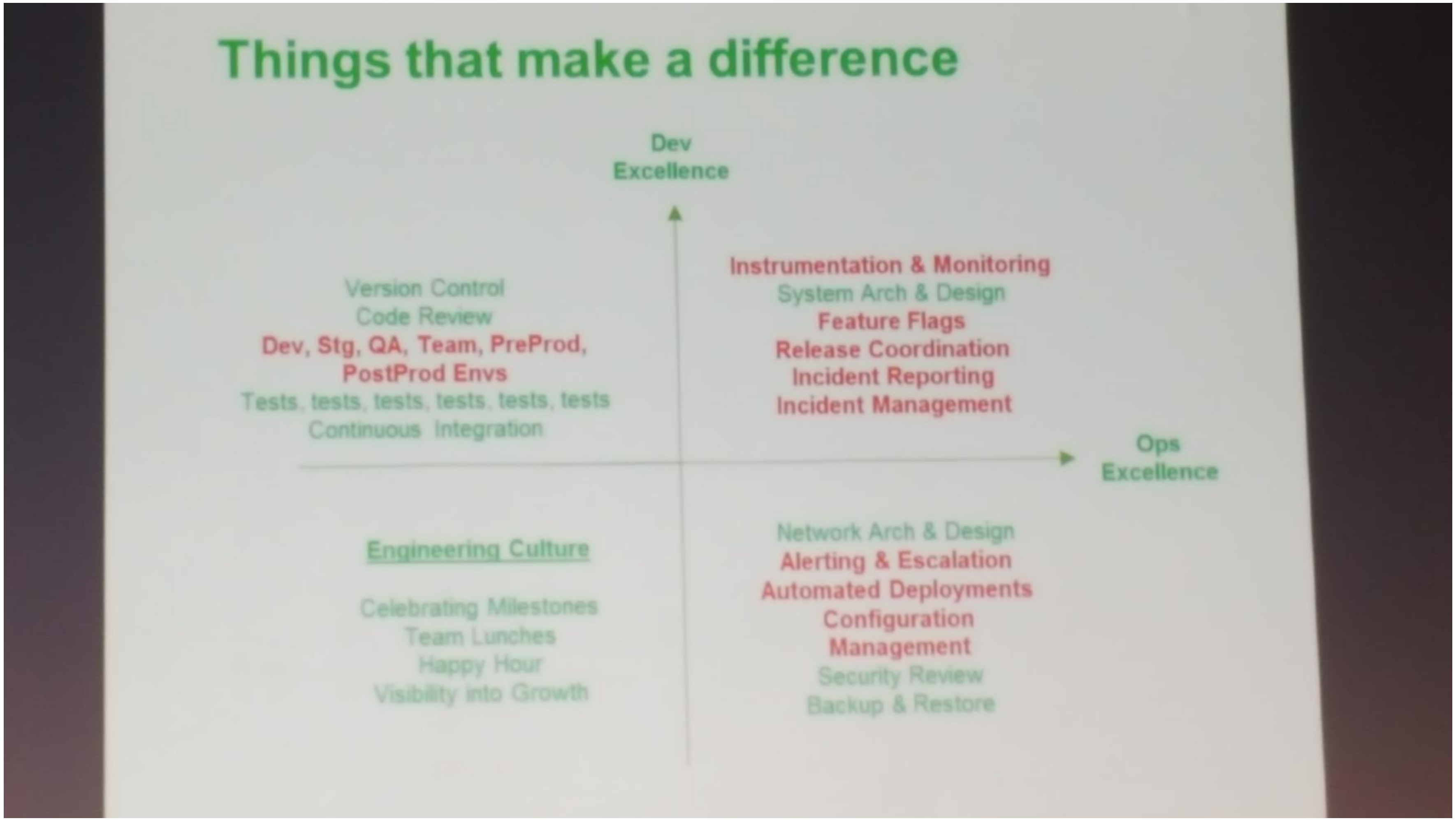
Programming Dynamic Features and Monitoring Distributed Software Systems

Hong-Linh Truong
Faculty of Informatics, TU Wien

hong-linh.truong@tuwien.ac.at
http://www.infosys.tuwien.ac.at/staff/truong
Twitter: @linhsolar



Things that make a difference

Dev
Excellence

Version Control
Code Review
Dev, Stg, QA, Team, PreProd,
PostProd Envs
Tests, tests, tests, tests, tests, tests
Continuous Integration

Instrumentation & Monitoring
System Arch & Design
Feature Flags
Release Coordination
Incident Reporting
Incident Management

Ops
Excellence

Engineering Culture

Celebrating Milestones
Team Lunches
Happy Hour
Visibility into Growth

Network Arch & Design
Alerting & Escalation
Automated Deployments
Configuration
Management
Security Review
Backup & Restore

Photo from “Lessons Learned From a Fast Growing Startup”, Arul Kumaravel, Vice President, Engineering, Grab at CloudAsia2016, Singapore

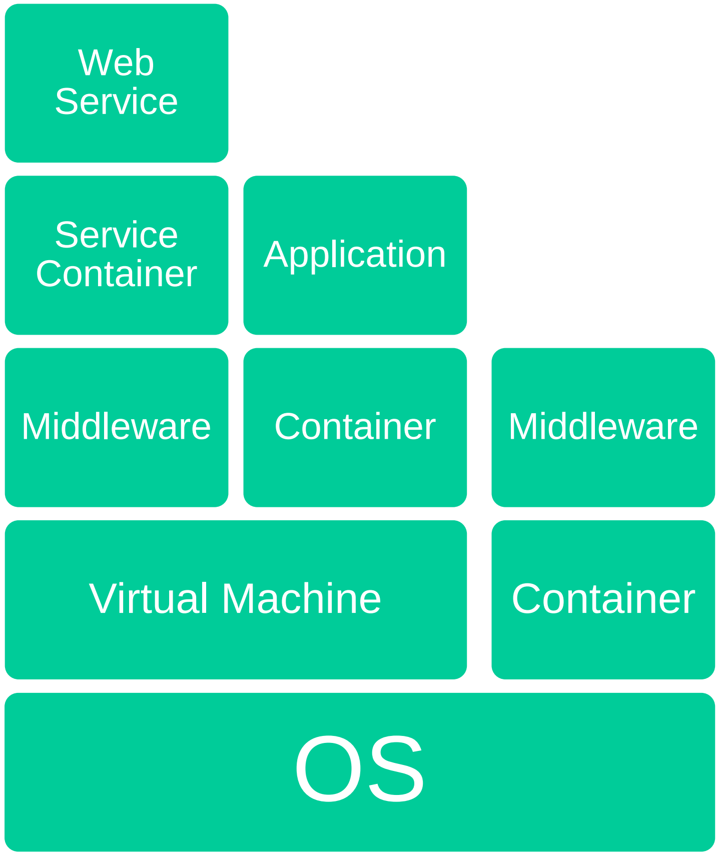
„In the context of computer programming, instrumentation refers to an ability to monitor or measure the level of a product's performance, to diagnose errors and to write trace information”
https://en.wikipedia.org/wiki/Instrumentation_%28computer_programming%29

“To monitor or monitoring generally means to be aware of the state of a system, to observe a situation for any changes which may occur over time, using a monitor or measuring device of some sort.”

https://en.wikipedia.org/wiki/Monitoring

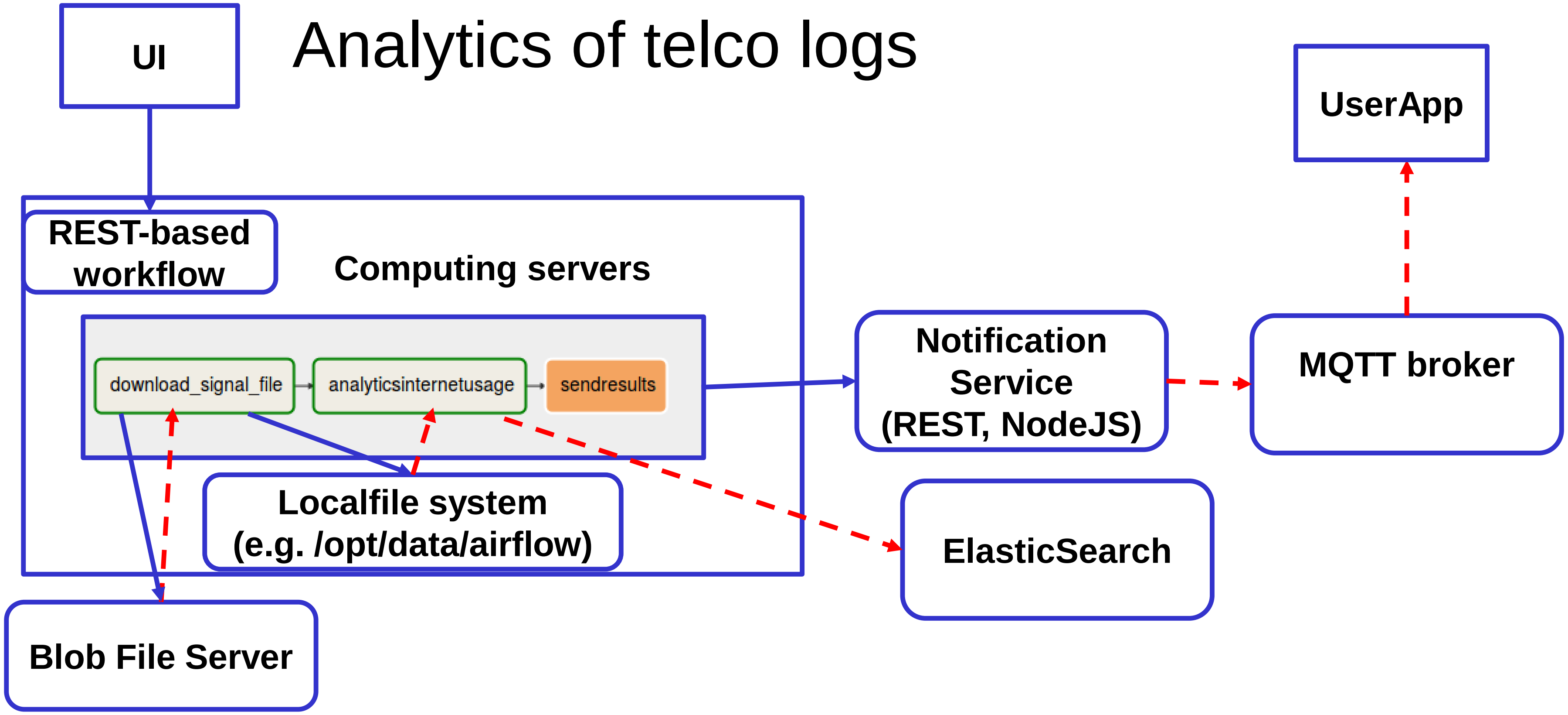
Programming dynamic features enable instrumentation and monitoring

Full stack monitoring



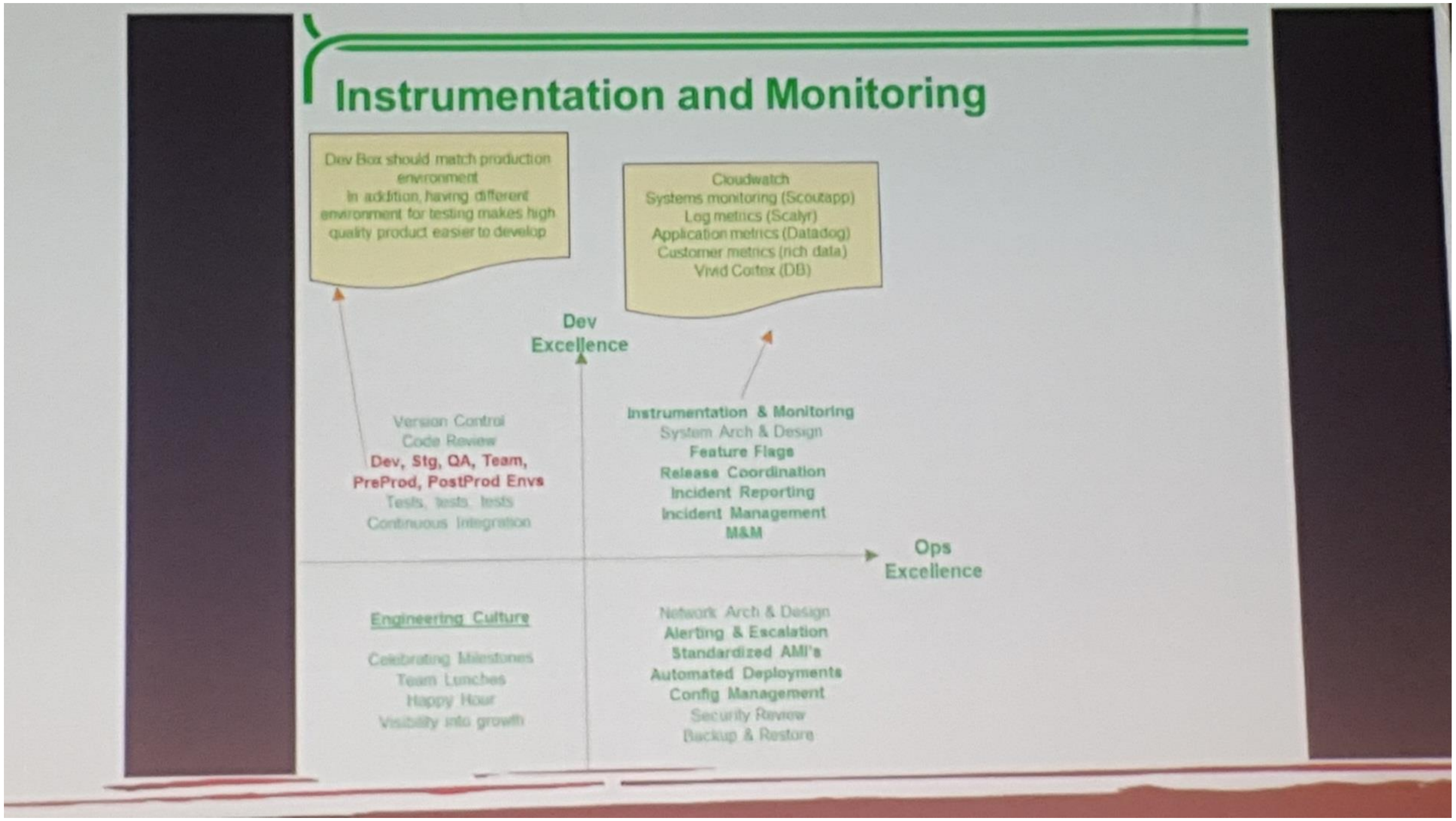
- You might need to monitor from OS to the application components
  - You might own or just rent them
- Artifacts: binary, runtime, source
- Monitoring functions about computation, data and network

Monitoring at the Large-scale



- Many distributed components across various enterprise boundaries
- Events/Measurement collection, Storage, Analytics and Visualization

Grab instrumentation and monitoring



Instrumentation and Monitoring

Dev Box should match production environment
In addition, having different environment for testing makes high quality product easier to develop

Cloudwatch
Systems monitoring (Scoutapp)
Log metrics (Scalyr)
Application metrics (Datadog)
Customer metrics (rich data)
Vivid Cortex (DB)

Dev Excellence

Version Control
Code Review
Dev, Stg, QA, Team, PreProd, PostProd Envs
Tests, tests, tests
Continuous Integration

Instrumentation & Monitoring
System Arch & Design
Feature Flags
Release Coordination
Incident Reporting
Incident Management
M&M

Ops Excellence

Engineering Culture

Celebrating Milestones
Team Lunches
Happy Hour
Visibility into growth

Network Arch & Design
Alerting & Escalation
Standardized AMI's
Automated Deployments
Config Management
Security Review
Backup & Restore

Photo from “Lessons Learned From a Fast Growing Startup”, Arul Kumaravel, Vice President, Engineering, Grab at CloudAsia2016, Singapore

Key techniques for today’s lecture

- Fundamentals for supporting dynamic features
  - Code inspection: Dynamic loading, Reflection, Dynamic proxy
  - Instrumentation and Program Analysis
  - Annotation
  - Aspect-oriented Programming
- Large-scale cloud native applications and systems
  - full-stack and large-scale system and application monitoring

DYNAMICITY NEEDS

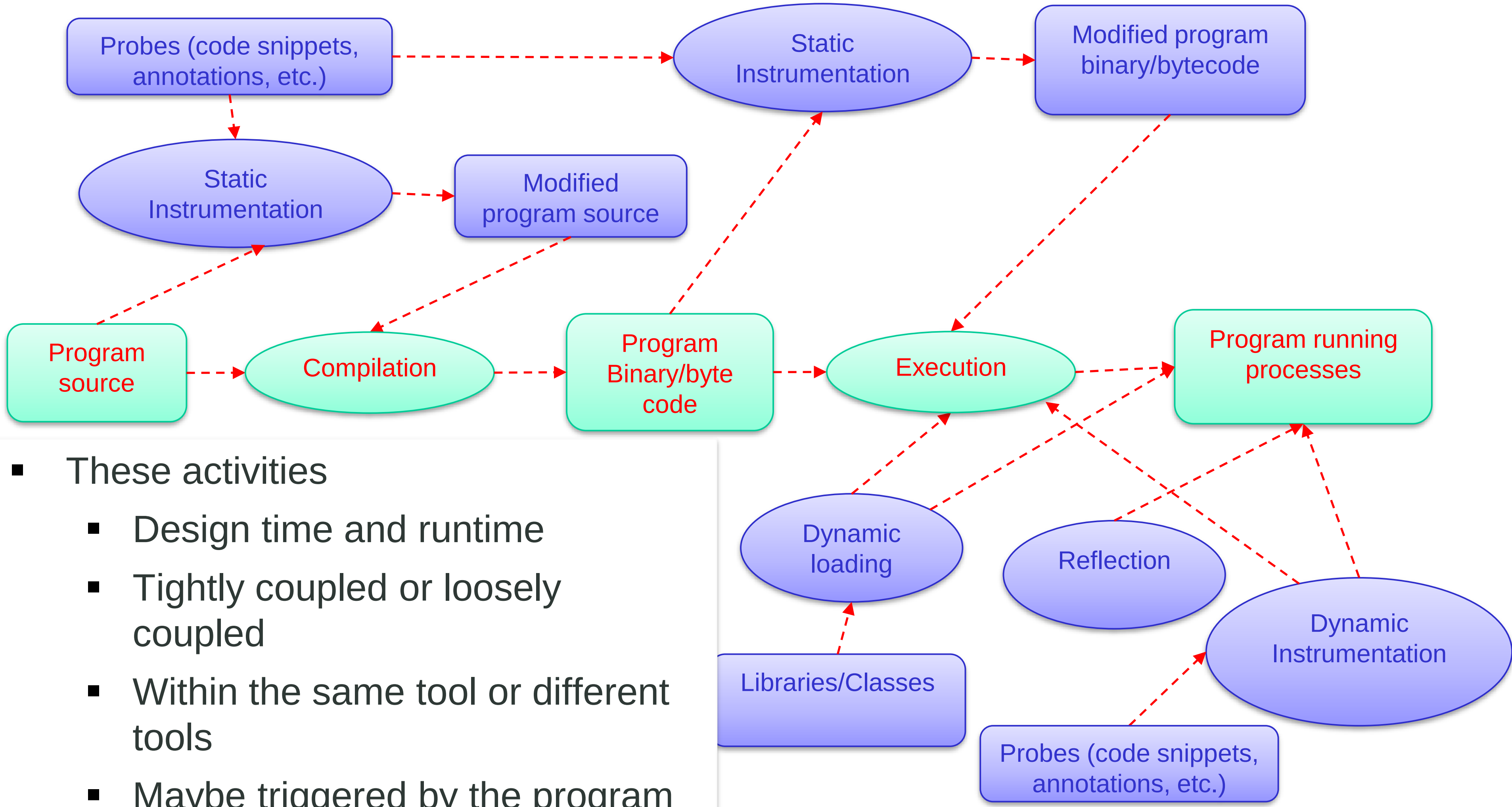
Dynamicity needs (1)

- Monitoring, performance analysis, tracing, debugging
  - Dependent on specific contexts → static ways are not good due to overhead and flexibility requirements
- Common concerns among several classes/objects/components
  - Do not want to repeat programming effort
- Provide metadata information for lifecycle/configuration management
  - Provisioning and runtime configuration

Dynamic needs (2)

- Provide metadata information for code generation
  - Service interfaces and validation
- Flexible software delivery for some core functions (e.g., patches)
- Enable continuous update without recompiling/redeploying applications
- Extensibility

Main activities for programming dynamic features



- These activities
  - Design time and runtime
  - Tightly coupled or loosely coupled
  - Within the same tool or different tools
  - Maybe triggered by the program its self

CODE INSPECTION

We want to understand the program

- How do we know program structures?
- Can we analyze the program structure within program processes during runtime?
- Are we able to examine statically and dynamically linked code?
- What kind of tasks we could do if we know the program structure?

Code inspection

- Code inspection/analysis
  - Analyze different forms of program code at design and runtime
- Source code analysis
- Bytecode and binary code analysis
- Program’s running process analysis

Dynamic loading

- Code can be dynamically loaded into a running program
  - At runtime, libraries are loaded into a program memory
  - Variables, functions, etc. in these libraries can be used by the program
  - Dynamic linking: no libraries are needed at compile or before loading at runtime
- Implementations
  - C: void *dlopen(const char *filename, int flag);
  - Java ClassLoader and System.loadLibrary(String libname)

Code/Bytecode/Binary inspection

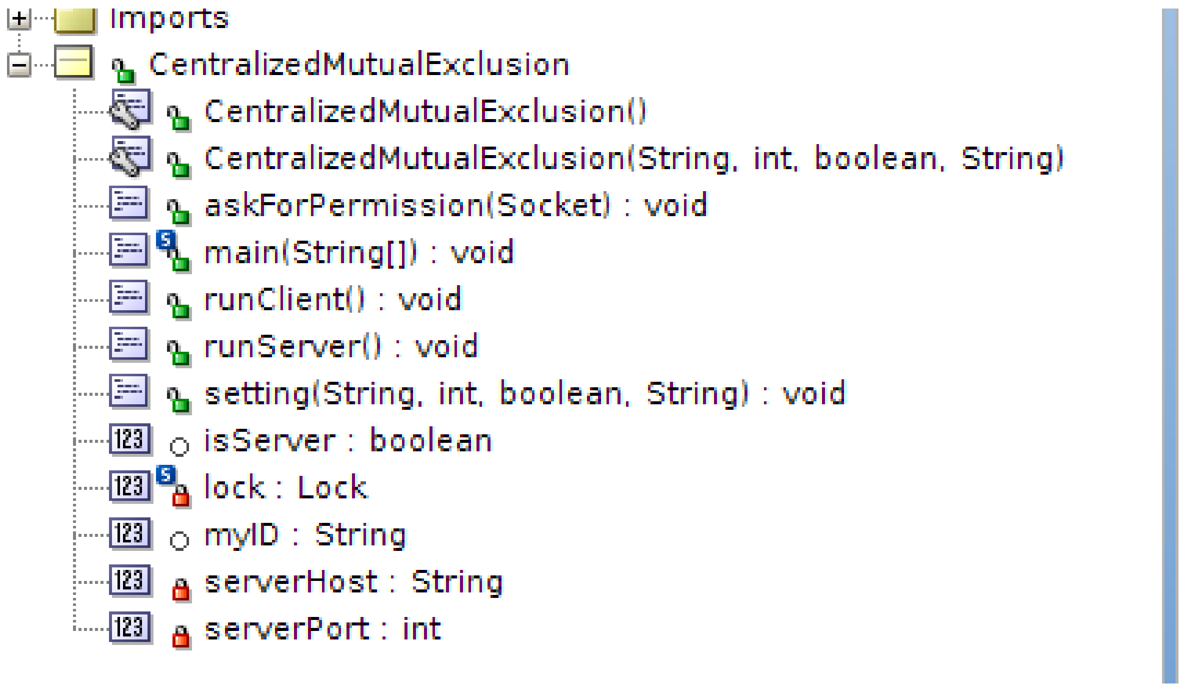
- Read and build program structures from (byte)code
  - Example, using javap –c to see Java bytecode
- Tools to process bytecodes
  - Javassist (https://github.com/jboss-javassist/javassist/releases)
  - BCEL (http://commons.apache.org/proper/commons-bcel/)
  - CGLIB (https://github.com/cglib/cglib)
- Cannot see the dynamic code which will be loaded at runtime

Reflection

- Allow a running program to introspect its own code structuren (e.g., methods and fields of a class)
- Enable the creation and invocation of object instances at runtime
- Basic feature in most current object-oriented languages

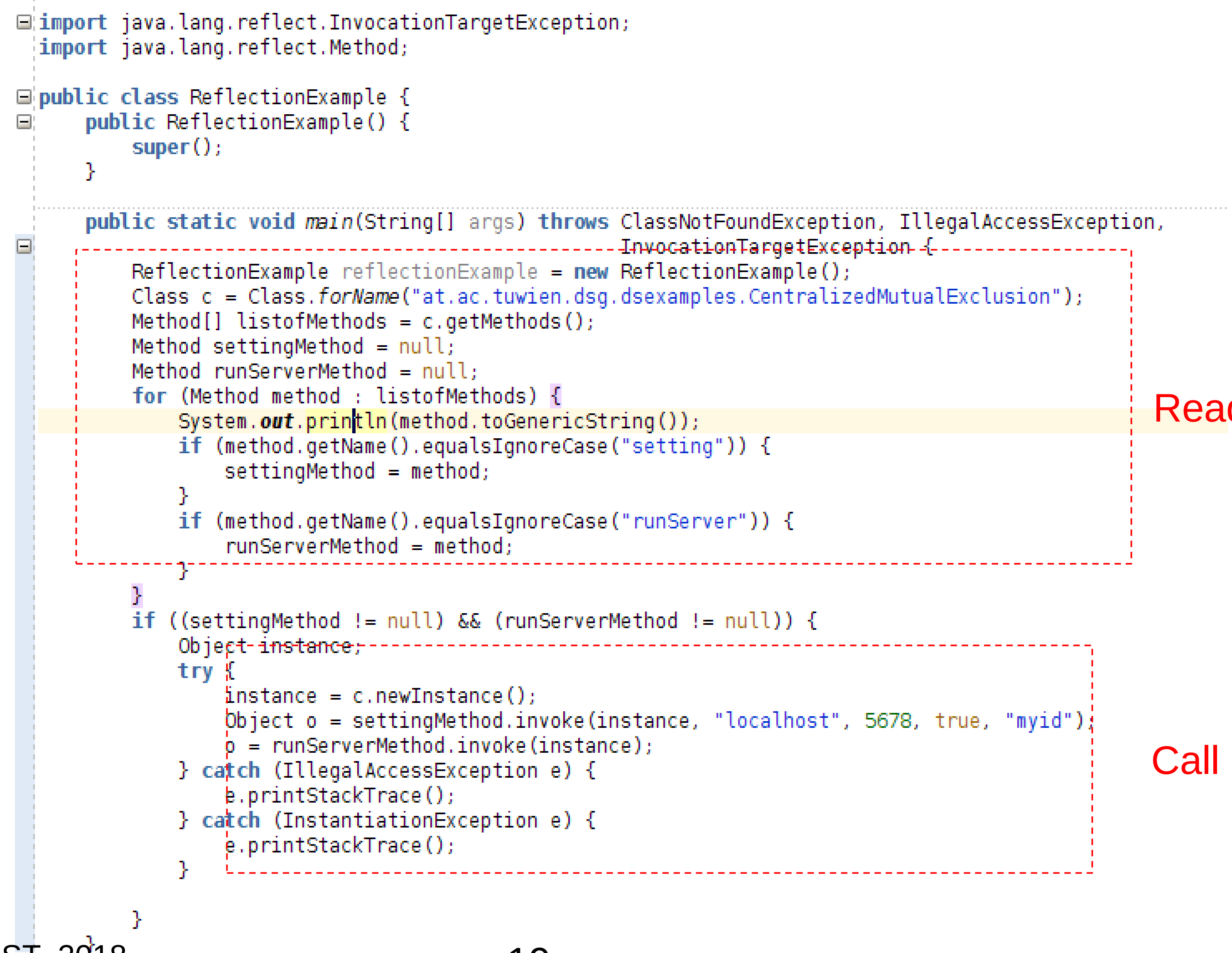
Example: Reflection in Java

Source code



```
Imports
CentralizedMutualExclusion
    CentralizedMutualExclusion()
    CentralizedMutualExclusion(String, int, boolean, String)
    askForPermission(Socket) : void
    main(String[]) : void
    runClient() : void
    runServer() : void
    setting(String, int, boolean, String) : void
    isServer : boolean
    lock : Lock
    myID : String
    serverHost : String
    serverPort : int
```

Example: Reflection in Java



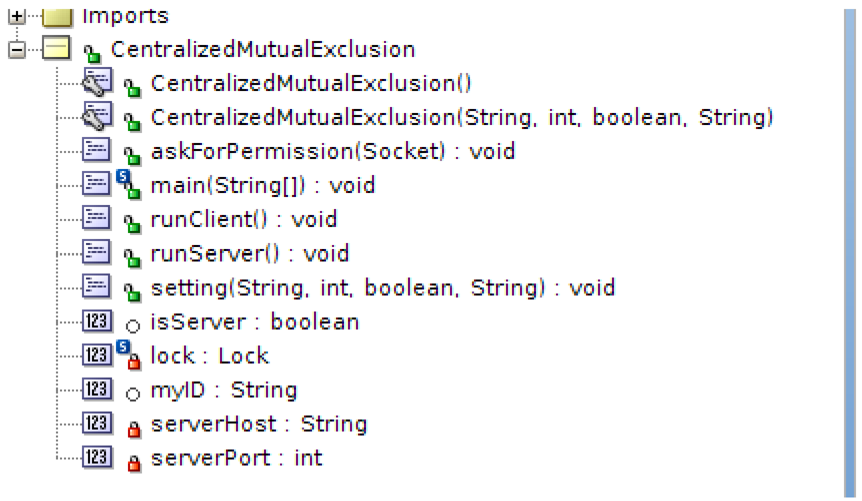
```
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class ReflectionExample {
    public ReflectionExample() {
        super();
    }

    public static void main(String[] args) throws ClassNotFoundException, IllegalAccessException,
                                                  InvocationTargetException {
        ReflectionExample reflectionExample = new ReflectionExample();
        Class c = Class.forName("at.ac.tuwien.dsg.dsexamples.CentralizedMutualExclusion");
        Method[] listofMethods = c.getMethods();
        Method settingMethod = null;
        Method runServerMethod = null;
        for (Method method : listofMethods) {
            System.out.println(method.toGenericString());
            if (method.getName().equalsIgnoreCase("setting")) {
                settingMethod = method;
            }
            if (method.getName().equalsIgnoreCase("runServer")) {
                runServerMethod = method;
            }
        }
        if ((settingMethod != null) && (runServerMethod != null)) {
            Object instance;
            try {
                instance = c.newInstance();
                Object o = settingMethod.invoke(instance, "localhost", 5678, true, "myid");
                o = runServerMethod.invoke(instance);
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            } catch (InstantiationException e) {
                e.printStackTrace();
            }

        }
}
```

Read structures

Call methods

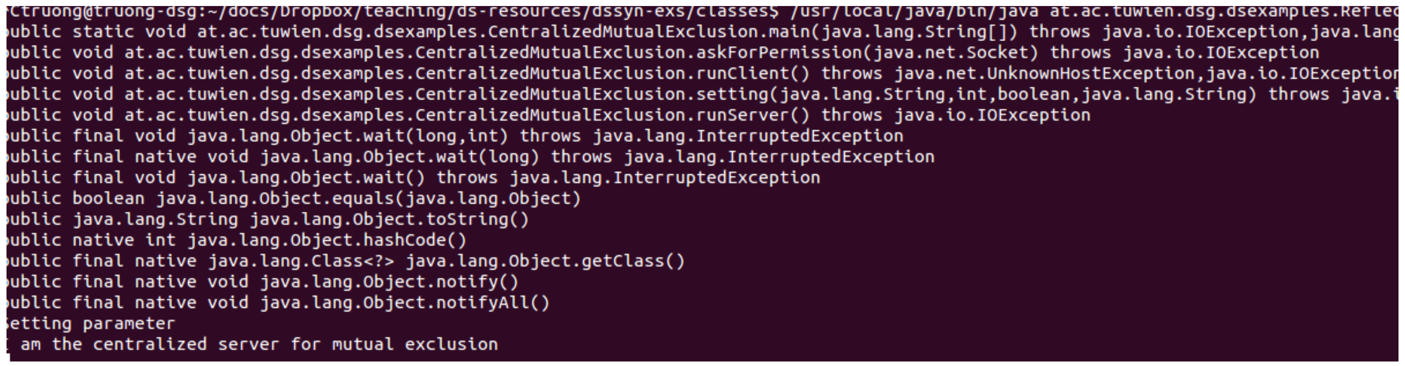
Example: Reflection in Java

Source code



```
Imports
CentralizedMutualExclusion
    CentralizedMutualExclusion()
    CentralizedMutualExclusion(String, int, boolean, String)
    askForPermission(Socket) : void
    main(String[]) : void
    runClient() : void
    runServer() : void
    setting(String, int, boolean, String) : void
    isServer : boolean
    lock : Lock
    myID : String
    serverHost : String
    serverPort : int
```

Inspection output



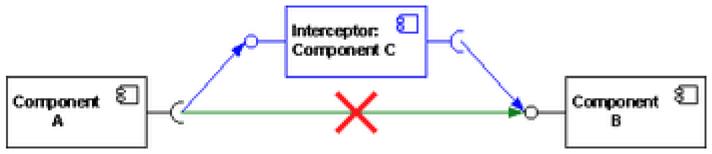
```
Ctruong@truong-dsg:~/docs/Dropbox/teaching/ds-resources/dssyn-exs/classes$ /usr/local/java/bin/java at.ac.tuwien.dsg.dsexamples.Refle
ublic static void at.ac.tuwien.dsg.dsexamples.CentralizedMutualExclusion.main(java.lang.String[]) throws java.io.IOException,java.lang
ublic void at.ac.tuwien.dsg.dsexamples.CentralizedMutualExclusion.askForPermission(java.net.Socket) throws java.io.IOException
ublic void at.ac.tuwien.dsg.dsexamples.CentralizedMutualExclusion.runClient() throws java.net.UnknownHostException,java.io.IOExceptio
ublic void at.ac.tuwien.dsg.dsexamples.CentralizedMutualExclusion.setting(java.lang.String,int,boolean,java.lang.String) throws java.
ublic void at.ac.tuwien.dsg.dsexamples.CentralizedMutualExclusion.runServer() throws java.io.IOException
ublic final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
ublic final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
ublic final void java.lang.Object.wait() throws java.lang.InterruptedException
ublic boolean java.lang.Object.equals(java.lang.Object)
ublic java.lang.String java.lang.Object.toString()
ublic native int java.lang.Object.hashCode()
ublic final native java.lang.Class<?> java.lang.Object.getClass()
ublic final native void java.lang.Object.notify()
ublic final native void java.lang.Object.notifyAll()
etting parameter
 am the centralized server for mutual exclusion
```

Reflection – Benefit?

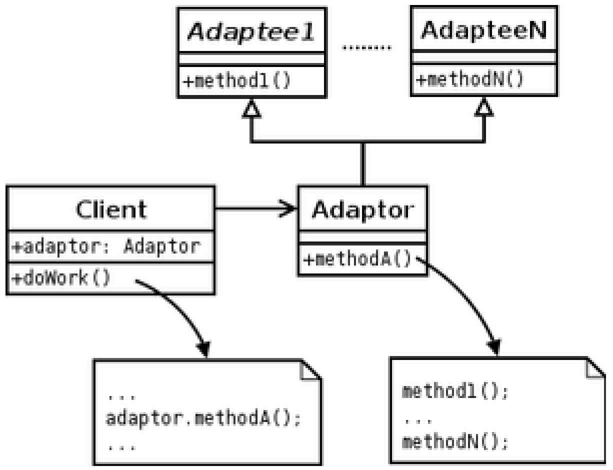
- Benefits?
  - Allows for flexible applications and design patterns
- Disadvantages:
  - Complex solutions are difficult to write and error-prone (method names in strings, etc.)
  - Performance degradation
  - Security restrictions
  - Reflection is read-only – it is not (easily) possible to add methods or change the inheritance hierarchy of an object

Dynamic proxy

- Allow us to implement a proxy class whose interfaces specified at runtime
- Create proxy instance with a set of interfaces
- Forward method invocations on interfaces of the proxy instance to another object

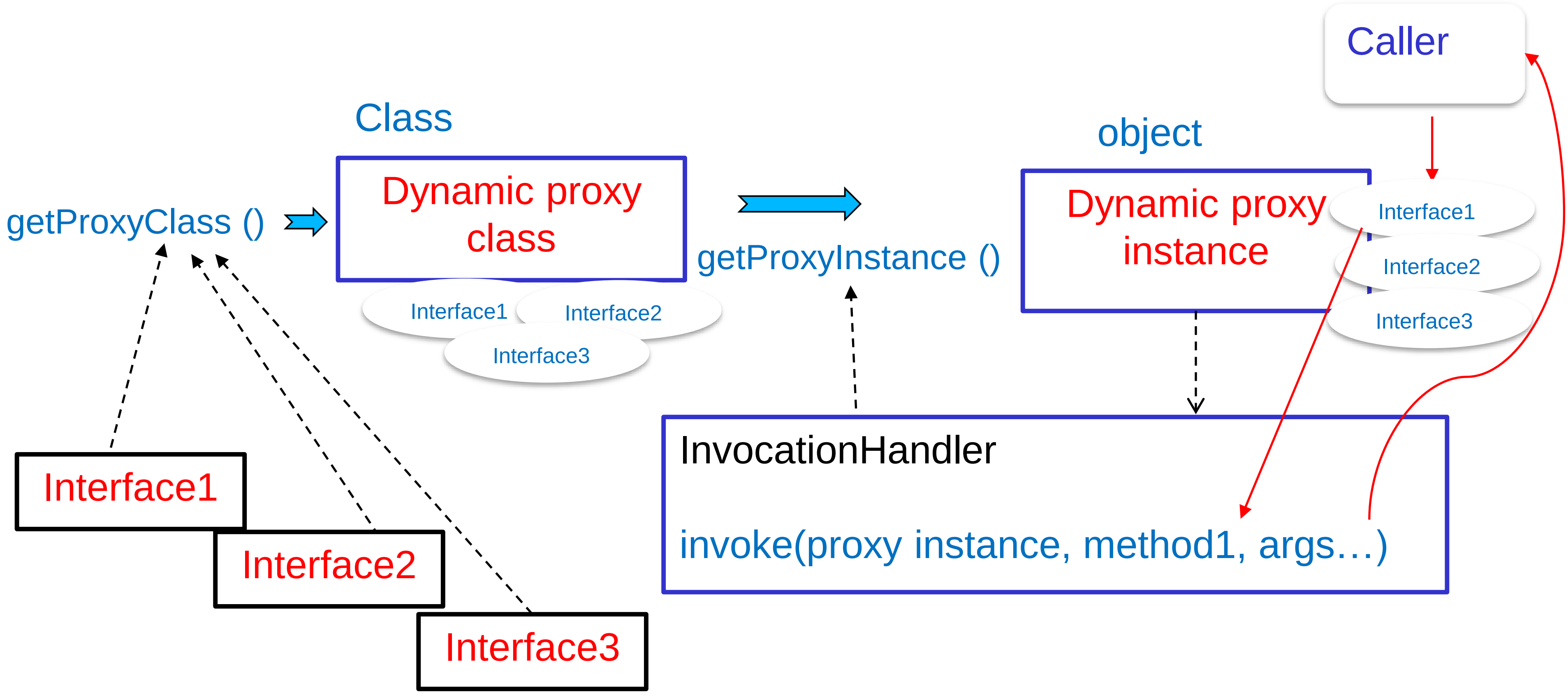


Source: http://en.wikipedia.org/wiki/Interceptor_pattern



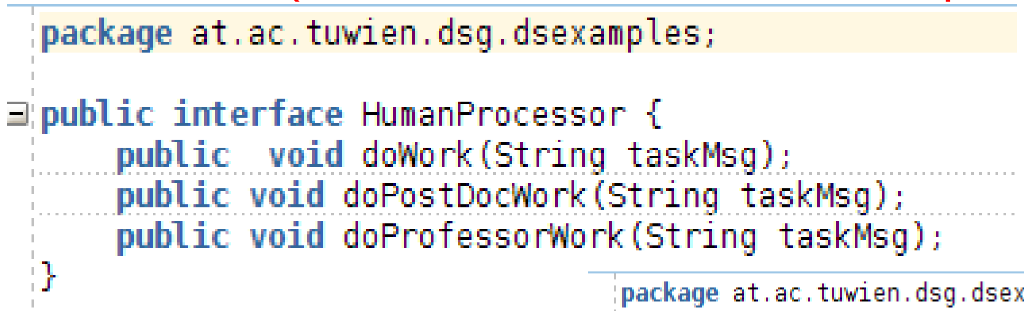
Source: http://en.wikipedia.org/wiki/Adapter_pattern

Dynamic Proxy – Conceptual Model

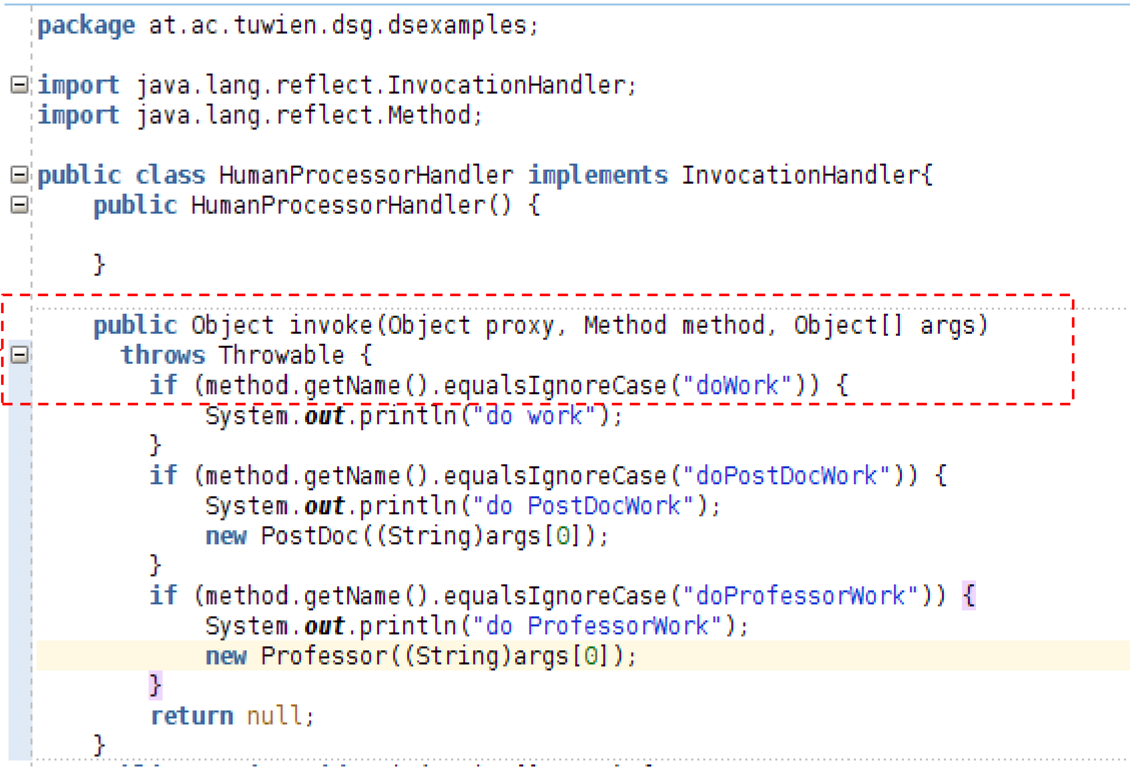


Example of Dynamic Proxy

Interfaces (Methods to be invoked on the proxy)



```
package at.ac.tuwien.dsg.dsexamples;

public interface HumanProcessor {
    public  void doWork(String taskMsg);
    public void doPostDocWork(String taskMsg);
    public void doProfessorWork(String taskMsg);
}
```

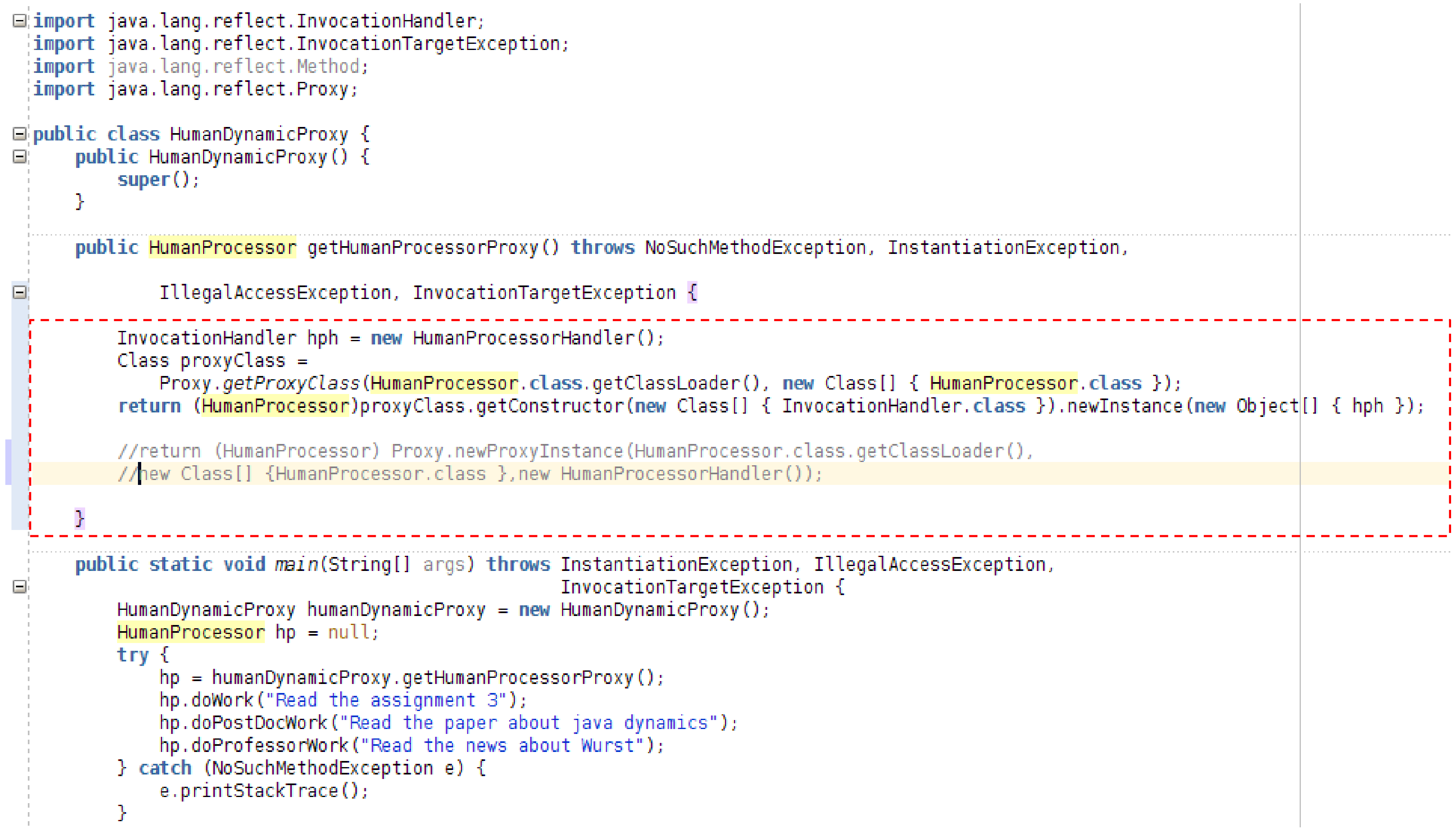


```
package at.ac.tuwien.dsg.dsexamples;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class HumanProcessorHandler implements InvocationHandler{
    public HumanProcessorHandler() {

    }

    public Object invoke(Object proxy, Method method, Object[] args)
      throws Throwable {
        if (method.getName().equalsIgnoreCase("doWork")) {
            System.out.println("do work");
        }
        if (method.getName().equalsIgnoreCase("doPostDocWork")) {
            System.out.println("do PostDocWork");
            new PostDoc((String)args[0]);
        }
        if (method.getName().equalsIgnoreCase("doProfessorWork")) {
            System.out.println("do ProfessorWork");
            new Professor((String)args[0]);
        }
        return null;
    }
```

must implement

Handler
(Proxy behavior)

Example of Dynamic Proxy



```java
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class HumanDynamicProxy {
    public HumanDynamicProxy() {
        super();
    }

    public HumanProcessor getHumanProcessorProxy() throws NoSuchMethodException, InstantiationException,

            IllegalAccessException, InvocationTargetException {

        InvocationHandler hph = new HumanProcessorHandler();
        Class proxyClass =
            Proxy.getProxyClass(HumanProcessor.class.getClassLoader(), new Class[] { HumanProcessor.class });
        return (HumanProcessor)proxyClass.getConstructor(new Class[] { InvocationHandler.class }).newInstance(new Object[] { hph });

        //return (HumanProcessor) Proxy.newProxyInstance(HumanProcessor.class.getClassLoader(),
        //new Class[] {HumanProcessor.class },new HumanProcessorHandler());

    }

    public static void main(String[] args) throws InstantiationException, IllegalAccessException,
                                                  InvocationTargetException {
        HumanDynamicProxy humanDynamicProxy = new HumanDynamicProxy();
        HumanProcessor hp = null;
        try {
            hp = humanDynamicProxy.getHumanProcessorProxy();
            hp.doWork("Read the assignment 3");
            hp.doPostDocWork("Read the paper about java dynamics");
            hp.doProfessorWork("Read the news about Wurst");
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        }
```

PROGRAM INSTRUMENTATION AND ANALYSIS

Program instrumentation

- A process to inspect and insert additional code/meta-data, etc., into a program/process
  - Static and runtime
  - Manual or automatic
- Examples:
  - Source code annotations/directives
  - Byte code modification
  - Dynamic code modification at loading time
  - Process instructions at runtime

Static versus Dynamic instrumentation

- Dynamic instrumentation
  - Perform the instrumentation during the process running
    - E.g., Dyninst (http://www.dyninst.org)
  - Java support:
      - E.g., Dtrace + Btrace, http://www.oracle.com/in/javaonedevelop/dtrace-j1-sritter-400745-en-in.pdf
      - Java instrumentation API
    - Some works on static + dynamic instrumentation based on dynamic class loading mechanisms
- Static instrumentation
  - Source code, bytecode and binary code levels
- In many cases: a combination of different methods

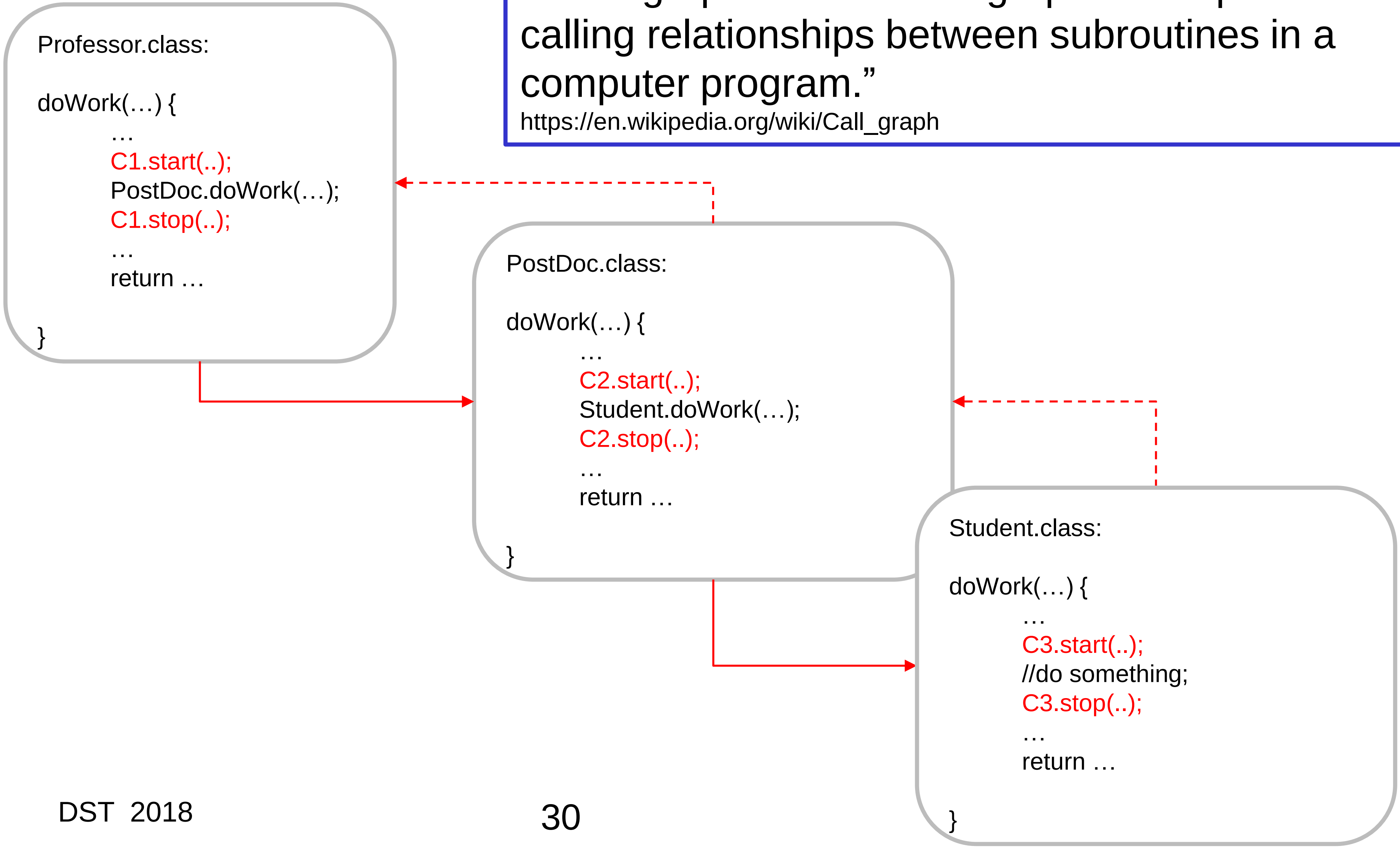
Where we can insert instrumented code into the program execution?

- At any join point: a point in the control flow of a program
- Examples:
  - method calls, entry/exit of method body, statements (set/get, assignment, etc.)

If we instrument probes before, after or around these join points, when the program execution reaches these points, the probes will be executed accordingly.

Example: Dynamic call graph

“A call graph is a directed graph that represents calling relationships between subroutines in a computer program.”
https://en.wikipedia.org/wiki/Call_graph

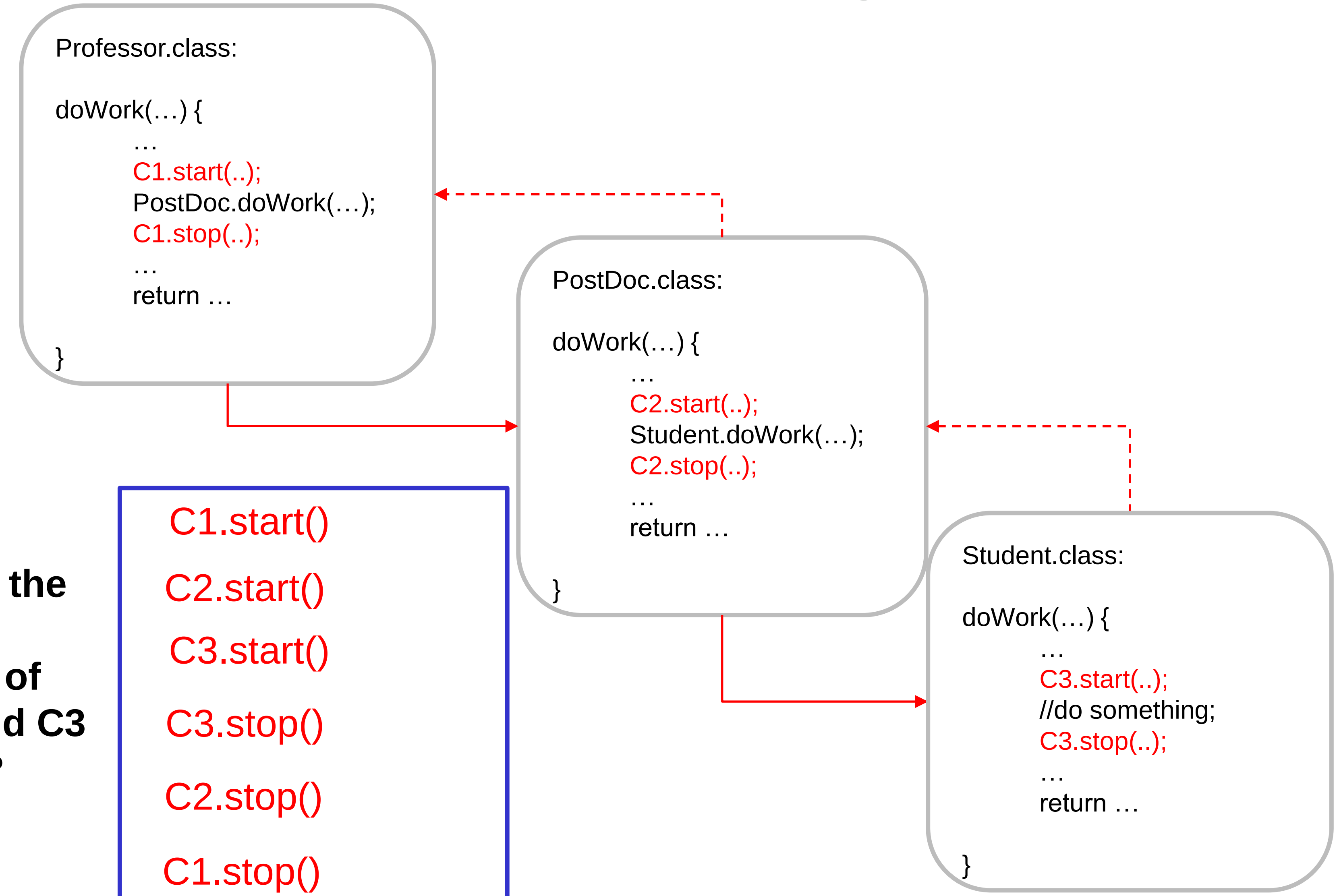


```
Professor.class:

doWork(…) {
          …
          C1.start(..);
          PostDoc.doWork(…);
          C1.stop(..);
          …
          return …

}
```

```
PostDoc.class:

doWork(…) {
          …
          C2.start(..);
          Student.doWork(…);
          C2.stop(..);
          …
          return …

}
```

```
Student.class:

doWork(…) {
          …
          C3.start(..);
          //do something;
          C3.stop(..);
          …
          return …

}
```

Example: Dynamic call graph



```
Professor.class:

doWork(…) {
        …
        C1.start(..);
        PostDoc.doWork(…);
        C1.stop(..);
        …
        return …

}
```

```
PostDoc.class:

doWork(…) {
        …
        C2.start(..);
        Student.doWork(…);
        C2.stop(..);
        …
        return …

}
```

```
Student.class:

doWork(…) {
        …
        C3.start(..);
        //do something;
        C3.stop(..);
        …
        return …

}
```

How does the execution sequence of C1, C2, and C3 look like ?

C1.start()

C2.start()

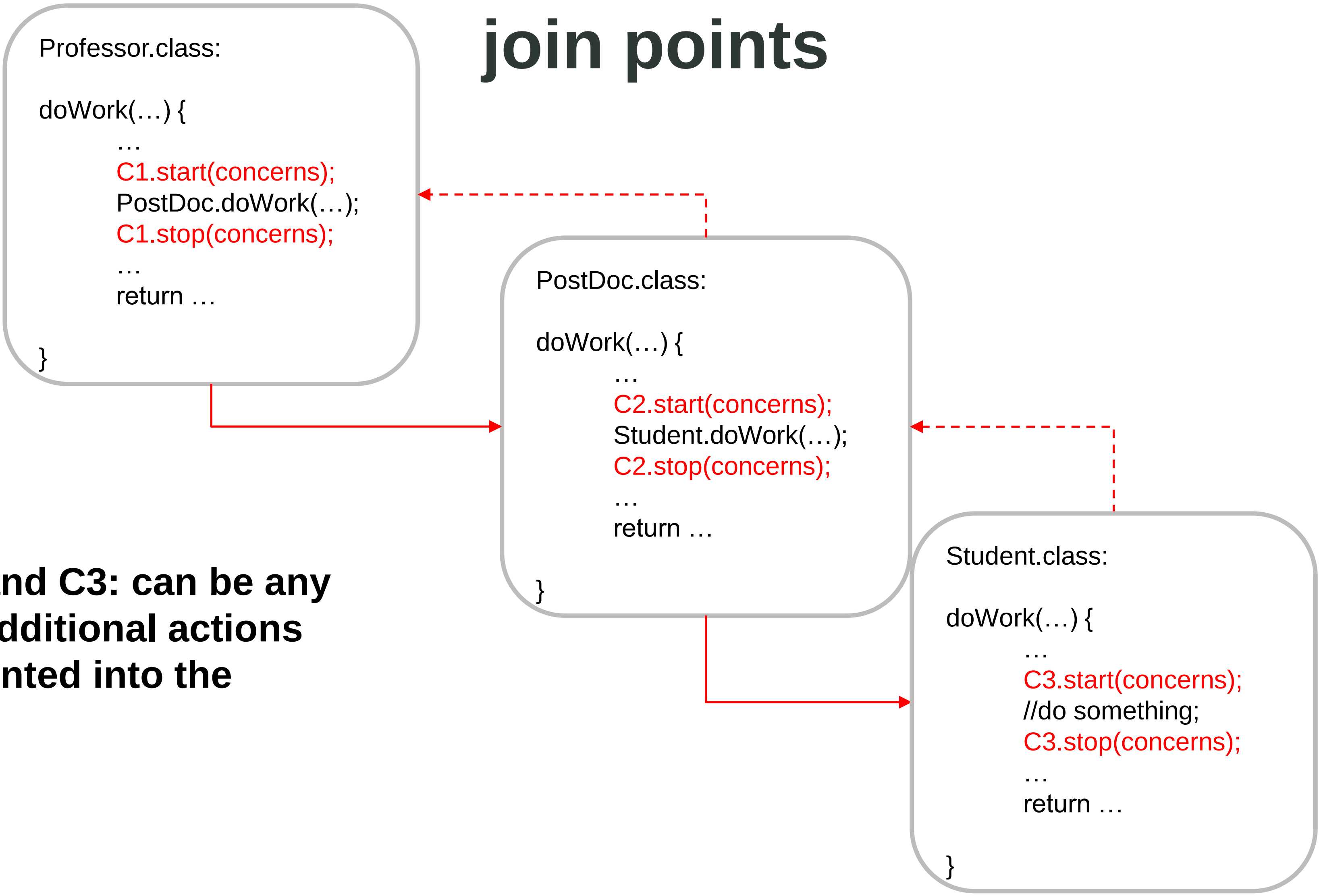
C3.start()

C3.stop()

C2.stop()

C1.stop()

If we want to deal with certain concerns at join points



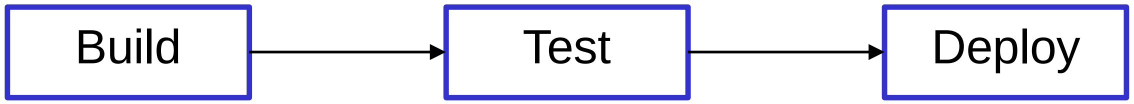
```
Professor.class:

doWork(…) {
        …
        C1.start(concerns);
        PostDoc.doWork(…);
        C1.stop(concerns);
        …
        return …

}
```

```
PostDoc.class:

doWork(…) {
        …
        C2.start(concerns);
        Student.doWork(…);
        C2.stop(concerns);
        …
        return …

}
```

```
Student.class:

doWork(…) {
        …
        C3.start(concerns);
        //do something;
        C3.stop(concerns);
        …
        return …

}
```

C1, C2, and C3: can be any kind of additional actions instrumented into the program

When and where we should use inspection, reflection, or instrumentation in our continuous integration (CI) pipelines?



Travis CI circleci GitLab Jenkins

SOURCE CODE ANNOTATION

Annotation

- Annotations are added into source code
  - Can be considered as static instrumentation
  - Can be considered as a part of typical programming activities
- Goal: provide additional metadata/instructions
  - For supporting compilation process
  - For code generation at compiling and deployment
  - For runtime processing
  - Etc.
- Very popular in Java/C#/Python, …

Java Annotation

- Format

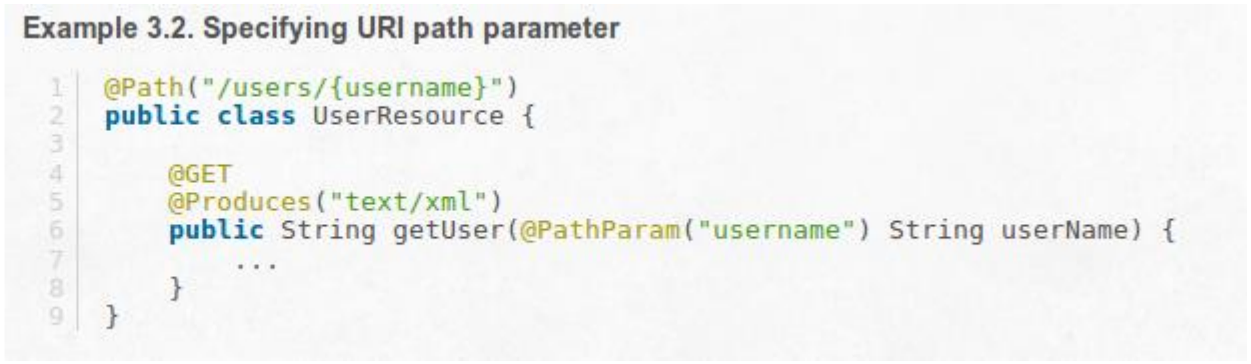
  @AnnotationName (….)

- Pre-defined versus user-defined
  - *Pre-defined:* supporting by runtime systems or some well-known libraries in programming frameworks
  - *User-defined:* it is up to the developer to define annotations
- Points at which annotations can be added
  - declarations of classes, fields, methods, and other program elements
  - type uses (Java 8, e.g. @NonNull String serverName)

Example of EE Annotation Support

- Common annotations in Java (https://jcp.org/en/jsr/detail?id=250)
  - Supported in Spring annotations (http://docs.spring.io/spring/docs)

- JAX-RS (https://jax-rs-spec.java.net/)



Example 3.2. Specifying URI path parameter

```java
@Path("/users/{username}")
public class UserResource {

    @GET
    @Produces("text/xml")
    public String getUser(@PathParam("username") String userName) {
        ...
    }
}
```

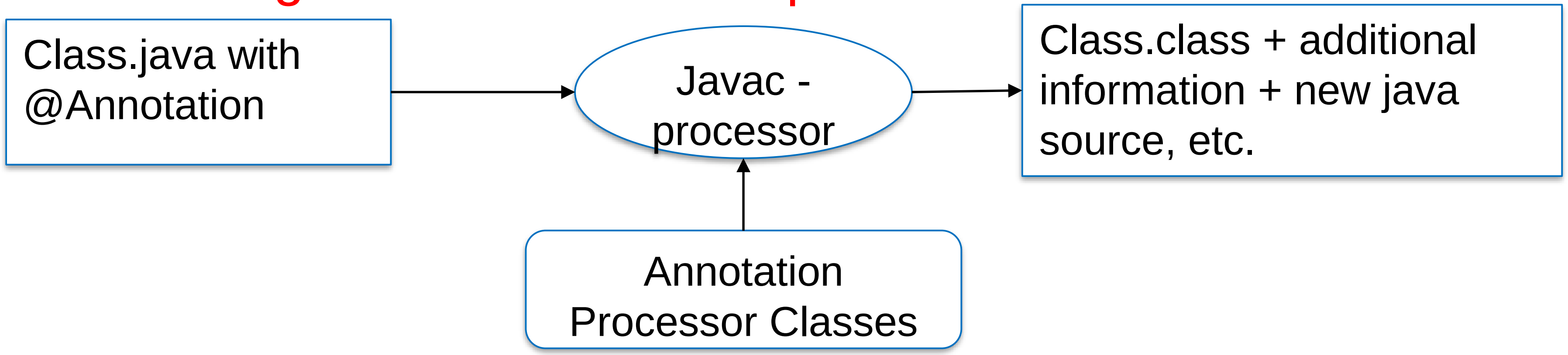
Source code: https://jersey.github.io/documentation/latest/jaxrs-resources.html

Java Annotation Processing

- Parsing source codes
- Reflection APIs also return Annotation

  Method.class: public Annotation[] getDeclaredAnnotations()
- Reading bytecode to get Annotation

Processing Model in Java compilation


Class.java with @Annotation
Javac - processor
Class.class + additional information + new java source, etc.
Annotation Processor Classes

Example – your case study with New Relic

Check:

https://docs.newrelic.com/docs/agents/java-agent/custom-instrumentation/java-instrumentation-annotation

```
@Trace
protected void methodWithinTransaction() {
  // work
}
```

ASPECT-ORIENTED PROGRAMMING

Cross-cutting concerns

- We have some common concerns that across multiple objects/methods
  - Tracing, measuring time, logging, checking security, etc.
- We want to have dynamically programming features to address these concerns

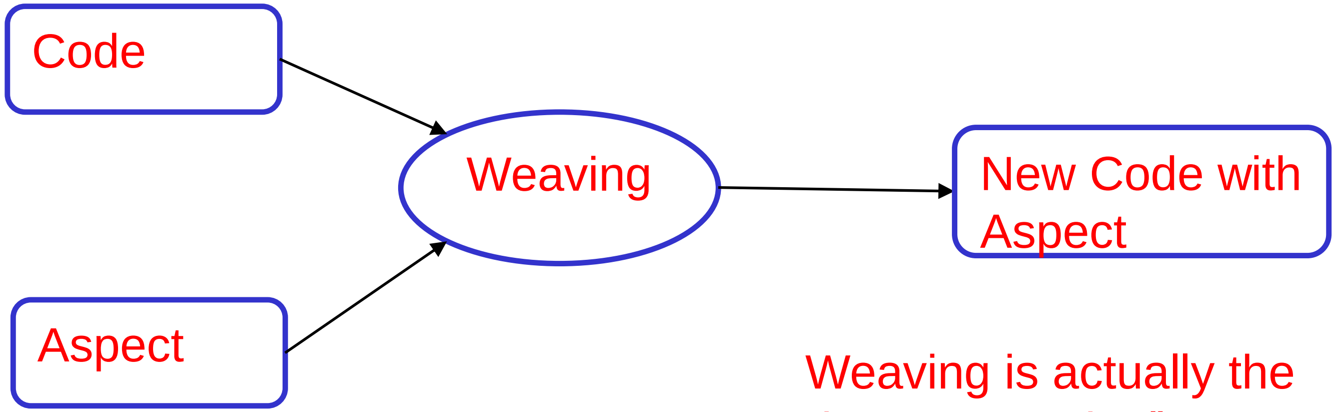
Cross-cutting concerns – when, where and how

- We can use „probes“ instrumented into targeting programs → creating hooks
  - Probes specify code for dealing these concerns
  - Probes create addition actions at runtime
- But we need dynamic and flexible way
  - Probes are instrumented when and where we need but they can be replaced!
- How
  - Can we use annotation? Can we use dynamic loading? Bytecode/binary instrumentation? Dynamic instrumentation?

Aspect-Oriented Programing

- Aspect: common feature in various methods, classes, objects, etc. → crosscutting concern
- Separate from functional concerns and cross-cutting concerns
  - In Aspect-Oriented Software Development (AOSD), functional concerns are built in the usual way
  - Cross-cutting concerns are built as independent modules
- Combining these two types of concerns using semi-automatic instrumentation techniques

Conceptual model – Aspect terminologies



Weaving is actually the „instrumentation" process

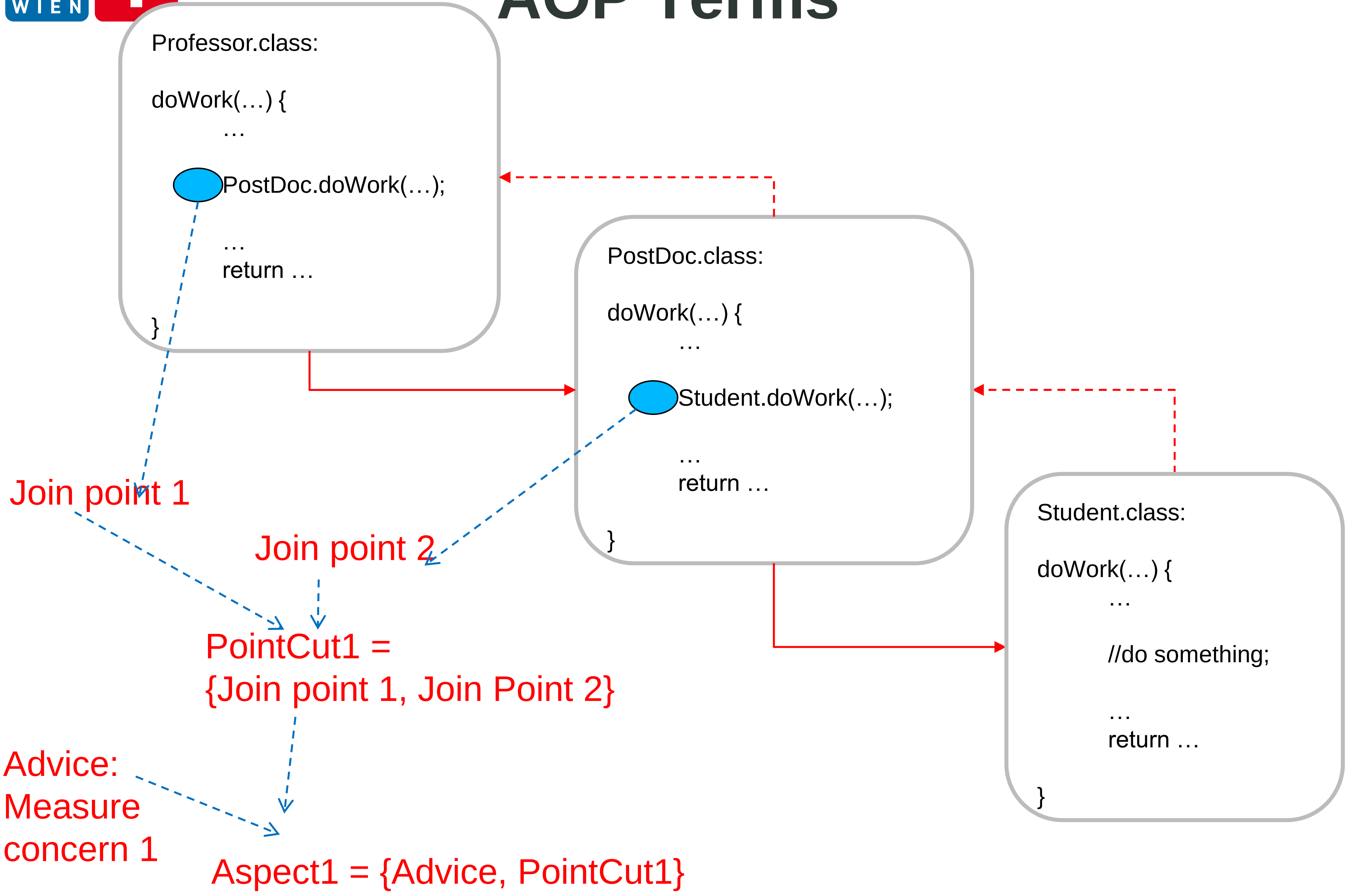
- Some java implementations
  - AspectJ
    - The standard implementation of AOP in Java
  - SpringAOP

AOP Terminologies

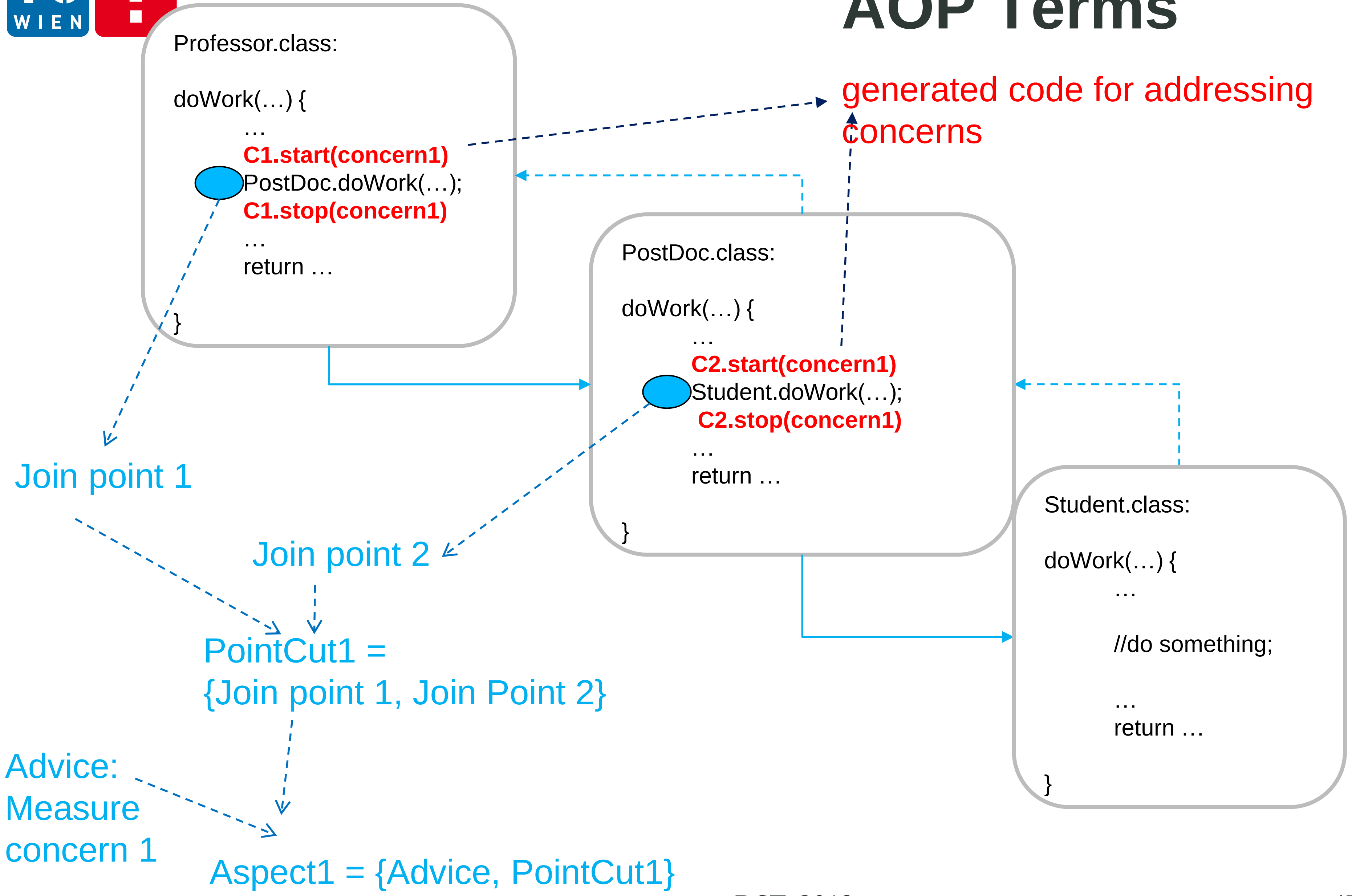
- Join Point
  - point in the execution, e.g. a call of a method with the signature “doWork(String)”
- Pointcut

  A set of join points (can also be composed using different operators such as &&, ||, !)
- Advice
  - Additional action that should be executed at join points in a pointcut
- Aspect
  - Cross-cutting type and its implementation (advices + others)

AOP Terms


Professor.class:
doWork(…) {
…
PostDoc.doWork(…);
…
return …
}
PostDoc.class:
doWork(…) {
…
Student.doWork(…);
…
return …
}
Student.class:
doWork(…) {
…
//do something;
…
return …
}
Join point 1
Join point 2
PointCut1 =
{Join point 1, Join Point 2}
Advice:
Measure
concern 1
Aspect1 = {Advice, PointCut1}

AOP Terms



```
Professor.class:

doWork(…) {
        …
        C1.start(concern1)
        PostDoc.doWork(…);
        C1.stop(concern1)
        …
        return …

}
```

```
PostDoc.class:

doWork(…) {
        …
        C2.start(concern1)
        Student.doWork(…);
         C2.stop(concern1)
        …
        return …

}
```

```
Student.class:

doWork(…) {
        …

        //do something;

        …
        return …

}
```

generated code for addressing concerns
Join point 1
Join point 2
PointCut1 =
{Join point 1, Join Point 2}
Advice:
Measure
concern 1
Aspect1 = {Advice, PointCut1}

Main types of Join Points

- Execution: when a method body executes
  execution(public void doWork(String))

- Call: when a method is called
  call(void doWork(String))

- Handler: when an exception handler executes

handler(ArrayOutOfBoundsException)

Main types of Join Points

- **this**: when the current executing object is of the specified type

  this(Student)
- **target**: when the target object is of the specified type

  target(Student)
- **within**: when the executing code within the specified class

  within(Student)
- **withincode**: within a method

  withincode(void doWork())
- **set/get**: field access/references

  set(String Student.name)

Call vs. Execution Join Points

- **Call** matches before or after a method is called (i.e., still in the scope of the caller)

```
      //call site: call point is here
   doWork()
   //call site: call point is here
```

- **Execution** matches when the method starts to execute (i.e. already in the scope of the callee)

```
   doWork() {
   //call site: call point is here (before)
   //...
   //call site: call point is here (after)
   }
```

Advice

- Advice defines code of aspect implementation that is executed at defined points
- Main types of advice

```
before () : methodCall() {
        …
}
after () : methodCall() {
        …
}
around () : methodCall() {
        ….
}
```

methodCall is a pointcut

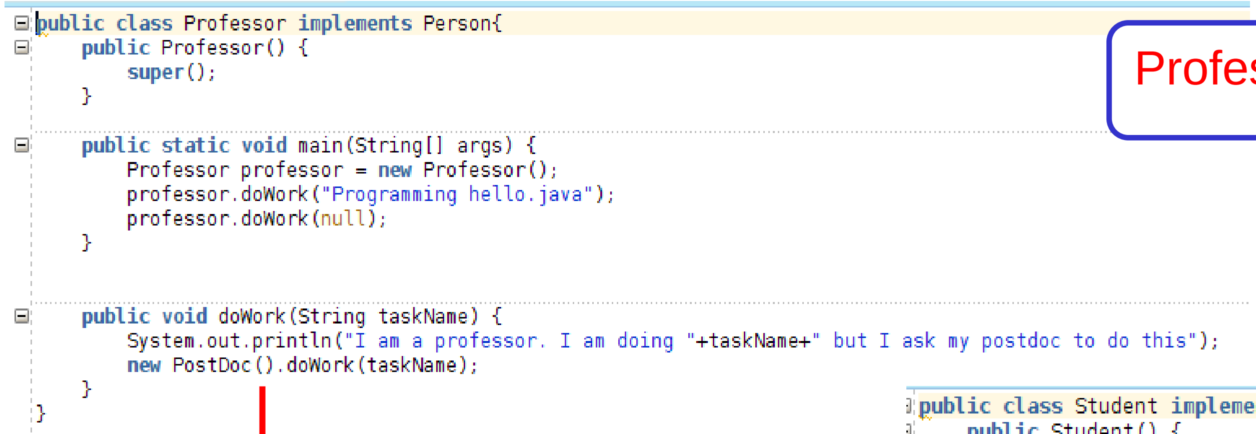
Weaving (Instrumentation)

- The process of merging aspects into the program code is called weaving → instrumentation
- Three ways of weaving:
  - Compile-Time Weaving (weave as part of source-to-binary compilation)
  - Post-compile Weaving (compile normally, then merge binaries in a post-compilation step)
  - Load-Time Weaving (like binary weaving, but done when the class is loaded by the classloader)
  - Runtime Weaving: using proxy

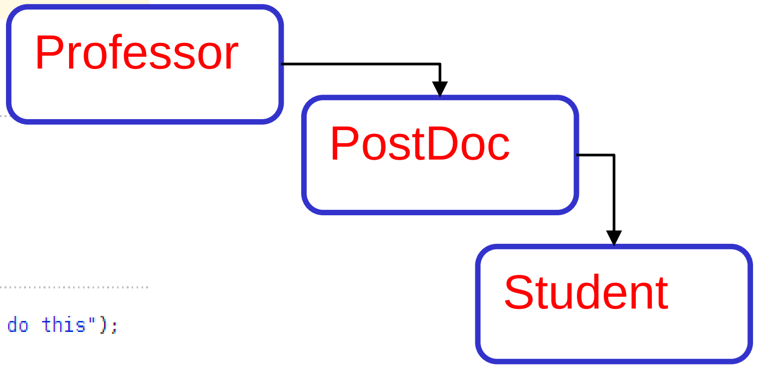
Your home work:

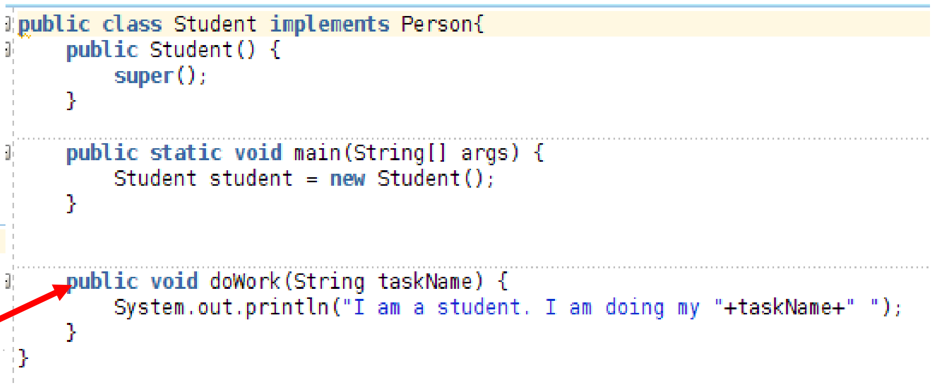
Pros and Cons of compile time, binary and load time weaving

Example of AOP with AspectJ

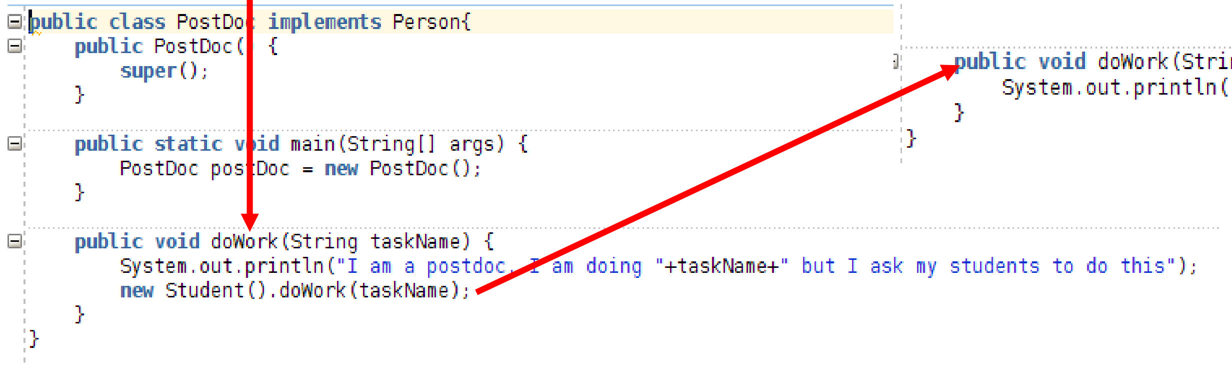


```java
public class Professor implements Person{
    public Professor() {
        super();
    }

    public static void main(String[] args) {
        Professor professor = new Professor();
        professor.doWork("Programming hello.java");
        professor.doWork(null);
    }

    public void doWork(String taskName) {
        System.out.println("I am a professor. I am doing "+taskName+" but I ask my postdoc to do this");
        new PostDoc().doWork(taskName);
    }
}
```



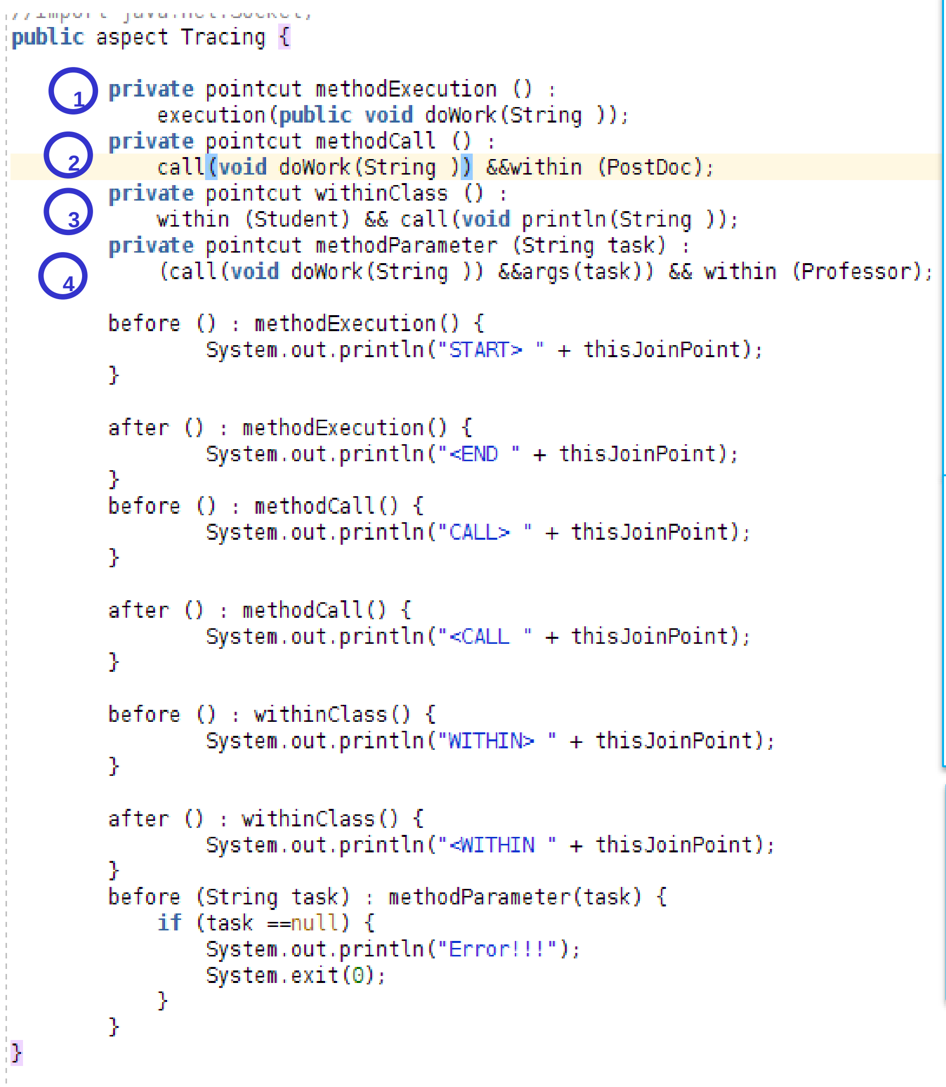


```java
public class Student implements Person{
    public Student() {
        super();
    }

    public static void main(String[] args) {
        Student student = new Student();
    }

    public void doWork(String taskName) {
        System.out.println("I am a student. I am doing my "+taskName+" ");
    }
}
```



```java
public class PostDoc implements Person{
    public PostDoc() {
        super();
    }

    public static void main(String[] args) {
        PostDoc postDoc = new PostDoc();
    }

    public void doWork(String taskName) {
        System.out.println("I am a postdoc, I am doing "+taskName+" but I ask my students to do this");
        new Student().doWork(taskName);
    }
}
```

Example of AOP with AspectJ



```
//import java.net.Socket;
public aspect Tracing {

1   private pointcut methodExecution () :
        execution(public void doWork(String ));
2   private pointcut methodCall () :
        call(void doWork(String )) &&within (PostDoc);
3   private pointcut withinClass () :
        within (Student) && call(void println(String ));
    private pointcut methodParameter (String task) :
4       (call(void doWork(String )) &&args(task)) && within (Professor);

    before () : methodExecution() {
            System.out.println("START> " + thisJoinPoint);
    }

    after () : methodExecution() {
            System.out.println("<END " + thisJoinPoint);
    }
    before () : methodCall() {
            System.out.println("CALL> " + thisJoinPoint);
    }

    after () : methodCall() {
            System.out.println("<CALL " + thisJoinPoint);
    }

    before () : withinClass() {
            System.out.println("WITHIN> " + thisJoinPoint);
    }

    after () : withinClass() {
            System.out.println("<WITHIN " + thisJoinPoint);
    }
    before (String task) : methodParameter(task) {
        if (task ==null) {
            System.out.println("Error!!!");
            System.exit(0);
        }
    }
}
```

Professor.class:

```
public void doWork(String taskName) {
1       System.out.println("I am a professor. I am doing "+taskName+" but I ask my postdoc to do this");
        new PostDoc().doWork(taskName);
  }
 public static void main(String[] args) {
        Professor professor = new Professor();        4
        professor.doWork("Programming hello.java");
          professor.doWork(null);
  }
```

PostDoc.class:

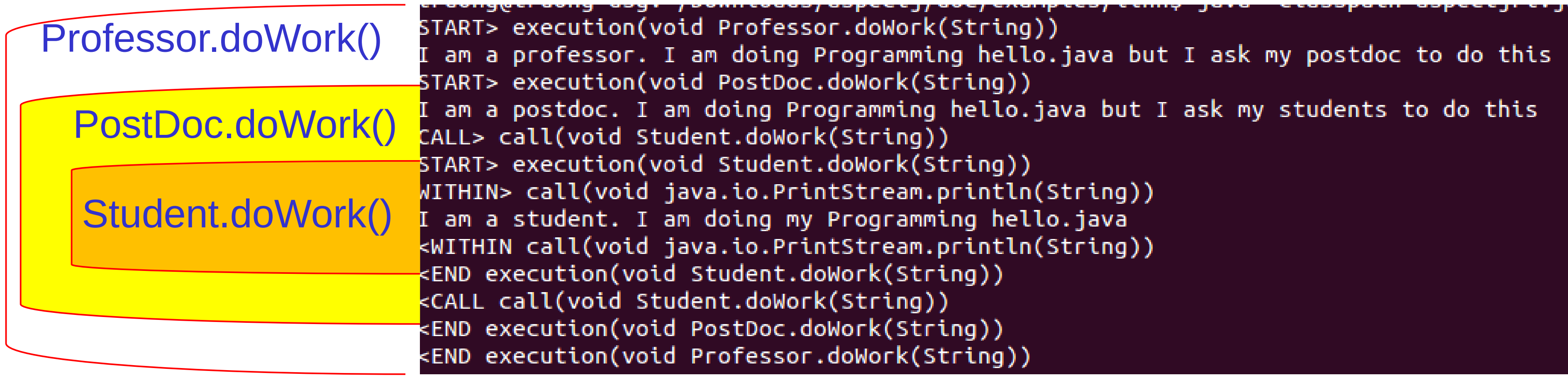
```
public void doWork(String taskName) {
1       System.out.println("I am a postdoc. I am doing "+taskName+" but I ask my students to do this"); 2
        new Student().doWork(taskName);
  }
```

Student.class:

```
 public void doWork(String taskName) {
1       System.out.println("I am a student. I am doing my "+taskName+" "); 3
   }
```

Example of AOP with AspectJ

Call gaph tracing information



```
START> execution(void Professor.doWork(String))
I am a professor. I am doing Programming hello.java but I ask my postdoc to do this
START> execution(void PostDoc.doWork(String))
I am a postdoc. I am doing Programming hello.java but I ask my students to do this
CALL> call(void Student.doWork(String))
START> execution(void Student.doWork(String))
WITHIN> call(void java.io.PrintStream.println(String))
I am a student. I am doing my Programming hello.java
<WITHIN call(void java.io.PrintStream.println(String))
<END execution(void Student.doWork(String))
<CALL call(void Student.doWork(String))
<END execution(void PostDoc.doWork(String))
<END execution(void Professor.doWork(String))
```

AOP in Spring

- Not all features are supported
  - String AOP only method execution join points
- Using Java annotation or XML
- Java Annotation
  - @Aspect, @Pointcut, @Before, @After, @AfterReturning, @Around
- Using XML
  - aop:config, aop:aspect, aop:before, etc.

What is the underlying mechanism?

→ Using dynamic proxy to delegate/process advices

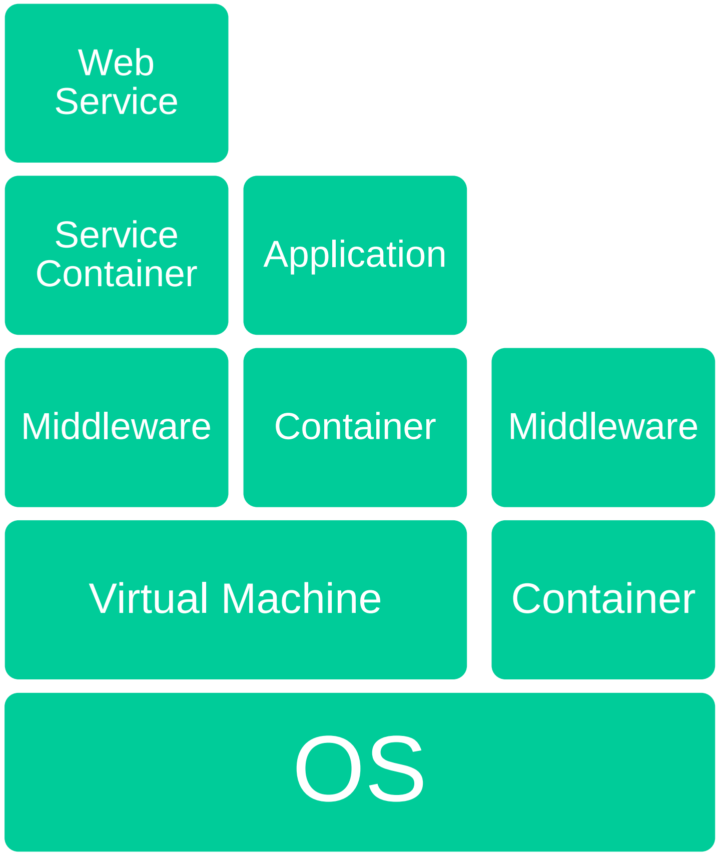
Think:

Assume that you run your cloud applications in different cloud infrastructures, for what kind of tasks could we benefit from AOP?

Where will we do this in the CI process?

CLOUD APPLICATIONS/SYSTEMS INSTRUMENTATION AND MONITORING AT SCALE

Full stack monitoring



- Combine many techniques: instrumentation, API interface, etc.
- Push and pull methods
- Exact measurement and sampling

Scale of systems and of monitoring

- Many monitoring components
  - Interfaces to different layers (systems and applications)
  - Different monitoring mechanisms
- Scalable middleware for relaying monitoring data
  - Various protocols, HTTP, AMQP,MQTT
- Scalable storage: file systems and time series data
- Visualization and other types of big/fast data analytics

What can we do with messaging, complex event processing (lecture 2) and dynamic features programming (lecture 3)?

Building real-world instrumentation and monitoring for (cloud-based) services
→ instrumentation and monitoring ecosystem for complex distributed systems

Example of log monitoring

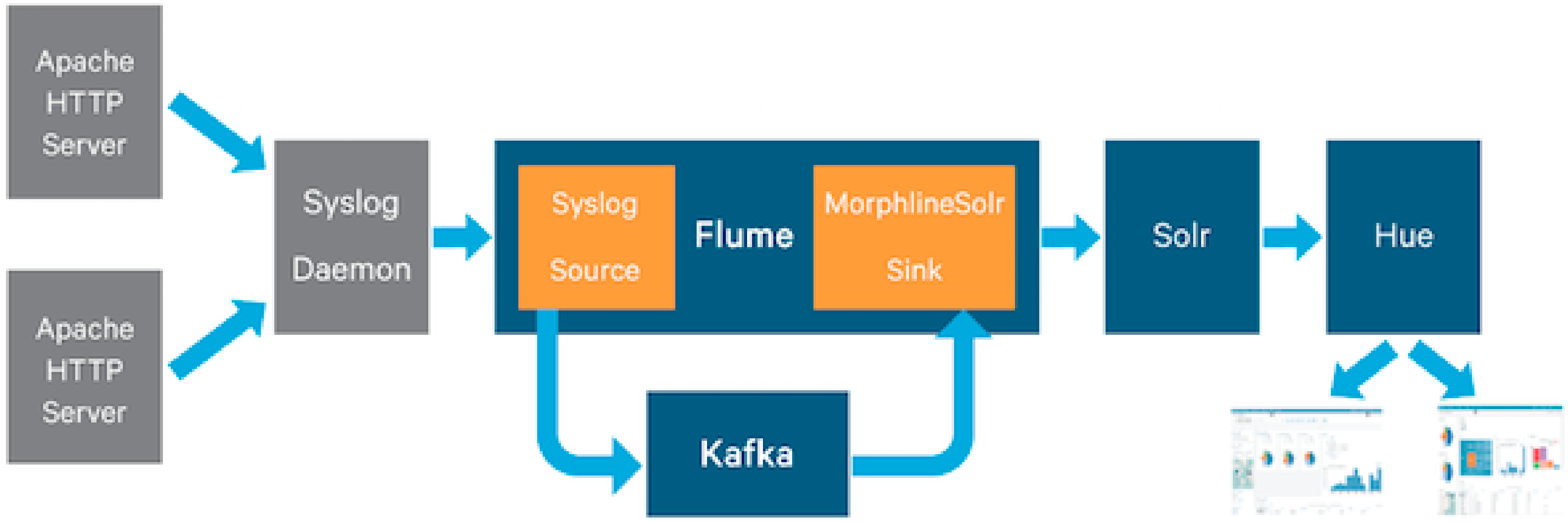
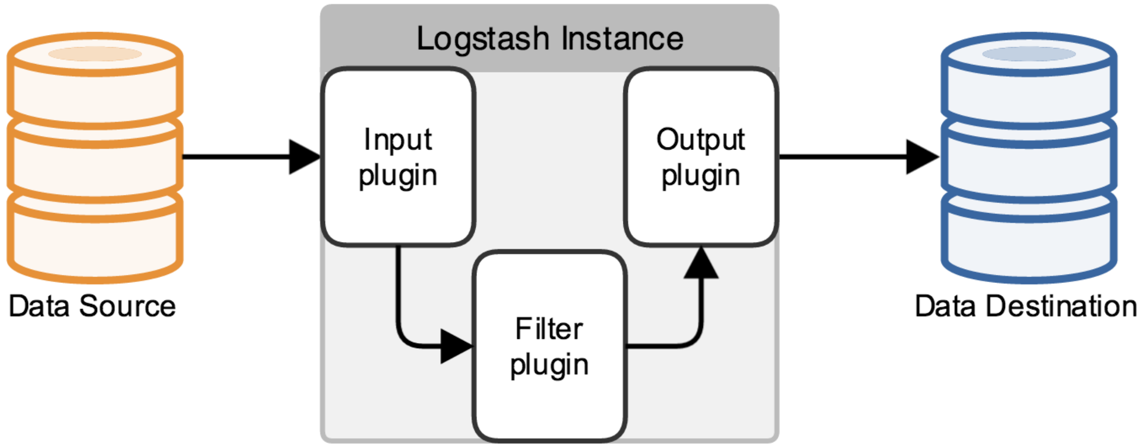


Figure source: https://blog.cloudera.com/blog/2015/02/how-to-do-real-time-log-analytics-with-apache-kafka-cloudera-search-and-hue/

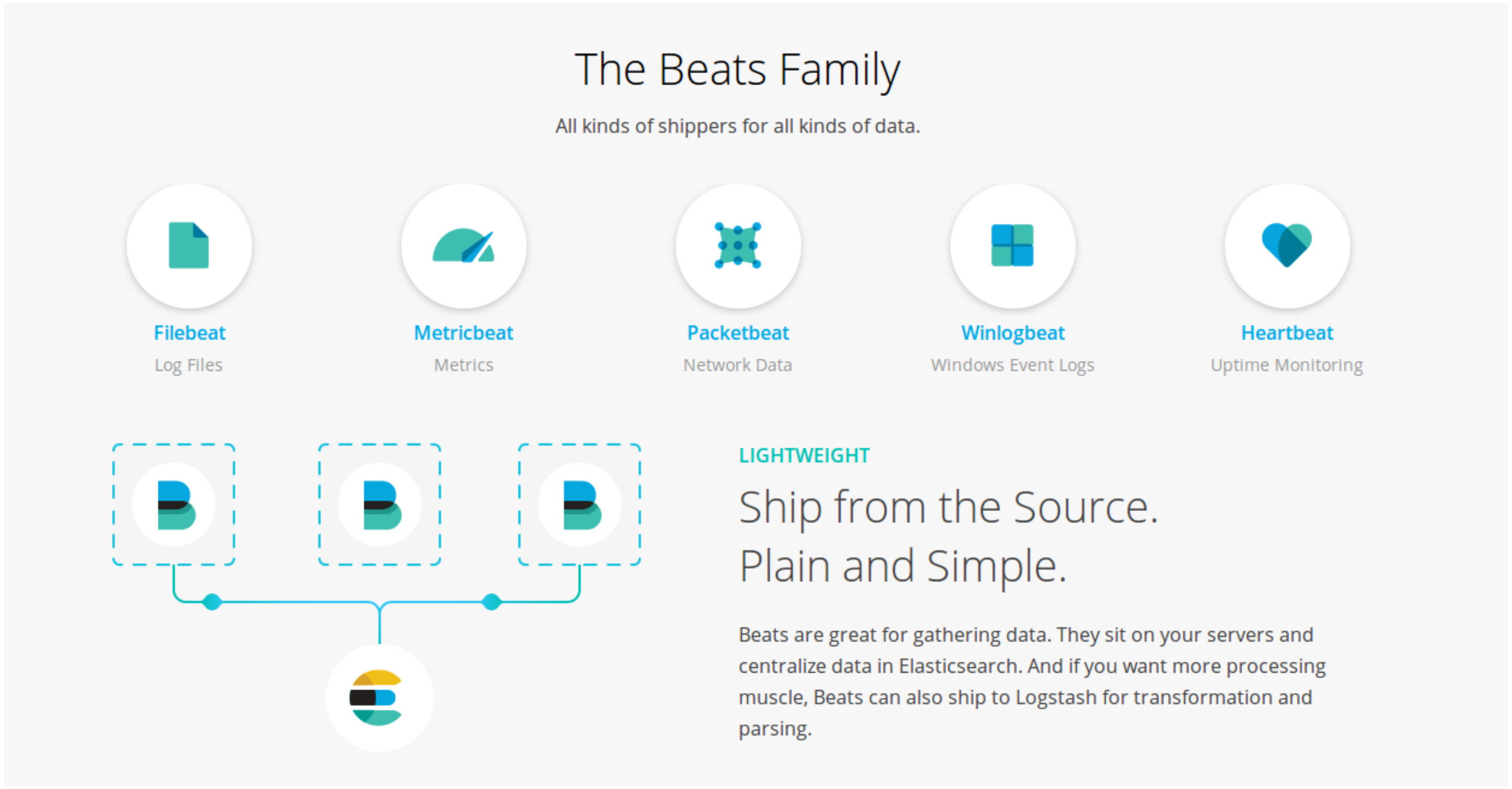
Remember Logstash?



- Codecs: stream filters within inputs or outputs that change data representation
- E.g.: multilines → a single event

Source: https://www.elastic.co/guide/en/logstash/current/advanced-pipeline.html

Using Beat to collect data



The Beats Family

All kinds of shippers for all kinds of data.

LIGHTWEIGHT

Ship from the Source.
Plain and Simple.

Beats are great for gathering data. They sit on your servers and centralize data in Elasticsearch. And if you want more processing muscle, Beats can also ship to Logstash for transformation and parsing.

https://www.elastic.co/products/beats

Using Fluent-bit for constrained devices

- Lightweight for constrainted devices
- Part of Fluentd ecosystem

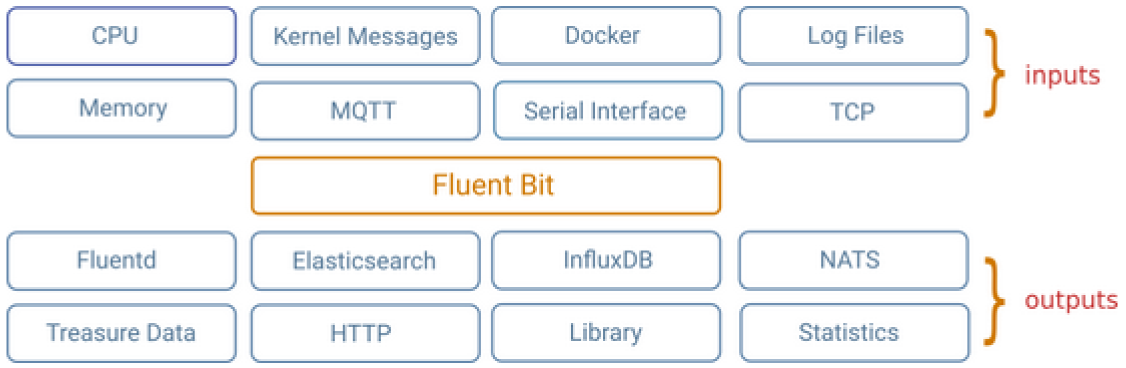


Figure source: https://fluentbit.io/

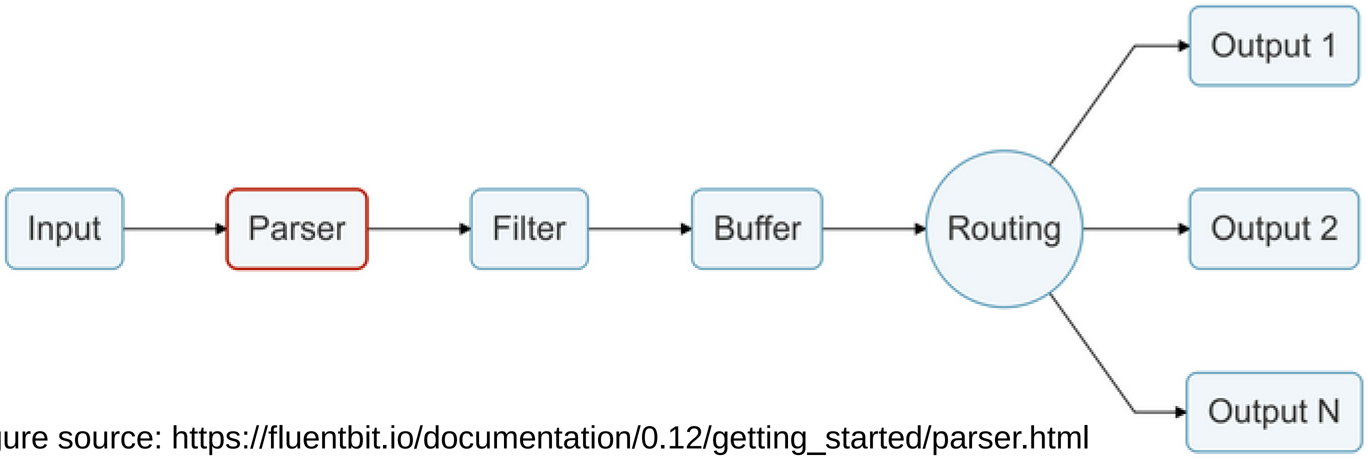
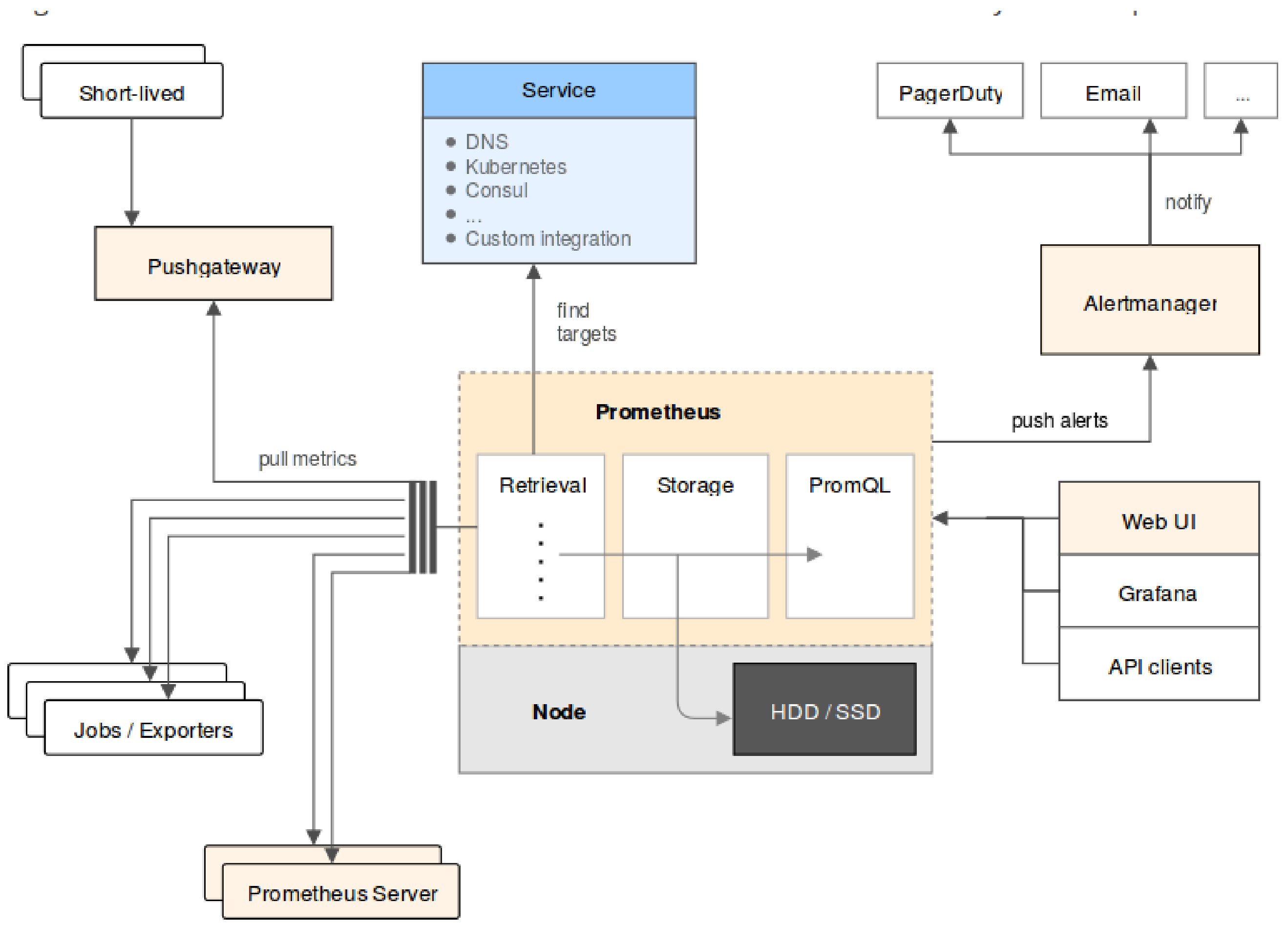


Figure source: https://fluentbit.io/documentation/0.12/getting_started/parser.html

Promethesus Architecture



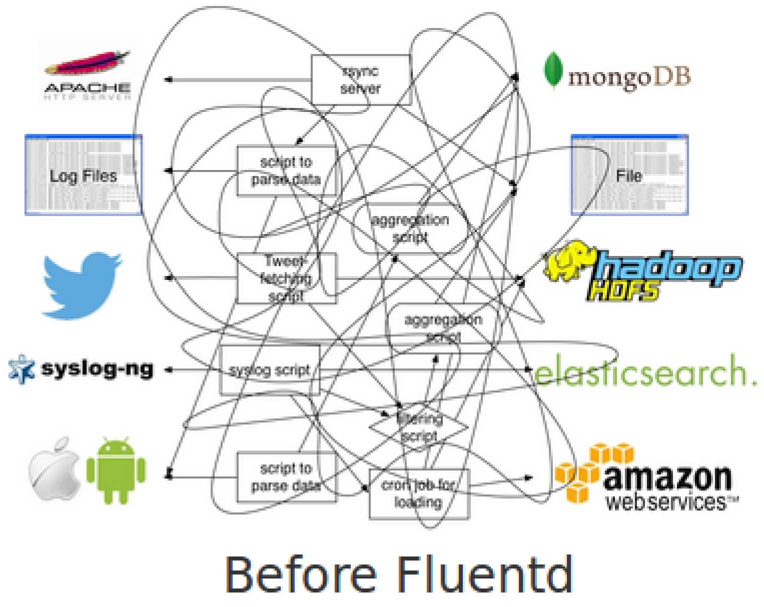
Source: https://prometheus.io/docs/introduction/overview/

Dealing with Cloud application Logs and traces

- In a distributed cloud application
  - Code written in different languages
  - Components deployed in distributed machines
- Key issues
  - Interoperability (format)
  - Scalability
  - Correlation across layers and systems
  - Multi programming language

Example: Fluentd

- Integrated monitoring and logging
- Widely used in cloud systems



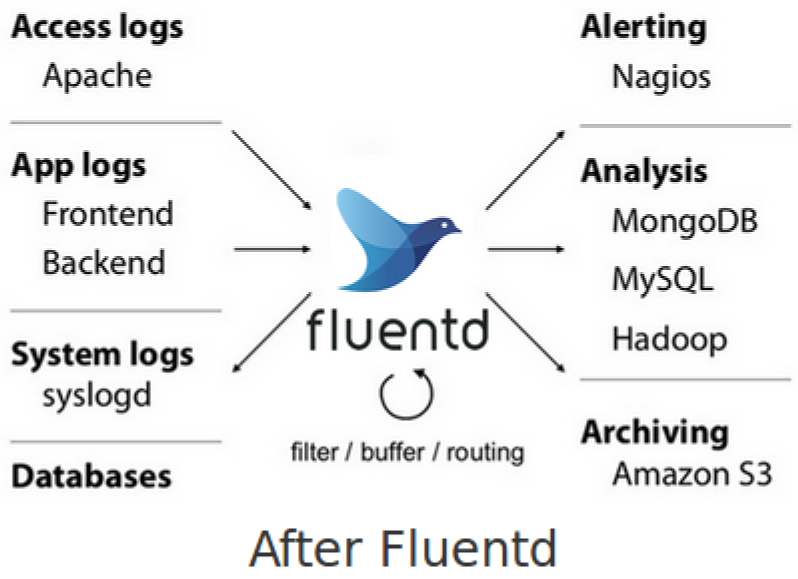
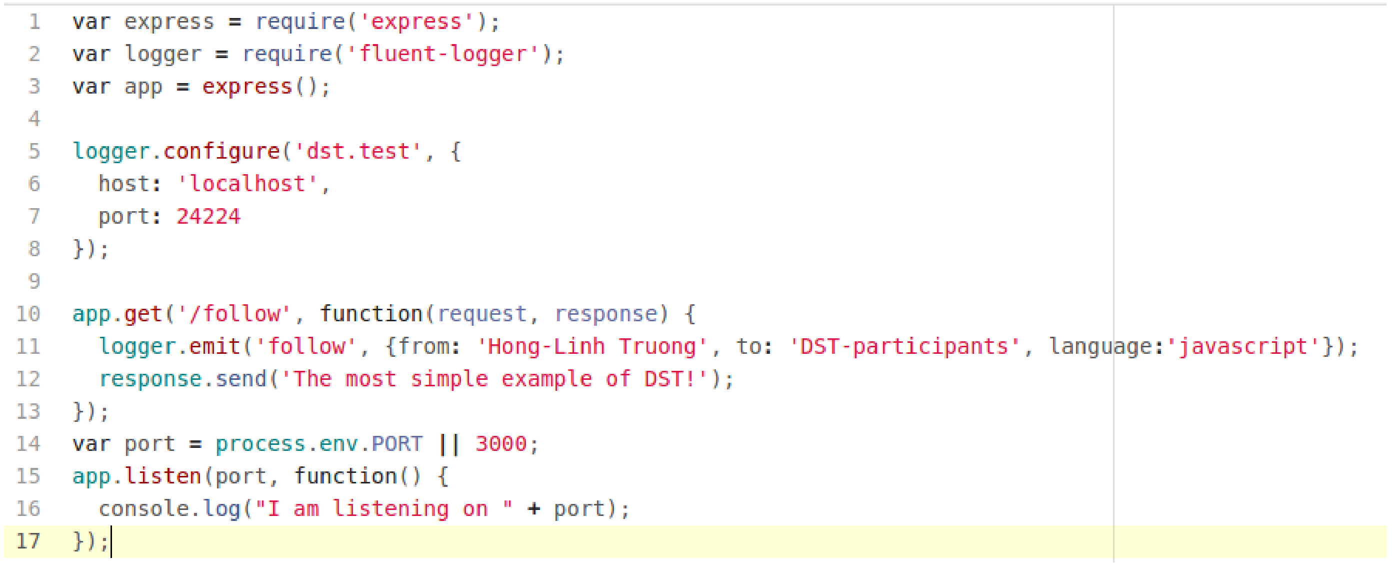
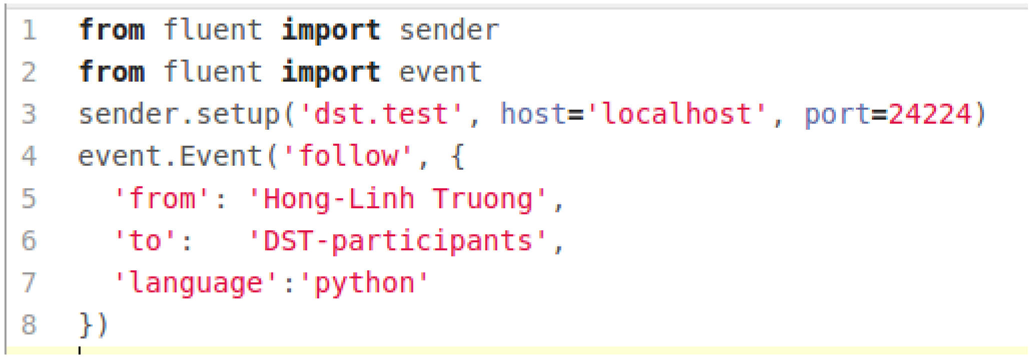


Figure source: https://www.fluentd.org/architecture

Hello World example



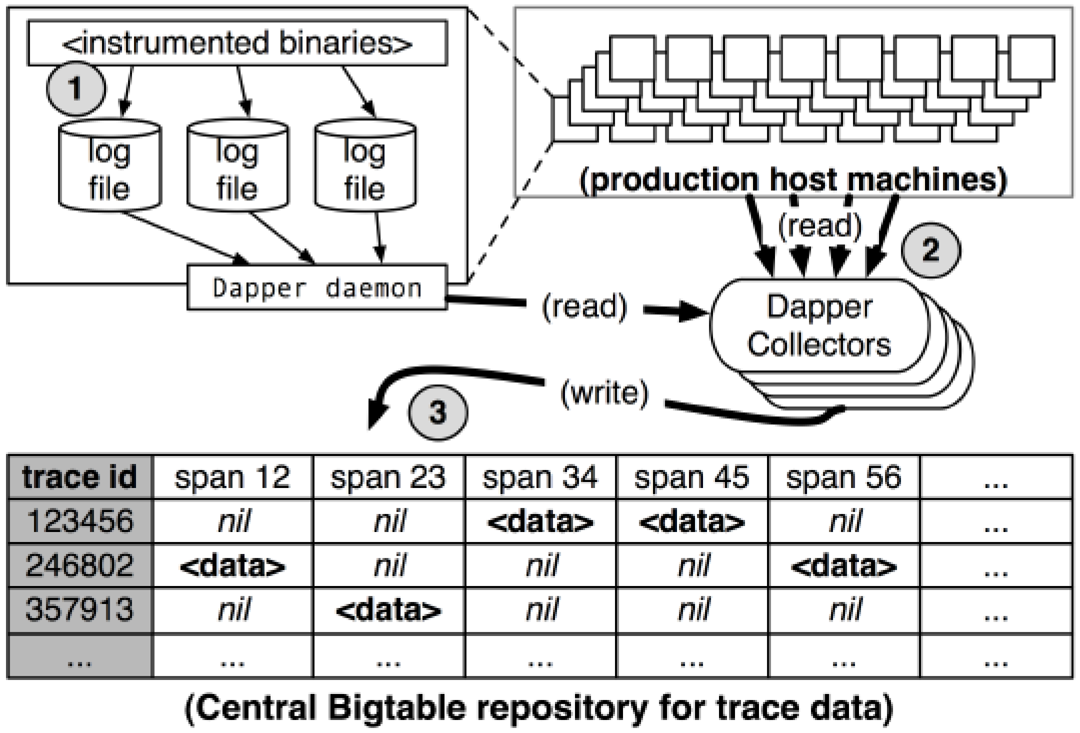
```javascript
var express = require('express');
var logger = require('fluent-logger');
var app = express();

logger.configure('dst.test', {
  host: 'localhost',
  port: 24224
});

app.get('/follow', function(request, response) {
  logger.emit('follow', {from: 'Hong-Linh Truong', to: 'DST-participants', language:'javascript'});
  response.send('The most simple example of DST!');
});
var port = process.env.PORT || 3000;
app.listen(port, function() {
  console.log("I am listening on " + port);
});
```



```python
from fluent import sender
from fluent import event
sender.setup('dst.test', host='localhost', port=24224)
event.Event('follow', {
  'from': 'Hong-Linh Truong',
  'to':   'DST-participants',
  'language':'python'
})
```

Tracing: Google Dapper

- Distributed tracing
- Open source: https://zipkin.io/



trace id	span 12	span 23	span 34	span 45	span 56	...
123456	*nil*	*nil*	**<data>**	**<data>**	*nil*	...
246802	**<data>**	*nil*	*nil*	*nil*	**<data>**	...
357913	*nil*	**<data>**	*nil*	*nil*	*nil*	...
...	...	...	...	...	...	...

Figure 5: An overview of the Dapper collection pipeline.

Figure source: Dapper, a Large-Scale Distributed Systems Tracing Infrastructure, https://research.google.com/pubs/pub36356.html

Tracing: JAEGER from Uber

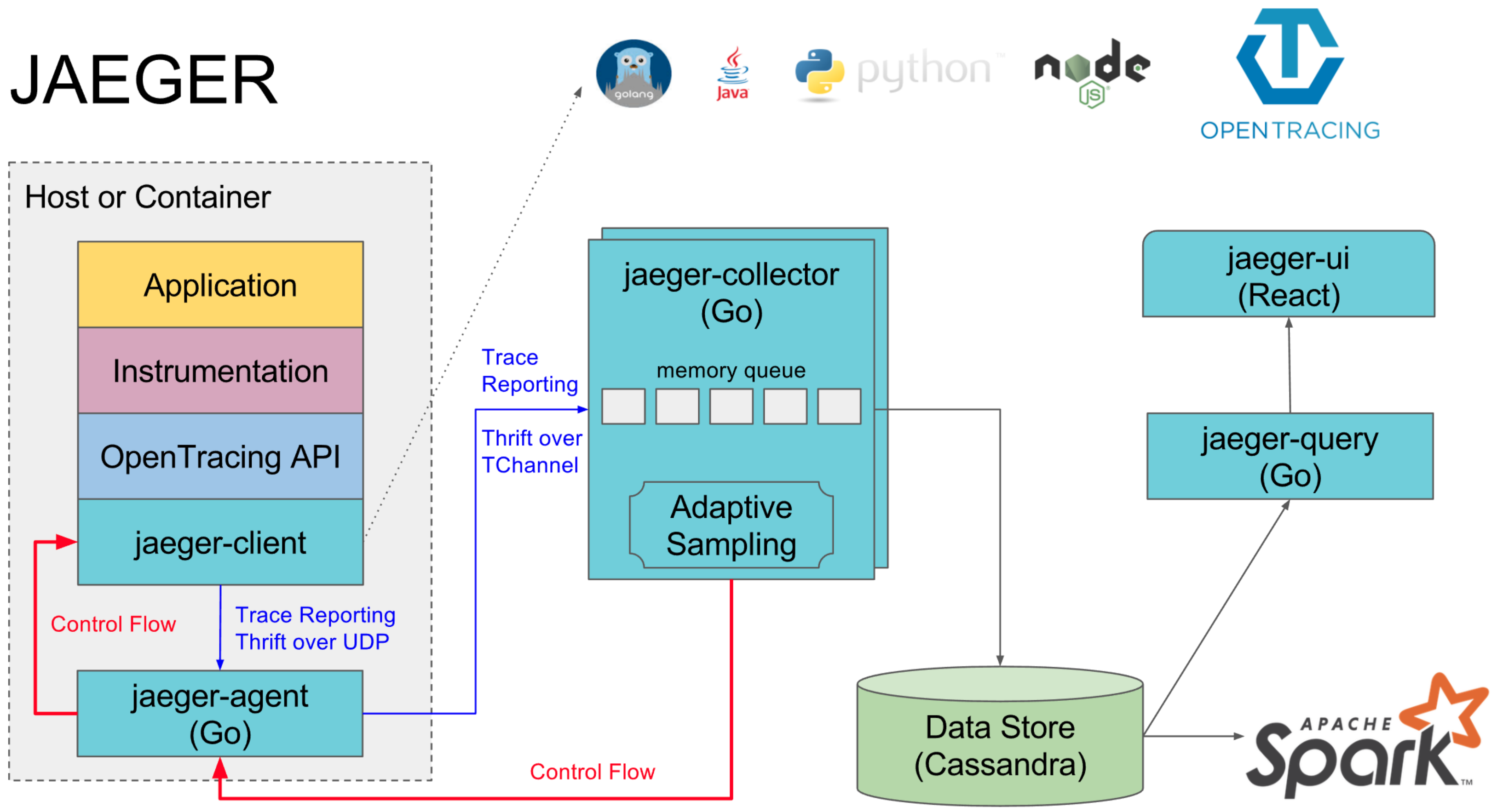
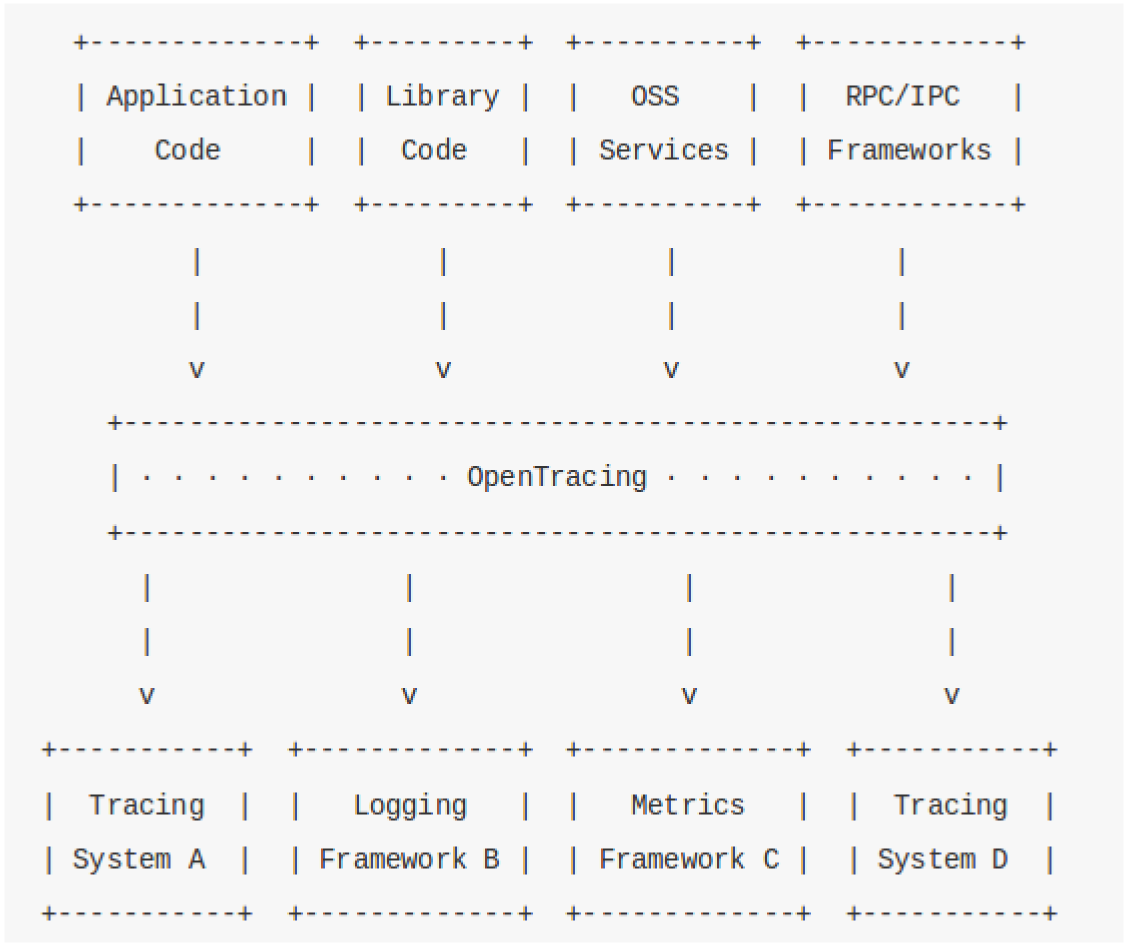


Figure source: https://www.jaegertracing.io/docs/architecture/

OpenTracing concept view



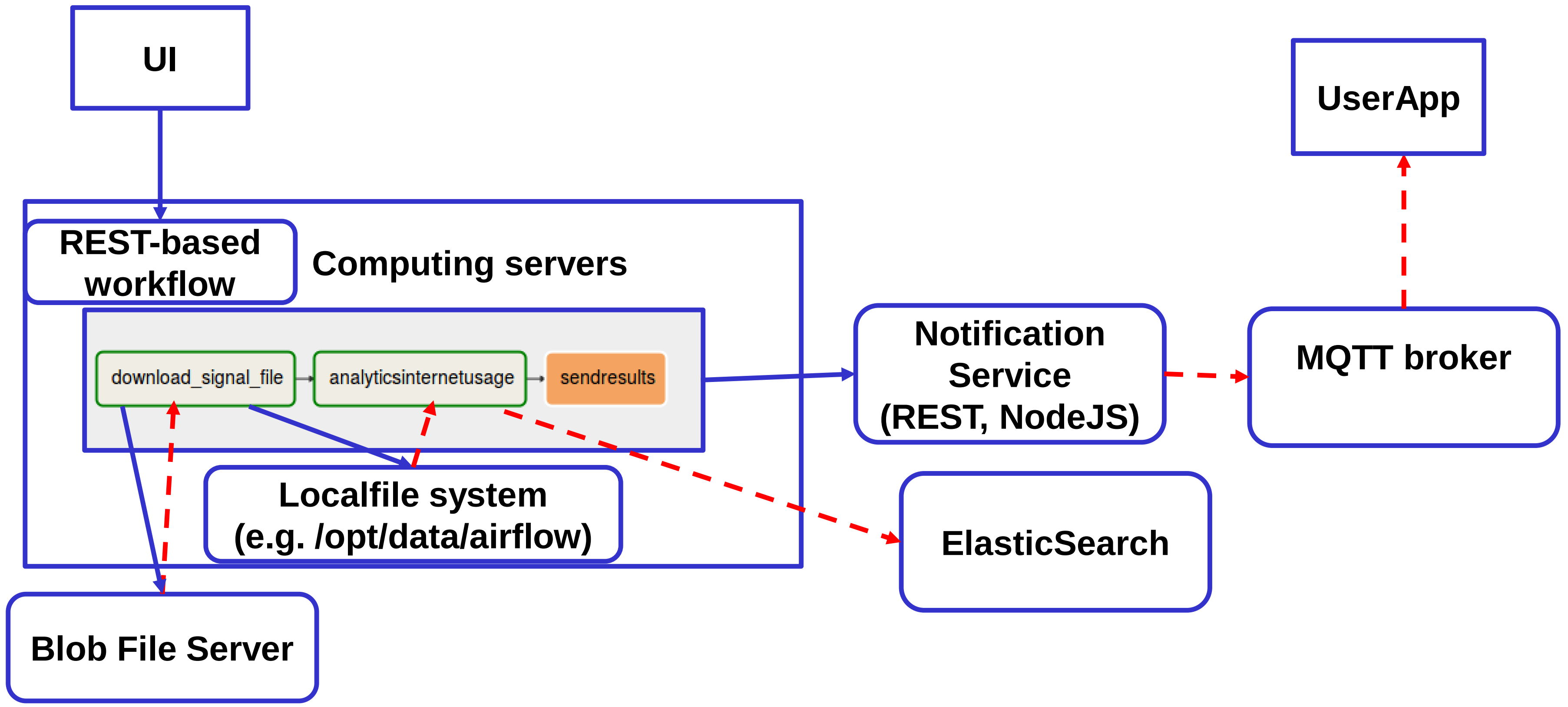
```
  +-------------+  +---------+  +----------+  +------------+
  | Application |  | Library |  |   OSS    |  |  RPC/IPC   |
  |    Code     |  |  Code   |  | Services |  | Frameworks |
  +-------------+  +---------+  +----------+  +------------+
         |              |             |             |
         |              |             |             |
         v              v             v             v
    +-----------------------------------------------------+
    | · · · · · · · · · · OpenTracing · · · · · · · · · · |
    +-----------------------------------------------------+
      |               |                |               |
      |               |                |               |
      v               v                v               v
+-----------+  +-------------+  +-------------+  +-----------+
|  Tracing  |  |   Logging   |  |   Metrics   |  |  Tracing  |
| System A  |  | Framework B |  | Framework C |  | System D  |
+-----------+  +-------------+  +-------------+  +-----------+
```

Source: http://opentracing.io/documentation/pages/instrumentation/common-use-cases.html

Quick check:

What are key issues if you want to monitor cloud applications across data centers?

Food for thoughts

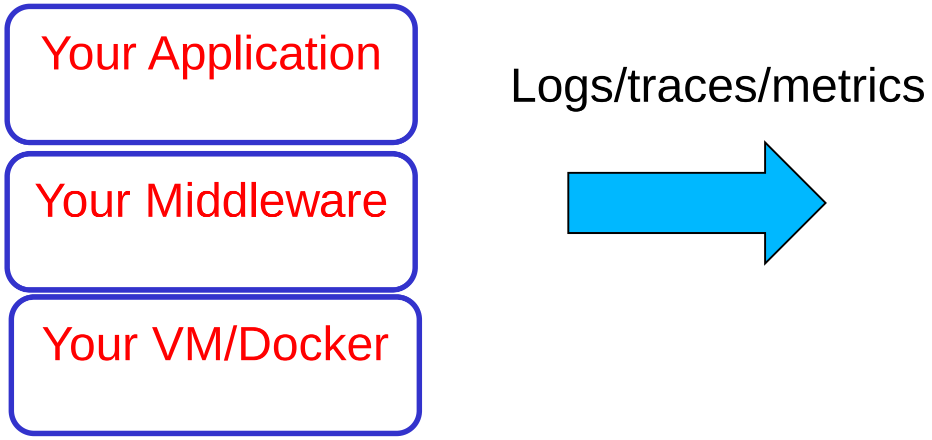


- How would you do the monitoring?

Build instrumentation and monitoring for your cloud application/services

Do a real-world test!

Instrumentation and monitoring (aspects, annotation, etc.)



Cloud monitoring services

STACKDRIVER MONITORING

For applications running on Google Cloud Platform and Amazon Web Services

TRY IT FREE VIEW DOCUMENTATION

SCALYR

fluentd

DATADOG

Amazon CloudWatch

Amazon CloudWatch is a monitoring service for AWS cloud resources

Scout App Monitoring Server Monitoring

Key takeway (e.g. for exams ☺)

To design your monitoring and instrumentation solutions together with communication and storage system middleware

Try to analyze existing examples and tools to see the complexity of programming dynamic features and monitoring (not just simple AOP)

Tracing is quite complex but monitoring and logging you should spend effort to learn!

Summary

- Dynamic features programming required by complex distributed software
- Dynamicity programming can be achieved through different design and runtime activities
- There are different tools for programming dynamic features, but we need to combine different techniques
- Understanding which instrumentation techniques should be used and what will be instrumented is crucial
- Three points you should master
  - Basic techniques (inspection, reflection, AOP, etc)
  - Integration of basic techniques in programming tasks and CI
  - Large-scale cloud instrumentation, monitoring and analysis

Further materials

- http://eclipse.org/aspectj/doc/released/progguide
- http://docs.spring.io/spring/docs/current/spring-framework-reference/
- http://docs.oracle.com/javase/tutorial/java/annotations/
- http://commons.apache.org/proper/commons-bcel/
- https://jcp.org/en/jsr/detail?id=160
- http://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html
- http://docs.oracle.com/javase/8/docs/technotes/guides/reflection
- http://docs.oracle.com/javase/tutorial/reflect/index.html
- https://medium.com/opentracing/towards-turnkey-distributed-tracing-5f4297d1736

Thanks for your attention

Hong-Linh Truong
Faculty of Informatics, TU Wien
hong-linh.truong@tuwien.ac.at
http://www.infosys.tuwien.ac.at/staff/truong