

Programming Dynamic Features and Monitoring Distributed Software Systems

Hong-Linh Truong
Distributed Systems Group, TU Wien

truong@dsg.tuwien.ac.at
dsg.tuwien.ac.at/staff/truong
[@linhsolar](https://www.linkedin.com/in/linhsolar)

OVERVIEW

Things that make a difference

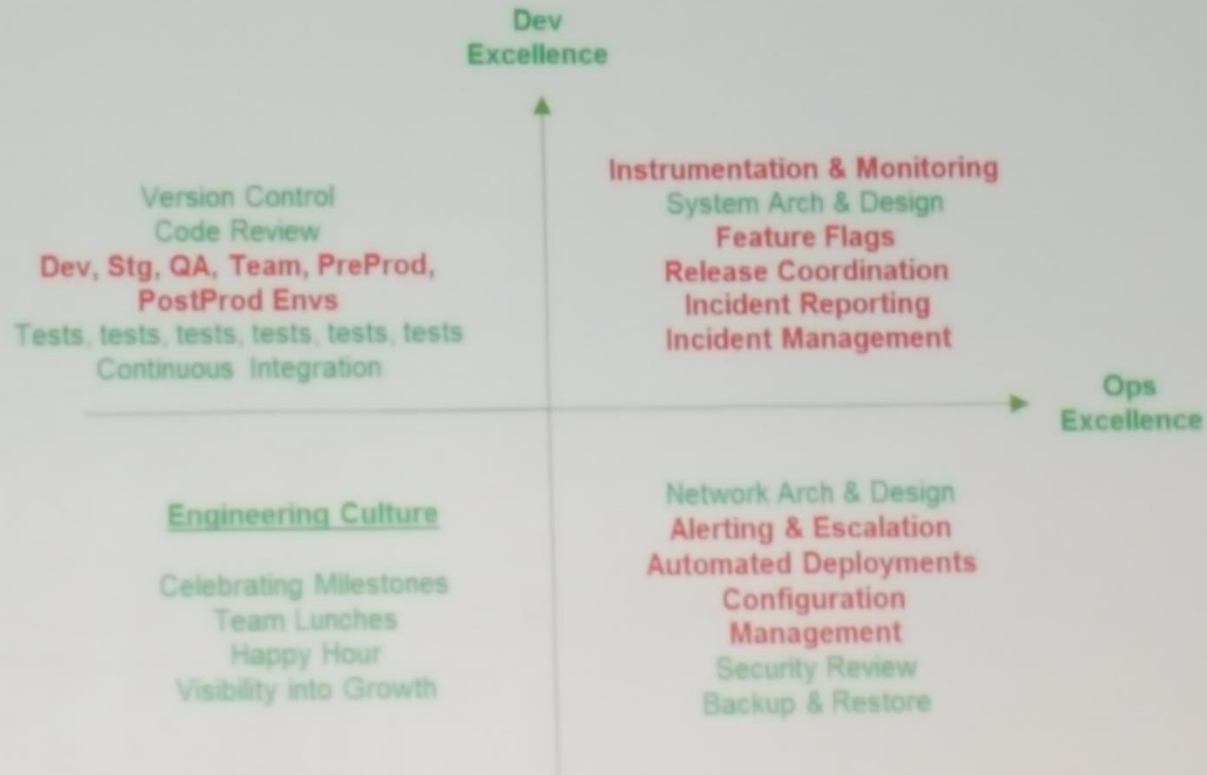


Photo from “Lessons Learned From a Fast Growing Startup”, Arul Kumaravel, Vice President, Engineering, Grab at CloudAsia2016, Singapore

„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>

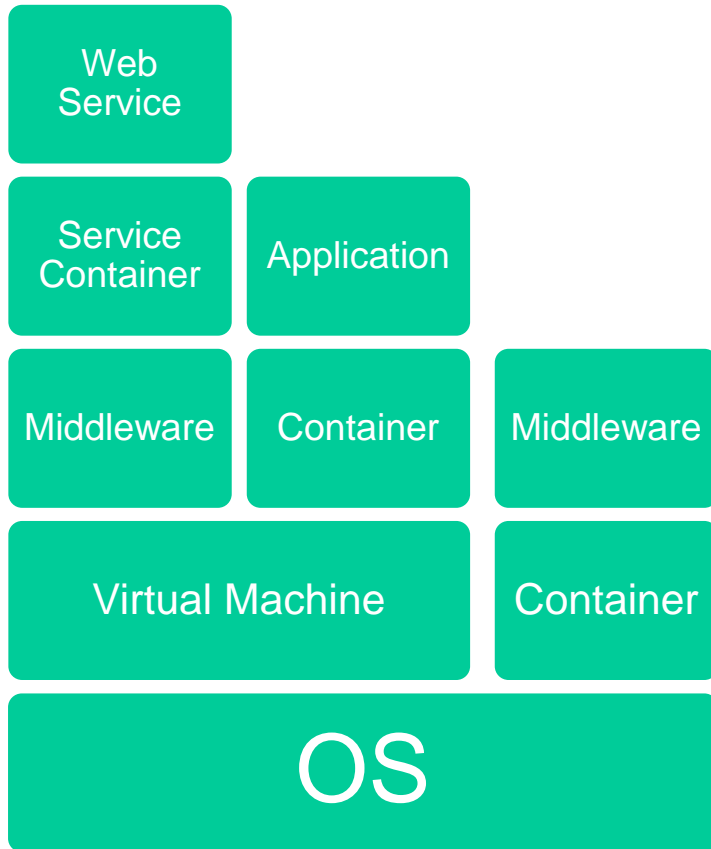
Programming dynamic features enable instrumentation and monitoring



Key techniques for today's lecture

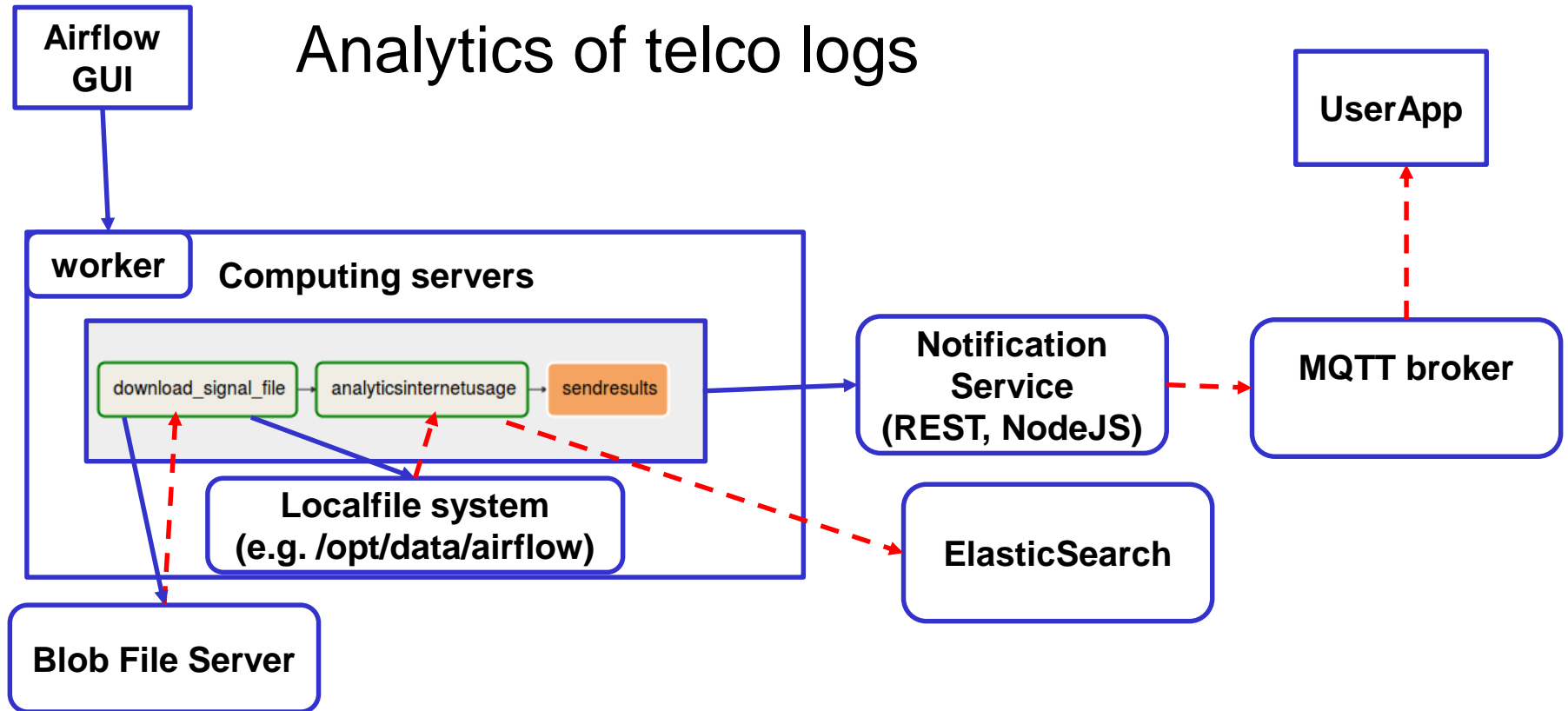
- Big picture in full-stack and large-scale monitoring
- Dynamic features
- Code inspection
 - Dynamic loading
 - Reflection
 - Dynamic proxy
- Instrumentation and Program Analysis
- Annotation
- Aspect-oriented Programming

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

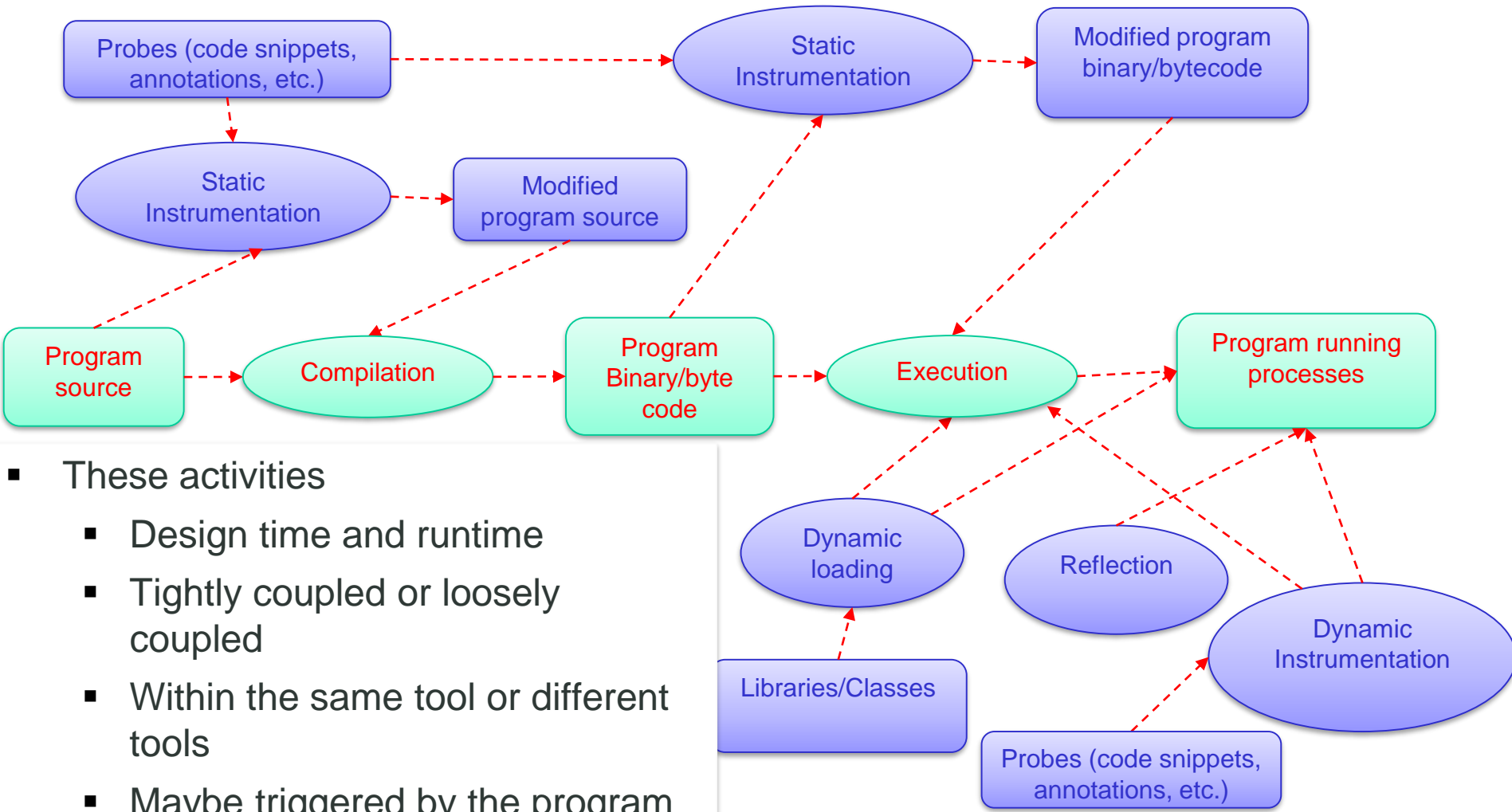
Dynamicity needs (1)

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

Grab instrumentation and monitoring

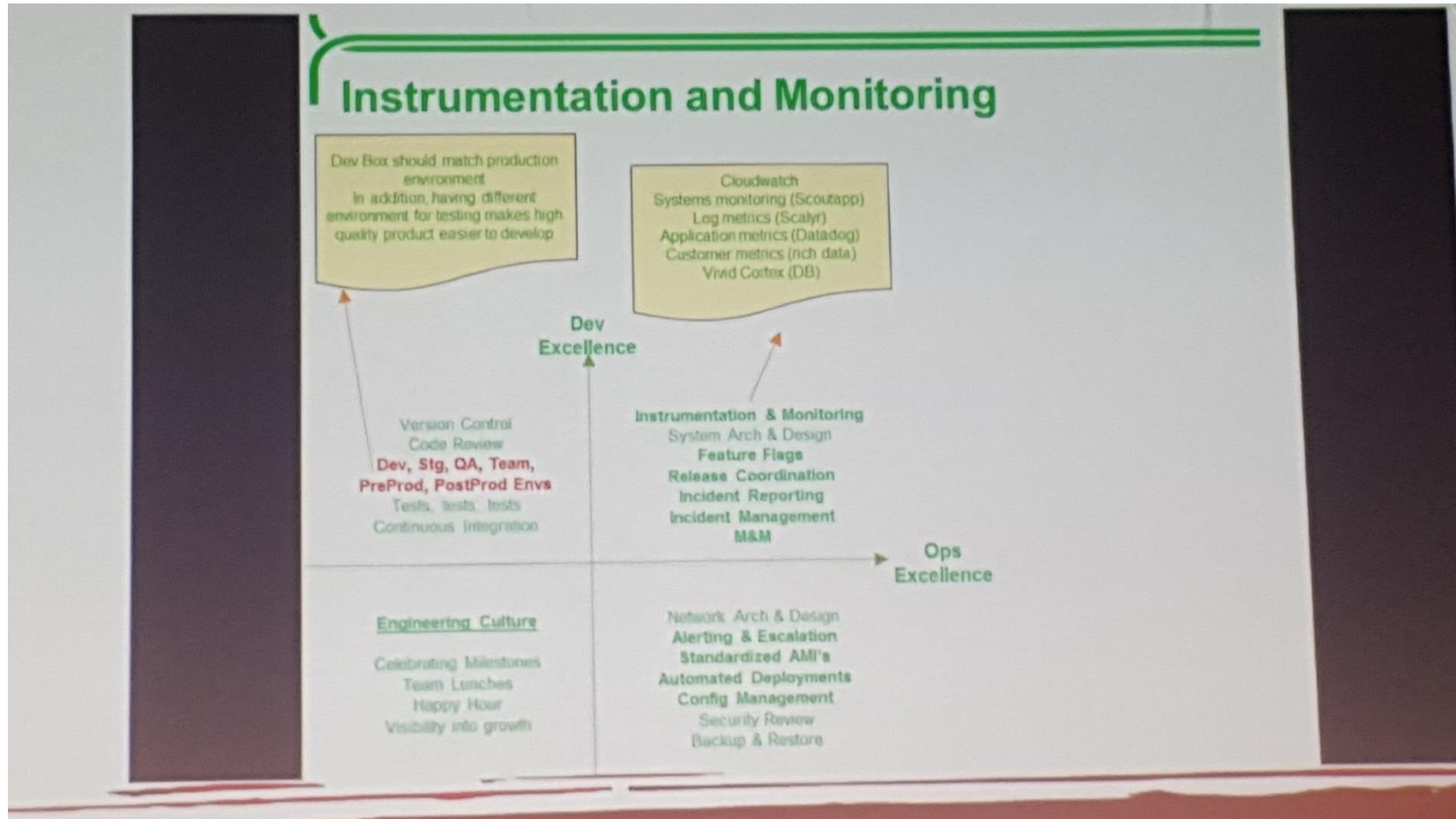


Photo from "Lessons Learned From a Fast Growing Startup", Arul Kumaravel, Vice President, Engineering, Grab at CloudAsia2016, Singapore

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)`

Bytecode/Binary inspection

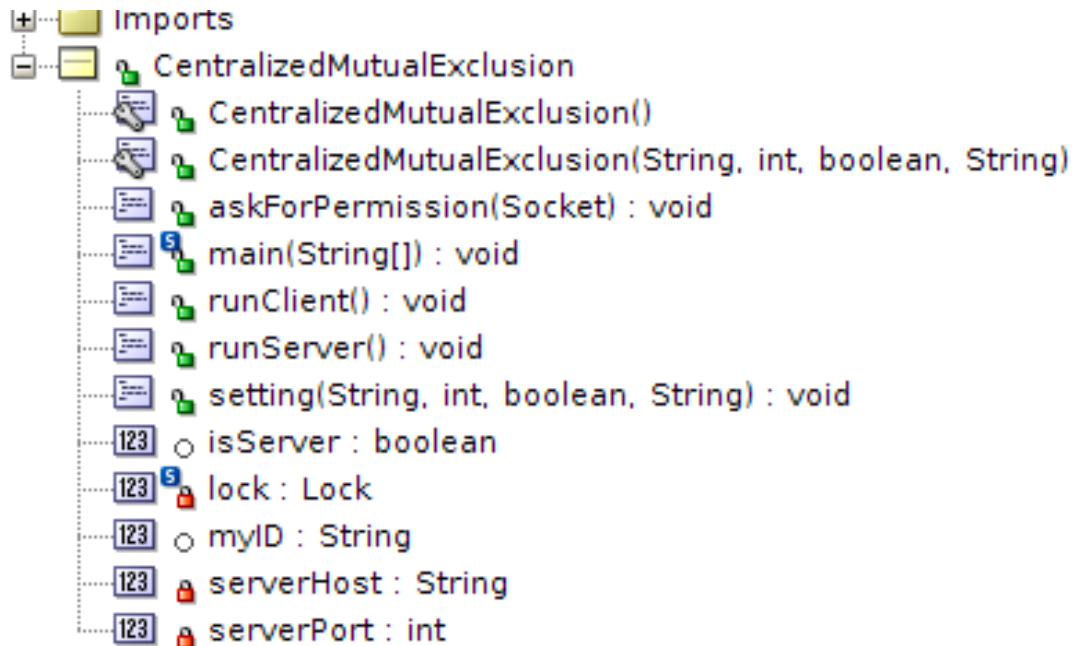
- Read and build program structures from bytecode
 - Example, using `javap -c` to see Java bytecode
- Tools to process bytecodes
 - Javassist (<https://github.com/jboss-javassist/javassist/releases>)
 - BCEL (<http://commons.apache.org/proper/commons-bcel/>)
 - CGLIB (<https://github.com/cglib/cglib>)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

Reflection

- Allows running program code to introspect its own definition (e.g., iterate methods and fields of a class)
- Additionally allows for creation and invocation of object instances at runtime
- Basic feature in most current object-oriented languages

Example: Reflection in Java

Source code



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");
                p = runServerMethod.invoke(instance);
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            } catch (InstantiationException e) {
                e.printStackTrace();
            }
        }
    }
}

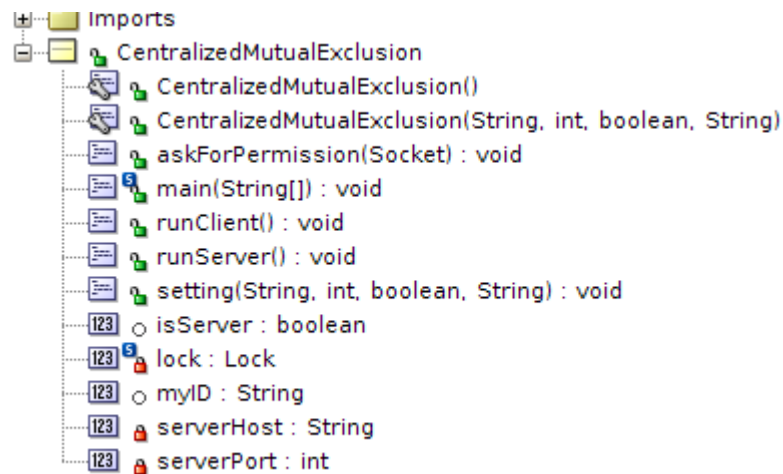
```

Read structures

Call methods

Example: Reflection in Java

Source code



```

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

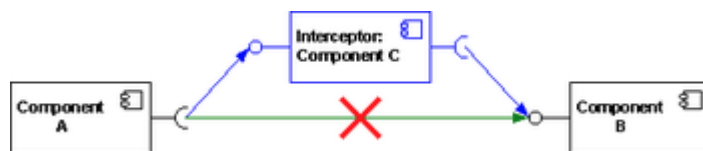
Inspection output

Reflection – Advantages and Disadvantages

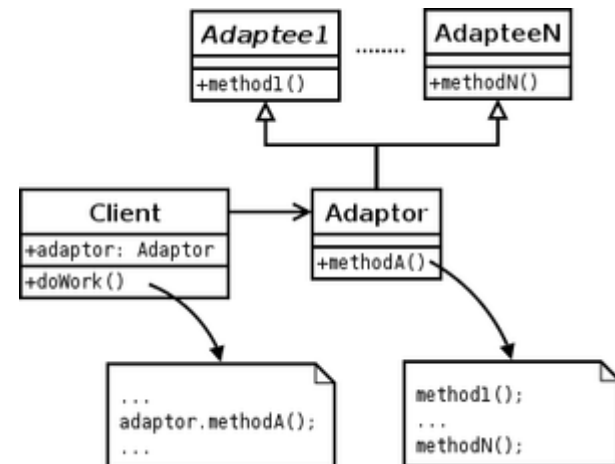
- **Advantages:**
 - Built-in features in most modern programming languages
 - Allows for flexible applications and design patterns
- **Disadvantages:**
 - Complex solutions are difficult to write and error-prone (method names in strings, etc.)
 - Performance degradation
 - Security restrictions
 - Reflection 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



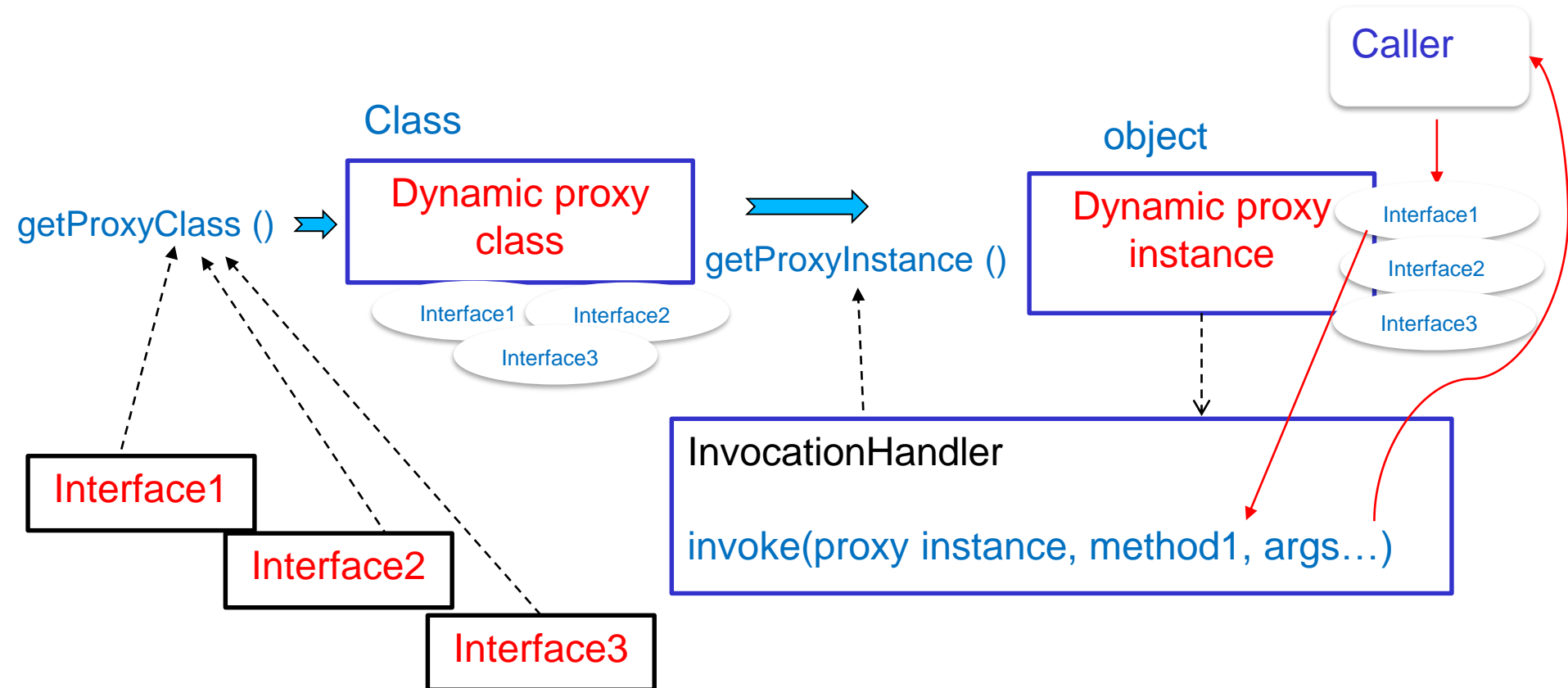
Source: http://en.wikipedia.org/wiki/Interceptor_pattern



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

```

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?

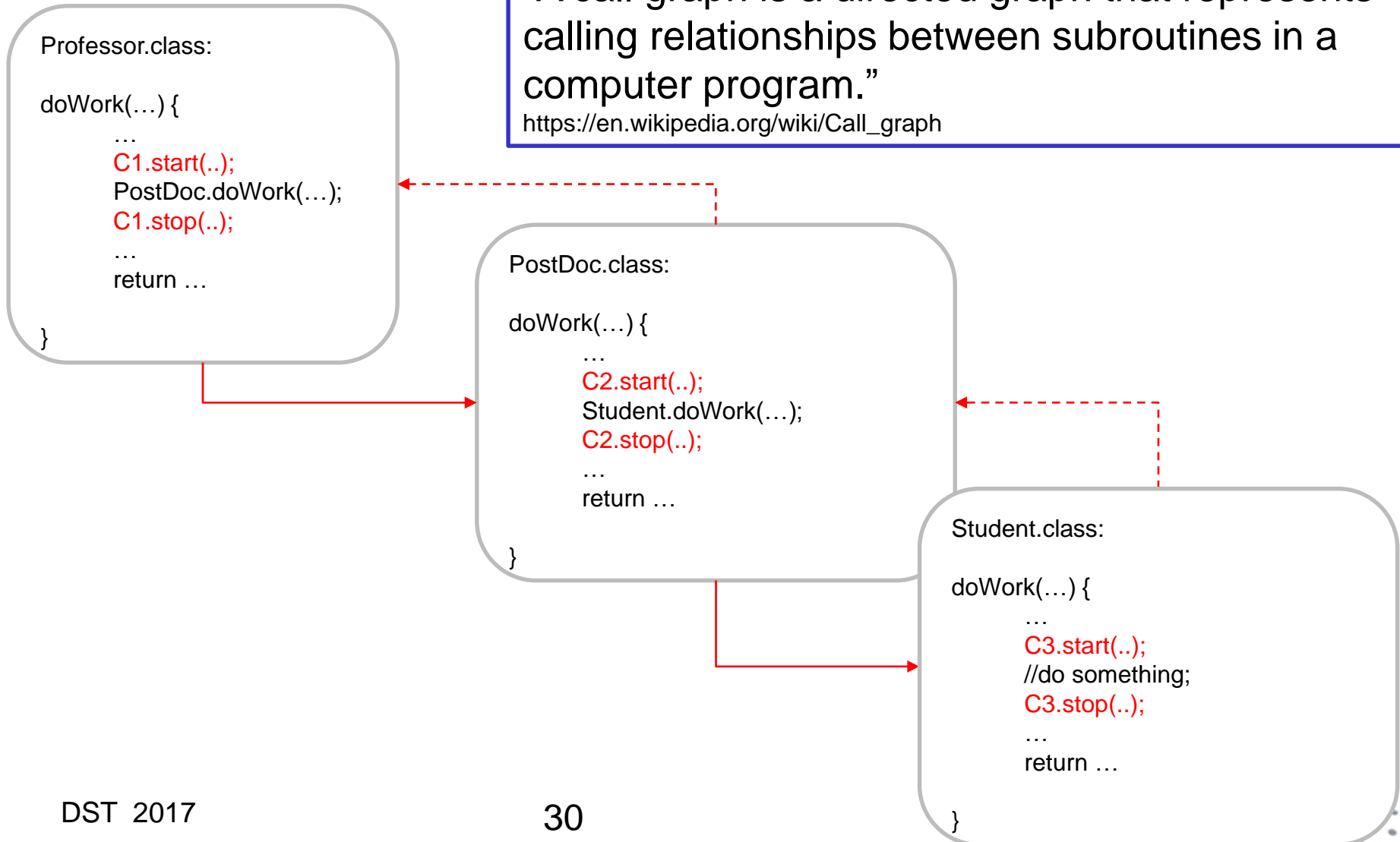
- **At any join point:** a point in the control flow of a program
- Examples:
 - method calls, entry/exit of method body, statements (set/get, assignment, etc.)

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

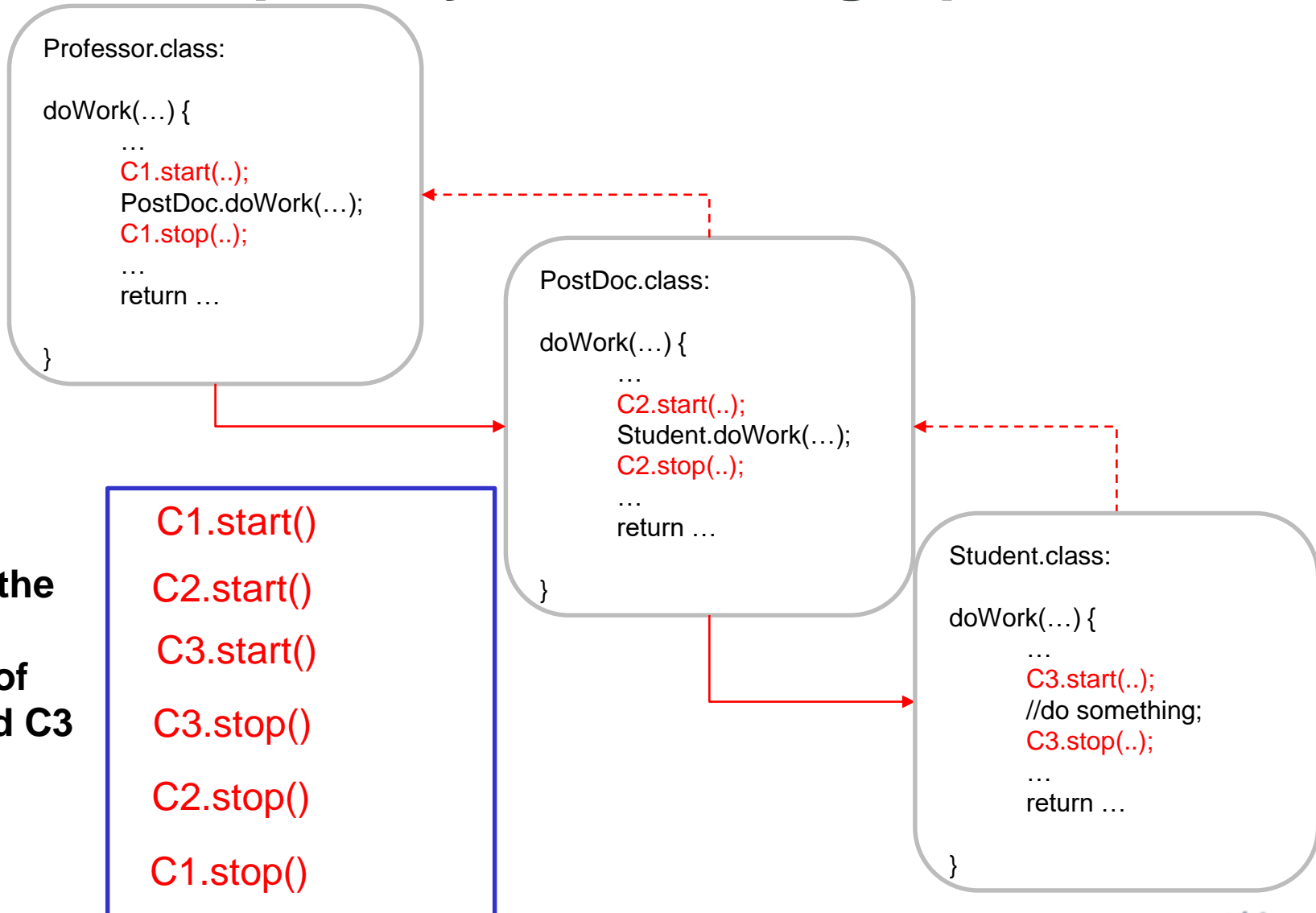
Example: Dynamic call graph

“A call graph is a directed graph that represents calling relationships between subroutines in a computer program.”

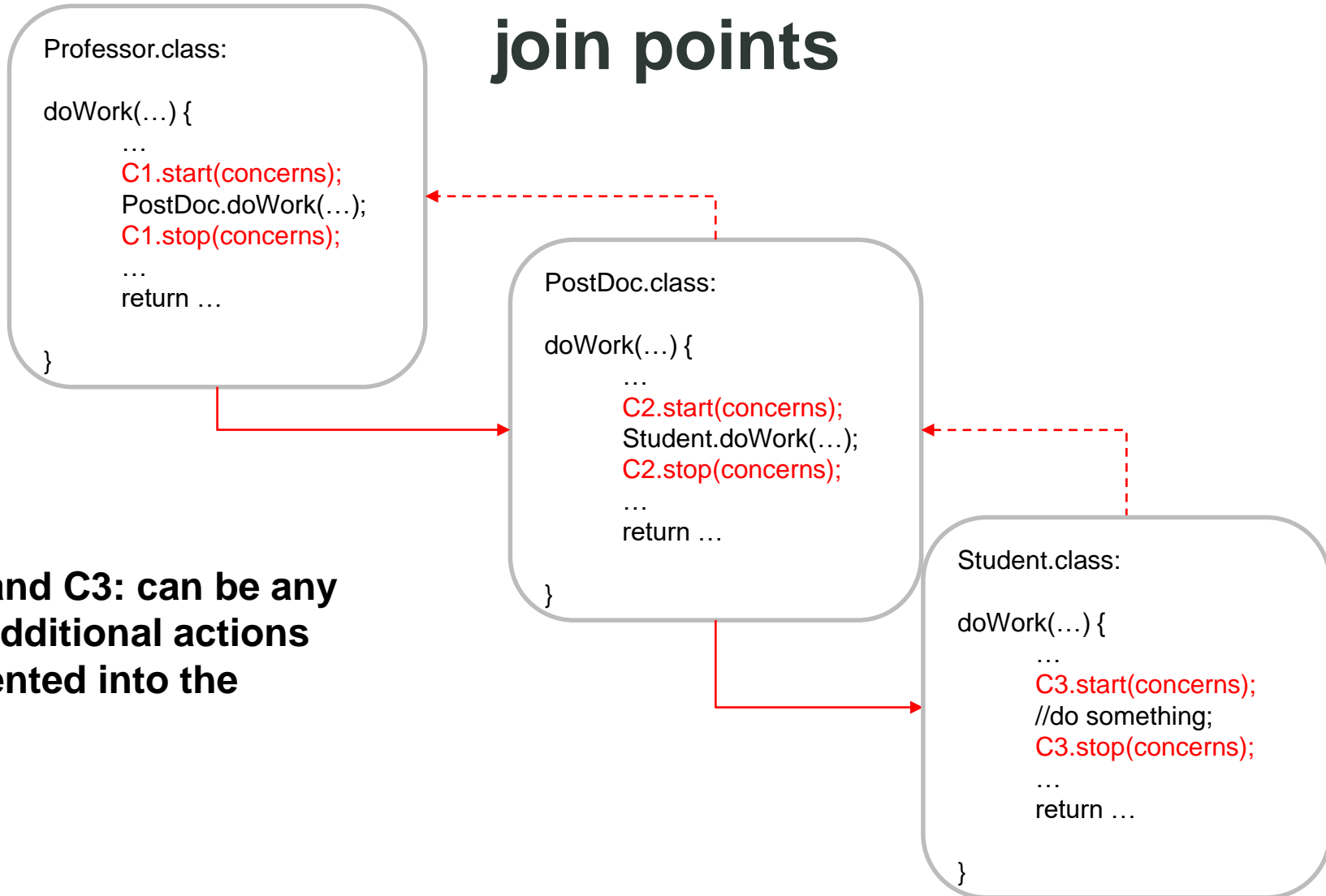
https://en.wikipedia.org/wiki/Call_graph



Example: Dynamic call graph



If we want to deal with certain concerns at join points



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

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

- In most EE programming frameworks
- Some Spring annotations (<http://docs.spring.io/spring/docs>)
 - `@Resource`, `@PostConstruct` and `@PreDestroy`

```
@IfProfileValue(name="java.vendor", value="Oracle Corporation")
@Test
public void testProcessWhichRunsOnlyOnOracleJvm() {
    // some logic that should run only on Java VMs from Oracle Corporation
}
```

- JAX-RS (<https://jax-rs-spec.java.net/>)
 - `@Path("scu/{scuid}")`
 - `@Consumes("application/json")`
 - `@Produces("application/xml")`

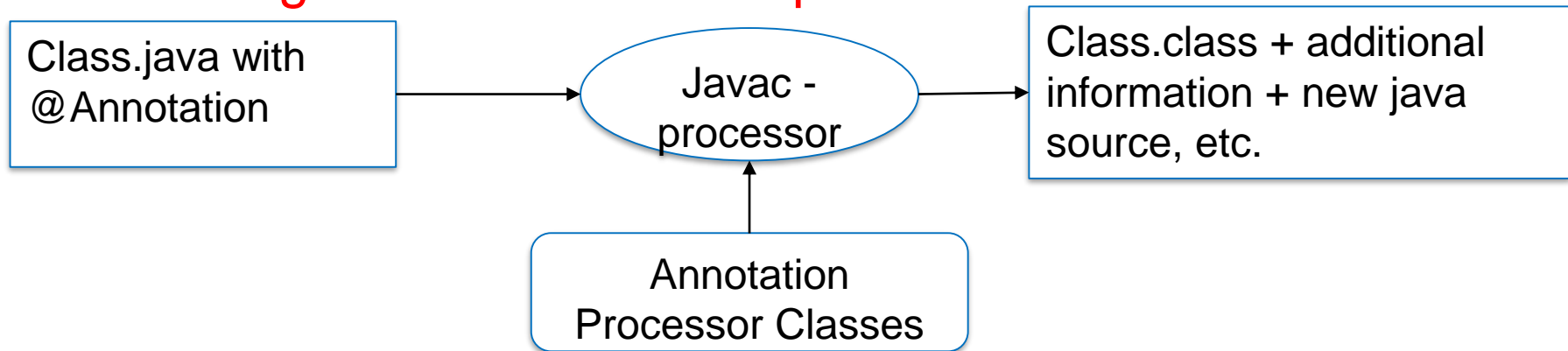
Java Annotation Processing

- Parsing source codes
- Reflection APIs also return Annotation

`Method.class: public Annotation[]
getDeclaredAnnotations()`

- Reading bytecode to get Annotation

Processing Model in Java compilation



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

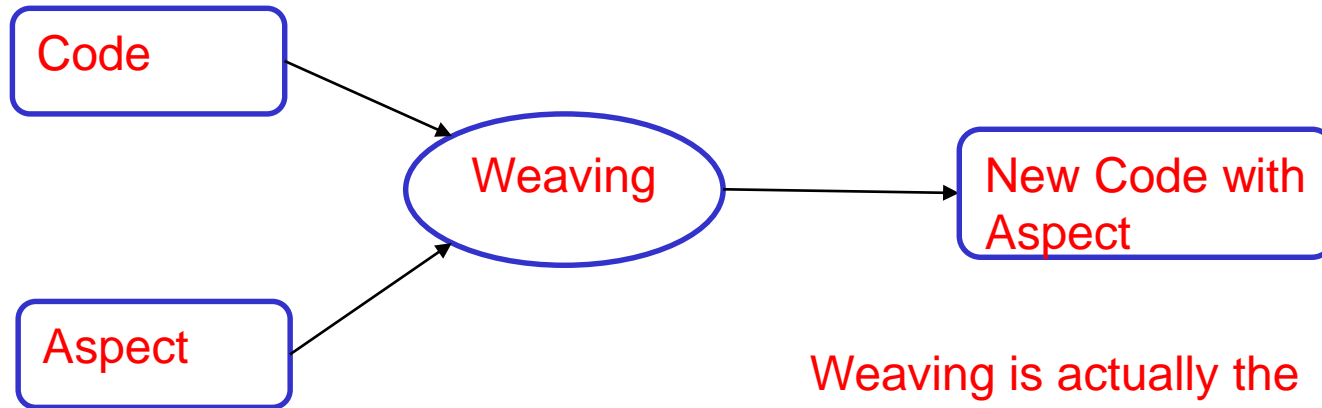
Cross-cutting concerns – when, where and how

- We can use „probes“ instrumented into targeting programs → creating hooks
 - Probes specify code for dealing these concerns
 - Probes create additional actions at runtime
- But we need a 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



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

- **Pointcut**

A set of join points (composed using different operators such as &&, ||, !)

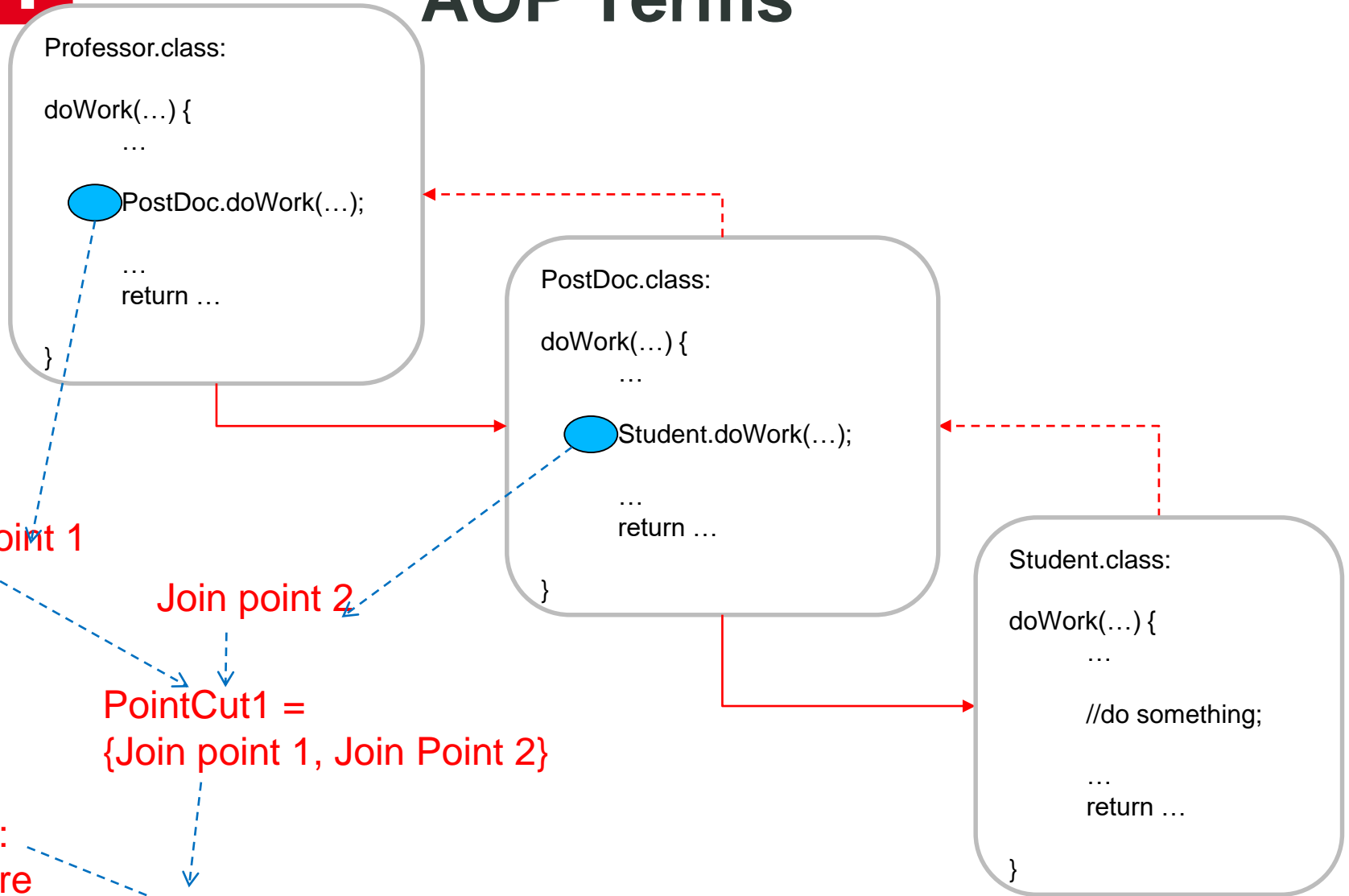
- **Advice**

- Additional action that should be executed at join points in a pointcut

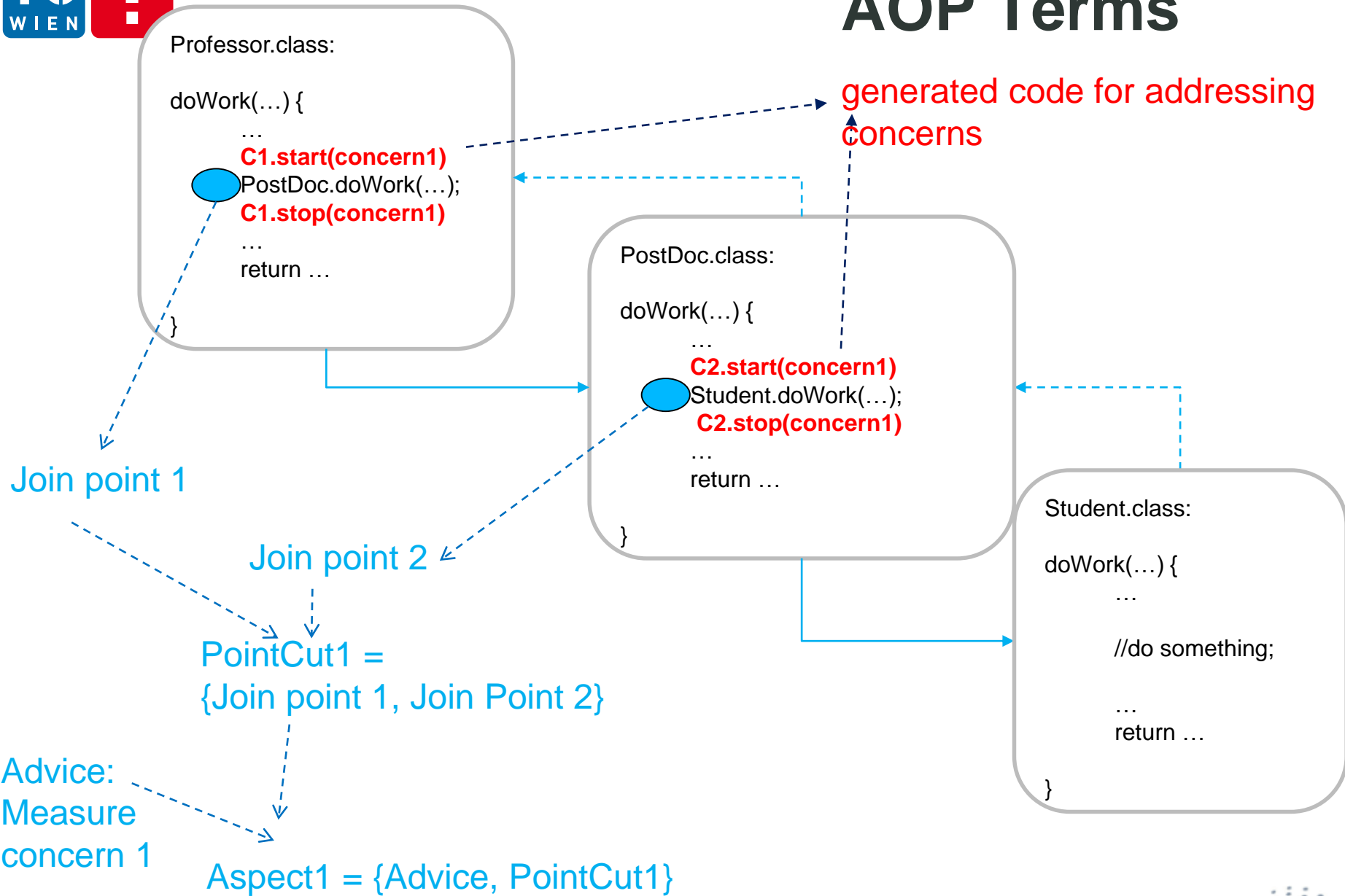
- **Aspect**

- Cross-cutting type and its implementation (advices + others)

AOP Terms



AOP Terms



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)`

Special note: Call vs. Execution Join Points

- Avoid to confuse **call** with **execution** join points for methods
- **Call** matches before or after a method is called (i.e., still in the scope of the caller)
 - E.g., the call join point is the last thing that happens before the method is actually invoked
- **Execution** matches when the method starts to execute (i.e. already in the scope of the callee)
 - E.g., the execution join point is the first thing that happens during method invocation

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

- The process of merging aspects into the program code is called weaving
- Three ways of weaving:
 - **Compile-Time Weaving** (weave as part of source-to-binary compilation)
 - **Binary 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 – implemented via Java agent mechanism)

Compile-Time Weaving

- **Advantages:**
 - No startup performance degradation
 - Allows you to store and see the produced source code (good for debugging, and error tracking)
- **Disadvantages:**
 - Can't weave third-party code (e.g., used libraries)
 - Java: requires usage of special compiler (ajc), which may produce worse regular Java code than your Sun or openjdk compiler

Binary Weaving

- **Advantages:**
 - No startup performance degradation
 - Allows you to store and see the produced binary (eases debugging)
 - Can be used with any compiler
 - Can weave third-party libraries, as long as you can live with permanently modifying them
- **Disadvantages:**
 - Complex when dealing with system libraries

Load-Time Weaving

- **Advantages:**
 - Can be used with any compiler
 - Can sensefully weave pretty much anything, even system libraries
- **Disadvantages:**
 - Slow startup
 - Hard to debug and understand, as the running code exists nowhere outside the classloader

Example of AOP with AspectJ

```

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);
    }
}

```

```

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);
    }
}

```

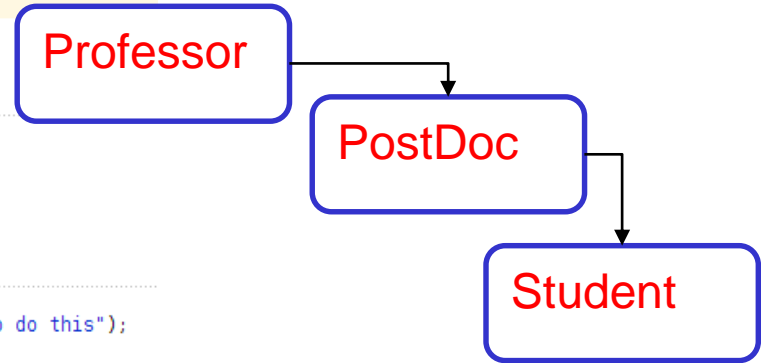
```

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+" ");
    }
}

```



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 ));
    ④ private pointcut methodParameter (String task) :
        (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();
        professor.doWork("Programming hello.java");
        professor.doWork(null);
    }
    ④
```

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");
        new Student().doWork(taskName);
    }
    ②
```

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 graph tracing information

Professor.doWork()

PostDoc.doWork()

Student.doWork()

```

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

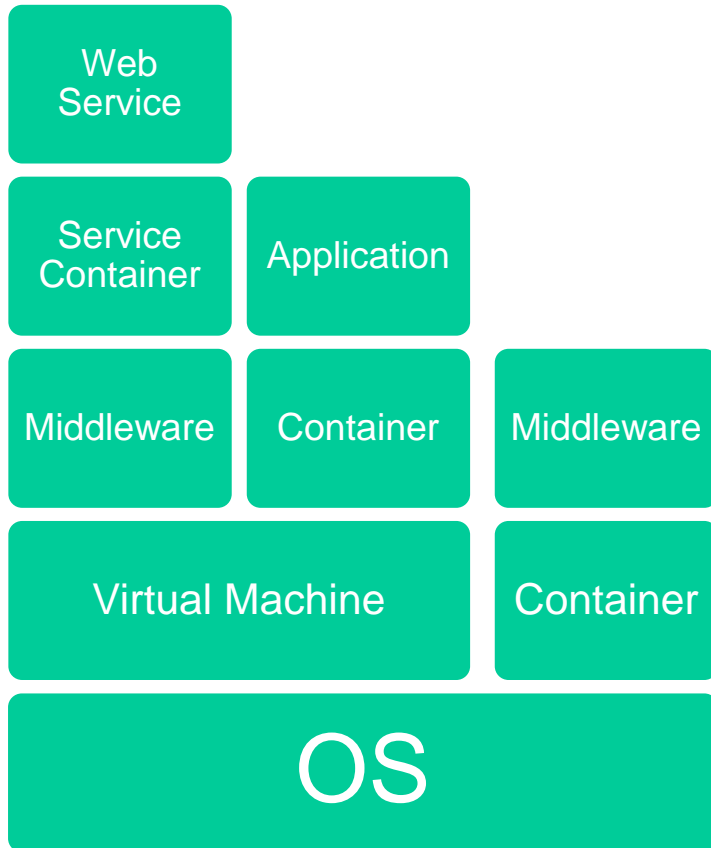
- Not all features are supported
 - String AOP only method execution join points
- Using Java annotation or XML
- Java Annotation
 - @Aspect, @Pointcut, @Before, @After, @AfterReturning, @Around
- Using XML
 - aop:config, aop:aspect, aop:before, etc.

What is the underlying mechanism?

→ Using dynamic proxy to delegate/process advices

INSTRUMENTATION AND MONITORING AT SCALE

Full stack monitoring

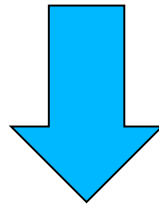


- Combine many techniques: instrumentation, API interface, etc.
- Push and pull methods
- Exact measurement and sampling

Scale of systems and of monitoring

- Many monitoring components
- 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

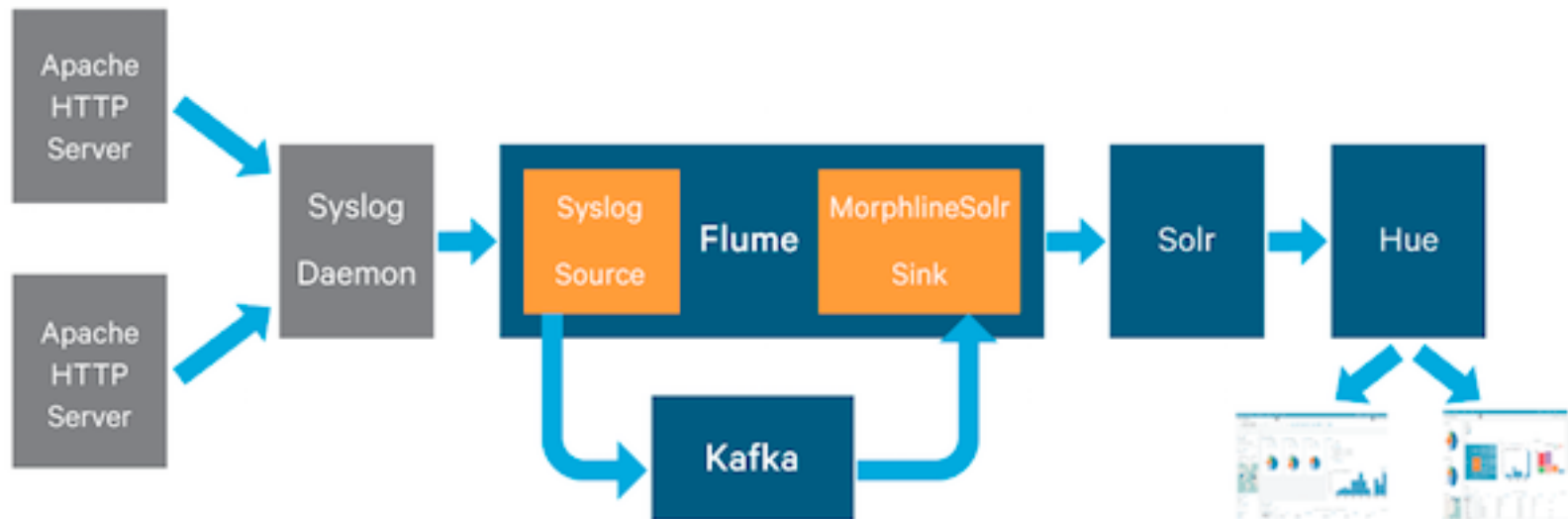
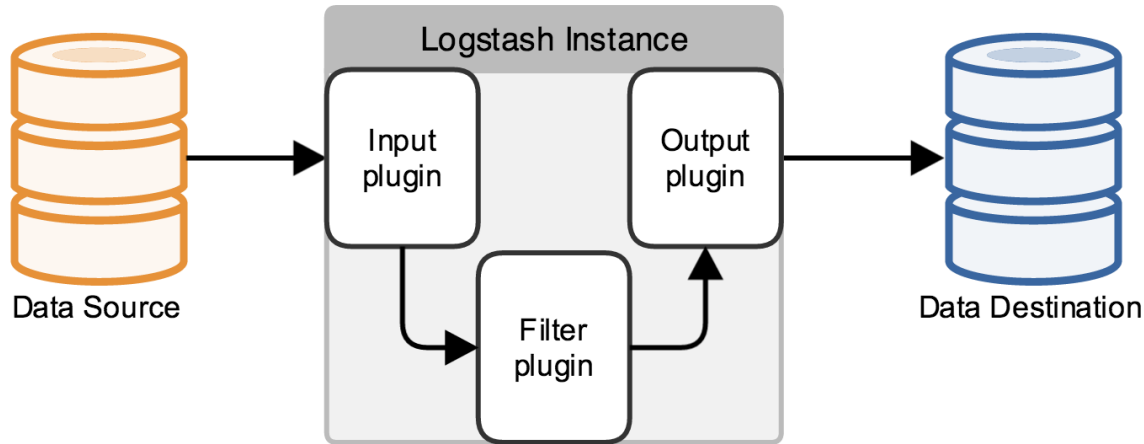


Figure source: <https://blog.cloudera.com/blog/2015/02/how-to-do-real-time-log-analytics-with-apache-kafka-cloudera-search-and-hue/>

Remember Logstash?




- Codecs: stream filters within inputs or outputs that change data representation
- E.g.: multiline
→ 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.




Filebeat
Log Files




Metricbeat
Metrics



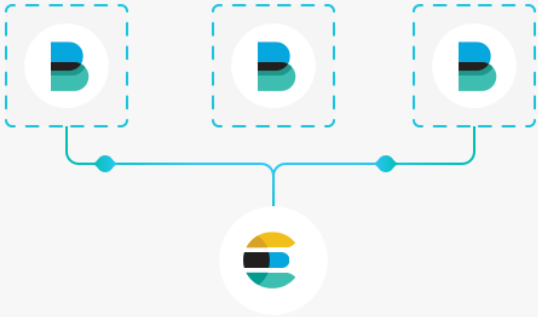
Packetbeat
Network Data



Winlogbeat
Windows Event Logs



Heartbeat
Uptime Monitoring



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.

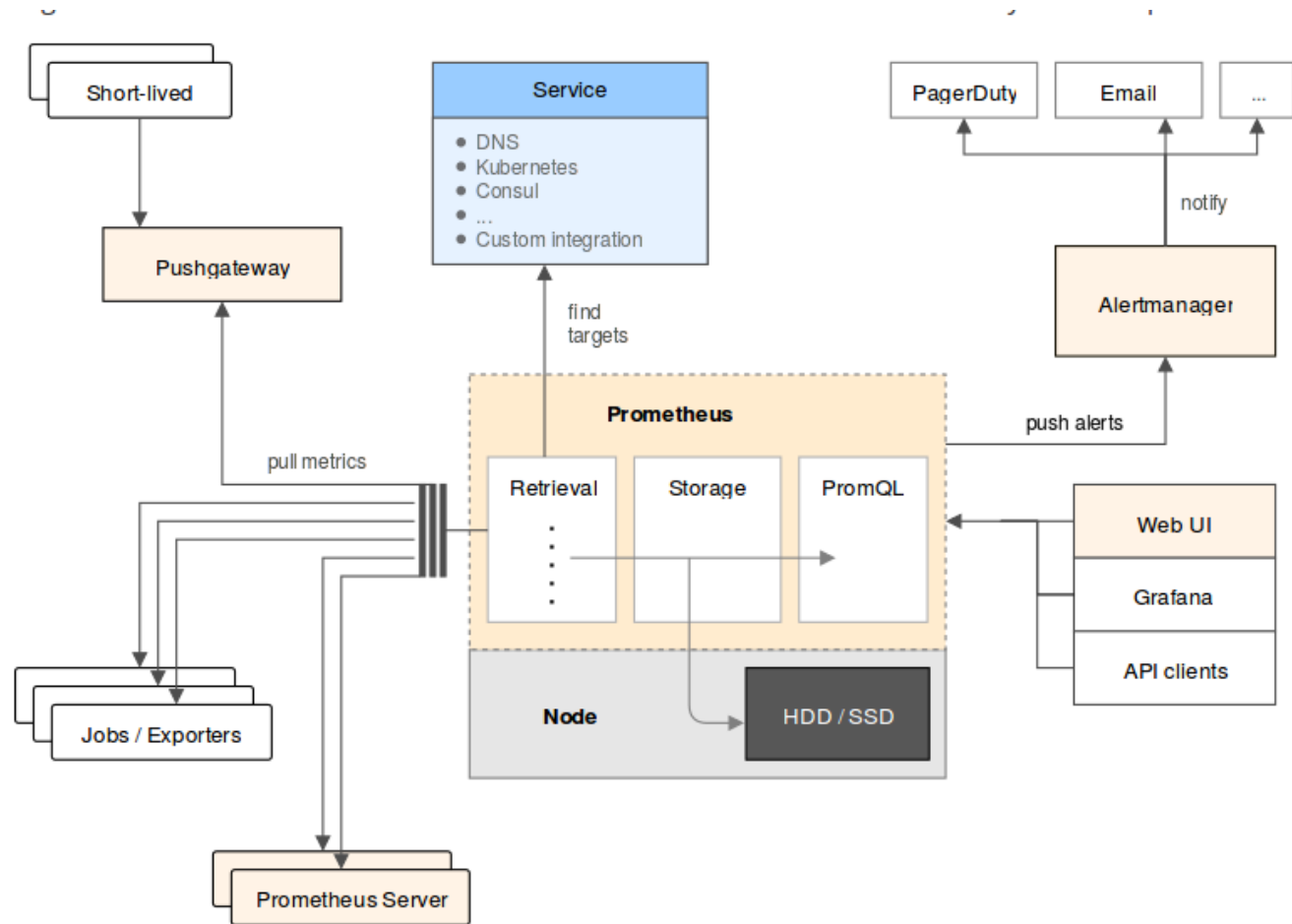
<https://www.elastic.co/products/beats>

Example: Prometheus



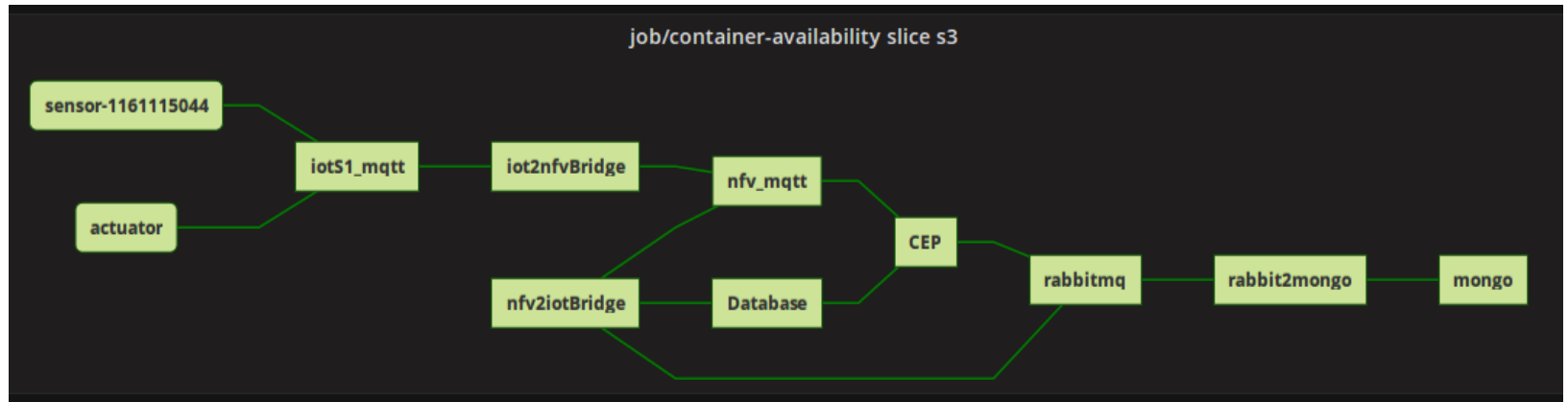
The screenshot shows the Prometheus website homepage. The header is dark with the Prometheus logo and name on the left, and navigation links for DOCS, DOWNLOAD, COMMUNITY, and BLOG on the right. The main content area has an orange background with a geometric pattern. The headline reads "From metrics to insight" and the sub-headline says "Power your metrics and alerting with a leading open-source monitoring solution." Below this are two buttons: "GET STARTED" and "DOWNLOAD". At the bottom of the page, there is a light gray banner with the text "CloudNativeCon 2017 videos are out! – [Watch the Prometheus track here](#)".

Prometheus Architecture



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

An example



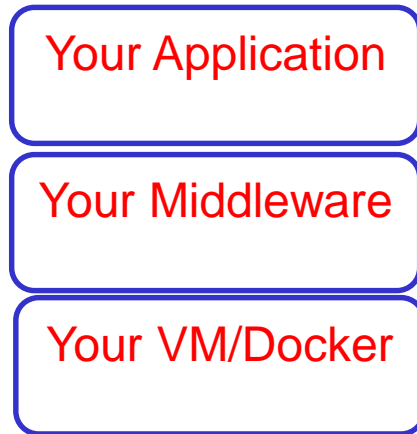
From Christian Proinger (Master thesis work TU Wien), using Prometheus and Grafana, etc. for Raspberry PI dockers + clouds.



Build instrumentation and monitoring for your cloud application/services

Do a real-world test!

Instrumentation and monitoring (aspects, annotation, etc.)



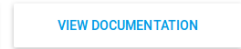
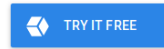
Logs/traces/metrics



Cloud monitoring services

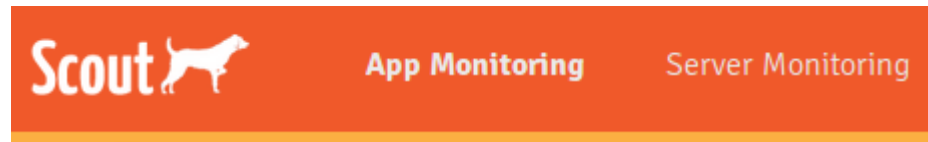
STACKDRIVER MONITORING

For applications running on Google Cloud Platform and Amazon Web Services



Amazon CloudWatch

Amazon CloudWatch is a monitoring service for AWS cloud resources



Key takeaway (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)

Summary

- Many features are generic but very useful for 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
- Understanding which instrumentation techniques should be used and what will be instrumented is crucial
- Often we need to combine different techniques

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>

Thanks for your attention

Hong-Linh Truong
Distributed Systems Group, TU Wien
truong@dsg.tuwien.ac.at
<http://dsg.tuwien.ac.at/staff/truong>