

Tipos de schema UML en Nereus

Visibility (4 niveles), Multiplicity (4 constructores) y una RELATION SCHEMA ParticipatesSchema que combina ambos sobre el caso Person × Meeting.

Validación: las tres specs pasan el analyzer con 0 errores. Solo queda 1 warning V1 sobre schemaConforms (operación derivada con LHS de variable libre — falso positivo conocido del workspace).

Decisiones de diseño (acordadas):

- **Visibility:** 4 niveles UML completos — Public, Private, Protected, Package.
- **Cota superior infinita en Multiplicity:** 0 - 1 como sentinel (-1 sin literal porque el parser no soporta enteros negativos).
- **Tres archivos separados** en lugar de un único monolito — Visibility y Multiplicity son reusables por cualquier spec.

1. Visibility — 4 niveles UML

Visibility.nereus

```
"kw">CLASS "typ">Visibility
"kw">BASIC CONSTRUCTORS "typ">Vis_Public, "typ">Vis_Private, "typ">Vis_Protected, "typ">Vis_Package
"kw">EFFECTIVE
"kw">TYPE "typ">Visibility

"kw">OPERATIONS
  "typ">Vis_Public:    -> "typ">Visibility;
  "typ">Vis_Private:   -> "typ">Visibility;
  "typ">Vis_Protected: -> "typ">Visibility;
  "typ">Vis_Package:   -> "typ">Visibility;

vis_symbol:          "typ">Visibility -> "typ">String;
vis_isAccessible: "typ">Visibility * "typ">Visibility -> "typ">Boolean;

"kw">AXIOMS  v: "typ">Visibility;

-- Rendering UML.
vis_symbol("typ">Vis_Public())    = "+";
vis_symbol("typ">Vis_Private())   = "-";
vis_symbol("typ">Vis_Protected()) = "#";
vis_symbol("typ">Vis_Package())   = "~";

-- Regla de accesibilidad: vis_isAccessible(callerVis, declaredVis).
vis_isAccessible(v, "typ">Vis_Public())    = "kw">True;
vis_isAccessible(v, "typ">Vis_Private())   = "kw">False;
vis_isAccessible(v, "typ">Vis_Protected()) = (v = "typ">Vis_Protected()) "kw">or (v = "typ">Vis_Private
vis_isAccessible(v, "typ">Vis_Package())   = (v = "typ">Vis_Package()) "kw">or (v = "typ">Vis_Private
```

```
"kw">END-"kw">CLASS
```

Tabla UML → Nereus

UML symbol	Nombre UML	Constructor Nereus	Significado
+	Public	Vis_Public()	Accesible desde cualquier parte.
-	Private	Vis_Private()	Solo accesible internamente.
#	Protected	Vis_Protected()	Accesible por subclases.
~	Package	Vis_Package()	Accesible dentro del paquete.

2. Multiplicity — 4 constructores algebraicos

Multiplicity.nereus

```
"kw">CLASS "typ">Multiplicity
"kw">BASIC CONSTRUCTORS "typ">Mult_Exact, "typ">Mult_Range, "typ">Mult_AtLeast, "typ">Mult_Star
"kw">EFFECTIVE
"kw">TYPE "typ">Multiplicity

"kw">OPERATIONS
  "typ">Mult_Star:    -> "typ">Multiplicity;
  "typ">Mult_AtLeast: "typ">Integer -> "typ">Multiplicity;
  "typ">Mult_Range:   "typ">Integer * "typ">Integer -> "typ">Multiplicity;
  "typ">Mult_Exact:   "typ">Integer -> "typ">Multiplicity;

  mult_lower:    "typ">Multiplicity -> "typ">Integer;
  mult_upper:    "typ">Multiplicity -> "typ">Integer;  -- (0 - 1) = ∞
  mult_isStar:   "typ">Multiplicity -> "typ">Boolean;
  mult_accepts:  "typ">Multiplicity * "typ">Integer -> "typ">Boolean;

"kw">AXIOMS  n, m, k: "typ">Integer;

  -- Cotas inferiores.
  mult_lower("typ">Mult_Star())      = 0;
  mult_lower("typ">Mult_AtLeast(n))   = n;
  mult_lower("typ">Mult_Range(n, m)) = n;
  mult_lower("typ">Mult_Exact(n))     = n;

  -- Cotas superiores. (0 - 1) representa ∞.
  mult_upper("typ">Mult_Star())      = 0 - 1;
  mult_upper("typ">Mult_AtLeast(n))  = 0 - 1;
  mult_upper("typ">Mult_Range(n, m)) = m;
  mult_upper("typ">Mult_Exact(n))    = n;
```

```

mult_isStar("typ">Mult_Star())      = "kw">True;
mult_isStar("typ">Mult_AtLeast(n)) = "kw">True;
mult_isStar("typ">Mult_Range(n, m))= "kw">False;
mult_isStar("typ">Mult_Exact(n))    = "kw">False;

-- Pertenencia: ¿k cae dentro del rango?
mult_accepts("typ">Mult_Star(),      k) = (k >= 0);
mult_accepts("typ">Mult_AtLeast(n),  k) = (k >= n);
mult_accepts("typ">Mult_Range(n, m), k) = (n <= k) "kw">and (k <= m);
mult_accepts("typ">Mult_Exact(n),    k) = (k = n);

"kw">END-"kw">CLASS

```

Tabla UML → Nereus

UML notation	Significado	Constructor Nereus	lower / upper
*	0.. ∞ (cero o más)	Mult_Star()	0 / ∞
0..*	0.. ∞ (idem)	Mult_Star() o Mult_AtLeast(0)	0 / ∞
1..*	uno o más	Mult_AtLeast(1)	1 / ∞
n..*	n o más	Mult_AtLeast(n)	n / ∞
n..m	entre n y m	Mult_Range(n, m)	n / m
n	exactamente n	Mult_Exact(n)	n / n
0..1	opcional	Mult_Range(0, 1)	0 / 1
1	obligatorio simple	Mult_Exact(1)	1 / 1

3. ParticipatesSchema — RELATION SCHEMA Person × Meeting

Esta spec aplica los dos tipos anteriores al caso Person&Meeting del libro. Mantiene los axiomas algebraicos estándar (create, addLink, isRelated, get_meetings, get_participants, etc.) y agrega **metadata de schema** que se observa con operaciones a nivel-asociación: role1/role2, mult1/mult2, visibility1/visibility2, isFrozen/isChangeable/isAddOnly.

ParticipatesSchema.nereus (extracto)

```

"kw">RELATION SCHEMA "typ">ParticipatesSchema
"kw">IMPORTS "typ">Person, "typ">Meeting, "typ">Multiplicity, "typ">Visibility,
          "typ">Set["typ">Person:"typ">ANY], "typ">Set["typ">Meeting:"typ">ANY]
"kw">BASIC CONSTRUCTORS create, addLink
"kw">EFFECTIVE
"kw">TYPE "typ">ParticipatesSchema

"kw">OPERATIONS

```

```

-- Constructores algebraicos.
create: -> "typ">ParticipatesSchema;
addLink: "typ">ParticipatesSchema(a) * "typ">Person(p) * "typ">Meeting(m)
        -> "typ">ParticipatesSchema
        "kw">pre: "kw">not isRelated(a, p, m);

-- Metadata del schema.
role1, role2:      "typ">ParticipatesSchema -> "typ">String;
mult1, mult2:      "typ">ParticipatesSchema -> "typ">Multiplicity;
visibility1, visibility2: "typ">ParticipatesSchema -> "typ">Visibility;
isFrozen, isChangeable, isAddOnly: "typ">ParticipatesSchema -> "typ">Boolean;

-- Predicados y cardinalidades observadas.
isRelated:      "typ">ParticipatesSchema * "typ">Person * "typ">Meeting -> "typ">Boolean;
isRightLinked:  "typ">ParticipatesSchema * "typ">Person -> "typ">Boolean;
isLeftLinked:   "typ">ParticipatesSchema * "typ">Meeting -> "typ">Boolean;
rightCardinality: "typ">ParticipatesSchema * "typ">Person -> "typ">Integer;
leftCardinality: "typ">ParticipatesSchema * "typ">Meeting -> "typ">Integer;

-- Navegación.
get_meetings:    "typ">ParticipatesSchema * "typ">Person -> "typ">Set["typ">Meeting];
get_participants: "typ">ParticipatesSchema * "typ">Meeting -> "typ">Set["typ">Person];

-- Remover y verificar conformidad al schema.
remove: "typ">ParticipatesSchema(a) * "typ">Person(p) * "typ">Meeting(m)
        -> "typ">ParticipatesSchema
        "kw">pre: isRelated(a, p, m) "kw">and "kw">not isFrozen(a) "kw">and "kw">not isAddOnly(a);
schemaConforms: "typ">ParticipatesSchema * "typ">Person * "typ">Meeting -> "typ">Boolean;

"kw">AXIOMS a: "typ">ParticipatesSchema; p, p1: "typ">Person; m, m1: "typ">Meeting;

-- — Metadata del schema (constantes por instancia) —
role1(create())      = "participant";
role1(addLink(a, p, m)) = role1(a);

role2(create())      = "scheduled_meeting";
role2(addLink(a, p, m)) = role2(a);

mult1(create())      = "typ">Mult_AtLeast(1); -- 1..*
mult1(addLink(a, p, m)) = mult1(a);

mult2(create())      = "typ">Mult_Star();      -- 0..*
mult2(addLink(a, p, m)) = mult2(a);

visibility1(create())      = "typ">Vis_Public();
visibility1(addLink(a, p, m)) = visibility1(a);

visibility2(create())      = "typ">Vis_Public();
visibility2(addLink(a, p, m)) = visibility2(a);

isFrozen(create())      = "kw">False;
isFrozen(addLink(a, p, m)) = isFrozen(a);

isChangeable(create())      = "kw">True;
isChangeable(addLink(a, p, m)) = isChangeable(a);

isAddOnly(create())      = "kw">False;
isAddOnly(addLink(a, p, m)) = isAddOnly(a);

```

```

-- — Conformidad al schema —————
-- schemaConforms(a, p, m): La cardinalidad observada de p
-- en a cumple mult2, y la de m en a cumple mult1.
schemaConforms(a, p, m) =
    mult_accepts(mult2(a), rightCardinality(a, p))
    "kw">and
    mult_accepts(mult1(a), leftCardinality(a, m));

-- (Los axiomas de isRelated, get_meetings, get_participants,
-- cardinalidades, remove, etc. siguen el patrón estándar
-- de ASSOCIATION; ver el archivo completo.)

"kw">END- "kw">CLASS

```

4. Ejemplo concreto: tres Persons, dos Meetings, tres links

Para ilustrar cómo se usa schemaConforms, simulo una sesión:

Construcción	Valor algebraico	Visión semántica
Empty	<code>a0 = create()</code>	<code>{}</code>
Insert (Alice, M1)	<code>a1 = addLink(a0, Alice, M1)</code>	<code>{ (Alice, M1) }</code>
Insert (Bob, M1)	<code>a2 = addLink(a1, Bob, M1)</code>	<code>{ (Alice, M1), (Bob, M1) }</code>
Insert (Alice, M2)	<code>a3 = addLink(a2, Alice, M2)</code>	<code>{ (Alice, M1), (Bob, M1), (Alice, M2) }</code>

Evaluación de metadata y conformidad sobre a3

Expresión	Resultado	Por qué
<code>role1(a3)</code>	<code>"participant"</code>	Constante propagada por axiomas; no depende de los links.
<code>mult1(a3)</code>	<code>Mult_AtLeast(1)</code>	Constante. Cada reunión necesita ≥ 1 participante.
<code>mult2(a3)</code>	<code>Mult_Star()</code>	Constante. Una persona puede tener 0 o más reuniones.
<code>visibility1(a3)</code>	<code>Vis_Public()</code>	Constante. Se puede leer la asociación libremente.
<code>rightCardinality(a3, Alice)</code>	<code>2</code>	Alice está en M1 y M2.
<code>rightCardinality(a3, Bob)</code>	<code>1</code>	Bob solo en M1.
<code>leftCardinality(a3, M1)</code>	<code>2</code>	M1 tiene a Alice y Bob.
<code>leftCardinality(a3, M2)</code>	<code>1</code>	M2 solo tiene a Alice.

<code>mult_accepts(Mult_AtLeast(1), 2)</code>	True	M1 cumple mult1 (≥ 1 participante). ✓
<code>mult_accepts(Mult_Star(), 2)</code>	True	Alice cumple mult2 ($0..*$). ✓
<code>schemaConforms(a3, Alice, M1)</code>	True	Ambas cardinalidades dentro del rango. ✓
<code>schemaConforms(a3, Alice, M2)</code>	True	$1 \in [1..\infty]$, $2 \in [0..\infty]$. ✓

Caso de violación: agregar (Carol, M3) donde M3 no tiene a nadie aún

Si hubiéramos construido `a4 = addLink(create(), Carol, M3)` directamente (sin más links), la cardinalidad izquierda de M3 es 1, lo que **cumple** `mult1 = 1..*`. Pero si por algún motivo se removiera ese único link (`remove(a4, Carol, M3)`) y M3 quedara con 0 participantes, `leftCardinality` daría 0, y `mult_accepts(Mult_AtLeast(1), 0) = False` — el schema dejaría de conformar. Ese chequeo se hace en runtime con `schemaConforms`.

5. Limitaciones del parser detectadas

1. Literales enteros negativos no soportados. -1 falla en el parser; usamos `0 - 1` como workaround para la convención de "cota superior infinita". El significado es el mismo (el evaluador computa la resta antes de comparar).

2. END-RELATION no reconocido. Aunque el parser *sí* acepta el header `RELATION SCHEMA ...`, no acepta el cierre simétrico `END-RELATION`. Como workaround, las specs `RELATION SCHEMA` deben cerrar con `END-CLASS`. Esta es una asimetría sintáctica del analyzer que vale la pena reportar — la misma asimetría aparece en `Bidirectional_3.nereus` con el mismo error.

3. Concatenación de strings (+ sobre String). No está soportada. `mult_render(Mult_Range(1, 5)) = intToString(1) + ".." + intToString(5)` falla. Por eso eliminé `mult_render`: el rendering UML se hace en el target (Haxe/C++/etc.) si hace falta.

6. Mapeo a códigos objetivo

Tipo Nereus	Haxe	C++	Java
Visibility (4 constructores nullary)	<pre>enum Visibility { Vis_Public; Vis_Private; Vis_Protected; Vis_Package; }</pre>	<pre>enum class Visibility { Public, Private, Protected, Package };</pre>	<pre>enum Visibility { PUBLIC, PRIVATE, PROTECTED, PACKAGE }</pre>
Multiplicity (4 constructores, paramétricos)	<pre>enum Multiplicity { Mult_Star; Mult_AtLeast(n:Int); Mult_Range(n:Int, m:Int);</pre>	<pre>std::variant<MultStar, MultAtLeast, MultRange, MultExact> o jerarquía sellada</pre>	Sealed class jerarquía: sealed class Multiplicity permits MultStar,

	<code>Mult_Exact(n:Int); }</code> (GADT)		<code>MultAtLeast,</code> <code>MultRange,</code> <code>MultExact {}</code>
--	---	--	---

Tres archivos generados: `Visibility.nereus` (0/0), `Multiplicity.nereus` (0/0), `ParticipatesSchema.nereus` (0 errores, 1 warning V1 false positive). Convenciones aplicadas: pattern-match por constructor, axiomas `implies`, metadata del schema como pares (`create+propagación`), conformidad chequeada vía `mult_accepts`.