos utilizan prioritariamente modelos ágiles para el desarrollo de sus aplicaciones. Sin embargo, estos modelos ágiles son reservados para sus Apps y sistemas de banca on-line, que justamente enfrentan escenarios cambiantes. A la hora de modificar sus sistemas centrales o “core”, que gestionan operaciones críticas y sensibles, como el procesamiento de transacciones, la seguridad y el cumplimiento normativo, los bancos suelen adoptar prácticas de desarrollo más tradicionales y rigurosas, que incluyen análisis exhaustivos, pruebas rigurosas y una mayor planificación. Esto se debe a la necesidad de garantizar la estabilidad y la seguridad de los sistemas centrales, así como el cumplimiento de regulaciones y estándares exigentes.

o intentar un desarrollo ágil en una empresa con procesos formales[18].A modo de resumen podríamos analizar el siguiente cuadro:

Modelos Tradicionales	*Modelos Ágiles*
Útiles para organizaciones que operan en entornos estables y con normas y procesos definidos. También para software con funcionalidades críticas o alta seguridad.	Útiles para organizaciones que operan en entornos cambiantes, con procesos flexibles y donde se privilegia tener un software siempre listo y con posibilidad de cambios.
Requisitos definidos al inicio.	No existe una especificación inicial detallada del sistema.
Sin posibilidad de cambios.	Los cambios son frecuentes conforme se avanza con el desarrollo.
El sistema se entrega en forma completa (o, en modelos incrementales, en sucesivas versiones completas).	El sistema está en siempre en desarrollo. Se incorporan permanentemente nuevas funcionalidades al que ya está funcionando.
El usuario no forma parte del equipo de desarrollo. Solo toma contacto con el sistema cuando lo recibe para las pruebas finales.	El usuario participa activamente durante todo el proceso de desarrollo y puede ser parte del equipo de desarrollo.
El usuario de adapta al sistema	El sistema de adapta al contexto

3. El "Manifiesto Ágil"

El "Manifiesto Ágil"[19]. es un documento fundamental que aborda técnicas y procesos para el desarrollo de software. Fue redactado en febrero de 2001 en la ciudad de Utah, Estados Unidos, por Kent Beck y otros 16 desarrolladores expertos en programación, quienes eran críticos de los enfoques tradicionales. En este manifiesto se establecen cuatro valores y doce principios que conforman la filosofía de los métodos ágiles. A partir de estos valores y principios, se han desarrollado diversos modelos de desarrollo ágil de software

Valores del manifiesto Ágil

Los autores del manifiesto destacan que "*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**

[18] Puede ampliarse esta comparación en el apunte "Conceptos Fundamentales del Desarrollo de Software Dirigido por un Plan". También el capítulo 3 del libro "Ingeniería de Software" 9na Ed. de Ian Sommerville tratan estos puntos con mayor profundidad.

[19] https://agilemanifesto.org/iso/es/manifesto.html

Se valora el recurso humano como el principal factor de éxito. Contar con un equipo de trabajo capacitado da mayor garantía de éxito por sobre priorizar herramientas y procesos rigurosos. Se buscan recursos humanos que provean:

- Alta capacidad técnica.
- Compromiso con los objetivos comunes.
- Habilidades para el trabajo en equipo.
- Capacidad para tomar decisiones.
- Capacidad de resolver problemas.
- Confianza y respeto mutuo.
- Autogestión y organización personal.

- **Software funcionando sobre documentación extensiva**
  Se respeta la importancia de la documentación como parte del proceso. Sin embargo, las metodologías ágiles hacen énfasis en que se deben producir los documentos estrictamente necesarios. Los mismos deben ser cortos y limitarse a lo fundamental.

- **Colaboración con el cliente sobre negociación contractual**
  El cliente es parte del equipo de trabajo. Es un ingrediente más en el camino al éxito en un proyecto de desarrollo de software. La participación del cliente debe ser constante, desde el comienzo hasta la culminación del proyecto, y su interacción con el equipo de desarrollo, de excelente calidad.

- **Respuesta ante el cambio sobre seguir un plan**
  En entornos cambiantes y por la dinámica de la tecnología, un proyecto de desarrollo de software se enfrenta a frecuentes modificaciones durante su ejecución. La planificación no debe ser estricta, puesto que hay muchas variables en juego, sino que debe ser flexible para poder adaptarse a las nuevas necesidades que puedan surgir.

Principios que rigen el desarrollo ágil

1. Nuestra principal prioridad es satisfacer al cliente a través de la entrega temprana y continua de software de valor.

2. Son bienvenidos los requisitos cambiantes, incluso si llegan tarde al desarrollo. Los procesos ágiles se doblegan al cambio como ventaja competitiva para el cliente.

3. Entregar con frecuencia software que funcione, en periodos de un par de semanas hasta un par de meses, con preferencia en los períodos breves.

4. Las personas del negocio y los desarrolladores deben trabajar juntos de forma cotidiana en el proyecto.

5. Construcción de proyectos en torno a individuos motivados, dándoles la oportunidad y el respaldo que necesitan, y procurándoles confianza para que realicen la tarea.

6. La forma más eficiente y efectiva de comunicar información de ida y vuelta dentro de un equipo de desarrollo es mediante la conversación cara a cara.

7. El software que funciona es la principal medida del progreso.

8. Los procesos ágiles promueven el desarrollo sostenido. Los patrocinadores, desarrolladores y usuarios deben mantener un ritmo constante de forma indefinida.

9. La atención continua a la excelencia técnica enaltece la agilidad.

10. Es esencial la simplicidad como arte de maximizar la cantidad de trabajo que se hace.

11. Las mejores arquitecturas, requisitos y diseños emergen de equipos que se autoorganizan.

12. En intervalos regulares, el equipo reflexiona sobre la forma de ser más efectivo y ajusta su conducta en consecuencia.

4. Programación Extrema

La programación extrema (Extreme Programming, XP) es un método ágil ampliamente adoptado que engloba un conjunto de prácticas de programación efectivas. Esta metodología se destaca por su enfoque en las liberaciones frecuentes del software, la mejora continua y la participación activa del cliente en el equipo de desarrollo. XP se centra en potenciar las relaciones interpersonales como un factor clave para el éxito en el desarrollo de software, promoviendo el trabajo en equipo, fomentando el aprendizaje de los desarrolladores y creando un ambiente de trabajo positivo. El término "Extreme Programming" fue acuñado por Kent Beck en el año 2000.

Valores de XP

Los valores en los cuales se afianza la programación extrema son:

- ***Simplicidad:*** La simplicidad es la base de XP. Se simplifica el diseño para agilizar el desarrollo y facilitar el entendimiento. Un diseño complejo del código, junto a sucesivas modificaciones por parte de diferentes desarrolladores, hace que la complejidad aumente. En ciertas ocasiones, incluso, se puede tornar necesaria la refactorización del código para hacerlo más simple.

- ***Comunicación:*** La comunicación es fundamental, dentro del equipo de trabajo y con el cliente. Crea un ambiente de cooperación y unidad, que ayuda a poder aplicar los demás valores. Al estar el cliente integrado, su opinión sobre el estado del proyecto se conoce en tiempo real.

- ***Retroalimentación:*** Realimentación concreta y frecuente del cliente, del equipo y de los usuarios finales da una mayor oportunidad de dirigir el esfuerzo eficientemente.

- ***Coraje:*** Muchas de las prácticas implican valentía. Una de ellas es programar para hoy y no para mañana. La valentía permite a los programadores sentirse cómodos con reconstruir su código cuando sea necesario. Otro ejemplo es saber cuándo desechar código obsoleto, sin

importar cuanto costo y esfuerzo llevó hacerlo. Valentía significa persistencia, un programador puede encontrarse estancado en un problema complejo por mucho tiempo y luego lo resolverá rápidamente solo si es persistente.

- ***Respeto:*** Si no hay respeto y aprecio entre el equipo, XP no se puede aplicar efectivamente.

Prácticas de XP

La programación extrema incluye una serie de prácticas las cuales reflejan los principios de los métodos ágiles:

- **Planeación del incremento:** Los requerimientos se registran en tarjetas de historia (story cards). Las historias que se van a incluir en la próxima liberación se determinan por el tiempo disponible y la prioridad relativa.

- **Liberaciones pequeñas:** Al principio, se desarrolla el conjunto mínimo de funcionalidad útil que ofrece valor para el negocio. Las liberaciones del sistema son frecuentes y agregan incrementalmente funcionalidad a la primera liberación.

- **Diseño simple:** Se realiza un diseño solo suficiente para cubrir aquellos requerimientos actuales.

- **Desarrollo de la primera prueba:** Las pruebas de unidad (para probar el funcionamiento del módulo de modo aislado del resto) son frecuentes y continuas. Se aconseja escribir el código de la prueba antes de la codificación de modo que sirva como un marco de referencia de prueba de unidad automatizada.

- **Refactorización:** Se espera que todos los desarrolladores refactoricen de manera continua el código, tan pronto como sea posible y se encuentren mejoras de éste. Lo anterior conserva el código simple y mantenible.

- **Programación en pares:** Los desarrolladores trabajan en pares, y cada uno comprueba el trabajo del otro; además, ofrecen apoyo para que se realice siempre un buen trabajo.

- **Propiedad colectiva:** Los desarrolladores en pares laboran en todas las áreas del sistema. Todos los programadores se responsabilizan por el código, de modo que no se generen “islas de experiencia”. Cualquiera puede cambiar cualquier función.

- **Integración continua:** Tan pronto como esté completa una tarea, se integra en todo el sistema. Después de tal integración, deben probarse todas las pruebas de unidad y de integración en el sistema.

- **Ritmo sustentable:** Grandes cantidades de tiempo extra no se consideran aceptables, pues el efecto neto de este tiempo libre con frecuencia es reducir la calidad del código y la productividad de término medio.

- **Cliente en sitio:** Un representante del usuario final del sistema (el cliente) tiene que disponer de tiempo completo para formar parte del equipo XP. En un proceso de programación extrema, el cliente es miembro del equipo de desarrollo y responsable de llevar los requerimientos del sistema al grupo para su implementación.

Roles en XP

- **Programador**: Es quien escribe las pruebas unitarias y produce el código del sistema. Debe existir una comunicación y coordinación adecuada entre los programadores y otros miembros del equipo.

- **Cliente**: Encargado de escribir las historias de usuario y las pruebas funcionales para validar su implementación. Además, es quien asigna la prioridad a las historias de usuario y decide cuáles se implementan en cada iteración centrándose en aportar mayor valor al negocio. El cliente es solo uno dentro del proyecto, pero puede corresponder a un interlocutor que está representando a varias personas que se verán afectadas por el sistema.

- **Encargado de pruebas** (Tester): Ayuda al cliente a diseñar las pruebas funcionales, de ejecutarlas y de brindar los resultados al resto del equipo.

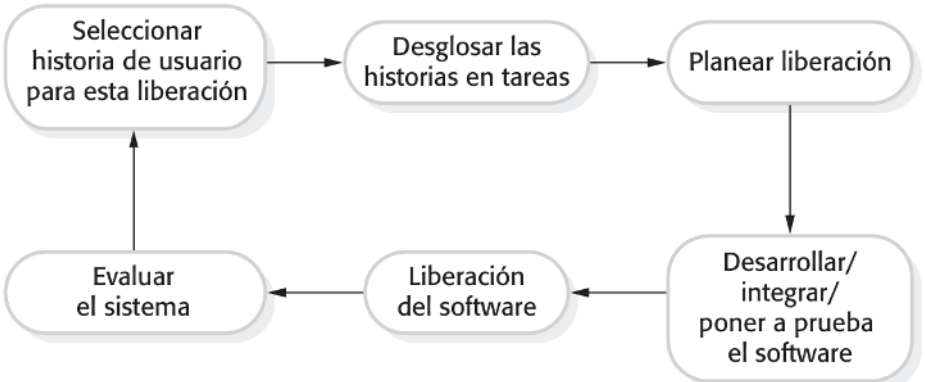
- **Encargado de seguimiento** (Tracker). Es el responsable de que se cumplan las estimaciones realizadas y de ver si los objetivos trazados en cada iteración fueron alcanzados.

- **Entrenador** (Coach). Es la persona que conoce a fondo el proceso XP y el responsable del proceso global. Debe proveer guía a todos los miembros del equipo para que el proceso sea satisfactorio.

- **Consultor**. Es un miembro externo del equipo con un conocimiento específico en algún tema necesario para el proyecto. Guía al equipo para resolver un problema específico.

- **Gestor** (Big Boss). Es quien coordina el vínculo entre clientes y programadores, el que ayuda a que las condiciones de trabajo sean las adecuadas.

Ciclo de desarrollo de la programación extrema



Los requerimientos de software se expresan en forma de escenarios, conocidos como historias de usuario. Los clientes colaboran estrechamente con el equipo de desarrollo para definir y priorizar estos requerimientos. Cada historia de usuario es clara y concreta y limitada, lo que permite a los programadores implementarla en un período corto de tiempo, generalmente unas pocas semanas.

Una vez definidas las historias de usuario, el equipo de desarrollo las desglosa en tareas y estima los recursos y esfuerzos necesarios para llevar a cabo cada una de ellas. Los programadores trabajan en parejas y crean pruebas para cada tarea antes de comenzar a escribir el código. Todas las pruebas deben ejecutarse satisfactoriamente al integrar el nuevo código en el sistema.

El cliente desempeña un papel activo en el desarrollo y es responsable de definir las pruebas necesarias para la aceptación final del software. El software se considera aceptado cuando todas las pruebas se ejecutan exitosamente. Este punto se convierte en la base para la siguiente iteración del sistema, dando continuidad al proceso de desarrollo.

Las historias de usuario

Queda fuera de alcance de este trabajo analizar en profundidad como es el relevamiento basado en historias de usuario, pero si se describirá en qué consisten estas historias a modo de poder comprender de modo general su funcionamiento.

Básicamente, una historia de usuario es un requerimiento que tiene un determinado usuario y que lo expresa, de modo simple, desde su perspectiva. Suelen tener el siguiente formato:

COMO <rol>
QUIERO <descripción de eventos que queremos que ocurra>
PARA <descripción de funcionalidad>

Ejemplo en un banco

COMO Oficial de atención al cliente
QUIERO Presionar una tecla y acceder a los productos que el cliente tiene contratados
PARA Ofrecerle nuevos productos

Si se decide que la historia es pertinente, se la estima en cuanto al esfuerzo que demandará realizarla y se priorizan para su realización. Cada vez que se termina un sprint se comienza a realizar que tenga mayor prioridad de las pendientes.

Las historias pueden completarse con alguna descripción mayor de la funcionalidad y también los criterios de aceptación. Por ejemplo: Primero deben mostrarse las tarjetas de débito, luego las de crédito y por ultimo los seguros.

Ocasionalmente, alguna historia debe ser particionada en tareas más sencillas. Por ejemplo, en el ejemplo, primero podría considerarse una historia para obtener el detalle de los productos que posee el cliente y, en una segunda historia, ordenar estos productos y mostrarlos en pantalla.

Pruebas en XP

Una fortaleza particular de la programación extrema, antes de crear una característica del programa, es el desarrollo de pruebas automatizadas.

Las características clave de poner a prueba en XP son:

- **Desarrollo de la primera prueba:** En lugar de escribir algún código y luego las pruebas para dicho código, las pruebas se elaboran antes de escribir el código. La prueba se corre a medida que se escribe el código, con el objetivo de descubrir errores durante el desarrollo.[20]

- **Desarrollo de pruebas incrementales a partir de escenarios:** El papel del cliente en este proceso es ayudar a desarrollar pruebas de aceptación para las historias, que deban implementarse en la siguiente liberación del sistema. En XP, la prueba de aceptación, como el desarrollo, es incremental.

- **Involucramiento del usuario en el desarrollo y la validación de pruebas:** El cliente que forma parte del equipo escribe pruebas conforme avanza el desarrollo. Por lo tanto, todo código nuevo se válida para garantizar que eso sea lo que necesita. Contar con el cliente para apoyar el desarrollo de pruebas de aceptación en ocasiones es una gran dificultad en el proceso de pruebas XP.

- **El uso de marcos de pruebas automatizadas:** La automatización de las pruebas es esencial para el desarrollo de la primera prueba. Estas se escriben como componentes ejecutables antes de implementar la tarea. Dichos componentes de pruebas deben ser independientes, simular el envío de la entrada a probar y verificar que el resultado cumple con la especificación de salida. Un marco de pruebas automatizadas es un sistema que facilita la escritura de pruebas realizables y envía una serie de pruebas para su ejecución.

Conforme se automatizan, siempre hay una serie de pruebas que se ejecutan rápida y fácilmente. Cada vez que se agregue cualquier funcionalidad al sistema, pueden correrse las pruebas y conocerse de inmediato los problemas que introduce el nuevo código.

Programación en pares

Otra práctica innovadora que se introdujo en XP es que los programadores trabajan en pares para desarrollar el software. Se requiere de dos programadores que participen en un esfuerzo combinado de desarrollo en un sitio de trabajo. Cada miembro realiza una acción que el otro no está haciendo actualmente, por ejemplo, mientras uno codifica, el otro se encarga de analizarlo para mejorarlo.

Los roles en esta técnica son el ***Conductor*** y el ***Acompañante*****.** El conductor es el que implementa el código y el acompañante tiene como tarea observar y corregir en todo momento los

[20] Puede consultarse el capítulo 3 del libro “ingeniería de Software” de Ian Sommerville para ver como más detalle y ejemplos sobre la primera prueba.

errores cometidos por el conductor, considerar soluciones alternativas y sugerir nuevos casos de prueba. Esta constante inspección produce código y un diseño de mayor calidad.

El uso de la programación en parejas tiene algunas ventajas:

- **Apoya la idea de la propiedad y responsabilidad colectivas para el sistema**. El software es propiedad del equipo como un todo y los individuos no son responsables por los problemas con el código. El equipo tiene responsabilidad colectiva para resolver dichos problemas.

- **Actúa como un proceso de revisión informal**, porque al menos dos personas observan cada línea de código. Las inspecciones y revisiones son muy eficientes para detectar un alto porcentaje de errores de software.

- **Ayuda a la refactorización,** que es un proceso de mejoramiento del software.

- **Enseñanza.** La programación por parejas realmente mejora la distribución de conocimiento entre el equipo de un modo sorprendentemente rápido.

- **Múltiples desarrolladores contribuyen al diseño.** Esto ayuda a crear mejores soluciones, especialmente cuando una pareja no puede resolver un problema difícil.

- **Mayor disciplina.** Se concentran en lo que tienen que hacer en lugar de tomar largos descansos.

Ventajas y desventajas de XP

Ventajas:

- Se adapta al desarrollo de sistemas pequeños y grandes.

- Optimiza el tiempo de desarrollo.

- Permite realizar el desarrollo del sistema en parejas para complementar los conocimientos.

- El código es sencillo y entendible, a pesar de la poca documentación a elaborar para el desarrollo del sistema.

Desventajas:

- No se tiene la definición del costo y el tiempo de desarrollo, nadie puede decir que el cliente no querrá una función más.

- Se necesita de la presencia constante del usuario, lo cual, en la realidad, es muy difícil de lograr dado que los usuarios tienen no pueden desatender sus propias tareas dentro de la organización.

- Algunos desarrolladores son celosos del código que escriben y no les es grato que alguien más modifique las funciones que realizó o que su código sea desechado por no cumplir con el estándar.

- Requieren mayor cantidad de programadores entrenados, que suelen ser costosos y difíciles de conseguir.

Para tener en cuenta

En la práctica, muchas compañías que adoptan XP pueden adaptar y personalizar las prácticas de programación extrema según sus necesidades y preferencias. Por ejemplo, algunas compañías encuentran beneficioso el enfoque de programación en pares, mientras que otras prefieren utilizar programación y revisiones individuales. Para adaptarse a diferentes niveles de habilidad, algunos programadores pueden optar por no realizar refactorizaciones en partes del sistema que no desarrollaron, y en lugar de historias de usuario, pueden utilizar requerimientos convencionales. Sin embargo, la mayoría de las compañías que adoptan una variante de XP siguen principios como liberaciones pequeñas, desarrollo basado en pruebas y la práctica de integración continua.

5. Scrum

El modelo Scrum es, quizá, el modelo más aplicado entre las organizaciones que utilizan desarrollo ágil. A diferencia de otros modelos, este no refiere solamente a un modelo de construcción de software, sino a metodología ágil enfocada en la gestión general de proyectos.

El proceso Scrum

El proceso de Scrum se compone de tres fases bien definidas. La primera fase es el **planeamiento del sprint** (“sprint planning”, en inglés), en la cual se establecen los objetivos generales del proyecto y se diseña la arquitectura de software. Esta etapa sienta las bases para el desarrollo posterior del sistema.

La segunda fase es conocida como los **Sprints**, que consisten en una serie de ciclos de trabajo. En cada sprint, el equipo se enfoca en desarrollar un incremento del sistema. Durante la planificación del sprint, se seleccionan las características o funcionalidades específicas a desarrollar y se realiza la implementación del software correspondiente. Al finalizar cada sprint, se entrega la funcionalidad completa a los participantes.

La tercera y última fase es la **Revisión del Sprint**, donde el equipo realiza una evaluación exhaustiva del sprint finalizado. Se identifican áreas de mejora y se definen acciones para el próximo ciclo. Este proceso de revisión y mejora continua asegura que el equipo se adapte y aprenda de cada iteración, optimizando así su desempeño.

Una vez concluida la revisión del sprint, el equipo reinicia el proceso, comenzando un nuevo ciclo con una nueva planificación del bosquejo. De esta manera, Scrum permite un enfoque

iterativo e incremental en el desarrollo de software, promoviendo la entrega continua de valor y la mejora constante del equipo

Las características clave de este proceso son las siguientes:

1. Los ***sprints*** tienen longitud fija, por lo general de dos a cuatro semanas. Una vez definida la duración ideal del sprint, en función de las características del desarrollo, la misma no debe cambiarse ya que, en cierto modo, pasan a conformar la medida del ritmo de trabajo.

2. El punto de partida para la planeación es el ***Product Backlog***, que es la lista de trabajo pendiente de realizar en el proyecto. Los nuevos requerimientos se van incorporando a esta lista. Al comenzar un sprint se revisan las tareas pendientes, se priorizan y se deciden cuál o cuáles de ellas serán desarrolladas en ese sprint.

3. La fase de selección de tareas incluye a todo el equipo del proyecto que trabaja con el cliente, con la finalidad de priorizar y elegir las características y la funcionalidad a desarrollar.

4. Una vez seleccionadas las tareas, el equipo se organiza para desarrollar el software. Con el objetivo de revisar el progreso y, si es necesario, volver a asignar prioridades al trabajo, se realizan **reuniones diarias** breves con todos los miembros del equipo. Estas reuniones diarias habitualmente se realizan sin sillas para que sean cortas y efectivas. Durante esta etapa, el equipo se aísla del cliente y la organización, y todas las comunicaciones se canalizan a través del llamado "**Scrum Master**". El papel de este último es proteger al equipo de desarrollo de distracciones externas y atender las necesidades que los miembros plantean en la reunión diaria.

5. La forma exacta en que el trabajo se realiza puede variar y depende del problema y del equipo.

6. Al final del sprint, el trabajo hecho se revisa y se presenta a los diferentes interesados, continuándose con el siguiente sprint.

Roles en Scrum

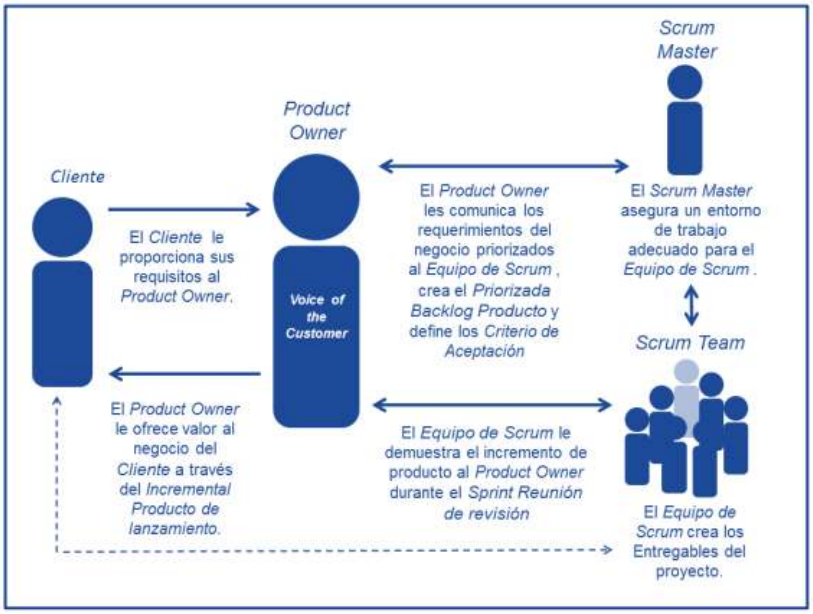
Los roles de Scrum se dividen en dos categorías:

1. **Roles comprometidos directamente con el proyecto:** Estos roles son los que obligatoriamente se requieren para producir el producto o del proyecto. Dichos roles son los responsables del éxito de cada sprint y del proyecto en sí:

    - ***Product Owner*** es la persona responsable de lograr el máximo valor empresarial para el proyecto. También es responsable de la articulación de requisitos del cliente. *El Product Owner* representa la voz del cliente dentro del proyecto.

    - **Equipo Scrum** es el grupo o equipo de personas responsables de la comprensión de los requisitos especificados por el *Product Owner*, y de la creación de los entregables del proyecto.

- **Scrum Master** es un facilitador que asegura que el equipo Scrum esté dotado de un ambiente propicio para completar el proyecto con éxito. El *Scrum Master* guía, facilita y les enseña las prácticas de Scrum a todos los involucrados en el proyecto; elimina los impedimentos que encuentra el equipo; y asegura que se estén siguiendo los procesos de Scrum.



2. **Otros Roles involucrados con el proyecto:** Son aquellos roles que, si bien son importantes, no participan directamente del proceso Scrum. No siempre están presentes y no son obligatorios, aunque en muchos proyectos desempeñan un papel muy importante y deben ser tenidos en especialmente en cuenta. Como ejemplo, podemos nombrar a los Stakeholder(s) (cliente, usuarios, accionistas).

Las ceremonias del Scrum

En base a lo anteriormente descripto cada Sprint estaría compuesto de **5 ceremonias** o eventos:

1. **Sprint Planning** o planificación inicial, con la participación de todo el equipo de desarrollo, es donde el *Product Owner* presenta la versión actualizada y priorizada del Backlog, para que el equipo pueda estimar que tareas podrán ser incluidas en el próximo sprint. Se define entonces **que** se va a hacer y **cómo** se realizará.

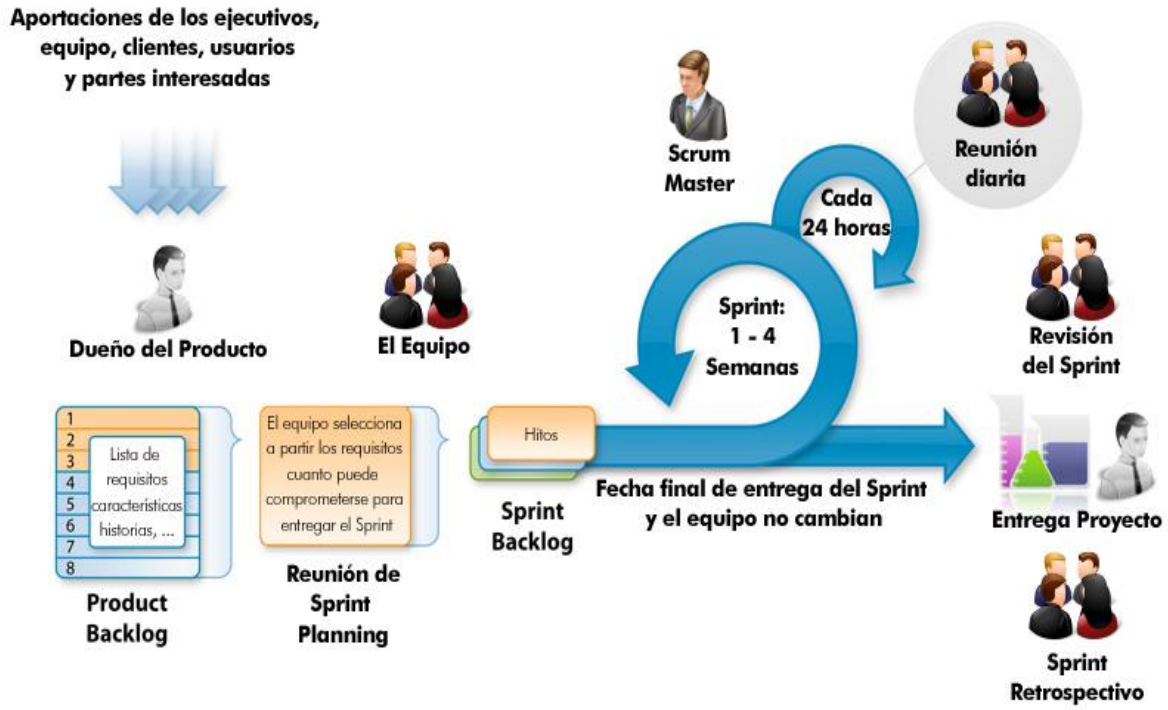
2. **Daily Meeting** o reuniones rápidas diarias, son encuentros breves para que cada miembro del equipo informe que tarea realizará ese día. También informará al Scrum Master si tiene algún impedimento para que éste pueda tratar de solucionarlo.

3. **Sprint Review**, es la reunión final al terminar el sprint, donde el *product owner* y el equipo de desarrollo presentan a los stakeholders el incremento terminado, para que lo inspeccionen y realicen las observaciones pertinentes. No se trata de una demo, sino que se

presenta el software funcionando en producción. De esta reunión pueden surgir nuevas necesidades que pasan a actualizar el Product Backlog.

4. **Sprint Retrospective**. Esta reunión, que puede realizarse en conjunto con lo anterior, consiste en reflexionar sobre el sprint que ha finalizado, identificando posibles cosas que puedan mejorarse. Es común analizar que salió bien, que ha fallado y que cosas podrían hacerse de un modo más eficiente.

5. **Refinamiento del Product Backlog.** Esta reunión adicional permite depurar el listado de pendiente y completar, refinar o aclarar ciertas historias de usuario que pudieron quedar pendientes durante el Sprint Planning.



El tablero Scrum

El tablero Scrum[21] es una herramienta muy útil en la metodología Scrum, ya que permite visualizar y gestionar el flujo de trabajo de manera efectiva. Cada columna del tablero representa una etapa del proceso, y las tarjetas representan las historias de usuario o tareas a realizar.

En un tablero Scrum, las tarjetas se mueven a través de diferentes columnas que representan las etapas del flujo de trabajo. Comúnmente, se utilizan las siguientes columnas: "Por hacer", "En progreso", "Testeo" y "Terminadas". Estas columnas reflejan el ciclo típico de trabajo en Scrum.

[21] Los tableros Scrum guardan similitud con los tablero Kanban, desarrollados por Toyota para administrar su producción en la década del 1940. Si bien fueron desarrollados en contextos diferentes (Scrum en el ámbito del desarrollo de software y Kanban en el ámbito de la producción), ambos enfoques se centran en la gestión ágil y promueven la transparencia, la colaboración y la adaptación.

En la columna "Por hacer", se encuentran las tareas que aún no han sido iniciadas. A medida que un miembro del equipo comienza a trabajar en una tarea, esta se mueve a la columna "En progreso". Una vez que la tarea está completa, se traslada a la columna "Testeo" para realizar las pruebas necesarias. Finalmente, cuando la tarea ha pasado exitosamente las pruebas, se coloca en la columna "Terminadas".

Es importante destacar que el tablero Scrum puede adaptarse a las necesidades específicas del equipo y el proyecto. Se pueden agregar columnas adicionales según las etapas o actividades que sean relevantes para el flujo de trabajo en particular. Por ejemplo, se pueden incluir columnas como "En revisión", "En espera" o "Bloqueadas" para identificar y abordar posibles cuellos de botella o problemas que requieren atención adicional.

La personalización del tablero Scrum permite que el equipo tenga una representación visual clara y específica de su flujo de trabajo, lo que facilita la identificación de áreas de mejora, la gestión de tareas y la resolución de problemas de manera más efectiva.

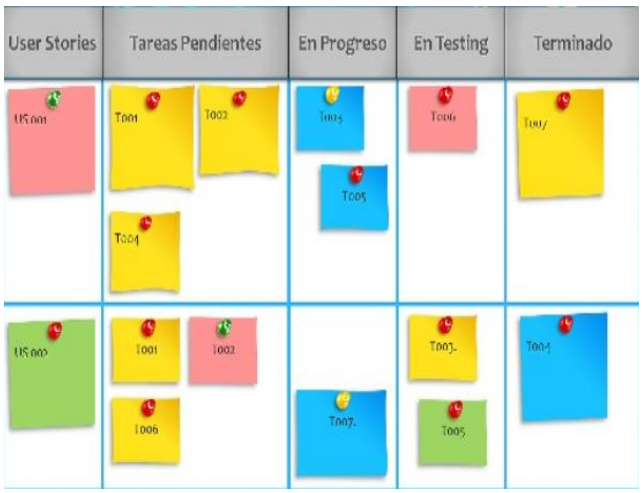
El tablero Scrum es una herramienta fundamental que proporciona una visualización clara del progreso de cada tarea y facilita la sincronización diaria en las reuniones del equipo Scrum. Permite identificar de manera rápida y sencilla las tareas que están en curso, las que están pendientes y las que ya han sido completadas. Esto ayuda a mantener a todos los miembros del equipo informados y al tanto del estado del proyecto. Esto facilita la colaboración y la toma de decisiones conjuntas, ya que todos tienen acceso a la misma información actualizada.

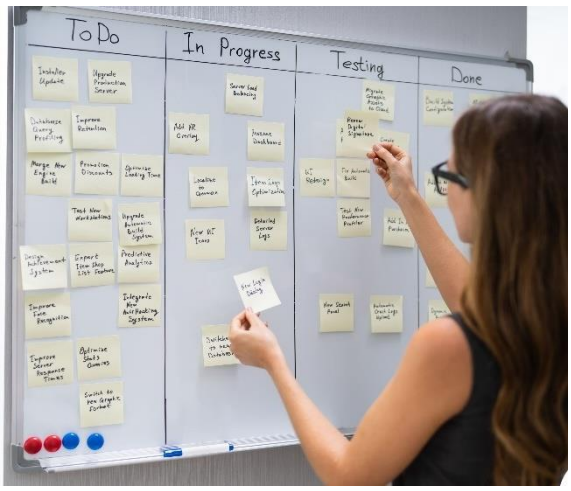
Además, el tablero Scrum ayuda a distribuir el trabajo de manera equitativa entre los miembros del equipo. Cuando una persona completa una tarea, puede pasar a ocuparse de la siguiente tarea pendiente, lo que ayuda a mantener un flujo constante y evita la acumulación de trabajo en una sola persona. Esto fomenta la colaboración y la capacidad del equipo para responder de manera ágil a los cambios y desafíos que puedan surgir durante el proyecto.

Es importante destacar que existen herramientas informáticas que permiten crear y gestionar tableros Scrum en línea[22], lo que facilita la colaboración y el seguimiento del progreso del proyecto por parte de todos los miembros del equipo, incluso si están trabajando de forma remota.

Ejemplo de un tablero Scrum:

[22] En las tiendas de aplicaciones, encontrar soluciones, para generar tableros Scrum/Kanban es muy sencillo. Existen diversas herramientas multiplataforma que permiten visualizar estos tableros tanto en la PC como en los celulares de los miembros del equipo, muchas de ellas gratuitas Una de las herramientas más reconocidas en este ámbito es Trello (disponible en https://trello.com/).





¿Por qué utilizar Scrum?

Algunas de las ventajas principales de la utilización de Scrum en cualquier proyecto son:

1. **Adaptabilidad:** las entregas iterativas hacen que los proyectos sean adaptables y abiertos a la incorporación de cambios.

2. **Transparencia:** por medio del Tablero de Scrum se visualiza el avance de cada Sprint y, al ser compartido, lleva a un ambiente de trabajo abierto.

3. **Retroalimentación continua:** se proporciona a través de las reuniones diarias y revisiones al finalizar cada sprint.

4. **Mejora continua:** los entregables se mejoran progresivamente sprint por sprint a través del proceso de priorización del *Product Backlog*.

5. **Entrega continua de valor:** los procesos iterativos permiten la entrega continua de valor tan frecuentemente como el cliente lo requiera.

6. **Ritmo sostenible:** los procesos scrum están diseñados de tal manera que las personas involucradas pueden trabajar a un paso cómodo (ritmo sostenible) que, en teoría, se puede continuar indefinidamente.

7. **Motivación:** los procesos de reuniones diarias y retrospectivas del Sprint conducen a mayores niveles de motivación entre los empleados.

8. **Resolución rápida de problemas:** la colaboración de equipos multifuncionales lleva a la resolución de problemas con mayor rapidez.

9. **Entregables efectivos:** los procesos de crear la lista de *Product Backlog* y revisiones periódicas después de la creación de entregables, asegura entregas efectivas para el Cliente.

Críticas a la metodología Scrum

- Complejidad de implementación: Scrum puede parecer simple en teoría, pero su implementación efectiva puede resultar desafiante. Requiere un cambio cultural y una adopción completa por parte del equipo y la organización, lo que puede llevar tiempo y esfuerzo significativos.

- El equipo puede sufrir estrés pues siempre estará de sprint en sprint, inmerso en un ritmo muy intenso de trabajo. Es saludable introducir, cada tanto, sprint que impliquen una tarea más relajada.

- Scrum se basa en equipos autoorganizados que toman decisiones colaborativas. Esto puede ser un desafío en entornos donde los miembros del equipo no tienen experiencia en autogestión o no tienen la capacidad de tomar decisiones conjuntas. Ocasionalmente, el equipo puede estar tentando a tomar el camino más corto para sumar los puntos del sprint, sacrificando calidad o seguridad.

- Esta metodología funciona bien en proyectos más simples y menos estructurados, pero puede tener dificultades para abordar proyectos complejos o con requisitos altamente detallados. En tales casos, otros marcos de trabajo pueden resultar más efectivos.

- Scrum se basa en la flexibilidad y la adaptabilidad, pero puede tener dificultades para gestionar proyectos con plazos fijos o fechas de entrega estrictas. Esto puede generar conflictos entre las expectativas del cliente y la capacidad de respuesta del equipo Scrum. Lo mismo ocurre con proyectos de precios cerrados por contrato. (ej. Licitaciones)

- Scrum se centra en la entrega de incrementos de trabajo en lugar de resultados finales. Esto puede dificultar la medición precisa del progreso y la evaluación del éxito del proyecto para las partes interesadas externas. Se requiere de un experto en la metodología que monitorice su cumplimiento.

- Si bien Scrum presupone un alto nivel de involucramiento y participación del cliente en el desarrollo, sus propias responsabilidades y obligaciones pueden dificultar mantener este nivel constante de involucramiento a lo largo del tiempo.

- Muchas reuniones, por cortas que sean, pueden ocasionar pérdidas de productividad.

Scrum, no solo para desarrollo de Software

Si bien es cierto que esta metodología nació para resolver problemas en el desarrollo de software, es importante destacar que no es exclusiva de ese ámbito. En la actualidad, los modelos ágiles de trabajo han trascendido el ámbito del software y las organizaciones los utilizan en diversos procesos. Si bien no se aplican todas las ceremonias, artefactos y roles, las adaptaciones que se realizan siguen la filosofía del modelo. Podemos mencionar los siguientes ejemplos:

- **Marketing**: Los equipos de marketing pueden aplicar metodologías ágiles para gestionar campañas publicitarias, lanzamientos de productos o estrategias de contenido. Pueden

utilizar tableros Kanban para visualizar las tareas, asignar prioridades y hacer un seguimiento de los plazos. También pueden realizar iteraciones rápidas y adaptar sus estrategias en función de los resultados y la retroalimentación obtenida.

- **Gestión de proyectos**: Las metodologías ágiles, son ampliamente utilizadas en la gestión de proyectos en diferentes industrias. Los equipos pueden dividir los proyectos en tareas más pequeñas, establecer plazos realistas y realizar entregas incrementales. Esto permite una mayor flexibilidad y adaptación a medida que se avanza en el proyecto.

- **Investigación y desarrollo**: Los equipos de investigación y desarrollo pueden beneficiarse de los modelos ágiles al abordar proyectos complejos. Pueden aplicar principios como la colaboración multidisciplinaria, la experimentación y la retroalimentación continua para agilizar el proceso de innovación y adaptarse rápidamente a los cambios en el entorno.

- **Planificación de eventos**: Los equipos encargados de organizar eventos, como conferencias o festivales, pueden utilizar una metodología ágil para gestionar las tareas, asignar responsabilidades y hacer un seguimiento del progreso. Pueden utilizar tableros Kanban para visualizar las etapas de planificación, asignar tarjetas a miembros del equipo y realizar un seguimiento de las tareas pendientes, en progreso y completadas.

- **Enseñanza**: Los educadores pueden aplicar los principios ágiles para organizar y llevar a cabo sus clases. Pueden utilizar metodologías como Scrum para planificar el contenido del curso, establecer objetivos y asignar actividades a los estudiantes. También pueden utilizar tableros visuales para gestionar el progreso de los estudiantes y adaptar el enfoque de enseñanza según las necesidades individuales.

- **Tareas del hogar:** En un entorno doméstico, las metodologías ágiles se pueden aplicar para organizar y distribuir las tareas del hogar de manera efectiva. Puedes utilizar tableros Kanban o aplicaciones de gestión de tareas para asignar responsabilidades a los miembros de la familia, establecer fechas límite y hacer un seguimiento del progreso de cada tarea.

Estos ejemplos demuestran que los modelos ágiles no se limitan al desarrollo de software, sino que pueden aplicarse de manera efectiva en una amplia variedad de contextos. La clave está en comprender los principios fundamentales de agilidad y adaptarlos de manera adecuada a cada situación específica, aprovechando las herramientas y técnicas ágiles disponibles, como los tableros Scrum y Kanban.

6. Otras metodologías ágiles

Si bien Scrum y XP son las metodologías más utilizadas y difundidas, desde luego no son las únicas. En todas estas metodologías los subyacen los principios del manifiesto ágil, tendiendo todas ellas una estructura similar. Queda fuera del alcance de este apunte ahondar sobre las mismas, pero si vale la pena al menos mencionar algunas de ellas

Metodología Crystal

Creada por Creada por Alistair Cockburn, uno de los firmantes del manifiesto ágil, se trata en verdad de un conjunto de metodologías que proponen alternativas para diferentes tamaños de equipos de trabajo y dificultad del proyecto, categorizándolos mediante una codificación de colores. Esta centradas en las personas que componen el equipo (de ellas depende el éxito del proyecto) y la reducción al máximo del número de artefactos producidos. El equipo de desarrollo es un factor clave, por lo que se deben invertir esfuerzos en mejorar sus habilidades y destrezas, así como tener políticas de trabajo en equipo definidas. Dada la vital importancia a las personas que componen el equipo de un proyecto, sus puntos de estudio son:

- Aspecto humano del equipo
- Tamaño de un equipo (número de componentes)
- Comunicación entre los componentes
- Espacio físico de trabajo adecuado
- Políticas claras para todo el equipo.

Los valores propuestos son concordantes con los principios ágiles:

- Entrega frecuente y constante de software operativo a intervalos que pueden ser diarios, semanales o mensuales.
- Comunicación permanente de todo el equipo de trabajo, preferentemente estando en el mismo espacio (comunicación osmótica)
- Reflexión y mejora continua. Es beneficioso dedicar un tiempo regular (ya sea unas pocas horas cada semana o una vez al mes) para reflexionar sobre el trabajo realizado, revisar notas, analizar el progreso y discutir posibles mejoras..
- Hablar con los compañeros cuando algo molesta dentro del grupo.
- Focalizarse en lo que se está haciendo, teniendo claro el objetivo y contando con el tiempo necesario para hacerlo.
- Tener fácil acceso a los usuarios y contar con desarrolladores expertos para consultas puntuales.

Método de Desarrollo de Sistemas Dinámicos

(en inglés Dynamic Systems Development Method o DSDM)

Este método de desarrollo surge en Inglaterra, en los años 90, y busca hacer frente a la problemática de desarrollar software en entornos con agendas y presupuestos apretados. Puede considerarse como una adaptación del modelo RAD a los principios ágiles y, de hecho, promueve la utilización de herramientas RAD.

Este framework de trabajo destaca 5 etapas, 3 de ellas previas a la construcción propiamente dicha:

1. Estudio de viabilidad y de negocio de los proyectos. Solo aquellos que puedan desarrollarse con un enfoque RAD pueden llevarse a cabo con esta metodología.
2. Estudio comercial o de la empresa. Se verifica que el sistema planificado pueda cumplir con los requisitos solicitados por la organización.

3. Iteración del modelo funcional, que implica construir un prototipo del sistema
4. Diseño e iteración de la estructura, donde se construye y prueba la solución definitiva.
5. Implementación.

Los principios de esta metodología, una vez más, están en línea con aquellos enumerados en el manifiesto ágil:

- Involucrar al cliente es la clave para llevar un proyecto eficiente y efectivo.
- El equipo del proyecto debe tener el poder para tomar decisiones que son importantes.
- DSDM se **centra en la entrega** frecuente de productos.
- El desarrollo es iterativo e incremental.
- Todos los cambios durante el desarrollo son reversibles.
- Las pruebas son realizadas durante todo el ciclo vital del proyecto.
- La comunicación y cooperación entre todas las partes interesadas.

7. Ventajas, dificultades y limitaciones de las metodologías ágiles

Las metodologías ágiles presentan ventajas a la hora de su utilización como también desventajas. A continuación, enumeramos algunas:

Ventajas

- Las metodologías ágiles ofrecen una rápida respuesta a cambios de requisitos a lo largo del desarrollo del proyecto, gracias a su proceso iterativo y a que el software está en un proceso de cambio permanente.

- El cliente puede observar cómo va avanzando el proyecto, y por supuesto, opinar sobre su evolución.

- El poder ofrecer entregables parciales permite ir atacando los objetivos más sencillos, ganado tiempo y experiencia tiempo para atacar luego los objetivos más complejos.

- Uniendo las dos anteriores se puede deducir que, al utilizar estas metodologías, los cambios que quiera realizar el cliente van a tener un menor impacto. Las entregas, intervalos cortos de tiempo contienen solo una porción del producto deseado. Si el cliente quiere cambiarlo nuevamente, solo se habrá perdido unas semanas de trabajo. Con las metodologías tradicionales las entregas al cliente se realizaban tras la realización de una gran parte del proyecto, eso quiere decir que el equipo ha estado trabajando meses para que luego un mínimo cambio que quiera realizar el cliente, conlleve la pérdida de todo ese trabajo.

- Se reducen los tiempos y costos de capacitación (el cliente participa del diseño y conoce el producto desde su desarrollo) e implementación.

- Se involucra desde un principio y se da un rol a todos los stakeholders.

Dificultades en la aplicación de los principios

- Aunque es atractiva la idea del involucramiento del cliente en el proceso de desarrollo, los representantes que asigna el cliente a menudo deben también seguir realizando sus tareas dentro de la organización. Tienen sus propios tiempos y presiones y no siempre pueden estar disposición de las necesidades del equipo de scrum.

- Quizás algunos miembros del equipo no cuenten con la personalidad adecuada para la participación intensa característica de los métodos ágiles y, en consecuencia, no podrán interactuar adecuadamente con los otros integrantes del equipo. Muchos desarrolladores desarrollan hábitos de trabajo en solitario y no siempre se consigue integrarlos eficazmente.

- Ocasionalmente, priorizar los cambios se torna muy difícil, sobre todo en sistemas donde existen muchos participantes. Cada uno puede tener diversas prioridades a diferentes cambios. Deben consensuarse, además, las necesidades operativas con las políticas.

- Mantener la simplicidad requiere trabajo adicional. Bajo la presión de fechas de entrega, es posible que los miembros del equipo carezcan de tiempo para realizar las simplificaciones deseables al sistema.

- Muchas organizaciones, especialmente las grandes compañías, pasan años cambiando su cultura, de tal modo que los procesos se definan y continúen. Para ellas, resulta difícil moverse hacia un modelo de trabajo donde los procesos sean informales y estén definidos por equipos de desarrollo.

Limitaciones

- Falta de documentación del diseño. Por lo general la única documentación con la que se cuenta son unos pocos comentarios en el código. Pueden producirse complicaciones con la reusabilidad de componentes a partir de esta falta.

- También la falta de procesos rigurosos y documentación escrita puede producir problemas si el software requiere ser certificado o auditado por equipos de contralor, los que seguramente exigirán constancias de las tareas realizadas en el proceso de desarrollo.

- Problemas derivados de la comunicación oral. No hace falta decir que algo que está escrito “no se puede borrar”, en cambio, algo dicho es muy fácil de olvidar o de tener ambigüedad. No siempre existe un entendimiento entre el usuario, que no entiende cuestiones técnicas, y los desarrolladores que carecen de conocimiento sobre cuestiones de negocios.

- El desarrollo no siempre tiene la misma secuencialidad ni necesidades que las prioridades fijadas. Puede ocurrir que para poder implementar alguna solicitud sea necesario que se programen funcionalidades extras o sea necesario implementar cosas que estaban previstas para más adelante.

- Restricciones en cuanto a tamaño de los proyectos, para el desarrollo de software de seguridad crítica, o para implementaciones dispersas geográficamente.

- Problemas derivados del fracaso de los proyectos ágiles. Si un proyecto ágil fracasa no hay documentación, o hay muy poca, para no volver a cometer los errores; lo mismo ocurre con el diseño. La comprensión del sistema se queda en las mentes de los desarrolladores.

- Los contratos por lo general son por tiempo insumido y no siempre fáciles de administrar ni de controlar.

- Son difíciles las subcontrataciones y la presupuestación.

8. Casos de estudio

En alguna de las actividades de la materia Ingeniería de Software se les solicita a los alumnos que expongan casos reales basados en su propia experiencia laboral. Resulta interesante analizar alguno de estos casos para ver cómo se aplican las metodologías ágiles en algunas empresas de nuestro entorno, en especial para ver cómo el aplicar la metodología en forma relajada casi siempre terminan en fracaso.

También queda reflejado como, en ocasiones, se aplica mal la metodología o directamente no se aplica. En ocasiones simplemente de dice que se usan metodologías ágiles porque suena "moderno", o quizá a los solos efectos de evitar alguna de las formalidades de los desarrollos basados en un plan. En estos casos los resultados suelen ser desalentadores.

Si bien se trata de relatos parciales y en muchos casos descontextualizados, sirven como base para ejemplificar los conceptos desarrollados en este trabajo. Con el permiso de los alumnos, aunque preservando su identidad y el nombre de la organización al que hacen referencia, estos son algunos ejemplos son valiosos para tener en cuenta[23]:

Caso A: Éxito en metodología Scrum en un banco

"Debido a la creciente necesidad del equipo de analistas de datos del banco de contar con una herramienta que cubra sus requerimientos de búsqueda de toda información contenida en el Data Warehouse y que sea de fácil escalabilidad para la utilización en el resto de las áreas, surge la idea de crear un equipo dedicado a la creación de un asistente virtual.

[23] Los alumnos tuvieron libertad para describir los casos según su criterio. Aunque se solicitaron casos reales, es importante destacar que no se presentan de manera rigurosa y completa, ni se basan en un análisis exhaustivo de las organizaciones mencionadas. Además, los alumnos tuvieron la posibilidad de agregar o omitir detalles para adaptar los casos a la actividad planteada. El análisis realizado se basa exclusivamente en la información proporcionada por los alumnos, y es importante tener en cuenta que podría variar si se contara con información adicional del contexto. Los casos de trascriben tal y como fueron presentados.

Para el desarrollo de este, se optó por una metodología ágil que permitiera la entrega de un producto iterativo e incremental que, al disponibilizarse para su uso, aportará valor al negocio de forma temprana. Se decidió realizar sprints con una duración de 2 semanas cada uno y entregas productivas (releases) cada 2 sprints, poniendo a disposición el producto para un grupo seleccionado de usuarios. Atravesando una profunda transformación, el banco decide formar un equipo integrado por Scrum Masters, facilitando uno de ellos para este proyecto. Además, se terminó de conformar el equipo con la contratación de un Product Owner, un Technical Owner y un equipo de desarrollo.

Durante los primeros sprints, se desarrolló una UI sencilla conectada a los servicios cognitivos. Esto permitió que el negocio pueda empezar a probar el asistente virtual y que se puedan ir realizando mejoras incrementales tanto en la interfaz, así como también, en los entrenamientos cognitivos. Además, cabe destacar que ayudó en el descubrimiento de necesidades futuras. En los incrementales siguientes, el asistente se conectó con diferentes servicios pasando a ser de esta manera un bot transaccional. Se agregó la analítica de datos para poder realizar explotación de estos y, actualmente, se está trabajando con sprints de seguridad y la incorporación de nuevas tecnologías."

Caso B: Después de mucho tiempo y esfuerzo, se logró el objetivo.

"Este es un caso de uno de los principales banco privados de Argentina, el cual tuvo varios problemas a la hora de hacer la transición desde el desarrollo de software con modelos basados en un plan hacia el uso de metodologías ágiles, durante aproximadamente dos años el banco tuvo inconvenientes con este cambio de paradigma de desarrollo, se utilizaron muchos recursos para cambiar la estructura del departamento de sistemas, y aun así no se consiguieron los resultados esperados, incluso pasaron varios scrum master por el sector sin poder lograr cambios significativos.

*Uno de los principales factores de porque se tardó mucho en cambiar el paradigma de desarrollo de forme eficiente, fue que gran parte del sector seguía muy aferrado al viejo modelo de desarrollo, es decir había un la falta de adaptabilidad por parte de los empleados y de los jefes, se insistía en se*guir *con el modelo tradicional de cascada, el cual era utilizado hasta ese momento, por otro lado, también influyó negativamente el no tener un plan de proyecto para realizar este tipo de cambios, así como un plan de contingencia para evitar un derroche de recursos innecesario, que fue lo que terminó ocurriendo.*

Por último, en términos de ingeniería de software, queda en claro que, para llevar a cabo este tipo de cambios de hacer las cosas, para imponer una nueva manera, no solo se requiere una voluntad de adaptabilidad por parte de los individuos, sino que también se necesita un plan que contenga al menos los principales riesgos posibles que puedan boicotear o ralentizar los objetivos deseados."

Caso C: Scrum en la industria automotriz.

"En este caso, hablamos de una empresa automotriz que subcontrata a empresas de IT para los desarrollos de sistemas. Tiene en agenda muchos cambios de sistemas para optimizar los procesos y hacer más dinámica la parte comercial. Si bien la mayoría de los sistemas antiguos han sido desarrollados con el modelo tradicional ahora intenta hacer esos cambios usando un desarrollo con metodologías agiles.

Esto dentro de un contexto de globalización donde la casa matriz francesa contrata, a través de su filial argentina, a una empresa local de IT para el desarrollo ágil de un sistema usado en Francia.
El software original utilizaba el lenguaje antiguo y era muy poco eficiente solo se conservaba por el costo y el riesgo de cambiar el sistema, hasta que el mismo software se volvió problemático y poco fiable. Dicho Sistema tenía como funcionalidad los pagos de primas a concesionarios y era clave para la gerencia comercial de la empresa.

Durante la migración de COBOL/DB2 a Java/Oracle hubo cambios frecuentes y nuevos pedidos por parte del usuario, cosa que no se hubieran podido procesar con el modelo tradicional donde se establecen los requerimientos al inicio sin posibilidad de cambio. Se incorporaron nuevas funcionalidades con las distintas liberaciones, se usó la metodología SCRUM donde se establecieron la longitud de SPRINT en cuatro semanas con la lista de trabajo (product backlog). Se hicieron pruebas en forma continua durante el desarrollo del sistema por medio del sector de QA antes de la salida a producción y sus correspondientes correcciones. El product owner de la empresa de IT fue clave para comunicar los requerimientos al scrum master.

Este caso muestra el éxito de como la ingeniería de software a través del uso correcto de metodologías ágiles, facilita los desarrollos y permite la migración de una plataforma vieja a una plataforma nueva, brindando una herramienta para lograr un software más escalable, funcional, eficiente, fiable y seguro.

A modo de conclusión, el desarrollo con metodologías ágiles del nuevo sistema de pago de primas a concesionarios fue el puntapié que hizo que la empresa de IT ganara varias licitaciones con esta multinacional francesa y logre afianzar su crecimiento dentro del mercado de empresas de IT."

Caso D: Problemas en metodología Scrum en una institución pública.

"La institución realiza los *desarrollos necesarios para poder llevar a cabo las comunicaciones y las integraciones para obtener los datos de las demás áreas de gobierno y así alimentar la base de datos logrando que las comunicaciones sean con datos actualizados. Actualmente desarrollan una plataforma de comunicación que centralice las diversas herramientas que se utilizan para la comunicación dentro de lo que es gobierno como Mailing, IVR (Respuesta de voz interactiva), audios, SMSs con los distintos proveedores contratados.*

En este caso se puede ver como por la incorrecta aplicación (desde el punto de vista conceptual) de metodologías ágiles, no se logró ningún beneficio y derivo tanto en pérdida de tiempo como en la falta de resolución de problemas. Utilizaron una especie de Scrum ya que lo adaptaron como les pareció útil. Por un lado, no se hacían las reuniones diarias, estas fueron intercambiadas por reuniones semanales. En las cuales capaz se planteaba el estatus del proyecto, pero no eran de utilidad ya que no se hablaba de los impedimentos para avanzar con el proyecto o lo que se iba a desarrollar la semana siguiente. Con los futuros usuarios no se hacia el relevamiento correcto de requerimientos lo que deriva en nuevas reuniones con los mismos para poder refinar las historias de usuario que en primera instancia estaban mal relevadas o relevadas de forma incompleta. Por otro lado, los product backlog

que se manejaban estaban mal armados, no estaban las tareas divididas en sprints como deberían.

La funcionalidad que se debía entregar en los releases presentaba anomalías debido a que por intentar cumplir con los sprints no se hacían las pruebas necesarias y se realizaban las entregas como estaban. No se llegaban con los tiempos por el retrabajo ocasionado por los errores de relevamiento y la mala organización de los tiempos donde no se les dio mucha importancia a las contingencias.

Caso E: Pérdida de tiempo y dinero en un banco.

"El proyecto constaba del desarrollo del Homebanking, plataforma donde se prestan los servicios bancarios a los titulares que registran una cuenta, (usuarios del banco), donde se puede acceder a través de internet por medio de computadoras, tablets o teléfonos celulares. El proyecto estaba organizado basado en un plan, con una combinación de metodologías Scrum. Todas las fechas ya se encontraban planificadas, determinadas, las cuales se tenían que cumplir al pie de la letra, (Por ejemplo, las salidas a producción).

En desarrollo los Sprints duraban 3 semanas, donde se pretendía llegar como sea a cumplir con las historias comprometidas en cada Prueba con los usuarios.

El problema fue que por mucho tiempo por falta de tiempo varios equipos dejaron de testear, hacer pruebas de software, y las incidencias que levantaban las personas de Testing no se les prestaba demasiada atención, nunca eran prioridad, esto se sostuvo durante varios meses. Llegó un día donde el producto que se estaba desarrollando tenía que salir a producción, cuando se empezó a llevar a cabo el proceso para poder realizar esto, todo exploto en los servidores, y afecto a otros módulos, debido a las fallas que se fueron acumulando y no se les había dado la atención necesaria.

Solucionar esas incidencias, y realizar las pruebas que tendrían que haberse hecho en su tiempo correspondiente produjo un enojo por parte de los PO, entre otras personas del banco. Se tuvo que modificar toda planificación del proyecto el cual se atrasó bastante tiempo, lo que a su vez hizo que se pierda una cantidad considerable de dinero."

Como vemos en estos dos últimos casos, ninguno aplicó correctamente la metodología Scrum, sino que hicieron adaptaciones y mezclas. Toda metodología puede ser adaptada conforme la organización y el tipo de proyecto, pero no puede ser desvirtuada al punto tal de perder su esencia.

En el "caso D" caso en lugar de aplicar correctamente Scrum hicieron una adaptación tan particular, que hasta las ceremonias fueron eliminadas, o en el mejor de los casos, adecuadas a como mejor le venía a quienes llevaron adelante el proyecto.

En el "caso E", se intentó aplicar una metodología ágil, como Scrum, a un proyecto que estaba rigurosamente planificado en términos de fechas y entregables. Es importante destacar que Scrum se recomienda para proyectos en los que existe la posibilidad de cambios y donde las fechas y entregables no pueden ser completamente definidos al inicio. En proyectos tan estrictamente

planificados como este, podría haber sido más adecuado utilizar un enfoque basado en un plan, que permite establecer un proceso bien definido y ajustado para cumplir con los plazos establecidos.

9. Comentario final

Como se ha visto, agilidad y desarrollo rápido no son sinónimo de un software construido a la ligera, con errores y sin procesos que aseguren la calidad del producto entregado. Programar software y entregarlo rápidamente, sin seguir ninguna regla ni metodología, no implica que se estén siguiendo modelos ágiles. Por el contrario, implican enormes riesgos y altas chances de fracaso. Es común que muchos desarrolladores afirmen utilizar metodologías ágiles por razones de moda o eficiencia, sin darse cuenta de que, si no las aplican correctamente, el éxito del proyecto queda a merced de la suerte.

Los modelos de desarrollo ágil tienden, como su nombre lo indica, a agilizar y acotar los procesos tradicionales. Sin embargo, existen metodologías y procesos para seguir. Lamentablemente, es frecuente encontrar organizaciones que dicen estar utilizando metodologías ágiles cuando en realidad no están aplicando metodología alguna.

En estos casos debemos recordar que:

- Metodologías ágiles no implica ausencia total de documentación.
- Metodología ágil no implica suprimir etapas del ciclo de vida, en especial las pruebas.
- Metodología ágil no significa hacer todo más rápido. Hacer un producto de lleva tiempo y la calidad no se negocia.
- Metodologías ágiles no implica dejar de tener necesidad de desarrollar actividades protectoras de la calidad y una adecuada gestión del proyecto.

10. Bibliografía y sitios consultados

IAN SOMMERVILLE: “Software Engineering”. 10ma edición. 2016. Pearson Education.
ROGER PRESSMAN: Ingeniería del Software, un enfoque práctico. 7ma Edición. 2010. McGraw-Hill.
CESAR A. BRIANO – GISELE AILIN BAUNALY – Desarrollo Ágil de Software
MANIFIESTO FOR AGILE SOFTWARE DEVELOPMENT http://agilemanifesto.org/
A Guide to the SCRUM BODY OF KNOWLEDGE (SBOKTM Guide). SCRUMstudy, 2013 Edition.
CEREMONIAS SCRUM https://www2.deloitte.com/es/es/pages/technology/articles/ceremonias-scrum.html

Apunte 4

Ingeniería de Requerimientos: Claves para un desarrollo exitoso

1. Introducción

Cuando se estudian las etapas del ciclo de vida del software, es común asociar la etapa de análisis con la recopilación de requisitos del sistema a través del relevamiento de los diferentes usuarios de la organización.

Generalmente, se examinan aspectos metodológicos de esta etapa, como la realización efectiva de entrevistas o la creación de casos de uso.

Sin embargo, a menudo se descuidan ciertos aspectos fundamentales, siendo el principal que los usuarios no siempre pueden proporcionar información precisa por diversas razones. Puede que no estén dispuestos a hacerlo, o si lo deseen, pero tengan dificultades para expresar adecuadamente sus necesidades. Además, algunos usuarios pueden preferir ocultar información relevante por temor a perder posiciones privilegiadas o, incluso, a que su trabajo sea reemplazado por el nuevo sistema. Como resultado, el relevamiento de información a través de los usuarios suele ser ineficiente en muchas ocasiones.

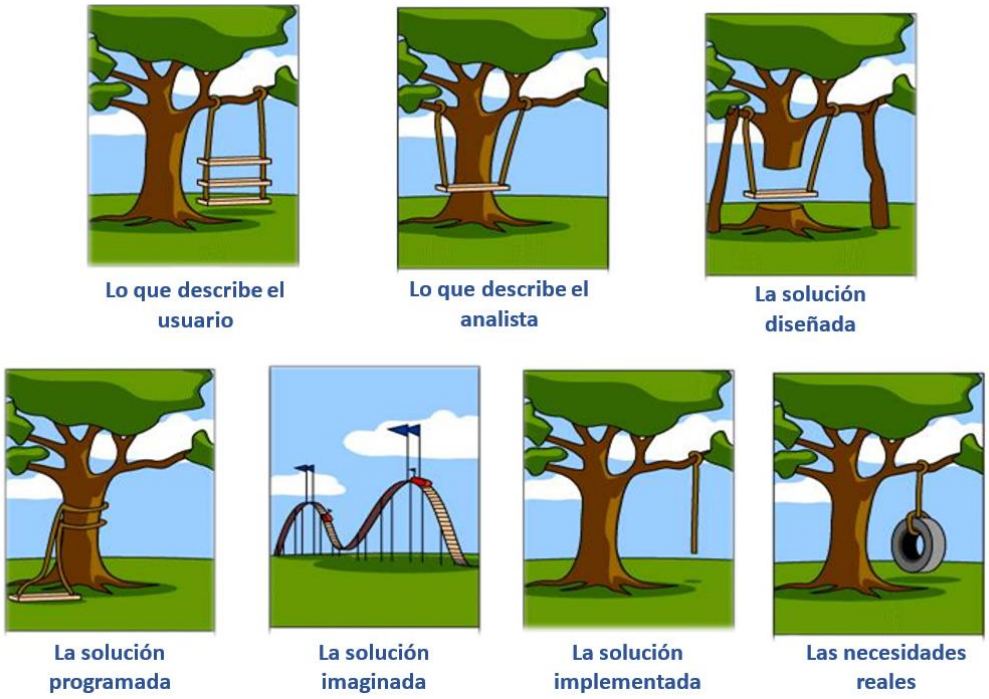
En los enfoques actuales de ingeniería de software, se ha dejado de utilizar el término "Análisis" para referirse a esta etapa, optando por el concepto de **"Ingeniería de Requerimientos"**. Sin embargo, este cambio va más allá de un simple ajuste terminológico. Representa una diferencia fundamental en la actitud requerida: ya no es suficiente esperar a que los requisitos se nos presenten, sino que debemos asumir la responsabilidad de construirlos. Debemos abandonar una actitud pasiva de solo registrar lo que el usuario nos comunica, y adoptar una postura proactiva en la cual seamos los impulsores de la construcción y validación de los requerimientos.

Pero aun cuando tomemos la etapa con esta concepción, seguimos encontrando serias diferencias ente lo que el usuario desea obtener y lo que finamente recibe. En este apunte se buscan mostrar algunas claves para producir relevamientos exitosos y contribuir a la construcción eficaz de requerimientos.

2. Los resultados no siempre son exitosos

En numerosas ocasiones, resultado final del software entregado no siempre es el que la organización esperaba. Ni siquiera es el que el usuario imaginó al momento de formular sus requerimientos.

Por diversos factores, hay una brecha grande entre realidad y expectativas, que se traduce, como mínimo, en una primera mala impresión. Comienza una búsqueda de responsabilidades y excusas. El usuario acusa al desarrollador de no haber hecho lo que le pidió. El analista se defiende y culpa al usuario de no haber explicado correctamente lo que quería. El programador aclara que, como no recibió los detalles necesarios, usó su criterio para definirá algunas funcionalidades. El líder del proyecto indica que se cumplió con lo pautado y firmado y que, eventualmente, es responsabilidad del cliente haber aceptado el diseño. Todos quieren despegarse del fracaso. La siguiente imagen meme, viral entre informáticos, que ralamente describen un escenario que, si bien puede resultar exagerado, no es tan diferente del que muchas veces ocurre en la realidad.



¿Quién o quiénes son los culpables de esta situación? Es importante reconocer que en esta situación suele haber múltiples factores causales y que las dinámicas organizacionales son complejas, con matices entre lo positivo y lo negativo. Sin embargo, más allá de las circunstancias y las dificultades inherentes, la principal responsabilidad recae en el profesional involucrado. Si bien el usuario puede tener conocimiento de su trabajo y comprender las limitaciones del sistema actual, no necesariamente está capacitado para elaborar especificaciones detalladas para el desarrollo de software. Por otro lado, el profesional de informática tiene la formación y experiencia para aplicar las buenas prácticas de esta etapa, como realizar un análisis completo que sirva como punto de partida para un desarrollo exitoso. A pesar de las posibles complicaciones, el profesional interviniente debe asumir la responsabilidad principal en la construcción de requisitos claros y precisos para el software.

Para ilustrar este punto, podemos establecer un paralelismo con el campo de la medicina. Si un paciente oculta algún síntoma o no lo expresa de manera adecuada, ¿de quién es la responsabilidad si se le administra un tratamiento incorrecto? En este caso, el médico es el profesional que entiende que no es suficiente con lo que el paciente le comunique verbalmente. Es probable que tome la temperatura, realice auscultaciones y solicite análisis más detallados. El médico comprende la importancia de recopilar toda la información necesaria antes de realizar un diagnóstico y no puede justificarse diciendo que el paciente no mencionó que, por ejemplo, también tenía dolor en la pierna.

De manera similar, en el ámbito de la informática, el profesional es quien posee el conocimiento y la experiencia para aplicar las mejores prácticas en el proceso de ingeniería de requerimientos. Reconoce que es su responsabilidad[24] y que debe obtener información completa y

[24] En varios países, y en algunas provincias de Argentina, el desarrollo de software es una profesión regulada, que solo puede ser ejercida por profesionales matriculados. Esto refuerza aún más el concepto sobre las implicancias y responsabilidades respecto de la generación de sistemas para organizaciones y empresas.

precisa, incluso si el usuario no puede proporcionarla de manera clara o consciente. Al igual que un médico, el profesional informático debe adoptar una actitud proactiva para construir y validar los requerimientos necesarios para el desarrollo exitoso del software

3. Requerimientos contrapuestos

Además de los desafíos mencionados anteriormente, existe otro problema común: ¿Qué ocurre cuando dos usuarios o áreas de una organización tienen visiones opuestas sobre cómo realizar una tarea? ¿Qué sucede si dos áreas responden de manera diferente ante una misma situación? ¿Y qué pasa si las dos áreas tienen necesidades distintas para una misma operación?

Al realizar el relevamiento de requerimientos, es necesario detectar estas ambigüedades y resolverlas. No se puede considerar válido únicamente lo que una persona, incluso si es el jefe, haya expresado. Es fundamental recopilar información de otros usuarios y áreas para asegurarse de que todos estén de acuerdo. También se deben revisar los circuitos administrativos y las normas establecidas para garantizar su cumplimiento. Finalmente, si se identifican inconsistencias, es necesario convocar a todas las partes involucradas y tomar decisiones sobre el camino que se seguirá en cada situación.

Es importante destacar que muchos sistemas existen para establecer orden en las organizaciones y evitar ambigüedades. Al contener las reglas de negocio y los circuitos formales, previenen que cada persona opere de manera arbitraria y aseguran que no se tomen diferentes enfoques de solución frente a un mismo problema.

Una de las situaciones que con frecuencia presenta dificultades es la decisión sobre qué datos se deben registrar en una operación. Algunos de estos datos pueden no ser relevantes para el usuario que realiza la carga inicial, pero podrían ser necesarios más adelante. Sin embargo, en ocasiones, no se dispone de acceso inmediato a estos datos o su carga puede demorar o aumentar la carga de trabajo del usuario. Por tanto, es necesario llegar a un consenso sobre qué datos se incluirán en una pantalla, cuáles podrían ser eliminados debido a su falta de utilidad posterior, y también qué datos podrían ser dejados pendientes para su carga en una etapa posterior.

En la actualidad, se cuenta con la posibilidad de acceder automáticamente a datos almacenados en sistemas de terceros (Afip, Renaper, Correos, etc.), mediante el uso de APIs[25]. Esto permite evitar la necesidad de ingresar manualmente dichos datos.

Un ejemplo concreto de esta situación es el despacho de combustible en las estaciones de servicio. El playero busca un sistema lo más simple posible: seleccionar el surtidor, verificar el importe, pasar la tarjeta, cobrar y continuar con el siguiente auto. Sin embargo, la estación de servicio tiene otros requisitos y necesidades adicionales. Es probable que desee identificar al cliente, tener acceso a información impositiva para emitir tickets con IVA discriminado y tal vez requiera la

[25] La API (*application programming interface*) permiten que dos piezas de software se comuniquen entre sí. Usando una API se pueden obtener datos de un servidor externo, sin que esto requiera necesariamente acceder a los sistemas u otros datos que operan dicho servidor. Por ejemplo, mediante una API se puede obtener acceso al mapa de Google de una ubicación determinada e insertarlo en mi página web, de modo gratuito y sin necesidad de un desarrollo complejo.

información de la patente del automóvil en caso de cuentas corrientes. Además, la petrolera puede tener sus propios requisitos, como tarjetas de puntos o descuentos.

Como resultado, la pantalla de carga ya no contendrá únicamente los datos mínimos que el playero indicó que necesitaba inicialmente. Se requerirá diseñar un sistema que pueda abordar las necesidades de la empresa sin sobrecargar innecesariamente la tarea del empleado que realiza el despacho de combustible, especialmente en los casos en los que se realiza el pago en efectivo sin puntos o descuentos.

En resumen, se debe buscar un equilibrio entre satisfacer las necesidades de información de la empresa y mantener la simplicidad y eficiencia en el proceso de despacho de combustible[26].

4. Tres conceptos clave sobre relevamiento

1) **La comprensión precisa de los requerimientos es esencial, ya que estos forman la base de la solución informática que se va a desarrollar**. Aunque a primera vista pueda parecer una tarea sencilla, existen varios factores que la convierten en uno de los desafíos más complejos de la ingeniería de software.

   En primer lugar, los requerimientos suelen ser expresados por personas con conocimientos técnicos limitados, lo que puede dificultar la comunicación efectiva y la expresión clara de las necesidades. Además, los requerimientos pueden ser ambiguos, contradictorios o incompletos, lo que genera incertidumbre sobre la verdadera intención del usuario.

   Otro factor complicado es que los requerimientos pueden cambiar a lo largo del tiempo debido a la evolución de las necesidades del usuario, avances tecnológicos o cambios en el entorno empresarial. Esto requiere una gestión adecuada del cambio y la capacidad de adaptación por parte de los desarrolladores.

   Además, la comprensión de los requerimientos implica considerar no solo las necesidades funcionales, sino también los aspectos no funcionales, como el rendimiento, la seguridad, la usabilidad y la escalabilidad del sistema. Estos aspectos deben ser tenidos en cuenta y equilibrados para lograr una solución informática exitosa.

2) **Existe la tentación de comenzar a construir el software incluso antes de que el problema haya sido completamente analizado y resuelto, quizás con la esperanza de que las cosas se aclaren durante el proceso de programación**. Sin embargo, esta práctica rara vez conduce a mejoras significativas. En cambio, es probable que surjan más dudas y problemas, y que

[26] Las nuevas tecnologías brindan oportunidades para agilizar este tipo de operaciones en los puntos de venta. Se pueden incorporar dispositivos de captura rápida que permiten una mayor eficiencia. Por ejemplo, los pagos a través de códigos QR ofrecen una forma rápida y conveniente de realizar transacciones. Además, el uso de aplicaciones móviles que integran todos los datos del cliente permite aplicar promociones y servicios adicionales de manera automática. También se pueden considerar opciones como la carga automatizada y sin intervención de personal, utilizando tecnologías de detección de patentes de vehículos

los riesgos de tener que rehacer el código aumenten considerablemente, lo que resulta en mayores costos financieros y tiempos de desarrollo.

Si bien es cierto que la formalización utilizada al codificar el software en un lenguaje de programación puede ayudar en algunos casos a resolver ambigüedades del lenguaje natural, no se puede considerar la construcción del software como una herramienta sistemática para mejorar los procesos de relevamiento.

3) **L as tareas de esta etapa deben ajustarse a las necesidades particulares de cada proyecto**. Los sistemas son diversos y los usuarios también, lo que implica que las estrategias utilizadas pueden variar. En algunos casos, se opta por un enfoque más breve, mientras que en otros se lleva a cabo con mayor rigor. En situaciones específicas, puede ser necesario buscar fuentes alternativas que respalden o mejoren la información proporcionada por los usuarios, como normativas o manuales de procedimiento.

5. El proceso de obtención de requerimientos

Precisamente el objetivo de este apunte es analizar la problemática de esta etapa, más allá del habitual alcance de la bibliografía que, muchas veces, se limita a describir cómo es proceso de relevamiento a los diferentes usuarios interesados. A modo de resumen y entendimiento general del tema, vale analizar en que consiste formalmente el proceso, utilizando para ello el libro "Ingeniería de Software" de Ian Somerville. Las actividades del proceso son:

1. **Descubrimiento de requerimientos:** Éste es el proceso de interactuar con los participantes del sistema para descubrir sus requerimientos. Existen numerosas técnicas complementarias que pueden usarse para el descubrimiento de requerimientos, como por ejemplo entrevistas, cuestionarios, descripción de escenarios o casos de uso.

2. **Clasificación y organización de requerimientos:** Esta actividad toma la compilación no estructurada de requerimientos, agrupa requerimientos relacionados y los organiza en grupos coherentes. En la práctica, la ingeniería de requerimientos y el diseño arquitectónico no son actividades separadas completamente.

3. **Priorización y negociación de requerimientos:** Inevitablemente, cuando intervienen diversos participantes, los requerimientos entrarán en conflicto. Esta actividad se preocupa por priorizar los requerimientos, así como por encontrar y resolver conflictos de requerimientos mediante la negociación. Por lo general, los participantes tienen que reunirse para resolver las diferencias y estar de acuerdo con el compromiso de los requerimientos.

4. **Especificación de requerimientos:** Los requerimientos se documentan y se formalizan en documentos de requerimientos. Estos documentos deben ser validados, aprobados y quedan listos para la siguiente etapa.

Por supuesto, y como queda de manifiesto en el gráfico ilustrativo del proceso, esta es una etapa cíclica, donde los se van trabajando y refinando hasta lograr el suficiente nivel de comprensión y detalle para poder continuar con las próximas etapas de construcción

6. Las situaciones habituales que dificultan el relevamiento

Como ya se insinuó anteriormente, no todos los requerimientos aparecen fácilmente en una entrevista o en cuestionarios que se les reparten a los usuarios. Por el contrario, existe una serie de situaciones que dificultan la tarea. Entre ellos destacamos:

a. Los usuarios [27] no tienen claro lo que realmente necesitan o no saben expresarlo.

b. Los usuarios, por diversos motivos, omiten requisitos u ocultan información, aun sabiendo de qué misma clave en el desarrollo de del sistema.

c. Los usuarios tienen temor que el nuevo sistema reemplace su trabajo o les haga perder poder relativo dentro de la organización. Los sistemas socializan y hace publica información que antes manejaba una sola persona.

d. Los usuarios no se involucran en la elaboración de requerimientos cuando estos formales o escritos, por temor a que esta información se haga pública o llegue a sus jefes. Ocasionalmente, pueden modifican lo expresado verbalmente cuando deben formalizarlo.

[27] A los efectos del presente apunte, el término usuarios debe considerarse en un sentido amplio, es decir que no solo se refiere a las personas que efectivamente utilizarán el sistema, sino también a todos aquellos interesados de algún modo en el sistema. El término en ingles ***Stakeholder*** podría ser más apropiado en este caso.

e. Los usuarios insisten en nuevos requisitos después de que el relevamiento ha finalizado. A veces realmente aparecen cosas nuevas, otras simplemente, las omitieron u olvidaron de expresar en su momento.

f. Los usuarios no comprenden los problemas técnicos ni entienden el proceso del análisis o desarrollo de software. Por supuesto no tienen por qué entenderlo.

g. Tanto los usuarios como los analistas usan terminología ambigua o que no coinciden con el sentido que tiene en un determinado contexto. Manejan un vocabulario propio de su tarea o profesión, que no siempre es comprendido por el otro.

h. Muchas veces, los relevamientos son parciales, conforme la facilidad o no de obtener datos. No se documentan los temas conforme su complejidad, sino en base a que tan fácil resulta relevar y obtener requisitos. Algunas áreas, donde los usuarios resultan más predispuestos, son relevadas "a fondo", mientras de otras, apenas se consiguen unos pocos comentarios.

i. Se intenta de "encajar" los requerimientos del sistema nuevo dentro de modelos ya existentes o en desarrollos anteriores. Se busca convencer al usuario de que utilice un esquema previamente desarrollado, aunque dicho esquema no sea el mejor para este caso.

j. Las ideas o los pedidos de los usuarios no siempre pueden ser llevadas a la práctica con la tecnología existente (o al menos dentro de los costos y plazos del proyecto).

k. Las ideas de los diseñadores no siempre son validadas por los usuarios. Muchas veces buscan la solución más simple desde el punto de vista técnico y no desde las reales necesidades del usuario.

l. No es fácil encontrar los usuarios claves, es decir aquellos que realmente puedan proporcionar información de calidad respecto del software que se busca desarrollar. Frecuentemente es la propia organización quien suele indicarnos a quien relevar, aunque estos usuarios no sean los que mejor conozcan las necesidades del proyecto.

7. Otros problemas habitualmente no considerados

A los inconvenientes mencionados anteriormente debemos agregar una serie de temas subyacentes, que habitualmente no son considerados al describir la etapa, pero que sin duda la afectan significativamente y, por lo tanto, es importante tenerlos en cuenta:

- Ocasionalmente, la propia organización desconoce realmente cuáles son sus problemas, las causas de ellos y/o sus necesidades reales. Nos contratan para desarrollar un sistema de un determinado tipo, cuando sus verdaderos problemas son otros. En extremo, pueden aparecer problemas de procesos o de relaciones de poder, que no se resuelven con un sistema informático.

- Cuando surgen nuevos requerimientos durante el desarrollo, la organización presiona para que sean incluidos en sistema que se está desarrollando, a veces respetando precios acordados y plazos de entrega. En paralelo, los desarrolladores buscarán dejarlos de lado y apegarse exclusivamente a lo acordado inicialmente. Cualquiera de los dos escenarios es problemático.

- Eu ocasiones, Los jefes del área son los que toman a su cargo la tarea de pensar en un nuevo sistema y también suelen ser ellos los que definen los requerimientos. Estas definiciones no siempre reflejan el verdadero modo que se hacen los empleados.

- Muchas veces se presenta documentación y procesos formales, buscando que sean volcados en la nueva solución informática. Esos procesos pueden estar desactualizados o, incluso, las tareas pueden realizarse con circuitos informales que terminan siendo la forma real en la que opera la organización (y la que debería replicar el sistema).

- Los jefes, aunque no lo digan abiertamente, pueden no querer implementar el nuevo sistema, en especial cuando les viene impuesto por otras áreas de la organización o les restringe su poder.

- Si los programadores tienen dudas, estas no son elevadas nuevamente a los usuarios correspondientes para que las respondan. Por el contrario, suelen son contestadas por los analistas. Esta respuesta no siempre coincide con las necesidades reales.

- No se tiene en cuenta que los usuarios son personas. Reaccionan de modo muy diferente ante situaciones en las que se ven analizados. Pueden sentir que se está evaluando el modo en el que realizan su trabajo y, por lo tanto, modificar sus respuestas. Incluso reaccionan de manera diferente en formales donde están sus jefes, en entrevistas individuales o en encuentros informales entre pares.

8. Escenarios hostiles

Los problemas mencionados anteriormente tienen puntos extremos que, en ocasiones, terminan transformando el escenario en un ambiente hostil. En este tipo de escenarios el relevamiento tradicional a usuarios se vuelve ineficaz, muy difícil o incluso imposible. Algunos ejemplos de escenarios hostiles son:

- El sistema viene impuesto “desde arriba”. El área o los usuarios están en contra de su implementación y harán cuanto esté a su alcance para que el relevamiento, y su posterior puesta en marcha, fracase.

- El área interna de sistemas se opone a la instalación de un software ajeno. Aun cuando ellos no tengan capacidad operativa para desarrollarlo, la idea de tener gente de informática externa no siempre es bienvenido. Pueden aparecer requisitos y trabas técnicas que seguramente harán más complejo el desarrollo.

- Pueden existir intereses contrarios entre dos o más áreas de la organización, habitualmente entre aquella que se beneficie con la nueva implementación versus aquella que tendrá más carga de trabajo o pierda el control sobre sus tareas.

- Tambien puede haber intereses políticos y/o económicos opuestos. Por ejemplo, cuando se selecciona a un proveedor de software diferente del que propuso un jefe para el desarrollo.

- Por tratarse de un sistema nuevo, afectar a un nuevo negocio y/o involucrar nueva tecnología, no se tienen usuarios para relevar.

En estos escenarios, puede resultar inviable depender únicamente de los usuarios para obtener los requerimientos, y es necesario explorar estrategias "no convencionales". Se deben buscar fuentes alternativas de información que complementen o reemplacen las entrevistas con los usuarios. Documentos comerciales, marcos legales y normativos, sistemas existentes, informes utilizados, manuales de procedimiento, descripciones de procesos y el conocimiento de expertos externos pueden ser algunas de las fuentes de información que se pueden aprovechar.

También puede ser eficaz detectar y mantener reuniones informales con aquellos usuarios que sabemos están a favor de nuestro trabajo, de modo de validar con ellos los datos relevados. Incluso puede negociarse con ellos la posibilidad de incorporar funcionalidades que ellos necesiten, además de las impuestas.

Resulta importante testear con frecuencia si quienes nos han contratado para desarrollar el software mantienen el poder y el control de la organización. Muchas veces ese poder es la única herramienta que tenemos para imponer el nuevo sistema. La utilización de modelo en espiral, con análisis de riesgo suele ser útil en estos casos.

9. Algunos mitos sobre el relevamiento

Es interesante observar una serie de cuestiones que habitualmente se dan como definiciones para esta etapa pero que finalmente no resultan ser del todo exactas en escenarios reales:

- **La primera etapa de un desarrollo de sistemas es el relevamiento. En esta etapa se busca detectar los problemas y las necesidades de una organización o un área determinada.**

  Previo al relevamiento existen otras etapas donde los responsables de la organización y los desarrolladores acuerdan el marco general del trabajo. Deben acordarse el alcance del nuevo sistema y acordar presupuestos en tiempo y dinero, que obviamente serán generales y sujetos a revisión, pero que servirán al menos para permitir cobrar el trabajo

que deba realizarse en esta etapa, aun cuando después por algún motivo no se continúe con el desarrollo del sistema[28].

- **El relevamiento debe centrarse en los usuarios claves que, habitualmente, son los jefes o los que más conocen la operatoria actual de la empresa.**

  De más está aclarar que no en todos los casos el jefe es el que mejor conoce la operatoria del área. En alguna ocasión pueda que además debamos apoyarnos en algún operario calificado o en personal con mucha antigüedad.

  Pero… es necesario también intentar reconocer la actitud del usuario respecto del proyecto. A veces, es preferible enfocarse y priorizarse aquellos usuarios que mejor pueden expresar sus conocimientos, en aquellos que pueden reconocer y transmitir mejor sus necesidades y en quienes tienen interés positivo en que el nuevo sistema de desarrolle.

  En el otro extremo, deben detectarse los "usuarios tóxicos" es decir aquellos que aparentan brindar la mejor y más amplia información, y hasta parecen que apoyan fervientemente nuestra tarea, pero, en realidad e internamente y por algún motivo, solo buscan el fracaso del proyecto.

- **Las entrevistas deben seguir los procedimientos formales que tenga la organización. Deben armarse previamente utilizando cuestionarios preestablecidos, para que de este modo se obtengan respuestas más concretas y certeras.**

  La formalidad es buena. Da un marco de seriedad y, en efecto, permite focalizar los temas sobre los que se quiere una respuesta. Sin embargo, no todas las personas reaccionan del mismo modo ante los cuestionarios y las entrevistas formales.

  Además, las organizaciones tienen diferente nivel de formalidad en sus procesos de evaluación. En algunas puede ser habitual que los empleados sean evaluados y contesten cuestionarios, mientras que en otras esto nunca ocurre. En este último caso quizá debamos buscar otros mecanismos.

  Muchas veces la utilización de métodos más informales, *"descontracturados"* o lúdicos permiten una mejor predisposición del usuario y, consecuentemente, una mejor calidad de la información recibida. Además, permiten que el usuario se exprese libremente y surjan temas no previstos originalmente.

  Hay que tener en cuenta que la participación de la alta dirección en los relevamientos es importante porque en cierto modo marcan la importancia que tiene el proyecto para la organización. Sin embargo, no todos los empleados se sienten cómodos hablando frente a sus superiores. Es bueno encontrar momentos y espacios seguros donde puedan expresarse libremente sobre aquellas cosas que creen que están bien y aquellas que creen que deberían ser modificadas con el nuevo desarrollo.

[28] Puede consultarse el apunte sobre "Estimación, Costeo y Precio del Software" para profundizar sobre el proceso de presupuestación

- **Cuanto más profundo es el relevamiento, mejor se entiende el problema y menos cambios futuros aparecen. Cuanto más tiempo se invierte en esta etapa, más calidad tendrá el diseño posterior y menos probabilidades habrá de rehacer código.**

  Cantidad no es sinónimo de calidad. Entrevistar a más cantidad de usuarios no nos asegurará mejor comprensión del problema. Siempre habrá alguna opinión más por considerar, algún usuario más para escuchar, pero llega el punto en que su aporte es marginal, no aparecen nuevas necesidades. Es más, en los extremos corremos el riesgo de quedarnos relevando *"eternamente".*

  El tipo y el tamaño del sistema, la experiencia del analista y el tipo de usuarios, acortan o alargan los tiempos de relevamiento.

  Es posible construir sistemas exitosos con relevamientos a usuarios muy breves o incluso nulos.

  En ciertos modelos el relevamiento es la única etapa en la cual se permiten los cambios y, por lo tanto, resulta un riesgo cerrar esta etapa. Pero hay que entender que los cambios pueden ocurrir en cualquier momento, incluso provenientes del entorno o de nueva normativa. Los mismos no pueden ser evitados ni limitados alargando esta etapa.

- **Los programadores no deben participar del relevamiento.**

  En términos puros, el relevamiento se realiza para lograr un diseño que luego pueda ser codificado por cualquier programador, aun cuando este no tenga contacto alguno con los usuarios.

  Sin embargo, la participación del programador en alguna parte del relevamiento puede ser muy beneficiosa para que este comprenda mejor el problema.

  Además, desde su experiencia técnica, puede tener dudas específicas o incluso aportar y proponer soluciones innovadoras, que quizá no pueden ser visualizadas ni por el analista ni por el usuario.

  Nuevamente la generación de prototipos es una forma de que usuarios, analistas y programadores trabajen en conjunto algún tema. El prototipo además ayuda a que las partes tengan un entendimiento común del problema que están resolviendo, aun cuando el vocabulario y las terminologías que usen sean diferentes.

10. Un enfoque alternativo

En este punto, cabe presentar un enfoque alternativo que, complementando las tareas tradicionales, ayuda a mitigar muchos de los problemas enunciados previamente. Este enfoque se basa en 3 conceptos clave:

a. ¡Cuidado! Hay personas del otro lado.

En el relevamiento se interactúa con gente. No pueden separarse los intereses personales de los laborales. Tampoco las relaciones internas y de poder que ocurren en la organización.

Es fundamental considerar el contexto y las circunstancias de cada individuo al realizar el relevamiento de requerimientos. Factores como los intereses, problemas, posición en el área y en la organización de cada persona pueden influir en su perspectiva y, por lo tanto, en la información que proporcionen. No obtendremos los mismos datos de alguien que está satisfecho con su trabajo y la compañía en la que trabaja, en comparación con un usuario que tiene una relación tensa con el equipo o está buscando cambiar de empleo.

También es importante analizar los procedimientos y las prácticas habituales de la organización al solicitar la opinión de sus empleados. Si el usuario está acostumbrado a participar en reuniones o entrevistas de cierto tipo, es razonable utilizar esos mecanismos para obtener su colaboración. Sin embargo, si la organización tiene procesos más informales, emplear técnicas a las que el usuario no está acostumbrado puede generar incomodidad y respuestas incompletas. Es fundamental adaptar nuestras técnicas de relevamiento a la cultura y las prácticas de la organización.

El uso de enfoques más amigables y que fomenten una relación relajada y de confianza puede llevar a obtener resultados de mayor calidad. Si bien los registros escritos ayudan a recopilar la información de manera más precisa, también pueden generar cierta resistencia. Los usuarios pueden no expresarse de la misma manera si cada una de sus palabras queda registrada textualmente. Además, es esencial establecer una buena comunicación con los usuarios y explicarles claramente el propósito y los beneficios del relevamiento de requerimientos. Esto les ayudará a comprender la importancia de su participación y a sentirse más cómodos al brindar información detallada.

La incorporación de dispositivos portátiles para el registro de entrevistas ofrece indudables ventajas. No obstante, también introduce barreras adicionales, como el hecho de que el entrevistador escriba detrás de una pantalla y el entrevistado no tenga visibilidad sobre lo que está ocurriendo. Para fomentar un ambiente más cómodo y obtener datos más precisos, es recomendable preguntar al entrevistado si se siente incómodo, informar qué aspectos se registrarán o permitir que visualice la pantalla durante el proceso

Tanto si se utilizan métodos manuales como dispositivo electrónico, es una buena estrategia consensuar lo que quedará finamente quedará escrito. De hecho, el usuario debería finarlo. Algunas cosas que manifieste “en confianza” quizá no deban quedar registradas, sino mantenida como información “off the record”[29].

[29] Esta expresión, en periodismo, se refiere a la información que se ha obtenido de fuentes confidenciales o extraoficiales, y que no debe ser publicada por el medio. Sirven al periodista para obtener otra información o para seguir investigando otras fuentes.

Una buena práctica para ganar confianza es negociar necesidades reales con necesidades impuestas. Dentro de nuestras posibilidades, deben contemplarse las necesidades particulares de los usuarios, aun cuando estas no tengan que ver con el sistema propiamente dicho (ej. Conseguir un mejor equipamiento, o acceso a algún sitio restringido)

En los casos más extremos, como en escenarios hostiles, el relevamiento podría reducirse al mínimo, tendiendo principalmente a generar algún punto de interés común, recurriendo después a fuentes alternativas para obtener requerimientos.

b. Ingeniería de requerimientos versus relevamiento.

El término "relevamiento" generalmente se entiende como la recopilación de información proporcionada por las personas a las que se releva. Sin embargo, como se ha explicado anteriormente, no podemos confiar en que los usuarios nos proporcionen todos los datos de manera completa. Los requerimientos deben buscarse activamente en lugar de esperar a que aparezcan. El analista debe adoptar una actitud proactiva, buscando la mejor manera de obtener y definir los requisitos. El término "ingeniería de requerimientos" es más adecuado para describir una etapa en la que los requisitos deben construirse a partir de la información que se obtenga.

Es posible preparar hipótesis de trabajo y definiciones anticipadas en función de la experiencia previa y la información disponible, así como del tipo de sistema que se pretende desarrollar. Las entrevistas con los usuarios no se limitarán únicamente a recopilar información sobre sus tareas y necesidades, sino que también buscarán obtener definiciones precisas que requieran aclaraciones específicas.

Para ejemplificar lo dicho puede pensarse que en un sistema de cobranzas es imprescindible conocer cómo se conforma la cuenta corriente. Básicamente hay dos alternativas: “Por Factura” o “Por Saldo”. En la primera cada factura debe ser cancelada individualmente la su totalidad, mediante uno o varios pagos. En la segunda todas las facturas generadas generan un saldo deudor que el cliente va cancelando. Puede pagar cualquier cifra que desee, incluso generando saldo a favor. El analista deberá tener en claro que debe obtener una definición sobre cual método es el utilizado para la cuenta corriente a la hora de relevar el proceso. Resuelto esto, muchas de las cosas que el usuario nos diga ya son parte de un proceso ya conocido, y del que sabemos cómo funciona. Por el contrario, si olvidamos preguntar sobre esta modalidad o el usuario no nos lo indica, no podremos continuar. Incluso puede ocurrir que, por los procesos manuales, hoy trabaje con imputación por factura, porque resulta más sencillo controlar manualmente y reclamar los comprobantes impagos, pero que a partir del nuevo sistema se opte por trabajar sobre saldo, un mecanismo más ágil y flexible.

También es importante considerar los aportes del profesional involucrado, basadas en sus conocimientos y experiencias, así como la adopción de mejores prácticas para los procesos. El nuevo sistema no necesariamente debe replicar exactamente las operaciones existentes en una organización, sino que este momento brinda la oportunidad de evaluar nuevas alternativas y enfoques para las tareas. De hecho, esto no ocurre por casualidad o como un valor añadido, ya que a menudo se contratan

consultoras que no solo se encargan del sistema, sino también de mejorar los flujos y procesos, y cobran un valor adicional por este servicio.

E l resultado final del relevamiento, una vez completado, debe ser presentado de manera formal a todas las partes interesadas. Es crucial que este resultado sea validado y firmado por cada uno de los involucrados, lo que garantiza su aceptación y compromiso. Sin embargo, en ciertas circunstancias, pueden surgir discrepancias o la necesidad de ajustar los plazos o acuerdos iniciales. En tales casos, es posible llevar a cabo negociaciones para encontrar soluciones satisfactorias y llegar a un consenso mutuo. Esto permite mantener una comunicación efectiva y gestionar las expectativas de todas las partes involucradas en el proceso de relevamiento.

c. Aprovechemos el primer encuentro con el usuario.

Durante esta etapa se tiene el primer contacto con los futuros usuarios y con él área donde más adelante se implementará el sistema. Esta primera impresión es entonces decisiva para lo que ocurrirá después.

En esquemas de desarrollo basados en un plan es posible que, nuestro segundo encuentro, sea ya en la etapa de implementación del sistema ya terminado. Es necesario entonces aprovechar al máximo esta primera reunión para recaudar la mayor información que podamos, a los efectos de aprovecharla en la construcción y que la experiencia de uso del sistema lo más orientada al usuario posible.

Es importante tener en cuenta que durante cada encuentro existe una comunicación bidireccional. El usuario también está evaluando al analista y formándose una primera impresión sobre cómo afectará la implementación de un nuevo sistema a su trabajo. Durante este proceso, se establecerán vínculos de confianza o rechazo, y se desarrollarán actitudes más o menos favorables hacia el proyecto en general. Es crucial que el usuario perciba que el nuevo sistema será beneficioso para su labor diaria y que nuestra intención es ayudarlo y ajustar el sistema lo mejor posible para lograr ese objetivo.

Eventualmente, puede ser conveniente plantear algunas reuniones de avance donde se vaya informando al usuario de lo que está pasando en las etapas en las que no participa. Por supuesto siempre está el riesgo de que en estas reuniones aparezcan cambios o nuevos requisitos, pero hay que comprender que la necesidad del cambio perdurará, la contemplemos o no.

Cuanta más información se tenga del usuario, de sus expectativas, de la cultura se la organización, de su área y entorno, mejor puede lograrse una implementación exitosa.

Incluso hay que prestar atención a cada área en particular, tratando de identificar aquellas que podrían ser más problemáticas a la hora de implementar el nuevo del sistema, detectando si es necesario realizar algún trabajo previo al despliegue.

Para ilustrar este punto, podemos tomar como ejemplo algo aparentemente ajeno al desarrollo de sistemas, como la afición de un usuario por un equipo de fútbol en particular. Si durante nuestro primer encuentro notamos que el usuario tiene en su escritorio un escudo y una bandera de un club específico, podemos utilizar esa

información para diseñar posteriormente una interfaz con los colores de su equipo. La predisposición hacia el nuevo sistema será diferente si presentamos una pantalla con los colores de su equipo favorito en comparación con los colores de su rival. Del mismo modo, sería un error intentar implementar un sistema utilizando los colores que utiliza habitualmente una empresa competidora o los colores de un grupo político opositor en la interfaz.

El concepto de experiencia de usuario es fundamental en la actualidad. Se refiere al conjunto de factores y elementos que influyen en la interacción entre el usuario y un entorno o dispositivo específico, y cómo esto se traduce en una percepción positiva o negativa del servicio, producto o dispositivo en cuestión. Para lograr una experiencia satisfactoria, es necesario comprender al usuario en un nivel más profundo, más allá de sus tareas específicas.

11. Relevamiento y Metodologías Agiles

En el desarrollo ágil de software, la obtención de los requerimientos de usuario presenta características distintivas. En lugar de buscar obtener todos los requerimientos desde el inicio, se adopta un enfoque incremental. Los requerimientos se expresan en forma de historias de usuario, que son pequeñas solicitudes o funcionalidades específicas. Estas historias se priorizan y se van implementando directamente en el software, el cual está en constante funcionamiento y evolución. En este proceso, el usuario desempeña un papel activo como parte integrante del equipo de desarrollo.

Las metodologías ágiles ofrecen ventajas significativas con relación a la problemática de la definición completa de requisitos al inicio del proceso. Sin embargo, es importante destacar que algunos de los temas planteados anteriormente también pueden surgir en el desarrollo ágil. Por lo tanto, algunas de las recomendaciones mencionadas previamente pueden ser aplicadas en este contexto para abordar estos desafíos de manera efectiva.

Aunque se adopten metodologías ágiles, es crucial recordar que los usuarios son personas y que la formalización de su trabajo en historias puede resultar novedosa para ellos. Las dinámicas de poder y el flujo de información también pueden cambiar, independientemente de la metodología utilizada. En ambos casos, la participación de usuarios clave sigue siendo de gran importancia. Además, es importante reconocer que los escenarios hostiles también pueden presentarse en el contexto del desarrollo ágil.

Para los usuarios, formar parte del equipo de desarrollo a menudo implica una carga de trabajo adicional a sus responsabilidades habituales. Por lo tanto, es crucial garantizar que reciban algún tipo de recompensa, ya sea en forma de bonos salariales, horas extras o días libres. En caso de que esto no sea posible, es fundamental que comprendan claramente los beneficios que obtendrán con la implementación del nuevo sistema, de manera que justifiquen el tiempo y esfuerzo invertidos.

En resumen… este trabajo aborda fundamentalmente la problemática del desarrollo tradicional, donde la obtención de requisitos es una tarea grande y que debe ser completada al inicio, pero también puede servir conceptualmente y como como referencia para quienes utilicen metodologías ágiles.

12. Conclusiones

La utilización de herramientas adecuadas (entrevistas, cuestionarios, etc.) para obtener los requisitos del usuario es solo el primer paso en un proceso complejo. Existen varios factores que deben considerarse para garantizar una obtención efectiva de los requisitos. No se trata únicamente de recopilar información, sino de asegurarse de que los requisitos reflejen las necesidades reales, estén completos y permitan el desarrollo de un software de calidad. Esta tarea va más allá de simplemente estudiar los componentes de la etapa de obtención de requisitos.

Es fundamental familiarizarse con los problemas comunes que suelen surgir durante el análisis de requisitos y tomar medidas preventivas para evitarlos. Esto implica identificar posibles sesgos, ambigüedades o contradicciones en los requisitos, así como comprender las limitaciones y restricciones del proyecto. Al conocer estos problemas frecuentes, se pueden aplicar diversas estrategias y técnicas para lograr un relevamiento exitoso y sentar las bases para el desarrollo de un software de calidad. Es un proceso que requiere un enfoque proactivo y un compromiso continuo para asegurar que los requisitos sean sólidos y satisfactorios.

13. Bibliografía

IAN SOMMERVILLE: “Software Engineering”. 10ma edición. 2016. Pearson Education.

Apunte 5

Conceptos Fundamentales del Diseño de Software

1. Notas preliminares

El Diseño es aquella etapa del ciclo de vida del desarrollo de software, en la cual se platea la solución informática a las necesidades que los analistas relevaron y que describieron el "Documento de Requerimientos de Software". Se analizan dichos requerimientos para producir una descripción detallada de la estructura y funcionamiento, de modo que los programadores puedan construirlo. La salida es un conjunto de modelos y artefactos que registran las principales decisiones adoptadas.

Con el paso del tiempo, a medida que los sistemas fueron evolucionando y empezaron incluso a trascender los límites de la organización, las necesidades de diseño fueron aumentando. Ya no solo basta "traducir" el relevamiento a "instrucciones detalladas" que el programador pudiera entender, sino que comienza a ser necesario definir muchas más cosas. Por ejemplo, cuál será la arquitectura en la cual estará montado el sistema, que servidores utilizará, donde estará alojada la base de datos y como se accederá a ella, como será la seguridad; entre varios otros temas. Los puntos vinculados a los diagramas y metodologías de diseño y a bases de datos exceden el alcance de este apunte.

2. Repaso

Ya se ha dicho en apuntes anteriores que en la etapa de diseño precisamente se diseña, valga la redundancia, una solución informática a los problemas encontrados; obviamente contemplando las necesidades de los usuarios. Generan la documentación con el suficiente detalle técnico para que los programadores puedan construir el código a partir de ella.

Como también se mencionó anteriormente, el resultado final de la Ingeniería de Requerimientos (Análisis) es un documento usualmente denominado **"Documento de Requerimientos de Software"** donde se plantea que es lo que se necesita que el sistema haga. Este documento, **que debe estar validado y firmado tanto por los analistas como por los usuarios** intervinientes en el proceso de relevamiento, debería contener:

- Los servicios que se ofrecerán al usuario, incluyendo la explicación detallada de los mismos.
- Panorama de alto nivel de la arquitectura del sistema.
- Requerimientos funcionales y no funcionales, ambos con detalle.
- Modelos y gráficos que ayuden al entendimiento.
- Plan de actualización del sistema.
- Glosario de términos.
- Apéndices con información de interés, por ejemplo, información relacionada a algún hardware específico que deba utilizarse, como puede ser un lector de códigos o una impresora particular.

A partir de este documento los diseñadores comienzan a trabajar en **qué acciones** deberá realizar el sistema para ir dando cumplimiento de todos y cada uno de los requerimientos planteados, dentro de las restricciones y sin omitir exigencias no funcionales[30] planteadas.

3. La etapa de diseño – Una mirada rápida

¿Qué es el diseño de software?

El diseño es el anhelo de todo ingeniero o arquitecto de software. Es el punto en el que las reglas de la creatividad convergen para crear un sistema que cumpla con los requisitos de los usuarios, las necesidades del negocio y las consideraciones técnicas. A través del diseño, se construye una representación o modelo del software. A diferencia del documento de requerimientos, que se enfoca en describir los datos necesarios, las funciones y el comportamiento, el modelo de diseño ofrece detalles técnicos sobre cómo el software debe operar. Esto abarca aspectos como la arquitectura, las estructuras de datos, las interfaces y otros componentes necesarios para implementar el sistema.

El diseño puede compararse con el plano de una casa. Al igual que un plano proporciona a los constructores toda la información necesaria para edificar una casa, el diseño del software ofrece una guía detallada para su implementación. Así como un plano indica dónde deben ubicarse los caños, las ventanas y la dirección de apertura de las puertas, el diseño del software establece la arquitectura, la disposición de los componentes y las interacciones entre ellos. Proporciona una visión clara de cómo debe construirse el software y cómo se integran las diferentes partes para lograr un funcionamiento coherente y eficiente.

¿Cómo podemos definirlo formalmente?

Es el proceso de aplicar distintas técnicas y principios arquitectónicos, con el propósito de definir un dispositivo, proceso o artefacto[31], con los suficientes detalles como para permitir su realización física.

El producto final del diseño es un documento (o varios) estandarizado y con el detalle técnico necesario, para que un programador pueda construir la solución informática, aunque no haya tenido participación en las etapas previas [32].

---

[30] El concepto de **exigencias no funcionales** hace referencia a aquellos pedidos de la organización que, aun cuando no tengan que ver directamente con el funcionamiento de software, deben ser contemplados en la construcción. Por ejemplo, que el fondo de pantalla tenga un determinado color, o que el tiempo de capacitación sea menor de 4 horas, o que el sistema deba contar con una ayuda en línea, o permitir operar con 10 usuarios concurrentes.

[31] Artefacto es un objeto construido con algún tipo de ingenio y técnica para un desempeño específico. En este caso un diagrama o modelo que ayuda a describir cómo debe funcionar el software que se irá a construir.

[32] Idealmente, cualquier programador debería poder desarrollar un software basándose solamente en las instrucciones que recibe en el diseño. En la realidad, esto no siempre es posible. No solo porque dichas instrucciones

¿Quién lo hace?

Siguiendo con el ejemplo de las casas, para dibujar un plano se requieren conocimientos técnicos específicos. Hay que respetar escalas y convenciones para que los constructores lo puedan interpretar.

El diseño de software requiere el conocimiento de metodologías y notaciones estándares. Esta etapa se considera fundamental y es la primera actividad en la que se requiere tener conocimientos específicos para definir la solución utilizando un lenguaje estándar, como el UML (Lenguaje Unificado de Modelado), que permite especificar las características sin ambigüedades y que puede ser comprendido por cualquier programador. Utilizar un lenguaje estándar facilita la comunicación y comprensión del diseño entre los miembros del equipo de desarrollo, permitiendo establecer una base sólida para la implementación del software.

Además, se requiere conocer las diversas arquitecturas que pueden utilizarse para construir software. Por lo tanto, el diseño debe ser realizado por quien tenga conocimientos de ingeniería de software y de los estándares que se usan para modelar el sistema.

¿Cuáles son los pasos?

El diseño del software se representa de diversas maneras. En primer lugar, se aborda la representación de la arquitectura general del sistema o producto. Luego, se enfoca en modelar las interfaces que conectan el software con los usuarios finales, otros sistemas, dispositivos y sus propios componentes internos. Por último, se lleva a cabo el diseño de cada uno de los componentes individuales que conformarán el software. Esta aproximación permite un enfoque sistemático y estructurado para crear un diseño completo y coherente del software.

¿Cuál es el producto final?

Durante la etapa de diseño del software, se genera principalmente un modelo que engloba los diferentes artefactos y representaciones arquitectónicas, las interfaces, los detalles de cada componente y las instrucciones de despliegue (instalación y puesta en marcha del software). Es fundamental seguir estándares técnicos específicos para construir este modelo, de manera que pueda ser interpretado por cualquier programador. Estos estándares aseguran la consistencia y comprensión del diseño, permitiendo una implementación efectiva y colaborativa del software. Al adherirse a estos estándares, se facilita la comunicación y el intercambio de conocimientos entre los miembros del equipo de desarrollo

rara vez tengan el suficiente detalle, sino porque la propia complejidad de un desarrollo hace que los programadores deban tener una comprensión mucho más amplia que la del módulo que les toca codificar.

¿Cómo se asegura la calidad del diseño?

Al finalizar esta etapa es necesario realizar una auditoria o revisión técnica. Debe asegurarse que el diseño contempla acabadamente los requerimientos del usuario, que no tiene errores o inconsistencias, que no existen mejores alternativas, y que se respetan las restricciones.

También será necesario auditar que se han contemplado los patrones que la organización aprobó como estándares y que se puede implementar en los plazos y los costos comprometidos. Solo después que esta auditoría aprueba el diseño estará listo para ser entregado a los programadores.

4. El proceso del diseño

El diseño del software se enmarca en el ámbito técnico de la ingeniería de software y se aplica independientemente del modelo de proceso utilizado. Esta etapa comienza una vez que los requerimientos han sido analizados y modelados. El diseño del software se centra en crear una solución que cumpla con los requerimientos identificados, teniendo en cuenta aspectos como la arquitectura, las interfaces, los componentes y otros elementos relevantes para la implementación del sistema. Es en esta fase donde se define la estructura y el funcionamiento del software, sentando las bases para su posterior desarrollo y construcción.

El diseño es crucial para el éxito de la ingeniería de software. A principios de la década de 1990, Mitch Kapor, creador de Lotus 1-2-3, publicó en Dr. Dobbs Journal un "**manifiesto del diseño de software**". Decía lo siguiente:

> *"Vitruvio[33] afirmaba que los edificios bien diseñados eran aquellos que tenían resistencia, funcionalidad y belleza. Lo mismo se aplica al buen software.* ***Resistencia:*** *un programa no debe tener ningún error que impida su funcionamiento.* ***Funcionalidad****: un programa debe se apropiado para los fines que persigue.* ***Belleza****: la experiencia de usar el programa debe ser placentera. Éstos son los comienzos de una teoría del diseño de software."*

Aun con todo el tiempo trascurrido, esta afirmación sigue hoy vigente y esas son características que todos buscamos en cualquier software o app:

En primer lugar, la **resistencia** del software se refiere a su capacidad de funcionar sin errores que impidan su correcto desempeño. Un programa confiable y sólido es fundamental para garantizar su utilidad y eficiencia en el cumplimiento de sus fines.

La **funcionalidad** del software se relaciona con su capacidad de cumplir los objetivos y requerimientos específicos para los que fue creado. Un programa debe ser apropiado para los fines

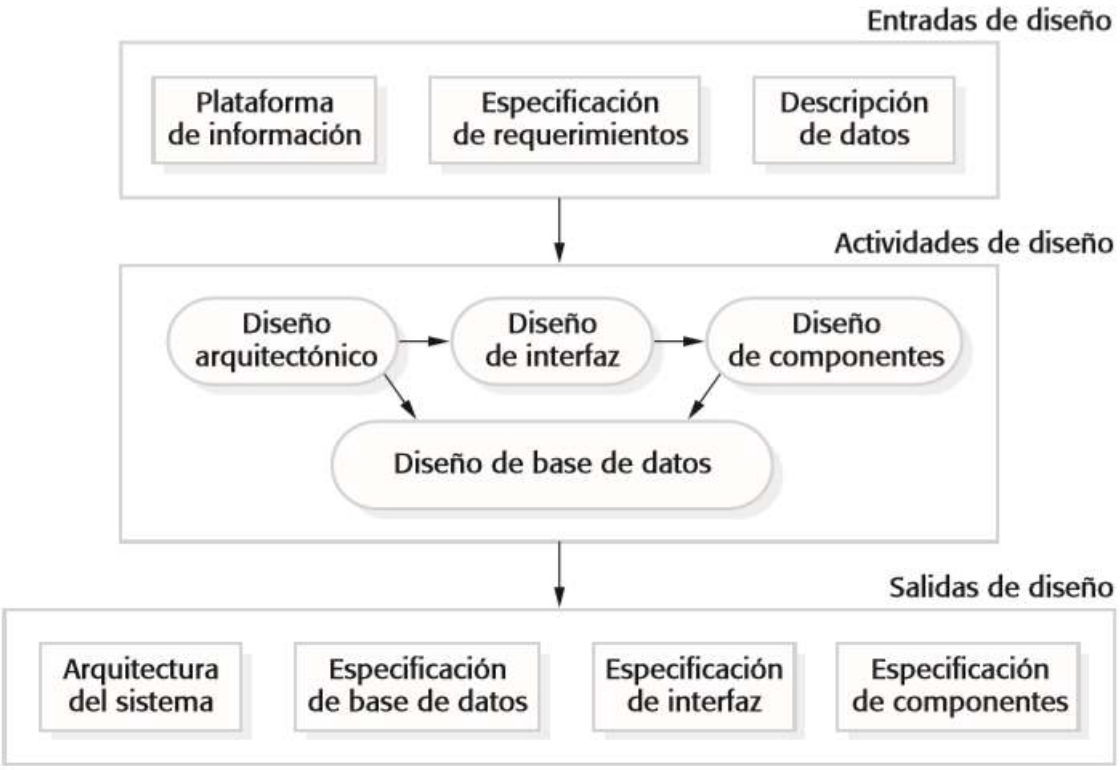
[33] Vitruvio fue un arquitecto y escritor romano que vivió en el siglo I a.C. Es conocido principalmente por su obra "De architectura" (Sobre la arquitectura), también conocida como "Los diez libros de arquitectura". Este tratado es una de las obras más importantes y completas sobre arquitectura de la antigüedad y ha sido una influencia significativa en la teoría y la práctica de la arquitectura a lo largo de los siglos.

que persigue y ofrecer las características y capacidades necesarias para satisfacer las necesidades de los usuarios.

Por último, la **belleza** del software se refiere a la experiencia de uso y a la interfaz de usuario. Un programa con una interfaz intuitiva, atractiva y fácil de usar proporciona una experiencia placentera para los usuarios. La estética visual, la usabilidad y la capacidad de respuesta son aspectos importantes que contribuyen a la percepción de belleza en el software.

Estas características siguen siendo relevantes y buscadas en el diseño de software en la actualidad. La combinación de resistencia, funcionalidad y belleza proporciona una base sólida para desarrollar aplicaciones exitosas y satisfactorias para los usuarios.

El proceso del diseño puede comprenderse con el siguiente esquema:



Las salidas del diseño son entonces:

- **Arquitectura del sistema**: Es el "plano general" del software. Refleja que módulos tendrá el sistema y cómo se relacionan entre sí. También se definirán dónde estarán almacenados cada uno de los componentes, teniendo en cuenta que, en sistemas muy grandes, es posible distribuirlos, permitiendo que alguno de ellos cumpla funciones específicas, mejorando la performance y la seguridad del sistema.

  En la construcción de casas existen determinados patrones de diseño arquitectónicos que se utilizan en determinados escenarios. En el polo norte las casas se construyen de hielo, con un formato particular que conocemos como *Iglú*; en los bosques, aparecen las *cabañas* de troncos como arquitectura dominante y en las grandes ciudades apilamos una casa arriba de otra en lo que llamamos *edificio*. De igual modo, también existen

determinados formatos estándares que los arquitectos pueden elegir para desarrollar su software.

Estos formatos arquitectónicos estándares se denominan **patrones** y resultan ser una solución rápida y efectiva a un problema común diseño. Esta solución ya se ha probado como válida, resolviendo problemas similares en ocasiones anteriores. **En el anexo 1 se muestran ejemplos de algunos de los patrones frecuentemente utilizados**

- **Especificación de la base de datos**: En base a que información deberá procesar el software, deberá establecerse qué datos manejará, donde se almacenarán y que atributos tiene cada uno de ellos (Por ejemplo, APELLIDO será un campo de texto de 35 caracteres y no podrá estar vacío; PRECIO será un campo numérico que solo acepte valores positivos, y CUIT será un campo de texto con el formato NN-NNNNNNNN-N)

  También se definirán las relaciones entre ellos (cuando se ingrese un país, este deberá previamente existir en la tabla de países). Incluso es posible establecerse condiciones especiales para algún dato (por ejemplo, el cálculo de dígito verificador), o condiciones especiales de actualización o de eliminación (por ejemplo, no debe permitir eliminar clientes que tengan saldos en las cuentas corrientes).

  Con esta especificación el arquitecto de datos (o quien tenga a su cargo esa función) podrá construir la base de datos con todas sus tablas, campos, atributos y relaciones.

- **Especificación de la interfaz**: describe como se comunicará el software consigo mismo, con los sistemas que operan con él y con dispositivos especiales (lectores de huella digital, lector de cheques, posnet, etc.). Y obviamente, también definen como serán las pantallas y la interrelación con las personas que lo operen. Lo dicho anteriormente respecto de patrones también es aplicable en este punto.

  La interfaz gráfica requiere una especial atención. No solo será el modo con el que los usuarios operan el sistema, sino que será la imagen que moldee su primera percepción sobre nuestro software. Una interfaz mal diseñada, confusa, con colores inadecuados dará una la idea, quizá errónea, de que el software es complejo, viejo, o difícil de operar. Seguramente predispondrá negativamente a los usuarios.

  Debemos buscar diseños de interfaz que permitan la mejor experiencia de uso, contemplando los diferentes tamaños y características de pantalla de los dispositivos con los que se acceda. No es lo mismo diseñar una pantalla de carga para un sistema que vaya a ser operado con mouse, teclado y un monitor de 15 pulgadas, que otro que será utilizado desde la pantalla chica y táctil de un celular.

  Los desarrolladores de sistemas operativos (Microsoft, Google, Apple) proveen patrones de diseño que permiten a los desarrolladores generar interfases comunes con el ecosistema. **En el anexo 2 tendrán como ejemplo las guías de diseño de los mencionados proveedores.**

- **Especificación de componentes**: Es el Plano “detallado”. Describe qué tareas realizará cada uno de los componentes que forman parte del software, cómo se organiza

internamente, cómo se relaciona con la base de datos, y cómo con los demás componentes del sistema y el contexto.

Por ejemplo, si estamos programando un listado de operaciones mensuales para calcular el IVA, tendremos que especificar que columnas tendrá, de que archivo se sacarán los datos de cada columna, que operaciones habrá que hacer con esos datos, como se calcularán los totales y e incluso, que módulo externo se ejecutará para comprobar que la impresora esté en línea.

También hay que definir cuestiones externas como por ejemplo en que menú estará colgado; que permisos se requerirán para su uso; si se puede ejecutar en modo concurrente con otros módulos del sistema; si por una cuestión de performance debe quedar en ejecutándose en segundo plano, y si puede o no ser accedido por otros módulos que precisen sus servicios. Hasta pensando en la reutilización, debe indicarse que partes del componente deben ser programados como rutinas separadas.

En resumen… el programador que reciba esta especificación deberá contar con todo el detalle necesario (como ya se dijo reiteradamente, expresado de un modo estandarizado) para poder desarrollar el componente sin necesidad de requerir ninguna información adicional, sin que existan ambigüedades ni cosas que queden a su criterio.

Tal como se ha dicho al comienzo del apunte, para cada una de estas cosas hay que utilizar diagramas y una terminología estándar, de modo que pueda ser comprendida por cualquier desarrollador, independientemente de que hayan tenido o no contacto previo con los usuarios o diseñadores. Nuevamente corresponde aclarar que el estudio detallado de estos diagramas pertenece a las materias como “Metodología de sistemas de Información” y “Sistema de datos”

5. El “arquitecto de software”

YA hemos comentado que o, a medida que los sistemas fueron evolucionando y empezaron incluso a trascender los límites de la organización, las necesidades de diseño fueron aumentando.

En desarrollos más grandes será necesario contar con una persona con conocimientos mucho más amplio y específicos, que colabore con equipo de desarrollo en la definición de la infraestructura sobre la que funcionará el nuevo sistema.

Según la CESSI, la cámara de la industria argentina del software, el arquitecto de software “es el responsable de definir la arquitectura de los sistemas tomando las decisiones de diseño de alto nivel y estableciendo los estándares técnicos, incluyendo plataformas, herramientas y estándares de programación, teniendo en cuenta los requisitos funcionales, no funcionales y las necesidades del negocio. En cooperación con el Líder de Proyecto, participa en la toma de decisiones adecuadas para lograr una arquitectura del sistema que garantice un mejor desempeño, flexibilidad, mantenibilidad, robustez, reúso o las cualidades que se pretendan de la aplicación”.

Entre las actividades que realiza un arquitecto de software:

- Negociar con el propietario de la aplicación y el líder de proyecto para tomar las decisiones de diseño de alto nivel que correspondan.

- Seleccionar el software: Seleccionar la tecnología a utilizar en conjunto con el líder de proyecto; y definir y revisar estándares y normas aplicables al diseño y construcción, brindando coaching técnico al equipo de desarrollo.

- Diseñar la arquitectura: Conducir la construcción del modelo de arquitectura/diseño, subdividiendo aplicaciones complejas en partes o componentes menores, más fáciles de manejar.

- Realizar un seguimiento del proceso de desarrollo para asegurarse que sus instrucciones se implementen en forma adecuada; y registrar los cambios de arquitectura que se produzcan.

- Asegurar la calidad: Medir la performance de la aplicación y conducir pruebas en relación con la performance, seguridad, etc.

- Facilitador: Colabora con otras áreas como seguridad informática, base de datos, operaciones, el equipo de mantenimiento, etc.

- Liderazgo Técnico: Capacidad para asumir la dirección técnica y asegurar todos los aspectos de la arquitectura con responsabilidad y autoridad. Poder realizar coaching y mentoring[34] sobre problemas técnicos, ayudando a la evolución profesional del equipo de programadores.

- En conjunto con el líder de pruebas, definir las pruebas de los requisitos no funcionales y de integración que se realizaran y efectuar un seguimiento de estas.

6. Diseño de la Interfaz de usuario

La interfaz de usuario es un componente fundamental en los sistemas actuales. Durante el proceso de diseño, es crucial presentar a los usuarios una interfaz clara, funcional y especialmente cómoda de usar en diversos dispositivos y tamaños de pantalla. Un diseño adecuado facilitará la operación del sistema y minimizará los errores de interacción. Además, como se mencionó anteriormente, la interfaz de usuario ayudará a superar la resistencia inicial hacia la nueva aplicación y dará forma a la percepción general de la facilidad o complejidad del nuevo sistema que se utilizará.

La interfaz de usuario desempeña un papel fundamental en la experiencia del usuario, ya que determina cómo se sentirá al utilizar el sistema. No existe una solución única para todas las

[34] El Coaching es el proceso por el cual se acompaña a un individuo a que cumpla sus objetivos o adquiera determinadas habilidades. El Mentoring significa que una persona experimentada, o con mayor conocimiento, ayuda a otra con menos experiencia.

interfaces ni para todos los usuarios. Es necesario estudiar el hardware utilizado, la tarea realizada e incluso el entorno de trabajo. La experiencia de un sistema que se ejecuta en una computadora de escritorio en una oficina será diferente a la de un quiosco informativo en un centro comercial, una estación de servicio o la caja de un banco. Además, los sistemas que se ejecutan en un navegador web también presentan limitaciones y características propias de los estándares de Internet.

Los diferentes tamaños de pantalla y la comodidad del usuario al utilizar un teclado físico o un ratón también influyen en el diseño de interfaces. Es necesario adaptarse a estas variables para garantizar una experiencia de usuario óptima.

Incluso en ocasiones se suele emular entornos que el usuario ya conoce, para poder de ese modo achicar la curva de aprendizaje y vencer la resistencia al cambio. A continuación, dos pantallas muy similares, la primera es una interfaz Windows, la segunda Linux.

Ben Shneiderman, en su libro "***Designing the User Interface: Strategies for Effective Human-Computer Interaction***" describe **ocho reglas de oro** que pueden seguirse a la hora de construir una buena interfaz de usuario:

1. **Luchar por la consistencia:**
   Utilizar los mismos patrones de diseño y las mismas secuencias de acciones, para responder a situaciones similares: el uso de un mismo del color, misma tipografía y misma la terminología en las diferentes pantallas, comandos y menús. Por ejemplo, un botón con una "X" debe cerrar una ventana en cualquiera de los lugares en ellos que se utilice. Una tilde verde se utilizará siempre para mostrar confirmación o validez y se usará siempre el mismo y para lo mismo. La combinación CTRL + P permitirá siempre imprimir la pantalla.

2. **Buscar la usabilidad universal:**
   Reconocer las necesidades de los diferentes tipos de usuarios. Agregar funcionalidades para novatos (ayudas) y expertos (atajos, comandos ocultos para saltar pasos, etc.)

3. **Entregar información de feedback:**
   Es necesario mantener a los usuarios informados de lo que está ocurriendo en cada etapa de su proceso. Esta retroalimentación debe ser significativa, relevante, clara y adaptada al contexto. Si el proceso tiene varias pantallas, en la segunda debe mostrarse cierta información, por ejemplo, el nombre del cliente con el que se está operando. Esto permite

que si el usuario es interrumpido (por ejemplo, recibiendo un mensaje de texto), cuando vuelva a prestar atención a la pantalla recuerde los datos más importantes que cargo anteriormente.

4. **Diseñar diálogos de cierre:**
   Agrupa las secuencias de acciones y proporciona información de cierre antes de continuar con el siguiente paso, por ejemplo, muestra un resumen de la operación de compra, antes de que el usuario la confirme.

5. **Prevenir errores:**
   Una buena interfaz debe estar diseñada para evitar errores tanto como sea posible. No se le debe permitir al usuario ingrese datos inválidos (precios negativos o email sin "@"). Pero también se debe ser claro en el mensaje que se muestre ante el error y como solucionarlo. Por ejemplo, al no confirmar una operación no debería simplemente indicar que hay "datos inválidos" sino indicar que "El mail no tiene un formato valido", que "El precio no puede ser negativo", o que "la contraseña requiere números y letras".

6. **Permitir la fácil reversión de acciones:**
   Posibilitar que, en la mediad de lo posible, las acciones puedan ser canceladas, en forma parcial o completa antes de la confirmación final (tecla ESC, por ejemplo).

7. **Darle el control al usuario:**
   Proporcionar herramientas para que el usuario sienta que controla la interfaz y no al revés. Minimizar acciones innecesarias. Permitir la interacción flexible (mouse o teclado, según prefiera). Permitir que, eventualmente, pueda cambiar el color de la pantalla; controlar el tamaño del texto; la ubicación de ciertos los objetos de la interfaz, como la barra de menú; o, incluso, ocultar botones que no utilice.

8. **Reducir la memoria de corto plazo:**
   Mostrar información relevante de pantallas anteriores. Proporcionar contenido preestablecido (mostrar la fecha del día, traer el nombre cuando se ingresa el código de cliente, calcular las cosas que pueden calcularse). Simular el mundo real (si se carga información de una factura esta debe cargarse en el orden que se cargaría en una factura manual). Presentar información en forma jerárquica y progresiva (comenzar primero pidiendo el país, después las provincias y por ultimo las localidades. Incluso las localidades podrán solo aparecer aquellas que sean de la provincia indicada)

7. El valor económico del diseño

En la mayoría de los casos, dentro de un equipo de Fórmula 1, el piloto suele ser la persona con el salario más alto, seguido por el diseñador. Esto se debe a que se reconoce la importancia crucial del diseño del automóvil en el rendimiento y los resultados obtenidos en la pista. Un diseño de excelencia puede marcar la diferencia en términos de ganar milésimas de segundo en cada curva, reducir el desgaste de los neumáticos, aumentar la velocidad, mejorar la maniobrabilidad y proporcionar mayor potencia en los momentos clave, como los adelantamientos.

El diseñador juega un papel fundamental en el desarrollo del automóvil de Fórmula 1, ya que su expertise y conocimientos técnicos permiten crear un vehículo altamente competitivo. Cada aspecto del diseño, desde la aerodinámica hasta los sistemas mecánicos y electrónicos, se optimiza para lograr el mejor rendimiento posible en la pista.

Si bien el piloto es quien finalmente lleva el automóvil a la victoria, su desempeño está estrechamente ligado a la calidad del diseño y la ingeniería del vehículo. Por lo tanto, es comprensible que los diseñadores sean valorados y remunerados en consecuencia dentro de un equipo de Fórmula 1, ya que su labor influye directamente en el rendimiento del automóvil y en la competitividad del equipo en general.

En el mundo de la moda, existe una similitud con el caso de la Fórmula 1 en cuanto a la valoración y reconocimiento que se otorga a los diseñadores. Es común que cuando las actrices famosas desfilan por la alfombra roja, se destaque en las fotos quién fue el diseñador del vestido que llevan puesto. Los grandes diseñadores de renombre suelen ser altamente valorados y se les paga grandes sumas de dinero por sus creaciones.

Esto se debe a que el diseño de moda es considerado una forma de arte y expresión creativa. Los diseñadores son los encargados de concebir y materializar las ideas, creando prendas únicas y distintivas. Su visión creativa, habilidades técnicas y capacidad para captar las tendencias de la moda son altamente valoradas en la industria.

Aunque es cierto que quienes cosen los vestidos también desempeñan un papel importante en el proceso de producción, son los diseñadores quienes establecen la dirección creativa y el concepto detrás de cada prenda. Su nombre y reputación están asociados con la calidad, la innovación y el estilo de sus diseños.

Aunque no se destaque de la misma manera que en la moda o la Fórmula 1, el diseño de software influye significativamente en la forma en que un sistema se comporta, resuelve problemas, brinda una experiencia de usuario satisfactoria y se destaca estéticamente.

En el mercado de las aplicaciones, es común encontrar numerosos programas que realizan tareas similares. Sin embargo, la diferencia radica en cómo están diseñados y desarrollados. Aquellos que presentan un buen diseño suelen sobresalir y ofrecer una experiencia superior en comparación con otros.

Un buen diseño de software implica considerar aspectos como la usabilidad, la eficiencia, la estabilidad, la escalabilidad y la seguridad. Esto implica tomar decisiones cuidadosas en cuanto a la arquitectura del sistema, el diseño de la interfaz de usuario, la elección de tecnologías adecuadas, entre otros aspectos.

Un software bien diseñado se caracteriza por ser intuitivo, fácil de usar, tener un rendimiento eficiente, ser confiable y adaptarse a las necesidades del usuario. Además, un buen diseño permite una mayor flexibilidad y mantenibilidad del software a lo largo del tiempo, lo que facilita su actualización y mejora continua.

En definitiva, el diseño de software es crucial para diferenciar productos en el mercado y garantizar que cumplan con las expectativas de los usuarios. Un buen diseño contribuye a un mejor

rendimiento, una experiencia de usuario satisfactoria y la resolución efectiva de problemas, lo que puede marcar la diferencia entre un software exitoso y uno menos destacado.

En el contexto del desarrollo de software, el diseño tiene un valor económico y estratégico por sí mismo. Contratar a un diseñador experto y pagar por su trabajo puede ser una inversión valiosa, ya que un buen diseño puede influir positivamente en la aceptación y adopción del software, así como en la experiencia de usuario y la satisfacción del cliente. Ese diseño podrá ser entregado luego a un programador que será el encargado de desarrollarlo físicamente.

Pero, además, es recién al final de esta etapa, cuando el plano general está construido, que se tiene la real dimensión del proyecto y, por ende, recién en este momento se podrá estimar, cuantos recursos se necesitarán en las próximas etapas y cuánto tiempo llevará el desarrollo. Al terminar el diseño sabré que tipo de base de datos voy a necesitar, cuantos módulos de ABM (altas bajas y modificación de datos) existirán, qué tan complejas serán las pantallas, qué procesos de cálculo habrá que realizar, y cuántos listados tendrá el sistema. **La real dimensión del proyecto se conoce en este momento. Recién entonces se podrá estimar cuantos recursos se necesitarán en las próximas etapas y cuánto tiempo y dinero llevará el desarrollo[35].**

8. Bibliografía

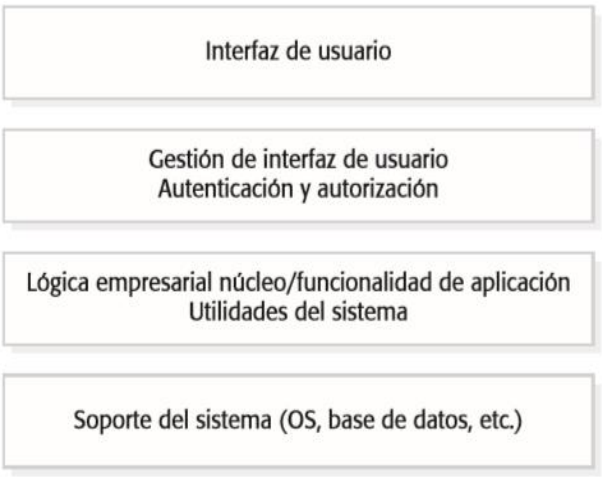
IAN SOMMERVILLE: “Software Engineering”. 10ma edición. 2016. Pearson Education.
ROGER PRESSMAN: “Ingeniería del Software”. 7ta edición. 2008. Ed. McGraw-Hill.
LAUDON Y LAUDON: “Sistemas de Información Gerencial”. 14 edición. 2016. Pearson Education
BEN SHNEIDERMAN: “Designing the User Interface: Strategies for Effective Human-Computer Interaction”, Sixth Edition. 2016 Pearson
CESSI: Perfiles ocupacionales de la industria IT: https://www.cessi.org.ar/perfilesit/

[35] Podrá profundizarse este tema con el apunte “Conceptos Fundamentales de la Estimación, Costeo y Precio del Software”

ANEXO 1 – Patrones Arquitectónicos

Patrón de diseño en Capas

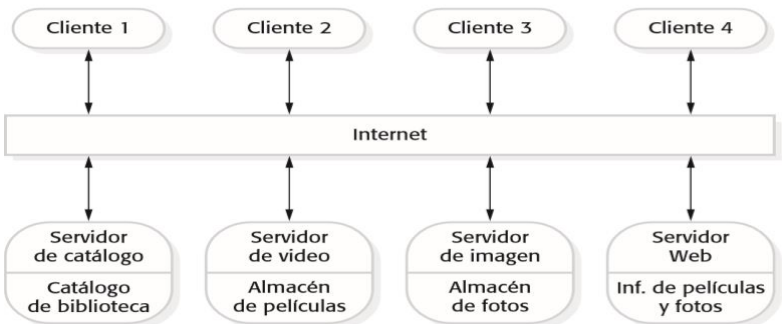
Los componentes del sistema se separan en capas. Cada capa presta servicios a la capa superior. Un modelo estándar es utilizar 4 capas.



La ventaja de este modelo es que podemos cambiar cualquiera de las capas sin modificar el resto. Podríamos por ejemplo cambiar de base de datos, sin que el sistema sufra otros cambios. O, en un homebanking, sería factible actualizar la interfaz de usuario sin afectar ni la lógica del negocio ni la seguridad integrada.

Patrón de Arquitectura Cliente Servidor

La arquitectura cliente-servidor es un modelo de diseño de software, en el que las tareas se reparten entre los proveedores de recursos o servicios, llamados **servidores**, y los demandantes de dichos servicios, llamados **clientes**. Cuando un cliente realiza peticiones es el servidor, quien le da respuesta. Si bien la idea se puede aplicar a programas que se ejecutan sobre una sola computadora, es más ventajosa en un sistema multiusuario, distribuido a través de una red de computadoras. En una arquitectura cliente-servidor, la funcionalidad del sistema se organiza en servicios, y cada servicio lo entrega un servidor independiente. Los clientes son usuarios de dichos servicios y para utilizarlos ingresan a los servidores.



Patrón de Arquitectura de Repositorio

Todos los datos en un sistema se gestionan en un repositorio central, accesible a todos los componentes del sistema. Los componentes no interactúan directamente, sino tan solo a través del repositorio. Este patrón se usa, generalmente, en sistemas que operan grandes volúmenes de datos.

Para abordar este tema con mayor profundidad puede consultar
IAN SOMMERVILLE: Ingeniería de Software. 9na Edición. 2011. Pearson Education.
Capítulo 6

ANEXO 2 – Patrones de diseño

En el presente anexo se presentan, a modo de ejemplo, los patrones de diseño que Google, Microsoft y Apple ponen a disposición de los desarrolladores. Estos patrones **no son obligatorios**, pero buscan que todo el ecosistema de aplicaciones tenga lineamientos comunes. Los usuarios terminan acostumbrándose y adaptándose a ese diseño y preferirán aplicaciones que lo respeten, salvo, claro está, en aquellas que requieran por algún motivo un diseño justificadamente diferente.

De este modo podremos observar cómo aplicaciones Android tienen características similares. Incluso las por Microsoft se comportan de un modo similar a las de Google, y viceversa con aplicaciones de Google en Windows.

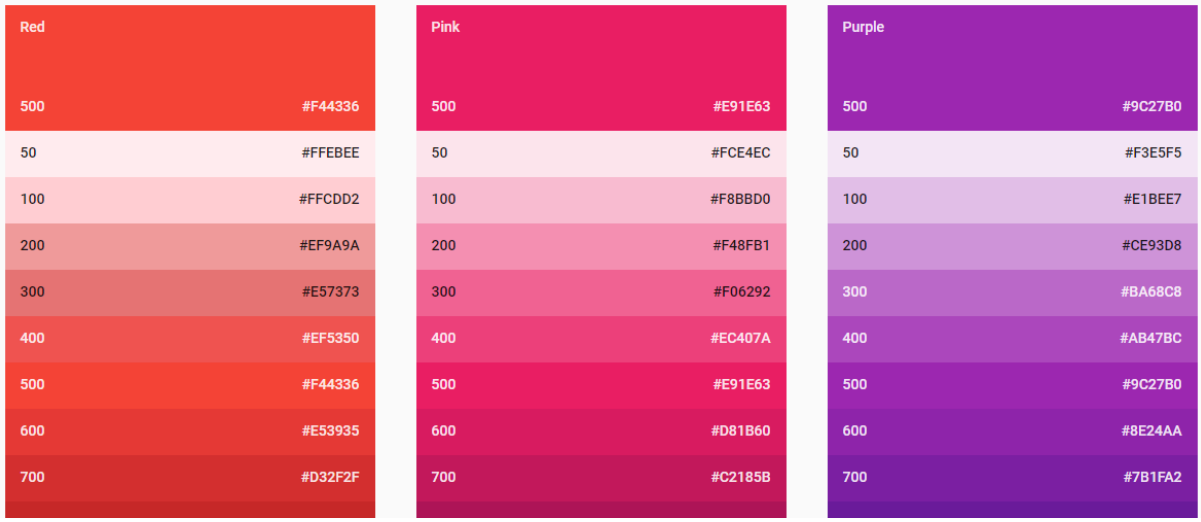
Otro ejemplo para mencionar son las aplicaciones de Office. No importa si usamos Word, Excel o Power Point, todos tienen menús similares, las opciones se llaman igual y funcionan del mismo modo (abrir es abrir, para todos los programas), los iconos son los mismos, los botones y los atajos de teclado también cumplen la misma función no importa lo que aplicación se use.

Material Design by Google – Vigente desde Android Loolipop 2014

Este patrón define desde la escala de colores a utilizar, hasta la distribución de los diferentes elementos de la pantalla.

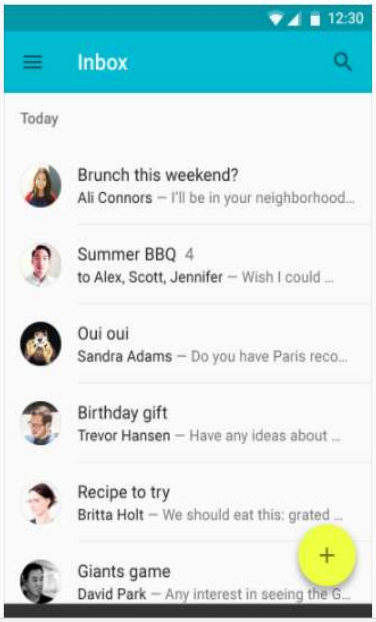
Color palette

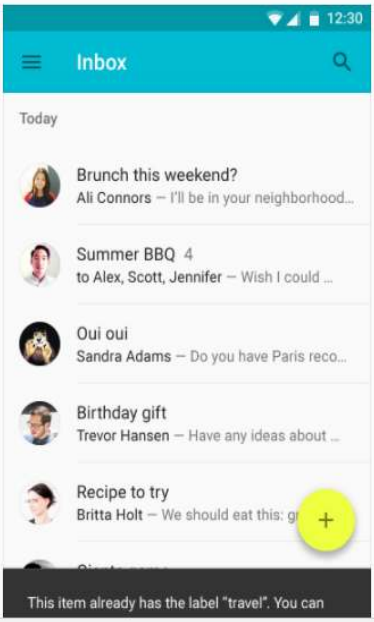
This color palette comprises primary and accent colors that can be used for illustration or to develop your brand colors. They've been designed to work harmoniously with each other. The color palette starts with primary colors and fills in the spectrum to create a complete and usable palette for Android, Web, and iOS. Google suggests using the 500 colors as the primary colors in your app and the other colors as accents colors.



Mobile

- Single-line snackbar height: 48dp
- Multi-line snackbar height: 80dp
- Text: Roboto Regular 14sp
- Action button: Roboto Medium 14sp, all-caps text
- Default background fill: #323232 100%





Others have failed, I will not.

40dp

Science, my boy, is made up of mistakes, but they are mistakes which it is useful to make, because they lead little by little to the truth.

Where others have failed, I will not fail.

32dp

Science, my boy, is made up of mistakes, but they are mistakes which it is useful to make, because they lead little by little to the truth.

Others have failed, I will not.

40dp

Science, my boy, is made up of mistakes, but they are mistakes which it is useful to make, because they lead little by little to the truth.

And whichsoever way thou goest, may fortune follow.

32dp

Science, my boy, is made up of mistakes, but they are mistakes which it is useful to make, because they lead little by little to the truth.

Type 45sp, Leading 48pt

Type 34sp, Leading 40pt

Type 24sp, Leading 32pt

Type 15sp and 16sp, Leading 28pt

Type 15sp and 16sp, Leading 24pt

Type 13sp and 14sp, Leading 24pt

Type 13sp and 14sp, Leading 20pt

Ratio keylines

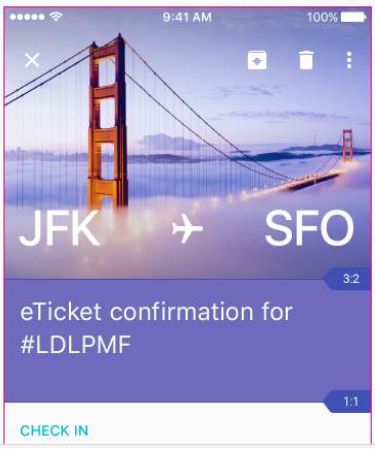
The proportion of an element's width to its height (called the **aspect ratio**) applies to both UI elements and screen size. It is written as width:height.

These aspect ratios are recommended:

- 16:9
- 3:2
- 4:3
- 1:1
- 3:4
- 2:3

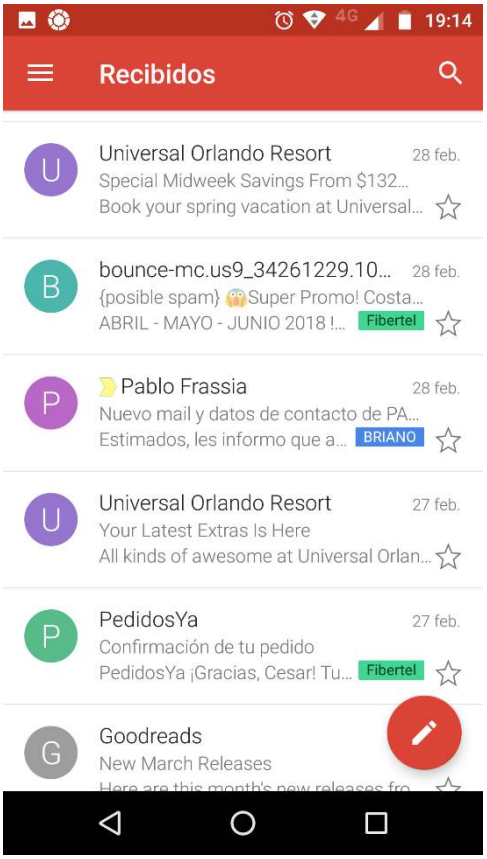
For example:

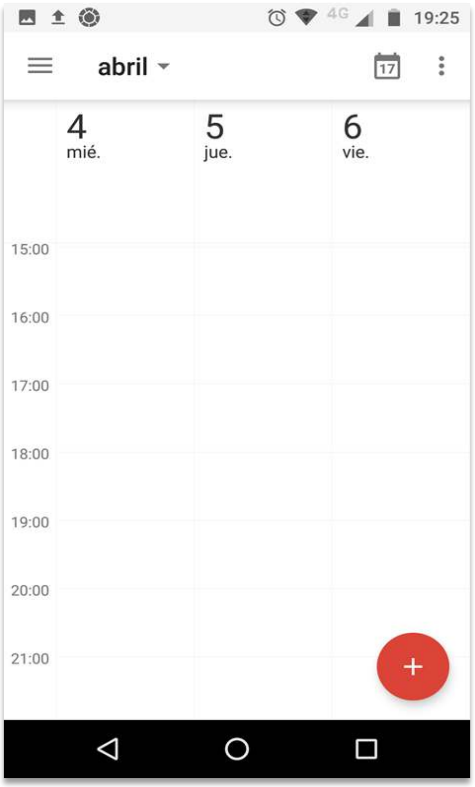
- A 1:1 aspect ratio means an element has equal height and width

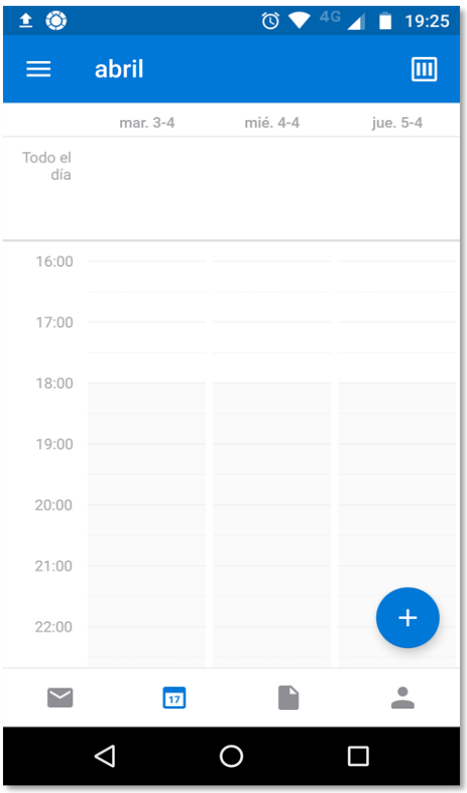


En este ejemplo vemos como dos aplicaciones de calendario, desarrolladas por dos empresas diferentes, parecen idénticas en su funcionalidad. Solo cambian en el color

Gmail y Calendario de Google	Outlook y Calendario de Microsoft

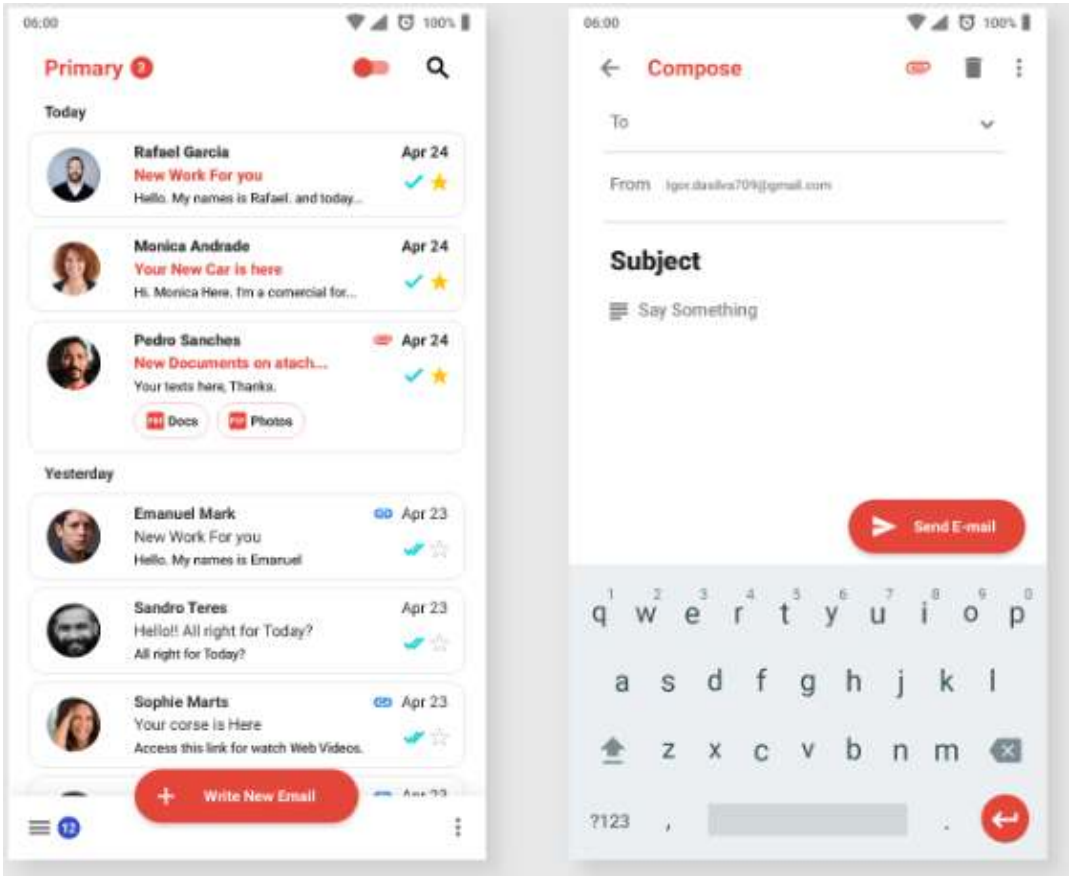


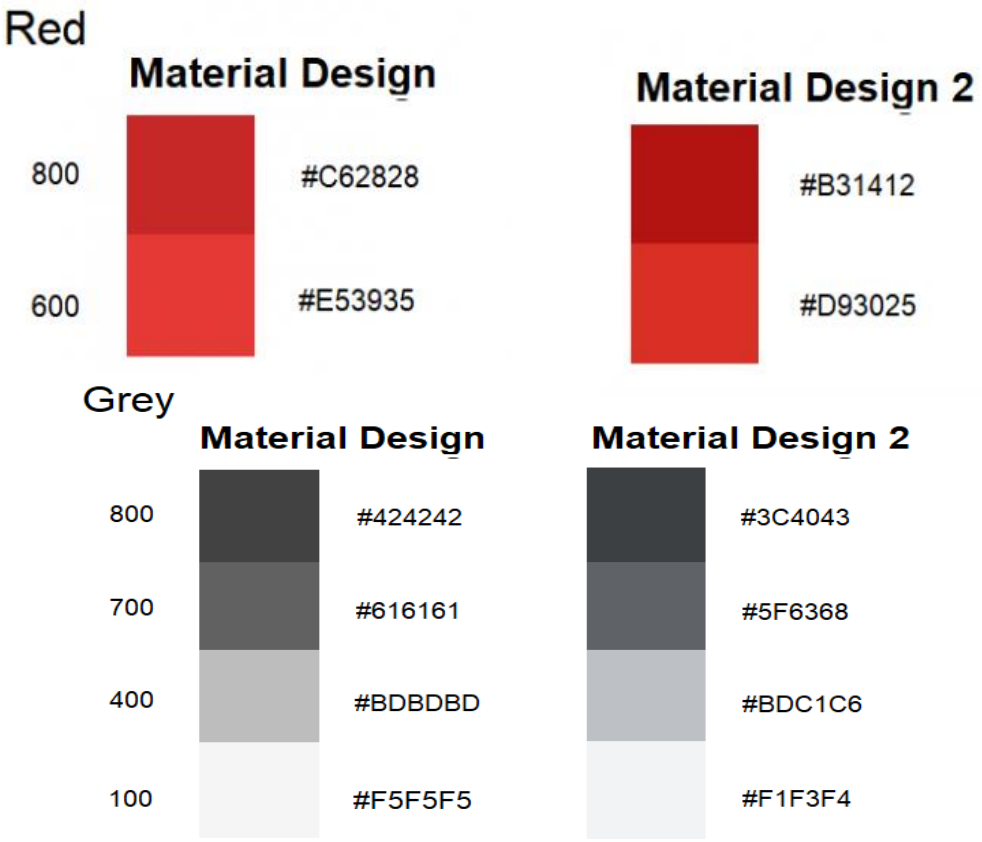


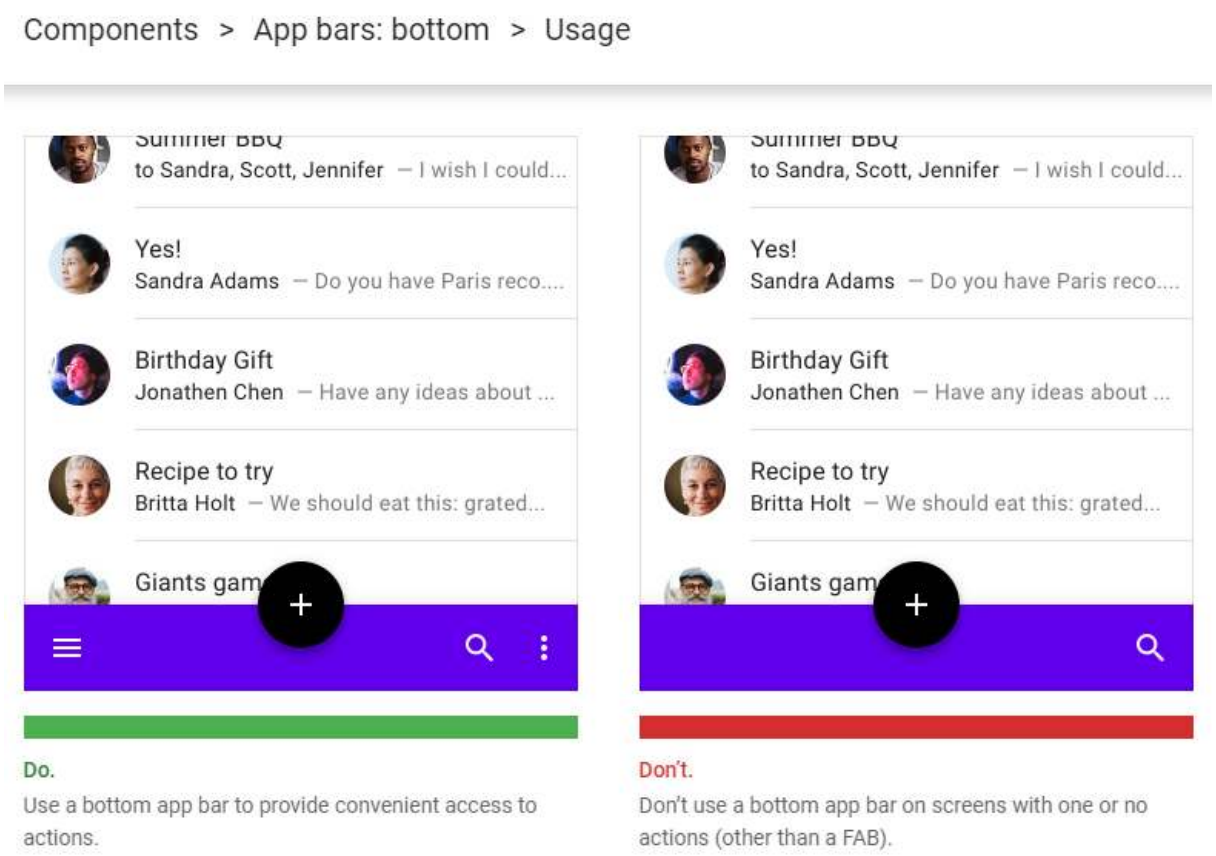


Material Design 2 by Google – **Vigente desde mediados del 2018**

Cambian, entre otras cosas, el patrón de color (colores más vivos y con menos matices) y la ubicación de menús y los botones para hacer más fácil de usar las aplicaciones.



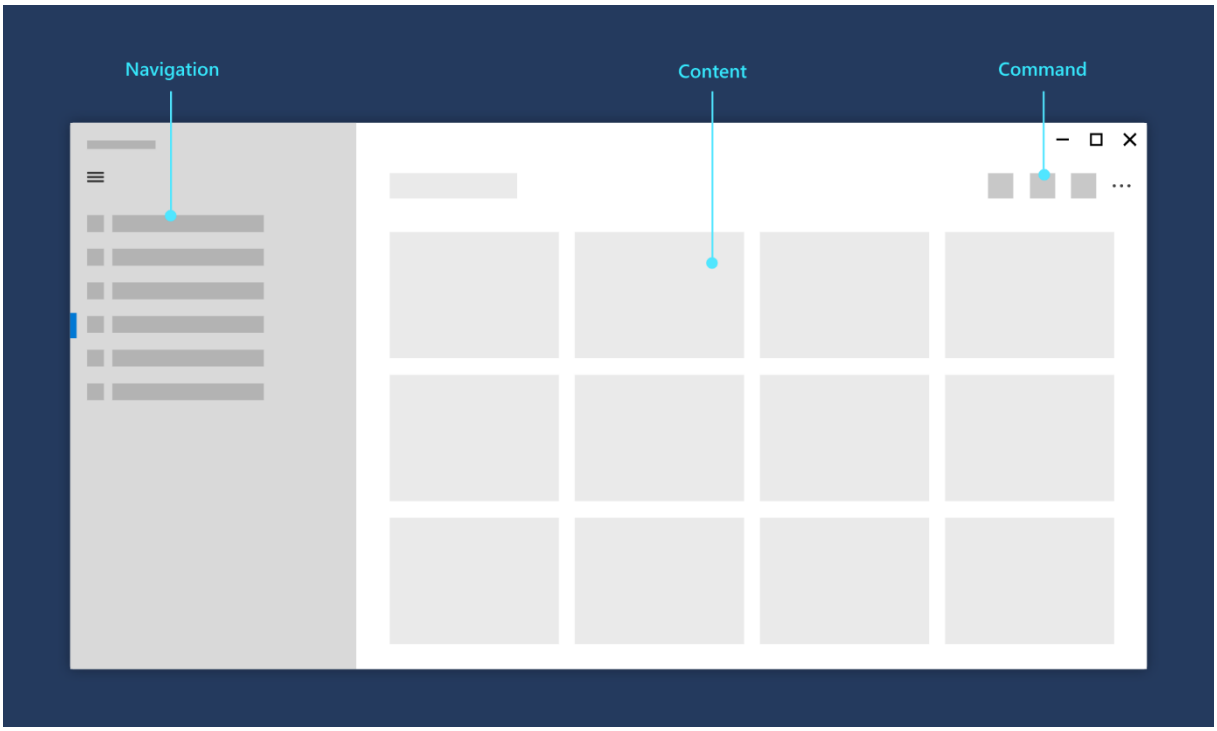


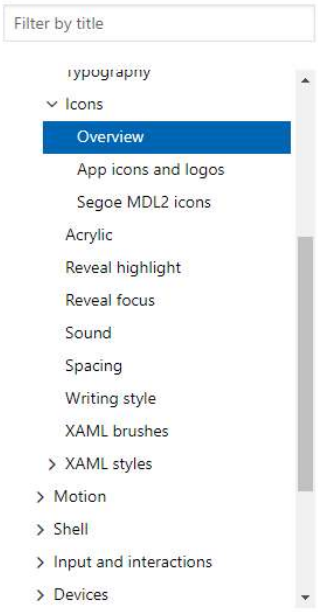


Mas información: **https://material.io/guidelines/**

Fluent Desing by Microsoft – **Patrones para Windows 10 a partir del 2018**

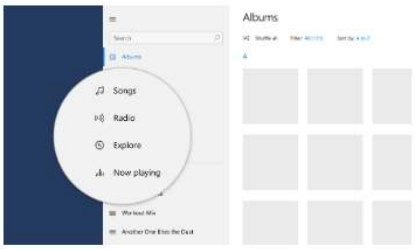
Diseño y disposición de las pantallas de aplicación, por ejemplo, de las ventanas del Sistema Operativo





Icons can appear in apps—and outside them:

Icons inside the app



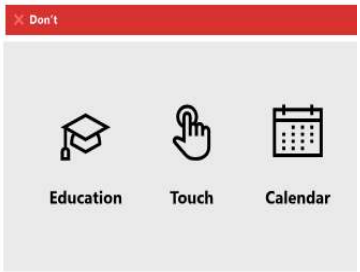
Inside your app, you use icons to represent an action, such as copying text or navigating to the settings page.

Icons outside the app

Outside your app, Windows uses an icon to represent your app in the start menu and in the taskbar. If the user chooses to pin your app to the start menu, your app's start tile can feature your app's icon. Your app's icon appears in the title bar and you can choose to create a splash screen with your app's logo.

This article describes icons within your app. To learn about icons outside your app (app icons), see the app and tile icons article.

Use an icon for actions, like cut, copy, paste, and save, or for navigation items in a navigation menu.



Use an icon if one already exists for the concept you want t represent. (To see whether an icon exists, check the Segoe icon list.)

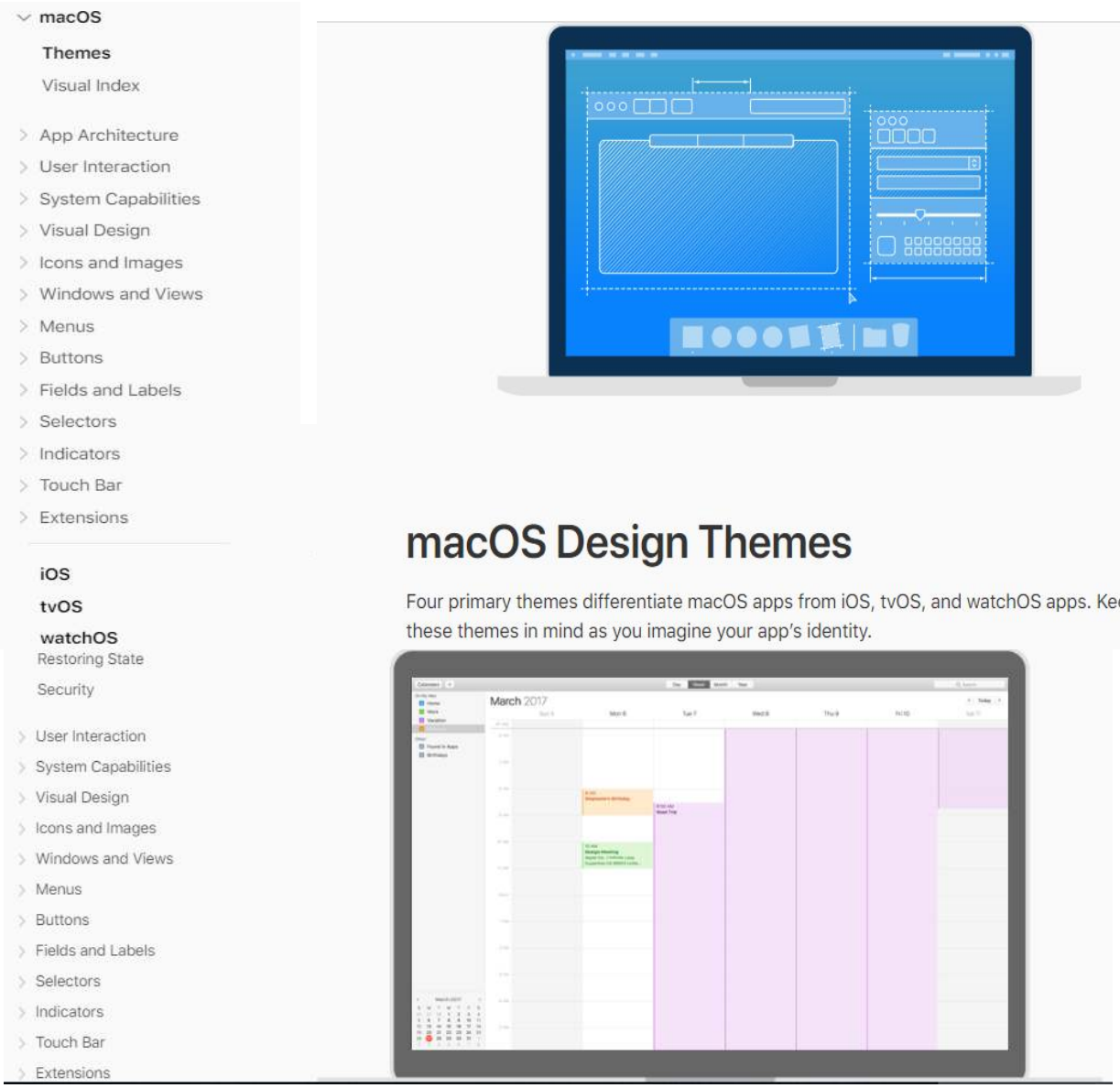
Use an icon if it's easy for the user to understand what the icon means and it's simple enough to be clear at small sizes.

Don't use an icon if its meaning isn't clear, or if making it c requires a complex shape.

Icon list

Please keep in mind that the **Segoe MDL2 Assets** font includes many mc are intended for specialized purposed and are not typically used anywhe

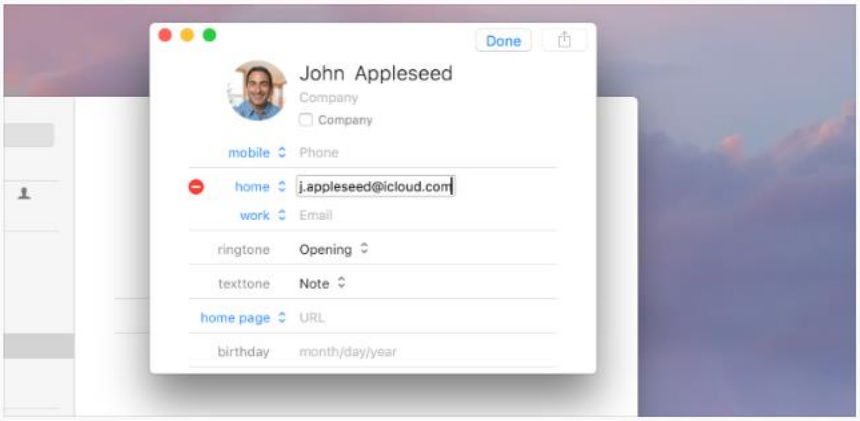
Symbol	Unicode point	Description
	E700	GlobalNavigationButton
	E701	Wifi
	E702	Bluetooth
	E703	Connect
	E704	InternetSharing
	E705	VPN
	E706	Brightness
	E707	MapPin


macOS
Themes
Visual Index
App Architecture
User Interaction
System Capabilities
Visual Design
Icons and Images
Windows and Views
Menus
Buttons
Fields and Labels
Selectors
Indicators
Touch Bar
Extensions
iOS
tvOS
watchOS
Restoring State
Security
User Interaction
System Capabilities
Visual Design
Icons and Images
Windows and Views
Menus
Buttons
Fields and Labels
Selectors
Indicators
Touch Bar
Extensions
macOS Design Themes
Four primary themes differentiate macOS apps from iOS, tvOS, and watchOS apps. Keep these themes in mind as you imagine your app's identity.
March 2017

- macOS
- App Architecture
- **User Interaction**
  - Authentication
  - **Data Entry**
  - Drag and Drop
  - File Handling
  - Help
  - Keyboard
  - Mouse and Trackpad
  - Providing User Feedback
  - Requesting Permission
- System Capabilities
- Visual Design
- Icons and Images
- Windows and Views
- Menus
- Buttons
- Fields and Labels
- Selectors
- Indicators

Data Entry

Whether using a keyboard, mouse, trackpad, or your voice, inputting information can be a tedious and sometimes error-prone process. When an app asks for lots of input before doing anything useful, people can get discouraged quickly.

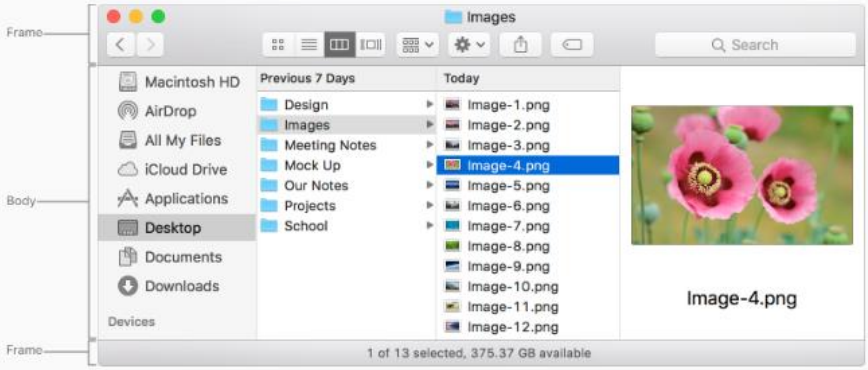


Let people make choices whenever possible. Consider using a table, pop-up button, or set of radio buttons instead of a text field. The ability to choose from a list of predefined options rather than type a response makes data entry quicker and more efficient.

- macOS
- App Architecture
- User Interaction
- System Capabilities
- Visual Design
- Icons and Images
- **Windows and Views**
  - **Window Anatomy**
  - Alerts
  - Boxes
  - Column Views
  - Dialogs
  - Image Views
  - Outline Views
  - Panels
  - Popovers
  - Scroll Views
  - Sheets
  - Sidebars
  - Split Views
  - Tab Views
  - Table Views

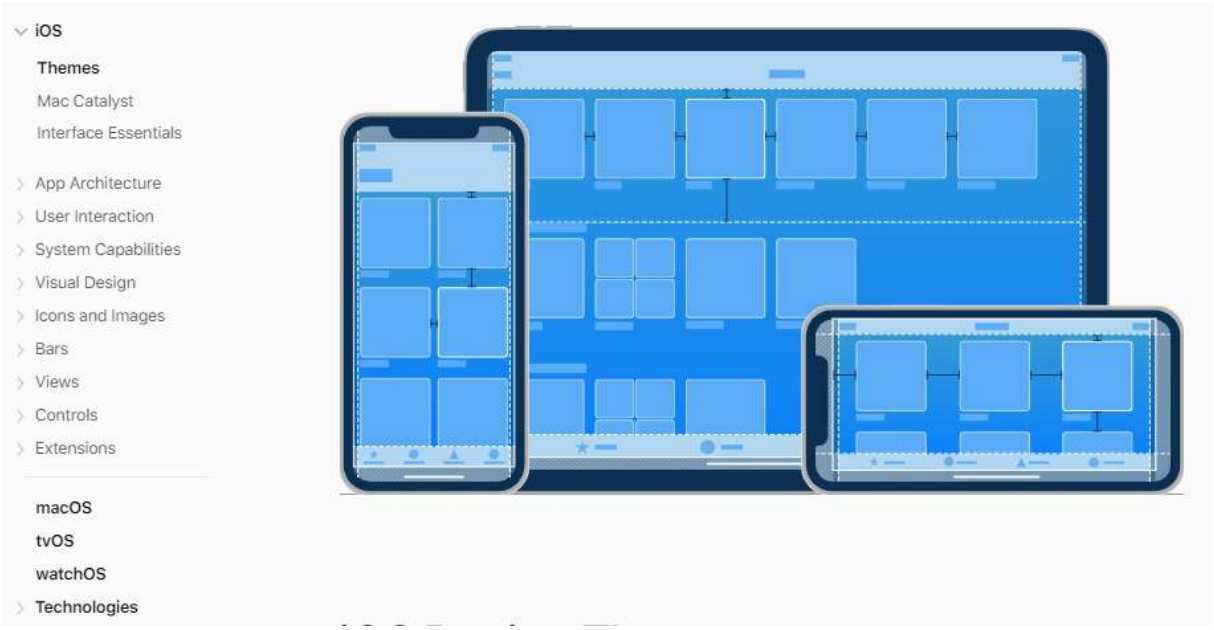
Window Anatomy

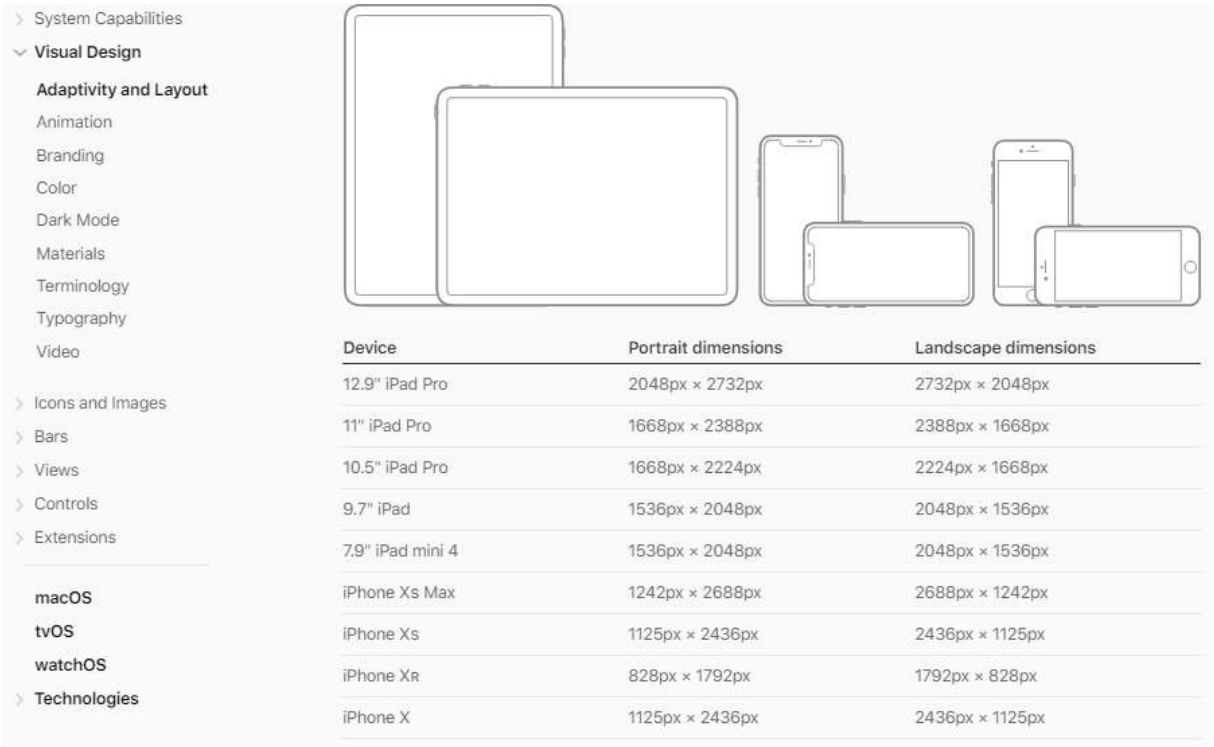
A window consists of a frame area and body area that let the user view and interact with content in an app. A window can appear onscreen alongside other windows, or it can fill the entire screen (see Full-Screen Mode). In a window that's not full-screen, the user can click and drag the frame to reposition the window on screen. Users can also click and drag the edges of the window to resize it, if the window supports resizing.



For developer guidance, see NSWindow.

macOS
App Architecture
User Interaction
System Capabilities
Visual Design
Icons and Images
Windows and Views
Menus
Menu Anatomy
Contextual Menus
Dock Menus
Menu Bar Menus
Buttons
Fields and Labels
Selectors
Indicators
Touch Bar
Extensions
iOS
tvOS
watchOS
Menu Anatomy
A menu presents a list of items—commands, attributes, or states—from which a user can choose. An item within a menu is known as a menu item, and may be configured to initiate an action, toggle a state on or off, or display a submenu of additional menu items when selected or in response to an associated keyboard shortcut. Menus can also include separators, and menu items can contain icons and symbols, like checkmarks. By default, all menus adopt translucency.
Safari
File
Edit
View
History
Bookmarks
Window
Help
New Window ⌘N
New Private Window ⇧⌘N
New Tab ⌘T
Open File... ⌘O
Open Location... ⌘L
Close Window ⇧⌘W
Close All Windows ⌥⇧⌘W
Close Tab
Save As... ⇧⌘S
Share
Export as PDF...
Open in Dashboard...
Import Bookmarks...
Export Bookmarks...
Print... ⌘P
Messages
Notes
Reminders
Email Link to This Page ⇧⌘I
AirDrop
More...


iOS
Themes
Mac Catalyst
Interface Essentials
App Architecture
User Interaction
System Capabilities
Visual Design
Icons and Images
Bars
Views
Controls
Extensions
macOS
tvOS
watchOS
Technologies



Device	Portrait dimensions	Landscape dimensions
12.9" iPad Pro	2048px × 2732px	2732px × 2048px
11" iPad Pro	1668px × 2388px	2388px × 1668px
10.5" iPad Pro	1668px × 2224px	2224px × 1668px
9.7" iPad	1536px × 2048px	2048px × 1536px
7.9" iPad mini 4	1536px × 2048px	2048px × 1536px
iPhone Xs Max	1242px × 2688px	2688px × 1242px
iPhone Xs	1125px × 2436px	2436px × 1125px
iPhone Xʀ	828px × 1792px	1792px × 828px
iPhone X	1125px × 2436px	2436px × 1125px

Mas información: **https://developer.apple.com/design/**

Apunte 6

Conceptos Fundamentales de la Codificación de Software

1. Introducción

Habitualmente la bibliografía de Ingeniería de Software suele pasar por alto la etapa de codificación. Seguramente entendiendo que es una actividad técnica que contempla una problemática específica y, por lo tanto, que es abarcada por toda otra gran rama de la literatura: aquella que se dedica específicamente a la problemática de la codificación y a la enseñanza de algún lenguaje de programación.

De hecho, esta división también ocurre en la Licenciatura en Sistemas: en la materia "Ingeniería de Software" se pasa por alto este tema, ya que hay 2 materias específicas que abordan esta problemática: "Teoría de los Lenguajes y Algoritmos" y "Construcción de Aplicaciones Informáticas".

La idea de este apunte es conocer, no en profundidad sino a título meramente informativo, en que consiste la etapa de Codificación, pensando especialmente en aquellos estudiantes que no cuentan con conocimiento previo en programación, ni han cursado las materias anteriormente mencionadas.

2. La etapa de codificación de software

¿Qué es la codificación de software?

Es la etapa donde se interpretan los requerimientos que los diseñadores expresaron al detallar la arquitectura del sistema, la especificación de la base de datos, la especificación de la interfaz y la especificación de los componentes y, a partir de ellos, desarrollar los algoritmos y los programas que se ejecutarán la computadora. **En esta etapa se programa y construye el software propiamente dicho.**

Es importante destacar que la codificación no se trata solo de escribir líneas de código, sino también de seguir buenas prácticas de programación, como la modularización, la legibilidad del código, la reutilización de componentes y la adopción de estándares de codificación. Esto ayuda a asegurar la calidad, la mantenibilidad y la escalabilidad del software.

¿Quién la hace?

Esta tarea es desarrollada por programadores o desarrolladores. Deben tener habilidades para interpretar los diagramas que componen el diseño y **para codificarlo en algún lenguaje de computación**.

¿Cuáles son los pasos?

Reciben e interpretan los diagramas del diseño, elaboran algoritmos y diagramas que le ayudan a resolver el paso a paso de lo que necesitan realizar, codifican el programa usando algún lenguaje, lo documentan, lo ejecutan en la computadora y, por último, verifican, que funcione sin errores.

¿Cuál es el producto final?

Uno o varios programas ejecutables, bases de datos, archivos de documentación, archivos adicionales que se requieran para el funcionamiento del programa (librerías externas, imágenes, videos, et.), más toda aquella documentación e instrucciones de instalación; para que el sistema pueda ser ejecutado en el entorno operativo del cliente.

¿Cómo se asegura que el programa no tenga errores?

Una vez que el software ha sido programado, antes de su liberación para su uso, se lleva a cabo una etapa de prueba o testeo. Durante esta fase, el programa se somete a condiciones extremas de funcionamiento con el fin de identificar posibles errores. Esta etapa de testeo no reemplaza la prueba inicial realizada por el programador, sino que la complementa. En el caso de desarrollos grandes, las empresas suelen contar con un departamento de testeo independiente, también conocido como aseguramiento de la calidad, que cuenta con especialistas dedicados a esta tarea.

3. ¿Qué tareas realiza un programador?

La primera y más obvia: programar

Para esto tendrá que aprender como programar en un lenguaje de computación, operar con diferentes tipos de archivos y bases de datos, conocer de redes y de información distribuida, interactuar con uno o varios dispositivos, incluyendo tiendas de aplicaciones y sitios para poner código a disposición de quien lo necesite usar.

Si bien los programadores suelen especializarse en algún lenguaje especifico, generalmente poseen habilidades y conocimientos para utilizar más de uno. Esto presenta una ventaja a adicional: la posibilidad de construir algún módulo o rutina específica utilizando un leguaje que sea más apropiado para dicha tarea.

Por otro lado, últimamente han ido tomando mayor relevancia nuevas herramientas que permiten tanto a los desarrolladores, como analistas e, incluso a usuarios finales, programar sus propias aplicaciones a través de interfaces gráficas. Este “movimiento” de **no-code** (sin código) o **low-code** (poco código) promete aumentar significativamente el universo de personas que pueden automatizar nuevos flujos de datos. Incluso desarrollar componentes o aplicaciones completas. En lugar de escribir todo el código desde cero, los desarrolladores utilizan una interfaz gráfica de usuario (GUI) y herramientas visuales para construir aplicaciones arrastrando y soltando componentes predefinidos y configurando opciones.
Algunas características clave de la programación de bajo código incluyen:

- **Entorno visual**: Los desarrolladores trabajan en un entorno visual, utilizando herramientas visuales como editores de flujo de trabajo, generadores de formularios y constructores de interfaces de usuario para diseñar y configurar la lógica de la aplicación.

- **Componentes reutilizables**: Las plataformas de bajo código proporcionan una biblioteca de componentes predefinidos y reutilizables, como botones, campos de entrada, tablas

y elementos de navegación, que se pueden combinar y configurar según las necesidades de la aplicación.

- **Automatización**: La programación de bajo código permite la automatización de tareas comunes, como la generación de código, la creación de interfaces de usuario y la gestión de bases de datos. Esto reduce la cantidad de código manual requerido y acelera el desarrollo.

- **Integración de sistemas**: Las plataformas de bajo código suelen incluir capacidades de integración que permiten conectar la aplicación con sistemas externos, como bases de datos, servicios web y API. Esto facilita la integración de datos y funcionalidades de otros sistemas en la aplicación desarrollada.

La programación low code es especialmente útil en escenarios donde se requiere desarrollar aplicaciones rápidamente, como prototipos, aplicaciones internas o soluciones empresariales simples. También puede ser beneficioso para desarrolladores con menos experiencia en programación, ya que les permite crear aplicaciones funcionales sin tener que dominar lenguajes de programación complejos.

Sin embargo, es importante tener en cuenta este tipo de desarrollos pueden tener limitaciones en términos de flexibilidad y personalización. En aplicaciones más complejas o que requieren lógica de negocio altamente personalizada, es posible que se necesite un enfoque de programación tradicional más manual.

Comprender los requerimientos

Salvo excepciones o desarrollos muy chicos, el programador no es quien realiza directamente la recolección de requisitos que plantean los usuarios.

Como ya se explicó, los analistas recopilan los requerimientos. Luego los diseñadores son quienes plantean conceptualmente la solución informática, utilizando diferentes técnicas, diagramas y documentos, que contienen el detalle de lo que se deberá programar. En cierto modo, es el equivalente al plano del sistema.

Y así como el constructor debe saber leer un plano para comprender que debe construir, un programador debe poder interpretar esos documentos para entender que es lo que tiene que programar.

Documentar

La documentación es una parte crucial de la programación, independientemente del enfoque utilizado. La documentación adecuada tiene varios propósitos:

1. **Facilitar la comprensión del código**: La documentación sirve como una guía para que otros desarrolladores (y el propio programador) puedan entender el propósito y la funcionalidad código. Se busca comentar la lógica subyacente, los algoritmos utilizados

y las decisiones de diseño. De este modo se consigue un código sea más legible y comprensible.

2. **Permitir el mantenimiento y la modificación**: Si alguien más o el propio programador necesita realizar modificaciones o correcciones en el código en el futuro, la documentación bien estructurada les proporcionará información clave sobre cómo funciona el código y cómo interactúa con otros componentes. Esto facilita el proceso de mantenimiento y evita posibles errores o malentendidos.

3. **Mejorar la colaboración en equipo**: En entornos de desarrollo colaborativo, la documentación clara y concisa fomenta una comunicación eficiente entre los miembros del equipo. Al proporcionar descripciones detalladas de las funciones, interfaces y dependencias, se facilita la colaboración y se evitan malentendidos.

4. **Apoyar la reutilización de código**: La documentación adecuada permite al programador recordar y reutilizar su propio código en futuros proyectos. Al describir las funcionalidades y las interfaces del código, se facilita su comprensión y uso posterior.

La documentación puede incluir diferentes elementos, como comentarios en el código, archivos de documentación separados, descripciones de funciones y clases, ejemplos de uso y diagramas de flujo. La elección de las herramientas y formatos de documentación puede variar según las preferencias del equipo de desarrollo y los estándares de la industria.

Respetar estándares y utilizar patrones

La programación de software tiene mucho de arte, de impronta personal. Casi con seguridad, dos programadores que construyen el mismo software lo harán de modo diferente.

En una empresa o en un desarrollo colaborativo esto debe minimizarse. Los sistemas deben responder a determinados estándares e integrarse con otras aplicaciones que estén funcionado. La utilización de patrones no solo garantiza un desarrollo homogéneo, sino que también mejorará la calidad del código, ya que permite utilizar prácticas ya probadas.

Testear

No existe el programa perfecto. Es casi imposible que un código recién programado funcione sin ningún error y, por esto, es necesario que el programador haga la primera prueba. Pero testear no es comprobar que el programa funcione sino forzarlo y llevarlo a situaciones extremas para que falle.

Esto, que parece una obviedad, no lo es tal. Quien pasó horas tratando de desarrollar un código buscará que su producto ande bien, y lo testeará con este objetivo. Debe entender que el éxito de probar software es encontrar errores y no asegurar que el programa funcione. El programador debe tener claro que las fallas que no encuentre, las encontrará el equipo de testeo o, peor aún, el usuario.

Mantener el software existente

El software evoluciona en el tiempo. Aparecen errores que hay que corregir y nuevas funcionalidades que son necesarias agregar. Entonces, una de las tareas que debe desempeñar un programador es la de mantener actualizados sus propios programas. Eventualmente, también deberá ocuparse de algún otro software de la compañía, aunque no haya sido quien lo construyó. Como se dijo, la documentación del propio código, la adopción de patrones ya definidos, la reutilización de componentes y emplear un estilo de programación simple y claro, facilitarán esta tarea.

Llevar un adecuado control de versiones

Es frecuente que un de un mismo programa exista más de una versión. Además de la que está en funcionamiento, puede estar trabajando en nuevas versiones que se lanzarán próximamente, o incluso tener varias en simultáneo, para ser utilizadas en diferentes sistemas operativos, idiomas o por diferentes clientes.

También es habitual que el desarrollador que tenga más de un espacio de trabajo (la computadora de la oficina y en su casa, por ejemplo). Si bien hoy en día es simple sincronizar dispositivos, debe prestar especial cuidado en que no se pisen archivos.

A la hora de entregar un programa para que sea puesto en funcionamiento, es necesario que se asegure que el mismo contiene los componentes adecuados y actualizados que corresponden a la versión que esté liberando para su uso.

Por supuesto la organización también tendrá que establecer protocolos de trabajo para asegurar una eficaz gestión del versionado[36].

Resguardar su trabajo

Además de poseer copias de resguardo personales, debe asegurarse que todo su código se encuentra resguardado y almacenado en los espacios corporativos que estén protegidos por las copias de respaldo de la organización. Un buen backup debe brindar protección a las siguientes situaciones:

1. **Protección contra pérdida de datos**: La copia de seguridad debe permitir recuperar código perdido debido a un fallo del sistema, un error humano o un desastre.

2. **Recuperación de versiones anteriores**: El desarrollo de software implica cambios y actualizaciones constantes en el código. Existen ocasiones en los que puede ser necesario volver a una versión anterior del código en caso de encontrar problemas o errores en la versión actual. Las copias de seguridad deben permitir recuperar versiones anteriores del código y facilitan la resolución de problemas.

[36] Podrá consultarse apunte de “Despliegue del Software” para un mayor detalle sobre el versionado de Software.

3. **Cumplimiento normativo y legal**: Dependiendo del ámbito de aplicación y las regulaciones vigentes, puede ser obligatorio tener copias de seguridad del código fuente. Esto es especialmente importante en industrias como la banca, la salud o donde la integridad y disponibilidad de los datos son fundamentales. Un ejemplo de esto son los dispositivos de voto electrónico, donde el código fuente debe poder ser auditado en el futuro ante cualquier denuncia o anomalía.

Comunicarse

Si bien el desarrollo de código es una tarea que, por lo general, se realiza en forma individual, el programador debe tener contacto fluido con otros programadores del equipo y también con testadores, que le van a reportar los errores que deberá corregir. Por otra parte, por más que existan documentos formales, el diálogo con analistas y diseñadores, no está prohibido.

Si bien hemos dicho que un programador podría trabajar en forma autónoma solamente interpretando el diseño, lo cierto es que en ocasiones su experiencia en cómo se resuelven los problemas técnicos es muy valiosa. Puede sugerir mejores formas de hacer algo, proponer funciones que técnicamente son sencillas y no se les han ocurrido a los diseñadores, o también indicar que ciertas cosas que se han pedido tienen restricciones técnicas que impiden ser construidas, al menos en el tiempo y costo presupuestado.

Por supuesto también tendrá que comunicarse con el líder de proyecto para informarle grados de avance y potenciales problemas que afecten la calendarización del proyecto (es decir el tiempo de entrega prometido)

Por último, si bien el programador y el usuario suelen estar en puntos opuestos y hablar en términos diferentes, muchas veces la explicación directa del programador de cómo puede operar determinada parte de la aplicación suele ser oportuna. También escuchar de primera mano algún requerimiento puede ayudar a que entre ambos encuentren un modo diferente y novedoso de resolver algún problema.

Acudir a fuentes fiables de información

Internet es el repositorio de código más grande que jamás se ha construido. Los programadores pueden buscar sitios confiables donde hallar soluciones y mejoras para sus programas. Puede ingresar a foros de consulta entre pares, leer consejos de expertos, compartir su código para que la comunidad de programadores lo mejore (así funcionan gran parte de los desarrollos de software libre), buscar y utilizar código de terceros ya hecho y de libre disponibilidad. Incluso pueden adaptar y usar componentes de experiencia completa de terceros (por ejemplo, un carrito de compras o un botón de pago), que luego personaliza para usarlo en su sitio web.

Obviamente, cada componente disponible cuenta con sus respectivas condiciones de uso. Es importante no plagiar ni vulnerar derechos de autor y contar siempre con todas las licencias y los derechos de todo lo que se integre al final.

Cualquiera sea el caso, siempre debe validar que origen sea fiable y reconocido. El código descargado deber ser testeado para asegurar que no tienen rutinas ocultas, código malicioso, o que envía datos a terceros. Esto es especialmente importante cuando se utilicen componentes sobre los que no se tiene acceso al código fuente.

Tender a la excelencia

El programador tiene la responsabilidad de buscar siempre la mejor técnica para resolver los problemas, optimizando los recursos y el tiempo de procesamiento. No debe tener miedo de refactorizar su propio código cuando encuentre formas más eficientes de resolver algoritmos o mantenerlo actualizado según las mejoras que la industria ofrezca para favorecer su desarrollo.

Tener pasión

Programar es un desafío. Es maravilloso cuando uno ejecuta un programa que acaba de codificar y este hace lo que se previó sin problemas. La capacidad de controlar y dar vida a la máquina a través del código es única y fascinante.

Sin embargo, también es cierto que los desafíos y dificultades son parte inherente de la programación. A veces, los algoritmos no se comportan como se espera, y pueden surgir errores inesperados o problemas aparentemente insolubles. En esos momentos, los programadores se enfrentan a horas de trabajo intenso y a veces largas jornadas, ya que la pasión por lo que hacen y el deseo de superar los obstáculos los impulsan a seguir adelante., a veces sin horarios ni feriados.

Esto pone en aprietos a las organizaciones que no suelen estar preparadas para gente que a veces necesita quedarse toda la noche trabajando, o que no puede cortar arbitrariamente a las 5 de la tarde. Es fundamental fomentar una cultura de colaboración y apoyo entre los equipos de trabajo. Los jefes y colegas deben comprender y aceptar que los programadores pueden necesitar consultar o solicitar ayuda fuera de los horarios laborales regulares, especialmente en situaciones urgentes o críticas. La comunicación abierta y el establecimiento de canales efectivos de comunicación pueden facilitar la resolución de problemas y garantizar una respuesta oportuna

En el caso de programas críticos que requieren monitoreo las 24 horas del día, los 365 días del año, es común implementar turnos de guardia activa y pasiva para asegurar un soporte continuo. Esto implica que los programadores se turnen para estar disponibles y responder rápidamente en caso de que surjan problemas o emergencias relacionadas con el software.

El teletrabajo es una realidad. Pueden trabajar desde cualquier lugar del mundo y para cualquier persona que los contrate, no importa donde estén.

Muchas veces es necesario prever relaciones laborales diferentes a otras miembros de la compañía.

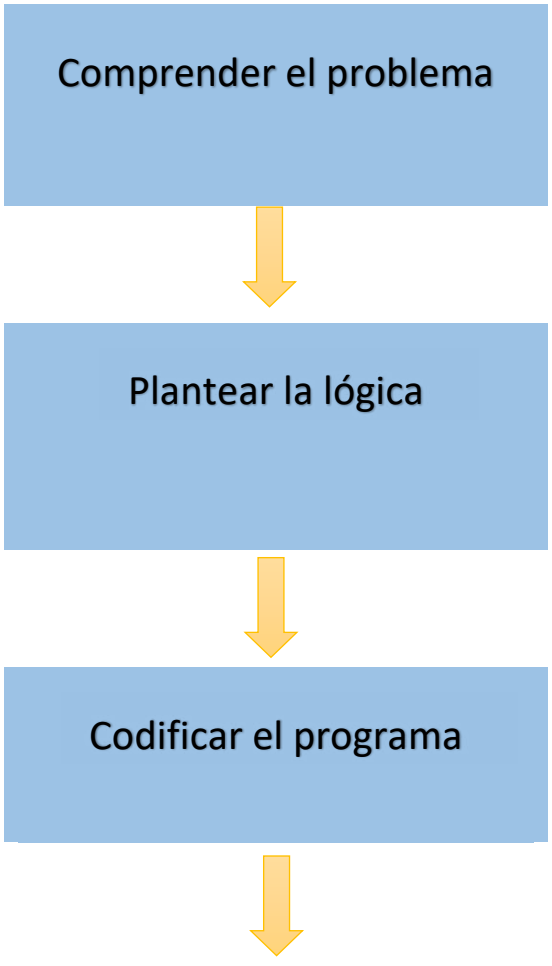
Respetar la ética

Los programadores tienen acceso casi ilimitado a la información de la empresa. Son ellos mismos los que programan las restricciones de seguridad y las rutinas de codificación de archivos. Algunos, incluso, tienen claves maestras que le permiten acceso ilimitado a cualquier parte de los sistemas. Se espera entonces que actúen éticamente y respetando no solo nomas legales, sino las políticas específicas sobre privacidad, resguardo siempre información a la que tenga acceso.

Capacitarse continuamente

Las computadoras evolucionan, los lenguajes también. Cada vez hay mayor capacidad para desarrollar aplicaciones y el programador debe aprovecharlas. Para ello es necesario que se mantenga actualizado y que se capacite en forma constante.

4. EL proceso de la programación



Al comenzar, si todo se ha hecho aplicando los procesos de la ingeniería del software, el programador recibe un diseño con el suficiente detalle para poder comprender el problema que debe resolver.

En programador deberá entonces plantear los algoritmos que le permitirán resolver el problema. Usualmente utiliza diagramas o seudocódigo, que le permiten pensar primero “en papel” y conceptualmente, como será el programa por codificar.

Una vez que el problema está resuelto conceptualmente, lo codificará utilizando algún lenguaje de computación.

Traducir el programa a lenguaje máquina

Los programadores utilizan lenguajes de programación que están diseñados para ser cercanos al lenguaje natural, como el inglés, lo que facilita la escritura del código. Sin embargo, la computadora solo entiende lenguaje de máquina, que consiste en una secuencia de instrucciones binarias. Por lo tanto, es necesario realizar una traducción del código escrito en lenguaje de programación a lenguaje de máquina para que la computadora pueda ejecutarlo.

Esta traducción se lleva a cabo de manera automática mediante el uso de herramientas como compiladores o intérpretes. Un compilador se encarga de traducir todo el código fuente a lenguaje de máquina de una vez, generando un archivo ejecutable que la computadora puede entender y ejecutar. Por otro lado, un intérprete traduce y ejecuta el código línea por línea a medida que se va ejecutando el programa.

Estas herramientas, integradas en el entorno de desarrollo, facilitan la tarea de traducción y permiten que los programadores puedan enfocarse en la lógica y estructura del programa sin preocuparse por los detalles de la traducción a lenguaje de máquina.

Probar el programa

Finalmente, es necesario realizar pruebas exhaustivas para garantizar que el programa esté libre de errores y resuelva el problema planteado. Una vez que ha pasado por las pruebas internas, el sistema está preparado para la etapa de testeo, donde un equipo independiente volverá a poner a prueba el software para asegurar su calidad y funcionalidad.

5. La elección del lenguaje de programación

Una pregunta obvia que se desprende al analizar las tareas de programación es ¿Qué lenguaje de programación se debe utilizar? y, atado a esta, ¿Quién elige dicho lenguaje?

La respuesta también parece obvia: El responsable técnico del proyecto debe elegir aquel lenguaje que mejor se adapte al desarrollo, conforme las características que este debe poseer. Por

ejemplo, si el desarrollo fuera una App para celular, seguramente se utilizaría un lenguaje diferente al que elegiríamos si necesitamos programar un sistema de conexión remota a una base de datos o una aplicación que corra sobre una computadora de escritorio sin conexión a internet.

Sin embargo, esta respuesta obvia no siempre es correcta, o no siempre puede elegirse de tal modo. Los proyectos pequeños, o las empresas de desarrollo más chicas, suelen contar con un grupo reducido de programadores y, por lo general, estos se especializan en uno o dos lenguajes, pero no en todos. Ya no se puede elegir el mejor, sino el que los programadores conozcan.

De igual modo, las organizaciones más grandes fijan sus propios estándares. Estos indican, entre otras cosas, que lenguajes y que bases de datos deben utilizarse. Nuevamente, quien realice un proyecto para esa organización, tendrá una elección condicionada. Por supuesto, esto es válido siempre y cuando sea un desarrollo interno, o un desarrollo sobre el cuál la empresa adquiere también los programas fuentes y la posibilidad de modificarlos. En los desarrollos del tipo "llave en mano", donde se recibe un sistema listo para usar, en realidad no importa en que lenguaje fue desarrollado, sí que cumpla con otros estándares, por ejemplo, que funcione en determinados servidores, con determinada base de datos, o que comparta información con otras aplicaciones de la compañía.

En sistemas grandes, con módulos de diversas funcionalidades, es posible que no exista un único lenguaje que sea "el mejor" para todos los usos. Y en este caso quizá sea necesario combinar más de uno dentro del mismo sistema. Por ejemplo, podría utilizar un lenguaje para programar la interfaz web, y otro distinto para interactuar con algún dispositivo que deba controlar, como ser un lector de huellas digitales.

Los desarrollos previos y el contenido reutilizable también condicionan la elección. Si resulta que ya tengo un 60% de los módulos ya construidos, seguramente va a ser mejor idea programar lo que falta con el mismo lenguaje de desarrollo.

El hardware también influye. En las arquitecturas x86 (refiere genéricamente a las PCs y servidores) hay una gran cantidad de lenguajes disponibles para elegir. Pero solo algunos de ellos pueden ser utilizados también en arquitecturas ARM (celulares o Raspberry). Y si la organización tiene algún mainframe (bancos, por ejemplo) el catálogo disponible queda reducido, a veces, a una única opción.

Al igual que el hardware, los sistemas operativos también imponen sus condicionamientos. Hay lenguajes e IDEs [37] que solo están disponibles para Linux, Android, o para Apple, como también los hay multiplataforma, como por el ejemplo Microsoft Visual Studio.

Y al mencionar a Visual Studio también hay que hablar del licenciamiento. Este IDE, al igual que otros, no son totalmente gratuitos. Proveen algunas funcionalidades básicas sin costo, pero otras más avanzadas que requieren contar con una licencia. Si una organización tiene licenciado

[37] IDE: Entorno de Desarrollo Integrado. Es una aplicación informática que provee al desarrollador de software de una serie servicios que facilitan su tarea de programar. Habitualmente cuentan con un editor de un editor de código fuente, optimizado para uno o varios lenguajes, herramientas de construcción automáticas y depuradores de código. Algunos también proveen asistentes y patrones predefinidos para facilitar y automatizar determinadas tareas, como la construcción de pantallas y listados.

algún entorno, seguramente tratará de que todas aplicaciones se desarrollen utilizándolo, junto con los lenguajes que provee.

Es interesante mencionar que, por lo general, un mismo programa realizado en el mismo lenguaje, puede comportarse de modo diferente en un sistema operativo que en otro. Incluso un mismo código puede funcionar también forma distinta, conforme que IDE se utilice. Esto es porque, ocasionalmente, las librerías adicionales que el programa consume pueden tener variaciones. La programación multiplataforma requiere entonces o bien cambiar el código para cada una de ella, o bien utilizar alguna plataforma de desarrollo que, ante un código genérico, prepara automáticamente las versiones para cada una de las plataformas a utilizar. Eclipse, Genexus o el propio Visual Studio son algunos ejemplos. Otra alternativa es que el software no se ejecute directamente, sino utilizando una máquina virtual. De este modo, cualquier dispositivo que ejecute dicha máquina virtual estará en condiciones de correr el programa. Así, por ejemplo, funcionan las aplicaciones JAVA y su Java Virtual Machine.

Atado a esto, los lenguajes de programación y las IDEs también evolucionan en el tiempo. Van apareciendo nuevas versiones que agregan funcionalidades y los hacen cada vez más potentes. Por lo general, no hay compatibilidad de código, en especial hacia atrás. Con el tiempo, algunos lenguajes se vuelven obsoletos y aparecen otros mejor preparados para los nuevos desafíos. Es decir que también existen condicionantes por ese camino: No solo hay que elegir el lenguaje, sino que versión y que IDE se utilizará.

En resumen… claramente la elección del lenguaje es una decisión principalmente técnica. El responsable técnico, seguramente con la colaboración los programadores, será quién deberá decidir cuál es el mejor lenguaje para desarrollar un determinado sistema. Pero esta decisión no será libre ni exclusivamente basada en las ventajas o desventajas que un lenguaje tiene para en nuevo desarrollo, sino condicionada por múltiples factores e imposiciones.

6. Asistentes de Inteligencia Artificial

Hasta el momento, las inteligencias artificiales desarrolladas están lejos de poder generar software de manera autónoma, al menos no como una aplicación completa y funcional. El futuro determinará si esto finalmente sucede. En el pasado, se han hecho muchas predicciones al respecto, pero ninguna se ha cumplido. Con el poder de cómputo actual, las inteligencias artificiales están ganando cada vez más capacidad, y resulta difícil predecir hasta qué punto podrán llegar.

Lo cierto es que, actualmente, es cada vez más habitual que los programadores utilicen asistentes o copilotos basados en inteligencia artificial para hacer más fácil su tarea. Una encuesta reciente a 500 desarrolladores profesionales de GitHub[38], mostró que 9 de cada 10 de ellos usaban herramientas de programación basadas en IA para su trabajo cotidiano.

[38] GitHub es una plataforma web, propiedad de Microsoft, que permite a desarrolladores almacenar, gestión y hacer colaborativos sus proyectos de desarrollo de software utilizando el sistema de control de versiones Git. Entre otros, proporciona un entorno en línea donde los desarrolladores pueden alojar y compartir su código, colaborar en equipo, realizar un seguimiento de los cambios, revisar el código y gestionar las tareas de desarrollo. Puede utilizarse en forma gratuita, aunque tiene diferentes planes pagos con funcionalidades extras.

Ya hemos visto anteriormente las ventajas que puede ofrecer la programación en parejas, donde un programador revisa y colabora en el código construido por otro, evitando errores, sugiriendo mejoras y, en última instancia, mejorando la calidad del código. En la actualidad, esta tarea puede ser desempeñada por asistentes de IA.

¿Cuáles son los beneficios de contar con un asistente IA? Su utilización mejora significativamente la productividad y eficiencia de los programadores de software. Ente las posibilidades de estos asistentes podemos enumerar:

1. **Autocompletar código**: Si se utilizan editores de código asistidos por IA; estos pueden sugerir automáticamente fragmentos de código o completar líneas basándose en el contexto y los patrones aprendidos de grandes repositorios.

2. **Generación de código**: Aunque con limitaciones, es posible dar instrucciones en lenguaje natural y que el código se genere automáticamente, en el lenguaje de programación que se elija.

3. **Traducción o cambio de lenguaje**: Es posible convertir piezas de código desarrolladas en un lenguaje de computación a otro distinto.

4. **Prueba, depuración y detección de errores**: Mediante el aprendizaje automático, puede analizare el código, detectar e identificar posibles errores o anomalías, y proporcionar sugerencias para solucionarlos. Tambien pueden ejecutarse pruebas automatizadas, con casos especialmente preparados, facilitando de este modo el testeo de software.

5. **Documentación automática**: Utilizando modelos de lenguaje natural, pueden generar comentarios de código y documentación de un de modo automático.

Es importante destacar que los asistentes de código utilizan algoritmos de Machine Learning (aprendizaje automático) que les permiten evolucionar con el tiempo. A medida que más programadores los utilicen y analicen más código, estos algoritmos se vuelven más completos y poderosos. En la actualidad, su objetivo principal no es reemplazar al programador, sino facilitar su trabajo. En cuanto a lo que depara el futuro, es difícil predecirlo con certeza.

7. Referencias

Centro de Formación de Alto Rendimiento en programación y Tecnología. https://keepcoding.io/

Ejemplos de asistentes IA. La lista no pretende recomendar asistentes ni mucho menos es completa, simplemente se adjunta a modo de referencia para quien desee profundizar sobre el tema.

GitHub Copilot	https://github.com/features/copilot
Tabnine	https://www.tabnine.com/
Chatgpt	https://chat.openai.com/
Amazon Guru	https://aws.amazon.com/es/codeguru/

OpenAI Codex ChatGPT	https://openai.com/blog/openai-codex https://chat.openai.com/
Kite	https://www.kite.com/
IBM Watson Code Assistant (Watson X)	https://www.ibm.com/products/watson-code-assistant

Apunte 7

Conceptos Fundamentales de las Pruebas de Software

1. Introducción

El proceso de prueba o testeo es indudablemente una herramienta fundamental para garantizar la calidad del software. Realizar pruebas exhaustivas en todos los componentes de un sistema, con el objetivo de detectar errores y defectos, es una tarea compleja pero extremadamente necesaria. Prácticamente no existen registros históricos de sistemas que hayan sido programados de manera perfecta y sin errores, por lo tanto, es absolutamente imprescindible identificar problemas y solucionarlos antes de que el software entre en funcionamiento en su versión final.

En la etapa de prueba del software, es importante cambiar la perspectiva y entender que el objetivo **no es simplemente comprobar que el sistema funcione, sino someterlo a situaciones extremas y forzar su falla**. La premisa básica es que un programa puede parecer funcionar correctamente en condiciones normales, pero puede presentar fallos cuando se lo somete a situaciones inusuales o se le ingresan datos erróneos.

Bajo este enfoque, el presente apunte busca resaltar algunos aspectos relevantes para tener en cuenta durante el proceso de prueba de software, más allá de la aplicación de diferentes tipos de pruebas y técnicas utilizadas por los responsables del testeo. Se busca destacar la importancia de explorar y someter al sistema a diversas situaciones límite para identificar posibles errores y asegurar así la calidad del software.

2. Buscar que falle, no probar que funcione

Usualmente, la etapa de prueba de software es definida como aquella en la cual se comprueba que el software no tiene errores, que cumple con los requisitos del cliente y, por lo tanto, puede ser liberado y puesto en operación. Esta definición tiene mucho de cierto, pero no es del todo apropiada.

El objetivo del tester[39], de software no es simplemente comprobar que el software funcione correctamente, sino buscar activamente situaciones en las que el software falle o presente errores. Es una tarea que va más allá de probar que el software "ande bien" en condiciones normales, ya que se busca identificar y reportar cualquier fallo, defecto o comportamiento inesperado que pueda surgir. Debe buscar las limitaciones del software, ponerlo a prueba en diferentes escenarios, introducir datos erróneos, realizar acciones inusuales y explorar situaciones extremas con el objetivo de descubrir errores ocultos, identificar puntos débiles y asegurar que el software cumpla con los requisitos y expectativas establecidas.

Una de las figuras más influyentes de la generación fundadora de la ciencia de la computación, Edsger Dijkstra enunció allá por 1972: "*Las pruebas pueden mostrar solo la presencia de errores, mas no su ausencia*". Esta afirmación destaca la naturaleza limitada de las pruebas de software y refuerza el concepto de que busca errores.

[39] La palabra "Tester" o "Testeador" no forman parte de lenguaje castellano, aunque se utilizan cotidianamente con varios usos. En este apunte hace referencia a la persona encargada de llevar a cabo las pruebas y testeos del software.

El programador, desde luego, trata de producir código sin errores. Su éxito es entregar aplicaciones sin fallas. El tester, por el contrario, asume la premisa de que el software tiene errores, sin importar que tan bien haya sido programado. Su tarea es encontrarlos antes que lleguen al usuario final. El éxito en su tarea es **encontrar la mayor cantidad de errores posibles, en el menor tiempo y con el menor costo**. Para esto, literalmente ataca al programa desde varios flancos y con diversas estrategias, para forzarlo a fallar. **El mejor tester es aquel que más errores encuentra**, no aquel que determina que el software funciona.

Un ejemplo claro de esta filosofía de pruebas exhaustivas es la evaluación que se realiza a los automóviles antes de su lanzamiento al mercado. En lugar de conducir el vehículo en condiciones normales y a velocidades controladas, se somete a pruebas extremas para poner a prueba su desempeño en situaciones límites. Durante estas pruebas, el automóvil se enfrenta a terrenos irregulares, se le exige en pendientes pronunciadas, se evalúa su capacidad de frenado en condiciones adversas como lluvia o nieve, e incluso se realizan pruebas de impacto para verificar la resistencia de los componentes y el diseño en caso de colisión. Estas pruebas tienen como objetivo llevar al límite al automóvil y descubrir cualquier defecto o falla. En los famosos test de choque (crash test), el vehículo se somete a una colisión controlada que puede llegar a destruirlo casi por completo. Aunque parezca contradictorio, esta destrucción planificada proporciona valiosa información a los ingenieros, ya que les permite conocer los límites del automóvil y garantizar que supere ampliamente las condiciones normales de uso.

En la industria de los electrodomésticos también se aplican rigurosas pruebas de calidad para garantizar su funcionamiento y durabilidad. Un ejemplo común es la prueba de apertura de las puertas de las heladeras. En lugar de simplemente verificar que las puertas se abran y cierren correctamente en condiciones normales, se someten a miles de ciclos de apertura y cierre repetitivos, hasta que eventualmente se rompan. El propósito de estas pruebas es evaluar la resistencia y durabilidad de los mecanismos de las puertas. La meta no es que la puerta supere las pruebas sin fallar, sino todo lo contrario: se busca detectar los puntos débiles del diseño, identificar los límites de resistencia y determinar si es necesario utilizar otros componentes o reforzar el diseño para asegurar un rendimiento óptimo durante el uso cotidiano. Solo al lograr que la puerta falle en estas pruebas se pueden tomar las medidas necesarias para mejorar su diseño y garantizar su funcionalidad a largo plazo.

De manera similar, en el ámbito del desarrollo de software, las pruebas tienen como objetivo forzar el software a situaciones extremas y descubrir cualquier error o comportamiento inesperado. Al igual que con los automóviles y las heladeras, estas pruebas rigurosas son fundamentales para asegurar la calidad del producto y brindar una experiencia confiable a los usuarios finales.

Por supuesto, es importante aclarar que ni el automóvil sometido a pruebas extremas ni la heladera con la puerta rota se comercializan. Estos prototipos se utilizan como referencia conceptual para la producción de productos similares que cumplen con las mismas condiciones de diseño y utilizan componentes similares a los que fueron sometidos a pruebas.

Cuando compramos una heladera, por ejemplo, es importante tener en cuenta que el modelo que adquirimos no ha pasado por los rigurosos test mencionados anteriormente. Sin embargo, se ha construido siguiendo los mismos estándares de diseño y utilizando componentes similares a los que fueron sometidos a pruebas exhaustivas.

Una ventaja en el ámbito del software es que, a diferencia de un automóvil o un electrodoméstico, cuando se encuentra un error durante las pruebas, es posible corregirlo y entregar al cliente una versión funcional y libre de fallos. Esta capacidad de realizar modificaciones y mejoras en el software antes de su entrega es una **característica fundamental** de este tipo de productos.

Cuando se descubren errores durante las pruebas de software, el equipo de desarrollo tiene la oportunidad de corregirlos y realizar las modificaciones necesarias para garantizar el correcto funcionamiento del programa. Una vez que se han realizado estas correcciones, se puede entregar al cliente una versión actualizada y depurada del software.

Además, es importante destacar que el cliente suele recibir la misma copia funcional del software que se ha sometido a pruebas. Esto significa que el producto final que recibe el cliente es una versión que ha sido validada y testeada, lo que proporciona mayor confianza en su funcionamiento.

No obstante, es importante tener en cuenta que, a pesar de los esfuerzos realizados durante el proceso de prueba y corrección de errores, es posible que en algunos casos se presenten problemas o fallos después de la entrega del software. Por esta razón, también deben implementar mecanismos de soporte y actualizaciones para abordar cualquier incidencia que pueda surgir en el uso continuado del programa.

3. La calidad y las pruebas

Las pruebas desempeñan un papel fundamental en el aseguramiento de la calidad del software (Software Quality Assurance o SQA), sin embargo, es importante destacar que las pruebas no son sinónimo de calidad por sí mismas. La calidad es un concepto más amplio que abarca todas las actividades del proceso de desarrollo de software.

Si bien las pruebas son esenciales para identificar y corregir errores, asegurando el correcto funcionamiento del software, la calidad implica mucho más que simplemente ejecutar pruebas. Involucra aspectos como el diseño y la arquitectura del software, la claridad y mantenibilidad del código, la usabilidad, el rendimiento, la seguridad y la satisfacción del cliente, entre otros.

Incluso la etapa de pruebas en sí misma debe ser sometida a evaluaciones y controles de calidad para garantizar su eficacia y fiabilidad. Esto implica llevar a cabo auditorías de pruebas, revisar la cobertura de pruebas, evaluar las técnicas utilizadas y analizar los resultados obtenidos. Estos procesos de auditoría y control de calidad en las pruebas ayudan a detectar posibles deficiencias y asegurar que las pruebas se realicen de manera adecuada y exhaustiva.

Al finalizar la etapa de testeos debería quedar demostrado que el software es confiable y seguro para ser operado. Pero para que esto ocurra se requieren varias cosas:

- Antes de comenzar cada tipo de prueba, se debe definir en que consiste la misma; en que entorno se realizará; que juego de datos se utilizaran y cuáles son las condiciones esperables para darla por finalizada y superada.

- Se deben realizar testeos y auditorias para verificar que en efecto se han realizado todas y cada una de las pruebas pautadas, cumpliendo con las definiciones de cada caso.

- Cada prueba debe tener un responsable, que firme y avale los testeos realizados.

- Debe documentarse todo el proceso de prueba y aprobación y poner estos resultados a disposición del usuario, ya que este que no tendrá la capacidad de realizar todas y cada una de las pruebas, sino que deberá confiar en estas certificaciones. En ocasiones, organismos de contralor controlan estos registros.

4. La prueba es la única actividad optativa en el proceso de desarrollo

¡El título de esta sección es peligroso, temerario, alocado, pero lamentablemente cierto!

Un software no puede prescindir de un análisis y diseño. Sin ellos no se sabría que construir. Mucho menos de la codificación, sin ella directamente no existiría el software. La puesta en marcha también es obligatoria si es que se quiere usar el sistema. Pero verdaderamente un software podría entregarse sin prueba, o con menos de la necesaria, acaso confiando en la suerte o en la excelencia de la programación.

¿Por qué alguien sería irresponsable al entregar software sin probarlo? ¿Por qué usaríamos software que sabemos tiene una alta probabilidad de fallar? Aunque no parezca tener una respuesta lógica, los aspectos financieros y los plazos pueden poner en peligro esta etapa. Los números gobiernan el mundo y, lamentablemente, existen razones monetarias y restricciones de tiempo que pueden tentar a algunos a tomar decisiones irresponsables.

Para eso, se juntan tres componentes explosivos: 1) Cuando mejor y más exhaustiva es la prueba, más costosa es. 2) En este momento del proyecto, el software ya está programado y podría usarse, aun con errores. 3) La etapa de pruebas ocurre justo antes de entregar el software. Si estos tres componentes se combinan con una gestión deficiente del proyecto, la situación puede volverse explosiva.

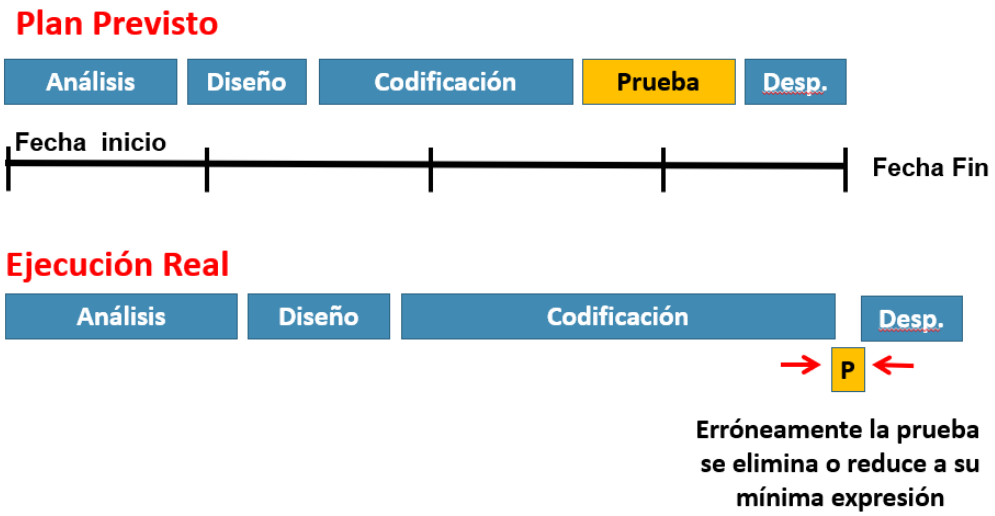
¿Cómo se desencadena esta explosión? Es posible visualizar un escenario en el que un proyecto enfrenta problemas tanto de costo como de tiempo. No es raro que las etapas iniciales del desarrollo, especialmente la codificación, requieran mucho más tiempo y recursos de lo inicialmente previsto. Llegar a la etapa de pruebas con un presupuesto agotado y con plazos de entrega próximos a vencer es una situación desafortunadamente común.

En este contexto, se plantea un dilema complicado: ¿Debería invertir el poco tiempo y dinero restante en realizar pruebas exhaustivas del software, o debería entregarlo en su estado actual y posponer la solución de posibles problemas? Incluso existe la opción de negociar un contrato de mantenimiento que cubra la corrección de errores. Es innegable que la tentación es alta.

Y aun cuando no se llegue a extremos de eliminarla, a veces se reduce a su mínima expresión. Ya no se buscan errores, sino que sólo se prueba que el software “más o menos” funcione y que

cumpla con las condiciones mínimas para entregarse: básicamente que no se “cuelgue” ni bien el usuario ingresa.

Esta imagen gráfica como la ejecución real del proyecto de software se va corriendo respecto de las fechas pautadas originalmente. Los tiempos disponibles para la prueba, la “única actividad optativa”, se reducen al mínimo.



Sin embargo, es crucial evaluar cuidadosamente las implicaciones a corto y largo plazo de esta decisión. Entregar un software sin pruebas adecuadas puede tener consecuencias significativas, como la insatisfacción del cliente, daños a la reputación de la empresa y costos adicionales para corregir los errores en una etapa posterior. Además, confiar en un contrato de mantenimiento para financiar las correcciones puede generar un impacto negativo en la relación con el cliente y en la confianza en el producto.

Además, la postergación de la corrección de errores puede llevar a un aumento en los costos y el tiempo necesario para solucionarlos en el futuro. Es posible que los problemas se acumulen y se vuelvan más complejos de abordar, lo que requerirá más recursos y esfuerzo para resolverlos.

Incluso en este punto la ética y la responsabilidad profesional debe entrar en juego. Sabemos que entregar software defectuoso pone en riesgo de modo directo la salud, la seguridad, los derechos, los bienes o la formación de los habitantes. Los y las profesionales en informática demos velar por proteger esos valores.

La calidad del software es innegociable, al igual que las pruebas. Aunque técnicamente las pruebas son opcionales, son tan fundamentales como cualquier otro componente dentro de la ingeniería del software. No podemos esperar que los errores desaparezcan mágicamente sólo por dejamos de probar. Lo único que estaremos logrando es que alguien encuentre las fallas y aceptando el eventual daño que causan cuando aparezcan. Quizás, en acuerdo con el cliente, podamos acortar algunos procesos. Incluso podemos implementar módulos del software mientras se prueban otros. Sin embargo, es crucial que todos comprendan que **el software no puede operar sin ser completamente probado**.

5. El equipo de pruebas

Casi siempre se suele hablar de que la prueba la debe realizar un equipo independiente. Parece lógico… la prueba consiste en "destruir" el software para encontrar fallas. El programador, que acaba de crear, con esfuerzo, una pieza de software. No parecería ser lógico pedirle que ataque seriamente a su propia obra para obligarla a fallar. Es razonable pensar que el desarrollador buscará su éxito y testeará la aplicación para demostrar que anda y que está bien programada, no para buscar que falle. Difícilmente vaya a fondo para probarse a sí mismo cuantos errores cometió. Contar con un equipo de pruebas independiente parece lógico.

No solamente se habla de independiente, sino de que el equipo debe ser externo al proyecto y no depender del líder de este. Suena razonable. Si quien testea tiene compromiso directo con el cronograma y con el presupuesto, se correrá el riesgo de que la prueba se ajuste al mismo para, por ejemplo, cumplir con los plazos de entrega.

La realidad, una vez más, no es tan simple. Armar un equipo de pruebas independiente es costoso. En los proyectos más chicos esto prácticamente es imposible y las tareas de prueba quedan relegadas a los desarrolladores y quizá alguna persona del equipo que conozca los requerimientos del usuario.

Los equipos externos muchas veces también son una quimera. Para que una organización pueda mantener un equipo de testers, deben tener un flujo de varios proyectos de desarrollo concurrentes que son testeados permanentemente por la unidad. Solo así pueden justificarse los costos. Sería difícil armar un equipo externo, utilizarlo en el momento de la prueba, y luego desactivarlo.

Queda una reflexión más: ¿Que tan bueno es que la prueba la realice sólo un equipo externo? ¿Puede un equipo que no conoce cómo se desarrolló el software probar eficazmente el software y encontrar todos los posibles errores? La respuesta es, definitivamente, no.

Si quienes desarrollaron el software no ayudan ni orientan a los testers, la prueba podría ser más difícil de lo esperado. Por ejemplo, se podrían malgastar recursos en probar módulos que, internamente, utilizan el mismo código (si funciona una para uno, funciona para el resto). Las funciones o cálculos más complejos podrían ser testeadas con menos rigurosidad, simplemente por no saber que son partes especialmente críticas o complejas. Incluso se podrían pasar por alto determinados acuerdos que se dan durante el desarrollo, incluso con la aprobación del cliente, para que determinadas cosas funcionen de un determinado modo. Mas adelante veremos que simplemente no es posible testear todos los caminos del software, por lo que la guia y ayuda de los programadores, incluso para confeccionar los casos de prueba, se vuelve decisiva.

6. Los casos de prueba

Es frecuente que, en empresas de desarrollo con áreas de testeo profesionales y separada de las de programación, una vez programado el software, este quede a disposición de la mencionada área. Esta realiza las pruebas conforme lo estipulado y o bien vuelve al área de desarrollo para corregir los errores encontrados, o dispongan la puesta en producción. Por supuesto que este

software que se recibe ya tiene una primera tanda de testeos, que es la que realiza el área de desarrollo para asegurarse que el software mínimamente funciona.

La pregunta que surge, entonces, es si el equipo de testeo cuenta con todas las herramientas necesarias para "atacar a fondo" al sistema. ¿Conoce cuáles son las reglas del negocio? ¿Conoce los requerimientos del usuario? ¿Conoce cuáles son los valores límites? ¿Puede probar funcionalidades vinculadas, por ejemplo, con el rendimiento?

Vayamos a un ejemplo. Supongamos un sistema de reservas de mesa de un restaurante. Se necesita que el sistema permita las reservas dentro de un rango de fechas determinada (por ejemplo, para los siguientes 15 días) y en un horario específico, que es cuando el restaurante toma reservas. Hasta aquí, la prueba es sencilla. Basta con seleccionar algunos valores que caigan dentro de los parámetros de fecha y hora y probar que la reserva se confirme. Por supuesto también hay que elegir valores fuera de rango y validar el rechazo. Pero que ocurre si, dentro de los 15 días, caen fechas especiales, por ejemplo, feriados, donde los horarios se modifican. ¿Puede el tester tener en cuenta esto y validarlo? Probablemente sí, pero para ello debiera conocer el negocio y los requerimientos, a los fines de armar una prueba específica que involucre un día feriado.

Y acá es donde cobran importancia los casos de prueba. Wikipedia los define como *"un conjunto de condiciones o variables bajo las cuales se determinará si una aplicación, un sistema de software o una característica o comportamiento de estos resulta o no aceptable."*

Es decir que, cuando dispone que el software vaya a ser testeado, este debe ir acompañado de una serie de indicaciones sobre diferentes escenarios que deben ser probados y cuáles son las condiciones de aceptabilidad de cada caso. Volviendo al ejemplo, *se deberá testear un rango de fechas que contenga al menos un feriado, verificando que los horarios de la víspera son los que rigen para un día sábado, y los horarios del día feriado serán los del día domingo*. Recibida esta instrucción, quien realiza el testeo sabe que tiene que verificar esta condición.

Otro ejemplo que complemente lo dicho puede vincularse al sistema de inscripciones de una facultad. Cuando hay algún cambio de plan, o de correlatividades, estos cambios a veces afectan a un grupo especifico de alumnos (los de una determinada carrera o los que ingresaron a partir de tal fecha). En este caso es necesario que los casos que se incluyen en la prueba sean alumnos del grupo especifico. El resto del sistema se supone, no sufrió modificaciones y por lo tanto no requiere un nuevo testeo. Obviamente de esto último hay que estar seguro.

También se pueden incluir sets de datos puntuales a probar, por ejemplo, testear solamente los documentos superiores a 90 millones, o los cuits comenzados en 30. Incluso pueden existir instrucciones específicas, como podría ser que la pantalla de carga de datos del usuario deba ser testeada tanto en equipos de escritorio como móviles. O que este módulo debe ser probado en tal horario, de modo de validarlo en los momentos de máxima carga del servidor.

En resumen, el "ataque" al sistema no debe hacerse en forma azarosa o simplemente confiando en la pericia de quien testea, sino que se deben armar una prueba y un ser de datos que contemple la mayor cantidad de casos reales posibles, y cuáles son los resultados que se esperan de cada una de ellas.

Y por supuesto, las pruebas son un componente más del proceso de aseguramiento de la calidad de software, y por lo tanto debemos asegurarnos de que se hicieron correctamente. Son necesarios

controles y auditorias sobre el proceso de prueba, de modo de validar que todas las pruebas previstas se han realizado, y que los resultados son los esperados.

7. Los tipos de prueba

Como se sabe, el software se va construyendo por partes. Incluso partes que se programan por separado, a veces por distintos equipos, y luego se integran. Esto dispara dos conceptos. 1) No es necesario esperar a tener el software terminado para comenzar a probarlo y 2) No se puede utilizar la misma estrategia para probar una sección de código que acaba de programarse y que, por ejemplo, todavía no tiene, una interfaz de usuario, que cuando se prueba la aplicación ya terminada. A veces se requiere construir algún modulo especial para probar otros.

Es por eso que existen varios tipos de prueba. Algunas son pensadas para que las realice el programador, ni bien termina de codificar un módulo. Otras, por ejemplo, están destinadas al equipo de testeo. Excede el propósito de este trabajo explicar cómo funciona cada una de las pruebas que se le realizan al software, pero si resumir los tipos de prueba que menciona Ian Somerville en su libro “Ingeniería de Software”, a modo de tener una visión general del tema:

Pruebas de desarrollo:

Se realizan en la misma etapa de codificación, habitualmente por el programador cuando acaba su desarrollo. Buscan encontrar errores en diferentes módulos de la aplicación, en su integración con otros módulos y en su funcionamiento dentro de un sistema.

Dentro de estas pruebas encontramos:

- **Pruebas de Unidad:** Se ponen a prueba unidades de programa, módulos o clases de objetos individuales. En este momento se prueban:
  - Todas las operaciones asociadas con el módulo y todos los caminos lógicos que tiene. Una instrucción que pregunte si el “precio > 0” dispara 3 caminos lógicos: Qué pasa si el precio es negativo, qué pasa si es mayor que cero y que pasa si es cero. Hay que probar que ocurre en cada caso. Generalmente es necesario tener acceso al código o a las especificaciones para determinar las rutas a probar.
  - Los mensajes de alerta de error que tiene el sistema para prevenir fallas. Para eso es necesario elegir entradas que fuercen al sistema a activar todas las validaciones, de modo de ver que advertencias se generan en cada caso (Por ejemplo, números muy grandes o email sin @). Por supuesto más grave es el caso de que un error no sea controlado y que el sistema falle y deje de funcionar (por ejemplo, ante una división por cero)
  - Como responde el módulo ante resultados de cálculo demasiado largos o pequeño, o que den números negativos o cero. Tambien es necesario comprobar los redondeos, ya que 2 números con podrían parecer ser iguales, pero operar como distintos si se comprueban varios decimales (5.14 y 5.14

son iguales a 2 decimales, pero la comparación aritmética tratarlos como distintos si fueran 5.14283 y 5.14284)

- Que los diferentes ciclos se ejecuten la cantidad exacta de veces y que todos tengan una salida (es decir que el sistema no entre en un loop infinito)

- **Pruebas de componentes**: Las unidades individuales ya probadas se integran para crear componentes compuestos, enfocándose en probar las interfases entre ellos. En este caso se busca:

  - Probar el correcto pase de parámetros entre dos módulos, sus tipos de datos y qué pasa si falta un parámetro o tiene valor nulo.

  - Verificar el correcto encapsulamiento de datos, es decir que módulos externos no pueda acceder ni modificar los datos internos.

  - Controlar los tiempos de actualización en sistemas multitarea, para asegurar el acceso a la información más actualizada. Por ejemplo, si se actualiza un precio o se carga un nuevo cliente, estos cambios deben estar inmediatamente reflejados y disponibles en el módulo de facturación.

  - Verificar los mensajes de error y recuperación cuando un componente no está disponible (Si un módulo no puede ser accedido por algún motivo, por ejemplo, porque puede estar siendo utilizado por otro módulo, el sistema no debe "colgarse" sino decir que intente más tarde)

- **Pruebas del sistema**: Algunos o todos los componentes del software se integran y el sistema se prueba como un todo, probando la interacción general. En este caso será necesario:

  - Verificar la integración de nuevos componentes que acaban de desarrollar (o se adquieran de terceros) con los demás componentes que ya existan en él sistema.

  - Probar la integración de componentes desarrollados por equipos de trabajo distintos.

  - Controlar que todos los accesos del menú estén disponibles y que todas las funcionalidades del sistema sean accesibles.

Pruebas de versión:

En estas pruebas ya se testea el sistema completo, comprobando que responda a los requerimientos del usuario y al rendimiento esperado.

- **Pruebas basadas en requerimientos**: Para cada requerimiento del usuario se diseña una prueba específica para probarlo y asegurar que este requerimiento se cumpla.

- **Pruebas de escenario**: En el caso que los requerimientos se hayan expresado como escenarios o casos de uso, las pruebas se basan en las descripciones que contiene aquel escenario, por lo general, probando varios requisitos a la vez.

- **Pruebas de rendimiento**: Deben diseñarse pruebas para forzar el sistema y garantizar que procese la carga pretendida. Las pruebas de esfuerzo son particularmente relevantes para los sistemas más grandes, con múltiples usuarios y que puedan saturar la red o los servidores.

Pruebas de usuario:

Son pruebas dende el usuario prueba el software para ver que cumpla con los requisitos y, eventualmente, apruebe el software para su liberación y puesta en marcha:

- **Pruebas alfa**: Los usuarios del software trabajan con el equipo de diseño para probar el software en el entorno del desarrollador.

- **Pruebas beta**: Una versión del software se pone a disposición de los usuarios, para que, en su propio entorno, puedan experimentar y descubrir problemas [40]

- **Pruebas de aceptación**: Los clientes prueban un sistema para decidir si está o no listo para ser liberado al entorno de operación. Deben definirse de antemano en el contrato cuales son los criterios de aceptación y negociarse como se continua a partir del resultado. Este tipo de pruebas son especialmente utilizadas en modelos ágiles de desarrollo.

Es importante refrescar el concepto de que la prueba busca encontrar errores. Y este concepto lo deben tener también los programadores y los usuarios que participen de la etapa de pruebas. El usuario no se debe limitar a probar que el sistema funciona para un uso normal, sino que debe forzar al mismo con operaciones extremas y no tan habituales, asegurando que el software responderá aun en dichos escenarios.

8. ¿Caja blanca o caja negra?

Muchos autores dividen las pruebas en aquellas que son de **caja blanca**, mirando el código, o **caja negra**, operando el sistema. Lo primero que vale decir es que dicha división no es precisa: La mayoría de las pruebas de software pueden combinar ambas estrategias.

Las pruebas de caja blanca analizan el código para probar procedimientos y caminos lógicos:

- Buscan recorrer todos los caminos lógicos al menos una vez.

[40] Es habitual encontrar en internet versiones Beta de un software. Estas versiones son justamente aquellas destinadas a los usuarios que harán de testers. No deben ser nunca utilizados como versiones finales ya pueden tener errores o funciones no terminadas.

- Controlan las estructuras de decisión, tanto el verdadero como el falso.
- Revisar el funcionamiento y la salida de bucles.
- Analizan estructura de datos.

Las pruebas de caja negra, por el contrario, se realizan operando el sistema, utilizando la interfaz del usuario:

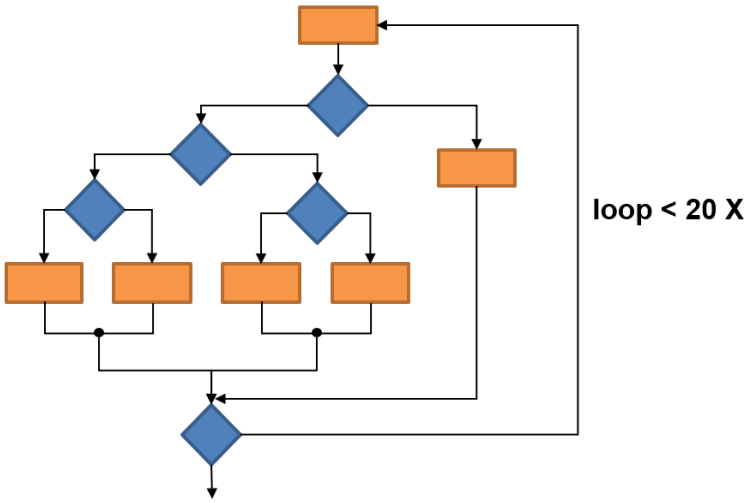
- Buscan funciones erróneas o faltantes.
- Detectan errores en la interfaz, por ejemplo, campos cortados o colores poco visibles.
- Aseguran que la base de datos o componentes externos sean accesibles.
- Revisan el comportamiento y rendimiento.

Como se ve, ambas estrategias se orientan a buscar diferentes problemas. Hay cosas que de detectarán solo con caja negra y otras que no aparecen salvo que uno mire el código. Si en una pantalla falta un botón para imprimir, es algo que difícilmente se note si no se visualiza esa pantalla (caja negra). Encontrar una rutina que haga algo que no debe, por ejemplo, desviar centavos de una operación a una cuenta particular, es algo casi imposible de detectar si no se inspecciona el código.

Ambas estrategias son, entonces, necesarias y complementarias. Un sistema, obviamente, no puede ser probado si no se lo ejecuta. Pero, también, hay que desconfiar de aquellas aplicaciones que solo han sido testeados sin revisar en ningún momento el código fuente que dio origen al sistema.

9. ¿Cuándo termina la prueba?

Podríamos encontrarnos con un software que responda al siguiente diagrama… con decisiones que disparan diferentes acciones dentro de un ciclo que se repite 20 veces.



Aunque a simple vista no lo parezca, al ejecutar este módulo la computadora puede optar por 1.000.000.000.000.000 de posibles caminos. Aun si recorriéramos con ayuda de un computador un cada camino en un segundo, nos llevaría 3000 años probarlos todos. Esto nos lleva a afirmar que

es humanamente imposible probar todas las opciones de recorrido que puede tener un software. Y entonces aparece un dilema entre seguir buscando errores y entregar el sistema:

Si…

Es imposible probar todas y cada una de las posibilidades lógicas de un software…
El éxito de la prueba es encontrar errores, pero esto no garantiza que se encuentren todos ello…
Es riesgoso, incluso mortal, encontrar software que contenga errores…

Entonces…

¿Cuándo estamos en condiciones de terminar la etapa de pruebas y entregar el software?

En visión extremista, nunca. Siempre tenemos la chande de seguir probando hasta que aparezca el próximo error. Pero claro… el peor software es el no entregado, ya nos priva de todos los beneficios de usarlo.

Por supuesto que dependerá mucho del tipo de software que se está construyendo. No da lo mismo un software para manejar vehículos autónomos, un sistema bancario o el software para registrar las ventas de un pequeño negocio. El impacto de un error encontrado a tiempo no será el mismo en un caso que otro.

Algunos de los criterios, aunque no los únicos, que podemos considerar para decidir entregar el sistema son:

- Cuando ya no sea razonable buscando errores. Todas las pruebas deben tener criterios de finalización, porque la búsqueda puede ser eterna.

- Cuando el costo de encontrar el próximo error sea mayor que el beneficio de encontrarlo. Por supuesto el costo en vidas siempre es infinito. Pero, en si un sistema de facturación debo esperar 5 días y gastar miles de pesos para tener chance de encontrar el próximo error que, si aparece, tendría un impacto de centavos, entonces no tiene sentido buscarlo.

- Cuando el beneficio de lanzar un sistema, aun con algún posible error, sea mayor que el perjuicio de no hacerlo. Por ejemplo, si necesitamos abrir un negocio en un día determinado, es peor seguir buscando fallas que abrir el negocio.

- Cuando el sistema por entregar sea mejor o tenga menos problemas y errores que la aplicación que hoy está usándose.

Si bien los criterios de aceptación de cada caso son definidos de antemano y acordados con todas las partes interesadas, usuario incluido, también hay que considerar la posibilidad de variar criterios en función de cómo avance el testeo. Por ejemplo, podría ocurrir que en las primeras

pruebas se detecten más errores que los previstos y esto nos lleve a reformular o profundizar la etapa.

10. La depuración o debugging

La depuración del software es una de las consecuencias del proceso de prueba: Los errores detectados deben corregirse. Durante la depuración debe localizarse el origen del error, corregirlo, volver a probar el software y ponerlo nuevamente a disposición de los usuarios.

Esta es una tarea compleja. Puede pasar que el reporte del error sea confuso o incompleto, o que no describa adecuadamente la situación en la que se produce. A veces, el problema aparece en el equipo del usuario, pero es difícil de reproducir en el entorno de desarrollo. Otras veces, el origen de este está en otro módulo diferente. Incluso puede originarse en causas externas como el hardware o el sistema operativo. En ocasiones el error puede intermitente o aparecer solo ante algunos eventos puntuales, como puede ser un micro corte de red.

Existen algunas técnicas que ayudan a la depuración:

1. **Fuerza bruta**: La corrección se da por prueba y error. Se prueban casos y soluciones reiteradamente hasta que se encuentra la correcta. Por supuesto es una técnica que no es eficiente, pero, ocasionalmente, la única que funciona.

2. **Vuelta atrás**: Se detienen el programa en algún punto y desde allí se comienza a desandar manualmente el camino, hasta encontrar al error. En determinados puntos del sistema esto es complejo porque los caminos a recorrer pueden ser múltiples.

3. **Eliminación de causas**: Se van eliminando del código posibles causas de error, buscando de este modo aislarlo. Por ejemplo, un cálculo complejo se puede dividir y eliminar temporalmente partes de este, hasta encontrar aquella que cause el error.

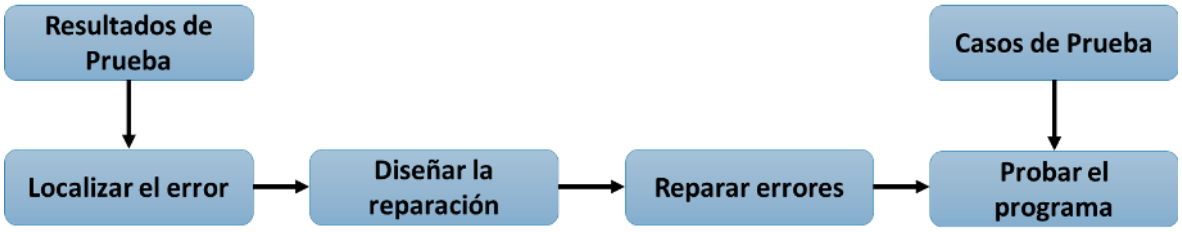
Los IDE, entornos integrados de desarrollo, proveen a los programadores de múltiples herramientas que facilitan la detección y corrección de errores. Entre ellas podemos mencionar:

- **Correctores automáticos de código** que señalan, por ejemplo, con códigos de colores, estructuras incompletas, paréntesis faltantes, funciones inexistentes, errores de sintaxis, etc. (algo similar al corrector ortográfico de Word)

- **Puntos de break**: permiten detener la ejecución de un programa en un punto exacto, por ejemplo, justo cuando de aprieta un botón, pero antes de que se ejecute el código asociado. En ese momento el programador puede ver el valor que en ese momento tienen las variables del sistema.

- **Ejecución paso a paso (Trace)**. Atado a lo anterior, una vez detenido el programa en un punto, el mismo puede continuar línea a línea en forma manual. De este modo el programador puede avanzar e ir chequeando cómo se comporta el sistema desde el último punto seguro.

En líneas generales, el proceso de depuración termina siendo un nuevo mini proceso de desarrollo que pasa por etapas similares que el desarrollo original:



Una vez reportado el error debe localizarse, diseñar una solucion, codificarla y, una vez construida la nueva versión, será momento de una la prrueba antes de volver a poner el software en funcionamiento. Al hacerlo es importante considerar que otros módulos del sistema podrían ser modificados a partir de las nuevos cambios.

11. Consideración final

Dentro de un programa, hay funciones que se repiten. Por ejemplo, la validación del número de CUIT podría estar presente en 4 ó 5 pantallas de carga. Si bien las buenas prácticas de programación señalan que en estos casos debe construirse un módulo reutilizable que se invoque cada vez que necesite, esto no siempre pasa. Hay veces que no se requieren exactamente las mismas funcionalidades y termina siendo frecuente que algunas líneas de código copien y se usen como base en algún otro lado.

Esto abre dos escenarios. Si el error detectado está en un módulo reutilizable, la solución aplicada a ese módulo obviamente corrige los problemas en cualquier lado donde se utilice. Si en cambio el problema está en una parte del código que fue copiada y utilizada en otra parte del sistema, en este caso el error podría replicarse. Es por eso que, cuando encontramos un error, deberíamos hacernos las siguientes preguntas: **¿Se repite la causa del error en otra parte del sistema? ¿Qué error podría presentarse con la corrección?**

Finalmente, y buscando siempre la mejora continua de los procesos, también nos deberíamos preguntar si el error no podría haberse prevenido, y si hubo alguna circunstancia que llevó a cometerlo y a no detectarlo, de modo de evitarse a futuro.

12. Bibliografía

IAN SOMMERVILLE: “Software Engineering”. 10ma edición. 2016. Pearson Education.
ROGER PRESSMAN: Ingeniería del Software. 7ta Edición. 2008. Ed. McGraw-Hill.

Apunte 8

Conceptos Fundamentales del Despliegue de Software

1. Introducción

El primer objetivo al desarrollar software es tener un programa que haya sido probado y esté listo para su operación. Sin embargo, este hito no marca el final del proceso, sino solo el comienzo de una etapa posterior. A continuación, es necesario desplegar el software, es decir, ponerlo en funcionamiento real. Este paso no debe ser subestimado, ya que implica considerar diversos aspectos, como el entorno de hardware específico de la empresa, la capacitación de los usuarios que utilizarán el software, la migración de datos existentes, la interacción con otros sistemas de la organización e incluso la resistencia al cambio por parte de las personas involucradas.

Aunque parezca natural que un software que se utilice, la verdad es que hay muchos casos donde esto no sucede. Hay desarrollos excelentes que fallan por el modo en el que son implementados. Otros que son geniales pero los usuarios no quieren usarlo porque son complejos o les generan mucho más trabajo que la operatoria manual. Incluso hay casos donde se demora tanto en construir que, para cuando está listo, ya no tiene sentido usarlo, o la competencia sacó uno mejor, o Google lo incorpora a su plataforma como servicio gratuito. Todas estas cosas hay que preverlas de antemano, en especial trabajar cobre la resistencia de los usuarios y los riesgos empresariales.

El "Pase a Producción" como muchas organizaciones lo denominan, implica atender muchas más cosas que la de simplemente que mover el software del entorno de desarrollo al productivo. Este apunte busca reflexionar sobre alguna de ellas.

2. Construir la versión final

Lo que genéricamente se denomina *software listo para entregar* es, en realidad, un conjunto de cosas. Entre ellas encontramos:

- El o los **programas ejecutables** propiamente dichos.
- Los **archivos auxiliares** que el sistema utilice: librerías internas y externas, imágenes, videos, archivos de idiomas, plantillas, etc.
- La **base de datos**, con tablas que pueden contener datos precargados (ej. tabla de países o provincias o tabla de parámetros), y otras vacías que almacenarán los datos del sistema. En caso de que la base de datos deba montarse sobre un gestor de base de datos ya operativo, habrá que dar todas las indicaciones y detalles para que el administrador de la base de datos pueda hacerlo.
- Los **archivos de configuración** que sirvan para personalizar el sistema.
- El detalle de **requerimientos técnicos de hardware**, tanto del servidor como de las estaciones de trabajo y redes, incluyendo las **versiones de sistema operativo** y todo aquel software que debe estar instalado. En caso de ser necesario, deberá detallarse también las actualizaciones y los parches de seguridad que se necesitan.

- Los **manuales técnicos**, que indiquen cómo instalar el software y la base de datos y los permisos que deban asignarse.

- Los **manuales e instrucciones de uso** para quienes deban operar el sistema. En nuestros días, lo usual es que estos manuales sean digitales y estén embebidos en el sistema (F1: Ayuda) o estén disponibles en línea, lo que permite que puedan ser actualizados ante los cambios que tenga el software.

- Las **certificaciones** y aprobaciones que correspondan. (Ej. certificación de la FAA para software de aviones).

- Copia de los demás **documentos** que se originaron durante el proceso de desarrollo del software y que sirvan para documentar el proyecto.

Cada una de estas herramientas tiene destinatarios específicos. El ejecutable, por poner un ejemplo, deberá se instalado en alguno de los servidores de la empresa; los requerimientos técnicos serán analizados por quién ser encargue de infraestructura; los manuales tendrán que estar a disposición de cada interesado.

Es importante asegurar que cada usuario que accederá el sistema tenga los permisos suficientes para acceder al servidor, incluyendo la posibilidad actualizar archivos. Estos permisos son de acceso genérico y complementarios a los más granulares que la aplicación le otorgue. Un usuario puede, por ejemplo, tener acceso a modificar datos, pero tener vedado emitir determinados listados. Esos permisos los maneja la propia aplicación, pero previamente debe tener acceso al servidor para poder ejecutar el sistema.

Si el software debe interactuar con otras aplicaciones de la organización, por ejemplo, aplicativos de la AFIP, se deberán indicar los modos de realizar dichas conexiones y los usuarios y permisos a utilizar en cada caso.

Una vez instalado el sistema también habrá que establecer cuáles serán los protocolos para las actualizaciones. Habrá que definir si es posible que estas se realicen en forma remota y en qué horarios podrán hacerse estos cambios, de modo de no afectar la disponibilidad del sistema en momentos en los que este operativo el negocio. También habrá que ver de qué forma se harán cambios a bases de datos cuando, por ejemplo, sea necesario agregar un campo o cambiar atributos; cosa que por supuesto debe hacerse sin afectar la integridad de los datos ya cargados. Incluso habrá que prever la contingencia de que la nueva versión tenga algún problema no detectado previamente y que deba volverse a la versión anterior de la aplicación o de la base de datos. Hoy día, la mayoría de los sistemas cuentan con procesos de actualizaciones automáticas, como Windows, tienen incluidos los procesos de actualización, de modo que el usuario simplemente actualice

En concordancia con lo dicho, un punto muy importante que se deberá documentar será el modo de hacer un backup completo de la aplicación, qué archivos deben resguardarse, dónde están ubicados y qué archivos externos son necesarios preservar. Habrá que proporcionar indicaciones detalladas de cómo restaurar la aplicación desde cero junto con los datos de la última sesión válida, en el hipotético caso de que, por ejemplo, sea necesario formatear algún equipo.

Hoy día, la mayoría de los sistemas cuentan con procesos de actualizaciones automáticas. El usuario solo debe autorizar la actualización y el propio sistema se ocupa de realizar todo el proceso.

De igual modo, tambien tienen incluidos opciones de backup (y de recupero), que funcionan de sin que el usuario deba realizar acción alguna. Se puede seleccionar la periodicidad; que sean completos o incrementales (es decir que solo se haga copia de los cambios diarios); elegir la nube como destino de las copias o que estas estén cifradas, como ocurre con Whatsapp. De más está decir que todo esto no ocurre porque si, y que son funcionalidades del sistema que fueron oportunamente diseñadas e incorporadas.

Por supuesto que cuando la instalación es complicada y requiere configuraciones muy específicas, es deseable que el equipo de desarrollo colabore con los administradores de servidores y bases de datos para lograr una implementación exitosa.

Todos los procesos anteriormente descriptos son fundamentales para asegurar una puesta en marcha exitosa y, por lo tanto, será importante prever una revisión o auditoría para asegurar que cada uno de los requerimientos de instalación han sido respetados. Por ejemplo, habrá que testear que en efecto los usuarios tengan todos los permisos que necesitan o que el hardware de las estaciones de trabajo cumple con las especificaciones. Del resultado de esta auditoría, además de asegurar el correcto funcionamiento del sistema, pueden surgir cambios en el proceso de instalación para las próximas versiones o para otros sistemas que se desarrollen.

3. El manejo de múltiples versiones

Es frecuente que los desarrolladores majen simultáneamente más de una versión del mismo software. Puede haber, por ejemplo, versiones para diferentes sistemas operativos o idiomas. También pueden existir personalizaciones o funcionalidades agregadas que hicieron para un cliente o grupo de clientes específicos.

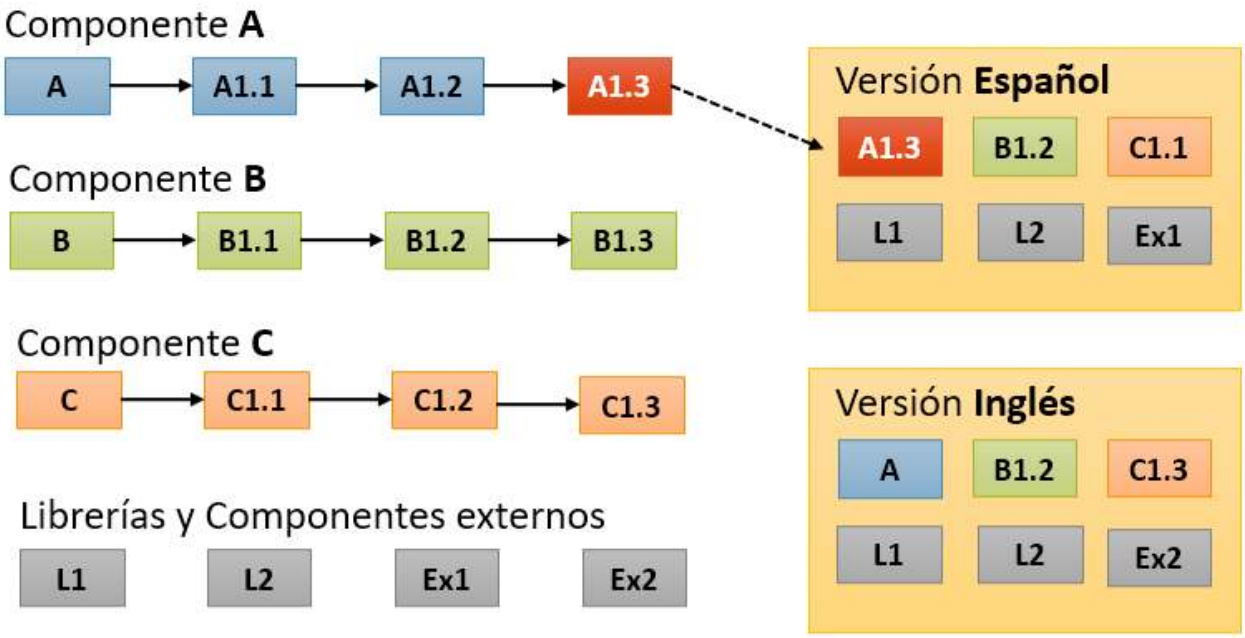
Incluso se puede estar trabajando en una actualización mayor, mientras se sigue manteniendo y corrigiendo eventuales problemas de la versión actualmente en operación. También puede pasar que dos programadores necesiten corregir a la vez un mismo código fuente.

Ante estas circunstancias es necesario ser muy cuidadoso a la hora de construir y distribuir el software. Cada ejecutable debe contener sus componentes específicos y a cada cliente debe recibir la versión que le corresponde.

Somerville define que "***La gestión de versiones*** *(VM, por las siglas de versión management) es el proceso de hacer un seguimiento de las diferentes versiones de los componentes de software o ítems de configuración, y los sistemas donde se usan dichos componentes. También incluye asegurar que los cambios hechos a dichas versiones por los diferentes desarrolladores no interfieran unos con otros. Por lo tanto, se puede considerar a la gestión de versiones como el proceso de administrar líneas de código y líneas base*[41]"

[41] El concepto de línea base hace referencia a una porción de código que ya ha sido testeada y aprobada y que no permite modificaciones, con el objeto de que no se introduzcan cambios que dañen partes vitales del sistema o que funcionan.

El siguiente gráfico muestra como diferentes componentes se integran en distintas versiones. Podemos pensarlo como que existen 2 versiones del sistema, una en español y en otra en inglés. Habrá componentes genéricos que forman parte de ambas versiones y otros específicos:



Es decir, si se necesita actualizar un componente, por ejemplo, el A, solamente para aquellos clientes que posean la versión en español, esta actualización generará el subcomponente A1.3. A la hora de compilar la nueva versión en español, se utiliza dicho componente. La versión en inglés del software, por el motivo que fuera, seguirá utilizando la versión A original del componente. Es decir que los nuevos cambios realizados no afectarán a los clientes que estén utilizando la versión inglesa y por lo tanto no hay necesidad de actualizarse.

Obviamente, si se modifica la librería L1, los cambios afectarán a ambas versiones porque es un componente compartido por las dos.

Si bien esto presupone complicaciones y una alta posibilidad de equivocar versiones y componentes, es posible utilizar **software de gestión de versiones**. Estos operan como grandes repositorios, donde se almacenan los componentes individuales y luego permiten generar, de modo automático, la compilación de archivos necesaria para cada cliente o versión específica. Estos sistemas también cuentan con la posibilidad de generar instaladores automatizados que le simplifican a los administradores de los sistemas de la empresa la tarea de copiar cada componente en el lugar en el que deba estar.

De este modo se prepara la versión final para la entrega (o reléase). Una **entrega (release)** de sistema es una versión del software que se distribuye a los clientes. Habitualmente estas versiones se identifican con números o nombres para poder identificarlas fácilmente. Para software de mercado masivo es posible identificar por lo general dos tipos de entregas: **Release Mayor**, que proporciona por lo general funcionalidades significativamente nuevas, y **Release Menor**, que repara bugs y corrige problemas reportados.

Una costumbre arraigada, aunque no obligatoria, es la de identificar los reléase mayores con un número y los menores con fracciones de este: Ej. Windows 11 versión 10.0.22621, Firefox 80.0.1, Word 16.0.12527.20986[42].

Android solía utilizar nombres de postres, los que se ordenaban alfabéticamente, para reemplazar los números de sus versiones mayores. Android 8 era conocido como Oreo, Android 9 era Pie. Estos nombres y códigos de versión pueden ser internos de la empresa que los desarrolla o trascender a los clientes.

La correcta identificación de versiones se vuelve central a la hora de las actualizaciones automáticas. Por ejemplo, será posible lanzar una actualización de Windows que solo sea aplicada a las versiones 1909, sin que afecte a otras versiones.

También es posible obligar a que todos los clientes actualicen las versiones consideradas obsoletas para seguir actualizando el software. Incluso el soporte técnico o las actualizaciones pueden finalizar para determinadas versiones o dejar de funcionar en determinado hardware: Ej. Windows 7 finalizó su suporte en enero de 2020, IOs 12 ya no funciona en Iphone 4 o Whatsapp no puede instalarse en Android 2.3 o IOs 8, o versiones anteriores.

4. La capacitación de usuarios

No importa cuán avanzado, probado o sólido sea un sistema, existe el riesgo de que falle si el usuario no sabe cómo utilizarlo correctamente. Incluso si el sistema ofrece numerosas y excelentes funcionalidades, carecerán de utilidad si el usuario no las conoce o solo utiliza las funciones más básicas.

Es fundamental reconocer que la adopción exitosa de un sistema no se limita a su diseño y desarrollo, sino que también implica asegurarse de que los usuarios estén capacitados y familiarizados con su uso. La capacitación y la educación del usuario son aspectos cruciales para garantizar el correcto aprovechamiento y la eficacia del sistema.

Por lo tanto, es importante dedicar tiempo y recursos para proporcionar a los usuarios una formación adecuada, manuales de uso claros y accesibles, y brindar soporte continuo para resolver dudas y ofrecer asistencia en caso de dificultades. De esta manera, se maximizará el potencial del sistema y se evitarán problemas derivados de una incorrecta utilización por parte de los usuarios.

La primera indicación que suele darse en estos casos es que todo software debe acompañarse de sus respectivos manuales de operación. Pero nada más alejado de la realidad si se quiere lograr capacitar eficazmente a usuarios reales.

Una ley no escrita dice que “cuanto más extensos y completos sean los manuales de usuario, menos gente los leerá”. Una segunda ley, tampoco escrita, establece que “cuanto mayor sean las

[42] En algunos casos, como Windows, el número de versión hace referencia al mes y año en el que se lanzó la actualización. En otros casos, como el Word o Firefox, con códigos que dan al desarrollador una mayor información respecto a que componentes contiene o que errores se han corregido.

actualizaciones que reciban los sistemas, menor será la actualización los manuales de uso". No solo nadie los lee, sino que estos quedan obsoletos en la primera actualización.

Por supuesto, los manuales de uso deben existir, pero deben ser pensados más como una herramienta de consulta y referencia que como una herramienta de capacitación. Son pocos los usuarios que aprenden a usar un sistema leyendo sus manuales.

No existen manuales de Facebook, ni de Instagram, ni de los home-banking. Mejor dicho, en realidad si existen, pero casi nadie los leyó. Sin embargo, los millones de usuarios usan a diario esos sistemas.

Las ayudas en línea, los chatbot (asistentes virtuales) y los videos explicativos (siempre que sean cortos y específicos) suelen tener mayor aceptación que los documentos de texto. Eventualmente se podrán prever reuniones de capacitación presenciales o virtuales. En estos casos siempre hay que recordar que el usuario debe atender sus propias obligaciones y los tiempos para capacitaciones intensivas no abundan. En ocasiones también será útil instalar entornos de prueba y capacitación en cuales el usuario podrá entrenarse y "jugar" libremente con el software, en sus propios tiempos, y sin el temor de afectar la operatoria real.

Debemos tratar de construir software que, por su simpleza y su similitud con el mundo real y la operación manual, no requiera grandes conocimientos para operarlo. Debemos minimizar la posibilidad de ingresar datos erróneos. Podemos usar la propia pantalla para indicarle al usuario qué es lo que está haciendo bien o mal (mostrándole un cartel que el precio ingresado es incorrecto, o que falta completar algún campo o poniendo en verde los datos sin errores) o cuál es la siguiente operación que se espera que realice (ej. revise su casilla, debió haber recibido un mail de confirmación de los datos ingresados). La mejor capacitación se da cuando el usuario usa realmente el sistema. Analizar qué es lo que hace el usuario, impedirle hacer cosas incorrectas y asistirlo en la propia pantalla es un método muy efectivo de entrenamiento.

Del mismo modo se puede utilizar el mismo sistema para informarle al usuario de nuevas funcionalidades; por ejemplo, resaltando de forma especial un nuevo botón, o generándole una alerta que le indique que se ha incorporado una nueva funcionalidad o un nuevo listado.

5. Fechas de implementación

Por lo general la fecha en la que se implementa el software viene dada por el cliente o por condiciones externas. Por ejemplo, puede necesitar tener lista la aplicación antes de comenzar con determinada campaña de venta; la apertura de una sucursal; el lanzamiento de un nuevo producto o incluso el vencimiento de la licencia de uso de la versión anterior. También puede haber plazos determinados por normativas legales; de organismos de contralor o, incluso, imposiciones de casa matriz.

Pero… ¿qué pasa cuando nosotros podemos elegir la fecha? ¿Da lo mismo hacerlo en cualquier momento del año? ¿Cuál es la mejor fecha para elegir?

Obviamente la primera y más obvia respuesta es que la mejor fecha para entregar un software cuando esté terminado y lo suficientemente probado. Más allá de esta obviedad, no todas

las fechas son iguales. Por supuesto que cada tipo organización y cada tipo de sistema van a condicionar cuáles son los mejores momentos para implementar software. Aquí van algunas consideraciones:

- **Primer día hábil del año:** Año nuevo vida nueva dice el refrán. Por qué no empezar también estrenando sistema. Muchas organizaciones suelen elegir esta fecha para implementar el software y es porque existen dos importantes ventajas. La primera es que la mayoría de las empresas cierra su ejercicio comercial en esa fecha, realiza inventarios, pone todo en orden y arranca con sus cuentas en cero. La segunda es que suele ser un mes donde hay menos movimiento comercial y por lo tanto habrá más tiempo para la capacitación y para corregir eventuales errores.
  Pero claro, también enero suele ser un mes de vacaciones del personal de la empresa. Los operadores habituales pueden no estar trabajando e incluso se complica la capacitación ya que hay que realizara en tandas. El personal técnico, tanto de la empresa usuaria como de la desarrolladora, también pueden estar de licencia y ser más difícil atender y corregir cualquier problema que surja.

- **Aprovechar los cierres de ciclo o ejercicio:** Enero es mes de vacaciones, pero más allá de eso, cualquier inicio de ciclo es un excelente momento para comenzar a utilizar un software. Todos los sistemas procesan datos acumulados y no da lo mismo poder arrancar con las cuentas en cero que tener que migrar operaciones anteriores. Cualquier comienzo de mes puede ser preferible a hacerlo en la mitad ya que, por ejemplo, datos impositivos como el IVA, tienen cortes mensuales.

- **Aprovechar los momentos de menor operatoria:** Buena parte de las empresas tienen operatoria estacional. Diciembre suele ser un mes muy complejo, con muchos feriados y muchas operaciones por las fiestas. No parecería ser el mejor para implementar sistemas. Sin embargo, a veces hay que hacerlo por cuestiones presupuestarias (si no se paga en diciembre la erogación impactante contra el presupuesto del año siguiente). Enero, como se dijo, suele ser tranquilo, salvo para empresas vinculadas al turismo. En fin, hay que conocer cuáles son los ciclos de la organización y procurar que la implementación sea en los momentos más favorables.

- **Los viernes:** Esta expresión no hay que considerarla tan literal, sino significando que puede ser conveniente implementar un sistema teniendo un margen, aunque sea pequeño, para solucionar algún problema que pueda presentarse. Si algo falla el viernes puede solucionarse durante el fin de semana, minimizando el impacto. Desde luego, el viernes debe ser evitado si la empresa opera el fin de semana y nadie puede atender un reclamo o solucionar un problema hasta el lunes.

- **Lejos de las auditorias o cierres de balance:** Las empresas suelen tener auditorias periódicas, controles externos y obligaciones de presentar balances o información a organismos de contralor. Estas operaciones suelen requerir preparar información especial y, ocasionalmente, generan trabajo extra. Suelen ser momentos de tensión en las organizaciones. No parece buena idea sumar en ese período la implementación de un nuevo software.
  Salvo que el nuevo sistema provea funcionalidades específicas o información pertinente para la auditoria (por ejemplo, que emita nuevos listados con la

información exacta que requiera el organismo auditor), es preferible que los sistemas se implementen lo más lejos posible de las mencionadas situaciones.

- **Considerar los tiempos y rutinas de los usuarios:** Ya se comentó que las organizaciones tienen ciclos y obligaciones puntuales en momentos específicos. Lo mismo ocurre con los usuarios. Hay áreas que hacia fin de año tienen picos de trabajo, otras hacia fin de mes, algunas que están enfrentando procesos de cambio o rotación del personal, jefes recién llegados… Cada área tiene su mejor momento para atender las nuevas tareas y obligaciones que les impondrá la implementación. Es importante el diálogo y la coordinación con los usuarios y negociar, dentro de lo posible, cuáles son los mejores momentos para los cambios.

- **Mantener ciclos de entrega:** En metodologías ágiles, donde las implementaciones son constantes, es importante que los ciclos de implementación (sprints) se mantengan fijos, para que el usuario sepa cuáles son las semanas en las que habrá nuevas versiones. En los casos de que sea necesario un cambio en la rutina habitual será necesario una nueva coordinación. Por ejemplo, si se realizan sprints de 2 semanas, las actualizaciones pueden caer en la semana 1 y 3. Si por algún motivo, como puede ser una semana de confinamiento o de feriados, esos ciclos se corren para la semana 2 y 4, podrían coincidir con cierre mensual y, por lo tanto, ser mal momento para cambios nuevas versiones.

- **Con la mayor anticipación posible:** Como se dijo, a veces las fechas son fijas. Siempre hay que considerar que los sistemas, por mejor testeados que estén, pueden tener fallas en el momento de implementación. Esto no solo es por errores que pueda tener el software, sino también por fallas humanas de quienes los operan. Un cajero bancario no va a poder operar con la soltura habitual el día que, por primera vez, deba usar un nuevo sistema de caja. Hay que considerar pues esta curva de errores y aprendizaje. Procurar que el software esté disponible para operar cierto tiempo antes de la fecha clave, para tener tiempo de entrenar, de solucionar algún problema y que cuando llegue el día, la operatoria sea lo más normal posible.

- **Según el calendario.** Determinadas industrias, como por ejemplo el desarrollo de video juegos, tienen momentos específicos del año donde las ventas de su software se disparan. Una de ellas, por ejemplo, son las navidades en EE.UU. No contar con la nueva versión del juego, o sacar al mercado un nuevo título después de esa fecha, puede ocasionar pérdidas millonarias por usuarios que no lo comprarán como regalo para esas fechas especiales.

- **Antes que mis competidores.** Cuando se desarrolla software genérico, es decir que no es a medida de una organización determinada, debo procurar estar a la vanguardia respecto de otros programas similares que haya en el mercado. Por ejemplo, si la AFIP obliga a la emisión de factura electrónica, contar prematuramente con un software que sirva para dicha emisión, supone fidelización y la posibilidad de atraer nuevos clientes. Del mismo modo si una actualización por cambio normativo se entrega con anticipación y sin errores, producirá beneplácito entre los clientes, mientras que una entrega tardía generará problemas y malestar con el sistema. El FIFA y el PES con dos ejemplos donde, la demora en actualizar versiones, pueden hacer que los usuarios miguen a la otra plataforma.

Una última observación: Existen casos en los que los sistemas deben ponerse en funcionamiento el día más crítico de todos y donde no es posible postergar la implementación. Ejemplos de este tipo pueden ser los softwares de acceso a recitales o a partidos de futbol, donde deben ponerse en funcionamiento para ese día sí o sí. Si bien se espera que hayan sido texteados, es imposible, a veces, emular las reales condiciones de operación, con miles de usuarios queriendo comprar su entrada o tratando de validar el acceso desde su celular, usando redes que, necesariamente, se congestionan. Acá las alternativas son pocas. No se pude pedirle a la gente que no entre toda al mismo día o que vaya ingresando por tandas en horarios diferentes. Mas allá de las pruebas, eventualmente habrá que prever mecanismos de contingencia por si el sistema falla, ya sea entregando entradas manuales o deshabilitando algunas funciones secundarias. La AFIP, por ejemplo, suele modificar accesos los días de vencimiento, para que no se sature la operatoria de su sitio Web. Lo mismo hacen los diarios ante noticias de alto impacto, liberando de su home de aquel contenido que pude bloquear las conexiones.

6. Migración de datos

En toda organización hay datos que se necesitan preservar. A veces, pertenecen a un sistema previo y deben ser incorporados al nuevo para continuar con la operatoria (por ejemplo, los saldos actuales de las cuentas corrientes). Otras veces, esos datos solo deben quedar a disposición para consultarse como históricos. Lo cierto es que en cada implementación se necesita resolver qué se hará con los datos almacenados en los sistemas anteriores (o existentes en registros manuales).

Existen casos, como los sistemas de recursos humanos, en donde se requiere tener acceso a toda la historia del empleado, por ejemplo, para calcular antigüedades, indemnizaciones o datos del cese laboral.

En estos casos deben proveerse mecanismos para migrar los datos del viejo formato al nuevo. A veces estas migraciones son sencillas, una simple copia, otras hay que construir procesos complejos. Por ejemplo, puede ser necesario dividir los datos que almacenan en un mismo campo el nombre y apellido, si es que el nuevo sistema requiere que se guarden por separado. También es posible que el nuevo sistema tenga campos obligatorios que no existían anteriormente, o en formato diferente (numérico versus texto).

Es común que los sistemas prevean opciones básicas de importación de datos, que obligan al usuario a preparar y tabular los datos existentes, por ejemplo, en Excel, respetando un formato exigido. En otros casos se incluyen migraciones automáticas desde determinados software estándares y conocidos. En aquellos casos que la migración es compleja, es posible que se requiera la construcción de un software específico para realizar dicha tarea, que será complementario y presupuestado aparte.

Pueden plantearse varias estrategias a la hora de planificar la migración de datos:

- **No migrar los datos antiguos:** El nuevo sistema comienza a una fecha determinada y de ahí hacia adelante. Los datos históricos deben consultarse en la vieja aplicación. Puede

incluso construirse alguna interfase para que los usuarios puedan acceder a esos datos sin salir del sistema actual.

- **Migración de totales a una fecha:** No se migra la totalidad de los datos sino totales acumulativos, por ejemplo, saldos de inicio de cuentas contables, o total de sueldos anuales.

- **Migración parcial:** Se migran todos los datos desde algún periodo determinado, por ejemplo, del año en curso.

- **Migración total:** Se migra la totalidad de los datos anteriores, procesándolos para que sean manejables por el nuevo sistema. En estos casos es importante asegurar que los datos almacenados en el sistema anterior sean completos y pertinentes, de modo de no incorporar "basura" al nuevo sistema.

De lo anterior se desprende que la fecha elegida para la implementación del nuevo sistema tendrá impacto en el proceso de migración, ya que no es lo mismo comenzar a operar el 1 de enero que a mediados o fines de año. En el primer caso, por citar un ejemplo, no será necesario tener información detallada para el cálculo de aguinaldos, mientras que si se empieza después si será necesario cargar los datos de meses anteriores. Atado a esto, está claro que los posibles atrasos en la entrega del software pueden originar mayores costos en el proceso de migración.

7. Comenzando a operar

Una vez que el nuevo sistema ha sido correctamente instalado en los servidores y dispositivos de los usuarios, llega el momento de comenzar efectivamente la operación real. Este momento suele denominarse "go-live", "salir en vivo", "salida a producción", "pasar a producción" o algún término similar.

Por lo general, un nuevo sistema reemplaza a alguno que ya tiene la organización. Este podrá ser un software o un sistema de registro manual. En cualquier caso, la operatoria ya funciona y el nuevo sistema puede mejorarla, pero también podría causar problemas y detenerla. ¿Cómo conviene implementar el nuevo software para minimizar los riesgos? Básicamente encontramos 3 alternativas:

- **Cambio abrupto**: En este caso, en una fecha determinada el actual, sistema deja de estar operativo y se comienza a operar con el nuevo. Es la opción más arriesgada porque cualquier problema implicará que la organización deje de funcionar o que su información contenga errores. Si bien es la opción más arriesgada, en ocasiones es la única viable. Por ejemplo, si se requiere implementar un hardware específico y éste solo funciona con un nuevo sistema (molinetes de acceso, tótems de información, terminales de autoservicio, etc.). Además de tomar recaudos extras en la etapa de testeo, requiere que los usuarios estén previamente bien entrenados. Usualmente es posible que el sistema reemplazado pueda quedar operativo para consultar datos históricos.

- **Corrida en paralelo**: Durante un período de tiempo ambos sistemas funcionan en forma paralela. Es una opción segura, pero costosa. Requiere los usuarios deban atender y carguen

datos en forma simultánea en los dos sistemas. Ocasionalmente, debe contarse con hardware; espacio físico o personal adicional para que a la vez se puedan realizar la doble operatoria. Es una opción válida, por ejemplo, para sistemas de sueldos, donde puede hacerse una liquidación en cada aplicativo y comparar que en ambos lleguen al mismo resultado. Pero claro, sería inviable en sistemas masivos de facturación en mostrador.

- **Implementación por módulos**: En sistemas modulares es posible que los mismos se vayan implantando en etapas sucesivas. Por ejemplo, podría implementarse primero la facturación en mostrador, luego el módulo de presupuestos, el de cuenta corriente, etc. Este tipo de implementación minimiza los riesgos y facilita la capacitación, aunque la empresa debe mantener activos dos sistemas y algunos usuarios tendrán que operar simultáneamente en 2 plataformas diferentes. Es el tipo de implementación que se utiliza en desarrollo ágil donde las funcionalidades se van agregando de modo incremental y constante. Como contrapartida, no todos los sistemas se pueden implementar de este modo. Por ejemplo, un sistema módulo de facturación no se podría implementar de modo separado que el módulo de stock, ya que la operatoria del primero impacta en el segundo. Claro, también es posible implementar versiones reducidas de ambos módulos por separado.

En algunos sistemas on-line, donde no se cambia la base de datos ni la lógica del negocio, sino solamente la interfaz de gráfica es posible mantener por un tiempo ambas versiones. De este modo se le puede ofrecer al usuario que utilice la nueva versión o que continúe usando aquella a la que está acostumbrado. Los desarrolladores pueden ir ajustando detalles, ofreciendo ventajas, armando instructivos y mejorándola; hasta lograr que finalmente sea ampliamente aceptada por todos. Este tipo de implementaciones es frecuente verlas es aplicaciones masivas, web o redes sociales, donde se le permite al usuario mantener por un tiempo la actual interfaz, mientras va tomándole la mano a la nueva.

Por supuesto cada uno de estos métodos tiene ventajas y contras. La elección final tendrá que ver con el tipo de sistema; el tipo de empresa; la criticidad de los datos que maneje; las capacidades de los usuarios y, por supuesto, si es posible y conveniente mantener operativo el sistema anterior.

8. La resistencia al cambio

No es posible finalizar este capítulo sin considerar la resistencia al cambio que puedan tener los usuarios, ya sea por cuestiones reales (más trabajo en la carga inicial o la necesidad de aprender una nueva operatoria) y/o percibidas (el sistema va a reemplazar mi tarea, miedo a lo nuevo, perdidas de control sobre los datos).

Lo cierto es que, independientemente de las ventajas y facilidades que aportan los nuevos sistemas, también es verdad que modifican el modo actual en el que se desarrollan las tareas. Además, se modifican las relaciones de poder en la organización. Quien antes manejaba datos en papel, pierde el poder de ser el único dueño de esa información. Si alguien le presentaba al gerente un informe todas las semanas, quizá dejará de hacerlo (porque el gerente lo obtendrá de modo directo). El que durante años manejó un sistema y lo conoce de memoria, ahora deberá aprender a usar uno nuevo. El que antes utilizaba cómodamente una PC ahora tendrá que operar un dispositivo móvil.

Múltiples pueden ser los motivos por los cuales algunos usuarios se resistan a la implementación de un nuevo sistema. Y aquí la coordinación entre los desarrolladores y los responsables internos del sistema juegan un rol fundamental. Debe quedar claro que desarrollar un nuevo software no fue una idea alocada ni técnica, sino que fue una necesidad operativa y política de la organización, y que cuenta con el aval de la más alta dirección.

La participación de los usuarios en las etapas de relevamiento, análisis y pruebas, haciéndolos sentir parte de este nuevo proceso, escuchando sus opiniones y necesidades, ayuda disminuir resistencias. Si el usuario realmente comprueba que el sistema hace lo que pidió y que además le facilita la tarea, terminará comprendiendo los beneficios de su implementación.

También es importante que los jefes y máximos responsables participen en las tareas de capacitación y reuniones previas al lanzamiento. Esto ayuda a comprender que el nuevo sistema es algo prioritario para la organización. Ellos podrán explicar los beneficios que traerá y despejar las dudas.

En algunos casos será útil contar en el equipo de desarrollo personas con conocimiento especifico en gestión del cambio organizacional, para trabajar especialmente en propuestas que permitan disminuir esta resistencia y lograr un despliegue exitoso.

9. Bibliografía

IAN SOMMERVILLE: “Software Engineering”. 10ma edición. 2016. Pearson Education.
ROGER PRESSMAN: Ingeniería del Software. 7ta Edición. 2008. Ed. McGraw-Hill.

Apunte 9

Conceptos Fundamentales del Mantenimiento de Software

1. Introducción

Cuando se habla de construcción de software se suele pensar, como objetivo final, en poner en marcha una aplicación informática, cumpliendo las especificaciones, los plazos y los costos previstos para el proyecto. Sin embargo, este es solo un primer paso. Las organizaciones siguen evolucionando y los sistemas deben acompañar dicha evolución. Los cambios y agregados son imprescindibles. Una vez puesto en marcha, el software habitualmente entra en una etapa de constante evolución. A dicha etapa nos referimos cuando hablamos de mantenimiento.

En ocasiones, este proceso termina siendo mucho más largo e importante que el de la construcción original y, con el tiempo, el software termina siendo mucho más completo; con más funcionalidades y mucho más seguro y confiable que el primer desarrollo.

El mantenimiento va más allá de reparar errores, agregar nuevas funcionalidades y adaptar el software al contexto. En este apunte se buscará poner foco de atención sobre varios temas de interés respecto de esta etapa.

2. Definiciones

Definir qué es el mantenimiento de software parece sencillo: Es la modificación de la aplicación luego de que ha sido puesta en funcionamiento. Estas modificaciones pueden ser para corregirla, adaptarla a cambios del contexto o agregarle nuevas funcionalidades. Ian Sommerville, en su libro “Ingeniería de Software” habla de 3 tipos de mantenimiento:

- **Reparaciones de fallas (Correctivo):**
  Busca corregir errores de codificación, de funcionamiento o de incumplimiento de los requerimientos. Estos últimos son habitualmente los más costosos de solucionar.

- **Adaptación ambiental (Adaptativo):**
  Este tipo de mantenimiento se requiere cuando cambia algún aspecto del entorno del sistema, como el hardware, el sistema operativo, o incluso, nuevos requisitos legales. El software debe modificarse para que siga funcionando y respete las nuevas normativas.

- **Adición de funcionalidad (Perfectivo):**
  Este tipo de mantenimiento es necesario cuando varían los requerimientos del sistema en respuesta a un cambio organizacional o empresarial. La escala de los cambios requeridos en este caso suele ser mucho mayor que en los otros tipos de mantenimiento.

Claro que esta distinción es más teórica que práctica. Quizá solo tenga sentido a los efectos de un mejor entendimiento de la etapa. Lo cierto es no importa en qué se origina la necesidad de cambio, si estamos preparando una falla pequeña o agregando una nueva funcionalidad, siempre será imperativo responder a estas solicitudes de la mejor manera. Para ello volverá comenzar un nuevo proyecto de desarrollo de software, obviamente acotado, que permita introducir el cambio de modo correcto. Por supuesto, habrá que cuidar de no degradar el resto sistema y manteniendo su calidad.

3. ¿Mantenimiento o Garantía?

La enorme mayoría de las publicaciones de ingeniería de software describen 6 etapas en el ciclo de vida de un sistema: Análisis, Diseño, Codificación, Prueba, Puesta en Marcha y Mantenimiento. Algunos autores ni siquiera contemplan al mantenimiento y todo culmina cuando se implementó el sistema.

Pero… **¿Qué pasa con la garantía?** ¿Por qué somos tan exigentes en solicitar garantía en casi cualquier producto que compramos, salvo en el software? ¿Por qué solemos devolver el hardware defectuoso para que lo cambien por otro, pero no procedemos igual con las aplicaciones informáticas?

La respuesta quizá esté en la naturaleza propia de un software. Si compramos una zapatilla y notamos que la suela esta rajada, o que el color no es el mismo que pedimos, sabemos que eso no puede arreglarse. Al menos no sin comprometer al producto, una suela reparada no es lo mismo que una suela nueva de fábrica. Diferente es el caso del software. Este puede repararse y funcionar como nuevo. Si se le agrega un campo a una pantalla, la misma se comporta exactamente igual a que si ese campo hubiera existido en el desarrollo original. Lo mismo si se cambia de color. Quien opera una aplicación es incapaz de determinar si en esa pantalla hubo cambios o agregados posteriores al desarrollo original. No tiene sentido quedarnos con una computadora que no funciona o con una zapatilla rota, pero sabemos que el software puede repararse a nuevo.

Independientemente de esto, no siempre el software tiene garantía. O si la tiene es muy corta y en seguida se busca pasar a la etapa de mantenimiento. Es cierto… el mantenimiento correctivo se parece mucho a lo que uno pretendería de una garantía, pero con dos diferencias muy importantes:

1. **La garantía debería ser obligatoria, el mantenimiento no**. El desarrollador debe asegurar que el software funciona sin errores y, además, que ejecuta eficazmente aquellas tareas para las que fue programado. Todo aquel que recibe un software tiene derecho a reclamar que funcione, independientemente de si ese contratará o no un mantenimiento con el proveedor.

2. **El mantenimiento tiene un costo adicional para quien lo contrata, por fuera del precio inicial pautado. La garantía de funcionamiento es parte del desarrollo y, su costo, debería estar incluido en el precio original**. El usuario no tiene por qué pagar extras a lo acordado inicialmente para poder corregir errores o problemas que el software nunca debió tener.

Hay que comprender que tanto el proceso de testeo (donde se encuentran los errores), como el de depuración (donde se corrigen) forman parte de la creación de software. El cliente debe en realidad asumir el costo de ambos, pero no como un adicional, porque ninguno de los dos procesos mencionados es extra u optativo. Queda entendido que cuando alguien paga por un bien o servicio, esta debe funcionar correctamente. En el caso en que haya alguna falla el proveedor deberá hacerse cargo del costo de la reparación, sin exigir nada más al cliente. Por supuesto que, para hacer frente a estas reparaciones, quien vende suele cargar un valor en el precio que cubra esos eventuales costos.

Podría haber excepciones… muchos productos se compran y venden sin garantía, pero esos es algo que ambas partes saben y que están dispuestas a aceptarlo. Por supuesto, sería una opción demasiado arriesgada comprar un software que no pueda corregirse y confiar que este va a funcionar desde el inicio sin ningún error.

Para que se entienda mejor el concepto se puede plantear un ejemplo por el opuesto. Se comparan dos softwares para decidir cuál de ellos conviene comprar. El primero sale 1000$, pero el desarrollador nos asegura que cualquier error que aparezca en los primeros 6 meses será corregido sin cargo. El segundo sale 500$, no incluye ninguna garantía, aunque el programador nos dice que realiza todas las pruebas necesarias para que no haya errores y que además si los hubiera los corrige con un abono de mantenimiento obligatorio de $ 50 durante 6 meses. ¿Conviene elegir el segundo paquete, porque es más barato que el primero? Bueno, dependerá de la evaluación que se haga al segundo proveedor y los riesgos que se están dispuestos a asumir, pero está claro que un software que vale la mitad y que no tiene garantía de funcionamiento es posible que tenga muchísimos problemas. Además, a los costos de reparación hay que sumarle los costos por no poder operar correctamente el sistema.

En definitiva, es importante redefinir el proceso de ciclo de vida del software para agregar esta etapa: I**ngeniería de Requerimientos, Diseño, Codificación, Prueba, Despliegue, Garantía y Mantenimiento**

Hay que recordar también que, por las características propias del software, la mayoría de los problemas aparecen en los primeros tiempos de uso. Una vez que se estabiliza, no se estropea y funciona sin errores hasta que finaliza su vida útil. A diferencia de los productos tradicionales, donde una garantía corta cubre al fabricante de los problemas vinculados al desgaste, en software se pueden dar garantías muy largas, incluso a perpetuidad, a sabiendas que los errores generalmente afectan solo los primeros tiempos de uso.

4. Prueba versus Garantía

Uno de los fundamentos de aplicar Ingeniería de Software al desarrollo, es poder producir aplicaciones confiables y sin fallas. La prueba tiene como objetivo detectar los posibles errores, antes de que el sistema sea puesto en producción. Está probado que los procesos de calidad minimizan la posibilidad de que los errores lleguen a al usuario final. En este contexto… ¿qué tan importante es la garantía?

La garantía justamente le garantiza al usuario, valga la redundancia, de que todos los controles de calidad se cumplieron. Si se relajaron, el desarrollador se verá obligado a enfrentar mayores costos para resolver los problemas que aparezcan. Por este motivo, es de esperar que el desarrollador procure detectar los problemas antes de entregar el software. De ese modo se ahorrará los costos de resolverlos luego, en el período de garantía. Siempre es mucho más barato solucionar problemas mientras el software está aún en desarrollo, que una vez que este fue implementado.

A como sea, el usuario siempre sale beneficiado si el software tiene garantía, no solo porque no tiene que pagar para solucionar eventuales fallas, sino porque es el mejor modo de asegurarse

que el desarrollador buscó hacer las cosas de la manera correcta y puede suponer que tiene un software probado y seguro para operar.

5. La dificultad en los contratos de mantenimiento

La etapa de mantenimiento implica un acuerdo contractual entre partes, con más o menos formalidades. Por un lado, se fijarán que cosas estarán cubiertas en esta etapa y, por el otro, se establecerá una prestación monetaria que cubra los costos. En pólizas de garantías comunes, estamos acostumbrados a encontrar un extenso texto de “letra chica” con un motón de exenciones y limites, que protegen al consumidor, pero tambien al fabricante de cuestiones inculpables. Con el software, la cuestión tambien es compleja

Es necesario comprender que, a diferencia de los productos tangibles, el software puede presentar desafíos únicos en términos de mantenimiento. Puede requerir actualizaciones periódicas, parches de seguridad, corrección de errores y soporte técnico continuo. Además, es importante considerar que el mantenimiento del software puede ser afectado por factores externos, como cambios en el entorno tecnológico o requisitos del negocio, lo que puede requerir ajustes en los acuerdos de mantenimiento existentes.

¿Qué precio habría que fijar para que cubra libremente todo esto? ¿En un mes se pueden pedir decenas de cambios? ¿Cualquier adaptación está incluidas? ¿Toda nueva funcionalidad estará cubierta por el canon pagado por el mantenimiento?

Podrían establecerse límites. Por ejemplo, se podría establecer que todos los errores serán solucionados sin límite; que además se destinaran equis cantidad de horas por mes para cambios y mejoras y que las adaptaciones se presupuestarán aparte, a un valor hora especial. Incluso pueden establecerse alguna banda o escala, por ejemplo 50 horas gratis, 50 horas a un valor bonificado y el resto a un valor full. Además, pueden pautarse plazos de resolución según la urgencia de caso. Quizá los errores puedan resolverse en 24 horas, mientras que los demás cambios no tienen tanta urgencia y pueden demorar, por ejemplo, una semana.

Pero la realidad de nuevo complica las cosas. ¿Si algo no fue pedido correctamente… es un error o una modificación? Si el software hace algo, pero de un modo distinto a como el usuario lo quiere… ¿es error o modificación? ¿Cómo sabe el usuario que las modificaciones solicitadas están dentro el tope de horas mensuales incluidas en el precio? Si el gobierno modifica de un día para el otro la retención de ganancias… ¿De quién es la responsabilidad de que el sistema este modificado de modo urgente? ¿Y si cambia el sistema operativo o el cliente decide reemplazar las PCs, por dispositivo móviles?

Los contratos pueden ser más o menos detallados. Incluso elaborados por especialistas letrados y hasta ser auditados. Pero los más importante es que las partes actúen de buena fe y que entiendan que esta etapa es un juego de suma cero. Si alguna de las partes tiene beneficios extraordinarios, es porque la otra los pierde. El desarrollador tiene que poder cubrir razonablemente sus costos con en precio fijado, y la empresa que opera el software tiene que recibir prestaciones que justifiquen ese monto que abona periódicamente. La confianza, la transparencia y el dialogo siempre ayudan. Por ejemplo, se le puede explicar al usuario que el cambio solicitado

es complejo y que se hará, pero que el próximo mes no se aceptarán otros cambios. O que el error marcado no es tal, que se corregirá, pero no con la urgencia que se corrigen otros errores.

La relación cliente-desarrollador establece una alianza estratégica ante ambas partes. A los dos les conviene que el software ande bien y que evolucione. El desarrollador evaluará seguramente la posibilidad de participar de otros desarrollos que le presente el cliente. A su vez, este se beneficiará de seguir trabajando con un proveedor confiable. Es por eso que no siempre la garantía (o el mantenimiento) debe ser interpretado solamente bajo la fría letra de un contrato. A veces conviene otra dinámica de apuesta a futuro.

Otra de las cosas que deben acordarse es cuándo finaliza el contrato de mantenimiento y en qué condiciones se termina el vínculo. No resulta razonable plantear un mantenimiento indefinido. Llegará un momento que el usuario ya no pida más cosas y que no resulte conveniente seguir teniendo recursos ociosos a la espera de que se produzcan. Pero tampoco es posible que, por alguna razón intempestiva, se deje al usuario con un sistema que ya no puede seguir modificando. Es posible que, al finalizar un primer contrato, que aparezca otro que, por ejemplo, asegure a la organización usuaria que seguirá recibiendo actualizaciones urgentes, pero ya no podrá pedir cambios o modificaciones. Nuevamente, el contrato de mantenimiento debe ser beneficioso para ambas partes para que decidan continuarlo.

En los casos de software genérico (Windows, por ejemplo) el ciclo de soporte se encuentra claramente definido. Las distintas versiones van recibiendo diferentes tipos de mejoras, correcciones y actualizaciones por un tiempo determinado. Luego, pueden pasar a una etapa donde solo reciben parches de seguridad. Finalmente, algún momento ya los sistemas ya no reciben ningún tipo de soporte ni asistencia.

6. El mantenimiento ocurre al final, pero se piensa al inicio

Una de las cosas que deben definirse al comienzo de un proyecto es cómo será la evolución del sistema. Si se prevé que software cambiará luego de su primera versión, debe programarse de modo tal que estos cambios sean factibles, sencillos de realizar y que no degraden el sistema

La utilización de parámetros externos; la división modular; la programación sencilla; la documentación detallada; la inclusión de funciones de actualización automática; el uso de estándares; la posibilidad de que el usuario pueda agregar funcionalidad (por ejemplo, crear sus propios listados) son, entre otras, cosas que facilitan el mantenimiento posterior. **Cuando se diseña el sistema y en especial cuando se lo programa, se deben adoptar todas las buenas prácticas que faciliten el mantenimiento posterior**.

Ejemplificando, si se prevé que el sistema se actualice on-line, es deseable que esté que pueda actualizarse por módulos. No sería válido pensar que para corregir un error en la calculadora de Windows deba actualizarse todo el sistema operativo. La calculadora debe poder ser reemplazado por una versión corregida en forma fácil e independiente. Del mismo modo… si puedo cambiar la tasa de IVA en un archivo de parámetros, la actualización del sistema sería mucho más fácil que si el 21% forma parte del propio código fuente y debe cambiarse en cada lugar donde opere.

Obviamente, diseñar el software con funcionalidades que le permitan recibir correcciones automáticamente, o en forma remota, va a facilitar los cambios y reducir los costos de la etapa. Las actuales tiendas de aplicaciones permiten que, con solo subir una nueva versión, esta se despliegue en forma automática, y casi inmediata, a todos los usuarios. Sin embargo, en software empresarial esto no es tan fácil. Los despliegues automáticos no siempre son deseables ya que pueden interrumpir flujos de trabajo, o afectar otras partes críticas del sistema, que hoy funcionan bien, pero podrían dejar de hacerlo tras la automatización. El mantenimiento remoto también puede abrir brechas de seguridad y la posibilidad de que alguien use esas funcionalidades para filtrar código malicioso. Todo esto debe ser cuidadosamente revisado y tratado con toda la rigurosidad que el tema merece.

Por otro lado, no todo el software se piensa para para ser actualizado en el futuro. Hay software embebido dentro de ciertos dispositivos o programas de un solo uso (por ejemplo, un software que se desarrolla para alguna tarea puntual y única, como puede ser un conversor de archivos entre 2 sistemas diferentes). Pero, aun así, siempre es conveniente programarlo pensando en que cosas se le pueden agregar para que funcione a futuro en algún otro caso similar.

Los mayores costos que representa pensar de este modo (es más caro hacer software parametrizable que no hacerlo) terminan produciendo mayores beneficios a la hora de tener que corregir un error.

7. La preservación de los datos

Una de las cuestiones centrales a la hora de actualizar software es preservar los datos del usuario. Y esto tiene 2 cuestiones vinculadas:

- La primera y más obvia… **los datos deberían almacenarse en un archivo distinto de la aplicación principal**. De este modo, puedo hacer modificaciones en el programa, luego reemplazar el archivo ejecutable por el nuevo y los datos quedan intactos. Esto no ocurre si la lógica del sistema y los datos están en el mismo archivo. En Excel, por ejemplo, si yo modifico la planilla con nuevos cálculos y fórmulas no puedo simplemente reemplazar luego la planilla del cliente sin que pierda su información.

- Aun cuando los datos se encuentren separados de la lógica que los maneja, hay veces que los cambios solicitados requieren **cambios en la estructura de dichos datos**. Por ejemplo, cuando se necesita agregar un campo, modificar un tipo de datos o agregar una tabla. De nuevo en este caso no es posible pisar el archivo de datos con uno con la estructura corregida, sino que hay que hacer los cambios de modo que se preserven los datos ya cargados. Seguramente será necesario en estos casos realizar alguna aplicación para pasar los datos de la vieja estructura a la nueva, o realizar los cambios manualmente sobre la base de datos original.

En este sentido, siempre es deseable que cualquier nueva implementación, sea automática o manual, haga primero una copia de seguridad completa del sistema actual, antes de correr los procesos de actualización. También es importante siempre existan los mecanismos para permitir

volver a utilizar la versión anterior en el caso de que algo no funcione como se espera con el software actualizado.

8. Mantenimiento y gestión de sistemas heredados

Un tema particular para considerar son los sistemas heredados. Son sistemas que ya no reciben nuevos cambios ni a los que se le agregan funcionalidades, pero es necesario mantener ya que prestan algún servicio a la organización. También debemos incluir en este caso a los sistemas desarrollados por terceros que debamos administrar (incluso continuar con su mantenimiento, por ejemplo, corrigiendo errores si los hubiera.) En estos casos tenemos una serie de alternativas para analizar y ver cuál es la más conveniente:

- El caso más sencillo es donde el sistema ya no va a realizar un aporte efectivo a los procesos empresariales, o casi todas las funcionalidades principales han sido absorbidas por otras aplicaciones. En estos casos puede ser factible desechar completamente el sistema y quitarlo de los ambientes productivos.

- Otro caso es aquel sistema que es relativamente estable, no se le incorporan nuevas funcionalidades, no aparecen errores o problemas graves y los usuarios tienen relativamente pocas peticiones de cambio. Si es así, quizá sea conveniente continuar dando servicios de mantenimiento a este sistema.

- Cuando la calidad del sistema se haya degradado por el cambio constante y cada vez se dificulta más modificarlo, es hora de pensar en un recambio. Aparecen dos alternativas: Una, dejar la aplicación tal cual y sin cambios y construir otra en paralelo que tenga las nuevas funcionalidades. La otra opción, obviamente, es que el nuevo software termine reemplazando por completo al anterior.

- También existen factores externos que, en ocasiones, obligan a un reemplazo aun cuando el software preste servicios de un modo satisfactorio. Cuestiones como actualizaciones de hardware, vencimiento de licencias de software o simplemente cambios muy significativos en la operatoria de la organización, determinan que sea necesario pensar en un nuevo desarrollo.

9. Costos del mantenimiento

El mantenimiento de software suele representar una parte muy importante de los costos de las áreas de sistemas. En sistemas empresariales los costos suelen ser comparables con los de desarrollo inicial. Es decir, se gasta tanto dinero en mantener y modificar un sistema ya funcionando, como el que se gastó en su momento para programarlo.

Además, hay situaciones particulares que aumentan tanto la dificultad de modificar los sistemas, como sus costos y ya no resulta económicamente viable seguir manteniendo el software. Entre ellas podemos mencionar:

- El desarrollo original no se pensó para ser adaptable o modificable, con lo cual es más complejo y caro cambiarlo.

- Los continuos cambios, parches y agregados, degradan el software al punto que cada nueva corrección aumenta el riesgo de que el sistema deje de andar. Un caso que ejemplifica este punto es cuando se llegan a límites que originalmente parecían lejanos. Por ejemplo, cuando ya no se permiten agregar más campos a una tabla de una base de datos.

- La documentación está incompleta, no existe o no está actualizada.

- El paso del tiempo y los cambios en el equipo de desarrollo que hace que los programadores originales ya no estén, y quienes los reemplazan quizá no tengan el conocimiento necesario sobre aquel sistema.

- Prácticas deficientes que, tiempo atrás servían, pero ahora se vuelven ineficientes y obsoletas.

- La empresa ya no cuenta con programadores que utilicen el lenguaje de programación el que fue programado el software. Incluso, en ocasiones, tampoco son fáciles de encontrar en el mercado laboral.

- Se requieren mantener estructuras viejas de hardware o sistemas operativos. A veces, la virtualización permite simular entornos pasados, pero, de no lograrlo de ese modo, quizá necesita operar un hardware viejo que ya no se consigue o que es costoso de mantener (por ejemplo, un mainframe).

- Se requieren licencias operativas o de la base de datos que por algún motivo ya no se conviene mantener (por ejemplo, casos de licencias cotizadas en dólares que se vuelven impagables en determinado momento)

En estos casos siempre es conveniente evaluar si los costos de mantener un sistema no son mayores que del de construir uno nuevo.

10. Mantenimiento y Desarrollo Ágil.

Una de las características distintivas del desarrollo ágil es que los sistemas están cambio permanente. Los usuarios proponen cambios, mejoras y nuevas funcionalidades en el sistema que ya tienen operando, y los desarrolladores las van programando e implementando en forma de incrementos periódicos y permanentes (sprints en Scrum).

Estos conceptos se asemejan mucho al funcionamiento la etapa de mantenimiento de un software. Podríamos pensar que, en determinado momento, un proyecto desarrollado siguiendo un modelo dirigido por un plan, pasa a comportarse como un desarrollo ágil: El sistema queda abierto

al cambio constante y permanente. La verdad, no se equivoca demasiado quien piensa de este modo.

Lo que hay que definir, entonces, es que estrategia de desarrollo se utiliza durante esta etapa de mantenimiento. Puede utilizarse un modelo lineal secuencial reducido donde, cada vez que surge una necesidad de cambio, se elabora un mini proyecto de desarrollo donde de pasan por todas y cada una de las etapas del modelo. Puede pensarse en un modelo en espiral infinito, donde en cada iteración se agrega funcionalidad, analizando la conveniencia o no de tal cambio (análisis de riesgo) Pero también es factible convertir ese proyecto en un desarrollo ágil y aplicar un modelo scrum a partir de la implementación, no importa cómo se desarrolló originalmente.

Cualquiera alternativa que se elija será válida. Pero lo que hay que tener en cuenta es que hay que seguir un modelo de desarrollo, el que fuera. Ágil o dirigido por un plan, el que mejor se adapte al momento. Solo siguiendo un modelo se puede cumplir con todos los pasos que aseguren la calidad del cambio y, por lo tanto, la calidad del nuevo sistema.

Siempre hay que recordar que, ante cualquier modificación, el sistema deberá pasar nuevamente por toda la etapa de pruebas, testeando no solo la sección cambiada, sino todo el resto del software para asegurarse que no ha quedado comprometido al algún otro punto de este.

11. El dilema del mantenimiento infinito

Una de las funciones del mantenimiento del software (o del desarrollo ágil) es permitir que el sistema acompañe la evolución de las organizaciones, los cambios externos, se adapte a nuevo hardware o dispositivos y pueda utilizarse en nuevos sistemas operativos. Pero esto nos lleva a una serie de situaciones que debemos considerar.

La primera de ellas el software tiene límites. Al igual que un edificio… se le pueden agregar pisos, se los puede modificar, pero llega un punto en que los cimientos no alcanzan. Agregar una construcción en la terraza puede hacer que el edificio colapse. En el software pasa algo similar. Cada cambio que se le introduce lo va mejorando, lo va ampliando, pero también va degradando su estructura o su base de datos. El sistema puede volverse lento, pesado y hasta complejo de usar. Llega un punto donde es necesaria una reingeniería completa del sistema para que volver a tener un sistema optimizado y listo para continuar recibiendo mejoras.

¿Y qué pasa con las APPs o con sistemas genéricos no atados a una determinada empresa? ¿Qué pasa cuando el software en cuestión depende de desarrolladores que tienen como objetivo seguir agregando funcionalidades y mantenerlo activo? Acá es donde muchas veces se cae en una trampa peligrosa: sigo actualizando el software agregándole cosas que nadie pidió ni necesita, o lo dejo tal y como esta (asumiendo obviamente que funciona bien y sin errores), corriendo el riesgo de que se piense que es una aplicación obsoleta y abandonada.

Muchas veces las aplicaciones agregan funcionalidades de manera periódica, la mayoría de las cuales pocos utilizan. Obviamente esto hace que la aplicación se vuelve más demandante de recursos, más pesada en su uso, con una interfaz que comienza a complicarse. Llega un punto que los usuarios prefieren pasarse a otra aplicación más simple que cubre específicamente nuestras necesidades.

Esto se agrava en ocasiones donde el equipo encargado del mantenimiento tiene menos recursos que el que programó la aplicación original o, incluso, es otro. Y, además, se enfrenta a una interesante dualidad: Por un lado, su objetivo es mejorar el sistema y agregarle nuevas funcionalidades; por el otro, los usuarios están acostumbrados a una nueva versión que cumple con sus requisitos y no necesitan cambiar. Eso sin contar con que, al aumentar las funcionalidades, y con ello las líneas de código, la posibilidad de fallas y errores aumenta.

Se pueden dar decenas de ejemplos de sistemas, aplicaciones y juegos que los usuarios abandonan por perder la simpleza original o por exigir dispositivos cada vez más potentes para su uso. O simplemente por imponer cosas que el usuario no necesita y no le aportan nada especial. Los "Mosaicos" o "Tiles" que pretendieron reemplazar al clásico menú de inicio de Windows son un ejemplo. Estos tenían todo el sentido en pantallas táctiles, pero no en equipos tradicionales. Las quejas y el pedido para que vuelva el clásico menú fue masivo. Los "Live Tiles", esos mosaicos animados que conviven hoy en Windows 10 con los íconos tradicionales del menú, son rechazados por muchos usuarios, que los perciben como un innecesario consumo que recursos. Todo indica que dicha funcionalidad desaparecerá en la próxima versión del sistema operativo.

Dos reglas no escritas ni formales pueden aplicarse en estos casos.

1. "*Lo que anda bien, no debe arreglase*". Si una aplicación funciona si errores y cumple con las necesidades del usuario, lo mejor es dejarla tal y como está. Cualquier cambio en el código, por menor que sea, implica el riesgo de que aparezca alguna falla. Hay veces que el usuario valora actualizaciones frecuentes, pero en general prefiere aplicaciones estables, a cuyo uso está familiarizado y cuyos sus resultados son confiables. Imaginemos el caso de que un auto autónomo choque, sólo porque a un ingeniero de software se le ocurrió que el código debía optimizarse.

2. "*El usuario siempre tiene razón*". Independientemente de lo que piense el líder de proyecto o el grupo desarrollador, el usuario es el que define, en última instancia, lo que necesita o no de un sistema, y el modo en el cual quiere resolverlo. Si el equipo de desarrollo piensa que una nueva interfaz puede modernizar nuestra aplicación y la puede hacer más intuitiva, pero el usuario cree lo contrario, no tiene sentido forzar el cambio.

Por supuesto, además del usuario u el equipo de desarrollo, otros factores intervienen en un sistema. A veces los cambios son necesarios porque debemos adaptar el software a un nuevo sistema operativo o a nuevos estándares de desarrollo. A veces debemos hacerlo más seguro, o debemos incorporar controles que las autoridades de contralor exigen. En esos casos el cambio será inevitable, pero habrá que explicar bien al usuario el origen de este y cuáles serán los beneficios que traerá su implementación.

12. ¿Todo el software requiere mantenimiento?

En contraposición con el punto anterior, cabe también preguntarse si todo el software que se desarrolla debe ser modificado en algún momento de su vida útil. La respuesta es que no, aunque, generalmente, esto es una excepción.

Existe software que podríamos “de única vez” y que tiene un propósito puntual y específico que una vez cumplido, deja de existir. Por ejemplo, el software que se realiza para migrar datos de un sistema a otro. Una vez realizada la migración ya no tiene más utilidad.

También existe un montón de aplicaciones que son instaladas en dispositivos no actualizables, como relojes, maquinaria, juguetes, automóviles, televisores, calculadoras, electrodomésticos (No confundir con la versión “Smart” de dichos dispositivos). Ese software sigue funcionando mientras el dispositivo siga utilizándose. Hay programas que, si bien reciben periódicamente actualizaciones y mejoras, sus primeras versiones siguen funcionando tan bien como el primer día. O incluso mejor, dado que los procesadores más poderosos de hoy día pueden mejorar el funcionamiento. El buscaminas o el solitario de Windows 95 son dos ejemplos, pero hay miles de aplicaciones de antaño que podemos seguir usando. Ya se ha dicho anteriormente que el software no se desgasta. Una vez alcanzada su madurez, puede seguir ejecutándose de por vida. En todo caso, un software deja de poder utilizare solo cuando ya no es soportado por las computadoras actuales.

¿Pero qué pasa con los softwares comerciales y de las organizaciones? Acá la cosa es diferente. Uno no se imagina que un software, por mejor programado que esté, pueda brindar servicios a una organización durante 20 años. Pero no por un defecto del software en sí, sino porque los procesos organizacionales cambian y vuelven obsoleto al software. También aparecen nuevas computadoras, dispositivos y sistemas operativos. Entonces hay que terminar por optar entre mantenerse con hardware y software del pasado, o dar el paso y actualizar los sistemas.

Recientemente, tanto Apple como Google, comenzaron a retirar de sus tiendas aplicaciones que llevan tiempo sin actualizarse (además de otros factores como la cantidad de descargas). ¿Es razonable pensar que las Apps deban ser actualizadas o condenadas a desaparecer? Mas allá de las quejas y de las excepciones, la realidad es que es tiene sentido que así suceda. Los dispositivos cambian, las pantallas incorporan, por ejemplo, los famosos “notch” en la parte superior para dar lugar a la cámara, aparecen y desaparecen botones. El propio sistema operativo va a agregando funcionalidades y capas de seguridad. Un software que utiliza componentes del sistema operativo que son obsoletos o inseguros necesariamente deben ser forzados a cambiar. Por supuesto que también hay funcionalidades sugeridas, y que los desarrolladores pueden optar por incluir o no, como nuevos diseños de los menús, nuevas paletas de colores, modo nocturno y una gran cantidad de etcéteras.

En resumen, por lo general el software recibe mantenimiento durante toda su vida útil, ya sea para corregir defectos o para ir acompañando los cambios en el entorno, en la organización, en el sistema operativo o en los dispositivos en los que opera, aunque no es algo que no es algo que obligatoriamente deba pasar.

13. Bibliografía

IAN SOMMERVILLE: “Software Engineering”. 10ma edición. 2016. Pearson Education.
ROGER PRESSMAN: Ingeniería del Software. 7ta Edición. 2008. Ed. McGraw-Hill.

Apunte 10

Conceptos Fundamentales de la Estimación, Costeo y Precio del Software

1. Introducción

Una de las funciones principales de cualquier administrador de proyectos es la de estimar o presupuestar. Estimar qué recursos se van a necesitar, a partir de qué momento, y por cuánto tiempo serán utilizados; es la base necesaria para determinar el costo de un desarrollo de software. Además, armar un plan y presupuesto, permite al administrador tenerlo bajo control, anticiparse a posibles desvíos, y cumplir los plazos y costos comprometidos.

El desarrollo de software tiene sus particularidades. La posibilidad de ir agregando, ampliando y modificando el producto sin degradarlo, como también el hecho de ser una actividad de desarrollo intelectual hacen, que la estimación se vuelva una tarea a veces casi imposible. El objetivo de este trabajo es exponer la problemática y brindar algunas posibles alternativas de solución.

2. Estimación de software

La estimación de un proyecto es quizá la parte tarea más compleja que debe desarrollar el encargado de gestionarlo. En determinadas industrias, en función de los estándares y la experiencia, existen tablas estimativas que facilitan la tarea. Hay tiempos estimados de cuanto lleva construir una casa; las automotrices saben cuánto les lleva el proceso de fabricación de un vehículo, pero… ¿existe tal cosa en el software?

En principio no. Y si existen… solo serían válidas si el proyecto de desarrollo y el software a construir fuera muy parecido a otros anteriores, cosa que rara vez ocurre en la realidad. Y aun cuando así sea, casi con seguridad va a correr en un entorno de hardware distinto o incluso sobre versiones actualizadas del sistema operativo.

¿Cómo pueden estimarse, de un modo razonable, los recursos que se utilizarán en la construcción de un software? Aunque estimar es tanto un arte como una ciencia, esta acción no necesariamente puede llevarse a cabo de manera fortuita Para comenzar, pueden destacarse dos técnicas para aplicar:

- **Técnicas basadas en la experiencia**: La estimación de los requerimientos futuros se basan en la experiencia del administrador con proyectos similares anteriores. En esencia, es el propio administrador quien emite un juicio informado de cuáles serán los recursos necesarios. Para que esta técnica sea lo más precisa posible, es necesario que en los proyectos anteriores se haya recolectado toda la información cuantitativa necesaria, de modo que sirvan para compararlo con el proyecto que se habrá de estimar. La utilización de métricas, sin duda, ayudan en esta tarea.

- **Modelos algorítmicos de costos:** En este caso se usa un enfoque formulista para calcular recursos, en base a estimaciones de atributos del producto (por ejemplo, el tamaño), como así también las características del proceso (por ejemplo, la experiencia del personal implicado). Las empresas de desarrollo ms grandes suelen

desarrollar sus propios modelos, aunque existen algunos modelos estándares como el COCOMO[43],

Estas técnicas, y la estimación en general, se ven afectadas por una serie de fatores:

Complejidad del proyecto: Cuanto más complejo sea el proyecto, obviamente más difícil será estimar el esfuerzo necesario para llevarlo a cabo. Sin embargo, complejidad, no es una medida absoluta. Por el contrario, se ve relacionada con la experiencia pasada. El primer desarrollo de una aplicación de comercio electrónico seguramente será excesivamente complejo. Pero si el equipo de desarrollo ha construido varias similares anteriormente, la complejidad no será percibida como importante.

El tamaño del proyecto: Cuanto mayor sea el proyecto, la cantidad de componentes a construir y la interdependencia entre ellos aumenta fuertemente. Dividir el proyecto en etapas puede ayudar en estos casos.

La incertidumbre del contexto: La posibilidad de que los requerimientos varíen a medida que avanza el proyecto, por cuestiones internas o externas, también pueden afectar la estimación. No siempre es posible pautar desarrollos cerrados que no admitan modificaciones, menos en proyectos grandes que se extiendan el en tiempo.

La información histórica: Se ha dicho que la experiencia es fundamental a la hora de estimar. Pero… para que esta experiencia sea confiable es preciso que se construya sobre datos válidos. Debo cuantificar y registrar datos del desarrollo actual, no solo para medir su avance, sino para que me sirva como parámetro para proyectos futuros. Los registros de información histórica se vuelven muy valiosos.

¿Y qué cosas deben estimarse? Básicamente **recursos** y el **período de tiempo** durante el cual usaré esos recursos. Será necesario estimar:

- **Recursos humanos**: La actividad desarrollo de software es una actividad que depende fundamentalmente de los recursos humanos y de su capacidad intelectual. Será necesario contratar líderes de proyecto, analistas, diseñadores, programadores, testar, implementadores, especialistas en riesgo y seguridad. Si el proyecto es más grande, aparecerán seguramente muchas más funciones (analistas de experiencia de usuario, arquitecto se software, analistas de calidad, entre otros). Para cada uno de los recursos necesarios deberá estimarse **la cantidad**, **los conocimientos requeridos y donde estarán ubicados físicamente.**

- **Recursos ambientales**: Contemplan tanto los espacios físicos, como las computadoras, redes, internet y las licencias del software de desarrollo. En el caso de que el software se construya para algún hardware en particular (por ejemplo, un determinado modelo de celular, o alguna maquinaria, por ejemplo, una tragamonedas), será necesario proveer al equipo de desarrollo acceso al mismo.

[43] Puede consultarse punto 23.6 del libro “Software Engineering” Teenth Edition, de Ian Sommerville, para una profundización del tema de modelos empíricos de costos y el modelo COCOMO.

- **Componentes[44] a desarrollar:** Será necesario construir **nuevos componentes,** podrán reutilizarse **componentes ya desarrollados** (por ejemplo, el componente que gestiona el login de usuarios, o tablas ya cargadas o la documentación de ciertas funcionalidades), o también modificar alguno y adaptarlo al nuevo desarrollo.
  También existe la posibilidad de comprar o conseguir externamente componentes completos e incluirlos en el nuevo sistema, por ejemplo, un carrito de compras o un botón de pago.

En cualquier proyecto, la precisión de la estimación varía en el tiempo. A medida que el proyecto avanza y se desarrollan ciertas tareas, la incertidumbre va disminuyendo. No es lo mismo tratar de estimar los tiempos que llevará un desarrollo antes de iniciar la programación, con la dificulta que lleva determinar cuánto se tardará en codificar un módulo, que una vez que esta etapa está terminada.

A medida que el proyecto avanza, la certeza también. En el extremo, una vez que el proyecto está terminado, la estimación será 100% cierta. Pero claro, también es inútil estimar algo que ya pasó. Cuanto más temprana sea la estimación más útil será para el proyecto, pero mayor también el margen de error y la posibilidad de no cumplirla. Este tema será vuelto a analizado más adelante con la presupuestación.

3. Restricciones

No es del todo cierto que el software se estime libremente. Habitualmente el software debe ser implementado en una fecha determinada, por ejemplo, antes de fin de año, o para la apertura de un nuevo local. Tampoco es cierto que el precio del software pueda ajustarse libremente una vez

[44] El término componentes no solo hace referencia a programas de computación. Muchas otras cosas deben construirse, como manuales, datos precargados, documentación, diseños de pantalla, etc.

que se valorizaron los componentes. Por el contrario, muchas veces la organización cuenta con un determinado presupuesto y hay que ajustarse a él.

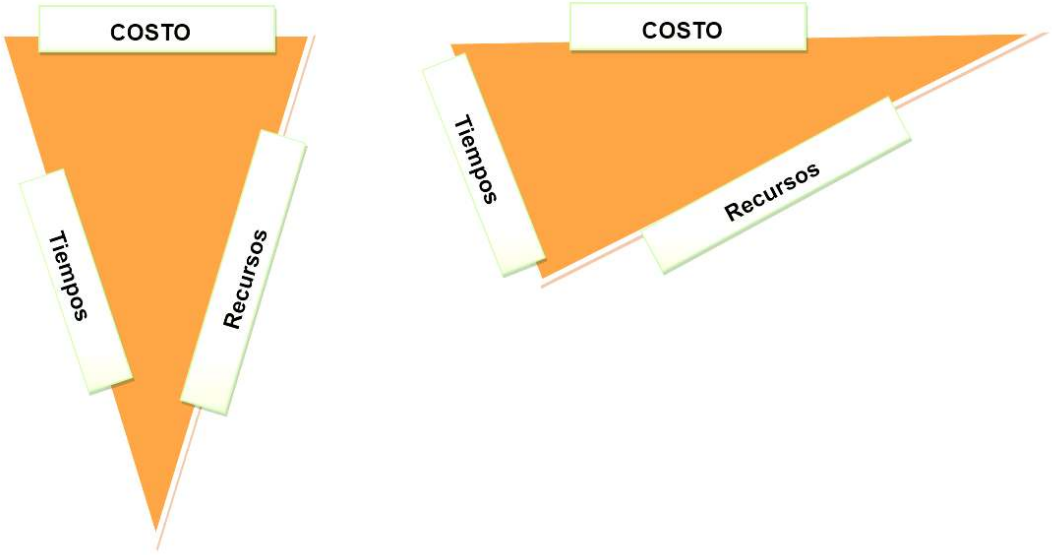
Tampoco los recursos humanos disponibles son ilimitados. No puede contratarse libremente a todo el personal que se desee[45]. Además, muchas veces, las empresas desarrollan en paralelo más de un sistema, y deben ir asignando a un proyecto los recursos en tanto se liberen de otro.

Es interesante considerar a las restricciones como lados de un triángulo. Esto permite aumentar o disminuir uno de sus lados, a expensas de los otros dos. Es decir, si fuera necesario acortar la duración de un proyecto, habrá que contratar más recursos y aumenta el costo. Si, por ejemplo, se requiere reducir costos, se podrá aumentar el tiempo, permitiendo eventualmente los recursos se compartan con otros proyectos.

Las siguientes figuras muestran cómo es posible acotar costos, pero alargando el tiempo necesario y el uso de recursos. O como se puede reducir tiempos, pero con más costo y tiempos[46].

[45] Los recursos humanos del área de IT son altamente demandados a nivel local y regional. Los recursos que demanda la industria son, en muchos casos, mayores a los profesionales que se egresar anualmente en el sistema universitario. También es difícil capacitar personal y/o reemplazar al existente. Hay una curva inicial de aprendizaje importante hasta que el nuevo integrante se alinea con el modo de trabajo y los estándares de la organización. A menudo, el tiempo que demora un programador en rendir del mismo modo que el que reemplaza, suele ser mayor que lo que necesita el proyecto.

[46] Mas adelante se observará que el costo está relacionado con el tiempo en el que se usen los recursos. Sin embargo, es posible pensar en utilizar recursos en lugar de un programador senior full time, 2 programadores junior part time. Seguramente estos dos programadores serán más baratos que el primero, pero también demorarán más tiempo.



Pero… ¿Qué pasa si en el proyecto deben contemplarse la vez 2 restricciones? Siguiendo con el ejemplo de los triángulos, algebraicamente no será posible construir un triángulo con igual superficie si se reducen el largo de 2 de sus lados. En sistemas no hay una imposibilidad matemática, pero sería sumamente extraño lograr un sistema en menos tiempo y con menos recursos, o tratar de disminuir a la vez los costos y los tiempos.

Sin embargo, hay otra alternativa. Es posible reducir a la vez y de modo proporcional todos los lados del triángulo si se acuerda que el sistema tenga menos alcance y, por lo tanto, requiera construir menos código.

Hay que admitir que otro modo es resignar calidad. Por ejemplo, si no se hicieran controles de calidad o se limitaran las pruebas, el software será más barato, utilizaría menos recursos y llevaría menso tiempo. A esta altura del libro está claro de lo riesgoso e inconveniente que puede ser eta alternativa.

4. Costo del software

El desarrollo de software no sigue los esquemas propios de un desarrollo tradicional. Sus costos, por lo tanto, no pueden ser calculados siguiendo la fórmula de costeo estándar:

Costo de producción = materia prima +
mano de obra directa +
costos indirectos de fabricación.

Para empezar, no hay materia prima. Y si la hay, porque rara vez los desarrollos se inician desde cero, sino que reutilizan algunos componentes, estos no tienen costo (o ya fueron cobrados cuando se desarrollaron originalmente).

Por otro lado… los costos indirectos de fabricación muchas veces son irrelevantes[47]. Si bien existe un producto concreto (el software) hay más semejanzas con los servicios profesionales, que con la fabricación de bienes.

¿Cómo se calculan entonces los costos de un proyecto? Se dijo anteriormente que para desarrollar software es necesario contar con determinadores recursos, fundamentalmente recursos humanos, que se utilizarán durante parte o todo el tiempo de desarrollo. Esto permite armar la siguiente ecuación:

COSTO TOTAL = ∑ (COSTO del RECURSO * TIEMPO DE UTILIZACION)
+ GASTOS INDIRECTOS

De este modo, sumando el **costo** de cada recurso a emplear, según el **tiempo**[48] que se haya estimado utilizarlo, y agregándole el proporcional de los gastos indirectos, se habrá estimado, razonablemente, el costo del software. Si a eso se le agrega el **margen de utilidad** esperado, se tendrá entonces calculado el **precio** al que habrá que vender el software.

Lamentablemente hay una pequeña trampa… para estimar recursos se necesitará saber que componentes tendrán que construir esos recursos. ¡Y dichos componentes no se conocen hasta no haber avanzado en el proyecto! En efecto, hasta no finalizar el relevamiento, no será posible saber la funcionalidad del software. Y hasta no diseñar la solución, no será posible conocer que tan complejos serán los módulos que habrá que programar y, mucho menos, imaginar cuantos recursos serán necesarios emplear en su construcción.

Pero claro… avanzar sobre las etapas de análisis y diseño tiene un costo. Sin cubrirlo, no pueden iniciarse el proyecto. Un círculo vicioso. ¿Se puede salir?

[47] Además de difícil de calcular, la amortización de una de una PC, el costo proporcional de Internet o la energía eléctrica insumida para producir una pieza del software, resultan claramente insignificantes, por ejemplo, comparados contra el valor hora que se les paga a los programadores.

[48] Algunos recursos son independientes del tiempo que dure el desarrollo, por ejemplo, licencias del entorno de desarrollo, componentes reutilizables, hardware específico para el proyecto. En estos casos la ecuación funciona igual, sumándolos en forma directa sin ponderación de tiempos.

5. Presupuestación de software

Primero hay que comprender que, en realidad, un desarrollo de software no comienza con por la etapa de análisis[49]. Se ha marcado anteriormente la existencia de una primera etapa de definición inicial. Con esos datos preliminares se puede hacer un **presupuesto inicial estimado**, basado en la experiencia, y en métricas de proyectos anteriores. Seguramente será provisorio, tendrá un elevado margen de error, pero servirá como idea global de la magnitud del proyecto.

La segunda idea consiste en dividir el proyecto en partes. No se buscará presupuestar todo el software al detalle sino solamente las primeras etapas. Luego, con la información de las primeras tareas, se podrán presupuestar las siguientes. Existen principalmente dos técnicas para dividir y descomponer las tareas de un proyecto:

- **Descomposición basada en el problema**: Divide el software en base a que problema resuelve (ABM, listados, pantallas, entrevistas, historias de usuario)

- **Descomposición basada en el proceso:** Se descompone el proyecto en función de los procesos (análisis, diseño, etc.)

El este último caso es posible comenzar presupuestando solamente las etapas de análisis y diseño. Terminada esta última etapa, ya con el panorama más claro de que habrá que construir, se presupuestan las siguientes instancias, refinando y ajustando el presupuesto global inicial.

Este enfoque puede parecer extraño, en especial para quien pretende saber cuánto le va a costar un sistema. Sin embargo, es ampliamente extendido en otras industrias: primero se cobra el desarrollo de un plano o diseño y, luego de aprobado, se presupuesta el costo de la obra.

Es importante destacar que el diseño tiene un valor económico por sí mismo, y aun cuando no se acepte el segundo presupuesto, no habrá perdido dinero[50]. Ese diseño servirá para que otros programadores lo realicen o para que se construya más adelante cuando se consiga el prepuesto.

En ocasiones, en proyectos muy grandes y complejos, esta división podría ser más granular, presupuestando cada etapa por separado o incluso dividiendo el proyecto global para ir entregando el software por partes, o versiones sucesivas. Las técnicas de descomposición basada en el problema también pueden ayudar para presupuestar un proceso (ej. cuantos listados deberán construirse en la etapa de codificación, a cuántos usuarios habrá que capacitar en la etapa de implementación)

En resumen, un proyecto de software podría tener, mínimo, tres instancias de presupuestación. El primero, estimativo, ocurrirá en la etapa de definición inicial, para dar una idea global de magnitud del proyecto. Un segundo presupuesto, más ajustado y basado en información recolectada en la etapa mencionada, que abarque solamente el análisis y el diseño. Finalmente, y ya con la especificación de los componentes que deberán construirse, se presupuesta resto de las tareas. De nuevo, en proyectos más grandes y complejos, posiblemente otras etapas deban ser

[49] Puede consultarse el apunte “Conceptos Fundamentales del Desarrollo de Software Dirigido por un Plan” para un mayor detalle las etapas de desarrollo de software y la comunicación inicial con el cliente.

[50] Para más detalle, ver “El Valor económico del Diseño”, en el apunte “Conceptos fundamentales del diseño de Software” de César A. Briano

presupuestadas por separado. De este modo, el riesgo de trabajar sin un presupuesto aprobado queda acotado solamente la etapa de definición inicial. Riego que por otro lado es razonables e inherentes a la profesión.

Dado que el muchas veces un proyecto de software no es un producto cerrado, sino que puede tener modificaciones y cambios, los presupuestos deben considerar también esta posibilidad. De no permitirlo se corre el riesgo de forzar a cumplir con el presupuesto, privando a la organización de la posibilidad de obtener un mejor software.

En línea con lo anterior también hay riesgos de estimaciones autocumplidas: Si la estimación de tiempos o de costos no es la correcta, se termina ajustando el proyecto de desarrollo para que la cumpla. Y esto se hace a expensas de quitar funcionalidad al software o afectar su calidad. En ocasiones, hasta incluso se llega a reducir la prueba del software al mínimo (incluso eliminarla) solo para cumplir con los plazos de entrega. Esto puede ser catastrófico.

6. Determinación del precio del software

La determinación del precio del software no solo depende de una adecuada estimación de los recursos y del valor de estos. Intervienen una serie de consideraciones adicionales que deben tener sen cuenta:

- Como ya se indicó, es imposible fijar precios finales y realistas antes de realizar la ingeniería de requerimientos. Puede darse una idea global en función de la experiencia y del valor de proyectos anteriores.

- Al igual que lo indicado para el presupuesto, puede darse un precio que solo incluya las etapas iniciales de análisis y diseño. El precio de la codificación solo será posible de determinar una vez que se conozca qué es lo que se quiere programar.

- Cuando se calcula un precio, hay que hacer consideraciones más amplias de índole organizacional, económica, política y empresarial. A veces conviene hacer un proyecto aun sin ganar dinero, apostando, por ejemplo, a que se continúen solicitando cambios y mejoras. También puede buscarse una relación estratégica con el cliente respecto de futuros desarrollo que éste tenga previstos.

- Hay que considerar los riesgos asociados con el proyecto. En escenarios muy riesgosos seguramente se cotizará un precio mayor para cubrir posibles contingencias. La misma consideración podrá tenerse si detecta alta volatilidad de los requerimientos, suponiendo que puedan implicar contramarchas y reprogramaciones.

- Debe prestarse especial atención a aquellos componentes del sistema que tengan una especial dificultad técnica, y cuya estimación inicial del tiempo de desarrollo sea riesgosa. Se pueden prever recursos adicionales que eventualmente podrían utilizarse o bien acordar con el cliente algún esquema de compensación de modo de que entre ambos cubran riesgo.

- La capacitación de usuarios debe estar considerada en el precio. Sin embargo, hay que establecer en que consiste esa capacitación, cuanto tiempo dura y a cuantos usuarios de capacitarán. Podrán incluirse capacitaciones básicas y luego prever otras adicionales y opcionales, que el usuario podrá adquirir en el caso de considerarlo necesario

- Dentro de los costos indirectos hay que considerar la garantía, ya que corresponderá hacernos cargo del costo de reparación de los errores que se encuentren, tanto en la etapa de pruebas como una vez implementado el software.

- Personalización y servicios adicionales: En algunos casos, la empresa desarrolladora ya tiene un sistema base ya desarrollado. El precio del software puede estar determinado por el precio base que se fije por ese sistema y el nivel de personalización o adaptación que se requiera para satisfacer las necesidades específicas de un cliente.

- En el precio, también se pueden ofrecer servicios adicionales, como implementación, consultoría, capacitación o soporte técnico. Por ejemplo, a partir de la implementación de un nuevo software, la organización puede requerir modificar circuitos administrativos, cambiar sus operaciones o mejorar sus prácticas. Los desarrolladores pueden cotizar estos servicios y venderlos como adicional al software-

7. Componentes ya desarrollados

En la actualidad, existen infinidad de componentes de software disponibles en bibliotecas de código de Internet. Estos recursos se pueden integrar a nuestro proyecto, ahorrando tiempos y valiosos recursos de programación. En algunos casos, habrá que pagar por su utilización, pero tambien es posible que sean gratis, de código abierto y de uso público[51]. Por lo general, estos artefactos requieren cierta configuración y personalización, pero aun así será preferible a desarrollarlos desde cero.

Además de estos recursos externos, con el tiempo, los desarrolladores van construyendo decenas de recursos propios, que pueden ir reutilizando. Esto va desde una simple función para validar el CUIT, hasta un módulo completo de seguridad y de validación de usuarios. Incluso pueden reutilizarse manuales de usuario, instaladores o datos, como, por ejemplo, las tablas de países o de localidades que ya pueden entregarse con los datos cargados.

¿Qué tratamiento debe darse a estos componentes que no suman costos al proyecto? Hay dos posibles criterios, igualmente válidos, y por supuesto decenas de opciones intermedias.

Por un lado, nada impide que el desarrollador cotice el código todo ya desarrollado, calculando el costo de las horas que le llevaría volver a realizarlo; estimar cuando valdría construir los componentes gratuitos obtenidos de internet; y sumarlos al costo del sistema. En definitiva, el

[51] Cuando se incorporen componentes es preciso leer bien los contratos de uso y licenciamiento. Existen incluso componentes que son de libre descarga para uso personal, pero que requieren adquirir licencias si es que son incorporados a software de uso comercial.

cliente recibe un componente nuevo y funcional, que incluso tiene la ventaja de ya haber sido probado con anterioridad.

Otra posibilidad es que el programador utilice estas herramientas para reducir sus costos y poder pasar una mejor cotización, logrando que sea considerado con una mejor oferta. Dependerá, claro, de que tanto necesite obtener el nuevo desarrollo.

Es obvio, pero tambien podrá elegir por cobrar aquellos componentes que considere más importante y especiales, incorporando otros sin costo.

Es importante considerar que, aunque no es habitual, en ocasiones se firman contratos de desarrollo donde todo el código le pertenece al cliente. En ese caso, este ya no puede incorporarse luego a otros sistemas. Será importante revisar si existen limitaciones en dicho sentido.

8. Particularidades del software

Entre tantas otras particularidades que tiene la construcción de software, una es que se pueden hacer copias exactas a costo casi cero. Producir una unidad, dos o cientos de ellas, prácticamente sale lo mismo. Dicho de otro modo, **en el desarrollo del software todo el costo se lo lleva la primera unidad**.

Esta particularidad cambia la ecuación del precio del software en 2 sentidos:

- Si se está produciendo un desarrollo comercial abierto, destinado a múltiples clientes (ej. un juego, Windows, una app para celular), obviamente no va a ser el primero que lo compre, el que cargue con todo el costo. Por el contrario, será proporcionado entre todas las unidades que se estimar vender. Si finalmente se venden más copias que las estimadas inicialmente, será todo ganancia. Por otro lado, en el caso de que se vendan menos que las previstas, el proyecto podría no recuperar sus costos.

- Si el desarrollo es para un cliente en particular, la ecuación es diferente. En este caso se produce una sola unidad y es lógico que este cliente pague todo el costo de desarrollo. Pero hay una consideración. Que se haga un desarrollo particular no quiere decir que el desarrollador no se quede con experiencia y con componentes que luego pueda vender en otro desarrollo. Incluso también está la posibilidad de conseguir un segundo cliente, con necesidades similares, que pueda recibir el sistema con unos pocos cambios. En este caso, el desarrollador recibiría una ganancia extraordinaria: vendería dos o más veces un sistema que ya cobró en forma completa.

  Es importante en este punto analizar cómo será el contrato que se firmará con el primer cliente. Si se permite que el desarrollador conserve la propiedad intelectual del software y la posibilidad de que revenda todo o parte del desarrollo, es posible pautar un precio menor con el primer cliente, teniendo en cuenta la chance de obtener ganancias futuras con la reventa del sistema[52].

[52] Muchas empresas prefieren firmar contratos que aseguren la propiedad total del software, aun pagando más por el sistema. Si bien esto es posible… debe tenerse en cuenta que, en definitiva, el desarrollador que lo programó una vez

9. Formas de venta o distribución del software.

A adquirir un software, las organizaciones pueden optar por dos enfoques principales:

- **Desarrollo personalizado**: En este caso, la organización contrata a un equipo de programadores o una empresa de desarrollo de software para crear un software a medida que se ajuste a sus necesidades específicas. El desarrollo personalizado implica un enfoque más completo y personalizado, donde la organización tiene el control total sobre el software y posee la propiedad intelectual del mismo. El software se desarrolla y se entrega como una solución exclusiva para la organización, y esta es responsable de su mantenimiento, actualizaciones y soporte técnico.

- **Licenciamiento de software**: En esta opción, la organización adquiere el derecho de utilizar un software ya desarrollado, bajo los términos y condiciones establecidos en un contrato de licencia. La empresa que desarrolló el software retiene la propiedad intelectual y los derechos de autor de este. La organización paga una tarifa o licencia para poder utilizar el software según las condiciones y limitaciones establecidas en el acuerdo de licencia. El proveedor del software generalmente se encarga del mantenimiento, actualizaciones y soporte técnico del producto.

Desde luego, ambos esquemas tienen ventajas y desventajas. Contratar un desarrollo propio permite disponer libremente del software, modificarlo e incluso revenderlo, pero implica tambien la necesidad de contar con un equipo de desarrollo para mantenerlo. Y claro, seria imposible utilizar este esquema para, por ejemplo, un sistema operativo. No podríamos contratar a Microsoft para que nos diseñe un Windows a medida de nuestra organización

Las empresas desarrolladoras pueden licenciar sus productos según una amplia variedad de formas de licenciamiento entre las que se destacan:

- **Licencia por usuario:** En este modelo, el software se vende bajo licencias individuales para cada usuario. El precio se establece en función del número de usuarios que utilizarán el software dentro de la organización. Generalmente, a medida que aumenta el número de usuarios, el precio unitario por usuario tiende a disminuir.

- **Licencia perpetua**: En este caso, el software se vende bajo una licencia de uso perpetuo. Los clientes pagan una tarifa única para adquirir el software y pueden utilizarlo de manera indefinida. A menudo, se pueden ofrecer actualizaciones y soporte adicional como servicios complementarios con o sin costo adicional.

- **Suscripción**: En este modelo, el software se ofrece mediante una suscripción periódica, como mensual o anual. Los clientes pagan una tarifa recurrente para acceder y utilizar el software durante el período de suscripción. El precio puede variar en función de

puede volver a hacerlo. Bastarán unos pocos cambios en la interfase como para que sea difícil demostrar que se trata del mismo software.

diferentes niveles de servicio, cantidad de usuarios o características adicionales incluidas en cada plan de suscripción.

- **Modelo freemium:** En este enfoque, el software se ofrece de forma gratuita en su versión básica, pero se cobran tarifas por funciones o características avanzadas. Los usuarios pueden acceder y utilizar la versión básica sin costo, y si desean aprovechar funcionalidades adicionales, deben pagar una tarifa.

10. Presupuesto y costos de un desarrollo ágil

Tal como se ha analizado en unidades anteriores, La planificación inicial del proyecto y la estimación son claves para poder realizar un presupuesto y costear el software. Sin embargo, ninguna de estas dos cosas existe en desarrollos ágiles. Por un lado… no hay plan que presupuestar. Por el otro… el sistema está en permanente desarrollo y por lo tanto el cliente puede pedir que se incorporen todas características que necesite.

Desde el lado del desarrollador, este pretenderá recuperar, mes a mes, el costo de los recursos asignados al proyecto, y tener además un margen de ganancia. El cliente, por otro lado, buscará que beneficio que obtenga por la utilización del software y sus nuevas funcionalidades sea mayor que la inversión que está realizando para su desarrollo[53].

Esta ecuación también se debe ser positiva si fuera un desarrollo interno: El beneficio obtenido por el software debe superar el costo de los recursos que asigna la empresa para desarrollarlo.

Un criterio comúnmente utilizado es el de facturar horas efectivamente utilizadas. Pero para esto se requiere una relación de confianza entre el cliente y el proveedor. Si, por ejemplo, facturasen más horas que las realmente insumidas, la relación cantidad de modificaciones entregadas / horas facturadas disminuye y, para el cliente, puede resultarle menos atractivo continuar con el proyecto.

Otra opción es fijar un precio por componente. Por ejemplo, se puede costear el valor del equipo de trabajo y en función de eso valorizar un sprint (costo de todos los recursos del equipo por el tiempo de duración del sprint). También pueden valorizarse las historias de usuario (con algún tipo de ponderación, conforme el esfuerzo que demanda su desarrollo) y cobrar luego en función de las historias de usuario resueltas en el período. Claro que, aun elegido esté criterio, tampoco va a ser posible saber cuántos sprints o historias tendrá finalmente el proyecto.

Otra cuestión que no puede dejar de mencionarse es como se definen las duraciones mínimas del proyecto, de modo que el cliente no deba continuar con el desarrollo más allá de lo razonable, pero tampoco abandonar el proyecto antes de del tiempo, dejando al desarrollador con recursos comprometidos que no podrá solventar. La experiencia y las métricas obtenidas en proyectos anteriores son útiles para hacer algún tipo de “estimación de mínima” de modo que se

[53] Este es concepto de Retorno de la Inversión (ROI, por sus siglas en inglés). Esta métrica permite a las organizaciones calcular si ha ganado dinero por las inversiones que realiza. Su fórmula más habitual consiste en restar a la ganancia total, los costos de esta y dividir ese valor por los costos de la inversión. **ROI = (Ganancia Total – Costos de la Inversión) / Costos de la inversión**. Se busca que el valor sea positivo y lo más alto posible.

pueda decir que las primeras o principales historias de usuario se pueden desarrollar en un tiempo determinado.

A partir de ese tiempo, el cliente puede decir si continua o no agregando funcionalidad al software.

El resumen. Un presupuesto y costeo preciso requiere un plan detallado a seguir. Si se acuerda no trabajar guiándose por un plan, debe acordarse también no trabajar apegándose a costos definidos de antemano. El desarrollador buscará satisfacer al cliente con entrega periódica de software valioso; de modo que siga siendo beneficioso para el cliente seguir solventando el proyecto.

Para muchas organizaciones que se manejan presupuestos más estrictos, o que deben conseguir fondos de aplicación específica, por ejemplo, en casa matrices, esta es una de las grandes trabas a la hora de encarar desarrollos en modelos ágiles.

Para finalizar, es importante decir que muchas organizaciones con el tiempo logran desarrollar modelos empíricos de costos, basados en su propia experiencia que les permite mejorando el costeo y fijación de precios.

11. Bibliografía

IAN SOMMERVILLE: Software Engineering. Tenth Edition. 2016. Pearson Education.
ROGER PRESSMAN: Software Engineering. Eigth Edition. 2015. Ed. McGraw-Hill.

Apunte 11

Decisiones estratégicas antes de desarrollar software.

1. Introducción

En el apunte 2 de este libro, se plantea que todo desarrollo de software comienza con la "Comunicación Inicial" entre el cliente y el desarrollador. Allí es donde se acuerdan los lineamientos básicos del proyecto. Esto es una verdad, pero a medias.

Antes que una organización decida desarrollar un sistema, aparen una serie de alternativas que deben considerarse. La primera es si realmente un software solucionará el problema que tiene la organización. Luego deberá analizar si lo desarrollará o comprará hecho, si lo instalará en sus servidores o lo usará desde la nube o, incluso, podrá pensar no usar software y tercerizar el proceso.

Conocer a fondo cada una de estas alternativas exceden el alcance de este apunte. La tercerización (u **outsourcing),** por ejemplo, es un tema vasto y complejo del que se han escrito libros enteros. Comprar software, otro ejemplo, implica conocer de políticas y contratos de licenciamiento (en nuestra Carrera, materias como Derecho Informático se ocupan de ello). La propuesta de este apunte no es abarcar discos temas a fondo, pero si alertar sobre la necesidad de tomar ciertas decisiones estratégicas que, incluso, pueden derivar en que se desista de la idea de desarrollar el software que se tenía pensado.

2. El software no siempre es la solución

No hace falta en este punto hablar de los beneficios que un sistema informático le trae a una organización. No solamente porque puede hacer determinadas cosas más rápido, con menor costo y operar con mayores volúmenes de datos, sino porque mejora el modo de trabajo de los empleados, y ayuda a formalizar y ordenar los procesos de la organización.

Pero… ¿siempre son beneficios? Usar un sistema informático que, además de facilitar las operaciones, estandariza procesos es algo bueno. Evitar, por ejemplo, que un empleado anote las salidas de un depósito en un cuaderno según su propia codificación y criterio y, en lugar de eso, pase a registrarlo de un modo estándar en un sistema, sin dudas es beneficioso para la empresa. Pero si hacer esa registración genera una demora extra en entregar cada pieza y con eso se retrasa la producción, hay que dudar de los beneficios.

Es necesario destacar los sistemas son cada vez más poderosos, pero no hacen magia ni resuelven por sí solos los problemas de las organizaciones. Un mal proceso administrativo será siempre malo, informatizado o no. A veces desarrollar software ayuda también a repensar un proceso, a eliminar trabas, a hacerlo más rápido y eficiente. Pero otras veces no.

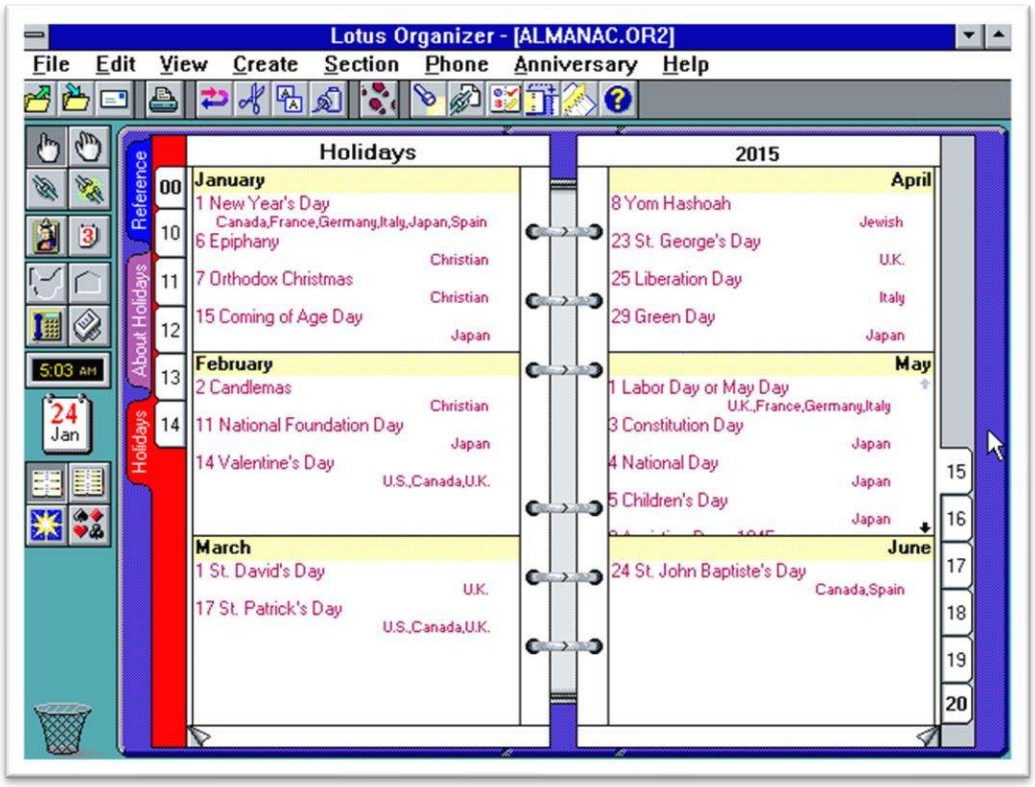
Quizá lo primero que hay que tener en cuenta es que los procesos manuales suelen ser eficientes por naturaleza: nadie quiere más cosas que le requieran más esfuerzo que las necesarias. Un sistema puede potenciarlo, puede hacerlo más rápido, puede mejorar el proceso posterior de los datos, pero para ello, la aplicación no debería copiar exactamente al proceso manual.

Para comprender lo dicho, podemos usar como ejemplo la confección de cheques. Los datos que se completan em un cheque de papel son los mínimos necesarios para poder transferir el dinero

a la persona que debe recibirlo. El escribir el importe dos veces, uno en letras y otro en números, es un modo de dificultar que quien lo recibe, pueda cambiar el monto de la operación. Sin embargo, ninguna aplicación bancaria copia ese modelo. Es innecesario pedir la fecha de emisión, no hace falta escribir el monto dos veces (el receptor no puede cambiarla) y sería ridículo pretender una firma ológrafa en un celular para validad la operación. Las APPs bancarias no copian fielmente la operatoria manual, que es eficiente para el mundo físico, sino que la mejoran y la potencian a partir de las posibilidades que da un entorno digital.

Un par más de ejemplos concretos quizá sirvan para comprender mejor el punto. El primero es un caso personal:

Muchos años atrás, la empresa Lotus, famosa en el mercado por ser la desarrolladora de una de las primaras planillas de cálculo que existieron, lanzó al mercado una agenda llamada Lotus Organizer. El software era una agenda que copiaba al detalle a las agendas en papel, desde la solapas hasta las hojas con anillas. Así se veía la agenda.



Más por una cuestión de modernidad que de practicidad, muchos comenzamos a usarla. Y digo de practicidad porque, al menos las primeras versiones, no agregaban demasiada funcionalidad a las agendas manuales. Pero además… tenía una gran contra: no era portable. Solo se instalaba en una sola PC. Es decir que cada vez que se quería anotar una cita había que ir hasta la PC. Obviamente, como no existía internet, la agenda de la PC de casa no se sincronizaba con la de la oficina. En el mejor de los casos se la podía instalar en una notebook, pero había que cargarla a todos lados para consultar la agenda. Mucho más complicado era su uso cotidiano: Cierta vez anoté la dirección de

un cliente al que debíamos visitar con mi socio[54] y al llegar al edificio no recordábamos el piso. En el hall de entrada tuvimos que abrir el portafolio, sacar la notebook, encenderla (y esperar a que encienda), acceder a la agenda y ver el piso. Todo esto nos llevó unos 4 o 5 minutos de incomodidad, sentados en la escalera. Consultar nuestra agenda en papel, que hasta entonces llevábamos en nuestros bolsillos, nos hubiera permitido resolver el problema en segundos. Que un sistema copie exactamente un sistema manual, sin agregarle ninguna funcionalidad, no suele ser buena idea.

Otro ejemplo tiene que ver con las estaciones de servicio. En un tiempo al terminar de cargar combustible, se pagaba (con efectivo o tarjeta) y se seguía viaje. Hoy los playeros tienen que registrar datos en un sistema, usando una computadora ubicada en un lugar incómodo, usando mouse que no siempre suele estar en lugar más accesible (peor aun cuando se opta por pantallas "touch" en lugares donde suele haber grasa y suciedad y los operadores usan guantes). Recién cuando terminan el registro y desperdician montón de papel en tickets que van a la basura (algunos son requisitos legales) se produce el cobro. Aun para pagar con QR, que debería ser inmediato, se necesita esperar que se registre la operación. Supongo que sistematizar esta operatoria traerá algún tipo de beneficio a las petroleras o dueños de la estación, pero lo cierto es que para el usuario solo le significan demoras. Si registrar la operación en un sistema implica demorar la carga 2 minutos (y en ocasiones demoran incluso más tiempo), cada 30 autos que carguen nafta, la estación de servicio habrá estado inactiva por 1 hora.

Antes de pensar en desarrollar un sistema hay que pensar si los problemas que hoy tiene la operatoria realmente se van a solucionar con ese sistema y, fundamentalmente, si no van a aparecer otros. A veces, mejor que un nuevo sistema, es preferible usar mejor el que se tiene (sea manual o informatizado). En otras ocasiones, bastará con cambiar el proceso administrativo. Pero siempre hay que tener en mente que copiar en un sistema algo que anda mal, posiblemente lo haga andar peor.

Por último, hay que analizar que software y aplicaciones tenemos disponibles actualmente y si con dichas aplicaciones no basta para atender las necesidades. Los paquetes de ofimática[55] son cada vez más poderosos y permiten resolver cada vez más temas de la organización. Con Excel, se pueden construir tablas y planillas que habitualmente son todo lo que un área necesita. Con Google Forms se pueden generar formularios de carga Web para pedidos remotos, sin necesidad de tener siquiera conocimientos en programación.

Los listados, las consultas, las diferentes salidas y modos de presentar la información gerencial son uno de los temas centrales a la hora de diseñar software. Y es complejo de pensarlas y diseñarlas. En primer lugar, por la alta volatilidad, frecuentemente se necesita agregar o quitar algún dato de un listado, sumar un nuevo total, presentar un gráfico, etc. Pero también porque, a medida que se va usando un sistema y se descubre su potencial, y se detecta que es posible obtener más y mejor información que la que se pensó al comienzo. Hoy día podemos pedir que la única salida que tenga un sistema es que permita exportar todos los datos a Excel y luego usar esa herramienta para analizar todos los datos y presentarlos del modo que el usuario quiera.

[54] Por aquel entonces, mi socio (y amigo) era el profesor Victor Veriansky, quien hoy me acompaña como docente en la cátedra, y con quien compartí esta y muchas otras de las experiencias con las que aparecen en este libro.

[55] Microsoft Office, Google WorkSpace o LibreOffice son algunos ejemplos de software que proveen valiosas funcionalidades para empresas de todo tipo.

3. Desarrollar versus Comprar

Una vez evaluado el contexto y decidido que implementar un nuevo software es necesario, la siguiente decisión que hay que tomar es si se va a desarrollar[56] uno desde cero o se comprará alguna solución que ya exista en el mercado. Incluso a veces ni siquiera hay que comprarla, porque hay excelente software gratuito listo para implementar.

Un software desarrollado a media es sin dudas algo bueno para la organización. El sistema hace todo lo que necesita la empresa y del modo en el cual lo hace. El software se adapta a la operatoria y no al revés. Cumple con todos sus requisitos, inclusive los no funcionales y permite que el operador lo personalice a su gusto.

Un buen traje, confeccionado a medida por un buen sastre, sin dudas nos hará lucir impecables en una fiesta importante. Pero… ¿y si el sastre no resulta tan bueno como creíamos? O quizá sea excelente y use las mejores telas importadas, pero sus costos son demasiado altos como para poder págalo o justificar su compra. Incluso la realización y hechura puede demorar mucho y ni estar listo a tiempo. Al menos conviene explorar si un traje estándar, comprado en una casa de moda, no cumple razonablemente con el cometido. A lo mejor no me calza perfecto, pero me lo puedo probar, ver cómo me queda, hacerle algún arreglo pequeño y tenerlo ya listo para la fiesta. Seguramente será más barato y puedo destinar el dinero que me sobre a una nueva corbata y zapatos.

A lo largo de todo el libro se ha podido ver que desarrollar software de calidad no es tarea sencilla, tampoco barata. Si se encuentra un software ya desarrollado que cumpla razonablemente con los requerimientos de la organización, a la larga, terminar siendo una mejor alternativa que encarar una construcción a medida. Puedo implementar el sistema en forma más rápida e incluso ahorrar dinero para poder, por ejemplo, actualizar el hardware.

Por supuesto, salvo que mi negocio opere de un modo muy simple y estandarizado, será casi imposible encontrar un software en el mercado que cumpla con todos mis requerimientos. Algunos autores indican que, si encontramos un software que cumpla con el 80% de los mismos, será mucho mejor comprar, que desarrollar a medida. Otros incluso hablan de un porcentaje menor. Sin embargo, hablar de porcentajes de cumplimiento tiene poco sentido. Ningún sistema, ni siquiera uno desarrollado a medida, cumple con el 100% de los requerimientos. Entre otras cosas, porque hay limitaciones tecnológicas, de costos y de tiempos. Pero ninguno pude dejar de cumplir aquellos que son prioritarios y fundamentales para la organización. De qué sirve que un sistema de facturación cumpla con el 95% de lo que necesita la empresa si resulta que no permite imprimir una factura en las condiciones que se necesitan.

¿Qué cosas deben considerarse entonces a la hora evaluar de comprar o desarrollar software? Roger Pressman en su libro Ingeniería de Software nos sugiere las siguientes preguntas:

[56] El desarrollo podrá ser interno o encargado a un tercero. A los efectos de la decisión desarrollo compra, es indistinto.

1. ¿La fecha de entrega del producto de software será más próxima que la del software que se desarrolle internamente?

2. ¿El costo de adquisición más el costo de personalización será menor que el costo que implica desarrollar el software internamente?

3. ¿El costo del apoyo exterior (por ejemplo, un contrato de mantenimiento) será menor que el costo del apoyo interno?

Del mismo modo, Ian Sommerville marca algunas ventajas a la hora de elegir utilizar lo que el autor denomina Sistemas COTS[57], pero que se refiere a lo que más comúnmente se lo conoce como sistemas pre-planeados o, más comúnmente, enlatados:

- Implementación rápida de un sistema fiable, que ya está funcionando en otros clientes.

- Es posible analizar el software antes de implementarlo, y ver si es adecuado para mis necesidades. Incluso se pueden comparar varias alternativas para ver cuál es el que mejor se adapta a mis necesidades.

- Se evitan riesgos y demoras en el cumplimiento del plan de desarrollo y las fechas de implementación.

- No deben dedicarse recursos al desarrollo de un nuevo sistema (aunque sí a la personalización e implementación)

- El proveedor mantiene actualizado el software y corrige las fallas.

- Es posible utilizar software desde la nube.

Otra de las cosas se las que se saca beneficio al comprar un sistema ya desarrollado es que la organización puede imponer las mejores prácticas respecto del modo en el que el sistema resuelve determinados procesos. Cuando una empresa compra algún sistema de clase mundial, como por ejemplo SAP, está adoptando los procesos estándares que dicha empresa sugiere para determinadas funcionalidades. Lo mismo ocurre cuando un banco incorpora un sistema que ya utilizan otros bancos, incorpora las mejores prácticas del sector a su operatoria. Por supuesto estos sistemas suelen permitir cambios y personalizaciones, e incluso elegir entre más de un modo de operar.

[57] COTS: Comercial-Off-The-Shelf. Refiere a aquellos sistemas comerciales que ya están listos para usar y que el usuario puede tomar por sí mismo de estante de una tienda. Hoy el software ya no se vende en tiendas, pero es el equivalente a descargar de internet un software o instalar una aplicación y usarla, sin necesidad incluso de que el proveedor intervenga.

Pero claro, también hay desventajas. Entre ellas podemos mencionar:

- La organización debe, aunque sea mínimamente, adaptarse al sistema y a la forma de operar que este impone.

- Atado con lo anterior, para el usuario, la personalización es menor. Este comenzar debe trabajar del modo en el que el proveedor del software pensó, que puede ser diferente e, incluso, menos eficiente que el modo de trabajo actual.

- Ningún sistema enlatado va a cubrir el 100% de las necesidades. ¿Qué se hace con los requerimientos no cubiertos? Algunos quizá sean requerimientos secundarios y quizá pueda prescindirse de ellos. Otros no, y quizá sea necesario pedirle al proveedor que, en el caso de ser posible, personalice nuestra versión para incluirlo. Esto pude suponer salirnos de la línea estándar de actualizaciones y que cuando salgan nuevas versiones no las pueda utilizar hasta que no sea compatible con mi personalización. Otra opción es que no sea el proveedor quien construya la personalización, sino que sea el cliente, por ejemplo, y como ya se mencionó anteriormente, utilizase Excel para armar alguna consulta o presentar gráficos si el sistema comercial no los tuviera de fondo. Nuevamente en este caso siempre hay riesgos de que actualizaciones futuras invaliden el desarrollo hecho por afuera.

- Pérdida de diferenciación y de ventajas competitivas. Como ya se analizó anteriormente, los sistemas potencian los procesos y usar un sistema diferente y mejor que el de sus competidores supone hacer valer el modo distintivo que una organización opera.

- Se depende de proyecto de negocio de la empresa desarrolladora. Se supone que quien tiene un software bueno y que pudo vender a varios clientes, va a seguir manteniendo, adaptándolo y mejorándolo. Pero este puede no ser el plan del proveedor. Muchas circunstancias lo pueden llevar a cambiar de estrategia, desde cuestiones macroeconómicas y de mercado, hasta haber encontrado otro nicho de negocio que le proporcione mayor rentabilidad. Incluso determinadas actualizaciones, como por ejemplo las impuestas tras cambios de sistemas operativos, pueden requerir una inversión muy grande que no esté dispuesto a realizar. Hay múltiples ejemplos, incluso en proveedores tan importantes como Microsoft o Google, que dejan de ofrecer sus productos, o al menos dejan de actualizarlos, y nuestro sistema queda virtualmente abandonado. En el caso de sistemas comerciales, a veces se pueden hacer acuerdos que permitan, en el caso de que el proveedor decida no continuar con el desarrollo de sus sistemas, que la empresa que lo adquirió reciba el código fuente y la posibilidad de continuar ella actualizando el software.

Para finalizar hay que recordar que también puede construirse un sistema a medida comprando ciertos componentes y desarrollando otros, o integrar uno software comprado que haga alguna funcionalidad, con otro a medida que se ocupe de otras. Determinados softwares enlatados tienen módulos abiertos que permiten que desarrolladores externos programen personalizaciones y agregados. Siempre hay que recordar la premisa de no alejarse demasiado de los estándares para no quedar fuera de las actualizaciones.

Incluso si consideramos como sistema toda la organización, es frecuente encontrar empresas que tienen algún software enlatado, por ejemplo, para la contabilidad, y después desarrollos a medida para ventas al mostrador y que a su vez se integran con carritos de compra on-line.

4. *"On Premises" versus "Cloud"*

Otra de las cosas que habrá que definir es donde estará instalado el software. Sin entrar en el análisis de los tipos de licenciamiento y sus diferentes costos, las opciones más comunes son cuatro:

- **On Premises**[58]**.** El software se instala y ejecuta en las computadoras de la organización que lo utiliza. En esta modalidad, la responsabilidad total del funcionamiento recae en el cliente, quien debe asegurarse de contar con la infraestructura necesaria en términos de servidores y computadoras para alojar el sistema y procesar la información. Además, es fundamental contratar el personal de IT necesario para garantizar la operación continua y solucionar cualquier eventualidad o fallo que surja.

- **"Cloud" o en la nuble.** En esta alternativa el software que desarrollamos no se encuentra en las computadoras del cliente, sino en una infraestructura tecnológica accesible a través de Internet. Esta infraestructura puede ser proporcionada por el proveedor mismo, ofreciendo acceso remoto a sus servidores, o por un tercero especializado en servicios de la nube. Por ejemplo, es posible alojar el sistema en servicios en la nube como Amazon Web Services o Microsoft Azure, entre otros. En esta modalidad, el cliente se despreocupa de la infraestructura tecnológica y solo necesita asegurar una buena conectividad, ya que, si se interrumpe la conexión a Internet, los sistemas serán inaccesibles. Aunque el software se encuentra en una nube compartida, su uso es exclusivo para cada cliente. Si el desarrollador tiene más de una implementación, lo habitual es utilizar copias diferentes alojadas en espacios de almacenamiento separados. En cuanto a los datos de la empresa y sus copias de seguridad, pueden residir tanto en la nube como de manera local, o en una alternativa mixta. La alternativa mas frecuente es que los datos residan junto al software en la nube y los backup puedan configurarse en almacenamiento local.

- **SaaS: Software como servicio**. Una alternativa similar a la anterior es optar por contratar un software genérico en lugar de desarrollar uno propio. Este tipo de software está diseñado para atender a múltiples usuarios simultáneamente a través de Internet. En este modelo, el cliente no adquiere el software en sí, sino la capacidad de utilizar los servicios que este provee mediante un pago o suscripción. A diferencia del modelo anterior, aunque los datos se mantienen separados, el software es único para todos los clientes. Microsoft Office 365 es un ejemplo de este tipo de servicios, donde un único software proporciona servicios a miles de usuarios simultáneamente. Debido a que el software es único, cualquier cambio realizado en él tendrá un impacto instantáneo en todos los clientes que lo utilizan. Este tipo de software se mantiene constantemente

[58] Si bien suele pronunciarse o encontrarse escrito en singular, como "on premise", el modo correcto de utilizar este término en inglés es con su forma plural: "on premises"

actualizado y los errores suelen ser corregidos de manera rápida. Los cambios significativos, especialmente aquellos que afectan la forma en que se opera el sistema, suelen ser anunciados con anticipación. Además, es posible proporcionar versiones de entrenamiento para que los usuarios se familiaricen con los cambios antes de su implementación.

- **En un servidor propio e individual, pero instalado físicamente en las instalaciones del proveedor**. Esta opción brinda al desarrollador la responsabilidad de gestionar la infraestructura y las actualizaciones del software. Al estar alojado en un servidor separado del resto, proporciona una garantía adicional en términos de privacidad de los datos. Además, esta configuración garantiza un rendimiento y una velocidad de respuesta superiores, ya que el software se encuentra en un servidor dedicado exclusivamente para atender las necesidades operativas del cliente, sin compartir recursos con otros usuarios del proveedor. Conforme las nubes públicas se vuelven más seguras, rápidas y confiables, este enfoque se está volviendo menos utilizado.

Está claro que cada una de estas opciones presenta ventajas y desventajas. Ser dueño exclusivo de un software y de su operación sin dudas es la opción más segura para que nada caiga en manos de terceros o de competidores, pero también requiere pagar un alto costo para mantener la infraestructura.

Compartir un software instalado en la nube con otros clientes hace que la empresa pierda el control total de sus datos y su operación, pero también ahorra costos y se asegura de contar siempre con un software que recibe de modo prioritario las mejoras y actualizaciones.

5. Tercerización (outsourcing)

La tercerización o outsourcing implica contratar a un proveedor externo para que se encargue de realizar ciertos procesos o tareas en lugar de desarrollarlos internamente. De este modo, no se desarrolla un software, tampoco se compra uno ya desarrollado. Ni siquiera se usa como servicio, sino que directamente toda la operatoria pasa a un tercero.

En el pasado, la tercerización era popular debido a que permitía a las empresas evitar el desarrollo interno de sistemas y la necesidad de mantener una infraestructura informática propia. Esto era especialmente beneficioso cuando las empresas no contaban con los recursos o la experiencia necesaria para llevar a cabo ciertas funciones de manera eficiente. Sin embargo, con los avances en tecnología y la proliferación de soluciones en la nube, como los modelos SaaS (Software as a Service), muchas empresas han optado por adoptar estas soluciones en lugar de externalizar completamente sus procesos. Los modelos SaaS proporcionan software y servicios en línea que pueden ser utilizados por las empresas sin tener que desarrollarlos ni mantener una infraestructura propia. Esto ha reducido la necesidad de la tercerización en algunos casos.

Aun así, la tercerización sigue siendo una opción válida, especialmente cuando el proveedor del servicio agrega valor a la operación. Por ejemplo, muchas empresas pequeñas optan por delegar la gestión de sueldos a un tercero, como un estudio contable, en lugar de contar con un sistema interno. Esto les permite despreocuparse del sistema y eliminar la necesidad de contar con personal

especializado en liquidación de sueldos. Al final del mes, simplemente reciben los recibos y documentación respaldatoria procesada con el sistema del estudio contable.

En la actualidad, es más común encontrar modelos híbridos en lugar de la tercerización completa. Por ejemplo, es habitual que un tercero se encargue de operar el sistema en su totalidad, pero parte de la operatoria sea realizada por el propio usuario. Retomando el ejemplo de la gestión de sueldos, es posible que el estudio contable se encargue de todas las tareas operativas, mientras que el propio cliente cargue las novedades mensuales, como licencias, altas, horas extras, etc.

Otro ejemplo que, si bien no es una tercerización tradicional, pero podría caer en esta operaria son las plataformas de ventas por internet. La empresa no desarrolla ni mantiene un sistema de transacciones on-line, sino que terceriza toda la operatoria en Amazon o Mercado Libre, aunque mantiene parte de la operación como es la carga de artículos, precios y stock[59].

6. Conclusión

Para las organizaciones, los beneficios de implementar un software son innegables. Un software adecuado puede transformar los procesos empresariales, mejorar la eficiencia, impulsar la productividad y proporcionar una ventaja competitiva en el mercado. Es una herramienta invaluable para optimizar y potenciar las operaciones de una organización en prácticamente todas las áreas.

Antes de embarcarse en la construcción de un software, es crucial considerar una serie de aspectos que influirán en el tipo de proyecto que la empresa debe abordar. En ciertos casos, incluso puede llegarse a la conclusión de que un nuevo desarrollo no resolverá los problemas o mejorará el rendimiento de un área en particular. En lugar de eso, puede surgir la opción de mantener el sistema actual, incluso si implica un enfoque manual.

7. Bibliografía

IAN SOMMERVILLE: “Software Engineering”. 10ma edición. 2016. Pearson Education.
ROGER PRESSMAN: Ingeniería del Software. 7ta Edición. 2008. Ed. McGraw-Hill.

[59] Los modelos de negocios de las plataformas de venta on-line como Mercado Libre o Amazon, son mucho más complejos y abarcativos como para pensar en una siempre tercerización. Sin embargo, simplificándolos y desde el alcance de este apunte, bien pueden ser como ejemplos de una moderna forma de outsourcing: La empresa no desarrolla ni opera el software, sino que terceriza toda la operación de venta on-line, desde su publicación, venta, cobranza y a veces hasta la entrega. La empresa solamente se ocupa de cargar las novedades (nuevos artículos, precios o stock) como lo haría en un sistema de sueldos.

Licencia, acuerdo de uso y distribución:

El presente libro ha sido pensado como material de lectura complementario a la bibliografía de las materias de la Facultad de Ciencias Económicas de la UBA. Puede ser utilizado y distribuido libremente con fines educativos, quedando prohibida cualquier tipo de comercialización o lucro. Al distribuirse, no debe alterarse el formato y/o contenido original.

Compilación de apuntes sobre Conceptos Fundamentales de la Ingeniería de Software- Segunda Edición © 2023 by Cesar Ariel Briano is licensed under CCBY-NC-ND 4.0

Reconocimiento-No comercial-Sin derivados 4.0 Internacional (CC BY-NC-ND 4.0)

Este es un resumen legible por humanos de (y no un sustituto) de la licencia . Descargo de responsabilidad .

Eres libre de:

Compartir : copiar y redistribuir el material en cualquier medio o formato.

El licenciante no puede revocar estas libertades siempre que siga los términos de la licencia.

Bajo los siguientes términos:

Atribución : debe otorgar el crédito correspondiente , proporcionar un enlace a la licencia e indicar si se realizaron cambios . Puede hacerlo de cualquier manera razonable, pero no de ninguna manera que sugiera que el licenciante lo respalda a usted o su uso.

No comercial: no puede utilizar el material con fines comerciales .

Sin derivados : si remezcla, transforma o construye sobre el material, no puede distribuir el material modificado.

Sin restricciones adicionales : no puede aplicar términos legales o medidas tecnológicas que restrinjan legalmente a otros de hacer cualquier cosa que permita la licencia.