

Aspect Oriented Fault Injection Testing

Richard Steele



Improve your Tests with FIT

- **Unit testing is usually limited to functional and boundary condition cases**
- **It's difficult to test your code when something truly goes wrong**
- ***Fault Injection (or Insertion) Testing* is a time-honored way of running your code under faulty conditions**



Traditional FIT

- **FIT has long history in hardware and embedded systems**
- **Injecting faults into application typically shows this hardware legacy**
 - "Pull the drive cable 4.5 seconds after pressing OK."



FIT for Software Systems

- **Hardware systems have natural "fault lines"**
 - hardware subsystems
 - hardware/software boundary
- **Software intensive systems also have fault lines between subsystems, modules, components, etc., but**
 - Many are hidden inside binary distributions
 - Many are inaccessible during testing



Injecting Software Faults: The Old Way

- **Embed conditional code**
 - Leaves faults in production code
 - Complicates code: the injected code gets in the way of the real purpose
- **Use IoC to inject faulty implementations**
 - Extra code to write and maintain
 - Can inject only in those places that support it
- **Faulty mock objects**
 - Similar problems to IoC
- **Step through code with debugger**
 - Tedious
 - Time consuming
 - Not reproducible



The New Way: Use Aspects!

- **Aspects can inject faults wherever needed**
- **You're not limited by the design of your application**
- **You don't have to change your design**
- **You can inject faults without the source code!**
- **You can inject faults into third party code**



Example: Reading Configuration File

```
public aspect ConfigurationFileReaderFaultInjector {  
    pointcut faultFrom():  
        call(public int java.io.FileReader.read(..));  
  
    pointcut faultTo():  
        within(com.eplus.Configuration.readConfigFile());  
  
    pointcut faultLine():  
        faultFrom() && faultTo();  
  
    before() throws IOException: faultLine() {  
        throw new java.io.IOException("Simulated error");  
    }  
}
```



Pointcuts Control Granularity

- **Inject fault into specific method:**
`within(com.eplus.Configuration.readConfigFile());`
- **Inject fault into all public methods:**
`within(com.eplus.Configuration.*());`
- **Or even into an entire package:**
`within(com.eplus.*());`



Advice Triggers Faults

- **Advice might *conditionally* trigger a fault**
- **Conditional triggers might use:**
 - Method parameter values
 - Return value of the normal execution of the injected operation
 - State of the injected class
 - State of the system
 - Nondeterministic variables
 - Any combination of the above



Example: Conditional Trigger

```
public aspect ParameterValueFaultInjector {
    pointcut faultFrom():
        call(public Report com.biz..ReportGenerator.gen(int));

    pointcut faultTo():
        within(com.foo..*);

    pointcut faultLine():
        faultFrom() && faultTo();

    Report around(int type): faultLine() && args(type) {
        if (type == Report.HTML) {
            // make fault happen here
        }
        return proceed(type);
    }
}
```



Advice Also Controls What Happens

- **Throw an exception**
- **Suppress or change an exception**
- **Modify one or more parameter values**
- **Modify the operation result**
- **Modify class state**
- **Modify system state**
- **Introduce delays via Thread.sleep()**
- **Any combination of the above**



Static FIT

- **Build-time weaving can be used**
- **Be careful not to release the version with the injected faults**
- **Some safeguards:**
 - **Build process should separate the production build artifacts from the FIT woven artifacts**
 - **Have all FIT aspects check for system property or other global setting**
 - **Enforced with abstract aspect**
 - **Consider having tool scan release for evidence of injected faults**
 - **Abstract aspect as marker**
 - **Inject marker interface**
 - **Add marker methods to "breadcrumb" interface**



Load Time Weaving

- **Dynamically apply faults using custom class loader, agent, or other load-time mechanism**
- **Injected faults live only if injected at run-time, little risk of introducing "into the wild"**



FIT Unit Testing

- **Consider adding FIT capabilities to unit testing frameworks**
 - JUnit
 - TestNG
- **For each test class, for each test method**
 - Load class
 - Inject faults using aspect(s)
 - Run test as usual



Other Directions

- **Test your unit tests**
 - Inject faults into application and see if your unit tests find them
- **Prepare for production**
 - Develop FMEA-based failure documentation
 - Staff training
- **Production failure post-mortem analysis**
 - Inject faults to try to reproduce field failure



Questions & Comments

Contact Info

Richard Steele

ePlus Consulting

rsteale@eplus.com

Whitepaper available upon request


