Ghezzi, C., Jazayeri, M., Mandrioli, D., *Fundamentals of Software Engineering*, Prentice-Hall International, Singapore, 1991[1]

Capítulo 2.
SOFTWARE: SU NATURALEZA Y CUALIDADES (CARACTERÍSTICAS).
[**NOTA**: Quality se traduce como Cualidad o como Característica]

El objetivo de cualquier actividad en ingeniería es construir algo - un producto. El ingeniero civil construye un puente, el ingeniero aerospacial construye un avión, y el ingeniero eléctrico construye un circuito. El producto de la ingeniería de sistemas es un "sistema de software." No es tan tangible como los otros productos pero, no obstante, es un producto. Cumple una función.

En cierta forma los productos de software son similares a otros productos de ingeniería, y en otros aspectos son muy diferentes. La característica que quizás hace que el software se diferencie más de otros productos de ingeniería es que el software es *maleable.* Podemos modificar el producto en sí - a diferencia de su diseño - con bastante facilidad. Esto hace que el software sea bastante diferente a otros productos tales como autos u hornos.

A menudo, a la maleabilidad del software se la utiliza mal. Aunque definitivamente es posible modificar un puente o un avión para satisfacer nuevas necesidades - por ejemplo, hacer que el puente aguante más tráfico o que el avión transporte más carga - tal modificación no se toma a la ligera y, desde luego, no se intenta hacerla sin primero efectuar un cambio de diseño y verificar el impacto del cambio exhaustivamente. A los ingenieros de software, por otro lado, a menudo se les pide que lleven a cabo tales modificaciones sobre el software. Debido a su maleabilidad, pensamos que cambiar el software es fácil. En la práctica, no lo es.

Puede ser que podamos cambiar el código fácilmente con un editor de texto, pero cumplir con la necesidad para la cual estaba dirigido el cambio, no necesariamente puede hacerse tan fácilmente. Por cierto, en este aspecto necesitamos tratar al software como otros productos de ingeniería: un cambio en software debe ser considerado como un cambio en el diseño más que en el código, el cual es sólo un ejemplo (una instancia) del producto. Podemos, sin duda explotar esta propiedad de maleabilidad, pero es necesario que lo hagamos con disciplina.

Otra característica del software es que su creación es humano-intensiva: requiere, en su mayor parte, ingeniería en lugar de fabricación. En la mayoría de las otras disciplinas de ingeniería, el proceso de fabricación es cuidadosamente considerado, o planteado, porque determina el costo final del producto. También, el proceso debe ser controlado detenidamente, muy de cerca, para garantizar que no se introduzcan defectos. Los mismos factores se aplican a los productos de hardware de computadoras. Para el software, por otro lado, la "fabricación" es un proceso trivial de duplicación. El proceso de producción de software se ocupa del diseño e implementación, en lugar de la fabricación. Este proceso debe satisfacer ciertos criterios para asegurar la producción de software de alta calidad.

Se espera, de cualquier producto, que satisfaga alguna necesidad o requisito, y que cumpla con algunos estándares de aceptación que exponen las cualidades que debe tener. Un puente lleva a cabo la función de facilitar el viajar desde un punto a otro; una de las cualidades que se espera que tenga es que no se derrumbe cuando sople el primer viento fuerte o viaje a través de él una caravana de camiones. En las disciplinas tradicionales de ingeniería, el ingeniero tiene herramientas para describir las cualidades del producto distinguiéndolas claramente del diseño del producto. En ingeniería de software, la distinción aún no es tan clara. Las cualidades del producto de software a menudo se entremezclan en especificaciones con las cualidades del diseño.

En este capítulo, examinamos las características que guardan relación con los productos de software y con los procesos de producción de software. Estas cualidades se convertirán en nuestros objetivos en la práctica de la ingeniería de software. En el próximo capítulo, presentaremos principios de ingeniería de software que pueden aplicarse para obtener estos objetivos. La presencia de cualquier característica también deberá ser verificada y evaluada. Presentaremos este tema en la Sección 2.4, y la estudiaremos en el Capítulo 6.

2.1 CLASIFICACIÓN DE LAS CUALIDADES DEL SOFTWARE

Hay muchas cualidades de software que son deseables. Algunas de éstas se aplican tanto al producto como al proceso usado para producir el producto. El usuario quiere que el producto de software sea confiable, eficiente y fácil de usar. El productor del software quiere que sea verificable, mantenible [que se pueda mantener], portable y extensible. El director del proyecto de software quiere que el proceso de desarrollo del software sea productivo y fácil de controlar.

En esta sección, consideramos dos clasificaciones diferentes de cualidades relacionadas con el software: internas versus externas y producto versus proceso.

2.1.1 Cualidades Externas Versus Internas

Podemos dividir las características del software en cualidades externas e internas. Las externas son visibles a los usuarios del sistema; las cualidades internas son aquellas que incumben a los que desarrollan el sistema. En general, a los usuarios del software sólo les preocupan las cualidades externas, pero son las cualidades internas - las que se vinculan con gran parte de la estructura del software - las que ayudan a los que las desarrollan a obtener las cualidades externas. Por ejemplo, la cualidad interna de ser verificable es necesaria para obtener la cualidad externa de confiabilidad. En muchos casos, sin embargo, las cualidades se relacionan estrechamente y la distinción entre internas y externas no es nítida.

2.1.2 Cualidades de Producto y de Proceso

Usamos un *proceso* para producir un producto de software. También podemos atribuirle algunas cualidades al proceso, aunque las cualidades del proceso están a menudo estrechamente relacionadas con las cualidades del producto. Por ejemplo, si el proceso requiere una planificación cuidadosa de los datos de prueba del programa antes de comenzar cualquier diseño y desarrollo del sistema, la confiabilidad del producto aumentará. Algunas cualidades, tales como la eficiencia, se aplican a ambos, al producto y al proceso.

Es interesante examinar la palabra *producto* aquí. Generalmente se refiere a lo que es entregado al cliente. Aún cuando esta es una definición aceptable desde la perspectiva del cliente, no es adecuada para el que desarrolla el producto [developer], quien necesita una definición general de un producto de software que abarque no sólo el código objeto y el manual del usuario que se le entregan al cliente, sino también las necesidades o requerimientos, diseño, código fuente, datos de prueba, etc. Con tal definición, todos los artefactos que sean producidos durante el proceso constituyen parte del producto. De hecho, es posible entregar diferentes subconjuntos del mismo producto a diferentes clientes.

Por ejemplo, un fabricante de computadoras podría venderle a una compañía de control de proceso el código objeto para ser instalado en el hardware especializado para una aplicación empotrada [embedded]. Podría vender el código objeto y el manual del usuario a representantes de software. Podría hasta vender el diseño y el código fuente a los vendedores de software quienes los modifican para construir otros productos. En este caso, los que han desarrollado el sistema original ven un producto, los representantes en la misma compañía ve un conjunto de productos relacionados, y el usuario final y el vendedor de software ven aún otros productos, diferentes.

La gestión de configuración es la parte del proceso de producción del software que se ocupa de mantener y controlar la relación entre todas las piezas relacionadas de las diferentes, diversas versiones de un producto. Las herramientas de la gestión de configuración permiten el mantenimiento de familias de productos y sus componentes. Trataremos la gestión de configuración en el Capítulo 7.

2.2 CUALIDADES REPRESENTATIVAS

En esta sección, presentamos las cualidades más importantes de los productos y procesos de software. Donde sea apropiado, analizamos una cualidad con respecto a las clasificaciones que hemos tratado anteriormente [más arriba].

2.2.1 Corrección, Confiabilidad, Y Solidez

Los términos "corrección," "confiabilidad," y "robustez" a menudo se usan de manera intercambiable para describir una cualidad del software que da a entender que la aplicación lleva a cabo su función como se espera que lo haga. En otros momentos, los términos son usados con significados diferentes por diferentes personas, pero la terminología no está estandarizada. Esto es bastante desafortunado, porque estos términos se relacionan con temas importantes. Intentaremos aclarar estos temas más abajo, no sólo porque necesitamos que se use a lo largo del libro una terminología uniforme, sino también porque creemos que una aclaración de la terminología es necesaria para comprender y analizar mejor los temas subyacentes.

2.2.1.1 Corrección

Un programa es funcionalmente correcto si se comporta de acuerdo a la especificación de las funciones que debería proveer (llamadas 'especificaciones de requerimientos funcionales'). Es común usar simplemente el término "correcto" en vez de "funcionalmente correcto"; del mismo modo, en este contexto, el término "especificaciones" implica "especificaciones de requerimientos funcionales." Seguiremos esta convención cuando el contexto sea claro.

La definición de corrección asume que una especificación del sistema está disponible y que es posible determinar sin ambigüedad si el programa satisface o no las especificaciones. Con la mayoría de los sistemas de software actuales, no existe tal especificación. Si una especificación sí existe, está escrita por lo general en un estilo informal, usando lenguaje natural. Es probable que dicha especificación contenga muchas ambigüedades. A pesar de estas dificultades con las especificaciones actuales, sin embargo, la definición de corrección es útil. Claramente, la corrección es una propiedad conveniente, aconsejable para los sistemas de software.

La corrección es una propiedad matemática que establece la equivalencia entre el software y su especificación. Obviamente, podemos ser más sistemáticos y precisos al evaluar la corrección dependiendo de cuán rigurosos seamos en especificar los requerimientos funcionales. Como veremos en el capítulo 6, la corrección se puede evaluar a través de una variedad de métodos, algunos poniendo énfasis en un enfoque experimental (esto es, haciendo pruebas) otros poniendo énfasis en un enfoque analítico (esto es, verificación formal de corrección). La corrección también puede ser aumentada usando herramientas adecuadas tales como lenguajes de alto nivel, particularmente aquellos que soportan análisis estático extenso [exhaustivo]. Asimismo, puede ser mejorada usando algoritmos estándar o usando bibliotecas de módulos estándar, en lugar de inventar otros nuevos.

2.2.1.2 Confiabilidad

Informalmente, el software es confiable si el usuario puede contar con él.[1]. La literatura especializada sobre confiabilidad del software define a la confiabilidad en términos de comportamientos estadísticos - la probabilidad de que el software va a funcionar como debe en un intervalo de tiempo especificado; trataremos este enfoque en la Sección 6.7.2. A los efectos de este capítulo, la definición informal es suficiente.

La corrección es una cualidad absoluta: cualquier desviación de los requerimientos hace que el sistema sea incorrecto, sin tener en cuenta cuan menor o grave sea la consecuencia de la desviación. La noción de confiabilidad es, por otro lado, relativa: si la consecuencia de un error del software no es grave, el software incorrecto puede aún ser confiable.

Se <u>espera</u> de los productos de ingeniería que sean confiables. Los productos no confiables, en general, desaparecen rápidamente del mercado. Desafortunadamente, los productos de software no han logrado aún este estado envidiable. Los productos de software son comúnmente sacados a la venta junto con una lista de "Bugs Conocidos." Los usuarios de software dan por sentado que el Release I de un producto tiene "bugs". Este es uno de los síntomas más sorprendentes de la inmadurez del campo de la ingeniería de software como una disciplina de ingeniería.[2] [2: Dijkstra, 1989, afirma que aún el desprolijo término "bug" que a menudo usan los ingenieros de software, es un síntoma de falta de profesionalismo].

En las disciplinas clásicas de ingeniería, no se saca a la venta un producto si tiene "bugs". Uno no espera que se le haga entrega de un automóvil junto con una lista de defectos o un puente con una advertencia de no usar las barandas. Los errores de diseño son sumamente raros y merecedores de titulares en la prensa. Un puente que se derrumba puede hasta causar el procesamiento judicial de los diseñadores.

Por el contrario, los errores de diseño de software son tratados como inevitables. Lejos de sorprendernos cuando ocurren casos de errores de software, nosotros los **esperamos.** Mientras que con todos los otros productos el cliente recibe una garantía de confiabilidad, con el software obtenemos un descargo de responsabilidad, de que el fabricante del software no es responsable por daños debidos a errores del producto. La ingeniería de software puede verdaderamente ser llamada una disciplina de la ingeniería sólo cuando podamos lograr confiabilidad del software comparable a la confiabilidad de otros productos.

La Figura 2.1. [gráfico con dos círculos] ilustra la relación entre confiabilidad y corrección, bajo el supuesto que la especificación de requerimientos funcionales capte, en efecto, todas las propiedades convenientes [aconsejables] de la aplicación y que no se especifiquen en ella erróneamente propiedades no deseadas. La figura muestra que el conjunto [set]de todos los programas fiables incluye el conjunto de programas correctos, pero no viceversa. Desafortunadamente, en la práctica las cosas son diferentes. De hecho, la especificación es un modelo de lo que quiere el usuario, pero el modelo puede ser, o no, una formulación precisa de las necesidades y requerimientos reales del usuario. Todo lo que el software puede hacer es cumplir con los requerimientos especificados del modelo - no puede asegurar la corrección, la precisión del modelo.

De este modo, la Figura 2.1 representa una situación idealizada donde se asume que los requerimientos en sí mismos son correctos, vale decir, que son una representación fiel de lo que debe asegurar la implementación para satisfacer las necesidades de los posibles usuarios. Como veremos en profundidad en el Capítulo 7, hay a menudo obstáculos insalvables para alcanzar este objetivo. Lo que resulta de todo esto es que nosotros a menudo tenemos aplicaciones correctas que están planeadas para requerimientos "incorrectos," de manera que la corrección del software puede no ser suficiente para garantizarle al usuario que el software se comporte "como se espera." Esta situación se trata en la próxima subsección.

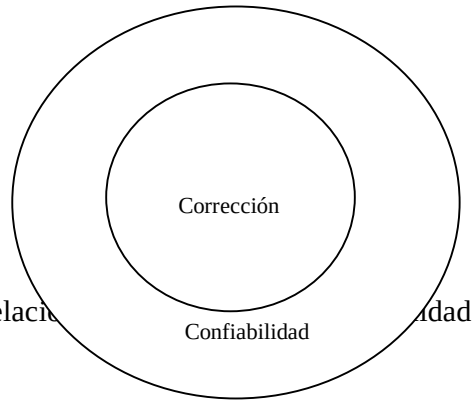

Figura 2.1 Relaci… …idad en un caso ideal.

2.2.1.3 Robustez

Un producto es robusto si se comporta “razonablemente”, aún en circunstancias que no fueron previstas en las especificaciones de requerimientos - por ejemplo, cuando se topa con datos de entrada incorrectos o alguna falla del hardware (digamos, una falla del disco). Un programa que asume que la entrada es perfecta y en tiempo de ejecución genera un error irrecuperable tan pronto como el usuario tipee, sin darse cuenta, un comando incorrecto, no sería robusto. Podría ser correcto, sin embargo, si la especificación de requerimientos no estipula cuál sería la acción al escribir un comando incorrecto. Obviamente, la robustez es una cualidad difícil de definir; después de todo, si pudiéramos establecer con precisión lo que deberíamos hacer para conseguir que una aplicación sea robusta, seríamos capaces de especificar su comportamiento “razonable” completamente. Así, la robustez se convertiría en equivalente a corrección (o confiabilidad, en el sentido de la Figura 2.1).

De nuevo, una analogía con puentes es instructiva. Dos puentes que conecten dos lados del mismo río, son “correctos” los dos si cada uno de ellos satisface los requerimientos establecidos. Sin embargo, si durante una lluvia torrencial, inesperada, sin precedentes, uno se derrumba y el otro no, podemos llamar a este último más robusto que el primero. Noten que la lección aprendida ante el derrumbe conducirá a establecer requerimientos más completos para puentes futuros, estableciendo la resistencia a lluvias torrenciales como un requisito de corrección. En otras palabras, a medida que el fenómeno sometido a estudio se hace más y más conocido, nos aproximaremos al caso ideal que se muestra en la Figura 2.1, donde las especificaciones captan exactamente los requerimientos esperados.

La cantidad de código dedicado a robustez depende del área de aplicación. Por ejemplo, un sistema escrito para ser usado por usuarios de computadoras novatos debe estar más preparado para lidiar con entradas mal formateadas que un sistema embebido que recibe su input de un sensor - aunque, si el sistema embebido está controlando el transbordador espacial o dispositivos esenciales para mantener la vida, entonces es aconsejable tener esa robustez extra.

En conclusión, podemos ver que la robustez y la corrección, están plenamente relacionadas, sin una línea divisoria definida entre ellas. Si colocamos un requerimiento en la especificación, su logro se convierte en una cuestión de corrección; si lo dejamos fuera de la especificación, puede convertirse en una cuestión de robustez. El límite [frontera] entre las dos cualidades es la especificación del sistema. Por último, la confiabilidad interviene porque no todos los comportamientos incorrectos indican problemas igualmente graves; algunos comportamientos incorrectos pueden, de hecho, ser tolerados.

La corrección, la robustez y la confiabilidad se aplican también al proceso de producción del software. Un proceso es robusto, por ejemplo, si puede tener cabida para cambios no previstos en el entorno, tales como un nuevo release del sistema operativo o el traspaso repentino de la mitad de los empleados a otro lugar. Un proceso es confiable si de manera sistemática lleva a la producción de productos de alta calidad. En muchas disciplinas de ingeniería, una parte considerable de la investigación está dedicada al descubrimiento de procesos confiables.

2.2.2 Performance

De cualquier producto de ingeniería se espera que cumpla con un cierto nivel de Performance. A diferencia de otras disciplinas, en ingeniería de software a menudo equiparamos performance con eficiencia. Seguiremos esta práctica aquí. Un sistema de software es eficiente si usa los recursos de computación de manera económica.

La performance es importante porque afecta la manera en que el sistema puede ser usado. Si un sistema de software es demasiado lento, reduce la productividad de los usuarios, posiblemente al punto de no satisfacer sus necesidades. Si un sistema de software ocupa demasiado espacio en el disco, puede ser demasiado caro para ejecutar. Si un sistema de software usa demasiada memoria, puede afectar las otras aplicaciones que son ejecutadas en el mismo sistema, o puede pasar

lentamente mientras el sistema operativo trata de equilibrar el uso de memoria de las diferentes aplicaciones.

Lo que hay debajo de todas estas afirmaciones - y también lo que torna difícil el tema de la eficiencia - son los límites cambiantes de la eficiencia a medida que cambia la tecnología. Nuestra opinión de lo que es “demasiado caro” está cambiando constantemente a medida que los avances en tecnología extienden los límites. Las computadoras de hoy cuestan órdenes de magnitud menos que las de hace unos años, sin embargo proveen órdenes de magnitud más potencia.

La performance también es importante porque afecta la forma en que se aumenta la escala de un sistema de software. Un algoritmo que es cuadrático puede funcionar con inputs pequeños pero puede no funcionar para nada para inputs más grandes. Por ejemplo, un compilador que usa un algoritmo de asignación de registro cuyo tiempo de ejecución es el cuadrado del número de las variables de programa pasará cada vez más lentamente a medida que aumente la extensión del programa que se esté compilando.

Hay varias formas de evaluar la performance de un sistema. Un método es medir la eficiencia analizando la complejidad de algoritmos. Existe una extensa teoría para calificar el caso medio o el peor de comportamiento de algoritmos, en términos de importantes requerimientos de recursos tales como tiempo y espacio, o - menos tradicionalmente- en términos de intercambio de mensajes (en el caso de sistemas distribuidos).

El análisis de la complejidad de algoritmos provee sólo información de caso medio o el peor, en lugar de información específica, acerca de una implementación particular. Para información más especifica, podemos usar técnicas de evaluación de performance. Los tres enfoques [métodos] básicos para evaluar la performance de un sistema son *medición, análisis* y *simulacro* [simulación]. Podemos medir la performance real de un sistema por medio de monitores que recopilan datos [información] mientras el sistema está trabajando [funcionando] y nos permiten descubrir embotellamientos [obstáculos] en el sistema. O podemos construir un modelo del producto y analizarlo. O, por último, podemos construir un modelo que simula el producto. Los modelos analíticos - basados a menudo en teoría de colas [queuing theory] - son generalmente más fáciles de construir pero son menos exactos mientras que los modelos de simulación son más costosos para construir pero son más exactos [precisos]. Podemos combinar las dos técnicas como sigue: al comienzo de un proyecto grande, un modelo analítico puede proveer [proporcionar] una comprensión general de las áreas críticas de performance del producto, señalando áreas donde se requiere un estudio más minucioso; luego podemos construir modelos de simulacro de estas áreas particulares.

En muchos proyectos de desarrollo de software, a la performance se la trata sólo después que se implementa versión inicial del producto. Es muy difícil - a veces hasta imposible - lograr mejoras importantes en la performance sin volver a diseñar el software. Aún un modelo simple, sin embargo, es útil para predecir la performance del sistema y guiar las elecciones de diseño de manera tal de minimizar la necesidad de volver a diseñar.

En algunos proyectos complejos, donde la factibilidad de los requerimientos de performance no es clara, se dedica mucho esfuerzo a construir modelos de performance. Dichos proyectos comienzan con un modelo de performance y lo usan inicialmente para contestar preguntas de factibilidad y más tarde para tomar decisiones de diseño. Estos modelos pueden ayudar a resolver cuestiones tales como si una función debiese ser provista por el software o por un dispositivo de hardware con un propósito especial.

Las observaciones anteriores se aplican en lo grande [“in the large”], esto es, cuando se concibe la estructura global. A menudo no se aplican en lo pequeño [“in the small”], donde los programas individuales pueden ser primero diseñados con la vista puesta en asegurar la corrección y luego ser modificados localmente para mejorar la eficiencia. Por ejemplo, los loops interiores son candidatos obvios para modificaciones de mejoramiento de eficiencia.

El concepto de performance se aplica también a un proceso, en cuyo caso lo llamamos productividad. La productividad es suficientemente importante para ser tratada como una cualidad independiente y es tratada como tal en la Sección 2.2.10.

2.2.3 Amigabilidad [User Friendliness]

Un sistema de software es amigable si sus usuarios humanos lo encuentran fácil de usar. Esta definición refleja la naturaleza subjetiva de la amigabilidad. Una aplicación que es usada por programadores novatos puede considerarse como amigable en virtud de propiedades diferentes que una aplicación que es usada por programadores expertos. Por ejemplo, un usuario novato puede apreciar mensajes ampulosos, mientras que un usuario con experiencia llega a detestarlos y pasarlos por alto. Del mismo modo, un no programador puede apreciar el uso de menúes, mientras que un programador puede sentirse más cómodo tipeando un comando.
La interfaz con el usuario es un componente importante de la amigabilidad. Un sistema de software que le brinda al usuario una interfaces con ventanas y un mouse es más amigable que uno que le exige al usuario usar un conjunto de comandos de una letra. Por otro lado, un usuario con experiencia podría preferir un conjunto de comandos que minimicen el número de pulsaciones en vez de una extravagante interfaz con ventanas a través del cual tenga que navegar para llegar al comando que todo el tiempo sabía que tenía que ejecutar. Trataremos cuestiones de interacción con usuario en el Capítulo 9.

Hay cosas más importantes para la amigabilidad, sin embargo, que la interacción con el usuario. Por ejemplo, un sistema de software embebido [embedded] no tiene una interacción con un usuario humano. En lugar de eso, interactúa con el hardware y quizás otros sistemas de software. En este caso la amigabilidad se refleja en la facilidad con la cual el sistema puede ser configurado y adaptado al ambiente del hardware.

En general, la amigabilidad de un sistema depende de la consistencia de las interacciones de su usuario y operador. Claramente, sin embargo, las otras cualidades mencionadas más arriba -tales como corrección y performance - también afectan a la amigabilidad. Un sistema de software que produce respuestas incorrectas no es amigable, a pesar de lo extravagante que pueda ser su interfaz con el usuario. También, un sistema de software que produzca respuestas más lentamente de lo que el usuario exige no es amigable aún si las respuestas son visualizadas en color.

La amigabilidad también es tratada bajo el tema “factores humanos.” Los factores humanos o la ingeniería humana [ergonomía] juega un rol importante en muchas disciplinas de la ingeniería. Por ejemplo, los fabricantes de automóviles dedican un esfuerzo importante al decidir la posición de las diversas perillas de control en el tablero de mandos. Los fabricantes de televisores y los de hornos de microondas también tratan de que sus productos sean fáciles de usar. Las decisiones de interacción con el usuario en estos campos clásicos de la ingeniería son hechas, no al azar por ingenieros, sino sólo después de estudios exhaustivos de las necesidades y actitudes del usuario, por especialistas en campos tales como diseño industrial o psicología.

Curiosamente, la facilidad de uso en muchas de estas disciplinas de la ingeniería se logra a través de la estandarización de la interacción humana. Una vez que el usuario sabe cómo usar un televisor, él o ella pueden operar casi cualquier otro modelo de televisor.[1] (1:Aunque los nuevos controles remotos son bastante complicados!). La importante actividad actual de investigación y desarrollo en el área de interacción estándar con el usuario para sistemas de software llevará al desarrollo de sistemas más amigables en el futuro.

Ejercicio

Analicen [hablen de] la relación entre los aspectos de human-interface del software y la confiabilidad.

2.2.4. Verificabilidad

Un sistema de software es verificable si sus propiedades pueden ser verificables fácilmente. Por ejemplo, la corrección o la perfomance de un sistema de software, son propiedades que son interesantes de ser verificadas. Como veremos en el capítulo 6, la verificación se puede llevar a cabo mediante métodos de análisis formales o a través de pruebas. Una técnica común para probar la verificabilidad es el uso de "monitores de software" esto es, un código insertado en el software para monitorear distintas cualidades tales como la perfomance o la corrección.

Diseño modular, práctica de codificación disciplinado y el uso de un apropiado lenguaje de programación, todo esto contribuye a la verificabilidad

La verificabilidad es usualmente una cualidad interna, aunque algunas veces se vuelve una cualidad externa. Por ejemplo, en muchas aplicaciones de seguridad crítica, el cliente requiere la verificabilidad de ciertas propiedades. El alto nivel del estándar de seguridad para un "sistema de computadora creíble", requiere de la verificabilidad del kernel del sistema operativo.

2.2.5. Maintainability

El término "mantenimiento de software" es comúnmente utilizado para referirse a las modificaciones que se hacen a un sistema de software después de su versión inicial. El mantenimiento ha sido usado simplemente como un "bug fixing" y fue afligente descubrir cuanto esfuerzo se ha gastado en corregir los defectos. Los estudios han mostrado, sin embargo, que la mayoría del tiempo gastado en mantenimiento es en el hecho de mejorar el producto con características que no estaban en las condiciones iniciales o que estaban establecidas incorrectamente allí.

"Mantenimiento" es, en sí mismo, no es una palabra apropiada para usar con el software. Primero, como se usa hoy en día, el término cubre un amplio rango de actividades, todas ellas se realizan mediante modificaciones de una parte existente en el software para hacer esa mejora. Un término que quizás mejor capture la esencia de ese proceso es "evolución del software". Segundo, para productos de ingeniería, tales como hardware de computadoras o automóviles o máquinas de lavar, "mantenimiento" se refiere a mantener el producto en respuesta al gradual deterioro de sus partes debido al uso extendido del producto, por ejemplo las transmisiones son aceitados y los filtros de aire se ensucian y periódicamente se cambian. El usar el término "mantenimiento" con software da una connotación errónea debido a que el software no se desgasta. Desafortunadamente, como siempre, el término es usado tan ampliamente que se continuará usando.

Hay evidencias que el costo del mantenimiento excede el 60% del total del costo del software. Para analizar los factores que afectan dichos costos, es necesario dividir el mantenimiento del software en tres categorías: mantenimiento correctivo, adaptativo y perfectivo.

El mantenimiento correctivo tiene que hacer en todo aquello que se hace para remover los errores residuales presentes en el producto cuando se liberó, como también los errores introducidos dentro del software durante su mantenimiento. Los costos de mantenimiento correctivo están alrededor del 20% de los costos de mantenimiento.

Los mantenimientos adaptativo y perfectivo son las fuentes reales de cambio en el software; ellos motivan la introducción de la evolucionabilidad [evolvalibility] (definida más abajo) como una calidad fundamental del software y la anticipación del cambio (definido en el capitulo 3) como un principio general que deberá guiar al ingeniero de software. Los costos de mantenimiento adaptativo son cercanas a otro 20% de los costos de mantenimiento mientras que más del 50% es absorbido por el mantenimiento perfectivo.

El mantenimiento adaptativo involucra ajustes en la aplicación de cambios en el medio ambiente, por ejemplo, una nueva versión de hardware o de sistema operativo o un nuevo sistema de base de datos. En otras palabras, en el mantenimiento adaptativo la necesidad para los cambios de software no pueden ser atribuidas a característica del software mismo, como la presencia de errores residuales o la incapacidad para proveer cierta funcionalidad requerida por el usuario. Más aún, el software debe cambiar debido a que el medio ambiente en el cual esta embebido, cambia.

Finalmente, el mantenimiento perfectivo involucra cambios en el software para mejorar algunas de estas cualidades. Aquí, los cambios son una necesidad de modificar las funciones ofertadas por la aplicación, sumada a nuevas funciones, mejorar la perfomance de la aplicación, hacer más fácil su uso, etc. Los requerimientos para lograr el mantenimiento perfectivo pueden provenir directamente de un ingeniero de software, para mejorar el status del producto en el mercado, o ellos pueden provenir de los usuarios, para desean encontrar nuevos requerimientos.

Nosotros podemos ver la mantenibilidad como dos cualidades separadas: la reparabilidad y la evolucionabilidad. El software es reparable si permite corregir los errores; y es evolucionable si permite cambios que lo capaciten para satisfacer nuevos requerimientos.

La distinción entre reparabilidad y evolucionabilidad no es siempre clara. Por ejemplo, si las especificaciones de los requerimientos son vagas, puede no ser claro si estamos corrigiendo un defecto o satisfaciendo un nuevo requerimiento. Discutiremos este punto mas adelante en el capítulo 7. En general, sin embargo, la distinción entre las dos cualidades es útil

2.2.5.1. Reparabilidad

Un sistema de software es reparable si permite la corrección de sus defectos, con una cantidad limitada de trabajo. En muchos productos ingenieriles, la reparabilidad es la meta mayor del diseño. Por ejemplo, los motores de automóviles se construyen con las partes que en su mayoría parecen fallar como la más accesible. En la ingeniería de hardware de computadoras, hay una subespecialidad llamada repairability, disponibilidad y serviceability (RAS).

En otros campos ingenieriles, cuando el costo decrece y el producto asume el status de un "commodity", la necesidad para la reparabilidad, decrece. Esto significa que es mas barato para remplazar la cosa completa, o por lo menos en su mayor parte, que repararla. Por ejemplo, en los aparatos de televisión antiguos, se podía reemplazar una sola válvula. Hoy se cambia toda la tarjeta.

De hecho, una técnica común para lograr la reparabilidad de tales productos es usar partes estándares que pueden ser reparadas fácilmente. Pero las partes del software no se deterioran. Luego, en tanto que el uso de partes estándar puede reducir el costo de *producción* del software, el concepto de partes reemplazables no parece aplicable a la reparabilidad del software. El Software es también diferente en este sentido, pues el costo del software está determinado, no por las partes tangibles. Sino por la actividad de diseño humano.

La reparabilidad es también afectada por el número de partes del producto. Por ejemplo, es más difícil reparar un defecto en el cuerpo de un auto monolítico que si el cuerpo está hecho de distintas partes de forma regular. En el último caso, se puede reemplazar una única parte mas fácilmente que el cuerpo completo. Por supuesto, si el cuerpo consiste en un montón de piezas, podría requerir muchas conexiones entre esas partes, y esto conducirá probablemente a que las conexiones por si mismas podrían necesitar reparación.

Una situación análoga se aplica al software; un producto de software que consiste en módulos bien diseñados, es mucho más fácil de analizar y reparar que uno monolítico. Solamente el incremento del número de módulos, sin embargo, no hace un producto más reparable. Tenemos que elegir la estructura modular correcta, haciendo las interfaces de módulos correctas para reducir la necesidad de interconexiones entre módulos. La modularización correcta, promueve la reparabilidad permitiendo que los errores sean confinados en unos pocos módulos, haciendo más fácil la localización y remoción de la misma. En el capítulo 4, examinaremos distintas técnicas de modularización, incluyendo información de tipos de datos ocultos y abstractos, en detalle.

La reparabilidad puede mejorarse mediante el uso de herramientas apropiadas. Por ejemplo, usando lenguajes de alto nivel más que un lenguaje assembler, conduce a una mejor reparabilidad. También, las herramientas como debuggers pueden ayudar a aislar y reparar errores.

La reparabilidad del producto afecta su confiabilidad. Por otro lado, la necesidad de reparabilidad decrece cuando la confiabilidad aumenta.

2.2.5.2. Evolucionabilidad

Como otros productos de ingeniería, los productos de software son modificados en el tiempo para proveer nuevas funciones o para cambiar funciones existentes. Más aún, el hecho que el software sea así de maleable hace que las modificaciones sean extremadamente fáciles de aplicar para la implementación. Esto es, sin embargo, la mayor diferencia entre las modificaciones de software y las modificaciones de otros tipos de productos de ingeniería. En el caso de otros tipos de productos de ingeniería, las modificaciones empiezan a nivel del diseño y luego se procede a su implementación del producto. Por ejemplo, si se decide agregar un segundo estar a la casa, lo primero que se debe hacer es el estudio de factibilidad para verificar si puede ser hecho de forma segura. Luego se requiere el diseño, basado en el diseño original de la casa. Luego este diseño debe ser aprobado, y después verificarse que no viole ninguno de los reglamentos existentes. Y, finalmente, la construcción de la nueva parte debe ser comisionada.

En el caso del software, desafortunadamente, la gente se ahorra proceder en una forma tan organizada. Aunque los cambios deberían ser cambios radicales en la aplicación, muy a menudo la implementación comienza sin hacer ningún estudio de factibilidad, dejando sólo un cambio en el diseño original. Aún peor, después que el cambio se ha logrado la documentación no es siempre actualizada a posteriori. Por ejemplo, las especificaciones no son puestas al día para reflejar los cambios. Esto hace que futuros cambios sean más difíciles para aplicar.

Por otro lado, los productos de software exitosos, son de bastante larga vida. Sus primeras versiones se usan por largo tiempo y cada sucesiva versión es el paso siguiente en la evolución de un sistema. Si el software es diseñado con cuidado, y si cada modificación se lleva a cabo cuidadosamente, entonces se puede evolucionar elegantemente.

Cuando el costo de producción del software y complejidad de la aplicación crece, la evolucionabilidad del software asume más y más importancia. Una razón para ello es la necesidad de leverage de la inversión hecha en el software cuando la tecnología del hardware avanza. Algunos de los sistemas anteriores desarrollados en los ’60, toman hoy ventajas del nuevo hardware, dispositivos y tecnología de redes. Por ejemplo, el sistema de reservas SABRE de American Airlines, inicialmente se desarrolló a mediados de la década de los ’60 y aún continúa evolucionando con nueva funcionalidad. Esto es una cosa sorprendente, considerando el incremento de la perfomance demandado por el sistema.

La mayoría de los sistemas de software comienzan siendo evolucionables, pero luego de años de evolución ellos alcanzan un estado en el que cualquier evolución importante corre el riesgo de romper las características existentes. De hecho la evolucionabilidad se logra por medio de la modularización y los sucesivos cambios tienden a reducir la modularidad del sistema original. Esto es aún peor si las modificaciones son aplicadas sin un estudio cuidadoso del diseño original y sin la descripción precisa de los cambios en ambos el diseño y en los requerimientos de la especificación.

Los estudios en sistemas de software grandes muestran que la evolucionabilidad decrece con cada versión de un producto de software.

Cada versión complica la estructura del software, de modo que las modificaciones futuras se tornan más difíciles. Para superar ese problema, el diseño original del producto, así como los cambios sucesivos, deben ser hechos con la evolucionabilidad en mente. La evolucionabilidad es una de las más importantes cualidades del software y los principios que presentamos en el próximo capítulo ayudarán a mostrarlo. En el capítulo 4, presentamos un concepto especial, tal como familias de programas que buscan exactamente para el propósito de fomentar la evolucionabilidad.

La evolucionabilidad es un producto y un proceso relacionado con la calidad. En términos de este último, el proceso debe ser capaz de acomodarse a las nuevas técnicas de gerenciamientos y organizacionales, cambios en la educación ingenieril, etc.

2.2.6. Reusabilidad

La reusabilidad está emparentado con la evolucionabilidad. En la evolución del producto, modificamos un producto para construir una nueva versión del mismo producto: en el reuso de un

producto, nosotros lo usamos al producto, tal vez, con menores cambios, para producir otro producto. La reusabilidad aparece como más aplicable a los componentes de software que a un producto completo pero ciertamente parece posible construir productos que son reusables.

Un buen ejemplo de producto reusable es el UNIX shell. El UNIX shell es un interprete del lenguaje de comandos; esto es, él acepta comandos del usuario y los ejecuta. Pero está diseñado para ser usado tanto interactivamente como en procesos por lotes. La habilidad para comenzar un nuevo shell con un archivo que contiene una lista de los comandos del entorno nos permite escribir programas - scripts – en el lenguaje de los comandos del shell. Podemos ver al programa como un nuevo producto que usa el shell como un componente. Mediante interfaces estándares favorables, el entorno UNIX, de hecho, soporta el reuso de algunos de estos comandos, así como el shell, en la construcción de utilidades poderosas.

Las bibliotecas científicas son los más conocidos componentes reusables. Distintas grandes bibliotecas de FORTRAN han existido por muchos años. Los usuarios pueden comprarlas y usarlas para construir sus propios productos, sin tener que reinventar o recodificar los bien conocidos algoritmos. Mas profundamente, distintas compañías son devotas de producir dichas bibliotecas.

Otro ejemplo exitoso de paquetes reusables es el desarrollo reciente de los sistemas con ventanas, tales como los X Windows o Motif, para los desarrolladores de interfaces de usuarios. Lo discutiremos en el capítulo 9.

Desafortunadamente, mientras que la reusabilidad es claramente una importante herramienta para reducir los costos de producción de software, los ejemplos de reuso de software en la práctica son mas bien raros.

La reusabilidad es difícil de lograr *a posteriori*, sin embargo, uno debería esforzarse por la reusabilidad cuando los componentes del software son desarrollados. En los próximos dos capítulos, vamos a examinar algunos principios y técnicas para lograr la reusabilidad. Una de las técnicas más promisorias es el uso de diseño orientado a objetos que puede unificar las cualidades de evolucionabilidad y reusabilidad.

Hasta aquí, hemos discutido la reusabilidad en el framework o la reusabilidad de los componentes, peor el concepto de aplicación más amplia: puede ocurrir a diferentes niveles y puede afectar tanto al producto como a los procesos. Un simple y ampliamente utilizado tipo de reusabilidad consiste en el reuso de la gente, es decir, reusar un conocimiento específico en un dominio de aplicación de un desarrollo o entorno destino. Y así sucesivamente, este nivel de uso es insatisfactorio, parcialmente debido a la rotación de los ingenieros de software. El conocimiento se va con las personas y nunca queda como una cuestión permanente.

Este nivel de reuso puede ocurrir para determinados niveles de requerimientos. Cuando una aplicación nueva es concebida, podemos probar identificar las partes que son similares a las partes usadas de la aplicación previa. Así podemos reusar las partes de especificaciones de los requerimientos previos en lugar de desarrollar completamente una nueva.

Como lo discutimos anteriormente, más niveles de reuso puede ocurrir cuando una aplicación es diseñada o aún el nivel de código. En el último caso, debería ser provisto con los componentes del software que son reusados desde aplicaciones previas. Algunos expertos en software plantean que en el futuro nuevas aplicaciones serán producidas por assembling junto a un conjunto de componentes de ya producidas para usar [ready-made, off the shelf components]. Las compañías de software invertirán en el desarrollo de sus propios catálogos de componentes reusables para que el conocimiento adquirido desarrollando aplicaciones no desaparezca cuando las personas se vayan, pero se acumularán progresivamente en los catálogos. Otras compañías invertirán sus esfuerzos en la producción de componentes generales reusables para ubicar en plaza para el uso de otros productores de software.

La reusabilidad aplicada en el proceso de software es buena. En efecto, las metodologías de software pueden ser vistas como una atención de reuso de iguales procesos para construir productos diferentes. Los distintos modelos de ciclo de vida hacen también el intento del reuso de los procesos de alto nivel

La reusabilidad es el factor clave que caracteriza la madurez del campo industrial. Vemos un alto grado de reusabilidad en áreas como la industria automotriz y los consumos electrónicos. Por ejemplo, en la industria automotriz, el motor es a menudo reusado de modelo en modelo. Además, el auto es construido por ensamble conjunto de componente que están altamente estandarizados y son usados a través de muchos modelos producidos por la misma industria. Finalmente, el proceso de manufactura es a menudo reusado. El bajo grado de reusabilidad en el software es un claro indicador que el campo debe evolucionar para conseguir el status de una bien establecida disciplina ingenieril.

2.2.7 Portabilidad.

El software es portable si puede correr en diferentes ambientes. El término "ambiente" puede referirse a una plataforma de hardware o un ambiente de software así como un sistema operativo particular. Con la proliferación de diferentes procesadores y sistemas operativos, la portabilidad ha llegado a ser un importante beneficio para los ingenieros de software.

Aún con un procesador familiar, la portabilidad puede ser importante por las variaciones en las capacidades de memoria e instrucciones adicionales. Una forma de conseguir portabilidad dentro de una arquitectura de una máquina es tener un sistema de software que asuma una configuración mínima tan lejos como la capacidad de memoria que es involucrada y use un subconjunto de las facilidades de la máquina que son garantizadas para que estén disponibles sobre todos los modelos de la arquitectura (así como instrucciones de máquina y facilidades del sistema operativo). Pero este penaliza a los grandes modelos porque, presumiblemente, el sistema puede ejecutar mejor sobre estos modelos si no asume tales restricciones. Por consiguiente, necesitamos usar técnicas que permitan que el software determine las capacidades del hardware y que se adapte a ellas. Un buen ejemplo de esta aproximación es la forma en que UNIX le permite a los programas interactuar con diferentes terminales sin explicitar lo que asume en los programas sobre las terminales que ellos están usando. El sistema X Windows extiende estas capacidades para permitir a las aplicaciones correr sobre alguna extensión de bit-mapeado.

Más generalmente, la portabilidad se refiere a la habilidad de correr un sistema sobre diferentes plataformas de hardware. En tanto crezca la relación de dinero gastado sobre las ganancias de software versus hardware, la portabilidad gana más importancia.

Algunos sistemas de software son inherentemente específicos de la máquina. Por ejemplo, un sistema operativo es escrito para controlar una computadora específica, y un compilador produce código para una máquina específica. Aún en estos caso, sin embargo, es posible conseguir algún nivel de portabilidad. Nuevamente, UNIX es un ejemplo de un sistema operativo que ha sido porteado a diferentes sistemas de hardware. Por supuesto, el esfuerzo para portearlo requiere meses de trabajo. Aún podemos llamar al software portable porque escribiendo el sistema desde el principio para el nuevo ambiente requeriría mucho más esfuerzo que portearlo.

Para muchas aplicaciones, es importante ser portable a través de los sistemas operativos. O, mirándolo de otra forma, el sistema operativo provee la portabilidad a través de las plataformas de hardware.

2.2.8 Comprensibilidad.

Algunos sistemas de software son más fáciles de entender que otros. Por supuesto, muchas tareas son inherentemente más complejas que otras. Por ejemplo, un sistema que hace el pronóstico meteorológico, no importa qué bien está escribiendo, será más difícil de entender que uno que imprima listas de correo. Dada una tarea de inherentemente de similar dificultad, podemos seguir ciertas pautas para producir más diseños comprensibles y escribir más programas comprensibles.

La comprensibilidad es una cualidad interna del producto, y ayuda a conseguir muchas otras cualidades tales como la evolucionabilidad y la verificabilidad. Desde un punto de vista externo, el

usuario considera al sistema comprensible si tiene un comportamiento predecible. La comprensibilidad es un componente de la amigabilidad con el usuario.

2.2.9 Interoperabilidad.

La interoperabilidad se refiere a la habilidad de un sistema de coexistir y cooperar con otros sistemas - por ejemplo, la habilidad de un procesador de palabras para incorporar un gráfico producido por un graficador, o la habilidad del graficador para graficar los datos producidos por una planilla de cálculo, o la habilidad de la planilla de cálculo para procesar una imagen escaneada por un scanner.

Mientras es raro en los productos de software, la interoperabilidad abunda en otros productos ingenieriles. Por ejemplo, los sistemas estereofónicos de varios fabricantes trabajan juntos y pueden ser conectados a televisores o grabadores. En efecto, los sistemas estereofónicos producidos décadas atrás acomodan nuevas tecnologías tales como los discos compactos, mientras virtualmente todos los sistemas operativos han sido modificados - a veces significantemente - antes que puedan trabajar con los nuevos discos ópticos.

Una vez más, el ambiente UNIX, con sus interfaces estándar, ofrecen un limitado ejemplo de interoperabilidad con un solo ambiente: UNIX fomenta a las aplicaciones tener una simple y estándar interfaz, que permitan que la salida de una aplicación sea usada como entrada en otra.

El UNIX también ilustra las limitaciones de interoperabilidad en los sistemas actuales: la interfaz estándar UNIX es una primitiva, orientada a caracteres. No es fácil para una aplicación usar datos estructurados - es decir, una planilla de cálculo o una imagen - producidos por otra aplicación. También, el sistema UNIX en sí mismo no puede operar en conjunción con otros sistemas operativos.

Otro ejemplo de las limitaciones de la interoperabilidad en el software actual es ilustrado por la mayoría del software de las computadoras personales. [1] Muchos vendedores producen productos "integrados", significando productos que incluyen varias funciones diferentes. Con mejor interoperabilidad, el vendedor podría producir diferentes productos y permitir que el usuario los combine si los necesita. Esto sería más fácil para el vendedor producir los productos, y le daría al usuario más libertad en, exactamente qué funciones pagar y combinar. En efecto, en muchas instancias, el vendedor del "producto integrado" también tiene varios productos independientes, cada uno soporta una sola función, pero estos productos no trabajan juntos. La interoperabilidad puede ser conseguida a través de la estandarización de las interfaces.

Un concepto relacionado a la interoperabilidad es este de un *sistema abierto*. Un sistema abierto es una colección extensible de aplicaciones escritas independientemente que cooperan con funciones como un sistema integrado. Un sistema abierto permite la suma de nueva funcionalidad por organizaciones independientes, después que el sistema es liberado. Esto puede ser conseguido, por ejemplo, librando los sistemas junto con una especificación de sus interfaces "abiertas". Algunos desarrolladores de aplicaciones pueden entonces tomar ventajas de esas interfaces. Algunas de las interfaces pueden ser usadas para comunicaciones entre diferentes aplicaciones o sistemas. Los sistemas abiertos permiten diferentes aplicaciones, escritas por diferentes organizaciones, para interoperar.

Un requerimiento interesante de los sistemas abiertos es que nuevas funcionalidades pueden ser sumadas sin bajar el sistema. Un sistema abierto es análogo para un crecimiento de la organización que evoluciona con el tiempo, adaptándose a los cambios en el ambiente. La importancia de la interoperabilidad ha disparado un crecimiento interesante en sistemas abiertos, produciendo muchos esfuerzos recientes de estandarización en esta área.

2.2.10 Productividad.

[1] Hay excepciones en el ambiente Macintosh, donde las aplicaciones son realizadas cada vez más altos niveles de interoperabilidad.

La productividad es una cualidad del proceso de producción del software; esta mide la eficiencia del proceso y, como dijimos antes, es la cualidad de la performance aplicada al proceso. Un proceso eficiente da por resultado una entrega más rápida del producto.

Los ingenieros individuales producen software en una cierta tasa, aunque hay grandes variaciones entre individuos de diferentes habilidades. Cuando los individuos son parte de un grupo, la productividad del grupo está en función de las productividades individuales. Muy a menudo, la productividad combinada es mucho menor que la suma de las partes. Las organizaciones y técnicas de los procesos son las que atienden la capitalización sobre la productividad individual de los miembros del grupo.

La productividad ofrece muchas negociaciones en la elección de un proceso. Por ejemplo, un proceso que requiere la especialización de miembros individuales de un grupo puede orientar la productividad en la producción de cierto tipo de producto, pero no en la producción de una variedad de productos. El software reusable es una técnica que preferencia a la productividad global de una organización que es involucrada en el desarrollo de muchos productos, pero el desarrollo de módulos reusables es más difícil que le desarrollo de módulos para un uso propio, lleva a la reducción de la productividad del grupo que está desarrollando los módulos reusables como parte del desarrollo de un producto.

Mientras que la productividad del software es de gran interés debido al incremento del costo del software, es una dificultad para medir. Claramente, necesitamos una métrica para medir la productividad - o cualquier otra cualidad – si tenemos cualquier objetivo de comparar diferentes procesos en términos de productividad. Métricas tempranas como el número de líneas de códigos productividad tienen muchas deficiencias. En el Capítulo 8, discutiremos métricas para medir productividad y organización de equipos para medir la productividad. Como en otras disciplinas ingenieriles, veremos que la eficacia del proceso es afectada fuertemente por la automatización. Las herramientas de ingeniería del software moderno y los ambientes predominantes llevan a aumentos en productividad. Estas herramientas serán discutidas en el capítulo 9.

2.2.11. Oportunidad.

Oportunidad es una cualidad relacionada con el proceso que se refiere a la habilidad para entregar un producto a tiempo. Históricamente, la oportunidad ha estado faltando en los procesos de producción de software, llevando a la "crisis del software", que a su vez llevó a la necesidad - y nacimiento de - la ingeniería de software en sí misma. Aún ahora, muchos procesos actuales no producen un producto oportuno.

El siguiente ejemplo (real) es típico de la práctica industrial actual (*circa 1988*). El primer release de un compilador Ada estaba prometido por un fabricante de computadoras para una cierta fecha. Cuando la fecha llegó, los clientes que habían ordenado el producto recibieron, a cambio del producto, una carta declarando que el producto aún contenía muchos defectos, el fabricante ha decidido que sería mejor postergar la entrega en lugar de entregar un producto que contiene defectos. El producto fue prometido para tres meses después.

En cuatro meses, el producto llegó, acompañado con una carta declarando que muchos, pero no todos, los defectos habían sido corregidos. Pero esta vez, el fabricante había decidido que era mejor permitir al cliente recibir el compilador Ada, aún pensando que contenía varios defectos serios, así que el cliente podría comenzar a desarrollar su propio producto usando Ada. El valor de la entrega temprana en este tiempo fue más pesado que el costo de entregar un producto defectuoso, en la opinión de un fabricante. Así, al final, lo que fue entregado llegó tarde y defectuoso.

La oportunidad por sí misma no es una calidad útil, aunque siendo tarde puede a veces incluir oportunidades de mercado. Llevar a tiempo un producto que le está faltando otras cualidades, tales como confiabilidad o performance, es en vano.

La oportunidad requiere cuidado fijando, la estimación de trabajo exacto, y claridad en los hitos específicos y comprobables. Todas las otras disciplinas ingenieriles técnicas de dirección de

proyectos estándar para conseguir oportunidad. Todavía hay muchas herramientas de dirección de proyecto soportadas por computadoras.

Las técnicas de dirección de proyectos estándar son difíciles de aplicar en ingeniería de software por la dificultad en medir la suma de trabajo requerido para producir una pieza de software dada, la dificultad en medir la productividad de los ingenieros - o aún teniendo una métrica fidedigna para la productividad - y el uso de hitos imprecisos e incomprobables.

Otra razón en la dificultad de conseguir oportunidad en el proceso de software es el continuo cambio de los requerimientos del usuario. La figura 2.2 grafica los requerimientos del usuario contra las capacidades del sistema actual y puede mostrar por qué fallan la mayoría de los desarrollos de software actual. (Las unidades de escala no son mostradas y pueden ser asumidas como no uniformes). Al tiempo t_0 la necesidad para un sistema de software es reconocida y el desarrollo comienza con un conocimiento más bien incompleto de los requerimientos. Como resultado, el producto inicial entregado en t_1 no satisface los requerimientos iniciales del tiempo t_0, ni los requerimientos del usuario del tiempo t_1. Entre el tiempo t_1 y el tiempo t_3, el producto es "mantenido", en función de obtener uno más cercano a las necesidades del usuario. Eventualmente, se satisfacen los requerimientos originales del usuario en el tiempo t_2. Por las razones que hemos visto en la Sección 2.2.5.5, en el tiempo t_3 el costo de mantenimiento es tan alto que el desarrollador del software decide hacer un rediseño mayor. El nuevo release llega a estar disponible en el tiempo t_4, pero el hueco [gap] con respecto a las necesidades del usuario en este punto es aún mayor que antes.

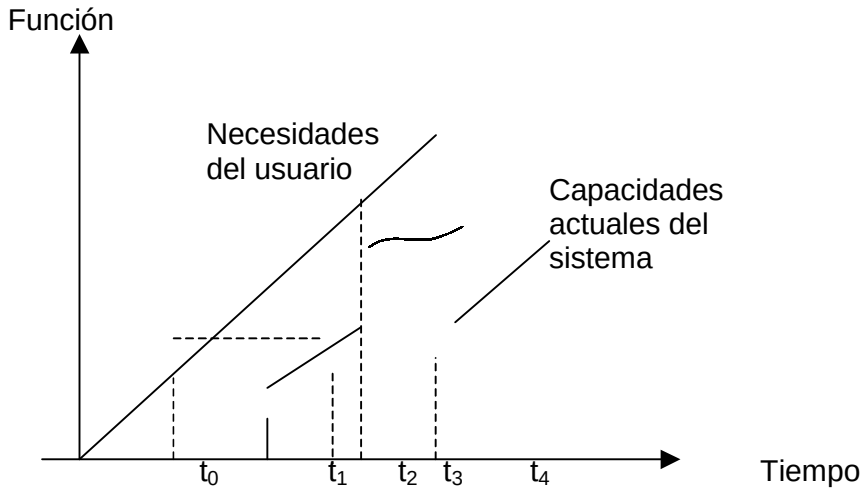

Figura 2.2. Falencias de oportunidad del software (De Davis [1988])

Una técnica para conseguir oportunidad es a través de la *entrega incremental* del producto. Esta técnica es ilustrada en el siguiente - más exitoso - ejemplo de la entrega de un compilador Ada por una compañía (real) diferente del descripto anteriormente. Esta compañía entregó muy tempranamente, un compilador que soportaba un muy pequeño subconjunto básico de lenguaje Ada, un subconjunto que es equivalente a un "paquete" en Pascal. El compilador no soportaba algunas de las nuevas formas del lenguaje, tales como el manejo de tareas y excepciones. El resultado fue la entrega temprana de un producto confiable. Como consecuencia el usuario comenzó a experimentar con el nuevo lenguaje y la compañía tomó más tiempo para entender las sutilezas de las nuevas

facilidades [features] de Ada. Tras varias entregas, que tomaron un período de dos años, el compilador Ada completo fue entregado.

La entrega incremental permite que el producto llegue a estar disponible en menos tiempo; y el uso del producto ayuda a refinar los requerimientos incrementalmente. Fuera de la ingeniería de software, un ejemplo clásico de la dificultad de tratar con los requerimientos de los sistemas complejos es ofrecido por sistemas de armas modernas. En varios casos bien publicitados, las armas han sido obsoletas por el tiempo que han sido entregadas, o ellas no han satisfecho los requerimientos, o, en muchos casos, ambas. Pero después de diez años de desarrollo, es difícil decidir qué hacer con un producto que no satisface un requerimiento declarado diez años antes. El problema es exacerbado por el hecho que los requerimientos no pueden ser formulados precisamente en estos casos porque la necesidad es para los sistemas más avanzados posible al tiempo de entrega, no al tiempo que los requerimientos son definidos.

Obviamente, la entrega incremental depende de la habilidad para abrir el conjunto de funciones del sistema requerido en subconjuntos que pueden ser entregados en incrementos. Si tales subconjuntos no pueden ser definidos, ningún proceso puede hacer que el producto esté disponible incrementalmente. Pero un proceso no incremental impide la producción de subconjuntos de productos aunque tales subconjuntos pueden ser identificados. La oportunidad puede conseguirla un producto que puede abrirse en subconjuntos y un proceso incremental.

La entrega incremental de subconjuntos no utilizables, por supuesto, no tiene valor. La oportunidad debe estar combinada con otras cualidades del software. En el capítulo 4 discutiremos muchas técnicas para obtener subconjuntos de productos, y en el capítulo 7 discutiremos técnicas para obtener procesos incrementales.

2.2.12. Visibilidad

Un proceso de desarrollo de software es visible si todos sus pasos y sus estados corrientes son claramente documentados. El término que describe mejor esta cualidad es la palabra Rusa *glasnost*. Otros términos son "transparencia" o "franqueza". La idea es que los pasos y el estado del proyecto estén disponibles y fácilmente accesible para el examen externo.

En muchos proyectos de software, la mayoría de los ingenieros y aún los ignoran el estado exacto del proyecto. Algunos pueden estar diseñando, otros codificando, y aún otros testeando, todos al mismo tiempo. Esto, por si mismo, no es malo. Ahora, si un ingeniero comienza a rediseñar una mayor parte del código justo antes que el software está supuestamente siendo entregado para el testeo de integración, el riesgo de problemas serios y demoras es alto.

La visibilidad permite a los ingenieros pesar el impacto de sus acciones y así guiarlas en la toma de decisiones. Esto permite a los miembros del grupo trabajar en la misma dirección, en lugar de, como es a menudo un caso corriente, en direcciones cruzadas. El ejemplo más común de la última situación es, como mencionamos más arriba, cuando el grupo de testeo de la integración ha estado testeando una versión del software asumiendo que la próxima versión involucrará defectos arreglados y será sólo mínimamente diferente de la versión actual, mientras que un ingeniero decide hacer un rediseño mayor para corregir un defecto menor. La tensión entre un grupo tratando de estabilizar el software mientras otra persona o grupo está desestabilizándolo - no intencionalmente, por supuesto - es común. El proceso debe estimular una visión consistente del estado y las metas actuales entre todos los participantes.

La visibilidad no es sólo una cualidad interna; es también externa. Durante el curso de un proyecto largo, hay muchas consultas sobre el estado del proyecto. A veces se requieren presentaciones formales del estado, y otras veces las consultas son informales. A veces los requerimientos vienen desde la dirección de la organización para el planeamiento futuro, y otras veces vienen desde afuera, quizás del cliente. Si el proceso de desarrollo del software tiene baja visibilidad, ese reporte de estado no será exacto, o requerirán un montón de esfuerzo prepararlo cada vez.

Una de las dificultades de dirección de proyectos grandes es manejar la rotación del personal. En muchos proyectos de software, la información crítica sobre los requerimientos de software y diseño tiene la forma de "folklore" conocido sólo por la gente que ha estado con el proyecto desde el comienzo o por un período de tiempo lo suficientemente largo. En tal situación, recuperarse de la pérdida de un ingeniero clave o sumar nuevos ingenieros al proyecto es muy dificultoso. En efecto, sumar nuevos ingenieros a menudo reduciría la productividad del proyecto entero mientras el "folklore" está siendo transferido lentamente desde la tripulación de ingenieros existentes a los nuevos ingenieros.

Los puntos anteriores que la visibilidad del proyecto requiere no sólo que todos los pasos del proceso estén documentados, sino también que el estado actual del producto intermedio, tales como las especificaciones de los requerimientos y las especificaciones del diseño, sean mantenidas sin errores; esto es, también es requerida la visibilidad del producto. Intuitivamente, un producto es visible si es claramente estructurado como una colección de módulos, con funciones claramente entendibles y de fácil acceso a la documentación.

---

[1] El presente es una traducción realizada por alumnos del curso 1999 del Master en Informática de la Universidad Nacional de La Matanza