PROGRAMACION CONCURRENTE Y DISTRIBUIDA

VIII.3: Slice: Specification Language for Ice

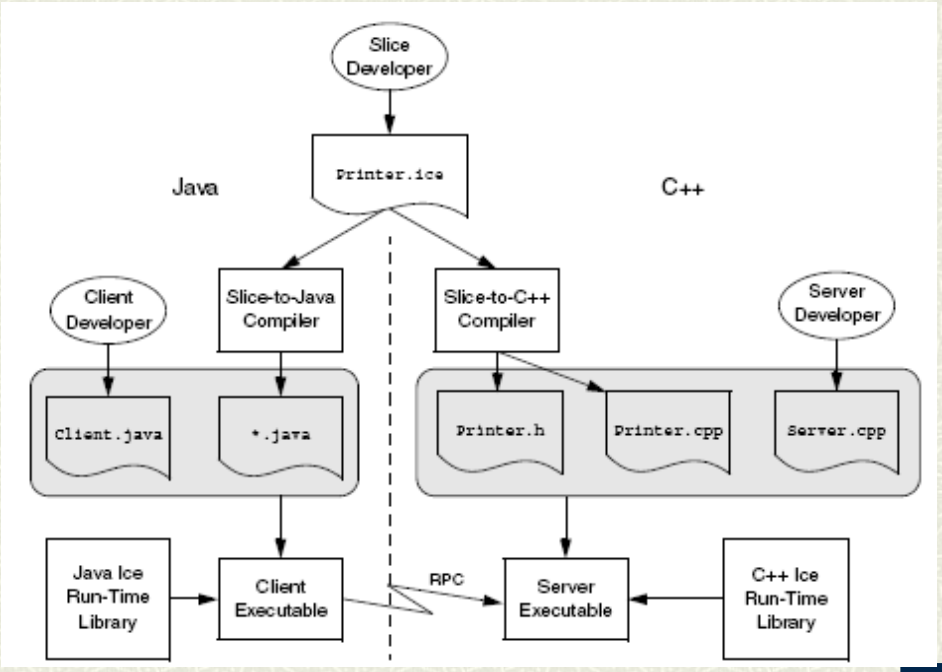
José M. Drake
L.Barros

Notas:

Posibilidades que ofrece Java para la comunicación en red: Socket,RMI y URL.

- Los lenguajes de especificación abstracta de interfaces es la herramienta básica para desacoplar la especificación de la implementación de los servicios.
- Slice formula el contrato entre un servidor y sus clientes de forma independiente a las características de la plataforma, y con independencia del lenguaje con el que se implementa el cliente y el servidor.
- Slice es un lenguaje puramente declarativo, describe la interfaz, pero no proporciona ningún detalle de como se implementa.
- La especificación Slice de una interfaz se compila con una herramienta específica para cada lenguaje de programación, y genera un conjunto de ficheros en código fuente que enlazados con el código de negocio permite generar servidores y clientes ejecutables.

Notas:


Slice Developer
Printer.ice
Java
C++
Client Developer
Slice-to-Java Compiler
Slice-to-C++ Compiler
Server Developer
Client.java
*.java
Printer.h
Printer.cpp
Server.cpp
Java Ice Run-Time Library
Client Executable
RPC
Server Executable
C++ Ice Run-Time Library

Notas:

Ficheros Slice

- Los ficheros con especificaciones Slices son ficheros de texto cuyo nombre debe tener la extención ".ice".

  Ejemplo: Printer.ice

- El formato es libre se pueden utilizar espacios, tabs, Line feed, return, etc de forma libre. Los espacios son separadores.
- Los constructores Slice (modulos, interfaces, tipos, etc) pueden aparecer en cualquier orden, pero cualquier identificador debe ser declarado antes de ser referenciado.
- Cometarios: Igual que java:

```
/*
 *  Esto es un comentario
 */
// Esto también
```


Notas:

KeyWord en Slice

- Slice usa un conjunto de palabras claves que están reservadas. En ellas el tipo de carácter (Mayúscula/minúscula) debe ser respetado.

The following identifiers are Slice keywords:

bool	enum	implements	module	string
byte	exception	int	nonmutating	struct
class	extends	interface	Object	throws
const	false	local	out	true
dictionary	float	LocalObject	sequence	void
double	idempotent	long	short	

Keywords must be capitalized as shown.

Notas:

Identificadores Slice

- Los identificadores comienzan por una letra y le sigue cualquier numero de letras y números.
  - No se admiten letras no inglesas (p.e. á, ü, ...)
  - No se admite el carácter "_" (undescore)
- Los identificadores con las mismas letras pero con mayúsculas/minuscula diferentes son idénticos. Sin embargo Slice exige que el identificador se escriba siempre de igual manera.
- Las keywords de los lenguajes se pueden utilizar, y el compilador las mapea añadiéndole un prefijo:
  - Por ejemplo: switch en java => _switch
    en C++ => _cpp_switch

  Mejor no utilizarlos.
- Las keywords de Slice se pueden utilizar poniéndole el prefijo "\"
  struct dictionary{... (No válido) struct \dictionary{... (Válido)
- No se pueden utilizar los grupos de letras: ICE, Helper, Prx y Ptr con cualquier capitalización. Por ejemplo "Icecream" no es un identificador válido.

Notas:

Módulos Slice

- Todos los elementos Slice deben estar incluidos en un módulo.
- Los módulos sirven para agrupar elementos relacionados y para definir un espacio de nombres

```
module ZeroC{
    module Client{
        // Definición de elementos aquí
    };
    module Server{
        // Definición de elementos aquí
    };
};
```

- Un módulo puede reabrirse múltiples veces y continuar agregándole nuevos elementos.
- Existe un único modulo predefinido que se denomina Ice.

Notas:

Tipos Slice básicos

Type	Range of Mapped Type	Size of Mapped Type
`bool`	`false` or `true`	≥ 1bit
`byte`	–128–127[a]	≥ 8 bits
`short`	-2^{15} to $2^{15}-1$	≥ 16 bits
`int`	-2^{31} to $2^{31}-1$	≥ 32 bits
`long`	-2^{63} to $2^{63}-1$	≥ 64 bits
`float`	IEEE single-precision	≥ 32 bits
`double`	IEEE double-precision	≥ 64 bits
`string`	All Unicode characters, excluding the character with all bits zero.	Variable-length

a. Or 0–255, depending on the language mapping

Notas:

Tipos enumerados

- Permite definir un tipo de variable enumerando los valores que puede tomar:

```
enum Frutas{Manzana, Pera, Naranja};
```

- No se permite asignar el valor de representación de cada valor, ni se pueden hacer hipótesis sobre el valor que se le asigna (puede ser dependiente del lenguajes)
- Cada definición de un tipo enumerado define un nuevo espacio de nombres:

```
enum Frutas{Manzana, Pera, Naranja};
enum Colores{Rojo, Naranja, Verde}; // Naranja se redefine
```


Notas:

Tipos Struct

- Permite definir un nuevo tipo agrupando un conjunto de tipos previamente definidos:

```
struct HoraDelDia{
    short hora;       // 0 – 23
    short minuto;     // 0 – 59
    short segundo;    // 0 - 59
};
```

```
struct Punto{
    short x;
    short y;
};
struct Linea{
    Punto inicial;
    Punto final;
};
```


Notas:

Tipos Sequence

- Permite definir un vector de algún tipo de elementos de longitud variable (de 0 hasta un valor ilimitado):

```
sequence<Frutas> Frutero;
```

- Se admite utilizar secuencia de secuencias:

```
sequece<Frutero> Fruteria;
```

- La sentencia es útil para definir, colecciones, conjuntos, listas colas, stack, árboles, etc. Depende de la semántica que se le de al orden.
- Dado que las secuencias pueden estar vacías, son útiles para definir campos opcionales:

```
sequence<long> NumSerieOpt;
struct Componente{
    string nombre;
    string descripcion;
    NumSerieOpt numeroSerie; // puede tener ningún elemento o un entero long
}
```


Notas:

Tipo Dictionary

- Es una lista de entradas de parejas de un campo que se denomina key y otro value. El campo key no puede estar repetido.

```
struct Empleado{
    long dni;
    string nombre;
    string fechaNacimiento;
}
dictionary<long,Empleado> Fichero;
```

- Para el campo key solo pueden utilizarse los siguientes tipos:
  - Tipos enteros: byte, short, int, long, bool y tipos enumerados
  - String
  - Secuences con elementos de tipos enteros o string
  - Struct que contengan tan sólo miembros de tipos enteros o string.

Notas:

Definición de constantes

- Se pueden definir constantes de los siguientes tipos:
  - Tipos enteros: byte, short, int, long, bool y tipos enumerados
  - Tipos float y double
  - String

```
const bool     AppendByDefault = true;
const byte     LowerNibble = 0x0f;
const string   Advice = "Don't Panic!";
const short    TheAnswer = 42;
const double PI = 3.1416;

enum Fruit { Apple, Pear, Orange };
const Fruit    FavoriteFruit = Pear;
```


Notas:

Definición de constantes numéricas

- Tienen un formato muy semejante a C++ o Java

```
const byte    TheAnswer = 42;
const byte    TheAnswerInOctal = 052;
const byte    TheAnswerInHex = 0x2A; // or 0x2a

const float   P1 = -3.14f;      // Integer & fraction, with suffix
const float   P2 = +3.1e-3;     // Integer, fraction, and exponent
const float   P3 = .1;          // Fraction part only
const float   P4 = 1.;          // Integer part only
const float   P5 = .9E5;        // Fraction part and exponent
const float   P6 = 5e2;         // Integer part and exponent
```


Notas:

Definición de constantes string

```
const string    AnOrdinaryString = "Hello World!";
const string    DoubleQuote = "\"";
const string    TwoSingleQuotes = "'\'";            // ' and \' are OK
const string    Newline = "\n";
const string    CarriageReturn = "\r";
const string    HorizontalTab = "\t";
const string    VerticalTab = "\v";
const string    FormFeed = "\f";
const string    Alert = "\a";
const string    Backspace = "\b";
const string    QuestionMark = "\?";
const string    Backslash = "\\";
const string    OctalEscape = "\009";               // Same as \a
const string    HexEscape = "\x09";                 // Ditto
const string    UniversalCharName = "\u03A9";       // Greek Omega
```


Notas:

Interfaces y operaciones Slice

- Las interfaces son el objeto básico de las declaraciones de Slice:

```
struct TimeOfDay {
        short hour;       // 0 - 23
        short minute;     // 0 - 59
        short second;     // 0 – 59
};
interface Clock {
        TimeOfDay getTime();
        void setTime(TimeOfDay time);
};
```

- Dentro de una interfaz sólo se pueden declarar operaciones . No se pueden declarar, tipos, ni datos, ni excepciones, etc.

Notas:

Parámetros y valores de retorno

- Una operación tiene siempre un valor de retorno (que puede ser void, y 0 o mas parámetros.
- Los parámetros son por defecto *in*, y si son out debe declarase explícitamente, y colocarse detrás de los parametros *in.*
- Slice no soporta parámetros *inOut*

```
void changeSleepPeriod(
                TimeOfDay startTime,
                TimeOfDay stopTime,
            out TimeOfDay prevStartTime,
            out TimeOfDay prevStopTime);
```

- Slice no admote ningúntipo de sobrecarga de nombres: Todas las operaciones de una interfaz deben tener diferentes nombres.

Notas:

Tipos de operaciones

- Nonmutating Operations: Son aquellas cuya ejecución no cambia el estado del servidor:

```
interface Clock{
        nonmutating TimeOfDay getTime();
                    void setTime(TimeOfDay time);
};
```

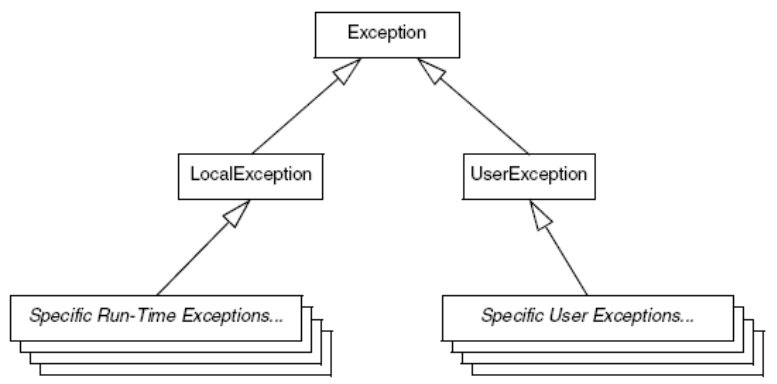
- Idempotent Operations: Son aquellos que si son ejecutadas dos veces obtienen el mismo resultado (p.e.: x=1)

```
interface Clock{
        nonmutating TimeOfDay getTime();
        idempotent   void setTime(TimeOfDay time);
};
```


Notas:

Excepciones Slice

- El siguiente árbol declara las herencias entre excepciones:



- Las excepciones de usuario deben ser declaras en la especificación Slice y deben ser especificadas en las operaciones que las lanzan.
- Las excepciones Ice Run-Time no pueden declarase, ya que su lanzamiento es posible en cualquier operación.

Notas:

Excepciones de usuario

- Las operaciones que pueden lanzar una excepción definida por el usuario deben declararlas en su especificación Slice:

```
exception Error {}; // Empty exceptions are legal
exception RangeError {
        TimeOfDay errorTime;
        TimeOfDay minTime;
        TimeOfDay maxTime;
 };

interface Clock{
        nonmutating TimeOfDay getTime();
        idempotent void setTime(TimeOfDay time)
                        throws RangeError, Error;
};
```

- Una Excepción no es un tipo de datos, y por ello:
  - No se puede pasar como el valor un parámetro.
  - No se pueden usar como el tipo de un miembro
  - No se puede usar como elemento de secuencia ni clave de un dictionary

Notas:

Herencia de excepciones

- Las excepciones soportan herencia.
- Si una operación especifica que lanza una excepción, puede en ejecución lanzar tipos mas específicos:

```
exception ErrorBase { string reason; };
enum RTError { DivideByZero, NegativeRoot, IllegalNull };
exception RuntimeError extends ErrorBase { RTError err; };
enum LError { ValueOutOfRange, ValuesInconsistent, };
exception LogicError extends ErrorBase { LError err; };
exception RangeError extends LogicError {
        TimeOfDay errorTime;
        TimeOfDay minTime;
        TimeOfDay maxTime;
};
interface Example{
      void op() throws ErrorBase;
           // Puede lanzar ErrorBase, RuntimeError, LogicError y RangeError
       };
```


Notas:

Semántica de interfaz y proxies

- Si se aplica el operador * a una interfaz (Clock*) representa un proxy a un objeto que ofrece ese tipo de interfaz.
- Puede ser un valor retornado por una operación, o el tipo de un parámetro de una operación.
- Un proxy juega el mismo papel que un puntero:
  - Puede ser null
  - Puede designar a un objeto que no existe.
  - Las operaciones invocadas por el proxy usan la semántica de late binding: Si el tipo actual del objeto designado es mas especializado que el tipo del proxy, se invocará la implementación mas especializada de la interface.

Notas:

Ejemplo de proxy.

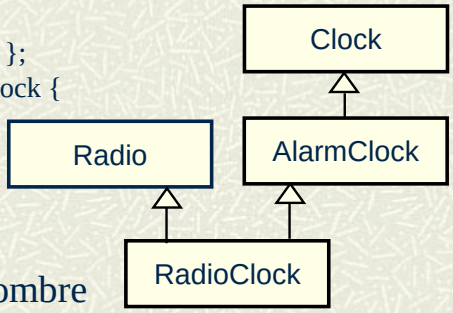
```
exception GenericError { string reason; };
struct TimeOfDay { short hour;     // 0 – 23
                   short minute;   // 0 - 59
                   short second;   // 0 - 59
};
exception BadTimeVal extends GenericError {};
interface Clock {
        nonmutating TimeOfDay getTime();
        idempotent void setTime(TimeOfDay time) throws BadTimeVal;
};
dictionary<string, Clock*> TimeMap;       // Time zone name to clock map
exception BadZoneName extends GenericError {};
interface WorldTime {
      idempotent void addZone(string zoneName, Clock* zoneClock);
                   void removeZone(string zoneName) throws BadZoneName;
      nonmutating Clock* findZone(string zoneName) throws BadZoneName;
      nonmutating TimeMap listZones();
      idempotent void setZones(TimeMap zones);
};
```


Notas:

Herencia en interfaces

- La interfaces admiten herencia simple o múltiple

```
interface AlarmClock extends Clock {
    nonmutating TimeOfDay getAlarmTime();
    idempotent void setAlarmTime(TimeOfDay alarmTime) throws BadTimeVal;
};
interface Radio {
    void setFrequency(long hertz) throws GenericError;
    void setVolume(long dB) throws GenericError;
};
enum AlarmMode { RadioAlarm, BeepAlarm };
interface RadioClock extends Radio, AlarmClock {
    void setMode(AlarmMode mode);
    AlarmMode getMode();
};
```



- Por herencia una interfaz no puede heredar dos operaciones de igual nombre

Notas:

Classes Slice

- Las clases son un hibrido de interfaz y struct.
- Mientras que las interfaces representan un servicio ofrecido por el servidor, las clases permiten definir un servicio ofrecido en el lado del cliente.
- Las clases admiten herencia simple, y admiten polimorfismo en las invocaciones.
- Las clases pueden implementar interfaces

Notas:

Ejemplo de Clase Slice

```
interface Time {
    string format();      // ...
};
class TimeOfDay implements Time {
    short hour;
    short minute;
    short second;
};
interface I1 {
    TimeOfDay get();                  // Pass by value
     void put(TimeOfDay time);        // Pass by value
};
interface I2 {
     Time* get();                     // Pass by reference
};
```


Notas:

Predeclaraciones Forward

- Las interfaces y las clases pueden predefinirse mediante declaraciones forward. Las declaraciones forward permiten crear objetos interdependientes:

```
module Family {
    interface Child;                          // Forward declaration
    sequence<Child*> Children;                // OK
    interface Parent {
       Children getChildren();                // OK
    };
    interface Child {                         // Definition
        Parent* getMother();
        Parent* getFather();
    };
};
```


Notas:

Interfaz Object

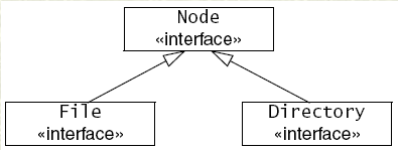
- Es una interfaz definida en Ice de la que heredan todas las interfaces.
- Su pseudo definición es:

```
sequence<string> StrSeq;
interface Object {   // "Pseudo" Slice!
        void ice_ping();
                        // Realiza una prueba de accesibilidad al objeto
        bool ice_isA(string typeID);
                        // Verifica si la interfaz corresponde con el ID
        string ice_id();
                        // Retorna el ID mas especializado de la interfaz
        StrSeq ice_ids();
                        // Retorna una lista de la ID que implementa la interfaz
};
```


Notas:

Ejemplo: Simple File System

```
module Filesystem {
        interface Node {
              nonmutating string name();
        };
        exception GenericError { string reason; };
        sequence<string> Lines;
        interface File extends Node {
                  nonmutating Lines read();
                  idempotent void write(Lines text) throws GenericError;
         };
         sequence<Node*> NodeSeq;
         interface Directory extends Node {
                  nonmutating NodeSeq list();
         };
         interface Filesys {
                  nonmutating Directory* getRoot();
         };
};
```



Notas:

Compilando la especificación Slice

- ICE proporciona compiladores a cada lenguaje que soporta:

  slice2cpp.exe slice2cs.exe slice2java.exe slice2vb.exe slice2ph.exe

- El comando de compilación es

  <compiler-name> [options] file...

- Algunas de las opciones que admite son:
  - -h, --help — *Muestra un mensaje de ayuda*
  - -v, --version — Muestra la versión del compilador
  - -Idir — Suma un directorio en el path de busqueda de ficheros #include.
  - -E — Muestra la salida del preprocesador en stdout.
  - -output-dir Dir — Situa los ficheros de salida en el directorio Dir
  - -d, --debug — Muestra la información de debug mostrando las operaciones del Slice Parser.

Notas:

Resultados de la precompilación en Java

- Cuando compilas un fichero .ice que contiene una interfaz con nombre <interfaz-name>, genera los siguientes ficheros:
  - Ficheros de interes para los clientes:
    - <interfaz-name>. Java // Declara la interfaz Java
    - <interfaz-name>Holder.java // Describe los tipos de la interfaz
    - <interfaz-name>Prx.java // Describe la interfaz del proxy
    - <interfaz-name>PrxHelper.java // Describe un fichero de ayuda del proxy
    - <interfaz-name>PrxHolder.java // Describe el fichero de tipos del proxy
    - <interfaz-name>Operations.java // Interfaz con las operaciones y Ctxt
    - <interfaz-name>OperationNC.java // Interfaz con las operaciones
  - Ficheros relativos al servidor:
    - <interfaz-name>Displ.java // Clase abstracta de la que se deriva e servant
    - <interfaz-name>Del.java // Clase utilizada por ICE
    - <interfaz-name>DelD.java // Clase utilizada por ICE
    - <interfaz-name>DelM.java // Clase utilizada por ICE

Notas:

Interface Proxy: <interfaz-name>Prx.java

- Define la interfaz java que ofrece hacia el cliente el proxy

Simple.ice

```
Interface Simple{
   void op();
};
```

```
public interface ObjectPrx{
      boolean equals(java.lang.Object r);
      Identity ice_getIdentity();
      int  ice_getHash();
      boolean ice_isA(String _id);
      String ice_id();
      void ice_ping();
....
}
```

SimplePrx.java

```
public interface SimplePrxextends Ice.ObjectPrx{
      public void op();
      public void op(java.util.Map _Context);
}
```


Notas:

Servant generado por herencia

- Por defecto el servant se construye extendiendo la clase abstracta la clase _<Interfase_name>Disp

Filesystem.ice

```
module Filesystem {
    interface Node {
        nonmutating string name();
    };
    // ...
};
```

NodeI.java

```
package Filesystem;
public final class NodeI extends _NodeDisp {
    public NodeI(String name) {
        _name = name;
    }
    public String name(Ice.Current current) {
```

Ice.Current: provee acceso a información sobre la petición actual a la implementación de una operación en el servidor

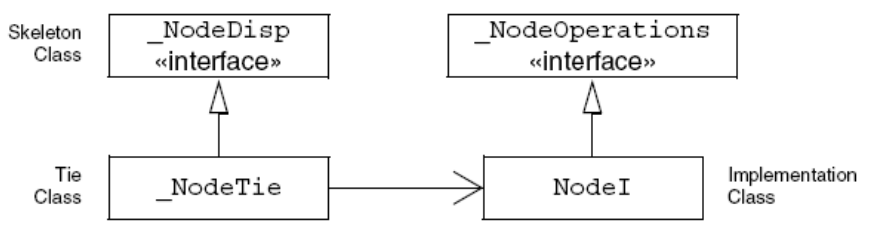
Server.java

```
NodeI node=new NodeI("fred");
adapter.add("Node", node);
```


Notas:

Servant generado por agregación

- La clase _NodeTie (extends _NodeDisp) delega cada invocacion de un método que corresponde a la operación Slice a la implementación de la clase.



Constructor de _NodeTie

```
public _NodeTie(_NodeOperations delegate) {
    _ice_delegate = delegate;
}
```


Notas:

Servant generado por agregación

- Usar este modelo es más costoso en términos de memoria porque cada objeto es encarnado por dos objetos Java, el tie y el delegado.

Filesystem.ice

```
module Filesystem {
    interface Node {
        nonmutating string name();
    };
    // ...
};
```

Server.java

```
package Filesystem;
public final class NodeI implements _NodeOperations {
    public NodeI(String name) {
        _name = name;
    }
    public String name(Ice.Current current) {
        return _name;
    }
    private String _name;
}
…
NodeI fred = new NodeI("Fred");        // Create implementation
_NodeTie servant = new _NodeTie(fred);  // Create tie
adapter.add("Node", servant);
```


Notas: