Principios de la Ingeniería de Software

Introduccción

En este documento se presentan algunos principios generales de importancia, que son centrales para desarrollar software en forma exitosa, y que tratan tanto del proceso de ingeniería de software como del producto final. El proceso adecuado ayudará a desarrollar el producto deseado, pero también el producto deseado afectará la elección del proceso a utilizar. Un problema tradicional de la ingeniería de software es poner el énfasis en el proceso o en el producto excluyendo al otro, sin embargo, ambos son importantes.

Estos principios son suficientemente generales para ser aplicados a lo largo del proceso de construcción y gestión del software, sin embargo no son suficientes para guiar el desarrollo ya que describen propiedades deseables de los procesos y productos de software; para aplicarlos es necesario contar con métodos apropiados y técnicas específicas. Los métodos son guías generales que gobiernan la ejecución de alguna actividad, presentan enfoques rigurosos, sistemáticos y disciplinados, por otro lado, las técnicas son más mecánicas y se refieren a aspectos más "técnicos" que los métodos y tienen aplicación restringida. Una metodología es un conjunto de métodos y técnicas cuyo propósito es promover cierto enfoque para la resolución de un problema mediante ese conjunto seleccionado. Las herramientas son desarrolladas para apoyar la aplicación de técnicas, métodos y metodologías. Los principios son la base de todos los métodos, técnicas, metodologías y herramientas.

Rigor y formalidad:

En cualquier proceso creativo existe la tendencia a seguir la inspiración del momento de forma no estructurada, sin ser precisos; el desarrollo de software es de por sí una actividad creativa. Por otro lado, el rigor es un complemento necesario de la creatividad en todas las actividades de la ingeniería; únicamente a través de un enfoque riguroso podrán producirse productos más confiables, controlando sus costos e incrementando el grado de confianza en los mismos. El rigor no tiene por qué restringir la creatividad, por el contrario, puede potenciar la creatividad aumentando la confianza del ingeniero en los resultados de la misma, una vez que estos son analizados a la luz de evaluaciones rigurosas. Paradójicamente el rigor es una cualidad intuitiva que no puede ser definida en forma rigurosa, pero sí pueden alcanzarse varios niveles de rigurosidad siendo el más alto la formalidad.

La formalidad es un requerimiento más fuerte que el rigor: requiere que el proceso de software sea guiado y evaluado por leyes matemáticas. Obviamente formalidad implica rigor pero no a la inversa: se puede ser riguroso incluso informalmente. En todos los campos de la ingeniería el proceso de diseño sigue una secuencia de pasos bien definidos, establecidos en forma precisa y posiblemente probados, siguiendo en cada paso algún método o aplicando alguna técnica. Estos métodos y técnicas estarán basados en alguna combinación de resultados teóricos derivados de un modelado formal de la realidad, ajustes empíricos que tienen en cuenta fenómenos no presentes en el modelo, y métodos prácticos de evaluación que dependen de la experiencia pasada ("rules of thumb").

Un ingeniero debe saber cómo y cuándo ser formal si es requerido, entendiendo el nivel de rigor y formalidad que debe ser alcanzado dependiendo de la dificultad conceptual de la tarea y su criticidad, lo que puede variar para diferentes partes del mismo sistema. Por ejemplo, partes críticas pueden requerir una descripción formal de las funciones esperadas y un enfoque formal para su evaluación mientras que partes estándares o bien entendidas requerirán enfoques más simples. Esto aplica también en el caso de la ingeniería de software, por ejemplo en el caso de la especificación del software la cual puede establecerse

de forma rigurosa utilizando lenguaje natural o también puede darse formalmente mediante una descripción formal en un lenguaje de sentencias lógicas. La ventaja de la formalidad sobre el rigor es que puede ser la base para la mecanización del proceso, por ejemplo si se quiere utilizar la descripción formal para crear el programa si éste no existe, o para mostrar que el programa se corresponde con las especificaciones establecidas si tanto el programa como las especificaciones existen.

Tradicionalmente es en la fase de codificación donde se utiliza un enfoque formal ya que los programas son objetos formales: son escritos en un lenguaje cuya sintaxis y semántica están completamente definidas. Los programas son descripciones formales que son manipuladas automáticamente por los compiladores que chequean su correctitud y las transforman en una forma equivalente en otro lenguaje (assembler o lenguaje de máquina), todo lo cual es posible gracias a la utilización de la formalidad en la programación.

La aplicación del principio de rigor y formalidad tiene influencia beneficiosa en la obtención de cualidades del software como la confiabilidad, verificabilidad, mantenibilidad, reusabilidad, portabilidad, comprensibilidad e interoperabilidad. Por ejemplo, una documentación del software rigurosa o incluso formal puede mejorar todas estas cualidades sobre una documentación informal que puede ser ambigua, inconsistente e incompleta.

El principio de rigor y formalidad también se aplica al proceso de software; la documentación rigurosa del proceso ayuda a que éste sea reutilizado en proyectos similares y también ayuda a mantener un producto existente permitiendo que las modificaciones se realicen partiendo del nivel intermedio apropiado, en lugar de hacerlo solamente sobre el código final. Si el proceso de software está especificado en forma rigurosa, los gerentes podrán controlar su adecuación y evaluar su oportunidad para mejorar la productividad.

Separación de intereses:

Este principio permite enfrentarse a los distintos aspectos individuales de un problema de forma de concentrarse en cada uno por separado. En el desarrollo de un producto de software deben tomarse muchas decisiones como las funciones que serán ofrecidas, la confiabilidad esperada, eficiencia de tiempo y espacio, relaciones con el ambiente como recursos de software o hardware especial, interfaces de usuario, entre otras. Otras decisiones tienen que ver con el proceso de desarrollo como el ambiente de desarrollo, la organización y estructura del equipo, la agenda, los procedimientos de control, las estrategias de diseño, los mecanismos de recuperación frente a errores, entre otras. Y otras más que tienen que ver con temas económicos y financieros. Muchas de estas decisiones pueden no estar relacionadas entre sí por lo que obviamente podrán ser tratadas en forma separada, pero muchas otras estarán fuertemente relacionadas y será prácticamente imposible tener en cuenta todos los temas al mismo tiempo o por parte de las mismas personas. La única forma de enfrentar la complejidad de un proyecto es separar los distintos intereses.

La primer forma en la que se pueden separar los distintos intereses es según el tiempo, lo que permite planificar las distintas actividades y eliminar el trabajo extra que implica cambiar de una a otra en forma no restringida. Esta separación según el tiempo es la motivación que hay tras el ciclo de vida del software; un modelo racional de la secuencia de actividades que deberían seguirse en la producción de software.

Otra forma de separación de intereses es en términos de las cualidades que deberían tratarse por separado, por ejemplo podrían enfrentarse separadamente la eficiencia y correctitud de un programa, primero diseñándolo cuidadosa y estructuradamente para garantizar su correctitud a priori y luego reestructurarlo para mejorar su eficiencia.

Otro tipo importante de separación de intereses permite que distintas visiones del software sean analizadas en forma separada, por ejemplo al analizar los requerimientos de una aplicación podría ser de ayuda concentrarse por un lado en los datos que fluyen de una actividad a otra y por otro lado en el flujo de control que gobierna la sincronización de dichas actividades. Ambas ayudan a entender el sistema y ninguna de las dos provee una visión completa del mismo.

Otra forma más de aplicación de este principio es enfrentar partes del mismo sistema en forma separada, esto es en términos de tamaño. Este es un concepto fundamental que debe dominarse para enfrentar la complejidad de la producción de software, y es tan importante que se trata como un punto aparte bajo el principio de modularidad.

Si bien podrían perderse algunas optimizaciones potenciales al no tener en cuenta el problema en su

conjunto, la complejidad global puede resolverse mucho mejor concentrándose en los distintos aspectos por separado, incluso si no fuera posible descomponer el problema en los distintos aspectos en forma inmediata, es posible tomar inicialmente algunas decisiones de diseño generales y luego aplicar el principio de separación de intereses en forma efectiva.

Como observación final, la separación de intereses podría resultar en la separación de responsabilidades al enfrentarse a los distintos aspectos a tener en cuenta, por lo tanto es la base para dividir el trabajo en un problema complejo en asignaciones de trabajo específicas posiblemente a personas distintas con distintas habilidades.

Modularidad:

Un sistema complejo puede dividirse en piezas más simples llamadas módulos, un sistema compuesto de módulos es llamado modular. El principal beneficio de la modularidad es que permite la aplicación del principio de separación de intereses en dos fases: al enfrentar los detalles de cada módulo por separado ignorando detalles de los otros módulos, y al enfrentar las características globales de todos los módulos y sus relaciones para integrarlos en un único sistema coherente. Si estas fases son ejecutadas en ese orden se dice que el sistema es diseñado de abajo hacia arriba (bottom up), en el orden inverso se dice que el sistema es diseñado de arriba hacia abajo (top down).

El principio de modularidad tiene tres (3) objetivos principales: capacidad de descomponer un sistema complejo, capacidad de componerlo a partir de módulos existentes y comprensión del sistema en piezas (o pedazos).

La posibilidad de descomponer un sistema se basa en dividir en subproblemas de forma top down el problema original y luego aplicar el principio a cada subproblema en forma recursiva. Este procedimiento refleja el bien conocido principio de Divide y Vencerás (Divide & Conquer).

La posibilidad de componer un sistema está basada en obtener el sistema final de forma bottom up a partir de componentes elementales. Idealmente en la producción de software se quisiera poder ensamblar nuevas aplicaciones tomando módulos de una biblioteca y combinándolos para formar el producto requerido; estos módulos deberían ser diseñados con el objetivo expreso de ser reusables.

La capacidad de comprender cada parte de un sistema en forma separada ayuda a la modificabilidad del sistema. Debido a la naturaleza evolutiva del software muchas veces se debe volver hacia atrás al trabajo previo y modificarlo. Si el sistema solo puede ser comprendido como un todo las modificaciones serán difíciles de aplicar y el resultado será poco confiable. Cuando se hace necesario reparar el sistema, la modularización apropiada ayuda a restringir la búsqueda de la fuente de error a componentes separados.

Para alcanzar estos objetivos los módulos en los que se divida el sistema deben tener alta cohesión y bajo acoplamiento. Un módulo tiene alta cohesión si todos sus elementos están fuertemente relacionados y son agrupados por una razón lógica, esto es todos cooperan para alcanzar un objetivo común que es la función del módulo. La cohesión es una propiedad interna de cada módulo, por el contrario el acoplamiento caracteriza las relaciones de un módulo con otros. El acoplamiento mide la interdependencia de dos módulos, por ejemplo si el módulo A hace una llamada a una rutina provista por el módulo B o accede a una variable declarada por el módulo B. Si dos módulos dependen fuertemente uno del otro tienen un alto acoplamiento lo que los vuelve difíciles de analizar, comprender, modificar, testear o reusar en forma separada. Idealmente se quiere que los módulos de un sistema tengan bajo acoplamiento.

Una estructura modular con alta cohesión y bajo acoplamiento permite ver los módulos como cajas negras cuando se describe la estructura global del sistema y luego encarar cada módulo por separado cuando se analiza o describe la funcionalidad del módulo.

Abstracción:

La abstracción es un proceso mediante el cual se identifican los aspectos relevantes de un problema ignorando los detalles; es un caso especial del principio de separación de intereses en el cual se separan los aspectos importantes de los detalles de menor importancia. Lo que se abstrae y lo que se considera dependerá del propósito de la abstracción, por lo que podrán hacerse distintas abstracciones de la misma

realidad cada una de las cuales proveerá una visión de la realidad que sirve para un propósito específico.

Por ejemplo, cuando los requerimientos de una nueva aplicación son analizados y especificados se construye un modelo de la aplicación propuesta, el cual podrá ser expresado en varias formas dependiendo del grado requerido de rigor y formalidad. Sin importar cual sea el lenguaje elegido para expresar los requerimientos, lo que se provee es un modelo que abstrae los detalles que se decidió que podían ser ignorados en forma segura. Los lenguajes de programación también son abstracciones construidas sobre el hardware que proveen constructores útiles y poderosos para escribir programas ignorando detalles como el número de bits que se utilizan para representar números o los mecanismos de direccionamiento, lo que permite concentrarse en el problema a resolver en lugar de la forma de instruir a la máquina para hacerlo.

El principio de abstracción es un principio importante que se aplica tanto a los productos de software como a los procesos. En este último caso, por ejemplo, al realizar la estimación de costos para una nueva aplicación una forma posible es identificar algunos factores claves del nuevo sistema y extrapolar los valores a partir de perfiles de costo de sistemas previos similares. Los factores claves utilizados para realizar el análisis son abstracciones del sistema.

Anticipación al cambio:

El software sufre cambios constantemente, como se vio al tratar la mantenibilidad del software estos cambios pueden surgir por la necesidad de eliminar errores que no fueron detectados antes de liberar la aplicación, o por la necesidad de apoyar la evolución de la aplicación debido a nuevos requerimientos o cambios en los requerimientos existentes.

La habilidad del software para evolucionar no viene sola sino que requiere esfuerzo especial para anticipar cómo y cuándo pueden ocurrir estos cambios. Cuando se identifican posibles cambios futuros, se debe tener cuidado de proceder de forma que estos sean fáciles de aplicar, es importante aislar los posibles cambios en porciones específicas del software de tal forma que estén restringidos a esas partes.

La anticipación al cambio es posiblemente el principio que más distingue al software de otros tipos de producción industrial. Muchas veces una aplicación de software es desarrollada mientras sus requerimientos aún no están completamente comprendidos, al ser liberado y obtener retroalimentación del usuario debe evolucionar con nuevos requerimientos o cambios a los requerimientos ya existentes los cuales pueden tener distintos orígenes, por ejemplo debido a cambios en el ambiente de la organización. Por lo tanto este principio puede ser utilizado para lograr la evolucionabilidad del software y también la reusabilidad de componentes, viendo la reusabilidad como evolucionabilidad de granularidad más fina, a nivel de componentes.

La aplicación de este principio requiere que se disponga de herramientas apropiadas para gestionar las varias versiones y revisiones del software en forma controlada. Debe ser posible almacenar y recuperar documentación, fuentes, ejecutables, etc. de una base de datos que actúe como repositorio central de componentes reusables, y el acceso a la misma debe estar controlado. Un sistema de software debe mantenerse consistente, incluso cuando se aplican cambios a algunos de sus componentes. La disciplina que estudia esta clase de problemas es la Gestión de Configuración y se verá posteriormente.

La anticipación al cambio también aplica al proceso de desarrollo de software, por ejemplo, en la gestión del proyecto los gerentes deberían anticipar los efectos de una reducción de personal, estimar los costos y diseñar la estructura de la organización que apoyará la evolución del software, y decidir cuando vale la pena invertir tiempo y esfuerzo en la producción de componentes reusables tanto como parte de un proyecto de desarrollo de software o como un esfuerzo de desarrollo paralelo.

Generalidad:

El principio de generalidad establece que al tener que resolver un problema se debe buscar un problema más general que posiblemente esté oculto tras el problema original, puesto que puede suceder que el problema general no sea mucho más complejo (a veces puede ser incluso más simple) que el original y posiblemente la solución al problema general tenga potencial de reuso, o exista en el mercado como producto off-the-shelf, o se diseñe un módulo que puede ser invocado por más de un punto en la aplicación en lugar de tener varias soluciones especializadas.

Por otro lado, una solución general posiblemente sea más costosa en términos de rapidez de ejecución, requerimientos de memoria o tiempo de desarrollo, que una solución especializada al problema original, por lo que debe evaluarse la generalidad respecto al costo y la eficiencia al momento de decidir qué vale más la pena, una solución general o una especializada.

La generalidad es un principio fundamental si se tiene como objetivo el desarrollo de herramientas generales o paquetes para el mercado, ya que para ser exitosas deberán cubrir las necesidades de distintas personas. Estos productos de propósito general, off-the-shelf como por ejemplo los procesadores de texto, representan una tendencia general en el software; para cada área específica de aplicación existen paquetes generales que proveen soluciones estándares a problemas comunes. Esta tendencia es idéntica a lo que ocurrió en otras áreas de la industria como por ejemplo, los automóviles que en los inicios de la tecnología automotriz era posible hacer autos de acuerdo a los requerimientos específicos de un cliente, pero a medida que el área se fue industrializando solo podían encargarse a partir de un catálogo y actualmente no es posible pedir un diseño de auto personal a menos que se esté dispuesto a pagar una enorme cantidad de dinero.

Incrementalidad:

La incrementalidad caracteriza un proceso que se desarrolla en forma de pasos, en incrementos, alcanzando el objetivo deseado mediante aproximaciones sucesivas al mismo, donde cada aproximación es alcanzada a través de un incremento de la previa.

Una forma de aplicar el principio de incrementalidad consiste en identificar subconjuntos tempranos de una aplicación que sean útiles de forma de obtener retroalimentación (feedback) temprana del cliente. Esto permite que la aplicación evolucione en forma controlada en los casos en que los requerimientos iniciales no están estables o completamente entendidos. La motivación de este principio es que muchas veces no es posible obtener todos los requerimientos antes de comenzar el desarrollo de una aplicación sino que éstos van emergiendo a partir de la experimentación con la aplicación o partes de ésta. Por lo tanto, lo antes que se pueda contar con feedback del usuario sobre la utilidad de la aplicación, más fácil será incorporar los cambios requeridos la producto. Este principio está ligado al principio de anticipación al cambio y es otro de los principios en los que se basa la evolucionabilidad.

La incrementalidad se aplica a muchas de las cualidades del software vistas previamente. Se puede por ejemplo, comenzar con un núcleo de la aplicación que sea útil e ir agregando funcionalidades, también se puede agregar performance en forma incremental si por ejemplo, la versión inicial enfatizaba las interfaces de usuario y la confiabilidad, luego sucesivas liberaciones irán mejorando la eficiencia en tiempo y espacio.

Cuando se construye una aplicación en forma incremental, los pasos intermedios pueden ser prototipos del producto final, esto es solamente una aproximación al mismo. Obviamente un ciclo de vida basado en prototipos es bastante distinto al tradicional modelo en cascada, y está basado en un modelo de desarrollo más flexible e iterativo. Estas diferencias tendrán efectos no solo en los aspectos técnicos sino también en los organizativos y de gestión.

Como se mencionaba en el principio de anticipación al cambio, el desarrollo de software en forma evolutiva requiere tener especial cuidado en la gestión de documentación, programas, datos de testeo, etc. que son desarrollados para las varias versiones del software. Cada incremento significativo debe ser registrado, la documentación debe poder ser fácilmente recuperada, los cambios deben aplicarse en forma ordenada, etc.
Si lo anterior no se realiza con cuidado, un intento de desarrollo evolutivo podría rápidamente transformarse en un desarrollo de software indisciplinado y perderse todas las ventajas potenciales de la evolucionabilidad.

A partir de: Fundamentals of Software Engineering – Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli. Prentice-Hall, Inc. 1991, edición en inglés. ISBN-0-13-820432-2, Capítulo 3 – Software Engineering Principles.