

Runtime Verification

From theory to practice and back

Gordon J. Pace
(joint course design with Christian Colombo)
University of Malta

July 2013

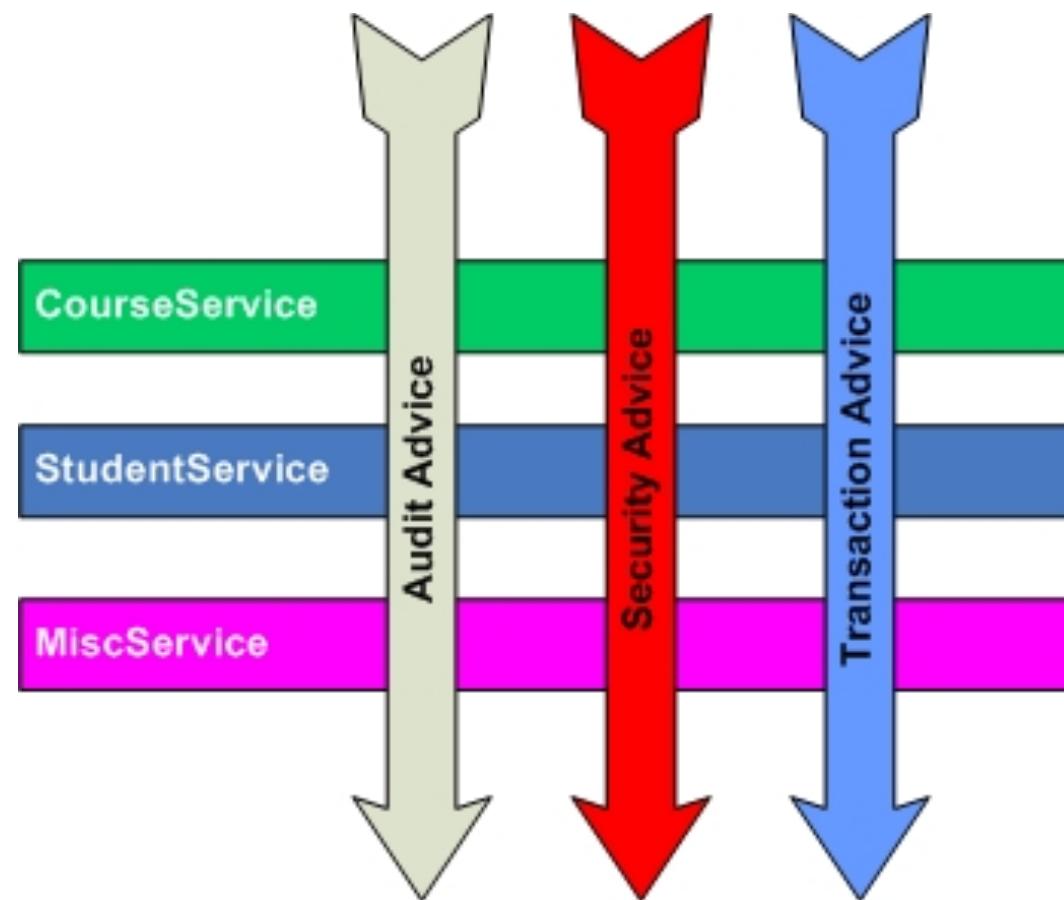
Part IIIb

Separation of Concerns

Some Observations (A Reminder)

- Adding the properties into the system code makes it difficult to separate: where does the property end and the system start.
- Some properties are not simply assertions, and may require additional logic – the code implementing this logic is also mixed with the system.
- Changes to the properties result in direct changes in the project code.
- If we want to change the mode of verification (e.g. produce logs to check offline), it will require reengineering the whole effort.

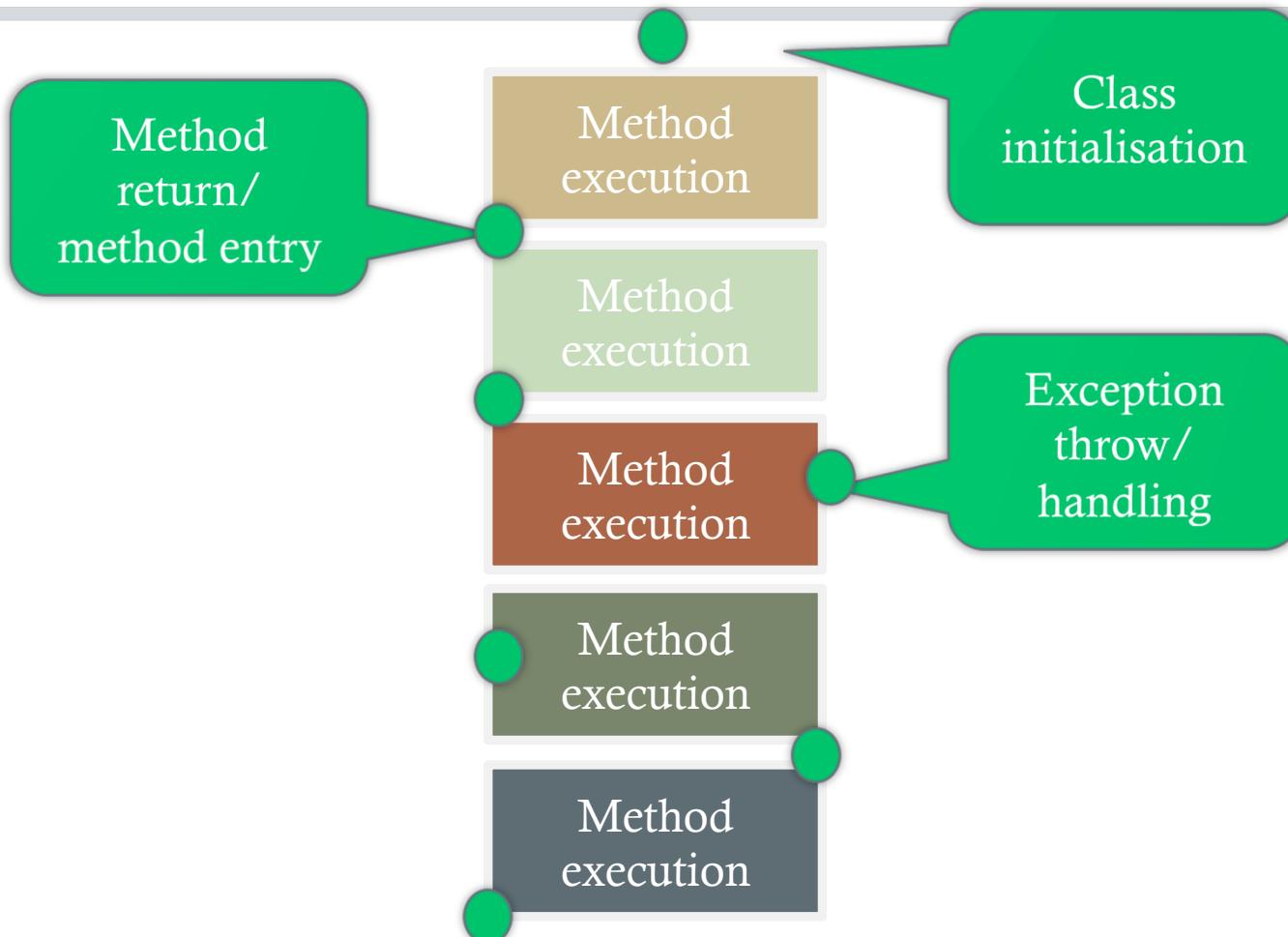
Programming Concerns



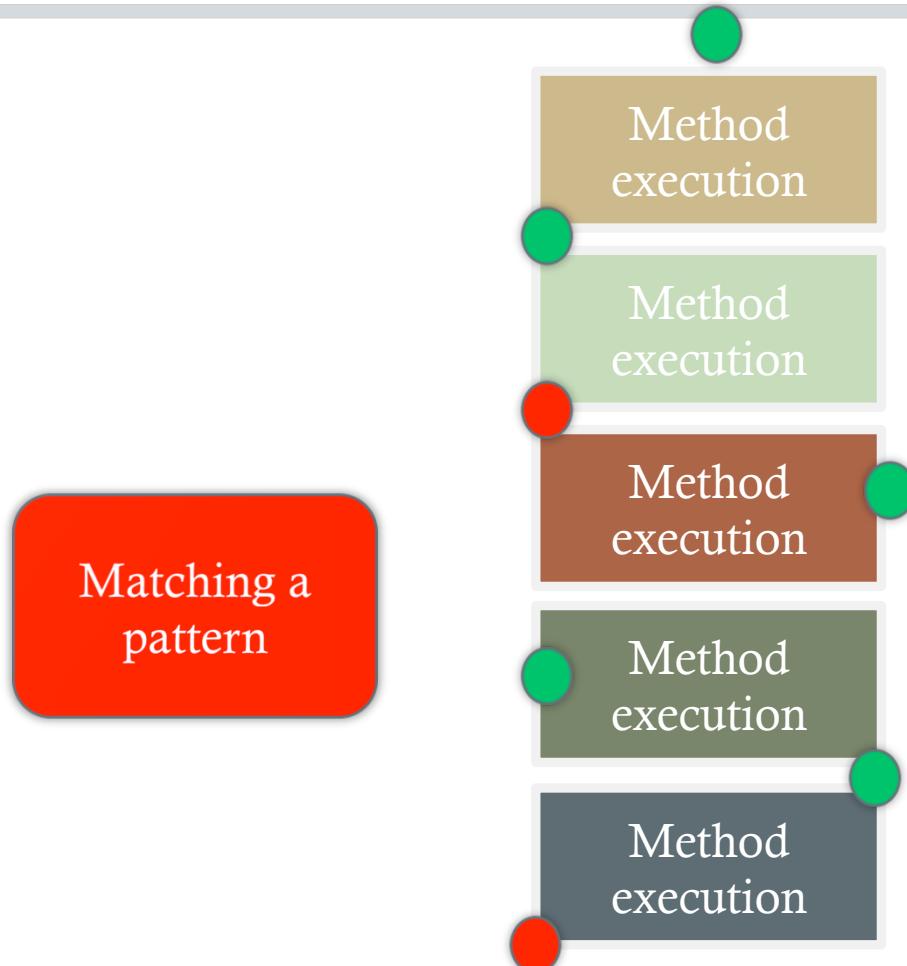
Aspect-Oriented Programming

- Aspect-Oriented Programming (AOP) provides a way of addressing cross-cutting concerns in code.
- Provides ways of linking with points in the code.
 - These positions are called *joinpoints*
 - Typical support for joinpoints such as:
 - Method and constructor execution
 - Method and constructor call
 - Field get and set
 - Exception handler execution
 - Static and dynamic initialization

Joinpoints



Pointcuts



Aspect-Oriented Programming

- An AOP script consists of a list of *pointcut* and *advice* pairs.
 - **Pointcut:** A rule (potentially) matching a number of joinpoints e.g. “just before method *login* is called”.
 - **Advice:** Code to be executed when the program reaches the related pointcut.

Aspect-Oriented Programming

- An AOP script consists of a list of *pointcut* and *advice* pairs.
 - **Pointcut:** A rule (potentially) matching a number of joinpoints e.g. “just before method *login* is called”.
 - **Advice:** Code to be executed when the program reaches the related pointcut.
- **Examples:**

```
before (): (* *.login(..)) { log.add("Logging in"); }
```

```
after (): (* *.closeSession(..)) { resources.release(); }
```

Aspect-Oriented Programming

- An AOP script consists of a list of *pointcut* and *advice* pairs.
 - **Pointcut:** joinpoints
 - **Advice:** C reaches the target pointcut.
- **Examples:**

```
before (): (* *.login(..)) { log.add("Logging in"); }  
after (): (* *.closeSession(..)) { resources.release(); }
```

Pointcuts

ing a number of
gin is called".

the program

Aspect-Oriented Programming

- An AOP script consists of a list of *pointcut* and *advice* pairs.
 - **Pointcut:** A pointcut defines the joinpoints e.g. methods, fields, constructor etc. that an advice can be applied to.
 - **Advice:** Code that is executed when the program reaches the related pointcut.
- **Examples:**

```
before (): (* *.login(..)) { log.add("Logging in"); }
```

```
after (): (* *.closeSession(..)) { resources.release(); }
```

AOP for RV

- AOP provides us with a perfect way of separating the writing of verification code from that of the system.

AOP for RV

- AOP provides us with a perfect way of separating the writing of verification code from that of the system.
- **Example:** *Logging out can only occur while logged in.*
A Verification class is defined as before together with the following aspect code:

```
before (): (* *.login(..)) { Verification.setLoggedIn(); }
before (): (* *.logout(..)) {
    Verification.assertion(Verification.isLoggedIn(),"ERR");
    Verification.setLoggedOut();
}
```

AspectJ

- AspectJ is an AOP tool for Java.
- Built as an extension to Java, allowing for general purpose aspect programming.
- Good support in Eclipse (and other IDEs/editors) – creating an AspectJ project allows for aspects to be added (in the form of .aj files) which are compiled together with the system.
- Here we will show AspectJ bare necessities to be able to use AOP for runtime verification...

Programming in AspectJ

- The anatomy of an AspectJ aspect declaration through a *HelloWorld* example:

```
public aspect Properties {  
    before (): call (* *.move(..)) {  
        System.out.println("Hello world");  
    }  
  
    after (): call (* *.move(..)) {  
        System.out.println("Hello world");  
    }  
}
```

Programming in AspectJ

Before a method
call...

- The anatomy
HelloWorld exam

```
public aspect Properties {  
    before (): call (* *.move(..)) {  
        System.out.println("before move");  
    }  
    after (): call (* *.move(..)) {  
        System.out.println("Hello world");  
    }  
}
```

After a method
call...

Programming in AspectJ

Access modifiers
and return type (or
* for *anything*)

- The anatomy
HelloWorld example

```
public aspect Properties {  
    before (): call (* *.move(..)) {  
        System.out.println("Hello world");  
    }  
  
    after (): call (* *.move(..)) {  
        System.out.println("Hello world");  
    }  
}
```

Programming in AspectJ

Class name (may use * to indicate *any class*) – may

- The anatomy also include packages
HelloWorld example

```
public aspect Properties {  
    before (): call (* *.move(..)) {  
        System.out.println("Hello world");  
    }  
  
    after (): call (* *.move(..)) {  
        System.out.println("Hello world");  
    }  
}
```

Programming in AspectJ

Method name (may
use * to indicate *any*
method)

- The anatomy of an *HelloWorld* example:

```
public aspect Properties {
    before (): call (* *.move(..)) {
        System.out.println("Hello world");
    }

    after (): call (* *.move(..)) {
        System.out.println("Hello world");
    }
}
```

Programming in AspectJ

Parameters of the
method (use .. to

- The anatomy signify *any*) declaration through a *HelloWorld* example:

```
public aspect Properties {  
    before (): call (* *.move(..)) {  
        System.out.println("Hello world");  
    }  
  
    after (): call (* *.move(..)) {  
        System.out.println("Hello world");  
    }  
}
```

Programming in AspectJ

The advice to be
executed

- The anatomy of a *HelloWorld* example

```
public aspect Properties {  
    before () call (* *.move(..)) {  
        System.out.println("Hello world");  
    }  
  
    after (): call (* *.move(..)) {  
        System.out.println("Hello world");  
    }  
}
```

Programming in AspectJ

- The *target* is the object on which the method captured is called.
- It can be captured as follows:

```
before (Shape x):  
    call (* Shape.move(..)) &&  
    target(x)  
    {  
        System.out.println("Hello"+x.toString());  
    }
```

Programming in AspectJ

- Capturing the return value:

```
after () returning(Position p):  
    call (* *.move(..)) {  
        System.out.println("Hello "+p.toString());  
    }
```

Programming in AspectJ

- Capturing the parameters:

```
before (double dx, double dy):  
    call (* *.move(..)) &&  
    args(dx,dy)  
{  
    System.out.println("Move "+dx+","+dy);  
}
```

Programming in AspectJ

- Accessing target, method parameters and its return value:

```
after
    (Shape s, double dx, double dy)
    returning (Position p):
        call (* *.move(..)) &&
        target(s) &&
        args(dx,dy)
    {
        code
    }
```

Exercises

Add the properties to
FiTS using AspectJ.

Exercises

Add the properties to
FiTS using

Run the scenarios you were
given with the code to
check that they run as
expected.