Evaluación de soluciones

Antes de poder programar las soluciones, es importante **asegurarse de que satisfagan adecuadamente el problema** y que lo hagan de manera **eficiente**. Esto se hace a través de la evaluación.

¿Qué es evaluación?

Una vez que se ha diseñado una solución utilizando el pensamiento computacional, es importante asegurarse de que la **solución** sea **adecuada** para su propósito.

La evaluación es el proceso que nos permite asegurarnos de que nuestra solución hace el trabajo para el que ha sido diseñada y pensar en cómo se podría mejorar.

Una vez escrito, se debe **verificar** un algoritmo para asegurarse de que:

- Se entiende fácilmente, ¿está completamente descompuesto?
- Está completa, ¿resuelve todos los aspectos del problema?
- Es eficiente, ¿resuelve el problema, haciendo el mejor uso de los recursos disponibles (p. ej., lo más rápido posible usando el menor espacio)?
- Cumple con cualquier criterio de diseño que se nos haya dado

Si un algoritmo cumple con estos cuatro criterios, es probable que funcione bien. Entonces se puede programar el algoritmo.

La falta de evaluación puede dificultar la escritura de un programa. La evaluación ayuda a **garantizar** que se nos encontraremos con la **menor cantidad posible de problemas** al programar la solución.

¿Por qué necesitamos evaluar nuestra solución?

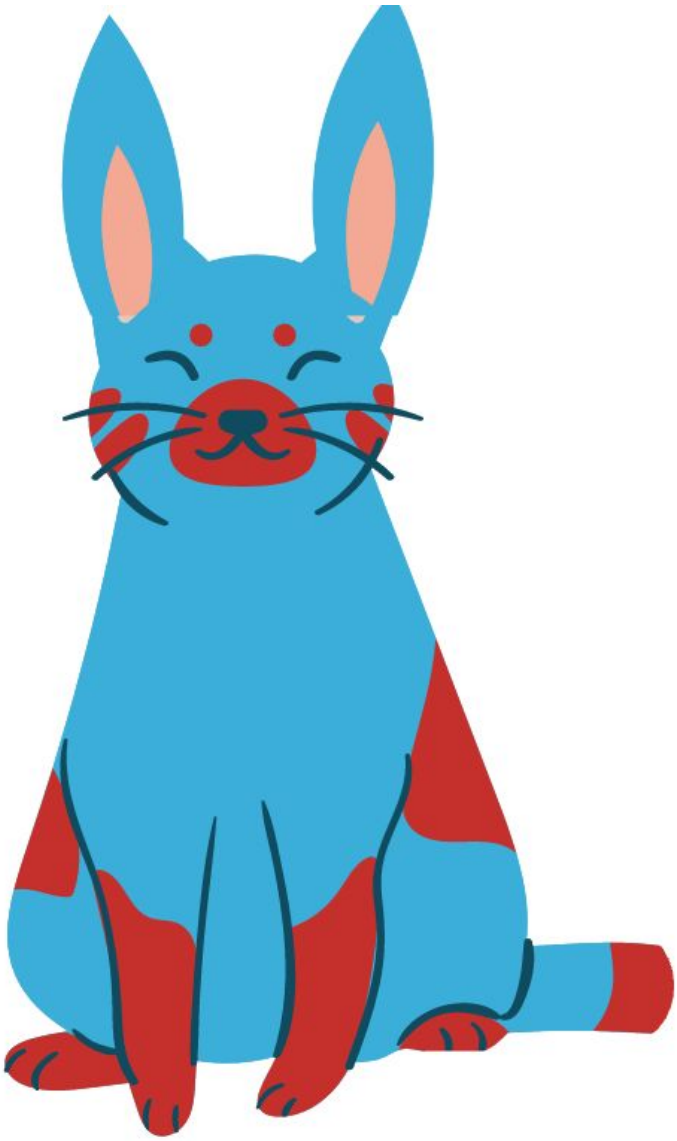
El pensamiento computacional ayuda a resolver problemas y a diseñar una solución, un algoritmo, que se puede usar para programar un ordenador. Sin embargo, **si el algoritmo falla**, puede ser difícil escribir el programa. Peor aún, es posible que el programa terminado no resuelva el problema correctamente.

La evaluación nos permite considerar la solución a un problema, asegurarnos de que cumpla con los criterios de diseño originales, produzca la solución correcta y se ajuste al propósito, **antes de que comience la ejecución.**

¿Qué ocurre si no evaluamos una solución?

Una vez que se ha decidido una solución y se ha diseñado el algoritmo, puede ser **tentador** perderse la etapa de evaluación y comenzar a la etapa de ejecución de inmediato. Sin embargo, sin la evaluación, **no se detectarán los errores** en el algoritmo, y es posible que el programa no resuelva correctamente el problema o no lo resuelva de la mejor manera.

Los fallos pueden ser menores y no muy importantes. Por ejemplo, si se creó una solución a la pregunta "¿cómo dibujar un gato?" y esta tenía errores, todo lo que estaría mal es que el gato dibujado podría no parecerse a un gato.



Sin embargo, los errores pueden tener **efectos enormes y terribles**, por ejemplo, si la solución para el piloto automático de un avión tuviera defectos.

Formas en que las soluciones pueden ser defectuosas

Podemos encontrar que las soluciones fallan **porque**:

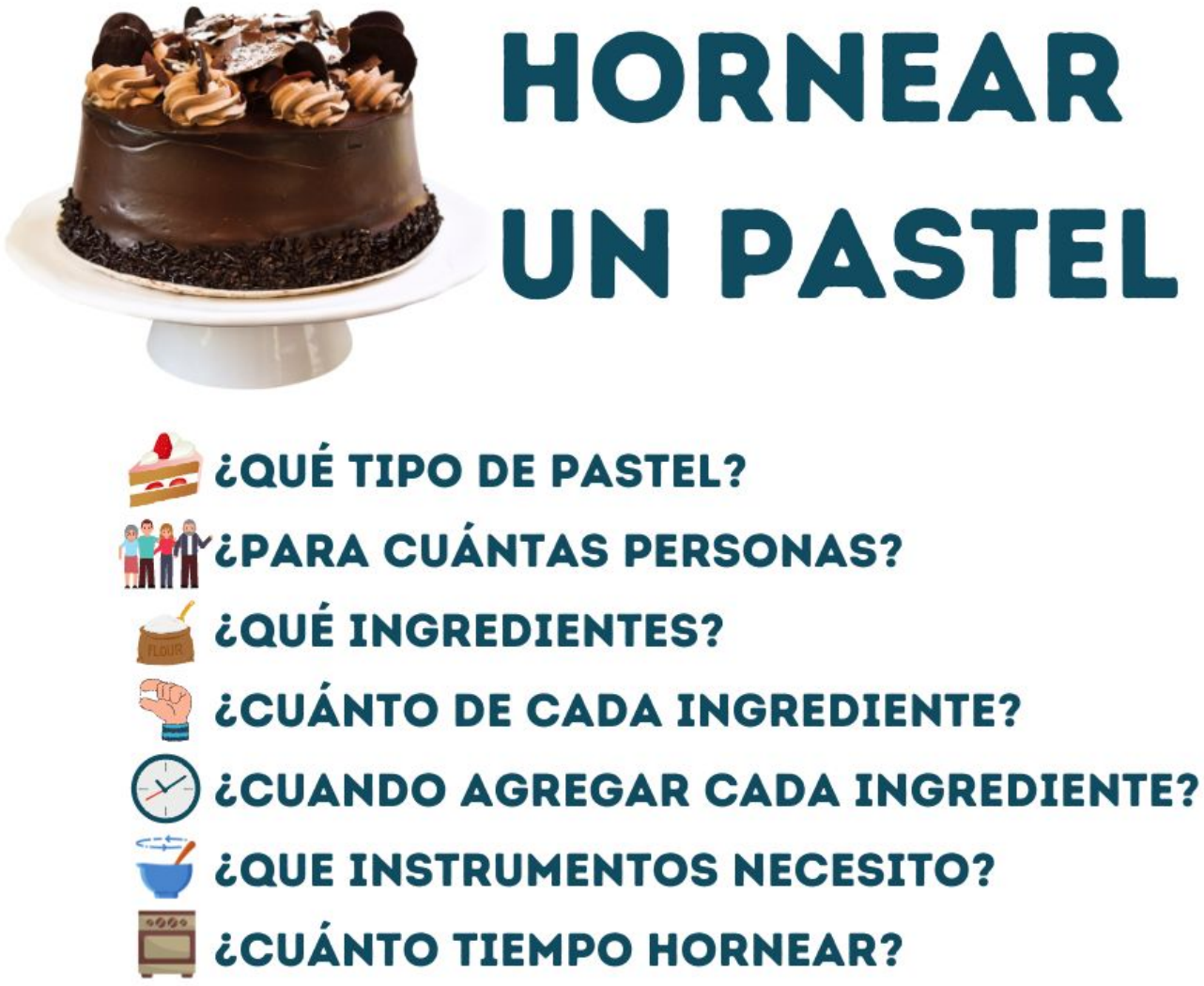
- **El problema no se entiende completamente**; es posible que no lo hayamos descompuesto correctamente
- **La solución está incompleta**; es posible que algunas partes del problema se hayan omitido accidentalmente
- **Es ineficiente**; la solución puede ser demasiado complicada o demasiado larga

- **No cumple con los criterios de diseño originales**, por lo que no es adecuada para su propósito

Una solución defectuosa puede incluir uno o más de estos errores.

Soluciones que no se descomponen correctamente

Si se aplican técnicas de pensamiento computacional al problema de cómo hornear un pastel, al descomponer el problema, es necesario saber:

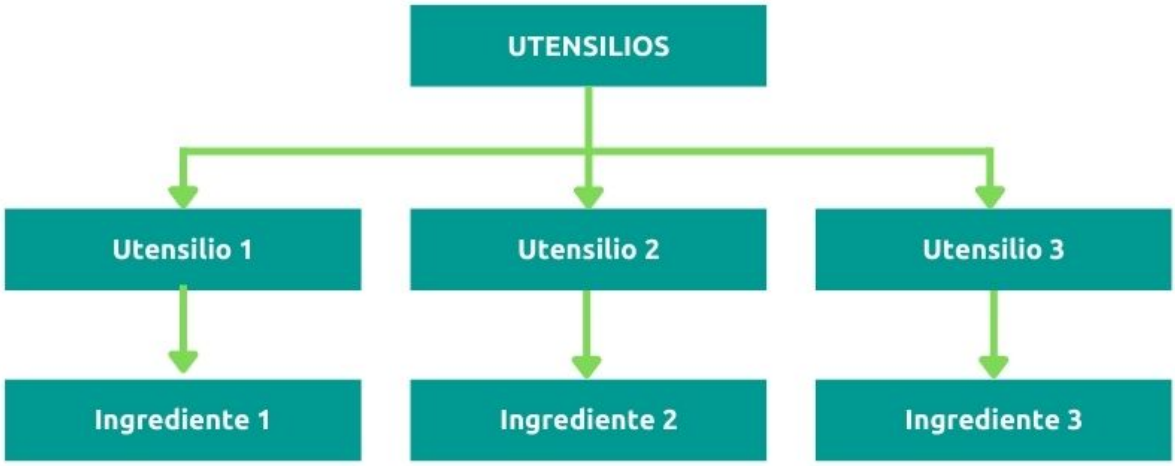


Un diagrama de una **descomposición de ingredientes** se vería así:

Por el momento, un diagrama de la descomposición adicional del equipo necesario se vería así:

La parte de 'Utensilios' **no está correctamente desglosada (o descompuesta)**. Por lo tanto, si la solución, o el algoritmo, se creara a partir de esto, hornear el pastel tendría problemas. El algoritmo diría qué utensilio se necesita, pero no cómo usarlo, por lo que una persona podría terminar tratando de usar un cuchillo para medir la harina y una batidora para cortar un trozo de mantequilla, por ejemplo. Esto sería incorrecto y, por supuesto, no funcionaría.

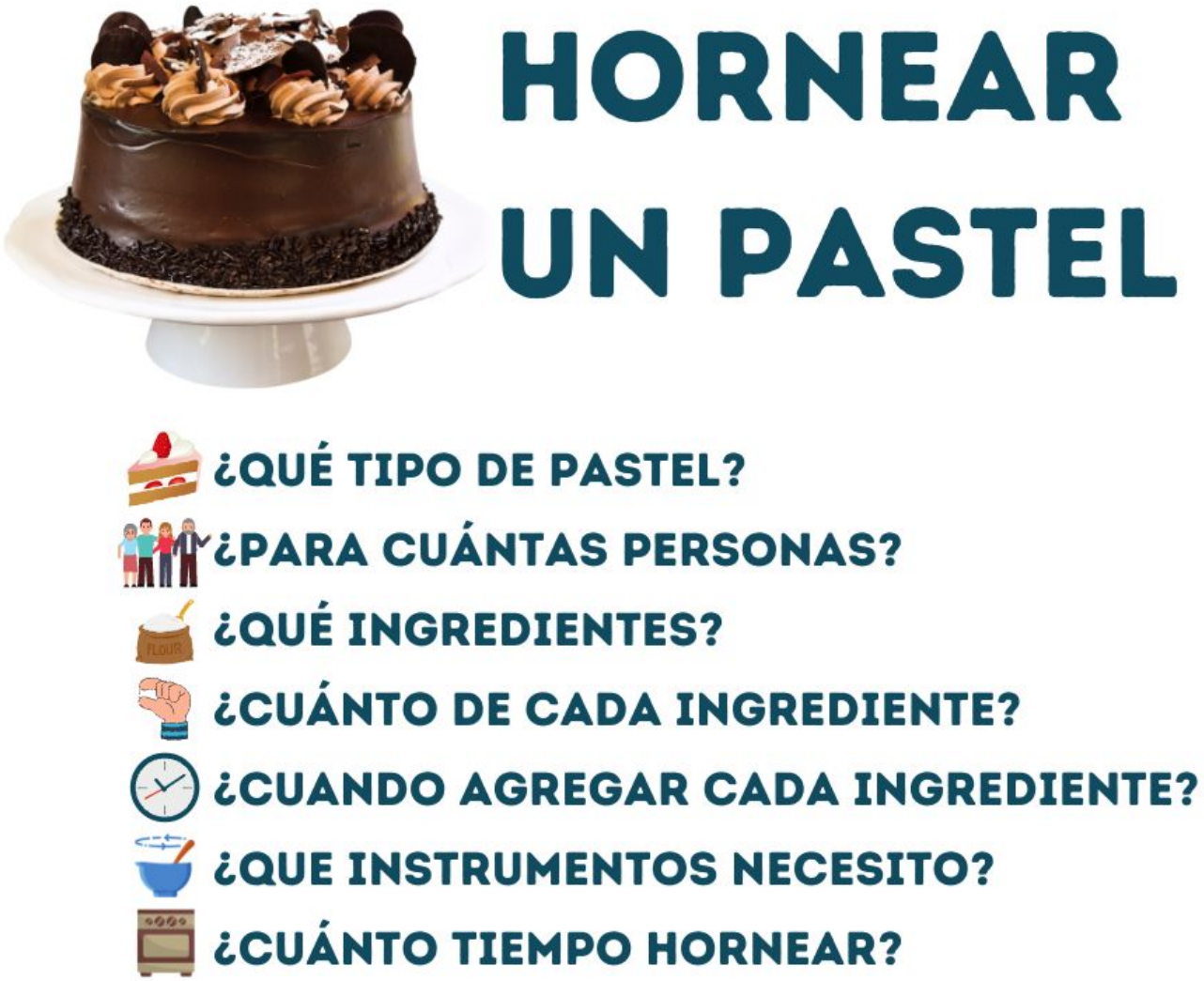
Idealmente, entonces, 'Utensilios' debería descomponerse aún más, para indicar **qué utensilios se necesitan y con qué ingredientes.**



El error se da aquí porque el problema de qué equipo usar y con qué ingredientes usarlo no se había descompuesto por completo.

Soluciones incompletas

Si se aplican técnicas de pensamiento computacional al problema de cómo hornear un pastel, al descomponer el problema, es necesario saber:



Sin embargo, esto está incompleto: se ha **omitido** parte del problema. Todavía necesitamos saber:

- Dónde hornear el pastel
- A qué temperatura hornear el pastel.

Por lo tanto, si esta información se usara para crear la solución, el algoritmo diría cuánto tiempo se debe hornear el pastel, pero no indicaría que se debe colocar el pastel en el horno o la temperatura a la que debe estar el horno. Incluso si el pastel llegó al horno, podría terminar poco cocido o quemado hasta convertirse en cenizas.

Se han omitido factores muy importantes, por lo que las posibilidades de hacer un gran pastel son escasas.

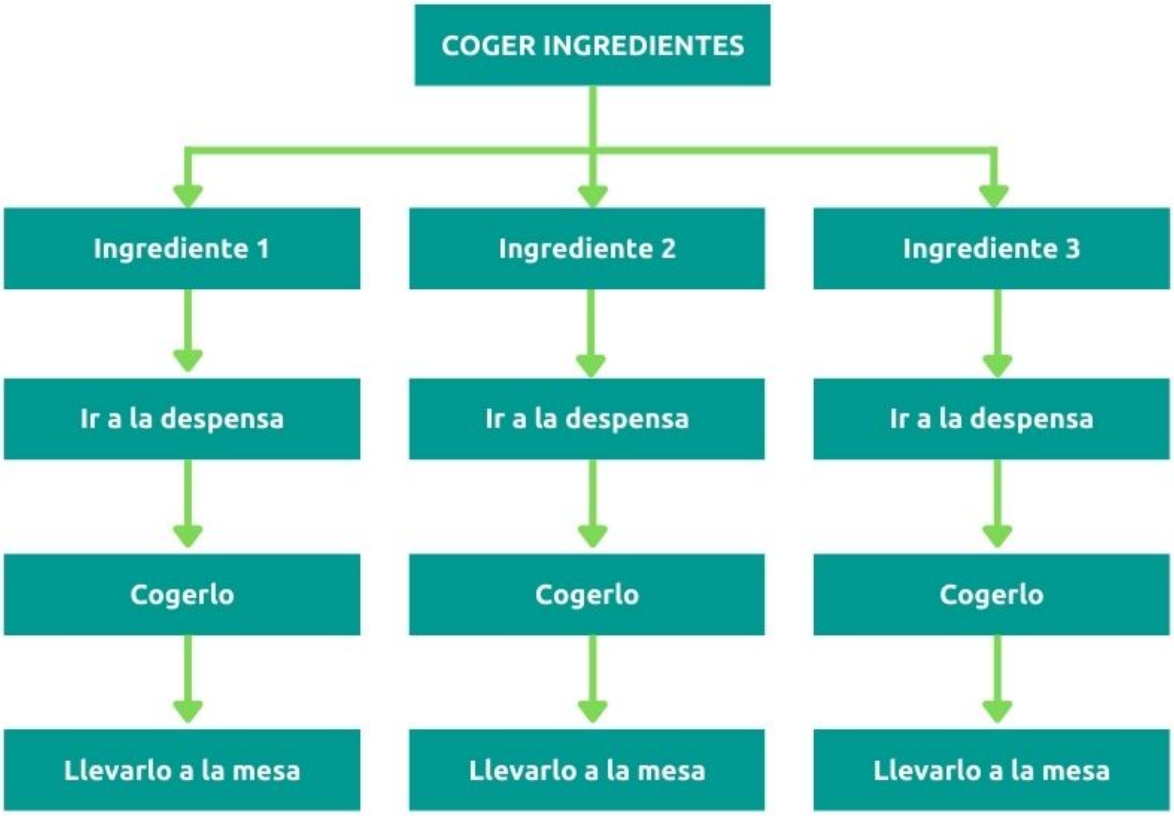
El error se dio aquí porque no se había incluido colocar el pastel en el horno y especificar la temperatura del horno, lo que hacía que la solución fuera incompleta.

Soluciones ineficientes

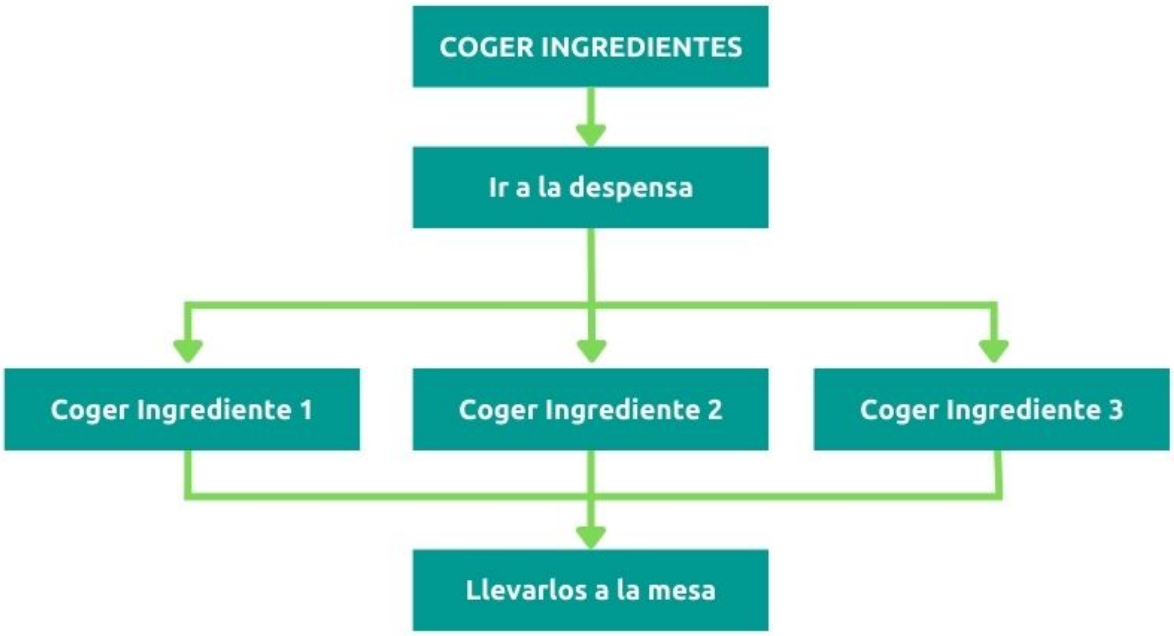
Si se aplican técnicas de pensamiento computacional al problema de cómo hornear un pastel, al descomponer el problema, la solución sería, entre otras cosas, que se necesitan **ciertas cantidades de ingredientes** particulares para hacer el pastel.

Para el primer ingrediente, podría decirnos que vayamos a la despensa, tomemos el ingrediente y lo traigamos de vuelta a la mesa. Para el segundo, y todos los demás ingredientes, podría decirnos que hagamos lo mismo.

Si el pastel tuviera tres ingredientes, eso significaría **tres viajes a la alacena**. Si bien el programa funcionaría así, sería innecesariamente largo y complicado:



Sería más eficiente obtener todos los ingredientes de una sola vez y, como resultado, el programa sería más corto:



La solución ahora es más **simple y eficiente**, y se ha reducido de nueve pasos a cinco.

El problema ocurrió aquí porque algunos pasos se repitieron innecesariamente, lo que hizo que la solución fuera ineficiente y demasiado larga.

Soluciones que no cumplen con los criterios de diseño originales

Las soluciones deben evaluarse según la **especificación original** o los **criterios de diseño** cuando sea posible. Esto asegura que la solución no se haya desviado demasiado de lo que se requería originalmente, que solucione el problema original y que sea adecuado para los usuarios.

Imagina tener que aplicar el pensamiento computacional al problema de cómo hornear un pastel. Al descomponer el problema, es necesario saber:

HORNEAR UN PASTEL

¿QUÉ TIPO DE PASTEL?

¿PARA CUÁNTAS PERSONAS?

¿QUÉ INGREDIENTES?

¿CUÁNTO DE CADA INGREDIENTE?

¿CUANDO AGREGAR CADA INGREDIENTE?

¿QUE INSTRUMENTOS NECESITO?

¿CUÁNTO TIEMPO HORNEAR?

El primer punto considera qué tipo de pastel hornear. A menudo, cuando se idean soluciones a problemas, se da una especificación para el diseño. Por ejemplo, es posible que el pastel tenga que ser un pastel de chocolate, que sigue siendo bastante general, o un pastel de dulce de chocolate con glaseado de chocolate y almendras por encima, que es más específico.

Para cumplir con los criterios de diseño, es importante asegurarse de hornear **exactamente el tipo correcto de pastel**. De lo contrario, la solución puede no ser adecuada para su propósito.

El error ocurrió aquí porque la solución no cumplía con los criterios de diseño originales, no era exactamente lo que se solicitaba.

¿Cómo evaluamos nuestra solución?

Hay varias formas de evaluar las soluciones. Para estar seguro de que la solución es correcta, es importante **preguntar**:

¿Tiene sentido la solución?

¿Entiendo ahora completamente cómo resolver el problema? Si todavía **no tengo claro** cómo hacer algo para solucionar nuestro problema, debo volver atrás y asegurarme de que todo se haya descompuesto correctamente. Una vez que sepa cómo hacer todo, nuestro problema se descompone completamente.

¿La solución cubre todas las partes del problema?

Por ejemplo, si dibujo un gato, ¿la solución describe **todo lo necesario** para dibujar un gato, no solo los ojos, la cola y el pelaje? De lo contrario, debo regresar y seguir agregando pasos a la solución hasta que esté completa.

¿La solución pide que se repitan las tareas?

Si es así, ¿hay alguna manera de reducir la repetición? He de regresar y **eliminar las repeticiones innecesarias** hasta que la solución sea eficiente.

Una vez que esté satisfecho/a con una solución, puedo pedirle a otra persona que la **revise**. Otro punto de vista suele ser bueno para detectar errores.

Es importante que tu solución siga los principios **KISS** y **DRY**.

> Los principios **KISS** (Keep It Simple, Stupid) y **DRY** (Don't Repeat Yourself) son dos conceptos importantes en el desarrollo de software y en la generación de algoritmos en general. El principio **KISS** se refiere a mantener las cosas **simples en el diseño y desarrollo**. Sugiere que es mejor optar por soluciones **simples** y **directas** en lugar de complicar innecesariamente el proceso. Por otro lado, el principio **DRY** se enfoca en evitar la repetición de tareas (o código) y de información en un sistema. En lugar de duplicar las tareas o la lógica, se promueve la creación de **abstracciones.** Aplicado a programación promueve la **reutilización** de código a través de funciones, clases o módulos. Esto reduce la duplicación, mejora la legibilidad y facilita los cambios, ya que solo se requiere modificar una única fuente de información. Ambos principios buscan fomentar la **claridad y** la **eficiencia.** Para más información sobre estos principios podéis consultar aquí.

Simulacro

Una de las mejores formas de probar una solución es realizar lo que se conoce como "simulacro". **Con lápiz y papel, trabaja en el algoritmo y traza un camino a través de él**.

Por ejemplo, en el apartado de Algoritmos, se creó un algoritmo simple para preguntarle a alguien su nombre y edad, y hacer un comentario basado en estos. Puedes probar este algoritmo usando dos edades, 15 y 75 años. Al usar 75 años, ¿adónde va el algoritmo? ¿Da la salida correcta? Si usas la edad de 15 años, ¿te lleva por un camino diferente? ¿Sigue dando la salida correcta?

Si el simulacro no da la respuesta correcta, hay algo mal que debe corregirse. Revisar la ruta a través del algoritmo ayudará a mostrar dónde está el error.

Los simulacros también se utilizan con programas completos. Los programadores usan ensayos para ayudar a encontrar errores en el código de su programa.

> **IDEA.** Puedes jugar a que se **intercambien sus soluciones para evaluarlas y detectar fallos**. Dependiendo de cuál haya sido la actividad propuesta, por ejemplo, si hemos hecho lo del pastel, pues se trataría de que otro grupo revisara la solución. Esto puede hacerse con papel y lápiz o pueden pequeñas representaciones teatrales de la solución: Voy a hacer como que hago un pastel con estas instrucciones. También puedes darles soluciones que tengan fallos y animarles a encontrarlos.

Revision #11
Created 19 June 2023 09:57:23 by Elena López de Arroyabe
Updated 26 June 2023 13:11:18 by Ana López Floría