

Introducción al diseño de TADs 1

Fernando Schapachnik¹

¹Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Buenos Aires, Argentina

8 de septiembre de 2014

(2) Hasta ahora...

- Nos preocupábamos por el *qué*, por ser *claros*.

(2) Hasta ahora...

- Nos preocupábamos por el *qué*, por ser *claros*.
- Hoy vamos a ver qué cosas hay que tener en cuenta cuando queremos pasar al *cómo*.

(2) Hasta ahora...

- Nos preocupábamos por el *qué*, por ser *claros*.
- Hoy vamos a ver qué cosas hay que tener en cuenta cuando queremos pasar al *cómo*.
- *Aprendamos de mi experiencia personal...*

(3) Yo quería...



(4) Pero me dieron...



¡¿Qué falló?!

(6) Cambio de mundos

- Hubo un cambio de mundos.

(6) Cambio de mundos

- Hubo un cambio de mundos.
- Nuevos elementos.

(6) Cambio de mundos

- Hubo un cambio de mundos.
- Nuevos elementos.
- Lo mismo pasa en el software.

(7) Más en concreto...

- Consideremos el TAD Conjunto.

(7) Más en concreto...

- Consideremos el TAD Conjunto.
- Veamos dos implementaciones posibles:

(7) Más en concreto...

- Consideremos el TAD Conjunto.
- Veamos dos implementaciones posibles:
 - Un arreglo redimensionable.
 - Una secuencia.

(7) Más en concreto...

- Consideremos el TAD Conjunto.
- Veamos dos implementaciones posibles:
 - Un arreglo redimensionable.
 - Inserción (sin repetidos): $O(n)$
 - Búsqueda: $O(\log(n))$
 - Una secuencia.
 - Inserción (sin repetidos): $O(1)$
 - Búsqueda: $O(n)$

(7) Más en concreto...

- Consideremos el TAD Conjunto.
- Veamos dos implementaciones posibles:
 - Un arreglo redimensionable.
 - Inserción (sin repetidos): $O(n)$
 - Búsqueda: $O(\log(n))$
 - Una secuencia.
 - Inserción (sin repetidos): $O(1)$
 - Búsqueda: $O(n)$
- ¿Cuál me conviene?

(7) Más en concreto...

- Consideremos el TAD Conjunto.
- Veamos dos implementaciones posibles:
 - Un arreglo redimensionable.
 - Inserción (sin repetidos): $O(n)$
 - Búsqueda: $O(\log(n))$
 - Una secuencia.
 - Inserción (sin repetidos): $O(1)$
 - Búsqueda: $O(n)$
- ¿Cuál me conviene?
- Depende de qué necesite...

Lo que está claro es que no podemos pasar de la especificación al código directamente, necesitamos una *etapa intermedia*:

La etapa de **diseño**.

(9) Diseño de tipos abstractos

Qué significa diseñar un tipo:

(9) Diseño de tipos abstractos

Qué significa diseñar un tipo:

- A nivel conceptual:

(9) Diseño de tipos abstractos

Qué significa diseñar un tipo:

- A nivel conceptual:
 - Preocuparnos no ya del *qué* sino del *cómo*.

(9) Diseño de tipos abstractos

Qué significa diseñar un tipo:

- A nivel conceptual:
 - Preocuparnos no ya del *qué* sino del *cómo*.
 - Cambiar de paradigma (del funcional al imperativo).

(9) Diseño de tipos abstractos

Qué significa diseñar un tipo:

- A nivel conceptual:
 - Preocuparnos no ya del *qué* sino del *cómo*.
 - Cambiar de paradigma (del funcional al imperativo).
 - Resolver los problemas que surgen como consecuencia de eso.

(9) Diseño de tipos abstractos

Qué significa diseñar un tipo:


- A nivel conceptual:
 - Preocuparnos no ya del *qué* sino del *cómo*.
 - Cambiar de paradigma (del funcional al imperativo).
 - Resolver los problemas que surgen como consecuencia de eso.

 En un plano un poco más concreto...

(9) Diseño de tipos abstractos

Qué significa diseñar un tipo:

- A nivel conceptual:
 - Preocuparnos no ya del *qué* sino del *cómo*.
 - Cambiar de paradigma (del funcional al imperativo).
 - Resolver los problemas que surgen como consecuencia de eso.

 En un plano un poco más concreto...

- Proveer una representación para los valores.

(9) Diseño de tipos abstractos

Qué significa diseñar un tipo:

- A nivel conceptual:
 - Preocuparnos no ya del *qué* sino del *cómo*.
 - Cambiar de paradigma (del funcional al imperativo).
 - Resolver los problemas que surgen como consecuencia de eso.




En un plano un poco más concreto...

- Proveer una representación para los valores.
- Definir las funciones del tipo.

(9) Diseño de tipos abstractos

Qué significa diseñar un tipo:

- A nivel conceptual:
 - Preocuparnos no ya del *qué* sino del *cómo*.
 - Cambiar de paradigma (del funcional al imperativo).
 - Resolver los problemas que surgen como consecuencia de eso.

 En un plano un poco más concreto...

- Proveer una representación para los valores.
- Definir las funciones del tipo.
- Demostrar que eso es correcto.

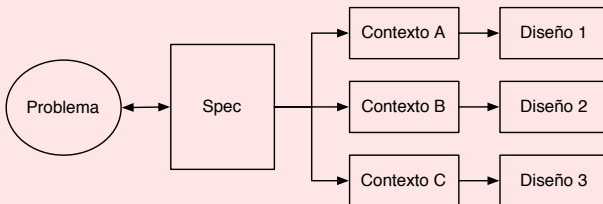
Volvamos al conjunto

¿Cómo discriminamos entre las dos soluciones?

¿Cómo discriminamos entre las dos soluciones?

Contexto de uso y
requerimientos de eficiencia. ⚠

Ejemplo



(11) Más en concreto...

- Consideremos el TAD Conjunto.
- Veamos dos implementaciones posibles:
 - Un arreglo redimensionable.
 - Inserción (sin repetidos): $O(n)$
 - Búsqueda: $O(\log(n))$
 - Una secuencia.
 - Inserción (sin repetidos): $O(1)$
 - Búsqueda: $O(n)$

Analicemos conjunto sobre secuencia:
No terminamos acá, de hecho tenemos
un nuevo “problema” por resolver.

Analicemos conjunto sobre secuencia:
No terminamos acá, de hecho tenemos
un nuevo “problema” por resolver.
¿Esto es realmente un problema?

Analicemos conjunto sobre secuencia:
No terminamos acá, de hecho tenemos
un nuevo “problema” por resolver.
¿Esto es realmente un problema?

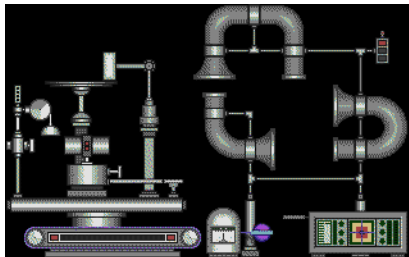
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



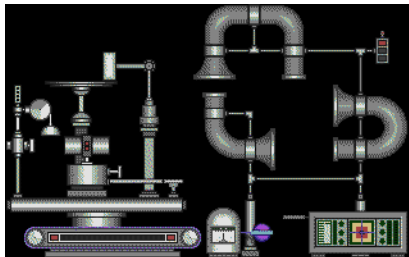
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



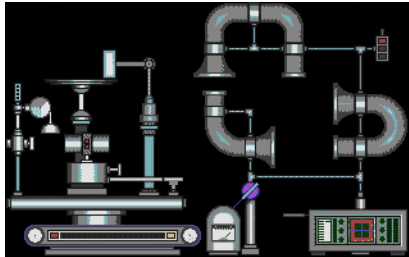
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



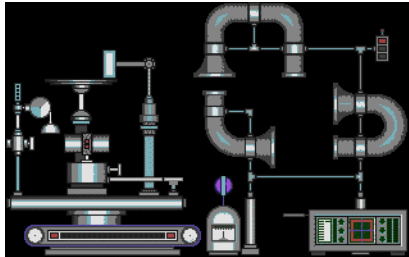
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



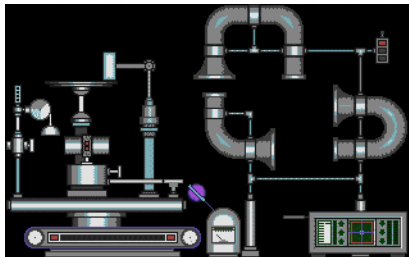
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



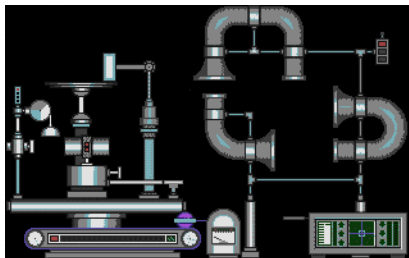
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



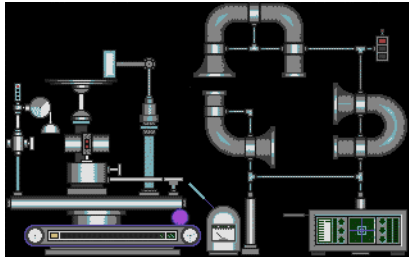
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



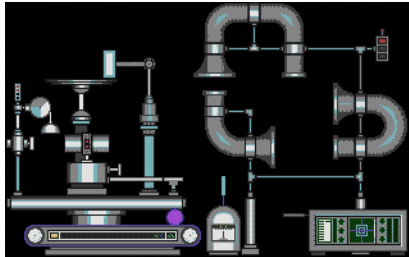
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



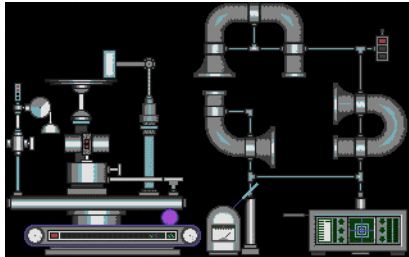
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



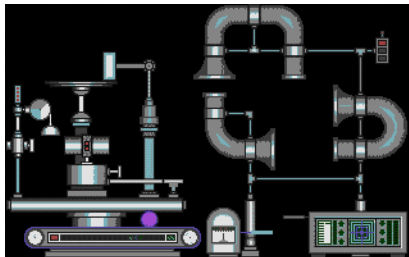
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



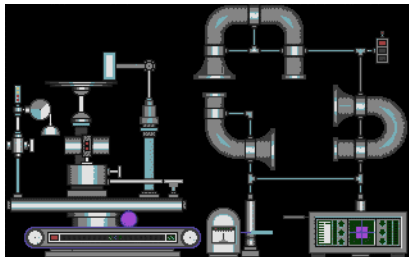
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



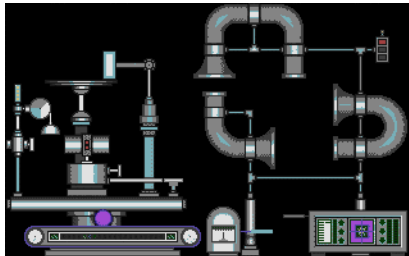
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



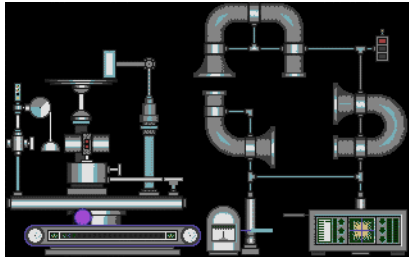
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



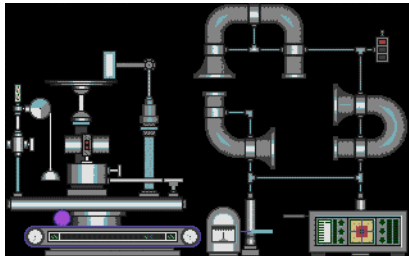
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



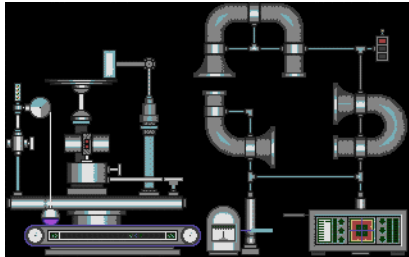
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



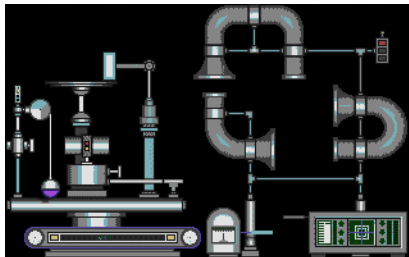
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



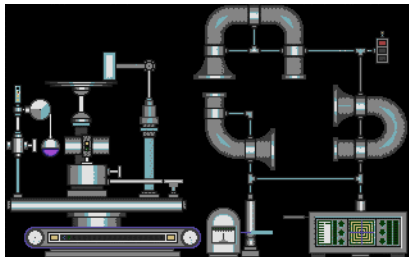
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



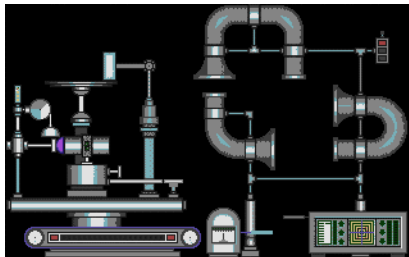
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



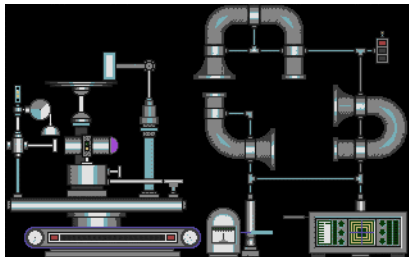
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



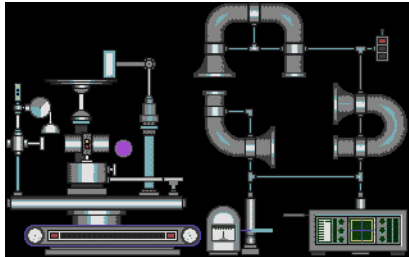
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



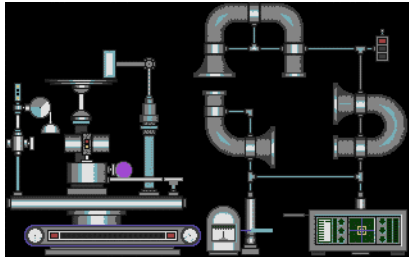
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



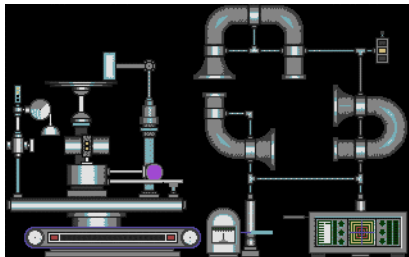
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



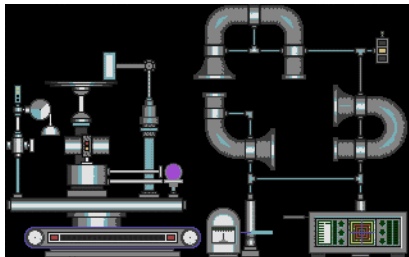
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



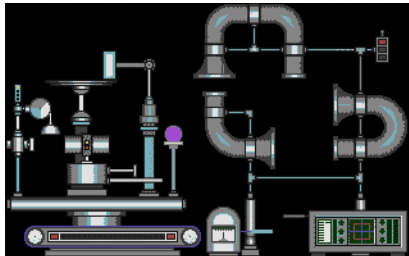
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



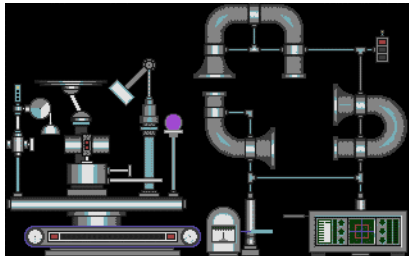
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



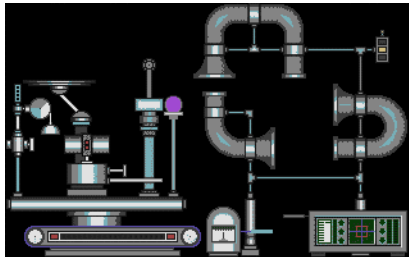
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



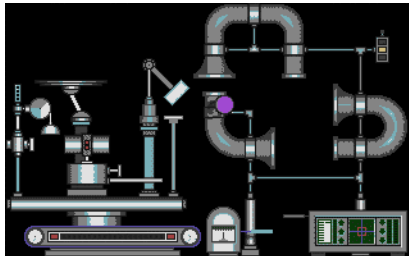
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



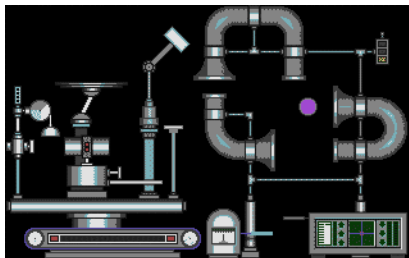
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



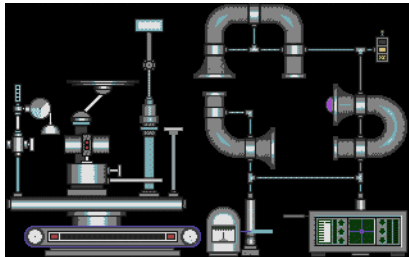
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



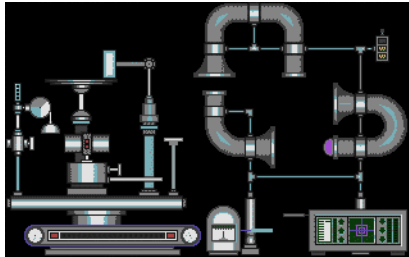
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



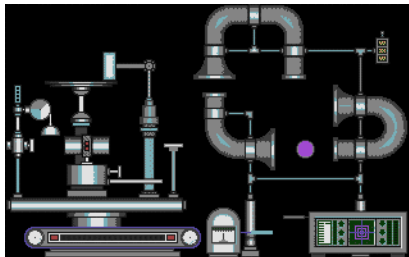
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



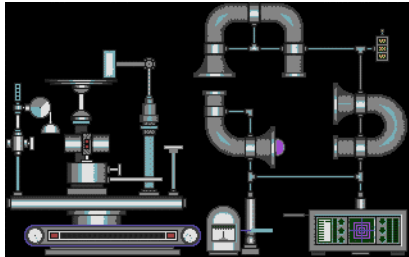
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



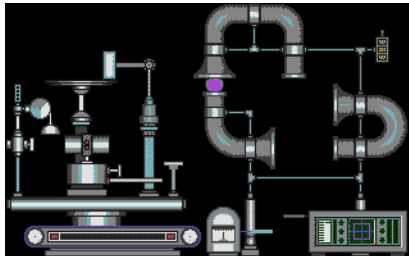
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



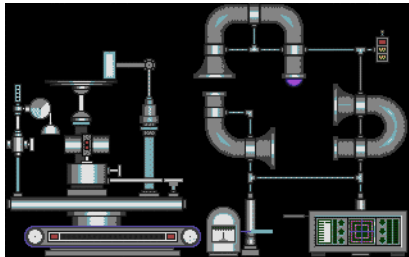
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



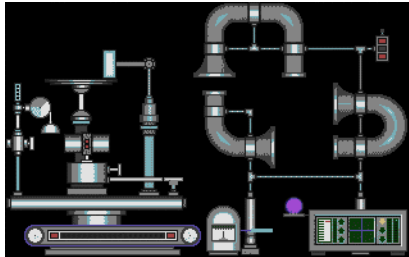
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



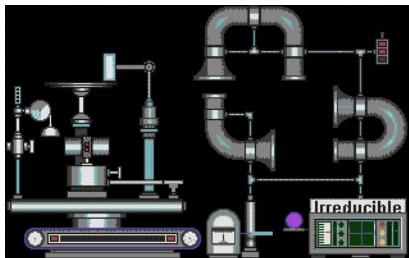
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...



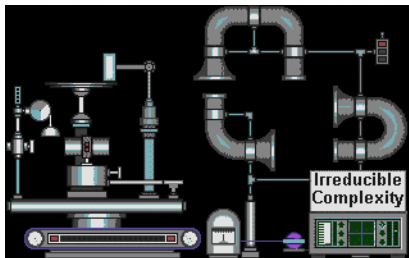
(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...

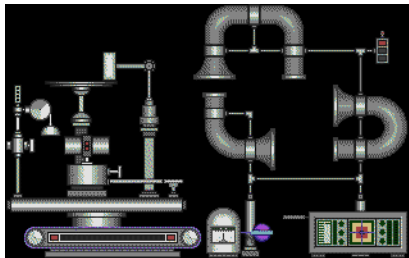


(13) Un poco de historia...

En 1960, una implementación se parecía más o menos a esto...

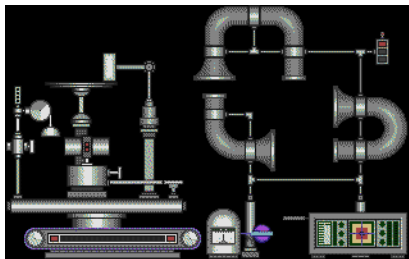


(14) Varios problemas...



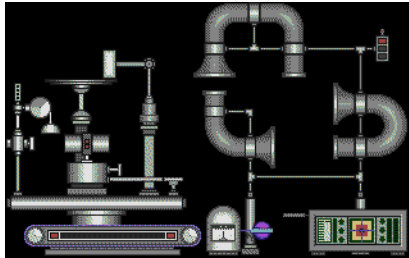
- Es complejo, difícil de entender.

(14) Varios problemas...



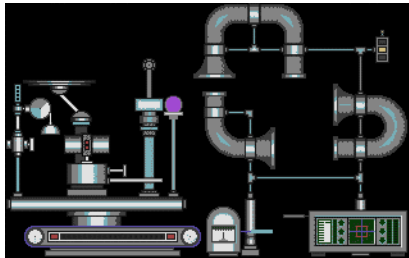
- Es complejo, difícil de entender.
- Las partes están muy interrelacionadas.

(14) Varios problemas...

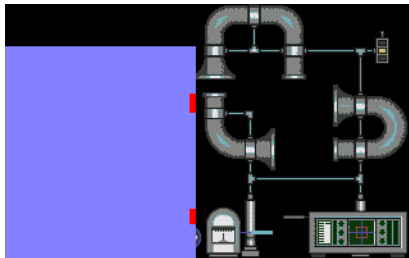


- Es complejo, difícil de entender.
- Las partes están muy interrelacionadas.
- Hay demasiada interacción entre las partes, lo que dificulta la comprensión.

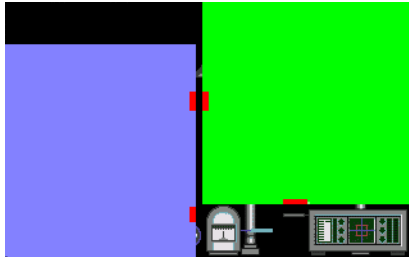
(15) Una visión alternativa...



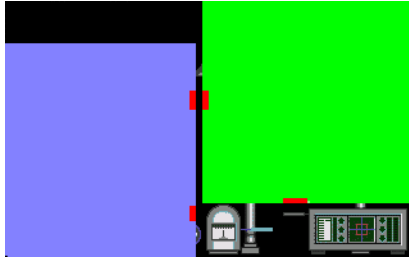
(15) Una visión alternativa...



(15) Una visión alternativa...

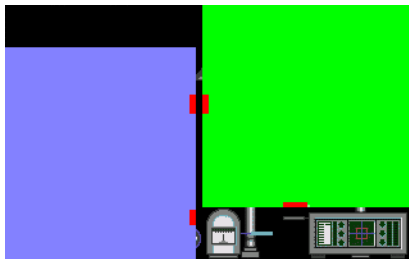


(16) Ventajas...



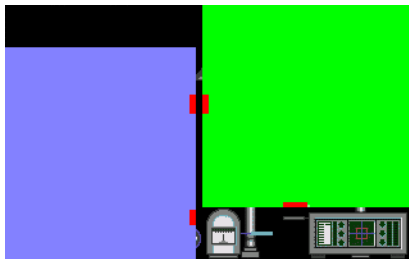
⚠ Cada fragmento presenta *interfaces claras*.

(16) Ventajas...



- ⚠ Cada fragmento presenta *interfaces claras*.
- Cada fragmento tiene una complejidad menor...

(16) Ventajas...




⚠ Cada fragmento presenta *interfaces claras*.

- Cada fragmento tiene una complejidad menor...
- ...y permite concentrarnos en *subproblemas* más fáciles de resolver.

(17) Método de diseño


 Filosofía “top-down”.


(17) Método de diseño

 Filosofía “top-down”.

 Vamos a descomponer el problema en subproblemas.

(17) Método de diseño

 Filosofía “top-down”.

 Vamos a descomponer el problema en subproblemas.

- Técnicamente hablando, en módulos, que usarán a otros módulos.

(17) Método de diseño

- ⚠ Filosofía “top-down”.
- ⚠ Vamos a descomponer el problema en subproblemas.
 - Técnicamente hablando, en módulos, que usarán a otros módulos.
- ⚠ Cada módulo dirá *qué hace y cuántos recursos necesita, pero ocultará cómo lo hace.*

(17) Método de diseño

- ⚠ Filosofía “top-down”.
- ⚠ Vamos a descomponer el problema en subproblemas.
 - Técnicamente hablando, en módulos, que usarán a otros módulos.
- ⚠ Cada módulo dirá *qué hace* y *cuántos recursos necesita*, pero ocultará *cómo lo hace*.

Interfaz de Conjunto (fragmento)

Agregar(inout C: conjunto(nat), in e: nat) $O(1)$

$\{C \equiv C_0 \wedge e \notin C\}$

$\{C \equiv \text{Agregar}(C_0, e)\}$

Pertenece(in C: conjunto(nat), in e: nat) \rightarrow res: bool $O(\#C)$

$\{true\}$

$\{res \equiv e \in C\}$

(17) Método de diseño

⚠ Filosofía “top-down”.

⚠ Vamos a descomponer el problema en subproblemas.

- Técnicamente hablando, en módulos, que usarán a otros módulos.

⚠ Cada módulo dirá *qué hace y cuántos recursos necesita*, pero ocultará *cómo lo hace*.

⚠ Con un enfoque iterativo.

(18) Interfaz

- Lo que cada módulo da a conocer al mundo sobre sí se llama interfaz.

(18) Interfaz

- Lo que cada módulo da a conocer al mundo sobre sí se llama **interfaz**.
- En ella declara a qué tipo abstracto está implementando.

(18) Interfaz

- Lo que cada módulo da a conocer al mundo sobre sí se llama **interfaz**.
- En ella declara a qué tipo abstracto está implementando.
- Y para cada operación, describe:

(18) Interfaz

- Lo que cada módulo da a conocer al mundo sobre sí se llama **interfaz**.
- En ella declara a qué tipo abstracto está implementando.
- Y para cada operación, describe:
 - Su **signatura**.

(18) Interfaz

- Lo que cada módulo da a conocer al mundo sobre sí se llama **interfaz**.
- En ella declara a qué tipo abstracto está implementando.
- Y para cada operación, describe:
 - Su **signatura**.
 - Su **pre y post condición** (más sobre esto en un rato).

(18) Interfaz

- Lo que cada módulo da a conocer al mundo sobre sí se llama **interfaz**.
- En ella declara a qué tipo abstracto está implementando.
- Y para cada operación, describe:
 - Su **signatura**.
 - Su pre y post condición (más sobre esto en un rato).
 - Su **complejidad**.

(18) Interfaz

- Lo que cada módulo da a conocer al mundo sobre sí se llama **interfaz**.
- En ella declara a qué tipo abstracto está implementando.
- Y para cada operación, describe:
 - Su **signatura**.
 - Su **pre y post condición** (más sobre esto en un rato).
 - Su **complejidad**.
 - **Otros aspectos que veremos más adelante**.

(19) ¡Momento!

- Si llamo a *Agregar*(C, x), ¿consumo $|x|$ extra bytes? ¿Y si lo borro?

(19) ¡Momento!

- Si llamo a *Agregar*(C, x), ¿consumo $|x|$ extra bytes? ¿Y si lo borro?
- Otra forma de preguntarse lo mismo: ¿el agregado es por copia o por referencia?

(19) ¡Momento!

- Si llamo a *Agregar*(C, x), ¿consumo $|x|$ extra bytes? ¿Y si lo borro?
- Otra forma de preguntarse lo mismo: ¿el agregado es por copia o por referencia?
- Respuesta: miremos la especificación...

(19) ¡Momento!

- Si llamo a *Agregar*(C, x), ¿consumo $|x|$ extra bytes? ¿Y si lo borro?
- Otra forma de preguntarse lo mismo: ¿el agregado es por copia o por referencia?
- Respuesta: miremos la especificación...
- ¿Qué sucede?

(19) ¡Momento!

- Si llamo a *Agregar*(C, x), ¿consumo $|x|$ extra bytes? ¿Y si lo borro?
- Otra forma de preguntarse lo mismo: ¿el agregado es por copia o por referencia?
- Respuesta: miremos la especificación...
- ¿Qué sucede?
- Cambio de paradigma.

(20) Paradigmas comparados

- Funcional vs. imperativo.

(20) Paradigmas comparados

- Funcional vs. imperativo.
- Abstracto vs. concreto (¿es tan así?)

(20) Paradigmas comparados

- Funcional vs. imperativo.
- Abstracto vs. concreto (¿es tan así?)
- Manchas sobre papel vs. ejecución que toma tiempo.

(20) Paradigmas comparados

- Funcional vs. imperativo.
- Abstracto vs. concreto (¿es tan así?)
- Manchas sobre papel vs. ejecución que toma tiempo.
- ⚠ Parámetros formales vs. parámetros de entrada/salida.

(20) Paradigmas comparados


- Funcional vs. imperativo.
- Abstracto vs. concreto (¿es tan así?)
- Manchas sobre papel vs. ejecución que toma tiempo.
- ⚠ Parámetros formales vs. parámetros de entrada/salida.
 - in: su valor no puede alterarse dentro de la función.

(20) Paradigmas comparados

- Funcional vs. imperativo.
- Abstracto vs. concreto (¿es tan así?)
- Manchas sobre papel vs. ejecución que toma tiempo.
- ⚠ Parámetros formales vs. parámetros de entrada/salida.
 - in: su valor no puede alterarse dentro de la función.
 - inout: su valor sí puede alterarse dentro de la función. Cuidado en la post.

(20) Paradigmas comparados

- Funcional vs. imperativo.
- Abstracto vs. concreto (¿es tan así?)
- Manchas sobre papel vs. ejecución que toma tiempo.

 Parámetros formales vs. parámetros de entrada/salida.

- in: su valor no puede alterarse dentro de la función.
- inout: su valor sí puede alterarse dentro de la función. Cuidado en la post.
- out: ídem inout, pero su valor a la entrada no importa, y podría no estar definido. Cuidado en la pre.

Paradigmas comparados (cont.)

Transparencia referencial.


Una expresión E es *referencialmente transparente* si cualquier subexpresión y su correspondiente valor (el resultado de evaluar la subexpresión) pueden ser intercambiados sin cambiar el valor de E .

Ejemplo: si $f(x) := \{\text{return } x + 1\}$, $f(4) + f(3)$ es ref. trans. Si $f(x) := \{y = G(x + 1); G ++; \text{return } y\}$, no.


Transparencia referencial (definición práctica alternativa)

Una función es *referencialmente transparente* si su resultado sólo depende de sus parámetros explícitos.


(22) Aliasing

 *Aliasing* es la forma en la que se denomina a tener más de un nombre para la misma cosa.

(22) Aliasing


 *Aliasing* es la forma en la que se denomina a tener más de un nombre para la misma cosa.

- En concreto, dos punteros o referencias hacia el mismo objeto.

 *Aliasing* es la forma en la que se denomina a tener más de un nombre para la misma cosa.

- En concreto, dos punteros o referencias hacia el mismo objeto.
- Debido a que el paradigma funcional tiene transparencia referencial, no teníamos este “problema”.

(22) Aliasing

 *Aliasing* es la forma en la que se denomina a tener más de un nombre para la misma cosa.

- En concreto, dos punteros o referencias hacia el mismo objeto.
- Debido a que el paradigma funcional tiene transparencia referencial, no teníamos este “problema”.
- ¿Es malo?

(22) Aliasing

- ⚠ *Aliasing* es la forma en la que se denomina a tener más de un nombre para la misma cosa.
- En concreto, dos punteros o referencias hacia el mismo objeto.
 - Debido a que el paradigma funcional tiene transparencia referencial, no teníamos este “problema”.
 - ¿Es malo?
- ⚠ No, pero *debe ser documentado* porque ¡es tan público como la complejidad! (Ver el apunte y la clase práctica.)

(23) Aprovechándonos del cambio de paradigma

- Ejemplo: `Desencolar() + Próximo() → Desencolar()`

(23) Aprovechándonos del cambio de paradigma

- Ejemplo: `Desencolar() + Próximo() → Desencolar()`
- `cambiosDeNombres a otros_formatos`

(23) Aprovechándonos del cambio de paradigma

- Ejemplo: `Desencolar() + Próximo() → Desencolar()`
- `cambiosDeNombres` a `otros_formatos`
- Manejo de errores.

(23) Aprovechándonos del cambio de paradigma

- Ejemplo: `Desencolar() + Próximo() → Desencolar()`
- `cambiosDeNombres` a `otros_formatos`
- Manejo de errores.
- Ejemplo: `Encolar(inout C: cola, in e: elem) → Encolar(inout C: cola, in e: elem, out s: status)`

(23) Aprovechándonos del cambio de paradigma

- Ejemplo: `Desencolar() + Próximo() → Desencolar()`
- `cambiosDeNombres` a `otros_formatos`
- Manejo de errores.
- Ejemplo: `Encolar(inout C: cola, in e: elem) → Encolar(inout C: cola, in e: elem, out s: status)`
- **Tampoco nos abusemos: ¿qué pasa en las pre y las post?**

(24) El sombrero como señal de respeto

- Analicemos: $\text{Ag}(\text{inout } C: \text{conjunto}(\text{nat}), \text{in } e: \text{nat})$
 $\{C \equiv C_0\} \{C \equiv \text{Agregar}(C_0, e)\}$

(24) El sombrero como señal de respeto

- Analicemos: $\text{Ag}(\text{inout } C: \text{conjunto}(\text{nat}), \text{in } e: \text{nat})$
 $\{C \equiv C_0\} \{C \equiv \text{Agregar}(C_0, e)\}$
- Si C y e están “hechos de bits”, ¿qué significa \equiv ahí?

(24) El sombrero como señal de respeto

- Analicemos: $\text{Ag}(\text{inout } C: \text{conjunto}(\text{nat}), \text{in } e: \text{nat})$
 $\{C \equiv C_0\} \{C \equiv \text{Agregar}(C_0, e)\}$
- Si C y e están “hechos de bits”, ¿qué significa \equiv ahí?
- Necesitamos a los términos “equivalentes” a C y e en el mundo funcional...

(24) El sombrero como señal de respeto

- Analicemos: $\text{Ag}(\text{inout } C: \text{conjunto}(\text{nat}), \text{in } e: \text{nat})$
 $\{C \equiv C_0\} \{C \equiv \text{Agregar}(C_0, e)\}$
- Si C y e están “hechos de bits”, ¿qué significa \equiv ahí?
- Necesitamos a los términos “equivalentes” a C y e en el mundo funcional...
- Chapeau: \hat{C} y \hat{e} .

(24) El sombrero como señal de respeto

- Analicemos: $\text{Ag}(\text{inout } C: \text{conjunto}(\text{nat}), \text{in } e: \text{nat})$
 $\{C \equiv C_0\} \{C \equiv \text{Agregar}(C_0, e)\}$
- Si C y e están “hechos de bits”, ¿qué significa \equiv ahí?
- Necesitamos a los términos “equivalentes” a C y e en el mundo funcional...
- Chapeau: \hat{C} y \hat{e} .

 Definición formal de $\hat{\cdot}$: lean el apunte.

(25) Dónde estamos

- Vimos

(25) Dónde estamos

- Vimos
 - La diferencia de mundos.

(25) Dónde estamos

- Vimos
 - La diferencia de mundos.
 - Cómo los requerimientos de eficiencia deciden la implementación.

(25) Dónde estamos

- Vimos
 - La diferencia de mundos.
 - Cómo los requerimientos de eficiencia deciden la implementación.
 - La idea de diseño top-down.

(25) Dónde estamos

- Vimos
 - La diferencia de mundos.
 - Cómo los requerimientos de eficiencia deciden la implementación.
 - La idea de diseño top-down.
 - El cambio de paradigma.

(25) Dónde estamos

- Vimos
 - La diferencia de mundos.
 - Cómo los requerimientos de eficiencia deciden la implementación.
 - La idea de diseño top-down.
 - El cambio de paradigma.
 - [Aliasing](#)

(25) Dónde estamos

- Vimos
 - La diferencia de mundos.
 - Cómo los requerimientos de eficiencia deciden la implementación.
 - La idea de diseño top-down.
 - El cambio de paradigma.
 - Aliasing
 - Sombrerito.

(25) Dónde estamos

- Vimos
 - La diferencia de mundos.
 - Cómo los requerimientos de eficiencia deciden la implementación.
 - La idea de diseño top-down.
 - El cambio de paradigma.
 - Aliasing
 - Sombrerito.
- Es decir, por qué es necesaria la etapa de diseño, qué cambios introduce y cuál es el lenguaje que usamos.

(25) Dónde estamos

- Vimos
 - La diferencia de mundos.
 - Cómo los requerimientos de eficiencia deciden la implementación.
 - La idea de diseño top-down.
 - El cambio de paradigma.
 - Aliasing
 - Sombrerito.
- Es decir, por qué es necesaria la etapa de diseño, qué cambios introduce y cuál es el lenguaje que usamos.
- Veremos

(25) Dónde estamos

- Vimos
 - La diferencia de mundos.
 - Cómo los requerimientos de eficiencia deciden la implementación.
 - La idea de diseño top-down.
 - El cambio de paradigma.
 - Aliasing
 - Sombrerito.
- Es decir, por qué es necesaria la etapa de diseño, qué cambios introduce y cuál es el lenguaje que usamos.
- Veremos
 - Cómo escribir un módulo.

(25) Dónde estamos

- Vimos
 - La diferencia de mundos.
 - Cómo los requerimientos de eficiencia deciden la implementación.
 - La idea de diseño top-down.
 - El cambio de paradigma.
 - Aliasing
 - Sombrerito.
- Es decir, por qué es necesaria la etapa de diseño, qué cambios introduce y cuál es el lenguaje que usamos.
- Veremos
 - Cómo escribir un módulo.
 - Qué cosas debemos considerar.

(25) Dónde estamos

- Vimos
 - La diferencia de mundos.
 - Cómo los requerimientos de eficiencia deciden la implementación.
 - La idea de diseño top-down.
 - El cambio de paradigma.
 - Aliasing
 - Sombrerito.
- Es decir, por qué es necesaria la etapa de diseño, qué cambios introduce y cuál es el lenguaje que usamos.
- Veremos
 - Cómo escribir un módulo.
 - Qué cosas debemos considerar.
 - Cómo verificar su relación con el TAD.

Introducción al diseño de TADs 2

Fernando Schapachnik¹

¹Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Buenos Aires, Argentina

8 de septiembre de 2014

(2) Dónde estamos

- Vimos
 - por qué es necesaria la etapa de diseño,
 - qué cambios introduce, y
 - cuál es el lenguaje que usamos.
- Es decir:
 - La diferencia de mundos.
 - Cómo los requerimientos de eficiencia deciden la implementación.
 - La idea de diseño top-down.
 - El cambio de paradigma.
 - Aliasing
 - Sombrerito.
- Veremos hoy:
 - Cómo escribir un módulo.
 - Qué cosas debemos considerar.
 - Cómo verificar su relación con el TAD.

(3) Ocultando información

- ¿Por qué tanto énfasis en la interfaz?
- ¿No es más fácil dar el código y listo?
- Primera piedra: *Information hiding*, David Parnas, “On the Criteria to Be Used in Decomposing Systems Into Modules” (Communications of the ACM, Diciembre de 1972).
- Tres definiciones...

(4) Ocultando información (cont.)

- **Abstracción:** “Abstraction is a process whereby we identify the important aspects of a phenomenon and ignore its details.” [Ghezzi et al, 1991]
- **Information hiding:**
 - “The [...] decomposition was made using ‘information hiding’ [...] as a criterion. [...] Every module [...] is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings.” [Parnas, 1972b]
 - “[...] the purpose of hiding is to make inaccessible certain details that should not affect other parts of a system.” [Ross et al, 1975]
- **Encapsulamiento:** “[...] A consumer has full visibility to the procedures offered by an object, and no visibility to its data. From a consumer’s point of view, and object is a seamless capsule that offers a number of services, with no visibility as to how these services are implemented [...] The technical term for this is encapsulation.” [Cox, 1986]

(5) Ocultando información (cont.)



Ventajas del ocultamiento, la abstracción y el encapsulamiento:

- La implementación se puede cambiar y mejorar sin afectar su uso.
 - Ayuda a modularizar.
 - Facilita la comprensión.
 - Favorece el reuso.
 - Los módulos son más fáciles de entender.
 - Y de programar.
 - El sistema es más resistente a los cambios.
- Aprender a elegir buenas descomposiciones no es fácil. Ese aprendizaje comienza ahora y continúa en Ingeniería del Software I.

(6) Information hiding, ¿hasta dónde?

- Dijimos que nos íbamos a basar en ocultar la información.
- ¿De quién ocultamos las cosas?
- No nos olvidemos de que si bien nos *hacemos* los misteriosos, las promesas hay que cumplirlas.
- La interfaz es una promesa.

(7) Manteniendo nuestra palabra

- Es hora de documentar nuestra estructura.
- Ejemplo: conjunto semi rápido de naturales. Los números del 1 al 100 deben manejarse en $O(1)$ porque se usan mucho. El resto, en $O(n)$. Rápidamente debo conocer la cardinalidad.
- Propuesta:
 - Un arreglo de 100 posiciones booleanas.
 - Una secuencia.
 - Un nat para la cardinalidad.
- En nuestro lenguaje de diseño, se expresa así:

conj_semi_rápido_nat **se representa con**
tupla <rápido: arreglo [0..100] de bool \times resto: secu(nat)
 \times cant: nat>



¿Y dónde aprendo el lenguaje? \rightarrow En el apunte de diseño.

(8) (¿Cómo elegir la representación adecuada?)

- ¿Se acuerdan?

 Contexto de uso y requerimientos de eficiencia.

- Todo un tema que iremos viendo en la segunda parte de la materia.
- Vamos a suponer que eso ya lo tenemos resuelto, para poder analizar otros aspectos.

(9) Estructura de representación

- Identifiquemos las partes

conj_semi_rápido_nat **src** estr
donde estr **es** tupla <rápido: arreglo [0..100] de bool ×
resto: secu(nat) × cant: nat>

- conj_semi_rápido_nat, la tupla, la secu, etc. “son de bits”.
- **src** es una abreviatura de “**se representa con**”.
- estr es una *macro* que se expande en la tupla.
- conj_semi_rápido_nat es el **género representado** y estr (su expansión) es el **genero de representación**.

(10) ¿Cualquier instancia es válida?

- $i < [0 \dots 0], < >, 8254 >$ es un conj_semi_rápido_nat válido?



¿Para qué nos serviría poder separar con facilidad instancias válidas e inválidas?

- Como una forma de documentar la estructura.
 - Como condición necesaria para establecer una relación con la abstracción (ver más adelante).
 - Para agregar a las postcondiciones, como una forma de garantizar que nuestros algoritmos no rompen la estructura.
 - Para agregar a las precondiciones, como una “garantía” con la que cuentan nuestros algoritmos.
 - Como una guía a la hora de escribir los algoritmos.
 - Si pudiésemos programar el chequeo, como una forma de detectar instancias corruptas.
- El invariante de representación.

(11) Invariante de representación

⚠ Es una función booleana con dominio en el género de representación que da *true* cuando recibe una instancia válida.

- ¿Podría el dominio ser el género representado? ¿Por qué?
- En realidad, si nos ponemos finos, el dominio es la *versión abstracta* del género de representación.

Ejemplo

Si representamos T_1 sobre T_2

Rep: $\widehat{T_2} \rightarrow \text{bool}$

$(\forall t : \widehat{T_2}) \text{Rep}(t) \equiv \dots \text{condiciones que garanticen que } t \text{ representa una instancia válida de } T_1 \dots$

- (¿Siempre existe $\widehat{T_2}$?)

(12) Invariante de representación (cont.)

- Recordemos nuestro ejemplo:

conj_semi_rápido_nat **src** estr

donde estr **es** tupla \langle rápido: arreglo $[0..100]$ de bool \times
resto: secu(nat) \times cant: nat \rangle

- ¿Qué debería decir el invariante?
 - 1 Que *resto* sólo tiene números mayores a 100, si tiene alguno.
 - 2 Que *resto* no tiene números repetidos.
 - 3 Que *cant* tiene la longitud de *resto* más la cantidad de celdas de *rápido* que están en *true*.
- Rep: $\widehat{estr} \rightarrow \text{bool}$
- $(\forall e : \widehat{estr}) \text{Rep}(e) \equiv$
 - 1 $(\forall n : \text{nat}) (\text{esta?}(n, e.\text{resto}) \Rightarrow n > 100) \wedge$
 - 2 $(\forall n : \text{nat}) (\text{cant_apariciones}(n, e.\text{resto}) \leq 1) \wedge$
 - 3 $e.\text{cant} = \text{long}(e.\text{resto}) + \text{contar_trues}(e.\text{rápido})$

⚠ ¡El invariante cambia la vida!

(14) ¿Cómo se “lee” nuestra estructura?

- ¿Cómo hay que entender a nuestra estr para pensarla como un conj_semi_rápido_nat?
- Para responder a esa pregunta vamos a definir una **función de abstracción**:
- Abs: $\widehat{T}_2 \text{ } e \rightarrow \widehat{T}_1 \text{ } (\text{Rep}(e))$
- Notar la restricción.
- Toma una instancia (abstracta) de la estructura de representación y devuelve una instancia (también abstracta) del genero representado.
- ¿Por qué toma géneros abstractos? Porque en el mundo abstracto es donde (mejor) sabemos razonar sobre las cosas.

En nuestro ejemplo

Abs: $\widehat{estr} \text{ } e \rightarrow \widehat{conj_semi_rapido_nat} \text{ } (\text{Rep}(e))$

$\text{Abs}(e) \equiv c \text{ /}$

$(\forall n: \text{nat}) (n \in c \iff ((n < 100 \wedge e.\text{rápido}[n]) \vee (n \geq 100 \wedge \text{está?}(n, e.\text{resto}))))$

(15) Algunas notas sobre Abs.

- Hay otra forma de escribir Abs (sobre los generadores del tipo de representación en lugar de sobre los observadores del tipo representado), pero eso lo van a ver en la práctica.
- ¿Es total o parcial? Una vez restringida ($\text{Rep}(e)$), deber ser total.
- ¿Debe ser sobreyectiva? Sí.
- ¿Debe ser inyectiva? No, pero puede serlo.
- Debe ser un homomorfismo respecto de la signatura del TAD:
 - Para toda operación o $\text{Abs}(o(p_1, \dots, p_n)) \equiv o(\text{Abs}(p_1), \dots, \text{Abs}(p_n))$
- Abs y Rep se las debemos a [Hoare, 1972].

(16) Un poco más de invariante

- Analicemos $Ag()$:

Interfaz

$Ag(\text{inout } C: \text{conj_semi_rápido_nat}, \text{in } e: \text{nat})$
 $\{\hat{C} \equiv C_0 \wedge \hat{e} \notin C_0\}$
 $\{\hat{C} \equiv Agregar(C_0, \hat{e})\}$

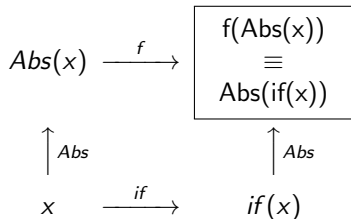
Implementación

$iAg(\text{inout } C: \text{estr}, \text{in } e: \text{nat})$
 $\{\text{Rep}(\hat{C}) \wedge_L Abs(\hat{C}) \equiv C_0 \wedge \hat{e} \notin C_0\}$
 $C.\text{cant}++$
if $(e < 100)$ then
 $C.\text{rápido}[e] := \text{true}$
else
 $ag_en_secu(C, e)$
fi
 $\{\text{Rep}(\hat{C}) \wedge_L Abs(\hat{C}) \equiv Agregar(C_0, \hat{e})\}$

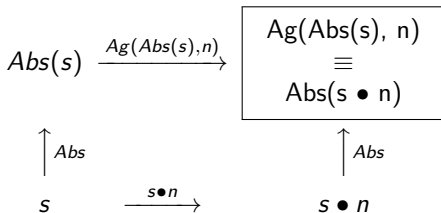
$ag_en_secu(\text{inout } E: \text{estr},$
 $\text{in } e: \text{nat})$
 $\{\hat{E} \equiv E_0 \wedge \hat{e} > 100\}$
// **Notar:** acá no vale $Rep(\hat{E})$
 $InsertarAlFinal(E.\text{resto}, e)$
 $\{\hat{E}.\text{resto} \equiv E_0.\text{resto} \bullet \hat{e}\}$

(17) Probando corrección

Para toda operación f que implementa una operación del TAD y toda x instancia del género de representación, debemos ver que el siguiente diagrama conmuta:



Ejemplo para conjunto sobre secuencia ($\forall s : secu, \forall n : nat$)



(18) Repaso

Vimos...

- La diferencia de mundos.
- Cómo los requerimientos de eficiencia deciden la implementación.
- La idea de diseño top-down.
- El cambio de paradigma.
 - Aliasing
 - Sombrerito.
- Encapsulamiento.
- Abstracción.
- Ocultamiento de información.
- Relación entre el tipo representado y el de representación.
 - Invariante.
 - Función de abstracción.
- “Eso de la *i* que apareció por ahí”.

(19) En las próximas clases

- Elección de estructuras.
- Cómo se propagan los contextos de uso y requerimientos de eficiencia.
- Cómo se escribe el código.
- Documentación.
- Ejemplos más complicados.

(20) Bibliografía

- “Abstraction, Encapsulation, and Information Hiding”. By Edward V. Berard. The Object Agency.
<http://www.itmweb.com/essay550.htm>
- [Parnas, 1972b] D.L. Parnas, “On the Criteria To Be Used in Decomposing Systems Into Modules,” Communications of the ACM, Vol. 5, No. 12, December 1972, pp. 1053-1058.
- [Ghezzi et al, 1991] C. Ghezzi, M. Jazayeri, and D. Mandrioli, Fundamentals of Software Engineering, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
- [Ross et al, 1975] D.T. Ross, J.B. Goodenough, and C.A. Irvine, “Software Engineering: Process, Principles, and Goals,” IEEE Computer, Vol. 8, No. 5, May 1975, pp. 17 - 27.
- [Cox, 1986] B.J. Cox, Object Oriented Programming: An Evolutionary Approach, Addison-Wesley, Reading, Massachusetts, 1986.
- [Hoare, 1972] C.A.R. Hoare. "Proof of correctness of Data Representation". Acta Informatica 1(1), 1972.