Software crítico para sistemas espaciales

2024/25

Master Universitario en Enxeñaría Aeroespacial

Arno Formella
Miguel Ramón González Castro
Manuel Pérez Cota

Departamento de Informática
Universidade de Vigo

24/25

interfaz

- una interfaz es una frontera
  - por la cual (por lo menos) dos entidades independientes
  - se encuentran (físicamente) and interactuan or comunican entre ellas
- interfaces pueden actuar en paralelo con requisitos adicionales especialmente si la acción posterior es crítica (de las películas: *double same time key interaction for security reasons to launch atomic bomb*)

tipos de interfaces

- interfaces hardware
  - existe una conexión física entre los componentes por la cual se transmite información
  - cable, ondas electromagnéticos, fibra, mecánica, etc.
- interfaces software
  - existe un protocolo de intercambio de datos entre los componentes por la cual se transmite información
  - paso de parámetros, llamadas a funciones de una API, protocoles UDP, TCP sobre internet, etc.
- interfaces lógicos
  - existe un algoritmo de intercambio de información entre los componentes
  - uso de *cookies*, intercambio de claves en redes seguros, etc.

tipos de interfaces

- interfaces síncronos
- interfaces asíncronos
- interfaces uni-direccionales
- interfaces bi-direccionales
- interfaces con estado
- interfaces sin estado
- interfaces con efectos secundarios
  (con cambio implícito de estado)
- interfaces sin efectos secundarios
  (sin cambio implícito de estado)

protocolos de interacción

- interfaces necesitan protocolos de interacción
- hardware: conectores compatibles, secuencia de señales físicos (de estados o pulsos etc.)
- software: secuencia de eventos (llamadas a funciones) (posiblemente con parámetros de entrada y salida)
- lógico: secuencia de acciones (procedimientos de actuación)

modelo de interconexión de componentes de OSI

el modelo OSI de ISO es un estándar que sirve como base de interconexiones entre componentes y sistemas:

7. capa de aplicación (*Application layer*), p.ej. HTTP
6. capa de presentación (*Presentation layer*), p.ej. SSL
5. capa de sesión (*Session layer*), p.ej. sockets
4. capa de transporte (*Transport layer*), p.ej. UDP
3. capa de red (*Network layer*), p.ej. IP
2. capa de enlace (*Data link layer*), p.ej. PPP
1. capa física (*Physical layer*), p.ej. byte transfer

reglas generales para el desarrollo (*rules of thumb*)

- especifica interacciones no implementaciones
- expone solamente lo necesario
- no expone detalles de una implementación
- sea preciso
- no usa *should*, *usually*, *may* etc., (parte de requisitos) sino verbos claros en voz activa
- no describe el uso de la interfaz (puede ser otro documento), ejemplos solamente describen un caso, no el comportamiento general
- especifaciones de interfaces pueden evolucionar con el riesgo adicional de tener problemas de compatibilidad

¿quién trabaja con especificaciones de interfaces?

- quien construye un elemento con tal interfaz
- quien comprueba un elemento
- quien usa tal elemento con tal interfaz
- quien analiza sistemas para nuevas construcciones
- quien construye sistemas basados en tales elementos con tales interfaces
- quien integra componentes en un sistema
- quien compone un producto completo
- quien inventa sistemas nuevos
- quien gestiona el desarrollo

la larga lista de Cs en el diseño de interfaces...

- complitud: hay que especificar exactamente lo necesario, no debe faltar nada, no debe incluir información supérflua
- compatibilidad: interfaces pueden evolucionar en el tiempo y ser adaptados a nuevas aplicaciones, puede ser interesante mantener que sigue siendo compatible en diferentes versiones, o documentar claramente que eso ya no es el caso
- comprobabilidad: las interacciones sobre las interfaces deben ser comprobables en pruebas de unidad, estas pruebas deben estar presentes en los banco de pruebas

...la larga lista de Cs en el diseño de interfaces

- coherencia: se debe mantener unas especificaciones coherentes e intuitivos en todos los interfaces, especialmente hacia un usario humano
- consistencia: se debe comprobar que la información que pasa por la interfaz en consistente en si mismo y no delegar el trato de inconsistencias a capas/fases posteriores
- complejidad: se debe controlar la complejidad tanto de la interfaz como de las acciones que están disponible en una API
- concurrencia: hay que saber que partes de un interfaz pueden estar usados concurrentemente y que pueden ser posibles efectos de tal uso

cuidado con lenguajes de programación respecto a interfaces

- clases abstractas (C++) o interfaces (Java) no son especificaciones de interfaces ya que no documentan todos los puntos mencionados antes
- solo mencionan la existencia de cierto comportamiento, es decir, las signaturas (parámetros de entrada y salida), muchas veces se limitan a la sintaxis del asunto no a la semántica de la interfaz
- muchos lenguajes de descripción (como DTD, CSS, SOAP, XML) revelan solamente la sintáctica no la semántica

- IRD de ECSS
- ICD de ECSS

recomendaciones adicionales para interfaces

- interfaces deben ser simulables para su uso en simulaciones y gemelos digitales
- la propia interfaz no debe generar nunca (en mi opinión) excepciones, sino debe hacer lo que está definido dado que es solamente una frontera sin contenido, las excepciones solo pueden estar generadas y tratadas por los actores

parámetros en interfaces

todos los parámetros que se usan en interfaces deben incluir en sus especificaciones por lo menos:

- rango, valor mínimo valor máximo, si abierto o cerrado
- valor por defecto
- granularidad, si procede (tipo de dato)
- precisión, si procede (interpretación de flotantes, p.ej. para tiempo)
- dependencias de otros parámetros (para garantizar consistencia)

interfaces al usuario humano

interfaces a un usuario humano, sobre todo para garantizar un uso seguro sin alteraciones ni estrés, deben:

- permitir retornar a algo predefinido, bien conocido, y seguro
- contener una pila de deshacer y rehacer
- permitir la carga de una configuración específica
- permitir un mecanismo *try'n'return* (prueba y retorna) (especialemente importante, por ejemplo, para la configuración de dispositivos de comunicación, resolución de pantallas, etc.)

interfaz hacia sistema de ficheros

todos usamos diariamente ficheros en un sistema de ficheros de un disposito... preguntas:

- ¿sabes cual es un nombre válido para un fichero en tu sistema?
- ¿sabes cuales son nombres válidos para su uso en sistemas diferentes (windows, linux, mac, android, etc.)?
- ¿sabes cuál es el tamaño máximo de un fichero en tu sistema?
- ¿sabes cuál es el tamaño físico de un fichero en tu sistema? (es decir, sabías que existen *sparse files*?)

software crítico también es

- software que se ejecuta en el mismo procesador que software crítico
- software que gestiona datos que son relevantes para el software crítico
- software de sistemas de verificación y validación de sistemas críticos

¿por qué el desarrollo de software crítico es complejo?

- algoritmos/procesos por implementar pueden ser complejos
- hay muchos componentes conectados con muchas interfaces entre ellos
- se tiene que adaptar a sistemas existentes y quizá viejos
- las APIs con muy grandes y mal documentados
- es un desarrollo distribuido con compleja gestión y muchos actores

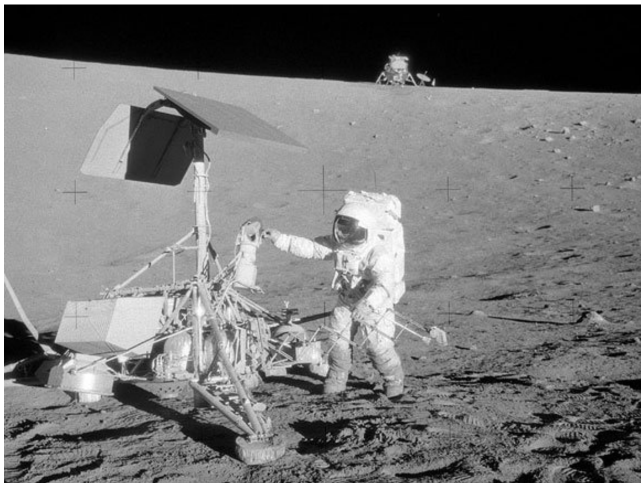
especificaciones

conversación entre Phyllis Buwalda y Richard Rumelt durante el proyecto Surveyor (alunador no-tripulado pre-Apollo, 1963):

- R.R. But, you really don't know what the moon is like. Why write a spec saying it is like the local dessert?
- P.B. This is what the smoother parts of the earth are like, so it is probably a pretty good guess as to what we'll find on the moon if we stay away from the mountains.
- R.R. But, you really have no idea what the surface of the moon is like! It could be powder, or jagged needles...
- P.B. Look, the engineers can't work without a specification. If it turns out to be a lot more difficult than this, we aren't going to be spending much time on the moon anyway.

Líderes tienen que tomar decisiones que permitan al equipo resolver problemas alcanzables.

surveyer3 on moon



lanzado 1967 (de 7 en total), visited by astronauts of Apollo 12

refactorización

¿qué es?

- refactorización mejora el diseño del software
- refactorización aumenta la legibilid y reutilizabilidad del software
- refactorización puede mejorar el rendimiento del software
- refactorización permite encontrar errores escondidos
- refactorización aumenta la velocidad de fases posteriores

refactorización

¿cuándo se debe refactorizar?

- si se nota una tercera vez que sería mejor cambiar la estructura del programa (no se hace este tercer apaño, ¡se refactoriza!)
- si para una nueva funcionalidad el código no está estructurado adecuadamente
  - primero se refactoriza
  - se realiza antes las pruebas que sigue funcionando todo como debe
    (ejecuta banco de pruebas ya existentes)
  - se incorpora la nueva funcionalidad
    (con sus pruebas pertinentes)

refactorización

adicionalmente se debe tener en consideración:

- refactorización puede ayudar que el código sea más entendible por un humano
- refactorización necesita su tiempo, pero se recupera el tiempo invertido en fases más tardes ya que se desarrolla más rápido y más robusto
- hay que refactorizar la documentación al mismo tiempo
- hay que comunicar la refactorización a otros (y retirar el código antiguo)

refactorización

- se genera un banco de pruebas automáticas adaptado
- refactorización suele/debe ser un proceso incremental en pasos pequeños que permiten detectar errores, dentro de lo que cabe, de forma sencillo
- se mantiene la refactorización en un repositorio con *commits* frecuentes
- eso permite *binary error search*

refactorización

- hay que abstenerse de realizar otros cambios al programa durante la refactorización
- accepta ideas de otros miembros del equipo para iniciar una fase de refactorización (especialmente durante procesos de revisión)
- refactorización debe ser incluido en la planificación de un proyecto

no se refactoriza

no se debe empezar un proceso de refactorización en las siguientes circunstancias:

- mientras se desarrolla nuevas funcionalidades, sino antes
- código que se usa, que esté comprobado y acordado, que no se va a modificar pronto en el futuro (mejor dejarlo en paz por el momento, ya vendrá su día...)
- en general: refactorización añade riesgos (especialmente en la planificación de *milestones*, pero también reduce riesgos,especialmente en el control de complejidad)
- a veces, incluso, es mejor empezar desde zero de nuevo

fuerzas contra refactorización

- necesidad de mantener compatibilidad
- dificultad para estimar el compromiso entre tiempo invertido para refactorización y ganancia en desarrollo en el futuro
- necesidad de mantener multiplataforma y multiherramienta (por ejemplo: código en C funcionando y su uso en aplicaciones en C++)

indicadores concretos

- existencia de código duplicado
- existencia de métodos/funciones largos (regla de 50 lines, una página)
- clases con muchos miembros y métodos
- listas largas de parámetros
- condiciones complejas
- cambios afectan multiplies puntos en el código
- necesidad de acceder a datos de otra parte de muchas maneras

indicadores concretos

- gran cantidad y tamaños de switch-sentencias (o if-else-secuencias)
- necesidad de implementar herencia en muchas clases en paralelo
- variables/miembros que se usan poco pero son críticos
- clases con interfaces públicos muy grandes
- desarrollo de subclases con muchos cambios en re-escribir métodos
- si hacen falta los comentarios para entender algo
- disponibilidad de nuevas características en el lenguaje de programación

documentación de la refactorización

la documentación sobre un proceso de refractorización debe incluir:

- el por qué se ha hecho qué cosa
- los cambios generales con ventajas y desventajas
- las nuevas interfaces, especialmente si hay más restricciones de uso
- posibles alternativas para código antiguo (*legacy code*)
- posiblemente se debe incluir *deprecated* u otro tipo de advertencia