Respuestas teóricas parcial 2C2024

Tomás Felipe Melli

May 5, 2025

Índice

1 Tema 1

1.1 Explique la paradoja de la tecnología según "The Design of Everyday Things". De un ejemplo.

Donald Norman escribe The Psycopathology of Everyday Things y en particular, en el capítulo sobre **The Design of Everyday Things**, él piensa en la velocidad con que la tecnología permite crear objetos de uso cotidiano con mayores prestaciones, con muchas funcionalidades y la necesidad de que dichas funcionalidades puedan ser utilizadas por el usuario final. Él hace referencia a un dishwasher(lavaplatos), que con una vasta cantidad de programas disponibles, su esposa siempre selecciona el mismo por la gran incertidumbre que genera seleccionar otro. Y él usa este ejemplo para denotar que la velocidad a la que se suman funcionalidades es mucho más rápida que aquella a la que se suman elementos de diseño capaces de dar facilidad de uso de las mismas. Para ello, se centra en varios criterios de diseño (visibility, affordances, causality, mental models, mapping, immediate feedback) que a su criterio, el lavarropas (también menciona el teléfono) como otros objetos, careces de estos elementos para conectar al usuario con la función del objeto. La paradoja entonces es que la tecnología permite que tengamos cada vez más objetos con muchas funciones, pero de las cuales sólo usamos una o dos por la carencia de diseños claros. Si bien existe una complejidad inherente, Norman habla de una curva de complejidad en forma de U, y lo dice por este motivo, los aparatos son difíciles de usar al principio, luego nos adaptamos y finalmente, crecen tan rápido las funcionalidades que nos quedamos atrás siempre. Concluye que para resolver esto, los dispositivos deben seguir criterios de diseño que los hagan intuitivos y así evitar que el usuario se sienta abrumado por su complejidad.

1.2 Explique a qué se refiere Dan Ingalls cuando habla de que SmallTalk fue construido alrededor de una poderosa metáfora uniforme en Design Principles Behing SmallTalk.

Dan Ingalls hace alusión a la metáfora **todo es un objeto** en SmallTalk que se cumple en absolutamente todo, el debugger, la pantalla, las clases, todo. Esto motiva a pensar en aquellos lenguajes que no siguen una metáfora uniforme, es decir, que según la circunstancia son una u otra cosa. Ejemplo Java:

```
1 int a = 5;
2 Integer b = new Integer(5);
```

Donde los tipos primitivos no son objetos. Las clases en Java no son objetos. En fin, la idea es que no exista esa coherencia filosófica que debería haber al tratarse de un **Lenguaje**. Esta idea de **uniformidad** cobra sentido cuando Ingalls va describiendo que la naturaleza de un lenguaje es conectar los modelos mentales a los de la computadora y que esa interacción sea fácil.

> *Para diseñar un lenguaje computacional eficaz, es esencial considerar tanto:*
> *- Los modelos mentales internos del usuario.*
> *- Como los medios de interacción externos del sistema.*

En esta línea, las aplicaciones complejas se construyen siguiendo el mismo modelo que las unidades más simples del sistema. Especialmente en Smalltalk, la interacción entre objetos simples (como números) es idéntica a la interacción de alto nivel entre el usuario y el sistema. Cada objeto en Smalltalk:
• Tiene un protocolo de mensajes: un conjunto de mensajes a los que puede responder.
• Tiene un contexto implicito, que incluye almacenamiento local y acceso a información compartida.

1.3 Según Brooks, cómo se pueden clasificar las dificultades que conlleva el desarrollo de software ? Explique sus diferencias.

Según Brooks en **No Silver Bullet**, el desarrollo de software tiene subclasificada la dificultad en dos. En **essence** y **accidents**. La primera tiene que ver con la *naturaleza inherente del software*, es decir, la dificultad que conlleva entender la estructura interconectada de conceptos del software como las estructuras y los algoritmos, las invocaciones a las funciones... Pero ms específicamente, Brooks dice que lo realmente complicado no es representarlos y testear esa representación, sino la especificación, el diseño y testear ese modelo conceptual. Esto último como lo más complejo. (Menciona 3 propiedades que se responderán en el tema 2).
Por otra parte, subclasifica la dificultad del desarrollo de software a lo productivo que nombra como accidental. Esto es, el tiempo que toma desarrollar software y la calidad lograda. Él indaga cómo algunas tecnologías sí mejoran la complejidad de este tipo pero no la inherente. Habla de los lenguajes de alto nivel que introducen una nueva capa de abstracción que permite no preocuparse por los detalles de bajo nivel, pero eventualmente, *the elaboration of a high-level language becomes a burden that increases,not reduces, the intellectual task of the user who rarely uses the esoteric constructs.* Menciona también cómo la unificación de los entornos de desarrollo ayudaron a que programas que trabajan juntos mitigen esa dificultad accidental al incorporar librerías integradas entre otras herramientas. Así como menciona estas tecnologías del pasado, menciona las **Hopes** que son tecnologías de su época como la **Object-Oriented Programming**, la **IA** entre otras. Con estas, va argumentando

qué dificultades accidentales resuelven o intentan resolver y concluye en cada una de ellas, que no serán capaces de imponerse como *silver bullet* a la dificultad inherente del desarrollo de software.

2 Tema 2

2.1 Según Brooks, cuáles son las propiedades inherentes a las dificultades esenciales de los sistemas modernos de software? Explique brevemente cada una de ellas.

Como hablamos en la respuesta al Tema 1, Brooks es muy claro con su tesis de que no hay *Silver Bullet* para la dificultad inherente y en particular, menciona 4 propiedades: **complexity, conformity, changeability, invisibility**.

1. La complejidad radica en la complejidad misma de la computadora dice, por el gran número de estados posibles que tiene. Esto hace que testear el software, entre otras cosas, se vuelva muy complicado. Menciona que la entidad del software como no contiene partes repetidas, al crecer en tamaño, la interacción entre sus partes es no-lineal y por tanto, la complejidad crece a un ritmo mayor. El factor humano también introduce una complejidad adicional por la ambigüedad de la comunicación. Esto es, el software crece, se complica, y comunicar esto entre miembros del equipo empeora la situación. Menciona esto como un *snow-ball effect*

   > *Many of the classical problems of developing software products derived from this essential complexity and its nonlinear increased with size. From the complexity comes the difficulty of communication among team members, which leads to product flaws, cost overruns, schedule delays. From the complexity comes the difficulty of enumerating, much less understanding, all the possible states of the program, and from that comes the unreliability. From the complexity of the functions comes the difficulty of invoking those functions, which makes programs hard to use. From complexity of structure comes the difficulty of extending programs to new functions without creating side effects. From the complexity of structure comes the unvisualized state that that constitute security trapdoors.*

2. La conformidad tiene que ver sobre qué dominio se crea software. Menciona que los físicos trabajan con problemas de naturaleza compleja pero con reglas claras, sin embargo, los desarrolladores de software, no. Esto se debe a que el dominio es un conjunto de sistemas humanos, que cambian según la época, tendencias, etc, un dominio de reglas completamente arbitrarias que obligan a adaptarse continuamente.

3. La cambiabilidad es el fenómeno de que, si cierto software funciona para cierto sistema, entonces, de alguna forma hacemos que funcione para otro distinto. Como ensartarlo en un sistema aparentemente parecido. Esto incurre en una presión al cambio, también impuesto por

   > *... a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product.*

4. La invisibilidad de lo que construimos dice *is not inherently in space.* Por ello, cuando queremos representarlo, armamos diagramas complicados de grafos, de control de flujo, de datos,.. Esta propiedad hace que no podamos diseñar dentro de nuestras mentes tan fácilmente y que luego sea muy complicado transmitirlo.

2.2 Según Naur, cuándo se considera que un programa está muerto ? Es posible revivirlo?

Peter Naur en este artículo motiva la **Theory Building View**, una perspectiva que considera el desarrollo de software como un proceso de construcción y validación de teorías sobre cómo debe comportarse un sistema. Esta idea se opone a la visión tradicional de ingeniería de software donde el desarrollo sucede en forma de cascada, con etapas claras y donde cada parte no vive en la mente del desarrollador, como es el caso de la **Theory Building View**. Dicho esto, Naur considera 3 momentos claves: **Life,Death,Revival** donde el primero muestra la vitalidad del programa dado que el conjunto de desarrolladores original posee la teoría del programa y puede modificarlo de manera inteligente por tal motivo. La etapa de muerte es cuando ya no está ese equipo en posesión de la teoría aunque el programa continúe funcionando. En esta etapa es muy difícil realizar cambios ya que los conocedores de la teoría ya no forman parte como para dar respuestas acordes a las demandas de cambios. Naur concibe otro estadío en el que se intenta revivir la teoría con un nuevo equipo, pero considera que no suele ser una buena idea ya que la teoría que construye el nuevo equipo podría contener diferencias con la original y así generar discrepancias en el texto del programa. La solución que propone, es comenzar nuevamente para evitar dichos conflictos y evitar incurrir en costos innecesarios.

2.3 Según el paper Polymorphic Hierarchy, qué es una jerarquía polimórfica y cuál es el rol de una clase plantilla (Template Class) en dicha jeraruquía?

Woolf inicia este paper comentando sobre una conclusión a la que llegó al escribir muchas funciones en las que se repite una y otra vez el mismo comportamiento. El autor concebía a las clases como unidades aisladas que permitían sí, herencia

de funcionalidades, pero concluye que la idea central es el uso del polimorfismo, es decir, como tenemos clases que tienen métodos que se comportan de la misma forma (no necesariamente todos, pero sí una **core interface**), es decir que aceptan los mismos parámteros, tienen el mismo efecto y devuelven el mismo tipo, eso se puede abstraer a una superclase o clase base. Esta idea final es la de jerarquía polimórfica. Una clase que tiene esos métodos como **superimplementators** en los cuáles se condensa ese comportamiento común.

> *When all implementors in a hierarchy have the same purpose, they're polymorphic. When all of the methods that subclasses subimplement are polymorphic with their inherite versions, the hierarchy is polymorphic.*

Luego, Woolf, menciona su idea de **Template Class** como un patrón para crear jerarquías polimórficas. Dice que se separa del patrón **Template Method** porque este último define la estructura de los métodos y delega a sus subclases los detalles. En el caso del Template Class, propone definir la estructura completa de la clase y delegar a sus subclases la implementación de sus métodos.