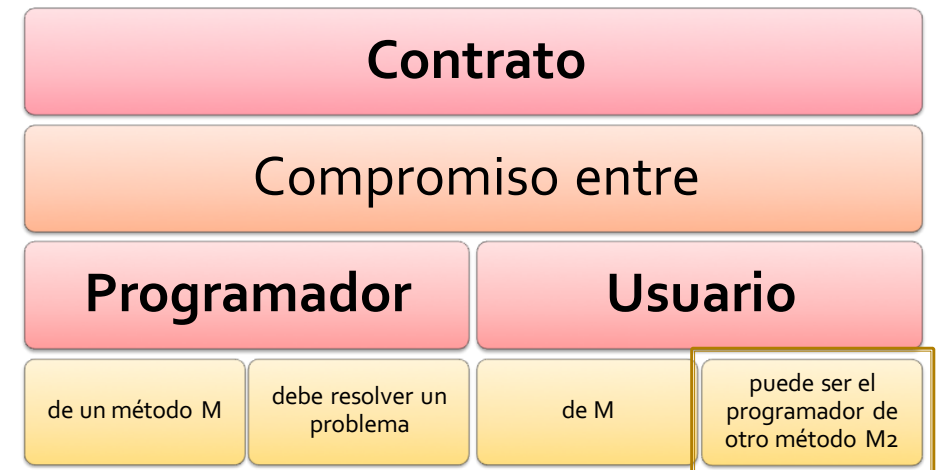


Análisis Automático de Programas

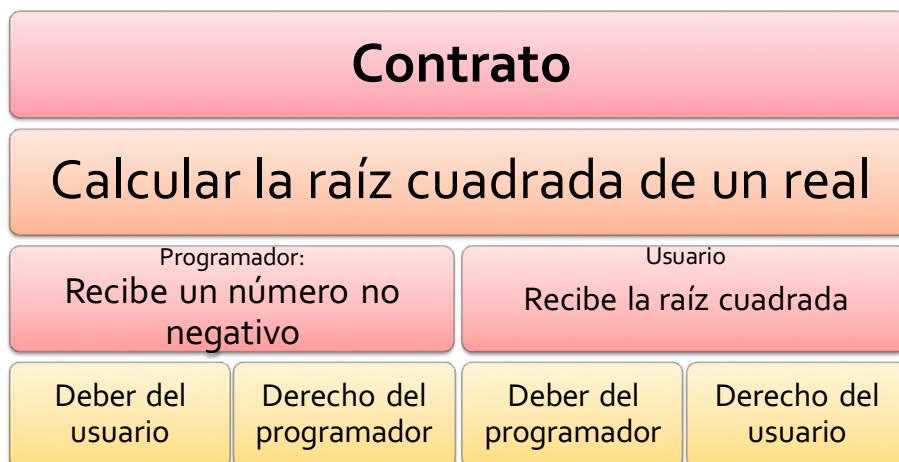
Programación por contratos

Diego Garbervetsky
Departamento de Computación
FCEyN – UBA

Design by contract



Ejemplo



Ventajas del uso de contratos

- Favorece la modularidad
 - Abstrae detalles particulares de la implementación
- Permite utilizar distintas soluciones
 - El cliente sólo evalúa el contrato a la hora de utilizarlo
 - Esto se aplica tanto a un programador como a un verificador automático
 - Sólo hace falta que cumplan con el contrato
 - Luego se elige por ejemplo por criterios de performance

Especificación de Contratos en Componentes

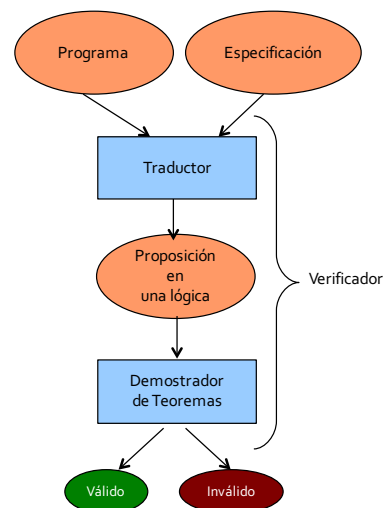
- Un componente implementa alguna entidad o elemento importante en nuestra solución
 - Componente = conjunto de clases
 - Clase = conjunto de métodos
- Contratos
 - Por cada método: **Requires, Ensures**
 - A nivel clase: **invariantes** de objetos
 - A nivel componente: sistemas de **ownerships** + invariantes conectados entre objetos

Compilador/Verificador

- El Compilador/Verificador
 - Realiza un chequeo automatizado de corrección del programa que compila
 - La corrección está especificada por tipos, aserciones y cualquier otra anotación redundante en el programa
- [Hoare, 2004]

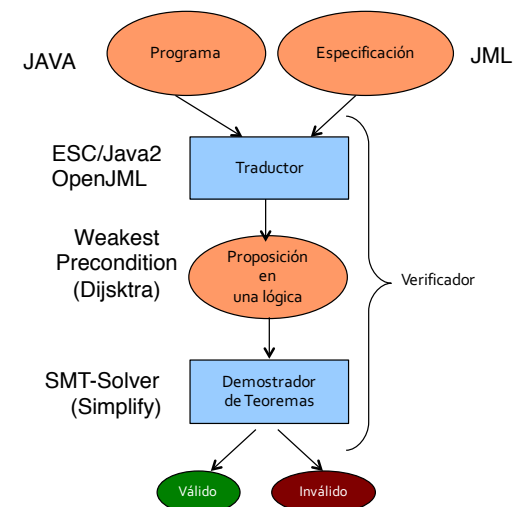
Verificador (1)

- Demostración de teoremas y software
- *Soundness*:
 - Si la proposición es un teorema **válido**, entonces el programa cumple la especificación
 - Si el teorema es **demostrable**, entonces es válido



Verificador (2)

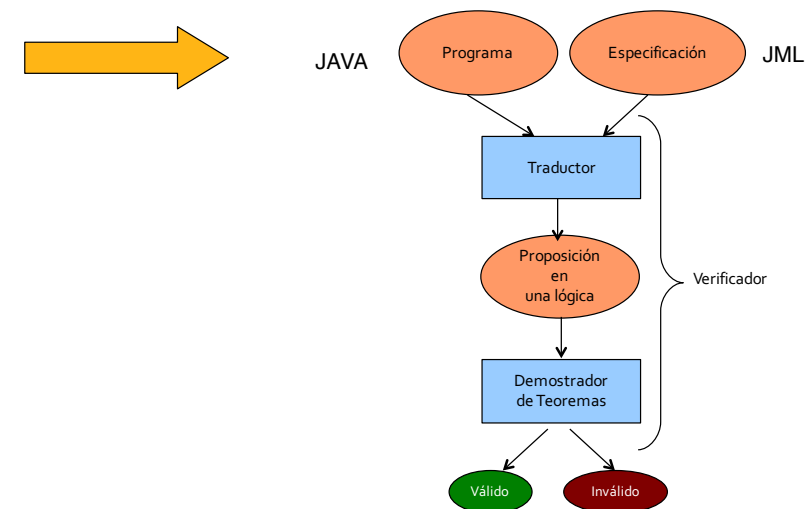
- Lenguaje de Programación
- Lenguaje de Especificación
- Representación lógica del programa
- Procedimiento automático de verificación



Propiedades deseables de un verificador

- **Corrección** (*soundness*)
 - Si el verificador no reporta fallas, entonces el programa no falla
- **Completitud**
 - Si el verificador reporta una falla, entonces el programa falla
- **Terminación**
 - El verificador termina de ejecutar con cualquier programa como entrada

El lenguaje de especificación



JML: Java Modeling Language

- Lenguaje de especificación formal para Java
- Objetivo: Un lenguaje que sea fácil de usar para cualquier programador Java
- Uso inicial: chequeos dinámicos
- Inspiración: lenguaje Eiffel
- Paralelo en C#: Spec#, CodeContracts

Cómo es una especificación JML

```
public class ArrayOps {
    private /*@ spec_public @*/ Object[] a;
    //@ public invariant 0 < a.length;

    /*@ requires 0 < arr.length;
    @ ensures this.a == arr;
    @*/
    public void init(Object[] arr) {
        this.a = arr;
    }
}
```

El campo es privado pero puede usarse para la especificación

Invariante del objeto

Especificación del método init

Ejemplo (simple)

- Queremos un componente que nos maneje números racionales de forma precisa
- Ejemplos
 - 1, 1/2, 2/3, 23/10, 2/4
- Operaciones deseables
 - Crear
 - Sumar, Restar, Multiplicar
 - ...

Representando racionales

- Idea: usar numerador y denominador
- Listo?

```
class Racional
{
    private int numerador;
    private int denominador;
    public int getNumerador();
    public int getDenominador();

    public Racional(int n, int d);
    public bool equals(Racional r);
    public void sumar(Racional r);
    public void restar(Racional r);
    public void multiplicar(Racional r);
}
```

- No! Ningun racional es representado por $n/0$;
- Diferentes representaciones para el mismo racional!
 - (1, 2), (2, 4), (3, 6) ...

Representando racionales

- Conviene tener una forma uniforme de representar

```
class Racional
{
    private int numerador;
    private int denominador;
    //@ invariant mcd(numerador, denominador)==1
    && denominador!=0;

    public int getNumerador();
    public int getDenominador();
    public Racional(int n, int d);
    public bool equals(Racional r);
    public void sumar(Racional r);
    public void restar(Racional r);
    public void multiplicar(Racional r);
}
```

ahora (2,4)
y (x,0) no son
racionales!

- Aun se "cuelan" (1,-2), (-1,2)

Representando racionales

- **Invariante:** predicado que indica una propiedad fundamental que debe mantenerse en el objeto

```
class Racional
{
    private int numerador;
    private int denominador;
    //@ invariant mcd(numerador, denominador)==1
    && denominador>0;

    public int getNumerador();
    public int getDenominador();

    public Racional(int n, int d);
    public bool equals(Racional r);
    public void sumar(Racional r);
    public void restar(Racional r);
    public void multiplicar(Racional r);
}
```

Invariantes de Clase

- Propiedad fundamental que debe cumplir un objeto
 - Para el que el objeto este en un estado consistente
- Fuera de la clase debe ser **siempre** verdadero
 - Pensar en un tipo
- Internamente
 - Puede ser asumido por los métodos de la clase
 - Debe ser mantenido también por ellos

Especificación del constructor

- Algún problema con esto?

```
class Racional
{
    private int numerador;
    private int denominador;
    //@ invariant mcd(numerador, denominador)==1
        && denominador>0;

    public Racional(int n, int d){

        numerador = n;
        denominador = d;
    }
}
```

Especificación del constructor

- Algún problema con esto?

```
class Racional
{
    private int numerador;
    private int denominador;
    //@ invariant mcd(numerador, denominador)==1
        && denominador>0;

    //@ requires d>0;
    public Racional(int n, int d){

        numerador = n;
        denominador = d;
    }
}
```

Especificación del constructor

- Ahora sí

```
class Racional
{
    private int numerador;
    private int denominador;
    //@ invariant mcd(numerador, denominador)==1
        && denominador>0;

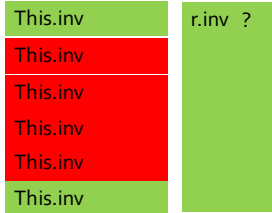
    //@ requires d>0;
    public Racional(int n, int d){
        int mcd = Math.gcd(n,d);
        numerador = n/mcd;
        denominador = d/mcd;
    }
}
```

Especificando operaciones

- Podemos asumir el invariante, pero debemos restablecerlo

```
class Racional
{
    private int numerador;
    private int denominador;
    //@ invariant mcd(numerador, denominador)==1
    && denominador>0;
    //@ requires r!=this
    public void suma(Racional r){
        numerador = numerador*r.denominador
        + r.numerador * denominador;
        denominador = denominador*r.denominador;

        int mcd = Math.gcd(numerador,denominador);
        numerador = numerador/mcd;
        denominador = denominador/mcd;
    }
}
```



Invariantes y abstracción

- Pregunta: Qué problema le ven a este método?

```
class C {
    private int x,y;
    //@ requires y!=0;
    public dividir(){
        x = 100/y;
    }
}
```

- Mejor este!

```
class C {
    private int x,y;
    //@ invariant y!=0;
    public dividir(){
        x = 100/y;
    }
}
```

Helper methods

- Los invariantes deben valer a la entrada y salida del método
 - A veces esto es muy restrictivo
- Helper methods:
 - Tienen que ser privados
 - No requieren que valga el invariante
 - Ni establecer el invariante

```
class Racional
{
    private int numerador;
    private int denominador;
    invariant mcd(numerador, denominador)== 1
    && denominador>0;

    public void suma(Racional r){
        numerador n = numerador*r.denominador
        + r.numerador * denominador;
        denominador =denominador*r.denominador;
        reconstruir();
    }
}
```

```
/* @helper */ void reconstruir()
{
    int mcd = Math.gcd(numerador,
        denominador);
    numerador = numerador/mcd;
    denominador = denominador/mcd;
}
```

Resumen

- Un objeto “complejo” necesita estar en un estado consistente
- Esta garantía puede lograrse mediante el uso de invariantes
- Mantener el invariante requiere un esfuerzo del programador
 - Pero facilita implementación de código que sea correcto porque se debe ser consciente de las presunciones que se realizan

Aserciones JML

- Agregadas como comentarios en el programa entre `/*@...@*/`, o luego de `//@`
- Expresiones Java booleanas extendidas con algunos operadores especiales
 - `\old`, `\forall`, `\result`, `\sum`...
- Varios tipos de anotaciones
 - `requires`, `ensures`, `signals`,
 - `invariant`
 - `assignable`, `pure`, `non_null`,...

JML: pre-, post-condiciones (1)

```
/*@ requires amount >= 0;  
   @ ensures balance == \old(balance) - amount;  
   @ ensures \result == balance  
   @*/  
public int debit(int amount) {...}
```

- `\old(...)` refiere al valor de la expresión antes de la ejecución
- `\result` refiere al valor de retorno del método

JML: pre-, post-condiciones (2)

```
/*@ requires amount >= 0;  
   @ ensures \result == balance  
   @*/  
public int debit(int amount) {...}
```

- Las especificaciones JML pueden ser tan débiles o fuertes como se quiera
- ¿Más fuerte o más débil que la anterior?

JML: assert (1)

- Un `assert` especifica una propiedad que debe valer en algún punto del código

```
if (i <= 0 || j < 0) {  
    ...  
} else if (j < 5) {  
    //@ assert i > 0 && 0 < j && j < 5;  
    ...  
}
```

JML: assert (2)

- El JML assert es más expresivo que el Java assert: permite el uso de expresiones JML

```
for (n=0; n<a.length; n++) {  
  if (a[n]==null) break;  
  //@ assert (\forallall int i; 0<=i && i<n; a[i]!=null);  
  ...  
}
```

- Es muy útil para tests de unidad

JML: assume

- Es como el JML assert, pero su predicado es considerado verdadero

```
//@ assume b!=null && b.length>0;  
b[0]=2;
```

- Práctico durante la etapa de desarrollo
 - Documentas las asunciones
 - “Ayuda” a un demostrador automático (filtra casos)

Invariante de ciclo


- Predicado que describe el progreso realizado por el ciclo
- Es parte del razonamiento (quizás inconsciente) que realizamos al escribir un ciclo
 - Descripción formal:
 - Lo que asumimos al principio de cada iteración
 - Lo que avanzamos al final de la iteración
- Fundamental para la verificación de un método si tiene ciclos

Invariantes de ciclo

```
int sumax(x: Int) {  
  //@ requires x >= 0;  
  //@ ensures result == \sum(i: int; 0<=i<=x; i)
```

```
int sumax (int x) {  
  int s = 0, i = 0;  
  while (i < x) {  
  
  }  
  return s;  
}
```

| i | s |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 3 |
| 3 | 6 |



Invariantes de ciclo

```
int sumax(x: Int) {
  //@ requires x >= 0;
  //@ ensures result == \sum(i: int; 0<=i<=x; i)
```

```
int sumax (int x) {
  int s = 0, i = 0;
  while (i < x) {

    i = i + 1;
    s = s + i;

  }
  return s;
}
```

| i | s |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 3 |
| 3 | 6 |

Invariantes de ciclo

```
int sumax(x: Int) {
  //@ requires x >= 0;
  //@ ensures result == \sum(i: int; 0<=i<=x; i)
```

```
int sumax (int x) {
  int s = 0, i = 0;
  while (i < x) {
    // estado 1
    i = i + 1;
    s = s + i;
    // estado 2
  }
  return s;
}
```

| i@e1 | s@e1 | i@e2 | s@e2 |
|------|------|------|------|
| 0 | 0 | 1 | 1 |
| 1 | 1 | 2 | 3 |
| 2 | 3 | 3 | 6 |
| 3 | 6 | 4 | 10 |

Invariante de ciclo

```
s == \sum(j: int; 0<=j<=i; j)
&& 0<=i<=x
```

JML: anotaciones de ciclos

- JML provee formas de anotar el invariante y la función variante de un ciclo:

```
//@ loop_invariant product==m*i && i>=0 && i<=n && n>0;
//@ decreases n-i;
while (i < n) {...}
```

- loop_invariant**: invariante de ciclo
- decreases**: función variante

Otras anotaciones: frame conditions (1)

- Una **frame condition** limita los efectos colaterales (*side-effects*) de los métodos:

```
/*@ requires amount>=0;
   @ modifies this.balance
   @ ensures this.balance == \old(balance)-amount
   @*/
public int debit(int amount) {
  balance = balance - amount;
}
```

- Especificaciones parciales: es complejo (o imposible) limitar cambios usando sólo la postcondición
- Por defecto: modifies **everything**

Otras anotaciones: frame conditions (2)

- Un método sin *side-effects* se llama “puro”
 - `public /*@ pure */ int getBalance() {...}`
- Un método puro equivale a la anotación:
 - `modifies \nothing`
- En las especificaciones sólo pueden ser invocados métodos puros
 - `//@ invariant 0<=this.getBalance() && ...`

JML: ghost fields

- A veces es conveniente introducir un campo extra, sólo por propósitos de especificación
- Un campo *ghost* es un campo normal, salvo por el hecho que es accesible sólo por la especificación
 - Ejemplo: `//@ ghost Object F;`
- Una sentencia especial **set** es usada para cambiar el valor de los campos *ghost*
 - En lugar de asignarle un valor, se utiliza una condición que captura el valor deseado.
 - Ejemplo: `//@ set F==null;`

JML: ghost fields

```
class Animal {  
  //@ ghost Zoo owner;  
  ....  
}
```

```
Class Zoo {  
  void add(Animal a) {  
    ...  
    //@ set a.owner==this;  
  }  
  
  //@ requires a.owner==this;  
  void feed(Animal a) {...}  
}
```

Otras anotaciones: \reach(...)

- Retorna el conjunto de elementos “alcanzables”
- Captura la clausura reflexo transitiva de una relación binaria
 - $R^* = \{\} \cup R \cup (R;R) \cup (R;R;R) \cup (R;R;R;R) \cup \dots$
- El conjunto retornado es un `JMLObjectSet`

Otras anotaciones: \reach(...)

- `\reach(this.f)`
 - Todos los objetos alcanzables usando todos los campos desde el objeto sito en `this.f`
- `\reach(this.f, T, f2)`
 - Todos los objetos alcanzables de tipo `T` usando SÓLO el campo `f2`

Ejercicio

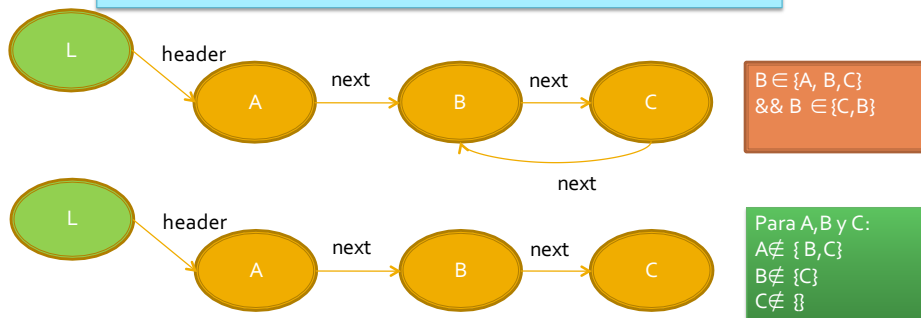
```
class List { Node header; }  
class Node { Node next; Object data; }
```

- Escribir un invariante de la clase `List` tal que la lista de nodos sea **acíclica**.

```
/*@  
  @ invariant (\forall Node n;  
    @   (\reach(this.header, Node, next)).has(n) ;  
    @   !(\reach(n.next, Node, next)).has(n) );  
  @*/
```

Analizando el predicado

```
/*@  
  @ invariant (\forall Node n;  
    @   (\reach(this.header, Node, next)).has(n) ;  
    @   !(\reach(n.next, Node, next)).has(n) );  
  @*/
```



Ejemplo

- Cual es el problema con este código

```
public class Counter {  
  private /*@ spec_public @*/ int val;  
  //@ modifies val;  
  //@ ensures val == \old(val + y.val);  
  //@ ensures y.val == \old(y.val);  
  public void addInto(Counter y) {  
    val += y.val; }  
}
```

Ejemplo

- Cual es el problema con este código?

```
public class Counter {
    private /*@ spec_public @*/ int val;
    //@ modifies val;
    //@ requires y!=this;
    //@ ensures val == \old(val + y.val);
    //@ ensures y.val == \old(y.val);
    public void addInto(Counter y){
        val += y.val; }
}
```

Constructores e invariantes

- Que problema hay en este código?
- Como se arregla?

```
public class SortedInts {
    /*@ invariant(\forall int i, j;
        @ 0 <= i && i < j && j < a.length;
        @ a[i] <= a[j]); @*/
    /*@ requires (\forall int i, j;
        @ 0 <= i && i < j && j < inp.length;
        @ inp[i] <= inp[j]);
        @ modifies a;
        @ ensures a == inp; @*/
    public SortedInts(int[] inp) {
        a = inp;
    }
}
```

Constructores e invariantes

```
class SortedInts2 {
    private /*@ spec_public rep @*/ int[] a;
    /*@ requires (\forall int i, j;
        @ 0 <= i && i < j && j < inp.length;
        @ inp[i] <= inp[j]);
        @ modifies a;
        @ ensures \fresh(a);
        @ ensures a.length == inp.length;
        @ ensures (\forall int i;
        @ 0 <= i && i < inp.length;
        @ a[i] == inp[i]); @*/
    public SortedInts2(int[] inp) {
        a = new /*@ rep @*/ int[inp.length];
        for (int i = 0; i < a.length; i++) {
            a[i] = inp[i];
        }
    }
}
```

Metodologías de uso de invariantes

- Problemas potenciales
 - Exponer la representación
 - Reentrada
- Semánticas de invariantes que tratan de lidiar con esto
 - Ownership type system
 - Obligar a restablecer el invariante en cada llamada
- Se busca que el invariante siempre valga al inicio de un método