Verificación y Validación de Software

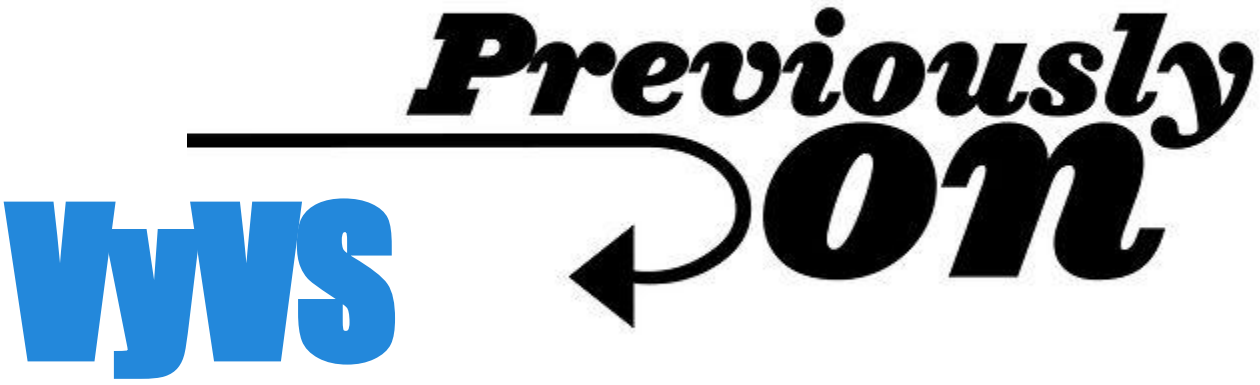
Ingeniería en Sistemas de Información

Departamento de Ciencias e Ingeniería de la Computación

2019

8

Verificación y Validación de Software



Verificación y Validación de Software

Evaluación y Ejecución Simbólica

Symbolic Evaluation and Execution

La evaluación simbólica es un análisis; donde análisis se define como una inspección para entender las propiedades y capacidades del objeto en cuestión

Verificación y Validación de Software

Evaluación y Ejecución Simbólica

Symbolic Evaluation and Execution

1. PROGRAM ATP	1. { (x = undef), (x = undef), (PC = true) }
2. VAR	2.
3. x, p : INTEGER;	3.
4. BEGIN	4.
5. read(x);	5. { (x = Q), (p = undef), (PC = true) }
6. read(p);	6. { (x = Q), (p = A), (PC = true) }
7. x = p;	7. { (x = A), (p = A), (PC = true) }
8. IF (x >= 10)	8. { (x = A), (p = A), (PC = true AND A > 10) } -> { (x = A), (p = A), (PC = A > 10) }
9. THEN BEGIN	9.
10. IF (y > 0)	10. { (x = A), (p = A), (PC = A > 10 AND A > 0) } -> { (x = A), (p = A), (PC = A > 10) }
11. THEN	

Verificación y Validación de Software

Testing de Caja Blanca

White Box Testing

1. Seleccionar el criterio de cubrimiento.
2. Seleccionar el conjunto de caminos que satisfagan [1].
3. Ejecución Simbólica sobre los caminos de [2] para determinar el PC correspondiente.
4. Seleccionar datos que satisfagan el PC de [3].
5. Ejecutar un test con los datos de [4].

Verificación y Validación de Software

Análisis de Flujo de Datos

Data Flow Analysis

Este análisis se enfoca en la evolución del valor asociado a las variables durante la ejecución de un programa.

La técnica utiliza un grafo llamado Grafo de Definición y Uso (def/use).

Verificación y Validación de Software

Análisis de Flujo de Datos

Data Flow Analysis

Una variable x puede ser usada en tres situaciones diferentes:

- Para asociarle un valor;
  - **def**
- Para usarla como parte de un cálculo
  - **c-use**
- Para tomar una decisión
  - **p-use**

Verificación y Validación de Software

Análisis de Flujo de Datos

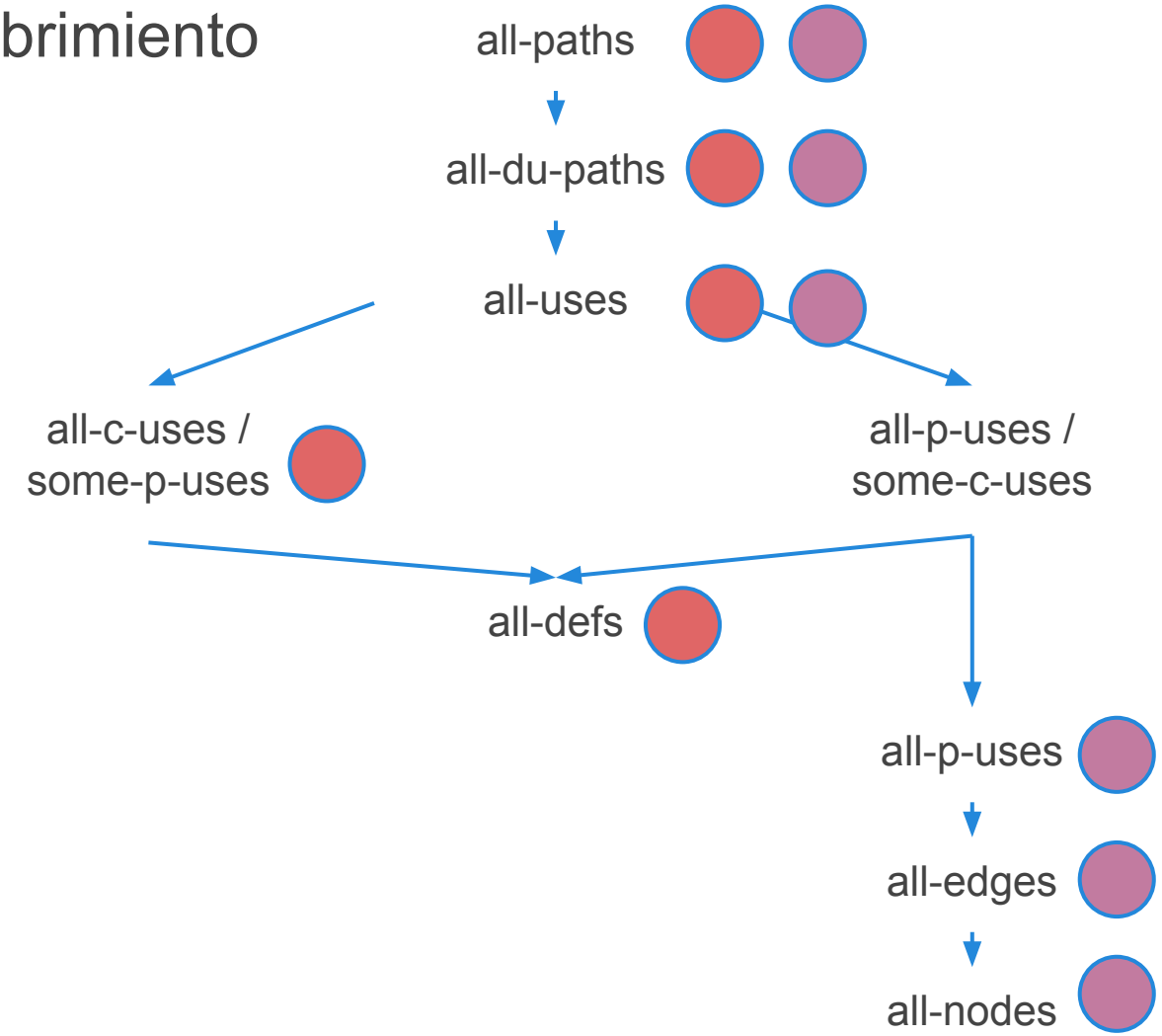
Data Flow Analysis

El objetivo es analizar el flujo de datos a través del programa.

Verificación y Validación de Software

Criterios de Cubrimiento

Análisis de Flujo de Datos



Error de Dominio

Error de Computo

Verificación y Validación de Software

Testing Dinámico

Dynamic Testing

El Testing Dinámico se puede dividir en cuatro categorías, con varias zonas grises en el medio.

- Caja Negra
- Caja Blanca
- Basado en Experiencia
- Basado en Defectos
- Análisis Dinámico

Los casos de test se diseñan en base a nuestra experiencia e intuición. También diseñamos los test en base a los tipos de errores que estamos buscando.

Verificación y Validación de Software

Testing Basado en Experiencia y Defectos

Experience and Defect Based Testing

Mientras que las técnicas anteriores son sistemáticas y estructuradas, éstas no. Estas técnicas no tienen procedimientos o protocolos, sino que suelen ser descripciones informales. Son igual de útiles que las técnicas anteriores.

Verificación y Validación de Software

Testing Basado en Experiencia y Defectos

Experience and Defect Based Testing

Al ser más informales, los conceptos de modelo y cubrimiento se vuelven más flexibles. El modelo se puede considerar como una lista de “defectos” a buscar, y el cubrimiento es simplemente “testear cada item de la lista”.

Verificación y Validación de Software

Testing Basado en Experiencia y Defectos

Experience and Defect Based Testing

La hipótesis detrás de estas técnicas es que solemos cometer los mismos errores. Lo que significa que el mismo equipo de desarrollo probablemente cometa los mismos errores proyecto, tras proyecto. O, que para los mismos tipos de proyectos, aparezcan siempre los mismos errores.

Verificación y Validación de Software

Testing Basado en Experiencia y Defectos

Experience and Defect Based Testing

Dentro de la informalidad, podemos tener algo de orden. En particular, los defectos pueden ser ubicados en categorías.

Verificación y Validación de Software

Testing Basado en Experiencia y Defectos

Experience and Defect Based Testing

- Funcionales
- Sistema
- Proceso
- Datos

hay más...

Verificación y Validación de Software

Testing Basado en Experiencia y Defectos

Experience and Defect Based Testing

Existe una técnica, dentro de este tipo de testing, que se llama “error guessing”. Como su nombre lo indica, el tester “adivina” cuales son los errores posibles y desarrolla tests en función de esto. Se lo conoce como “Caja Gris”.

Verificación y Validación de Software

Testing Basado en Experiencia y Defectos

Experience and Defect Based Testing

Otra técnica es la de Fallas o Sucia en donde los casos de test buscan romper todo. A partir del código fuente, se generan casos de test que disparen excepciones o condiciones no establecidas.

Verificación y Validación de Software

Testing Dinámico

Dynamic Testing

El Testing Dinámico se puede dividir en cinco categorías, con varias zonas grises en el medio.

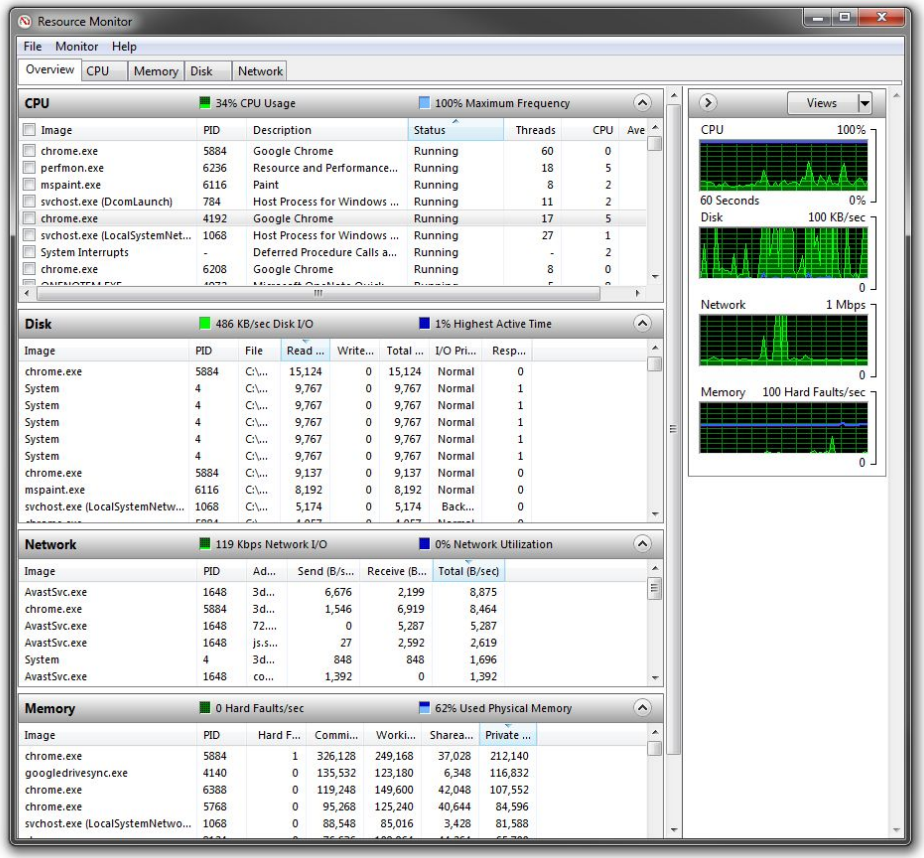
- Caja Negra
- Caja Blanca
- Basado en Experiencia
- Basado en Defectos
- Análisis Dinámico

Una aplicación se analiza mientras está ejecutando, se suele usar código entre el código o herramientas de inspección.

Verificación y Validación de Software

Análisis Dinámico

Dynamic Analysis



Verificación y Validación de Software

Análisis Dinámico

Dynamic Analysis

Sea f(x) una función que queremos testear

El sistema es f(n)

El testing es f(2) == $f_{esperado}(2)$. *Probamos para 2, sí funciona asumimos que funciona para todos los n.*

Verificación y Validación de Software

Análisis Dinámico

Dynamic Analysis

Sea *Sys* mi objeto de test

Verificación y Validación de Software

Análisis Dinámico

Dynamic Analysis

Sea *Sys* mi objeto de test

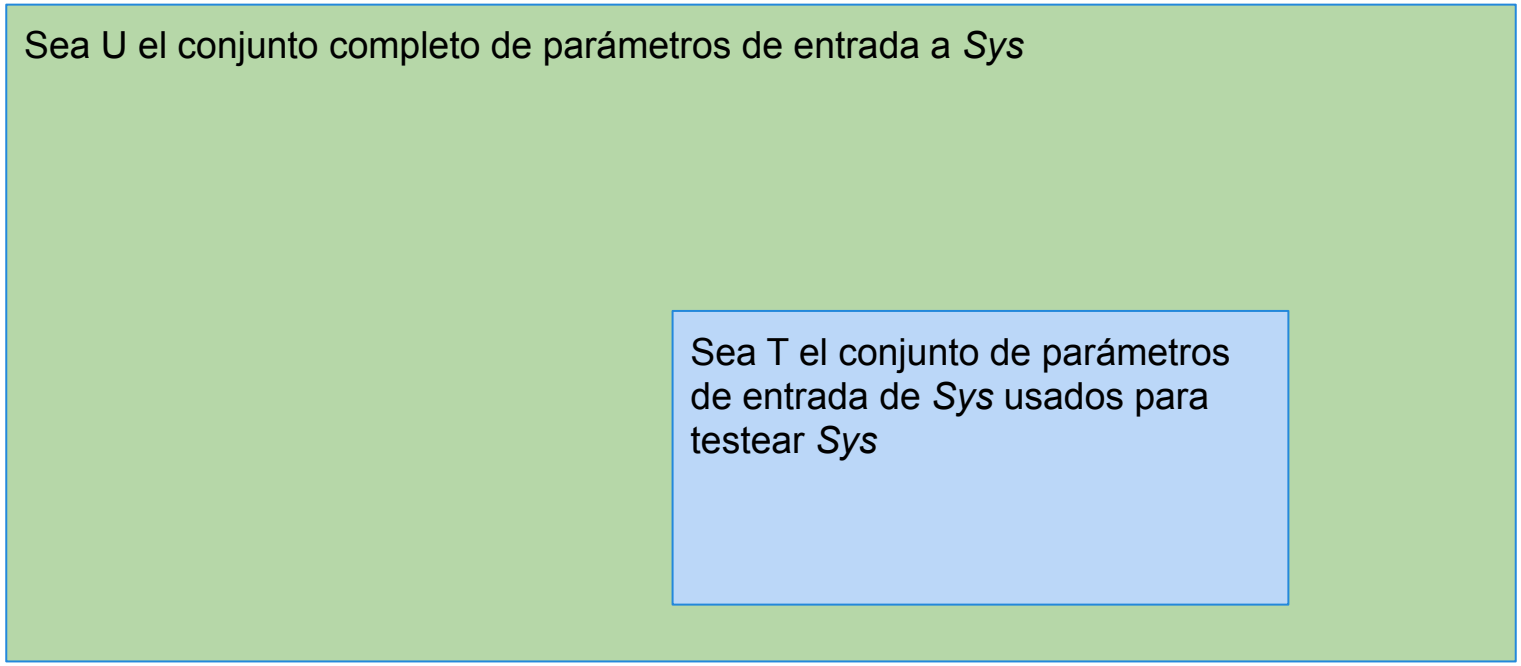
Sea U el conjunto completo de parámetros de entrada a *Sys*

Verificación y Validación de Software

Análisis Dinámico

Dynamic Analysis

Sea *Sys* mi objeto de test

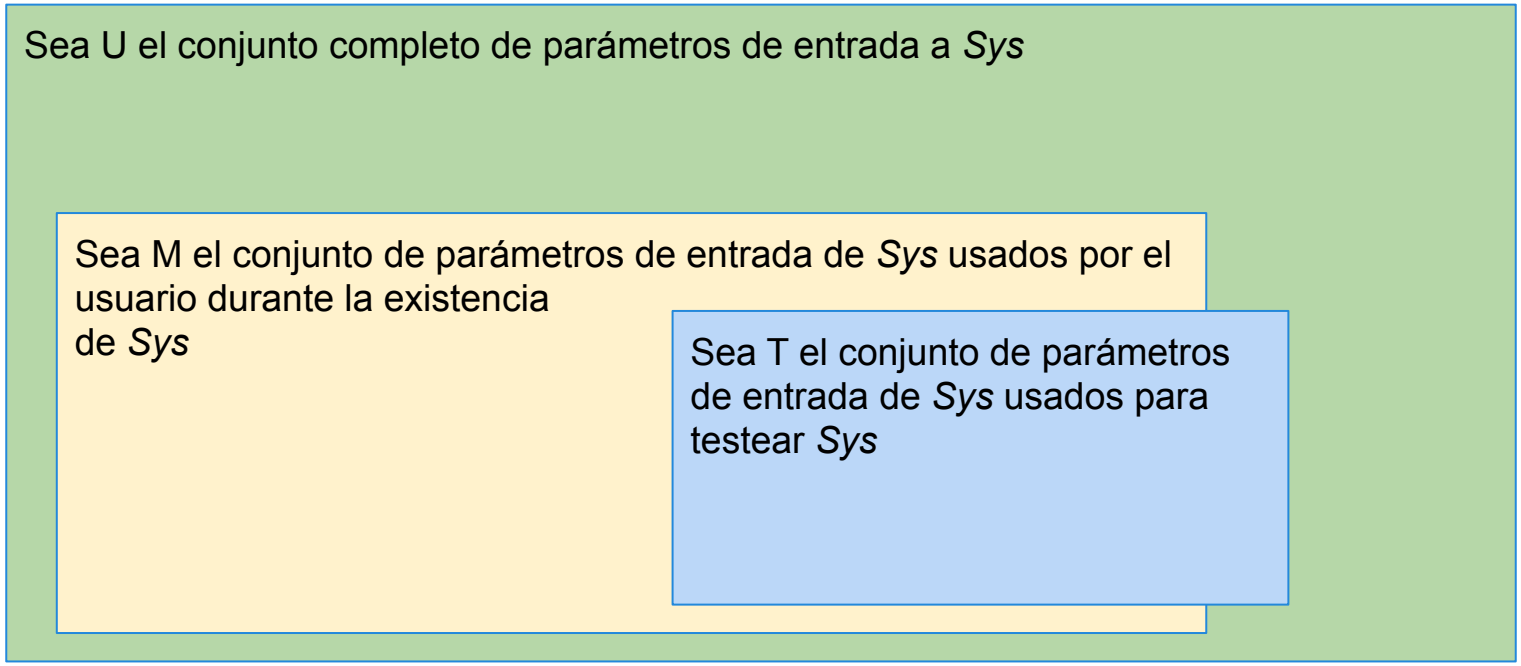


Verificación y Validación de Software

Análisis Dinámico

Dynamic Analysis

Sea *Sys* mi objeto de test

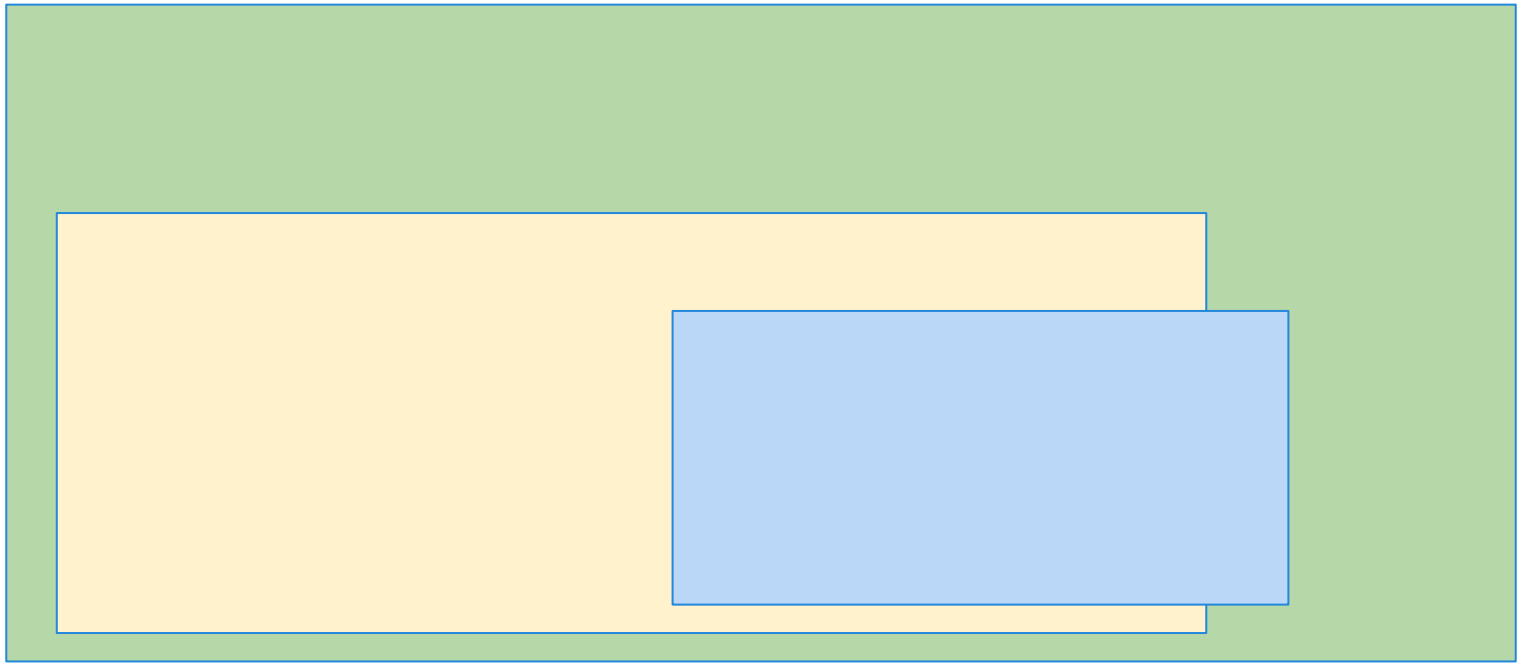


Verificación y Validación de Software

Análisis Dinámico

Dynamic Analysis

Sea *Sys* mi objeto de test

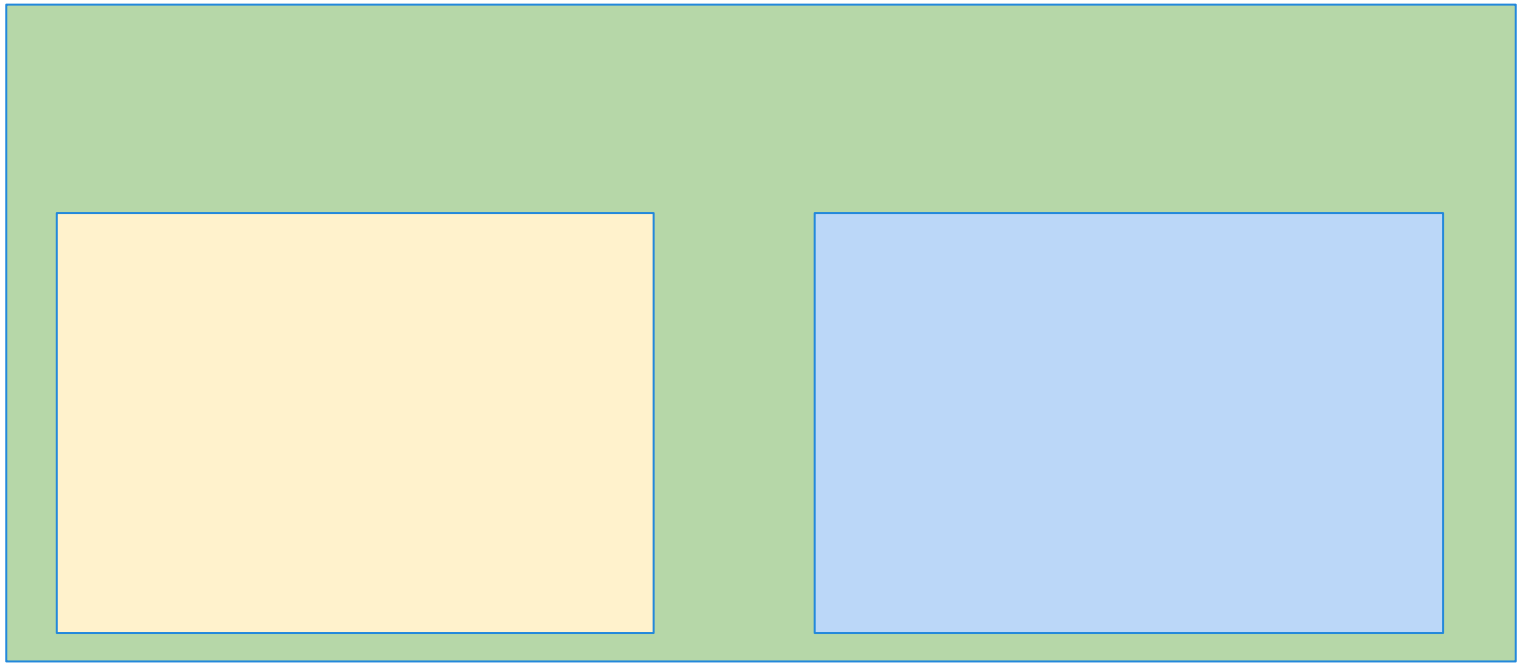


Verificación y Validación de Software

Análisis Dinámico

Dynamic Analysis

Sea *Sys* mi objeto de test

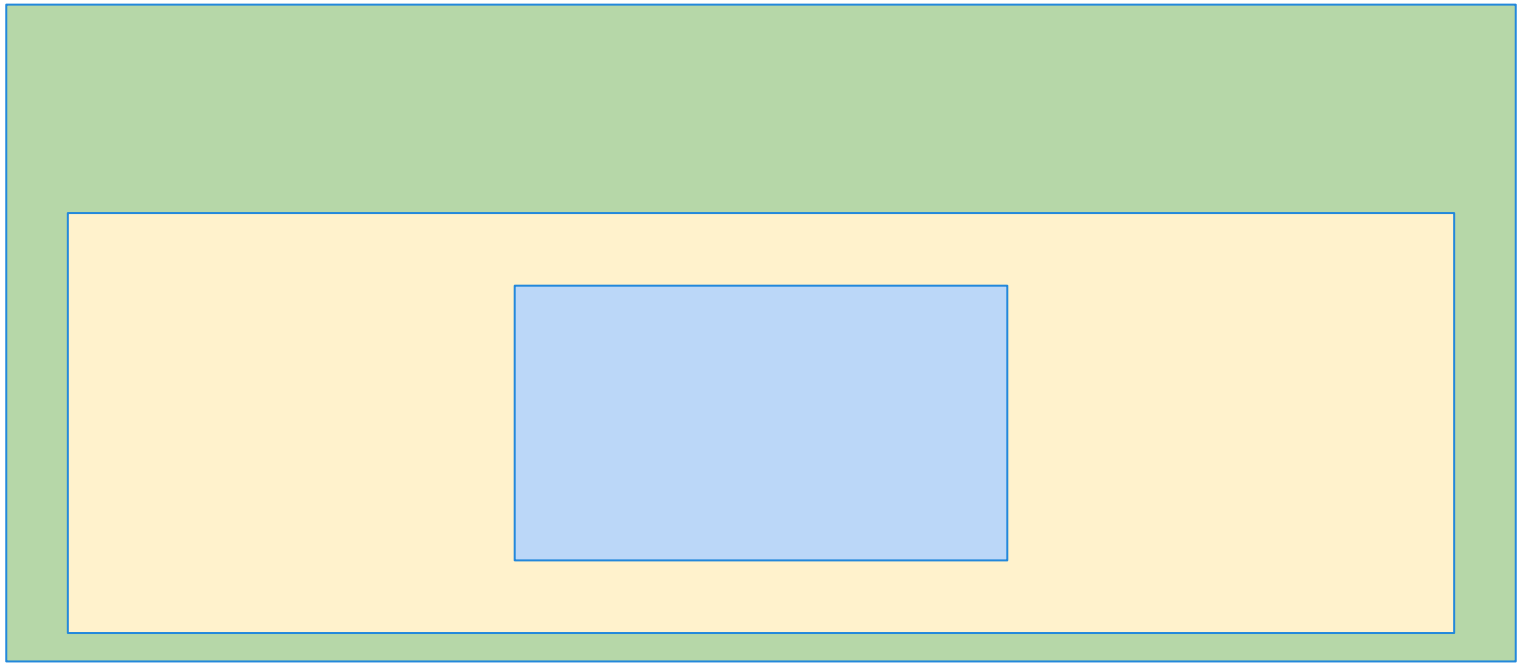


Verificación y Validación de Software

Análisis Dinámico

Dynamic Analysis

Sea *Sys* mi objeto de test



Verificación y Validación de Software

Análisis Dinámico

Dynamic Analysis

U: Universal, M: Usado por el Usuario, T: Usado para Testing

Verificación y Validación de Software

Análisis Dinámico

Dynamic Analysis

U: Universal, M: Usado por el Usuario, T: Usado para Testing

$$T < M < U$$

Verificación y Validación de Software

Análisis Dinámico

Dynamic Analysis

U: Universal, M: Usado por el Usuario, T: Usado para Testing

$$T < M < U$$

Idealmente, sería mucho mejor lograr T = M que T = U. Aunque T = U nos asegura la cobertura de M, sería más eficiente ir sólo por T = M.

Verificación y Validación de Software

Verificación en Ejecución

Runtime Verification

Verificación y Validación de Software

Verificación en Ejecución

Runtime Verification

Como no podemos testear el 100% de los caminos, entradas, posibilidades de la aplicación, nos dedicamos a seleccionar (inteligentemente) los caminos, entradas posibilidades más significativos. Pero...

Verificación y Validación de Software

Verificación en Ejecución

Runtime Verification

¿Por qué no seguir testeando mientras la aplicación está ejecutando (deployment) y por qué no dejar que el usuario sea el tester mientras usa la aplicación?

Verificación y Validación de Software

Verificación en Ejecución

Runtime Verification

¿Por qué no seguir testeando mientras la aplicación está ejecutando (deployment) y por qué no dejar que el usuario sea el tester mientras usa la aplicación?

Esto suena bien, pero no deberíamos hacer esto a costo de complicar el código del sistema o sobrecargar los sistemas del usuario.

Verificación y Validación de Software

Verificación en Ejecución

Runtime Verification

¿Por qué no seguir testeando mientras la aplicación está ejecutando (deployment) y por qué no dejar que el usuario sea el tester mientras usa la aplicación?

Esto suena bien, pero no deberíamos hacer esto a costo de complicar el código del sistema o sobrecargar los sistemas del usuario.

Programación Orientada a Aspectos

Verificación y Validación de Software

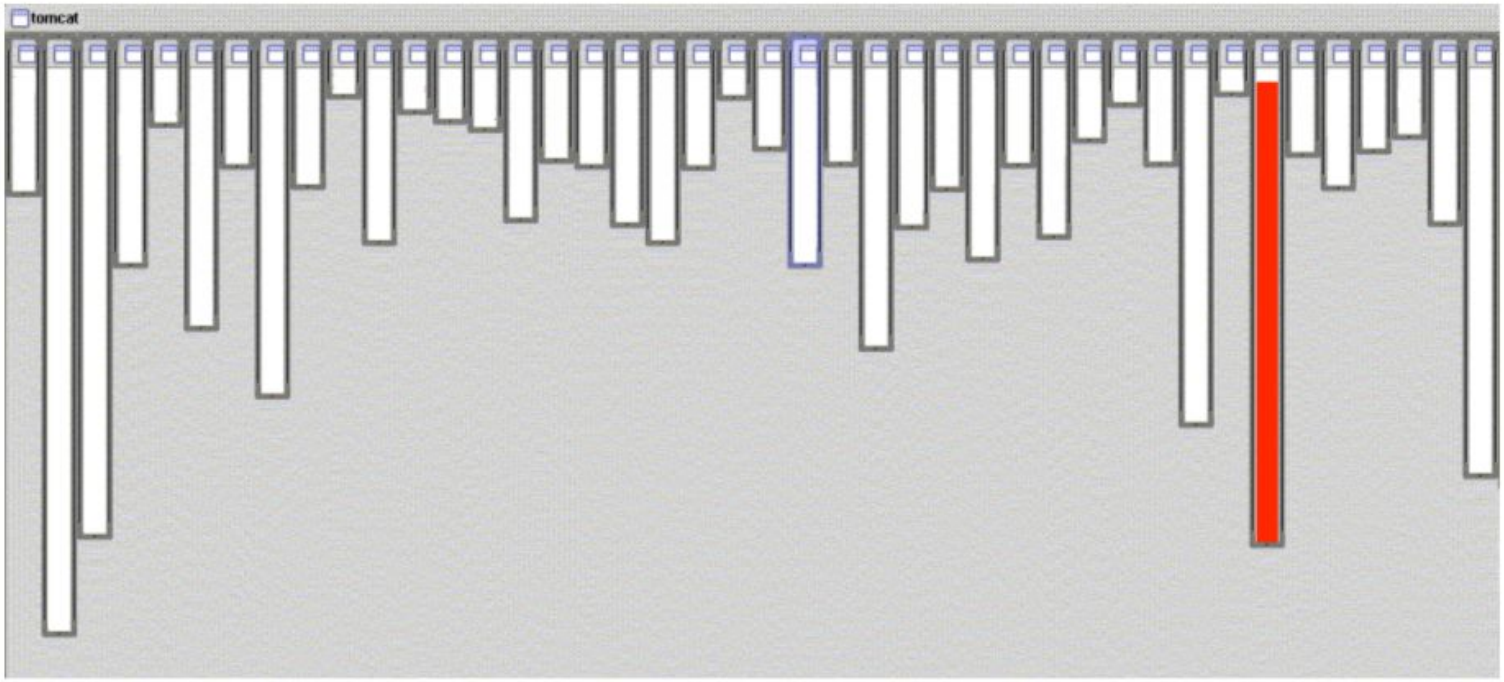
Programación Orientada a Aspectos

Aspect Oriented Programing

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

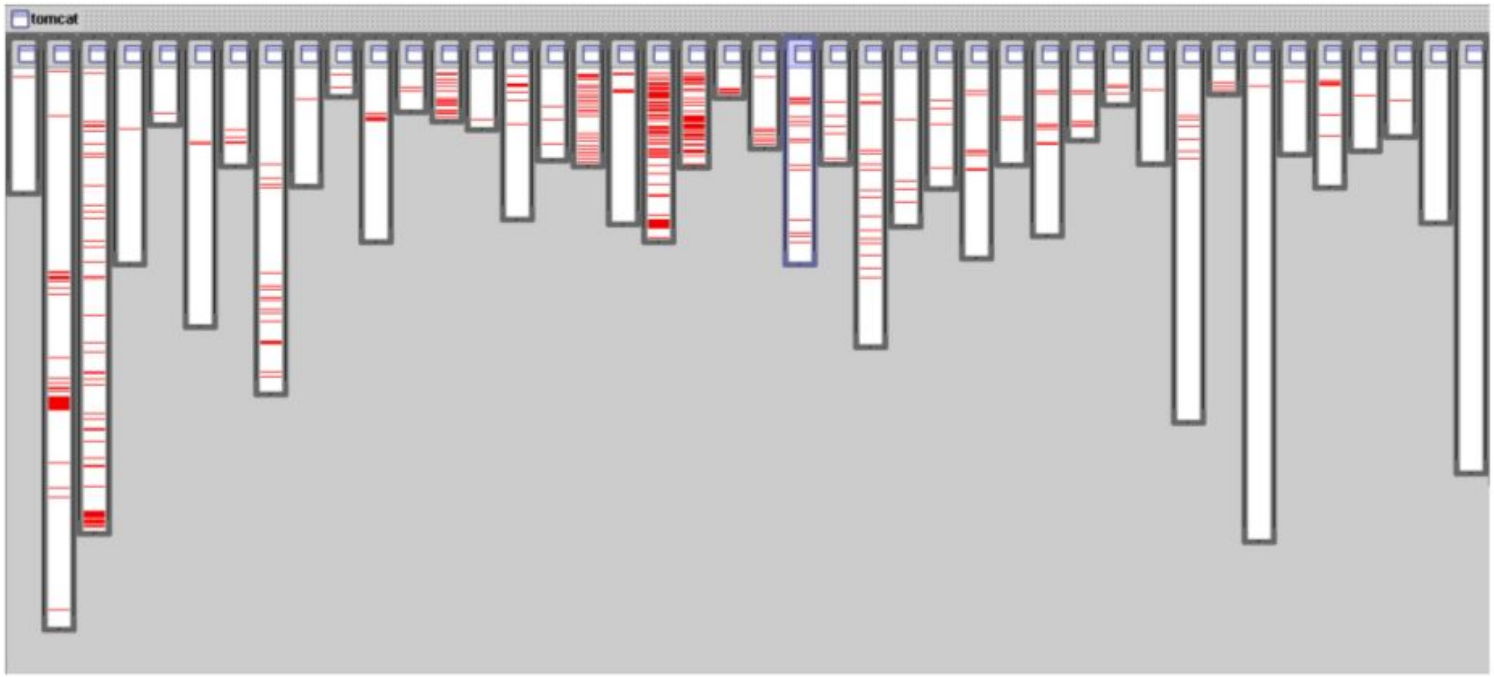


XML Parsing en org.apache.tomcat. Las líneas rojas representan LOC relevantes al XML Parsing. Un código bien modularizado.

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing



Logging en org.apache.tomcat. Las líneas rojas representan LOC que manejan logging. Un código mal modularizado.

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

Logging en org.apache.tomcat. Las líneas rojas representan LOC que manejan logging. Un código mal modularizado.

En realidad lo de “mal modularizado” no es culpa del equipo de desarrollo. No hay muchas alternativas.

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

Logging en org.apache.tomcat. Las líneas rojas representan LOC que manejan logging. Un código mal modularizado.

Imagen este requerimiento: “Cada vez que alguien llama a “.toString” se tiene que registrar en una base de datos la hora del llamado”.

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

Logging en org.apache.tomcat. Las líneas rojas representan LOC que manejan logging. Un código mal modularizado.

Imagen este requerimiento: “Cada vez que alguien llama a “.toString” se tiene que registrar en una base de datos la hora del llamado”.

No hay forma elegante de hacerlo… hasta ahora.

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

Imagen este requerimiento: “Cada vez que alguien llama a “.toString” se tiene que registrar en una base de datos la hora del llamado”.

Este requerimiento es un ejemplo de lo que se llama “cross-cutting concerns”. Estos “concerns”, o aspectos, son transversales a nuestra jerarquía de clases, lo que nos obliga a implementar soluciones poco elegantes.

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

La Programación Orientada a Aspectos (AOP) es una forma diferente de modularizar un programa. Cada aspecto se maneja en un único lugar.

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

La Programación Orientada a Aspectos (AOP) es una forma diferente de modularizar un programa. Cada aspecto se maneja en un único lugar.

El uso que vamos a ver acá de AOP es exclusivo para monitoreo y testing de aplicaciones. Se puede utilizar AOP como un paradigma completo de programación.

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

Ejemplos de aspectos son:

- Logging
- Verificación
- Políticas de Comportamiento
- Seguridad
- Profiling
- Manejo de Memoria
- ...

Verificación y Validación de Software

AspectJ

AspectJ

Verificación y Validación de Software

AspectJ

AspectJ

- Lanzado en 1998 en Xerox PARC
- Es una extensión de Java
- El compilador es gratuito y open source
- Buena relación con Eclipse y otras IDEs
- Genera archivos .class compatibles con cualquier JVM

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

Imagen este requerimiento: “Cada vez que alguien llama a “.toString” se tiene que registrar en una base de datos la hora del llamado”.

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

Imagen este requerimiento: “Cada vez que alguien llama a “.toString” se tiene que registrar en una base de datos la hora del llamado”.

Código Java	Pseudo AspectJ

Vector[Integer] a = new…
…
a.toString( );
Avion v = new …
…
v.toString( );
	cuando toString( ) es llamado {
	  logInDB(DateTime.Now( ));
	}

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

- join points
  - Puntos en un programa Java
- pointcut
  - Es quien “conecta” con los join points
- advice
  - modificadores sobre los join points y pointcut
- aspect
  - código, viejo y querido código.

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

- join points
  - Puntos en un programa Java
- pointcut
  - Es quien “conecta” con los join points
- advice
  - modificadores sobre los join points y pointcut + código
- aspect
  - conjunto de advices

Estos tres conceptos aparecen en AspectJ

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

En Java están los join points. En AspectJ están los pointcuts, que son un conjunto de join points.

El advice se define como en kind x pointcut x code, donde kind puede ser {before, after, around}. Finalmente un aspect es una lista de advices.

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

```java
public class Power {
    int balance;

    void deposit(int amount) {
        balance = balance + amount;
    }

    boolean withdraw(int amount) {
        if(balance - amount > 0) {
            balance = balance - amount;
            return true;
        } else return false;
    }
}
```

```java
public class Logger {
    private PrintStream stream;

    Logger() {
        //crea stream
    }

    void log(String message) {
        stream.println(message);
    }

}
```

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

Clase Power y Logger en la forma tradicional OO

```
public class Power2 {
    int balance;
    Logger logger = new Logger();

    void deposit(int amount) {
        logger.log("deposit amount: " + amount);
        balance = balance + amount;
    }

    boolean withdraw(int amount) {
        logger.log("withdraw amount: " + amount);
        if(balance - amount > 0) {
            balance = balance - amount;
            return true;
        } else return false;
    }
}
```

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

Clase Logger en AspectJ. Power en Java.

```java
public class Power {
    int balance;

    void deposit(int amount) {
        balance = balance + amount;
    }

    boolean withdraw(int amount) {
        if(balance - amount > 0) {
            balance = balance - amount;
            return true;
        } else return false;
    }
}
```

```java
public aspect Logging {
    Logger logger = new Logger();

    before(int amount):
        call(void Power.deposit(int)) && args(amount) {
            logger.log("deposit amount: " + amount);
    }

    before(int amount):
        call(boolean Power.withdraw(int)) && args(amount) {
            logger.log("withdraw amount: " + amount);
    }
}
```

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

```
public aspect Logging {
    Logger logger = new Logger();

    before(int amount):
        call(void Power.deposit(int)) && args(amount) {
            logger.log("deposit amount: " + amount);
    }

    before(int amount):
        call(boolean Power.withdraw(int)) && args(amount) {
            logger.log("withdraw amount: " + amount);
    }
}
```

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

Un poincut puede o no machear con uno o varios join points. Además, a través del pointcut se puede extraer valores del join point.

```
call(void Power.deposit(int))
```

Este es un pointcut que va a machear con métodos que se llamen deposit, en la clase Power, que tomen como parámetro un entero y no retornen valor.

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

Un poincut puede o no machear con uno o varios join points. Además, a través del pointcut se puede extraer valores del join point.

```
before(int amount):
    call(void Power.deposit(int)) && args(amount) {
        logger.log("deposit amount: " + amount);
}
```

Se declara una variable amount y se la liga al parámetro de deposit. Haciendo esto se puede utilizar la variable dentro del código.

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

Se pueden definir pointcuts propios.

```
pointcut powerChange(Power power) :
    (call(* deposit(..)) || call(* withdraw(..)))
    && target(power);

after(Power power): powerChange(power) {
    logger.log("balance = " + power.balance);
}
```

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

Signature with wildcards	Description
`* void MyClass.foo(int, float)void MyClass.foo(int, float)`	Captures join points on a method regardless of the modifier. Can also be achieved by leaving out the visibility entirely.

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

Signature with wildcards	Description
`* * MyClass.foo(int, float)*` `MyClass.foo(int, float)`	Captures join points on a method regardless of the modifier or return type.
`* * *.foo(int,float)* *` `foo(int,float)`	Captures join points on a method regardless of the modifier, return type, or class.
`* * *.*(int,float)`	Captures join points on a method regardless of the modifier, return type, class, or method.

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

`* * *.*(*,float)`	Captures join points on a method regardless of the modifier, return type, class, or method where the parameters include anything followed by a float.
`* * *.*(*,..)`	Captures join points on a method regardless of the modifier, return type, class, or method where the parameters include at least a single value followed by any number of parameters.
`* * *.*(..)* *(..)`	Captures join points on a method regardless of the modifier, return type, class, or method where there are any number of parameters.

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

`* mypackage..*.*(..)`	Captures join points on any method within the `mypackage` package and subpackages.
`* MyClass+.*(..)`	Captures join points on any method on the `MyClass` class and any sub-classes.

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

El chequeo de contratos en Java es un ambiente donde se puede aplicar AOP.

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

```
    boolean withdraw(int amount) {
        if(balance - amount > 0) {
            balance = balance - amount;
            return true;
        } else return false;
    }
}
```

```
public aspect WithDrawPreCond {
    final int MIN_BALANCE = 50;

    before(Power power, int amount):
        call(boolean Power.whithdraw(int)) &&
        target(power) && args(amount) {
        if(power.balance - amount > MIN_BALANCE) {
            //no verifica precondición
        }
    }
}
```

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

```
    boolean withdraw(int amount) {
        if(balance - amount > 0) {
            balance = balance - amount;
            return true;
        } else return false;
    }
}
    /* postcondición
     * result == (old(balance) - amount) >= 0 &&
     * balance == (result ? old(balance) - amount: old(balance))
     */
```

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

```
    boolean withdraw(int amount) {
        if(balance - amount > 0) {
            balance = balance - amount;
            return true;
        } else return false;
    }
}

    /* postcondición
     * result == (old(bala
     * balance == (result
     */
```

```
public aspect WithDrawPostCond {
    int old_balance;

    before(Power power):
        call(boolean Power.withdraw(int)) && target(power) {
        old_balance = power.balance;
    }

    after(Power power, int amount) returning(boolean changed):
        call(boolean Power.withdraw(int)) &&
        target(power) && args(amount) {
        if ( (old_balance - amount) >= 0 &&
                power.balance == (changed ? old_balance-amount : old_balance)) {
            //verifica postcondición
        }
    }
}
```

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

```
    boolean withdraw(int amount) {
        if(balance - amount > 0) {
            balance = balance - amount;
            return true;
        } else return false;
    }
}
    /* postcondición
     * result == (old(balance) - amount) >= 0 &&
     * balance == (result ? old(balance) - amount: old(ba
     */
```

```
public aspect WithDrawPostCondAround {
    int old_balance;

    boolean around(Power power, int amount):
        call(boolean Power.withdraw(int)) &&
        target(power) && args(amount) {
        old_balance = power.balance;
        boolean changed = proceed(power, amount);
        //verificación de postcondición
        return changed;
    }
}
```

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

¿Qué hace ésto?

```
public aspect LogExceptions {
    Logger logger = new Logger();

    after() throwing (Error e): call(* *(..)) {
        logger.log("exception throw " + thisJoinPoint + ":" + e);
    }
}
```

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

¿Qué hace ésto?

```
public aspect LogExceptions {
    Logger logger = new Logger();

    after() throwing (Error e): call(* *(..)) {
        logger.log("exception throw " + thisJoinPoint + ":" + e);
    }
}
```

```
exception thrown call(boolean core.Power.withdraw(int)):java.lang.AssertionError
```

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

```
public class Power {
    int balance;

    //invariant balance => 500;

    void deposit(int amount) {
        //..
    }


    boolean withdraw(int amount) {
        //..
        return true;
    }
}
```

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

```java
public class Power {
    int balance;

    //invariant balance => 500;

    void deposit(int amount) {
        //..
    }


    boolean withdraw(int amount) {
        //..
        return true;
    }
}
```

```java
public aspect Invariant {
    boolean invariant(int balance) {
        return balance >= 500;
    }

    pointcut write(int balance) :
        set(int Power.balance) && args(balance);

    before(int balance) : write(balance) {
        if(!invariant(balance)) {
            System.out.println("Violación de Invariante");
        }
    }
}
```

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

```
declare warning :
    within(com.xerox.printing..*)
    && set(!public * *) && !withincode(* set*(..))
    : "writing field outside setter" ;
```

Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing

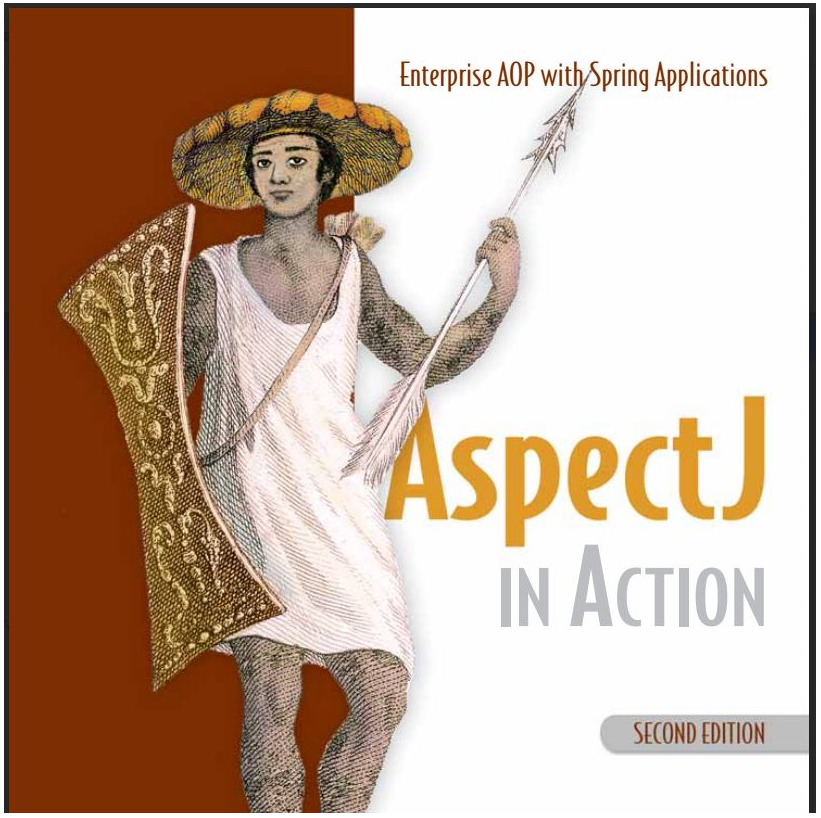
Se declara un “warning” cada vez que se modifica un atributo no público desde un método que no es el set

```
declare warning :
    within(com.xerox.printing..*)
    && set(!public * *) && !withincode(* set*(..))
    : "writing field outside setter" ;
```

Verificación y Validación de Software

Programación Orientada a Aspectos

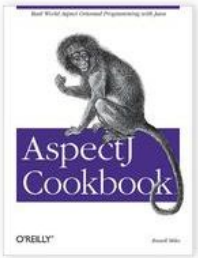
Aspect Oriented Programing



Verificación y Validación de Software

Programación Orientada a Aspectos

Aspect Oriented Programing



AspectJ Cookbook

by Russ Miles

Publisher: O'Reilly Media, Inc.

Release Date: December 2004

ISBN: 9780596006549

Topic: Software Development

Start reading now

View table of contents

Purchase this book Purchase this book

Book Description

When Object Oriented programming (OO) first appeared, it was a revelation. OO gave developers the ability to create software that was more flexible and robust, but as time went on and applications became more sophisticated, too, certain areas of "traditional" OO architectures were found wanting. Aspect-oriented programming (AOP) addresses those issues by extending the OO approach even further. Many developers are interested in AOP--especially in AspectJ, the open source extension of the Java programming language that explicitly supports the AOP approach. Yet, although AspectJ is included with Eclipse, the increasingly popular open source IDE for Java, finding a practical and non-theoretical way to learn this language and other AOP tools and techniques has been a real problem. Until now. The *AspectJ Cookbook* offers a hands-on solution--in fact, several--with a wide variety of code recipes for solving day-to-day design and coding problems using AOP's unique approach. AOP allows the global properties of a program to determine how it's compiled into an executable program. Before AOP, important program design decisions were difficult to capture in actual code. Instead, the implementation of those design decisions--known as "aspects"--were scattered throughout, resulting in "tangled" code that was hard to develop and maintain. AOP has been compared to the manufacturing of cloth, in which threads are automatically interwoven. Without AOP, programmers must stitch the threads by hand. The *AspectJ Cookbook* shows readers why, and how, common Java development problems can be solved by using AOP techniques. With our popular problem-solution-discussion format, the book presents real world examples to demonstrate that AOP is more than just a concept; it's a development process that will benefit users in an immediate and visible manner. If you're interested in how AOP is changing the way software is developed, and how you can use AspectJ to make code more modular, easier to develop, maintain, evolve and deploy, this is the book that really delivers.

https://www.safaribooksonline.com/library/view/aspectj-cookbook/0596006543/

RESUMEN

Verificación y Validación de Software

Testing Dinámico

Dynamic Testing

El Testing Dinámico se puede dividir en cuatro categorías, con varias zonas grises en el medio.

- Caja Negra
- Caja Blanca
- Basado en Experiencia
- Basado en Defectos
- Análisis Dinámico

Verificación y Validación de Software

Ingeniería en Sistemas de Información

Departamento de Ciencias e Ingeniería de la Computación

2017

8