



Universidad Nacional de Rosario

FACULTAD DE CIENCIAS EXACTAS, INGENIERÍA Y AGRIMENSURA

# RESÚMEN I

*Ingeniería de Software I*

Autor:  
Arroyo, Joaquín

12 de mayo de 2023

# 1. Introducción

## 1.1. ¿Que es la Ingeniería de Software?

Enfoque sistemático, disciplinado y cuantificable del desarrollo, operación y mantenimiento de software. NATO, 1968

La construcción de múltiples versiones de un software llevada a cabo por múltiples personas. Parnas, 1978

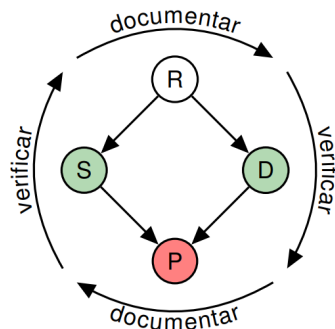
Construcción de software de una envergadura o complejidad tales que debe ser construido por equipos de ingenieros. Ghezzi, 1991

En resumen, la Ingeniería de Software es o debería ser:

- Desarrollo de software de dimensión industrial
- Desarrollo sistemático, disciplinado y cuantificable
- Desarrollo de productos que tienen una vida muy larga
- Desarrollo en equipo
- Diseños estándar
- Producir software garantizado

## 1.2. Las cuatro descripciones fundamentales

- Requerimientos del usuario (R) (Única descripción informal)
- Diseño de la estructura del programa (D)
- Especificación funcional del programa (S)
- Programa (P)



## 1.3. ¿Que debe saber un Ingeniero de Software?

- Dominar a fondo las técnicas de descripción. Esencialmente debe dominar los lenguajes formales
- Entender qué hace que una descripción particular sirva o no para un propósito determinado
- Moverse en distintos niveles de abstracción
- Describir modelos mediante lenguajes formales
- Verificar propiedades de los modelos
- Escribir modelos formales y abstractos del programa
- Documentar y validar los requerimientos del usuario
- Escribir un modelo abstracto semiformal del diseño

- Escribir una especificación funcional abstracta y formal
- Verificar que el programa satisface el diseño y la especificación funcional

## 1.4. Proceso de desarrollo de Software

El proceso que se sigue para construir, entregar y hacer evolucionar el software, desde la concepción de una idea hasta la entrega y el retiro del sistema.

**Propiedades:** confiable, predecible y eficiente.

Dividir una tarea compleja en etapas manejables, determinar el orden de las etapas, criterio de transición para pasar a la siguiente etapa, criterio para determinar la finalización de cada etapa, criterio para comenzar y elegir la siguiente.

Existen varios modelos de desarrollo que se pueden seguir:

### ■ Modelo de Cascada:

Primera versión: Flujo secuencial entre las etapas, cada etapa tiene una entrada y una salida, para comenzar con una etapa deben haber finalizado las anteriores.

Versión actual: Es posible volver a las etapas anteriores, no es necesario haber terminado con las anteriores, la verificación no se hace solo al final.

### ■ Modelo de transformaciones formales:

S se transforma progresivamente en un modelo menos abstracto.

$-abstracto = +concreto = +refinado$

Cada  $ref_i$  es una transformación formal que devuelve un modelo más concreto.

$P \Rightarrow S_{n-1} \Rightarrow \dots \Rightarrow S_1 \Rightarrow S$

La transformación final devuelve  $P$ . De esta forma  $P$  es correcto por construcción.

### ■ Desarrollo ágil:

Se basa en desarrollo iterativo e incremental

Descompone las tareas en pequeños incrementos con mínima planificación

Las iteraciones duran entre 2 y 4 semanas

En cada iteración un equipo realiza R, D, P y verificación para implementar un incremento

El sistema resultante se muestra al cliente

Comunicación cara a cara más que documentos técnicos

El equipo incluye a un representante del cliente

Software que funciona es la medida de progreso

Técnicas que se usan: xUnit, pair programming, test driven development, patrones de diseño, etc.

## 1.5. Introducción a los Métodos Formales

Lenguajes, técnicas y herramientas basadas en matemática y/o lógica para describir y verificar sistemas de software

Comprenden: Lenguajes de especificación formal, verificación de modelos (model checking), prueba de teoremas, testing basado en modelos, cálculo de refinamiento.

Varios estándares internacionales exigen el uso de métodos formales: RTCA DO-178B, IEC SCAISRS, ESA SES, etc.

**Lenguajes de especificación formal:** Una sintaxis formal y estandarizada. Una semántica formal descrita en términos operativos, denotacionales o lógicos. Un aparato deductivo, también formal, que permite manipular los elementos del lenguaje según su sintaxis para demostrar teoremas.

**Especificación funcional(S):** Los lenguajes de especificación formal se usan casi siempre para escribir la especificación funcional de un programa. S es el criterio de corrección para P.

## 1.6. Requisitos vs Especificaciones

Los requisitos son declaraciones o descripciones de las funcionalidades, características y restricciones que debe cumplir un sistema o software. Representan las necesidades y expectativas de los usuarios, clientes u otras partes interesadas.

Las especificaciones son descripciones detalladas y técnicas de cómo se implementarán los requisitos del sistema o software. Estas descripciones suelen incluir información sobre el diseño, la arquitectura, los algoritmos, las interfaces y otros detalles técnicos necesarios para construir el sistema.

## 2. Z

### 2.1. Definiciones básicas

#### 2.1.1. Tipos de datos elementales

Siempre en mayúscula y entre corchetes, por ejemplo:

[TITULO, DIRECTOR, GUIONISTA]

#### 2.1.2. Tipos de datos enumerados

$msg ::= ok \mid error$

#### 2.1.3. Conjuntos

Siempre los conjuntos se definen a partir del conjunto base  $\mathbb{Z}$ , por ejemplo:

$\mathbb{N} == \{n : \mathbb{Z} \mid n \geq 0\}$ ,  $DINERO == \mathbb{N}$

#### 2.1.4. Axiomas

Los axiomas se definen en cajas como la siguiente, sin nombre. Los identificadores que se utilizan en las demás definiciones, son los que se definen en dicho axioma, y estos funcionan como una '*variable global*'.

Por ejemplo:

$size : \mathbb{N}$
$size = 10$

## 2.2. Máquinas jerárquicas

Supongamos que queremos especificar una base de datos de Películas, podríamos definir las siguientes máquinas, teniendo en cuenta los tipos definidos en la sección anterior:

$Película$
$dirs : \mathbb{P} DIRECTOR$
$guions : \mathbb{P} GUIONISTA$

$BD$
$películas : TITULO \rightarrow Película$

SIEMPRE hacer el Init de las máquinas

$PelículaInit$
$Película$
$dirs = \emptyset$
$guions = \emptyset$

$BDInit$
$BD$
$películas = \emptyset$

## 2.3. Transiciones

Tenemos dos tipos, las  $\Delta$  y las  $\Xi$ , en la primera, uno tiene que asignar un valor a las variables primadas, en la otra, las variables primadas quedan con el mismo valor implícitamente. Tomando como ejemplo las definiciones de la sección anterior, tenemos:

$PelículaDelta$	
$\Delta Película$	
...	
...	
$dirs' = dirs \cup \dots$	
$guions' = guions \cup \dots$	
$PelículaXi$	
$\Xi Película$	
...	
...	

## 2.4. Promoción de operaciones

Esto se utiliza, cuando tenemos una operación definida en un nivel de una máquina, y queremos pasarla a otra máquina de mayor nivel, que en su definición, incluye dicha máquina de menor nivel.

Supongamos que tenemos las máquinas definidas en las secciones anteriores, en dicho caso, *Película* es la máquina de menor jerarquía, y *BD* la de mayor.

Para realizar la promoción, se utiliza el operador  $\theta$ , el cual construye instancias de un tipo;  $\theta A$  construye una instancia del tipo  $A$ . Si  $A$  es:

$A$	
$x : X$	
$y : Y$	

entonces las instancias de  $A$  son registros de la forma:

$$\langle x : cte\_de\_tipo\_x, y : cte\_de\_tipo\_y \rangle$$

donde  $cte\_de\_tipo\_x$  y  $cte\_de\_tipo\_y$  son constantes de tipo  $X$  e  $Y$ , respectivamente. Esto se interpreta diciendo que la variable  $x$  vale  $cte\_de\_tipo\_x$  y lo mismo para  $y$ . Por lo tanto,  $\theta A$  construye tuplas de esa forma.

Una expresión  $\theta$  no dice qué instancia en particular se crea, sino que da las reglas para crearla. Estas condiciones hay que buscarlas en el esquema donde aparecen las expresiones  $\theta$  y son:

- Todos los predicados del esquema en los cuales aparecen las expresiones  $\theta$ .
- Para las expresiones de la forma  $\theta A$ , todos los predicados donde aparecen libres las variables que se declaran en  $A$ .
- Siempre se utiliza para los tipos esquema; es decir los tipos que definen los esquemas.
- Para las expresiones de la forma  $\theta A'$ , todos los predicados donde aparecen libres y primadas las variables que se declaran en  $A$ .
- Para encontrar esos predicados muchas veces es necesario expandir esquemas.

Para ver el funcionamiento de  $\theta$  supongamos que tenemos la siguiente operación, la cuál modifica un director de una película.

A nivel de *Pelicula* tenemos:

$\text{ModificarDirOk}$ $\Delta \text{Pelicula}$ $do? : \text{DIRECTOR}$ $dn? : \text{DIRECTOR}$ $rep! : \text{MSG}$
$do? \in \text{dirs}$ $dn? \notin \text{dirs}$ $\text{dirs}' = (\text{dirs} \setminus do?) \cup dn?$ $\text{guions}' = \text{guions}$ $rep! = \text{ok}$

Para pasar a nivel de *BD*, realizamos lo siguiente:

$\text{ModificarDirBDOK}$ $\Delta \text{BD}$ $\text{ModificarDir}$ $t? : \text{TITULO}$ $rep! : \text{MSG}$
$t? \in \text{dom}(\text{peliculas})$ $\text{peliculas}(t) = \theta \text{Pelicula}$ $\text{peliculas}' = \text{peliculas} \oplus \{t? \mapsto \theta \text{Pelicula}'\}$ $rep! = \text{ok}$

Se utiliza la operación *ModificarDir* (Completa), en la operación de la máquina de mayor nivel. (Notar que en ambos casos, faltaría definir las operaciones '*error*' para poder definir las operaciones completas).

En algunos casos, se puede realizar un *Marco de promoción* general, para evitar redundancia. Por ejemplo:

$\text{PeliculaABD}$ $\Delta \text{BD}$ $\Delta \text{Pelicula}$ $t? : \text{TITULO}$
$t? \in \text{dom}(\text{peliculas})$ $\text{peliculas}(t?) = \theta \text{Pelicula}$ $\text{peliculas}' = \text{peliculas} \oplus \{t? \mapsto \theta \text{Pelicula}'\}$

Esto nos permite definir las siguientes operaciones *ModificarDirBDOK* y *ModificarGuiBDOK* de una forma mas sencilla:

$$\begin{aligned} \text{ModificarDirBDOK} &== \text{PeliculaABD} \wedge \text{ModificarDir} \\ \text{ModificarGuiBDOK} &== \text{PeliculaABD} \wedge \text{ModificarGui} \end{aligned}$$

En caso de que no nos alcance el marco general, podemos agregar lo que nos falta de la siguiente forma:

$$\text{ModificarDirBDOK} == [\text{PeliculaABD}; \dots \mid \dots] \wedge \text{ModificarDir}$$

También se puede hacer promoción de operaciones a varios elementos(batch) de la siguiente forma:

$ \begin{array}{l} \text{ModificarDirBatchOk} \\ \hline \Delta BD \\ batch? : TITULO \rightarrow PELICULA \\ rep! : MSG \\ \hline dom(batch?) \subseteq dom(películas) \\ ran(batch?) \subseteq ran(películas) \\ películas' = películas \oplus \{t? : dom(batch?); \text{ModificarDir} \mid películas(t) = \theta Película \\ \quad \bullet t \rightarrow \theta Película'\} \\ rep! = ok \end{array} $
--

## 2.5. Propagación de errores en promoción

Teniendo el ejemplo anterior, supongamos que tenemos dos errores a nivel de *Película* (máquina inferior) los cuales son *PelículaError1* y *PelículaError2*, que aplican sobre *ModificarDir*, y además que tenemos un error a nivel *BD* (máquina superior), *BDError1*. Para pasar los errores de la máquina inferior, a la definición final de *ModificarDir* a nivel de máquina superior, lo hacemos de la siguiente manera:

$$ModificarDirBD == ModificarDirBDOk \vee BDError1 \vee (\exists BD \wedge (PelículaError1 \vee PelículaError2))$$

## 2.6. Composición de operaciones

Se utiliza el operador §, por ejemplo  $OP1 == OP2 \S OP3$

La definición formal de este operador es la siguiente, supongamos que *OP2* y *OP3* se definen sobre el esquema *E* el cual declara las variables *e1* y *e2*, la composición entre *OP2* y *OP3* se define de la siguiente forma:

$$OP2 \S OP3 == (OP2[e1''/e1', e2''/e2'] \wedge OP3[e1''/e1, e2''/e2]) \setminus (e1'', e2'')$$

donde  $\setminus (e1'', e2'')$  oculta las variables introducidas por el renombramiento, de manera tal que desaparece el estado intermedio, esto quiere decir, si tenemos:

$$A == [x : X; y : Y \mid P(x, y)]$$

$$\text{entonces el esquema } B == A \setminus (x)$$

$$\text{nos deja } B == [y : Y \mid \exists x \bullet P(x, y)]$$

Generalmente, se utiliza el reemplazo de variables en estos casos, ya que *OP2* y *OP3* pueden tener variables de entrada que coinciden, o se puede dar el problema que se detalla en la siguiente sección. Esto se realiza de la siguiente manera:

$$OP2[n1?/n?] \S OP3[n2?/n?]$$

Se cambia *n?* por *n1?* en *OP1* y *n?* por *n2?* en *OP2*. Esto se debe realizar de una forma coherente dependiendo como se componen *OP1* y *OP2*.

## 2.7. Problema de promoción de dos o más operaciones

Supongamos el siguiente escenario:

$$A == [x : X, y : Y], B == [v, w : A]$$

$$AOP_s == [\Delta A \mid P_s], AOP_t == [\Delta A \mid P_t]$$

Si queremos hacer la siguiente operación, que aplique una operación sobre *v* y otra sobre *w*:

$ \begin{array}{l} OP_{12} \\ \hline \Delta B; AOP_s; AOP_t \\ \hline v = \theta A \wedge v' = \theta A' \\ w = \theta A \wedge w' = \theta A' \end{array} $
--

Lo cual no es correcto, ya que implica que  $v = w \wedge v' = w'$

La solución es utilizar decoración de variable de la siguiente forma:

$$AOP_u == AOP_t[x_1/x, x_2/x', y_1/y, y_2/y']$$

Luego, tenemos

$OP_{12}$
$\Delta B; AOP_s; AOP_u$
$v = \theta A \wedge v' = \theta A'$
$w = \theta A_1 \wedge w' = \theta A_2$

Por convención, 1 es el estado actual y 2 el estado siguiente.

Tener cuidado cuando se hace composición ya que este problema puede aparecer y es necesario el renombramiento.

## 2.8. Funciones booleanas

Las funciones (parciales) booleanas son funciones que parten de cierto tipo y llegan al tipo *Bool*. Por ejemplo, si modelamos una guardia de hospital donde hemos declarado el tipo (básico) *PACIENTE* podríamos definir la función booleana *atendido* : *PACIENTE*  $\rightarrow$  *Bool* tal que si *atendido* p es verdadero significa que el paciente p ya fue atendido, y si es falso significa que aun no fue atendido.

El tema es que en Z no existe el tipo *Bool* (aunque se lo podría definir como un tipo enumerado, *Bool* :: *t* | *f*) y las funciones booleanas no son en absoluto necesarias, y en general son perjudiciales e indican que el especificador no domina completamente la notación Z.

Toda función booleana en Z puede (y debe) ser reemplazada por un conjunto.

## 2.9. Invariantes

Los invariantes se pueden definir en la definición (valga la redundancia) de las máquinas jerárquicas, o al estilo de B o TLA en una definición aparte:

**Estilo 1:**

<i>Pelicula</i>
<i>dirs</i> : $\mathbb{P}$ <i>DIRECTOR</i>
<i>guions</i> : $\mathbb{P}$ <i>GUIONISTA</i>
$\forall d : \text{dirs} \bullet d : \text{guions}$

**Estilo 2:**

<i>PeliculaInv</i>
<i>Pelicula</i>
$\forall d : \text{dirs} \bullet d : \text{guions}$

Generalmente elegimos utilizar el **Estilo 2** porque es ventajoso para el programador y obliga al especificador a escribir especificaciones más claras dado que debe hacer explícitas todas las precondiciones.

## 2.10. Obligaciones de prueba

Lemas de invariancia:

Sean  $A_1, A_2, \dots, A_n$  las operaciones de una especificación.

Sea *I* una invariante de dicha especificación.

Sea  $S_0$  el estado inicial de la especificación.

Entonces, se deben probar los siguientes lemas de invariancia:

$L_i, L_{A_1}, L_{A_2}, \dots, L_{A_n}$  donde

$L_i = I \wedge S_0 \Rightarrow I'$

$L_{A_i} = I \wedge A_i \Rightarrow I'$  con  $i = 1, \dots, n$

Esto quiere decir, si el invariante vale en el estado de partida, y ejecuto la operación, en el estado que me devuelve dicha operación, sigue valiendo el invariante.



### 2.11. Definición de funciones extra

Se pueden definir funciones extra, que no esten definidas por defecto en  $\mathbb{Z}$ , segun convenciencia. Por ejemplo, podemos definir la función *sum* de una secuencia de enteros, o una función *len* sobre secuencias de cualquier tipo  $X$ :

$$\begin{array}{|l}
 \hline
 sum : seq \mathbb{Z} \rightarrow \mathbb{Z} \\
 \hline
 sum \langle \rangle = 0 \\
 \forall s : seq \mathbb{Z}; n : \mathbb{Z} \bullet sum(s \frown \langle n \rangle) = n + sum(s)
 \end{array}$$
  

$$\begin{array}{|l}
 \hline
 \forall X \bullet len : seq X \rightarrow \mathbb{N} \\
 \hline
 len \langle \rangle = 0 \\
 \forall X \bullet \forall s : seq X, x : X \bullet len(s \frown \langle x \rangle) = 1 + len(s)
 \end{array}$$