17 Apéndice E: Algunos Consejos para Programadores

Una de las diferencias entre el desarrollo de Software de un aficionado y profesional es si estás escribiendo Software para ti o para otra persona -es como la diferencia entre llevar un diario y ser periodista-. Las personas que sólo han escrito Software para su propio uso no tienen idea de cuánto trabajo implica escribir Software para otros. Tienes que imaginar mil cosas que podría hacer un usuario y que tú nunca harías. Tienes que decidir cuáles de estas cosas aceptarás y cuáles no permitirás. Y cuando decide no permitir una acción, debe decidir cómo hacerlo causando una irritación mínima al usuario.

Las aplicaciones GUI son particularmente difíciles de escribir, no porque sea difícil dibujar botones y cuadros en una pantalla, sino porque es difícil pensar en todas las formas en que un usuario podría llegar a un estado particular.

Entre escribir Software para usted y escribir Software para otros está escribir Software para personas muy parecidas a usted. El Software de código abierto comenzó de esta manera: los programadores alfa escribían Software para programadores alfa. Desde entonces, la comunidad Software libre ha mejorado mucho en la escritura de Software para usuarios generales, aunque todavía tiene margen de mejora.

Las personas que no son programadores profesionales a menudo no se dan cuenta de cómo la dificultad de escribir Software aumenta con el tamaño. Muchas personas que escribieron programas de 100 líneas en la universidad imaginan que podrían escribir programas de 1000[153] líneas si trabajaran en ello 10 veces más. O peor aún, imaginan que podrían escribir programas de 10,000 líneas si trabajaran 100 veces más.

No funciona de esa manera. La mayoría de las personas que pueden escribir un programa de 100 líneas nunca podrían terminar un programa de 10,000 líneas, sin importar cuánto tiempo trabajaron en él. Simplemente se ahogarían en la complejidad. Una de las características de un programador profesional es saber organizar el Software para que la complejidad siga siendo manejable a medida que aumenta el tamaño. Incluso entre profesionales

[153] Cuando hablo de un programa que tiene tantas líneas, me refiero a un programa que necesita tener esa longitud. No es ningún logro escribir 1000 líneas de código para un problema que sería razonable resolver en 10.

existen grandes diferencias en capacidad. Los programadores que pueden gestionar eficazmente proyectos de 100,000 líneas están en una liga diferente que aquellos que pueden gestionar proyectos de 10,000 líneas.

Escribir programas con errores grandes es difícil. Decir que un programa tiene errores es implicar que al menos tiene la calidad suficiente para aproximarse a lo que se supone que debe hacer la mayor parte del tiempo. Por ejemplo, no dirías que el Bloc de notas es un navegador Web con errores. Un programa tiene que mostrar páginas Web al menos ocasionalmente para ser considerado un navegador con errores.

Escribir programas extensos y correctos es mucho más difícil. Es incluso imposible, dependiendo de lo que entiendas por "grande" y "correcto". Ningún programa grande está completamente libre de errores, pero algunos programas grandes tienen una probabilidad muy pequeña de fallar.

Los mejores programadores pueden pensar en una docena de formas de resolver cualquier problema y eligen la forma que creen que tiene mayores posibilidades de implementarse correctamente. O eligen la forma que tiene más probabilidades de hacer evidente un error, si llega a ocurrir. Saben que es necesario probar el Software y lo diseñan para que sea más fácil de probar.

Si le preguntas a un aficionado si su programa es correcto, es probable que se ofenda. Te dirán que por supuesto que es correcto porque tuvieron cuidado al escribirlo. Si le hace la misma pregunta a un profesional, es posible que le diga que su programa probablemente tenga errores, pero luego le dirá cómo lo han probado y qué funciones de registro existen para ayudar a depurar errores cuando aparezcan más adelante.

Programadores Experimentados y Líneas de Código. Escuché recientemente de un estudio que concluyó que los programadores experimentados y sin experiencia escriben aproximadamente la misma cantidad de líneas de código por día . La diferencia es que los programadores experimentados conservan más líneas de código, logrando un progreso constante hacia una meta. Los programadores menos experimentados escriben grandes fragmentos de código sólo para arrancarlos y reescribirlos muchas veces hasta que el código parece funcionar. O en lugar de extraer el código, lo depuran durante días, cambiando una o dos líneas a la vez, casi al azar, hasta que el código parece funcionar.

Centrarse en la calidad en el desarrollo de Software a menudo también resulta en una mayor productividad. Se dedica más esfuerzo al progreso y

menos al retrabajo.

Los programadores experimentados no sólo producen más líneas de código que vale la pena conservar cada día, sino que también logran más por línea de código, a veces muchísimo más. Pero eso no es noticia. Es bien sabido que los mejores programadores no son sólo un poco más productivos que el promedio, sino que son uno o dos órdenes de magnitud más productivos[154]. Lo más interesante es que los mejores programadores no parecen tener una capacidad mucho mayor para producir y comprender líneas de código.

También se han realizado estudios que muestran que los programadores producen aproximadamente la misma cantidad de líneas de código por día, independientemente del lenguaje que utilicen. Se podría pensar que alguien que trabaja en lenguaje ensamblador podría producir más líneas por día que alguien que escribe en un lenguaje de nivel superior como VB o Java, pero ese no es el caso. Parece que, si bien contar líneas de código es una forma terrible de medir la productividad, es una buena manera de medir lo que se puede esperar que alguien pueda tener en la cabeza.

¿Un Enfoque de Software 100 Veces Mejor? Tenga en cuenta que aquí no estamos hablando de la productividad del programador individual. Las discusiones sobre programadores 10x o 100x generalmente suponen que se trata de personas que pueden utilizar básicamente el mismo enfoque y las mismas herramientas para realizar mucho más trabajo. Aquí estamos pensando en mejores enfoques más que en mejores trabajadores.

Digamos que tiene un sistema típico de 10,000,000 de líneas de código. ¿Cuántas líneas de código requeriría un sistema con las mismas características deseadas (no necesariamente todas las características reales)?:

- ¿Si el mismo equipo de desarrolladores hubiera trabajado desde el principio con un mejor diseño?
- ¿Si el mismo equipo de desarrolladores pudiera reescribir el sistema desde cero usando lo que aprendieron del sistema original?
- ¿Si un grupo reflexivo de desarrolladores pudiera reescribir el sistema sin presión de tiempo?
- Si una inteligencia sobrehumana pudiera reescribir el sistema, ¿algo que se acercara a la complejidad de Kolmogorov?

[154] Véase, por ejemplo, el libro de Joel Spolsky Smart and Gets Things Done

¿A dónde va el esfuerzo desperdiciado en los sistemas grandes y cuánto podría eliminarse? Parte del esfuerzo se destina a descubrir requisitos. Los sistemas grandes nunca comienzan con una especificación completa e inmutable. Eso se aborda en las dos primeras preguntas.

Hay una regla que dice que las burocracias siguen una ley de 3/2: para duplicar la productividad, se necesitan tres veces más empleados. Si un solo desarrollador pudiera producir 10,000 líneas de código en un período de tiempo, necesitaría duplicar esa cantidad 10 veces para llegar a 10,000,000 de líneas de código. Eso requeriría triplicar su fuerza laboral 10 veces, lo que daría como resultado más de 57,000 programadores. Esto suena razonable, tal vez incluso un poco optimista.

¿Es así como deben ser las cosas, que los genios escasean y la inteligencia ordinaria no escala de manera eficiente? ¿Cuánto mejor podríamos hacer con el talento disponible si adoptáramos un mejor enfoque para desarrollar Software?

Por qué los Programadores Escriben Código Innecesario Los programadores escriben una gran cantidad de código que nunca se utiliza. Hay numerosas razones para esto. En el libro Coders at Work de Peter Seibel , Peter Norvig explica por qué sucede esto:

> Seibel : ¿Por qué resulta tan tentador resolver un problema que en realidad no tenemos?
>
> Norvig : Quieres ser inteligente y quieres cerrar; quieres completar algo y pasar a otra cosa. Creo que las personas están hechas para manejar solo una cierta cantidad de cosas y quieres decir: "Esto está completamente hecho; Puedo sacarlo de mi mente y luego seguir adelante".

A veces los desarrolladores de Software creen que existe una alta probabilidad de que en el futuro se necesite alguna característica no solicitada. En general, sobreestiman dichas probabilidades. El acrónimo *no vas a necesitarlo* o YAGNI (You Aren´t Gonna Meed It) pretende recordar a los desarrolladores esta tendencia.

Es una gran sensación decir "ya hice eso" cuando alguien solicita una nueva función. Entonces eres el héroe, el sabio que anticipó lo que había que desarrollar. Cuando escribes código que no es necesario, tal vez nadie se da cuenta y puedes consolarte sabiendo que todavía llega el momento en que el

mundo querrá tu característica. Los momentos en los que acertaste son más vívidos en tu mente que los momentos en los que te equivocaste, por lo que sobreestimas la frecuencia con la que acertaste.

Pero a veces vale la pena resolver un problema mayor del necesario. Puede que tenga sentido crear una solución más completa de la que es actualmente necesaria mientras el problema está fresco en su mente ; Será más difícil retomar el problema en el futuro, así que si alguna vez vas a escribirlo, ahora es el momento.

Norvig señala con razón el lado negativo de buscar el cierre. Quizás el último 2% sea intelectualmente satisfactorio pero terriblemente difícil y no valga la pena el esfuerzo. Me he equivocado en ambos lados . Hace años, a menudo me equivocaba al desarrollar funciones que nunca se utilizaban. Luego, leer a Kent Beck me convenció de que YAGNI suele ser cierto. Desde entonces, me he equivocado al desear haber hecho más mientras tenía un proyecto fresco en la mente.

¿A Dónde va el Esfuerzo de Programación? Me encontré con esta cita de Mary Shaw:

> Menos del 10% del código tiene que ver con el propósito aparente del sistema; el resto se ocupa de entrada y salida, validación de datos, mantenimiento de la estructura de datos y otras tareas domésticas.

No conozco el contexto de su cita, pero podría estar hablando de cualquier proyecto de Software.

Cuando comencé a trabajar como programador profesional, me sorprendió que dedicaba la mayor parte de mi tiempo a trabajos que no estaban directamente relacionados con lo que quería lograr. Las clases de informática y la escritura de Software para mi propia investigación no me habían preparado para esto. Seguí pensando que a medida que adquiriera más experiencia, aumentaría la proporción de mi esfuerzo destinado directamente a lo que quería lograr. Lo hizo, pero muy lentamente y nunca mucho. Sólo más tarde se me ocurrió la razón: la gran mayoría del trabajo que hay que hacer no está directamente relacionado con el objetivo del proyecto.

Las personas que saben un poco sobre programación pueden convertirse en clientes difíciles porque pueden imaginar cómo podrían escribir el 10% central (o tal vez el 2%) central de un proyecto grande.

De vez en cuando alguien afirma tener una solución que cambiará las cosas. Están vendiendo un marco, un lenguaje, etc. que cambiará radicalmente las cosas. El argumento de venta es: "Pasas la mayor parte de tu tiempo haciendo cosas de bajo nivel. Utilice mi producto y sus programadores podrán centrarse en la parte de valor añadido y no hacer tanta plomería". Pero la plomería es un trabajo con valor agregado . (Llámelo "infraestructura" si lo desea; eso suena más importante). A veces, el trabajo de plomería se vuelve repetitivo y se puede reducir reutilizando el código, pero siempre hay nuevas plomería en las que trabajar. La mayor parte del trabajo por hacer es invisible y no preveo que esto cambie en el corto plazo.

Por qué a los Programadores no se les Paga en Proporción a su Productividad Los programadores más productivos son órdenes de magnitud más productivos que los programadores promedio. Pero los salarios suelen estar dentro de un rango bastante pequeño en cualquier empresa. Incluso en toda la profesión, los salarios no varían tanto. Si algunos programadores son 10 veces más productivos que otros, ¿por qué no se les paga 10 veces más?

En primer lugar, la productividad del programador varía enormemente según la profesión, pero puede que no varíe tanto dentro de una empresa determinada. Es probable que alguien que sea 10 veces más productivo que sus colegas se vaya, ya sea para trabajar con otros programadores muy talentosos o para iniciar su propio negocio. En segundo lugar, la productividad extrema puede no ser obvia.

¿Cómo puede alguien ser 10 veces más productivo que sus compañeros sin que nadie se dé cuenta? En algunas profesiones esa diferencia sería obvia. Un vendedor que vende 10 veces más que sus pares llamará la atención y se le compensará en consecuencia. Las ventas son fáciles de medir y algunos vendedores ganan muchísimo más dinero que otros. Si un albañil fuera 10 veces más productivo que sus pares, esto también sería obvio, pero no sucede: los mejores albañiles no pueden colocar 10 veces más ladrillos que los albañiles promedio. La producción de Software no se puede medir tan fácilmente como los dólares o los ladrillos. Los mejores programadores no necesariamente escriben 10 veces más líneas de código y ciertamente no trabajan 10 veces más horas.

Los programadores son más eficaces cuando evitan escribir código. Es posible que se den cuenta de que el problema que se les pide que resuelvan no necesita ser resuelto, que el cliente en realidad no quiere lo que están pidiendo.

Es posible que sepan dónde encontrar código reutilizable o reeditable que resuelva su problema. Pueden hacer trampa . Pero justo cuando están siendo más productivos, nadie dice "¡Guau! Eras 100 veces más productivo que si lo hubieras hecho de la manera más difícil. Te mereces un aumento". En el mejor de los casos dicen "¡Buena idea!" y continuar. Puede llevar un tiempo darse cuenta de que a alguien se le ocurren de forma rutinaria ideas que le permiten ahorrar tiempo. O, para decirlo negativamente, puede llevar mucho tiempo darse cuenta de que otros están programando con sonido y furia pero no producen nada.

La imagen romántica de un superprogramador es la de alguien que enciende Vi , escribe como una ametralladora y ofrece un producto final impecable desde cero. Una imagen más precisa sería la de alguien que mira en silencio al vacío durante unos minutos y luego dice "Hmm. Creo que he visto algo como esto antes".

17.1 Científicos Vs Programadores

Por otro lado, existe una división importante entre la forma en que los científicos y los programadores ven el Software que escriben:

- Los científicos ven su Software como una extensión de ellos mismos. El Software puede hacer un trabajo pesado, pero los científicos siguen participando activamente en su uso. El Software es una herramienta, no un producto autónomo.

- Los programadores ven su Software como algo que entregarán a otra persona, los programadores creen que su trabajo es encapsular la inteligencia en Software. Si los usuarios tienen que depender de los programadores después de escribir el Software, los programadores no terminaron su trabajo.

Trabajo con científicos y programadores, a menudo tendiendo puentes entre las dos culturas. Un punto de tensión es definir cuándo se realiza un proyecto. Para un científico, el Software está listo cuando obtiene lo que quiere de él, como una tabla de números para un artículo. Los programadores profesionales prestan más atención a la reproducibilidad, la mantenibilidad y la corrección.

Los programadores deben comprender que a veces un programa sólo necesita ejecutarse una vez, en un conjunto de entradas, con supervisión de expertos. Los científicos deben comprender que el código prototipo puede necesitar una reescritura completa antes de poder utilizarlo en producción.

La verdadera tensión surge cuando de repente se espera que un Software de investigación esté listo para producción. El científico dirá "el código ya ha sido escrito" y no puede imaginar que se requiera mucho trabajo, si es que requiere alguno, para preparar el Software para sus nuevas responsabilidades.

17.2 Bueno, Rápido o Barato: ¿Realmente Puedes Elegir Dos?

Hay un dicho que dice que los clientes pueden obtener productos buenos, rápidos o baratos. Elija dos, pero luego la tercera será lo que sea necesario en función de las otras dos opciones. Puedes tener algo bueno y rápido si estás dispuesto a gastar mucho dinero. Puedes conseguirlo rápido y barato, pero la calidad será mala. Incluso podrías conseguir algo bueno y barato, si estás dispuesto a esperar mucho tiempo.

Una variación de este tema es el triángulo de hierro[155]. Dibujas un triángulo con vértices etiquetados como "características", "tiempo" y "recursos". Si alargas dos de los lados, el tercero también debe alargarse. Aquí la bondad se define como un conjunto de características más que como calidad, pero se aplica el mismo principio.

Hay un problema con esta línea de razonamiento: no importa lo que digan los clientes, ellos quieren calidad. Es posible que digan que quieren productos rápidos y baratos, y si les dices que sacrificarás la calidad para

[155]Las iniciativas de desarrollo de Software a menudo fracasan porque la organización establece objetivos poco realistas para el "triángulo de hierro" del desarrollo de Software:

- Alcance (lo que se debe construir)
- Horario (cuándo debe construirse)
- Recursos (cuánto debe costar)

El equipo de desarrollo no ha logrado renegociar la situación y de todos modos se ve obligado a intentar cumplir esas limitaciones. Al final, si el equipo cumple, la calidad del producto entregado se ve afectada y la iniciativa casi siempre llega tarde y excede el presupuesto de todos modos. Para ser eficaces, los directores de proyectos deben comprender las implicaciones del triángulo de hierro.

entregar productos rápidos y baratos, serás un héroe... hasta que cumplas. Entonces quieren calidad. Como lo expresó Howard Newton

> La gente olvida lo rápido que hiciste un trabajo, pero recuerdan lo bien que lo hiciste.

A veces puedes eliminar características siempre y cuando hagas un buen trabajo en las características que quedan, pero solo hasta cierto punto. Los clientes no estarán contentos a menos que usted cumpla con sus expectativas, incluso si esas expectativas se contradicen explícitamente en un contrato. Puedes decirle a un cliente que eliminarás adornos para darle algo rápido y barato, y él estará de acuerdo con gusto. Pero todavía quieren sus lujos, o los querrán. El cliente puede sentirse silenciosamente decepcionado. O pueden estar abiertamente decepcionados, exigiendo funciones excluidas de forma gratuita y quejándose de tu trabajo. Con el tiempo, aprenderá qué funciones insistir en incluir, incluso si un cliente dice que puede vivir sin ellas.

La Arquitectura del Software en Función de la Confianza. Las discusiones sobre arquitectura de Software dan la impresión de que la única preocupación es el dominio del problema: cómo estructurar un sistema de gestión de contenidos, cómo estructurar un procesador de textos, etc. Esto deja fuera a las personas que desarrollarán el Software.

¿Cuánto confías en tus desarrolladores de Software? ¿Cuánto confías en su habilidad y su integridad? ¿Quiere apartarse del camino de sus desarrolladores o quiere protegerse contra desarrolladores incompetentes?

Es incómodo hablar de esto, por lo que la decisión suele quedar implícita. Nadie quiere decir en voz alta que están diseñando Software para que lo implemente un ejército de programadores mediocres, pero esa es la suposición predeterminada. Y con razón. La mayoría de los desarrolladores tienen una capacidad media, por definición.

Cuando los programadores destacados se quejan de los enfoques comunes para desarrollar Software, es posible que no consideren que la mayor parte del Software no está escrito por programadores destacados y la diferencia que eso supone. Por ejemplo, he escuchado a innumerables grandes programadores quejarse de Java. Pero Java no fue escrito para grandes programadores. Fue escrito para programadores promedio. Las restricciones en el lenguaje que irritan a los grandes programadores son beneficiosas para los equipos de programadores promedio.

Si confía en que sus desarrolladores son muy competentes y autodisciplinados, organizará su Software de manera diferente que si asume que los desarrolladores tienen habilidades y disciplina mediocres. Una forma en que esto se manifiesta es en qué medida estás dispuesto a confiar en las convenciones para mantener el orden. Por ejemplo, la arquitectura detrás de Emacs es notablemente simple y depende en gran medida de las convenciones. Este enfoque ha servido bastante bien a Emacs, pero no funcionaría para un gran equipo de desarrolladores mediocres. (Tampoco funcionaría con el Software que controla los frenos de un automóvil. Los errores en los editores de texto no tienen las mismas consecuencias).

En general, veo más dependencia de las convenciones en proyectos de código abierto que en proyectos empresariales. Una posible explicación es que los proyectos de código abierto tienen desarrolladores más motivados. No todos los desarrolladores de código abierto son voluntarios, pero muchos sí lo son. Y los voluntarios no sólo están más motivados, sino que también son más fáciles de despedir que los empleados. Si el código de alguien no cumple con los estándares, el proyecto puede simplemente negarse a utilizar su código. En teoría se podría decir lo mismo de un proyecto de Software empresarial, pero en la práctica no es tan sencillo.

Dos Tipos de Desafíos de Software Sobre lo que las universidades deberían enseñar en ingeniería de Software:

Durante años hemos enfatizado:

- Velocidad de ejecución,
- Limitaciones de memoria,
- Organización de datos,
- Gráficos llamativos, y
- Algoritmos para lograr todo esto.

Los planes de estudios también deben hacer hincapié:

- Robustez,
- Pruebas,

- Mantenibilidad,
- Facilidad de reemplazo,
- Seguridad, y
- Verificabilidad.

Los criterios de la primera lista son principalmente matemáticos. Los criterios de la segunda lista tienen más que ver con la naturaleza humana. Por ejemplo, el código se puede mantener si está organizado de manera que una persona pueda entenderlo y modificarlo fácilmente. Eso es una cuestión de psicología.

Más proyectos fracasan por problemas con la segunda lista. Los problemas con la primera lista tienden a estar localizados. Los problemas con la segunda lista tienden a impregnar todo proyecto. Una persona inteligente puede encontrar una solución rápida para los problemas de la primera lista. Las soluciones rápidas son raras para los problemas de la segunda lista.

El Código con Errores es un Código Sesgado Por muy malo que sea el Software corporativo, el Software académico suele ser peor. He trabajado en la industria y en el mundo académico y he visto de primera mano cuánto más bajo es el listón de calidad en el mundo académico -y no soy el único que se ha dado cuenta de esto-.

¿Por qué esto importa? Porque el código con errores es un código sesgado. Es más probable que se detecten y solucionen los errores que hacen que el Software proporcione resultados no deseados. Es más probable que los errores que hacen que el Software produzca los resultados esperados permanezcan vigentes.

Si su Software simula algunos fenómenos complejos, no sabe qué se supone que debe hacer; por eso estás simulando. Los errores son más fáciles de detectar en el Software de consumo. Un modelo climático necesita un mayor nivel de garantía de calidad que un procesador de textos porque los errores en este último son más obvios. El análisis genómico puede contener errores atroces y nadie lo sabe nunca, pero un error en un reproductor MP3 seguramente molestará a los usuarios.

Tienes que probar el Software de simulación cuidadosamente. Debe probar casos especiales y componentes individuales para tener confianza en el

resultado final. No puedes simplemente mirar los resultados y decir: "Sí, eso es lo que esperaba".

El Software que usa mucha gente tiene menos errores que el Software que usa poca gente, del mismo modo que los teoremas que la gente construye tienen menos errores que los que "dejan en la red del conocimiento". Es más probable que se reutilicen las subrutinas y bibliotecas útiles que los programas completos. Y como señaló Donald Knuth, el código reeditable es mejor que el código reutilizable de caja negra.

Todo el mundo sabe que el Software tiene errores, pero no todo el mundo se da cuenta de cuán defectuosos son los teoremas. Los errores en el Software son más obvios porque el papel no genera un fallo. Las pruebas y los programas son formas complementarias de validación. Intentar demostrar la exactitud de un algoritmo ciertamente reduce las posibilidades de que se produzca un error, pero las pruebas también son falibles. Citando nuevamente a Knuth, dijo una vez: "Cuidado con los errores en el código anterior; Sólo lo he demostrado correcto, no lo he probado". Los programas no solo pueden beneficiarse al parecerse más a una prueba, sino que las pruebas también pueden beneficiarse al parecerse más a un programa.

17.3 Escogiendo un Lenguaje de Programación para un Proyecto

¿Julia, Scala, Lua, TypeScript, Haskell, Go o Dart? A veces se proponen varios lenguajes informáticos nuevos y antiguos como mejores alternativas a los lenguajes convencionales. Pero en comparación con opciones convencionales como Python, C, C++ y Java, ¿vale la pena usarlas?

Sin duda, depende mucho del uso previsto: ¿se trata de un pequeño proyecto puntual o de una gran aplicación de Software a escala industrial?. Sin embargo, incluso un proyecto aislado puede crecer rápidamente hasta alcanzar una escala de producción, con los consiguientes problemas de crecimiento.

Las empresas emergentes a veces enfrentan una crisis de crecimiento cuando la base de código naciente se vuelve difícil de manejar y debe ser refactorizada o reescrita por completo (o podría hacer lo que hizo Facebook/Meta y simplemente escribir un nuevo compilador para que su base de código existente funcione mejor).

El alcance de los diferentes tipos de proyectos de Software y sus requisitos

es tan increíblemente diverso que cualquier punto de vista de la experiencia corre el riesgo de ser miope y, por tanto, inexacto para otros tipos de proyectos. Con esta advertencia, compartiré algo de mi propia experiencia al observar proyectos para muchas docenas de aplicaciones de Software a escala de producción escritas para sistemas informáticos de alto rendimiento a escala de liderazgo. Por lo general, se encuentran en una escala de 20 000 a 500 000 líneas de código y a menudo requieren soporte de bibliotecas y Middleware[156] matemáticos y científicos para soporte de compilación, paralelismo, visualización, E/S, gestión de datos y aprendizaje automático.

Estas son algunas de las principales cuestiones relacionadas con la elección de lenguajes de programación y compiladores para estos códigos:

- Lenguaje y sostenibilidad del compilador. Si bien la vida útil de los sistemas informáticos se mide en años, la vida útil de la base del código de una aplicación a veces se puede medir en décadas. ¿Es probable que el lenguaje en el que está escrito sobreviva y cuente con un buen soporte en el futuro? Por ejemplo, Fortran, aunque todavía se utiliza y se admite con frecuencia, es un lenguaje menos común, por lo que requiere un esfuerzo especial por parte de los proveedores, con menos recursos para desarrolladores que los lenguajes más populares. ¿Existe una diversidad de compiladores múltiples de diferentes proveedores para mitigar el riesgo? Un único proveedor significa un único punto de falla, un alto riesgo; ¿Qué pasa si el proveedor pierde financiación? ¿Es probable que el lenguaje y los compiladores se adapten a las tendencias futuras del Hardware informático (aunque a veces esto es difícil de predecir)? ¿Existe una gran base de clientes que ayude a garantizar el soporte futuro? De manera similar, ¿existe un grupo adecuado de programadores disponibles profundamente capacitados en el lenguaje? ¿Tiene el lenguaje un ecosistema de biblioteca estándar con buenas funciones y buen soporte para bibliotecas y marcos de terceros? ¿Existe un buen soporte de herramientas (depuradores, perfiladores, herramientas de compilación)?

[156] El Middleware es un Software con el que las diferentes aplicaciones se comunican entre sí. Brinda funcionalidad para conectar las aplicaciones de manera inteligente y eficiente, de forma que se pueda innovar más rápido. El Middleware actúa como un puente entre tecnologías, herramientas y bases de datos diversas para que pueda integrarlas sin dificultad en un único sistema. Este sistema único provee un servicio unificado a sus usuarios.

- Relacionada con esto está la cuestión de la gobernanza del lenguaje. ¿Cómo se toman las decisiones sobre futuras actualizaciones del lenguaje? ¿Existe una amplia participación de la comunidad de usuarios y capacidad de respuesta a sus necesidades? He sabido de miembros del comité de lenguaje C++; Según mi limitada experiencia, parecen muy razonables y reflexivos sobre las direcciones futuras del lenguaje. Por otro lado, algunos estándares han introducido características que casi nadie utiliza: una pérdida de tiempo y más desorden para el estándar.

- Productividad. Se dice que la productividad del programador está limitada por la capacidad de unas pocas líneas de código para expresar abstracciones de alto nivel que pueden hacer mucho con una sintaxis mínima. ¿El lenguaje lo permite? ¿Tiene sentido el estándar del lenguaje (coherente, cohesivo) y sigue el principio de mínima sorpresa? Al mismo tiempo, el lenguaje no debería abarcar lo que una biblioteca podría manejar mejor de forma modular. Por ejemplo, un producto matriz-matriz que está vinculado al lenguaje puede ser muy productivo para casos simples pero tener dificultades para admitir las muchas variantes del producto matriz-matriz proporcionadas, por ejemplo, por la biblioteca NVIDIA CUTLASS. Además, el soporte en el lenguaje para las operaciones de GPU CUDA, por ejemplo, haría difícil que el lenguaje no se quede atrás en soporte de las nuevas versiones frecuentes de CUDA.

- Ventaja estratégica. La regla de mejora 10X establece que sólo vale la pena adoptar una innovación si ofrece una mejora 10X en comparación con la práctica existente según alguna métrica. Si cambiar a un nuevo lenguaje determinado no aporta una mejora significativa, puede que no valga la pena hacerlo. Esto es particularmente cierto si existe una base de código de algún tamaño. Una pregunta relacionada es si el nuevo lenguaje ofrece una ruta de transición incremental para un código existente al nuevo lenguaje (en muchos casos esto es difícil o imposible).

- Rendimiento (velocidad de ejecución). ¿Permite el lenguaje llegar al rendimiento básico en lugar de pasar por abstracciones costosas o una capa de intérprete? ¿Están expuestas las características del Hardware subyacente para que el usuario pueda acceder a ellas? ¿Es predecible el desempeño? ¿Se puede tener una idea del rendimiento de cada línea de código simplemente mediante inspección, o esto está bloqueado por

abstracciones o un proceso de compilación complejo? ¿Es impredecible el uso de la compilación justo a tiempo o la recolección de basura, lo que podría ser un problema para la computación paralela en la que un proceso que realiza inesperadamente una de estas operaciones puede causar "bloqueos" inesperados? ¿Los desarrolladores del compilador brindan un buen soporte para opciones de optimización de código efectivas y precisas?

Los primeros usuarios proporcionan un vibrante sistema de "alerta temprana" para el desarrollo de nuevos lenguajes y herramientas que son útiles para proyectos pequeños y pueden tener un gran impacto. Python fue reconocido tempranamente en la comunidad informática científica por su potencial uso complementario con lenguajes estándar para grandes cálculos científicos. Sin embargo, cuando se trata de planificar proyectos de Software a gran escala, se debe considerar una variedad de factores basados en los requisitos del proyecto para garantizar la mayor probabilidad de éxito.

17.4 ¿Cómo Hago mi Programa?

En su artículo "How to program it", Simon Thompson plantea algunas preguntas a sus alumnos que son muy útiles para la etapa de diseño:

- ¿Han visto este problema antes, aunque sea de manera ligeramente diferente?.
- ¿Conocen un problema relacionado? ¿Conocen un programa que pueda ser útil?.
- Fíjense en la especificación. Traten de encontrar un problema que les resulte familiar y que tenga la misma especificación o una parecida.
- Acá hay un problema relacionado con el que ustedes tienen y que ya fue resuelto. ¿Lo pueden usar? ¿Pueden usar sus resultados? ¿Pueden usar sus métodos? ¿Pueden agregarle alguna parte auxiliar a ese programa del que ya disponen?.
- Si no pueden resolver el problema propuesto, traten de resolver uno relacionado. ¿Pueden imaginarse uno relacionado que sea más fácil de resolver? ¿Uno más general? ¿Uno más específico? ¿Un problema análogo?.

- ¿Pueden resolver una parte del problema? ¿Pueden sacar algo útil de los datos de entrada? ¿Pueden pensar qué información es útil para calcular las salidas? ¿De qué manera se pueden manipular las entradas y las salidas de modo tal que estén "más cerca" unas de las otras? ¿Usaron todos los datos de entrada?.
- ¿Usaron las condiciones especiales sobre los datos de entrada que aparecen en el enunciado? ¿Han tenido en cuenta todos los requisitos que se enuncian en la especificación?.

Otros conjunto de consejos es conocido como SOLID de la programación orientada a objetos y el segundo grupo de consejos son los 10 Principios de Diseño para POO.

17.5 Diez Principios de la Usabilidad

Cuando desarrollamos un programa o aplicación informática hay algunos principios que nos pueden guiar para su usabilidad, estos son:

1. Visibilidad del estatus del sistema: *Le da certeza a los usuarios de lo que ocurre en cada paso.*
2. Compatibilidad entre el sistema y la vida real: *Entre la experiencia de la vida real y la experiencia digital.*
3. Dale libertad al usuario: *Genera una vía de escape.*
4. Consistencia y estándares: *Mantenlo simple.*
5. Prevén el error: *El usuario seguro encontrará la forma de cometer un error.*
6. Reconocer vs. Recordar: *Cuanta más información se le proporcione al usuario, se le brindan los elementos para reconocer el proceso.*
7. Flexibilidad y eficiencia: *Los usuarios experimentados deben tener prácticas que le permitan navegar más rápido en la plataforma. Y a los usuarios nuevos que les permita acercarse fácilmente y adoptar esas medidas para navegar.*

8. Diseño minimalista: *Diseño simple que no permita perderse entre un montón de elementos.*

9. Ayuda a los usuarios a reponerse de los errores: *Proveer una solución.*

10. Ayuda y documentación.

17.6 Principios SOLID de la Programación Orientada a Objetos

SOLID[157] es un acrónimo acuñado por Robert C.Martin para definir los cinco principios básicos de la programación orientada a objetos:

- Single responsibility
- Open-closed
- Liskov substitution
- Interface segregation
- Dependency inversion.

Estos principios no dejan de ser una filosofía, una manera de hacer las cosas que a la larga, hará que tus proyectos sean más estables, mantenibles, robustos y escalables en el futuro. Muchas veces ni los plazos ni el presupuesto te dejan tiempo para aplicar todas las buenas prácticas que se te puedan ocurrir, pero aplicar esos principios debería ser un deber en todos tus proyectos.

> Los buenos programadores usan sus cerebros, pero unas buenas directrices nos ahorran de tener que hacerlo en cada caso
> Francis Glassborow

Aplicando estas guías o directrices eliminarás problemas en un futuro, tendrás todo mejor organizado, y evitarás refactorizar clases para adaptarlas a nuevas funcionalidades futuras.

[157] SOLID tiene bastante relación con los patrones de diseño.

S: Single Responsability Principle (SRP) El principio de responsabilidad única se basa en que cada clase o método sólo debe hacer una cosa, sencilla y concreta. Si un objeto tiene un sólo cometido, éste será más fácil de mantener.

> A Class should have only one reason to change
> Robert C. Martin

Si un módulo o una clase agrupa funcionalidad muy dispersa, como por ejemplo la típica clase de utilidades que se utiliza a modo cajón de sastre, se dice que su cohesión es baja. Por el contrario, tiene una única responsabilidad, su cohesión es alta.

Una alta cohesión es deseable para mejorar características de nuestro código como la mantenibilidad y la reusabilidad.

Se viola este principio cuando por ejemplo en un mismo método se mezcla la lógica de negocio con la lógica de presentación, para ésto es mejor dos clases distintas, cada una para manejar su responsabilidad.

Otro ejemplo muy bueno es la típica función para leer los datos de un JSON, e importarlos a tu base de datos. Si hacemos todo el proceso en una única función o método, estaremos inclumpliendo SRP. Lo ideal sería que dividas la clase en diferentes métodos:

- Un método que obtenga el JSON
- Otro que compruebe que la estructura es la que deseamos
- Al recorrer el JSON, un método que compruebe si el dato ya existe en nuestra BBDD
- Otro que sí existe actualiza, de lo contrario lo crea

De este modo, cada método tendrá una única responsabilidad, y será más fácil realizar test unitarios

O: Open / Closed Principle El segundo principio SOLID es el Open/Closed. Está estrechamente relacionado con el SRP que vimos en el punto anterior.

El principio Open/Closed dice que una clase/método debe estar abierto a extensiones pero cerrado a modificaciones.

> Software entities (classes, modules, functions, etc...) should be open for extension but closed for modification
>
> Bertrand Meyer

Probablemente represente el mayor desafío cuando te pones a desarrollar una clase o método: pensar en el mañana. ¿Te ha pasado alguna vez que te toca modificar algo que desarrollaste hace unos meses, y que al modificar el código de un método afecta a otra parte de la aplicación sin darte cuenta?

Si el día de mañana el cliente solicita un nuevo requisito para algo que ya tienes desarrollado, si tienes en cuenta este principio, el comportamiento de esa clase debería ser extendido, nunca modificado. En caso contrario la probabilidad de que te encuentres con daños colaterales es muy alta.

L: Liskov Substitution Principle Este principio recibe el nombre (más bien el apellido) de la persona que lo acuñó: Barbara Liskov.

> Subtype Requirement: Let ℓ(x) be a property provable about objects x of type T. Then ℓ(y) should be true for objects y of type S where S is a subtype of T.
>
> Barbara Liskov

Este principio trata sobre que los objetos de un desarrollo deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento del desarrollo. Dicho de otro modo: cualquier subclase debería poder ser sustituible por la clase padre.

De este modo, nuestros desarrollos mantendrán una integridad. De lo contrario, será un cochinero del que sólo tendremos constancia nosotros y luego el día de mañana lo heredará otra persona o tu yo del futuro, y lo que vendrá a continuación te suena, ¿no?

I: Interface Segregation Principle Este principio trata sobre algo parecido a SRP. Es mejor definir una serie de métodos abstractos a través de una serie de interfaces para que implementen nuestras clases.

Cada interface tendrá una única responsabilidad. Es preferible tener muchas interfaces que contengan pocos métodos que tener un interface con muchos métodos.

> Clients should not be forced to depend on methods they do not use
>
> Robert C. Martin

El objetivo no es otro que poder reutilizar estas interfaces en otras clases. Cada clase implementará las interfaces que necesite y use, ninguna más. No contendrá métodos que no utilice. De nuevo con el objetivo de aplicar sentido común y no ensuciar una clase con métodos o propiedades que no se utilizan.

D: Dependency Inversion Principle El objetivo de este principio es conseguir desacoplar las clases de nuestro desarrollo. Que una clase pueda funcionar por sí sola sin depender de otra. Es difícil, pero en todo diseño de Software al final suele existir un acoplamiento, pero hay que evitarlo en la medida de lo posible. Un sistema altamente acoplado es muy difícil de mantener.

> High-level modules should not depend on low-level modules. Both should depend on abstractions
>
> Abstractions should not depend upon details. Details should depend upon abstractions
>
> Robert C. Martin

El objetivo es conseguir que una clase interactúe con otras clases sin que las conozca directamente.

17.7 10 Principios de Diseño para POO

Empezando a programar con el paradigma de Objetos es un tema muy particular. Mucha gente cree que con el hecho de crear objetos y clases ya está aplicando el paradigma, pero en realidad no. El paradigma consiste no en un aspecto técnico, sino en un aspecto funcional, que puedas adaptarte a una forma de programar, en vez de utilizar formas de programación que cualquier lenguaje puede incluir.

Aquí les compartimos estos 10 principios de diseño para la programación orientada a objetos, esperando que les sirva para mejorar la forma en que hacen código y lo usan:

1. DRY (Don't repeat yourself) El principio de la POO es la de no repetir el código, hacer código reusable y abstracto, para que puedas usarlo de nuevo sin necesidad de volver a escribir desde cero. Si tu código es imposible de reusarlo en otra aplicación, o tienes que programar ciertas partes de nuevo porque no se puede adecuar, es un síntoma de que necesitas hacer funciones más abstractas, más genéricas para que puedan ser extendidas con requerimientos más puntuales.

2. Encapusla No hay mucha gente que entienda la importancia de la encapsulación, pero es muy simple. Aprende a encapsular el alcance de tus clases. La forma más fácil de hacerlo es poniendo privadas las variables y métodos que son exclusivos de tu clase, y poniendo públicos los métodos que pueden ser usados por otros objetos. Hay más formas de encapsular el código, pero esos dos son los básicos.

3. Principio de Diseño Abierto-Cerrado Este principio dice que las clases, métodos y funciones deben estar abiertos para extender su funcionalidad, pero cerrados para ser modificados. Esto significa que tu código debería ser modificado casi nunca pero sí debería servir como base para extenderlo, es decir, para usar herencia, para usar interfaces, para ser la base de nuevas clases, a fin de que nunca llegues a modificar tu código existente para crear nuevo, sino crear código sobre la base que ya tienes.

Esto podemos verlo como si fuera una caja negra, donde tu debes hacer que tu código sea imposible de modificar y solo pueda ser usado desde sus métodos públicos para poder ser extendido.

4. Principio de Responsabilidad Única Este principio hace referencia a que nuestras clases y métodos deben tener una sola tarea o actividad. Una clase que se encarga de operar datos no debería administrar componentes gráficos, ni un método que tiene que mandar un correo electrónico debería manejar funciones adicionales. Esto hace que tus funciones cumplan un solo propósito y hace más fácil que las puedas utilizar más adelante.

5. Inyección de Dependencias Las dependencias es cuando en una clase necesitas otra para poder trabajar. Piensa en una clase que toma como base otra para mandar un mensaje de texto. La clase original tiene una dependencia y si quisieras usar esa clase pero no quisieras mandar un mensaje

de texto hace que tengas que modificar el código para ello. La mejor forma de resolver un problema así es a través de la inyección de dependencias en donde en vez que que la clase original mande a llamar a la segunda clase, se le inyecte a través del uso de Frameworks o interfaces para mandar a llamar a algo que después tiene que ser completado, y de esa forma no dependa de otras clases.

6. Composición sobre Herencia La composición es justamente lo opuesto a la herencia, que es mandar a llamar objetos en vez de heredarlos. Esto es bueno en la medida en que a partir de una misma relación en vez de estar heredando clases y hacer una jerarquía grande, muchas veces para simplificar trabajar con objetos de una misma base en mejor hacerlo a través de la composición o creación de objetos. Para saber más sobre composición y asociación pueden leer este artículo.

7. Principio de Sustitución Liskov Este principio dice que si tienes un objeto de subtipo S que deriva de un tipo T deberías tener la posibilidad de usar el objeto de tipo T de la misma forma que uses al subtipo S. Como ejemplo, tenemos que un objeto de tipo Perro deriva de un objeto de tipo Animal, y si tienes una función que necesita un tipo Perro como parámetro, si colocas un objeto de tipo Animal debería funcionar la clase de la misma forma, sin errores.

8. Principio de Segregación de Interfaces Este principio refiere a que no incluyamos métodos que no va a utilizar el programador en una interfaz. Cuando creamos interfaces podemos colocar métodos que queremos que el programador use para extender su código, pero si metemos funciones que no va a usar o no siempre va a usar, estamos cayendo en este error de POO.

9. Programación de Interfaces no Implementaciones Tu código siempre debe estar enfocado a definir interfaces, no implementación de las mismas. Las interfaces ya nos dan una forma abstracta de poder hacer funciones genéricas que puedan ser usadas en muchos escenarios. Hay que mantener esa estrategia para crear código que puedas reusar.

10. Principios de Delegación Este último principio dice que no hay que hacer todo el código nosotros, ya hay clases a las que le podemos delegar esa tarea, como por ejemplo comparar dos objetos. Todos los objetos tienen la clase equals() por lo cual es innecesario implementar un nuevo método para esa funcionalidad. Es un ejemplo básico pero nos dice que hay que conocer los métodos de los objetos para saber qué ya hay código hecho que podemos usar y no todo a fuerza lo tenemos que implementar desde cero.

17.8 Errores en Programación

Al programar es necesario revisar nuestro código por un compilador y los errores son inherentes al proceso de programación. Los errores de programación responden a diferentes tipos y pueden clasificarse dependiendo de la fase en que se presenten. Algunos tipos de errores son más difíciles de detectar y reparar que otros, veamos entonces:

- Errores de sintaxis
- Advertencias
- Errores de enlazado
- Errores de ejecución
- Errores de diseño

Errores de sintaxis son errores en el código fuente. Pueden deberse a palabras reservadas mal escritas, expresiones erróneas o incompletas, variables que no han sido declaradas, etc. Los errores de sintaxis se detectan en la fase de compilación. El compilador, además de generar el código objeto, nos dará una lista de errores de sintaxis. De hecho nos dará sólo una cosa o la otra, ya que si hay errores no es posible generar un código objeto.

Advertencias además de errores, el compilador puede dar también advertencias (Warnings). Las advertencias son errores, pero no lo suficientemente graves como para impedir la generación del código objeto. No obstante, es importante corregir estos errores la mayoría de las veces, ya que ante un aviso el compilador tiene que tomar decisiones, y estas no tienen por qué coincidir con lo que nosotros pretendemos hacer, ya se basan en las

directivas que los creadores del compilador decidieron durante la creación del compilador. Por lo tanto en ocasiones, ignorar las advertencias puede ocasionar que nuestro programa arroje resultados inesperados o erróneos.

Errores de enlazado el programa enlazador también puede encontrar errores. Normalmente se refieren a funciones que no están definidas en ninguno de los ficheros objetos ni en las bibliotecas. Puede que hayamos olvidado incluir alguna biblioteca, o algún fichero objeto, o puede que hayamos olvidado definir alguna función o variable, o lo hayamos hecho mal.

Errores de ejecución incluso después de obtener un fichero ejecutable, es posible que se produzcan errores, durante la ejecución del código. En el caso de los errores de ejecución normalmente no obtendremos mensajes de error muy específicos o incluso puede que no obtengamos ningún error, sino que simplemente el programa terminará inesperadamente. Estos errores son más difíciles de detectar y corregir (pues se trata de la lógica como tal de nuestra aplicación). Existen herramientas auxiliares para buscar estos errores, son los llamados depuradores (Debuggers). Estos programas permiten detener la ejecución de nuestros programas, inspeccionar variables y ejecutar nuestro programa paso a paso (instrucción a instrucción). Esto resulta útil para detectar excepciones, errores sutiles, y fallos que se presentan dependiendo de circunstancias distintas. Generalmente los errores en tiempo de ejecución se dan por situaciones no consideradas en la aplicación, por ejemplo, que el usuario ingrese una letra en vez de un número y esto no se controle.

Errores de diseño finalmente los errores más difíciles de corregir y prevenir. Si nos hemos equivocado al diseñar nuestro algoritmo, no habrá ningún programa que nos pueda ayudar a corregirlos, pues es imposible que un programa pueda determinar qué es lo que tratamos de conseguir o un programa que realice aplicaciones cualquiera por nosotros. Contra estos errores sólo cabe practicar y pensar, realizar pruebas de escritorio, hacerle seguimiento y depuración a la aplicación hasta dar con el problema (una mala asignación, un valor inesperado, olvidar actualizar una variable, etc.), también es útil buscar un poco de ayuda de libros o en sitios y foros especializados.

En este resumen, se enumera algunas de las cosas que han aportado muchos programadores para todos aquellos que se inician en el arte y ciencia de

la programación (recordando que se puede pasar más tiempo encontrando y corrigiendo errores que en el desarrollo de nuevas características en el código). Se leen rápido, pero aplicarlos bien puede llevar toda la vida.

1. Tome el código y divida grandes partes de código en pequeñas funciones.
2. Si para cuando sale del trabajo no ha resuelto el problema. Apague la computadora y déjela para el día siguiente. No pienses más en el problema.
3. Principio YAGNI: "No lo necesitará" o no codifique más de lo que se le pidió. No anticipe el futuro y simplemente cree algo que funcione lo antes posible. Codifique solo las partes necesarias para resolver el problema actual.
4. No es necesario saberlo todo, ni todos los marcos existentes. Lo más significativo es tener una buena base. Conozca el lenguaje en profundidad antes de comenzar con un Framework y aprenda cosas fundamentales como los principios SOLID o cómo escribir código limpio.
5. KISS: "Keep it simple, stupid" o "keep it stupid simple" es un principio de diseño que establece que la mayoría de los sistemas funcionan mejor si se mantienen simples en lugar de complicados. Y si bien esto es lógico, a veces es difícil de lograr.
6. No lo pienses demasiado.
7. Si tiene un problema o un error durante demasiado tiempo, aléjese y vuelva más tarde. A menudo, las mejores soluciones a los problemas se me ocurren en el camino a la oficina, al baño. También es aconsejable alejarse cuando está enojado con un cliente o con un compañero de trabajo, especialmente si deseas conservar tu trabajo.
8. Aprenda a escribir pruebas útiles y aprenda a hacer TDD. TDD es un proceso de desarrollo de Software que se basa en la repetición de un ciclo de desarrollo muy corto: escriba una prueba, ejecute todas las pruebas y vea si la nueva falla, escriba algún código, ejecute pruebas, refactorice el código, repita.
9. Primero resuelva el problema y luego escriba el código. No empiece a codificar sin saber qué hacer.

10. No memorice el código, en su lugar, comprenda la lógica.

11. Si copia y pega una solución de desbordamiento de pila, asegúrese de comprenderla. Aprenda a usar Stack Overflow de una buena manera.

12. Si quieres aprender algo, practica. Haz ejemplos y haz que funcionen porque leer sobre algo no es suficiente.

13. Estudie el código de otras personas y deje que otros estudien su código de vez en cuando. La programación en pareja y las revisiones de código son una buena idea.

14. No reinvente la rueda.

15. Tu código es la mejor documentación.

16. Sepa cómo googlear cosas. Para ello, es necesario tener experiencia y leer mucho para saber qué buscar.

17. Tu código deberá ser mantenido por ti mismo en el futuro o por otros, así que escribe el código con el lector en mente, no tratando de ser la persona más inteligente. Haz que se lea como si estuvieras leyendo una historia.

18. La mejor manera de resolver un error con Google es copiarlo y pegarlo.

19. Nunca te rindas, al final, de una forma u otra lo resolverás. Hay días malos, pero pasarán.

20. Descanso, descanso y descanso. La mejor manera de resolver un problema es tener una mente tranquila.

21. Aprenda a utilizar los patrones de diseño de Software. Los patrones de diseño son soluciones a problemas comunes en el diseño de Software. Cada patrón es como un plano que puede personalizar para resolver un problema de diseño común en su código. (No reinvente la rueda)

22. Utilice herramientas de integración y automatice tanto como pueda.

23. Hacer katas de código. Un kata de código es un ejercicio de programación que ayuda a los programadores a mejorar sus habilidades mediante la práctica y la repetición.

24. Programe en una interfaz, no en una implementación. La inyección de dependencia es un requisito. Consulte los principios SOLID.

25. Refactor -Test-Refactor. La refactorización es una técnica para reestructurar un código existente, alterando y mejorando su estructura interna sin cambiar su comportamiento externo.

26. Pide ayuda cuando la necesites. No pierdas el tiempo.

27. La práctica hace la perfección.

28. Aunque a veces los comentarios pueden ayudarte, no les prestes demasiada atención. Probablemente estén desactualizados.

29. Conozca su entorno de desarrollo e invierta en uno lo suficientemente potente.

30. Reutilizar componentes.

31. Al desarrollar una aplicación Web, piense primero en los dispositivos móviles y en las restricciones de ancho de banda y energía asociadas.

32. No optimice ni refactorice antes de tiempo. Es más importante tener un producto mínimo viable lo antes posible.

33. Nunca elija una forma de atajo ineficiente para ahorrar unos minutos. ¡Cada vez que codifique, dé lo mejor!

34. Siga los estándares documentados.

35. Los usuarios no son personas técnicas. Piense en ello cuando desarrolle su interfaz de usuario.

36. Utilice siempre un sistema de control de código fuente como GitLab, Github o bitbucket y realice confirmaciones de git pequeñas y frecuentes.

37. Es mejor usar registros que depurar el código. Registre todas las partes críticas.

38. Sea consistente al codificar. Si usa un estilo, use siempre el mismo. Si trabaja con más personas, use el mismo estilo con todo el equipo.

39. No dejes de aprender, pero más que nuevos lenguajes o marcos, céntrate en los conceptos básicos del desarrollo de Software.

40. Y finalmente, paciencia y ama lo que haces.

17.8.1 Errores en Java

Ejemplo en Java:

```
class hola {
   public static void main(String[] args) {
      System.out.println("Hello world!");
   }
}
```

para compilarlo usamos *javac* en línea de comandos mediante:

```
$ javac hola.java
```

y lo ejecutamos con:

```
$ java hola
```

Errores en tiempo de compilación Los errores de compilación ocurren porque la sintaxis del lenguaje no es correcta, este tipo de errores no permiten que la aplicación se ejecute, por ejemplo:

- Olvidarnos de un punto y coma al final de una sentencia.
- No cerrar llaves en algún bloque de código, método, clase o en alguna estructura de control.
- Repetir variables con el mismo nombre aunque sean de diferente tipo.
- No distinguir bien los diferentes tipos primitivos (boolean, byte, short, int, long, float y double) y no saber elegir el más adecuado a cada problema.

- Asignar en una variable otra variable con tipo de dato diferente: es decir si tengo una variable String y su valor lo asigno en una variable de tipo int, en este caso el compilador me va dar un error de conversión de tipos y esto no va dejar que la aplicación compile, incluso un error parecido se puede dar en variables de grupos del mismo tipo, por ejemplo asignar el valor de un tipo int en un tipo short.

- Error en mayúsculas y minúsculas, Java es un lenguaje case sensitive, esto es que debemos respetar las mayúsculas y minúsculas. (miVar no es igual a mivar). Si hacemos referencia a un identificador que no está bien escrito entonces obtendremos un error de sintaxis.

- No llevar una cuenta clara y ordenada de los paréntesis (), llaves {} y comillas "" abiertos, lo cual conduce a que algunas ovejas vayan con parejas que no son las suyas o queden del todo desparejadas.

- No tener en cuenta el orden en que se realizan las operaciones en las expresiones de Java, de modo que en la práctica se obtiene un resultado distinto del que se espera.

- No poner bien las llaves en las sentencias if, while y for. Olvidarse por ejemplo de que una sentencia for(int i=0; i<n; i++); repite n veces la sentencia vacía (por la presencia del punto y coma final) en lugar de la sentencia siguiente al for.

- Usando equals contra la asignación (== versus =), en Java este error puede ser incluso un error de lógica o sintaxis. Para comparar 2 referencias para igualarlas se usa el operador == (el operador de igualdad). Para asignar el valor de la derecha a la variable de la izquierda se usa el operador = (operador de asignación). Los programadores novatos a veces escriben: if (miValor = valorEsperado) Este código intenta evaluar el valorEsperado como un valor booleano en lugar de intentar la evaluación de igualdad entre miValor y valorEsperado. Si valorEsperado es del tipo booleano, entonces el código tendrá un error de lógica y probará si el valorEsperado es verdadero o falso. Si valorEsperado no es del tipo booleano, entonces el código lanzará un error de compilación debido a que la estructura if requiere un valor booleano que sea retornado de la comparación (miValor = valorEsperado), pero en Java el operador = siempre retorna el valor de la derecha.

- No tener la idea clara de cómo funcionan los métodos, de lo que son los argumentos y de cómo se pasan, y de lo que es el valor de retorno y cómo se devuelve.
- Definir en un método una variable local con el mismo nombre que una variable miembro de su clase, y olvidarse de poner la referencia this cuando se quiere acceder a la variable miembro.
- Crear la referencia a un objeto y olvidarse de crear el objeto. Se obtiene un error cuando se intenta acceder a alguna variable miembro o método del objeto.
- No entender bien los métodos gráficos fundamentales (paint(), update() y repaint()) y no saber redefinirlos de acuerdo con el problema que se desea resolver.
- No saber encontrar en la documentación de Java lo que se necesita para resolver una determinada dificultad o problema de programación.
- No saber interpretar los mensajes de error del compilador (o no intentarlo siquiera), para obtener pistas que ayuden a su detección y corrección.
- No saber qué hacer cuando el problema compila y arranca bien, pero da errores en tiempo de ejecución o simplemente da un resultado incorrecto. "¿El debugger? Sí, me suena que era para algo de esto..."
- No tener una idea clara de lo que es un cast y de cuándo es imprescindible utilizarlo para que un programa funcione.
- No saber qué hacer cuando se desea utilizar un método de Java que lanza una excepción y nos da error al compilar por no haber capturado y gestionado dicha excepción.
- Tener un proyecto activo con una enorme confusión de ficheros y directorios, por ejemplo de modo que se utilicen y modifiquen ficheros del proyecto anterior destruyendo lo que ya funcionaba, etc., etc.
- No guardar con frecuencia los ficheros del proyecto, arriesgándose a perder en cualquier momento todo el trabajo realizado.

- No programar de una manera limpia, clara y ordenada; no introducir ningún comentario; elegir nombres de variables que no mantienen ninguna semejanza o relación con la magnitud a la que representan.
- No detenerse a pensar -preferiblemente sobre papel- el esquema general del programa a realizar. No intentar definir primero un esqueleto y luego todos los detalles de terminación. Conceder a todos los puntos exactamente la misma importancia.
- No desarrollar progresivamente, comprobando cada paso antes de pasar al siguiente. Olvidarse de que más vale un programa incompleto que lo que hace lo hace bien, que un programa que tiene todo pero que todo o casi todo lo hace mal.
- El nombre de la Clase pública no coincide con el nombre del archivo, cada archivo .java puede contener sólo una Clase pública. El nombre de esa Clase pública debe coincidir exactamente con el nombre del archivo antes de la extensión .java, respetando incluso las mayúsculas y minúsculas. Por ejemplo, una Clase pública llamada MyClass debe estar en un archivo MyClass.java y no en myclass.java. Este es un error de sintaxis.
- Una Clase no está en el directorio correcto, este error de sintaxis ocurre si el comando javac no puede encontrar un archivo .java en el directorio esperado. Si una Clase está en el paquete por defecto (no tiene la declaración del package en la primera línea), entonces esta Clase pertenece al directorio actual de trabajo o al directorio donde apunta el flag sourcepath del javac.

Errores en tiempo de ejecución Estos errores ocurren cuando la aplicación se está ejecutando, imagínate que cuando estás haciendo la presentación o probando tu aplicación en público o frente a tu profesor y de repente pummm la aplicación explota (se cuelga) por lo general a este tipo de errores se los conoce como errores de compilación, ahora, por qué ocurren estos errores?, bueno, hay muchos factores desde los más básicos como por ejemplo:

- El usuario ingresa valores diferentes a los que la aplicación recibe
- Acceder a una posición en un arreglo la cual no existe.

- Almacenar cadenas donde se debe almacenar números
- Divisiones por cero.
- Digamos que en una aplicación móvil consuma datos de un servicio web y que al momento de consumir esos datos no haya conexión a internet, esto hace que la aplicación se cuelgue.
- Olvidar que los índices en Java empiezan en 0, los índices de los arreglos de Java y las listas empiezan en 0, myArray[0], o myList.get(0). Asegurarse que su loop for no cause errores por este motivo. Si hacemos más loops de los que son posibles, entonces obtendremos el error: ArrayIndexOutBounds exception. Si hacemos menos loops de los requeridos, entonces tendremos un error de lógica.
- NullPointerException, el NullPointerException, la maldición de los programadores en Java. Los NullPointerExceptions son errores de lógica causados cuando un programa intenta acceder a métodos o atributos en una referencia que está nula. Si nuestro objeto no ha sido inicializado o ha sido establecido a null con el operador =, entonces la llamada el método o el acceso a uno de sus atributos no es válido.

Errores lógicos Estos son los más difíciles de detectar y corregir y pues bueno porque digo que son difíciles, es que con este tipo de errores la aplicación compila y se ejecuta de forma normal, pero, y entonces dónde está el error? El error se da porque la aplicación no muestra los resultados esperados digamos que ya hice el algoritmo y el resultado que debo obtener es 10, pero en la aplicación obtengo 5 o lo peor de todo que de seguro te a pasado, que escribes el código haces pruebas y los resultados son correctos subes a producción y resulta que para un caso particular no se obtuvo el resultado esperado. Pues en estos casos el problema son errores lógicos que a fin de cuentas estos ocurren por el mal diseño del algoritmo de la aplicación

Para terminar también mencionar que otro error de tipo lógico también ocurre cuando colocamos punto y coma después de una sentencia if o for y aunque parezcan errores tan obvios, se dan cuando estamos empezando a programar.

Por ello, estos son algunos consejos del buen programador:

- Tener muy cuenta las divisiones por cero, en cuanto a los arreglos no acceder a posiciones que no existen.

- Hacer uso de excepciones y validaciones cuando se tenga en cuenta que el usuario puede ingresar valores diferentes a los que se puede recibir la aplicación.
- Cuando diseñemos un algoritmo tener en cuenta todos los escenarios posibles que pueda tomar la aplicación durante su ejecución.
- Finalmente como dice el dicho divide y vencerás: una buena forma de contrarrestar los errores de tipo lógico es dividir algoritmos grandes y complejos en tareas pequeñas de forma que el código se más legible y cualquier error de tipo lógico se lo pueda depurar fácilmente.

Lista de errores comunes

- Incompatyble types: unexpected return value:

  (Tipos incompatibles: valor de retorno no esperado) El mensaje de error te indica que existen tipos incompatibles. ¿Qué hacer? Revisa el tipo de método y la instrucción return si la incluye.

  Recuerda que:

  Declaración del método void →Sentencia return no debe de existir en este tipo de método.

  Declaración del método int, float, double, char, boolean→Sentencia return debe existir, regresando un valor del tipo apropiado

- Cannot find symbol

  (No se puede encontrar el símbolo) El mensaje de error te indica que existe un símbolo que no se reconoce por que no está declarado. ¿Qué hacer? Revisa que la variable esté declarada como variable local del método, verifica que el nombre del método invocado sea el correcto, o asegúrate que el nombre de la variable global o local esté bien escrita.

- Invalid method declaration; return type required

(Declaración inválida del método; tipo de regreso requerido) El mensaje de error te indica el método no ha sido declarado apropiadamente y que se requiere un tipo. ¿Qué hacer? Revisa que el método tenga especificado un tipo de método, ya sea alguno de los tipos primitivos de datos o incluso en tipo void si no regresará valores, la declaración le falta el tipo de valor que el método regresará, ya sea int, float, char, double, boolean o void.

- Ilegal start of expression

  (Comienzo ilegal de expresión) Este mensaje de error aparece al inicio de un nuevo método, ya sea uno definido por el usuario o el mismo main. Puedes pasar horas revisando la sentencia donde el error se posiciona, y encontrarás que no hay error. Y eso es debido a que un método anterior no se ha cerrado, cualquier inicio de otro método generará error de sintaxis. ¿Qué hacer? Revisa que el método anterior tenga su llave de cierre.

- Missing return statement

  (Falta la sentencia return) Este error se genera simplemente porque se ha olvidado la sentencia return, la cual es obligatoria en los métodos de tipo int, char, float, double y boolean; además de cualquier tipo definido por el usuario. ¿Qué hacer? Asegúrate que, si el método no es de tipo void, exista la sentencia de return.

- class, interface or enum expected

  (Se esperabla class, interface o enum) El mensaje explica que se espera el inicio de una clase, interface o enumeración en esa parte del código. ¿Por qué? Muy sencillo, porque la clase actual ya ha sido cerrada. ¿Qué hacer? Revisa que en el método anterior no sobre una llave de cierre.

Esta es una lista muy pequeña de los errores más comunes que se pueden presentar cuando se trabaja con un programa modular.

17.8.2 Errores en Pyhton

Python es un lenguaje potente y flexible con muchos mecanismos y paradigmas que pueden mejorar significativamente el rendimiento. Sin embargo, como con cualquier herramienta de Software o lenguaje, con una comprensión o evaluación limitada de sus capacidades, pueden surgir problemas imprevistos durante el desarrollo.

Ejemplo en Python 2:

```
print "Hello World\n"
```

para compilarlo y ejecutarlo usamos *python2* en línea de comandos mediante:

```
$ python2 hola.py
```

Ejemplo en Python 3:

```
print ("Hello World\n")
```

para compilarlo y ejecutarlo usamos *python3* en línea de comandos mediante:

```
$ python3 hola.py
```

Errores en tiempo de compilación Los errores de compilación ocurren porque la sintaxis del lenguaje no es correcta, este tipo de errores no permiten que la aplicación se ejecute, por ejemplo:

- Python distingue entre las mayúsculas y las minúsculas ("case sensitive").

  MapSize = "34x44"

  mapsize = "22x34"

- Indentación incorrecta

Atención a la alineación del código. Esto es, a la hora de escribir funciones, etc., existen una sangrías que deben respetarse para que no nos de error. Mientras que la mayoría de los lenguajes utilizan la sangría para mejorar su legibilidad, pero no dependen de la práctica, Python ha tejido la sangría directamente en el tejido de su lenguaje. Esto significa que no puede permitirse cometer errores cuando se trata de formatear su código. Para Python, la regla es tener 4 espacios para sangrar. No mezcles con 2,3,5 o una cantidad diferente de espacios ya que Python simplemente no ejecutará tu programa. Necesitarás aprender a usar Python para separar tus bloques de código para que todo fluya bien. Es importante asegurarse de que lo haga bien la primera vez. No sangrar correctamente puede llevar a un error en su código si no tiene cuidado. Debugging ya es bastante pesado por sí solo, pero buscar un espacio que falta o un espacio extra te volverá loco. Por ejemplo en el caso de la sentencia "except" debe estar a la altura de "try" ya que funcionan como un bloque. Todas las sentencias tabuladas a la misma distancia pertenecen al mismo bloque de código hasta que no se encuentre una línea con menor tabulación

- Hay que tener cuidado a la hora de escribir las rutas para encontrar tus datos. La barra que entiende Python es / ó \\ (en esta posición \ posee otro significado para este lenguaje de programación).

- Los espacios como guión bajo. Hay ocasiones que escribimos las carpetas con espacios o recibimos datos que los incluye, es una mala costumbre al trabajar. A la hora de escribir tenemos que tenerlo muy en cuenta y poner dicho guión bajo.

- Al realizar una comparación entre dos objetos o valor, se tiene que utilizar el operador de igualdad (==), no el operador de asignación (=). El operador de asignación coloca un objeto o valor dentro de una variable y no compara nada.

- Olvidar poner los ":"al final de las sentencias tipo: try, if, elif, else, for, while, class, o def.

- Identificados con el código SyntaxError, son los que podemos apreciar repasando el código, por ejemplo al dejarnos de cerrar un paréntesis:

```
print("Hola"
```

- Errores de nombre, se producen cuando el sistema interpreta que debe ejecutar alguna función, método... pero no lo encuentra definido. Devuelven el código NameError:

  ```
  pint("Hola")
  ```

La mayoría de errores sintácticos y de nombre los identifican los editores de código antes de la ejecución, pero existen otros tipos que pasan más desapercibidos.

Errores semánticos, Estos errores son muy difíciles de identificar porque van ligados al sentido del funcionamiento y dependen de la situación. Algunas veces pueden ocurrir y otras no. La mejor forma de prevenirlos es programando mucho y aprendiendo de tus propios fallos, la experiencia es la clave. Veamos un par de ejemplos:

- Ejemplo pop() con lista vacía

  Si intentamos sacar un elemento de una lista vacía, algo que no tiene mucho sentido, el programa dará fallo de tipo IndexError. Esta situación ocurre sólo durante la ejecución del programa, por lo que los editores no lo detectarán:

  ```
  l = []
  l.pop()
  ```

  Para prevenir el error deberíamos comprobar que una lista tenga como mínimo un elemento antes de intentar sacarlo, algo factible utilizando la función len():

  ```
  l = []
  if len(l) > 0:
      l.pop()
  ```

- Ejemplo lectura de cadena y operación sin conversión a número

Cuando leemos un valor con la función input(), éste siempre se obtendrá como una cadena de caracteres. Si intentamos operarlo directamente con otros números tendremos un fallo TypeError que tampoco detectan los editores de código:

```
n = input("Introduce un número: ")
print("{}/{} = {}".format(n,m,n/m))
```

Como ya sabemos este error se puede prevenir transformando la cadena a entero o flotante:

```
n = float(input("Introduce un número: "))
m = 4
print("{}/{} = {}".format(n,m,n/m))
```

Sin embargo no siempre se puede prevenir, como cuando se introduce una cadena que no es un número:

```
n = float(input("Introduce un número: "))
m = 4
print("{}/{} = {}".format(n,m,n/m))
```

Como podemos suponer, es difícil prevenir fallos que ni siquiera nos habíamos planteado que podían existir. Por suerte para esas situaciones existen las excepciones.

17.9 Mantenimiento del Software

La idea del mantenimiento del Software suena absurda. ¿Por qué hay que mantener el Software? ¿Los bits intentan escaparse del disco para que alguien tenga que volver a colocarlos?

- El Software no cambia, pero el mundo cambia a partir de él.
- La gente descubre errores. Esto no cambia el Software sino más bien nuestro conocimiento del Software.
- A medida que las personas utilizan el Software, obtienen nuevas ideas sobre cómo quieren utilizarlo.

- El entorno humano alrededor del Software cambia. Las prioridades organizacionales cambian. Las leyes cambian. Los patrocinadores del proyecto y los usuarios se dan la vuelta.
- El entorno tecnológico del Software cambia. Los sistemas operativos, las redes y el Hardware cambian.
- Surgen nuevas posibilidades y nos hacen menos contentos con las viejas posibilidades.

La gente suele percibir estos cambios como cambios en el Software , como alguien parado en un muelle, con los ojos fijos en un barco, que siente que el muelle se mueve. Hablamos de Software como si fuera una cosa mecánica que se desgasta físicamente. Por supuesto que no lo es, pero el efecto puede ser el mismo.

Errores Superficiales versus Errores Reportados La comunidad de código abierto tiene un dicho: "Con suficientes ojos, todos los errores son superficiales". Cuando suficientes personas miran un fragmento de código, alguien encontrará y corregirá los errores.

Un principio relacionado es que con suficientes usuarios, se informarán todos los errores. Si suficientes personas usan el Software, alguien más se encontrará con el problema. Alguien lo reportará. Alguien hablará de ello en un foro en línea. Alguien escribirá un blog sobre esto y publicará una solución alternativa hasta que se solucione el error. Este principio merece más atención; No se cita con tanta frecuencia como el principio de los errores superficiales.

Lo ideal es utilizar un Software con muchos ojos y muchos usuarios. Firefox es un producto de código abierto con muchos ojos y muchos usuarios. Pero más a menudo hay que elegir ojos o usuarios. Tienes que elegir entre Software abierto pero oscuro y Software cerrado pero popular. Los proyectos de código abierto pueden tener más personas mirando el código fuente, por lo que tienen la máxima de "muchos ojos crean errores superficiales" que les funciona. Pero la base de usuarios de muchos proyectos de código abierto es pequeña en comparación con sus homólogos comerciales. La cantidad de usuarios que pueden encontrar e informar errores es pequeña, y la cantidad que documenta correcciones y soluciones alternativas es aún menor.

No estoy apegado ideológicamente al Software comercial o de código abierto. Yo uso ambos. Sólo quiero que mi Software funcione. Y cuando no funciona, quiero encontrar una solución rápidamente.

Errores, Características y Riesgos Todo el Software tiene errores. Alguien ha estimado que el código de producción tiene aproximadamente un error por cada 100 líneas. Por supuesto, hay alguna variación en este número. Algunos programas son mucho peores y otros son un poco mejores.

Pero los errores por línea de código no son muy útiles para evaluar el riesgo. El riesgo de sufrir un error es la probabilidad de toparse con él multiplicada por su impacto. Es mucho más probable que algunas líneas de código se ejecuten que otras, y algunos errores tienen muchas más consecuencias que otros.

Dedicar el mismo esfuerzo a probar todas las líneas de código sería un desperdicio. De todos modos, no encontrará todos los errores, por lo que debe concentrarse en las partes del código que tienen más probabilidades de ejecutarse y que producirían el mayor daño si estuvieran equivocadas.

Sin embargo, aquí hay una complicación. La probabilidad de encontrar un error puede cambiar con el tiempo a medida que las personas utilizan el Software de nuevas formas. Por alguna razón, la gente quiere utilizar funciones que no se habían utilizado antes, cuando lo hagan, es probable que descubran nuevos errores -esto ayuda a explicar por qué todo el mundo piensa que su Software preferido es más confiable que otros, cuando eres un usuario típico, sigues caminos bien probados, también aprendes, a menudo inconscientemente, a evitar caminos llenos de errores, cuando traes tus expectativas de un Software antiguo a uno nuevo, es más probable que descubra errores-.

Aunque los patrones de uso cambian, no cambian arbitrariamente. Sigue siendo cierto que es mucho más probable que se ejecute algún código que otro.

Los buenos desarrolladores de Software piensan en el futuro. Resuelven más de lo que se les pide que resuelvan. Piensan: "Voy a seguir adelante e incluir este otro caso mientras estoy en ello en caso de que lo necesiten más adelante". Son héroes cuando resulta que sus conjeturas sobre las necesidades futuras eran correctas.

Pero esta iniciativa tiene un inconveniente. Pagas por lo que no usas. Cada característica especulativa tiene que ser probada, incurriendo en más

gastos por adelantado, o entregarse sin ser probada, incurriendo en más riesgos. Esto sugiere que es mejor desactivar las funciones no utilizadas.

No se puede evitar la especulación por completo. Escribir Software mantenible requiere especular bien, anticipar y prepararse para el cambio. Los buenos desarrolladores de Software hacen buenas apuestas, y éstas tienden a ser apuestas pequeñas, haciendo un pequeño esfuerzo adicional para hacer que el Software sea mucho más flexible. Al igual que con los errores, hay que considerar las probabilidades y las consecuencias: ¿qué posibilidades hay de que cambie esta parte del Software y cuánto esfuerzo se necesitará para prepararse para ese cambio?.

Los desarrolladores aprenden por experiencia qué aspectos del Software es probable que cambien y se preparan para ese cambio. Pero luego se enojan con un novato que pierde mucho tiempo desarrollando alguna característica innecesaria. Es posible que no se den cuenta de que el novato está haciendo lo mismo que ellos, pero con una idea menos informada de lo que probablemente será necesario en el futuro.

Las disputas entre desarrolladores a menudo implican suposiciones ocultas sobre las probabilidades. Si algún aspecto del Software es una preparación responsable para el mantenimiento o un despilfarro de oro depende de su idea de lo que probablemente sucederá en el futuro.

El Software Peca de Omisión El Libro de Oración Común contiene la confesión

> ... hemos dejado de hacer lo que debíamos haber hecho, y hemos hecho lo que no debíamos haber hecho.

Las cosas que no se hacen se llaman pecados de omisión; las cosas que no deberían haberse hecho se llaman pecados de comisión.

En las pruebas y depuración de Software, nos centramos en los pecados de comisión, código que se implementó incorrectamente. Pero según Robert Glass, la mayoría de los errores son pecados de omisión. En Hechos fundamentales frecuentemente olvidados sobre la ingeniería de Software, Glass dice

Aproximadamente el 35 por ciento de los defectos de Software surgen de la falta de rutas lógicas y otro 40 por ciento se debe a la ejecución de una combinación única de rutas lógicas.

Si estas cifras son correctas, tres de cada cuatro errores de Software son pecados de omisión, errores debidos a cosas que se dejan sin hacer. Estos son errores debidos a contingencias que los desarrolladores no pensaron en manejar. Tres cuartas partes parece una proporción grande, pero es plausible. Sé que he escrito muchos errores que equivalían a no considerar suficientes posibilidades, particularmente en el Software de interfaz gráfica de usuario. Es difícil pensar en todo lo que un usuario podría hacer y en todas las formas en que podría llegar a un lugar en particular. (Cuando escribí aplicaciones de interfaz de usuario por primera vez, mi reacción ante un informe de error sería "¿Por qué alguien haría eso?" Si todos usaran mi Software como lo hago yo, todo estaría bien).

Importa si los errores son pecados de omisión o pecados de comisión. Los diferentes tipos de errores se detectan por diferentes medios, los desarrolladores han llegado a apreciar el valor de las pruebas unitarias últimamente, pero las pruebas unitarias detectan principalmente pecados de comisión. Si no pensó en programar algo en primer lugar, probablemente no pensará en escribir una prueba para ello. La cobertura de prueba completa solo podría encontrar el 25 % de los errores de un proyecto si se supone que el 75 % de los errores provienen de un código que nadie pensó en escribir.

La mejor manera de detectar los pecados de omisión es un par de ojos nuevos. Como dice Glass:

> Las revisiones rigurosas son más efectivas y rentables que cualquier otra estrategia de eliminación de errores, incluidas las pruebas. Pero no pueden ni deben reemplazar las pruebas.

Una forma de combinar los beneficios de las pruebas unitarias y las revisiones de código sería hacer que diferentes personas escriban las pruebas unitarias y el código de producción.

17.10 Por qué Siempre Habrá Programadores

En esta época de la Inteligencia Artificial (IA) una pregunta que surge una y otra vez es: ¿qué profesiones serán remplazadas por la IA? ... no se la respuesta, pero estoy seguro que la de programadores no será una de ellas.

Los sistemas están llenos de detalles y alguien tiene que gestionar esos detalles. Eso es lo que hacen los programadores. Quizás algún día, por ejemplo, dibujemos diagramas en lugar de escribir líneas de texto para especificar

sistemas informáticos. Bien, pero esos diagramas tendrán muchos detalles y necesitaremos profesionales que puedan gestionar esa complejidad. Llámalos programadores.

Parte de la responsabilidad de un programador es saber qué debe suceder cuando las cosas van mal, lo cual es parte de la gestión de los detalles de un sistema.

El desafío esencial de la programación es gestionar la complejidad . Aquellos que piensan que la dificultad radica principalmente en traducir ideas línea por línea a la sintaxis del lenguaje informático no han escrito programas grandes. Escribir programas pequeños es un desafío y no mucha gente puede hacerlo. Pero mucha menos gente puede escribir programas grandes.

Para escribir un programa grande, simplemente lo divides en muchos programas pequeños, ¿verdad?. Bueno, eso es cierto en cierto sentido, en el mismo sentido en que escribir un libro es simplemente una cuestión de escribir capítulos, que es simplemente una cuestión de escribir párrafos, etc. Pero escribir libros es difícil porque las piezas tienen que unirse en un todo coherente. Si una parte de un libro no encaja del todo con el todo, el resultado es estéticamente decepcionante. Si una parte de un programa no encaja del todo, se denomina bloqueo. El papel no aborta, pero los programas sí.