Intermediate Java Programming

Declarative programming using annotations

Copyright © 2003-2020 IPT – Intellectual Products & Technologies Ltd. Licensed under CC BY 3.0 license.

Where to Find The Code and Materials?

https://github.com/iproduct/course-java-web-development

Why Annotations?

- Java APIs often require boilerplate code for bootstrapping & maintenance.
- For example, in order to write SOAP-based web service with JAX-WS, you must provide paired interface and implementation. If we decorate java class methods using annotations, we can indicate which methods are meant to be exposed as web methods for remote access. This boilerplate code could be generated automatically by a tool based on annotations.
- REST services with JAX-RS can use annotations for bootstrapping and routing.
- Other APIs require configuration files to be maintained in parallel with code – e.g. JavaBeans require BeanInfo classes, Servlet-based web applications require web.xml deployment descriptor. It would less error-prone and more convenient to maintain this information as annotations in the program code, or combination of both approaches could be employed.

History of Declarative Programming in Java

- The Java platform always had various ad hoc annotation mechanisms:
  - **transient** modifier is an ad hoc annotation indicating that a field should be ignored during the serialization;
  - **@Deprecated** javadoc tag is an ad hoc annotation suggesting that the method should no longer be used.
- Java 5 introduced a general purpose mechanism called metadata annotations, permitting to extend these declarative programming mechanisms by defining custom annotations and applying them in code.
- Since then a numerous java frameworks and libraries have been redesigned to take advantage of the flexibility and ease of maintenance that annotations in the code offer – e.g. JavaEE Servlet API, JAX-RS, JAXB, EJB, CDI, JPA, Bean Validation, Hibernate, Spring, and many more.

Java Annotations

- The annotations mechanism consists of:
  - syntax for declaring annotation types;
  - syntax for annotating declarations and (since java 1.8) usages of a type;
  - APIs for reading annotations,
  - class file representation for annotations accessible using reflection in runtime
  - an annotation processing tool (APT) that was integrated in javac since v1.6.
- Annotations do not directly affect program semantics, but they do affect the way programs are treated by tools and frameworks, which can generate new code and in turn affect the semantics of the program to be executed by JVM. Annotations can be read from source files, class files, or reflectively at run time.

Declarative Programming Using Annotations

- Java annotation is a form of syntactic metadata that can be added to the source code.
- Classes, methods, variables, parameters and Java packages may be annotated. Since Java 1.8 annotations can also be applied to any type use - new, casts, implements and throws clauses, etc.
- Like Javadoc tags, Java annotations are read from source files. They can complement javadoc tags. If the markup is intended to produce documentation, it should be javadoc tag; otherwise, annotation.
- Unlike Javadoc tags, annotations can also be embedded in and read from Java class files generated by the Java compiler. This allows annotations to be retained by the JVM at run-time and read using reflection.
- Some annotations are “meta”-annotations allowing to create new ones.

What the Java Annotations Can Be Used For?

- Annotations, as a form of metadata, provide data about a program that is not part of the program itself. Annotations do not directly effect the operation of the program code they annotate.
- Annotations have a number of usages – they provide:
  - Information for the compiler — Annotations can be used by the compiler to detect errors or suppress warnings.
  - Compile-time and deployment-time processing - tools and frameworks can process annotation information to generate code, XML files, and other types of files.
  - Runtime processing — some annotations are available to be examined at runtime using reflection mechanisms discussed in previous lessons.

Annotation Types Used by the Java Language

- **@Deprecated** – indicates that the marked element is deprecated and should no longer be used.
- **@Override** – informs the compiler that the element is meant to override an element declared in a superclass.
- **@SuppressWarnings({"unchecked", "deprecation"})** – tells the compiler to suppress specific warnings that it would otherwise generate.
- **@SafeVarargs** – when applied to a method or constructor, asserts that the code does not perform potentially unsafe operations on its varargs parameter.
- **@FunctionalInterface** – since Java 1.8, indicates that the type declaration is intended to be a functional interface, as defined by JLS

Example: Using Annotation Types in Java Code

-
  ```
  @Override
  @SuppressWarnings("unchecked")
  public <T> T getBean(Class<T> cls) {
       List<BeanDescriptor> beanDescriptors =
                  beansByType.get(cls.getCanonicalName());
       if (beanDescriptors.size() != 1) {
            throw new BeanLookupException("There should be
                  exactly one bean of type '" + cls + "', but " +
                  beanDescriptors.size() + " found.");
       }
       return (T) getProxyInstance(cls, beanDescriptors.get(0));
  }
  ```

Annotations That Apply to Other Annotations (1)

- Annotations that apply to other annotations are called meta-annotations. There are several meta-annotation types defined in java.lang.annotation:

- **@Retention** – specifies how the marked annotation is stored:
  - RetentionPolicy.SOURCE – The marked annotation is retained only in the source level and is ignored by the compiler.
  - RetentionPolicy.CLASS – The marked annotation is retained by the compiler at compile time, but is ignored by the Java Virtual Machine (JVM).
  - RetentionPolicy.RUNTIME – The marked annotation is retained by the JVM so it can be used by the runtime environment.

Annotations That Apply to Other Annotations (2)

- **@Documented** – indicates that whenever the specified annotation is used those elements should be documented using the Javadoc tool.
- **@Target** – marks another annotation to restrict what kind of Java elements the annotation can be applied to:
  - ElementType.ANNOTATION_TYPE
  - ElementType.CONSTRUCTOR
  - ElementType.FIELD – field or property
  - ElementType.LOCAL_VARIABLE
  - ElementType.METHOD
  - ElementType.PACKAGE
  - ElementType.PARAMETER – method parameters
  - ElementType.TYPE – can be applied to any element of a class.

Annotations That Apply to Other Annotations (2)

- **@Inherited** – indicates that the annotation type can be inherited from the super class. (This is not true by default.) When the user queries the annotation type and the class has no annotation for this type, the class' superclass is queried for the annotation type. This annotation applies to class declarations only.
- @Repeatable – since Java 1.8 indicates that the marked annotation can be applied more than once to the same declaration or type use. E.g.:

```
@Schedules({@Schedule(dayOfMonth="last"),
            @Schedule(dayOfWeek="Fri", hour=23)})
public void doPeriodicCleanup() {
    //...
}
```

Example: Defining Repeatable Annotations

```java
@Retention(RUNTIME)
@Target(METHOD)
@Repeatable(Schedules.class)
public @interface Schedule {
   String dayOfMonth() default "first";
   String dayOfWeek() default "Mon";
   int hour() default 12;
}
```

```java
@Retention(RUNTIME)
@Target(METHOD)
public @interface Schedules {
     Schedule[] value();
}
```

Defining Class, Method, Parameter and Field annot.
(JSR 330: Dependency Injection for Java)

-
  ```
  @Target({ METHOD, CONSTRUCTOR, FIELD })
  @Retention(RUNTIME)
  @Documented
  public @interface Inject {}
  ```

-
  ```
  @Target(ANNOTATION_TYPE)
  @Retention(RUNTIME)
  @Documented
  public @interface Qualifier {}
  ```

-
  ```
  @Qualifier
  @Retention(RetentionPolicy.RUNTIME)
  @Target({FIELD, METHOD, TYPE, PARAMETER})
  public @interface JaxbRepository {}
  ```

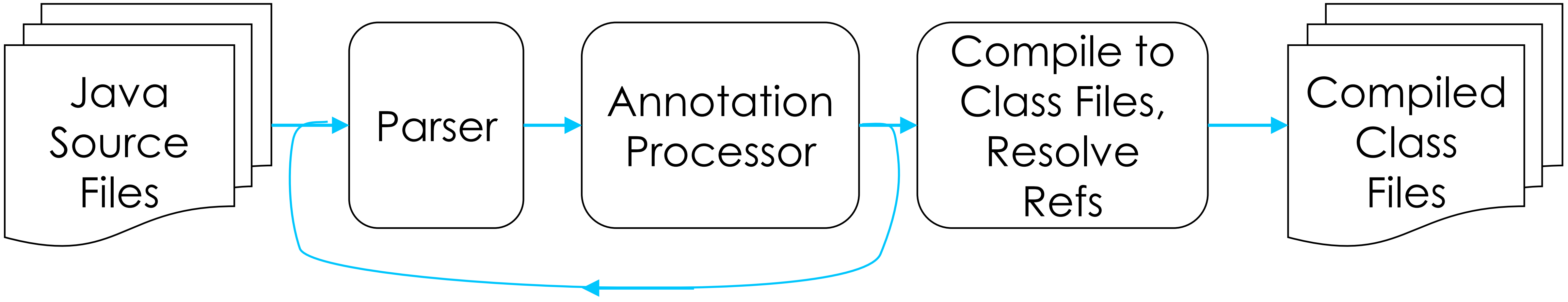
Using @Inject with Custom Qualifier Annotation
(JSR 330: Dependency Injection for Java)

-
  ```
  @Component
  public class UserControllerImpl implements UserController {
      @Inject @JaxbRepository
      private UserRepository repo;

      public UserRepository getRepo() {
          return repo;
      }

      public void setRepo(UserRepository repo) {
          this.repo = repo;
      }
      //...
  }
  ```

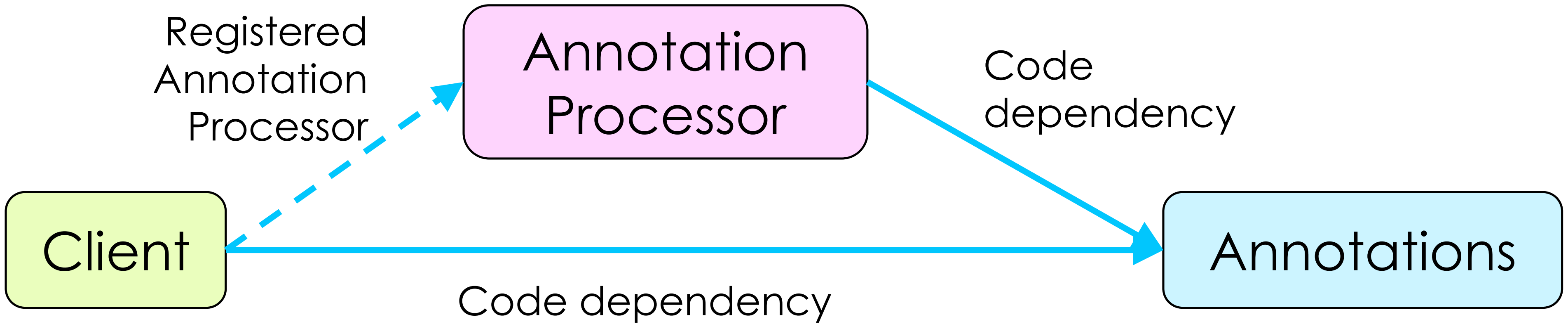
Compile-time Annotation Processing (1)

- Tools and frameworks can process annotation information to generate additional source files (code), XML files, metadata, documentation, resources, and other types of files, based on annotations in the source code.
- Tutorial: https://www.baeldung.com/java-annotation-processing-builder
- It can only be used to generate new files, not to change existing ones.
- Lombok is a noteworthy exception – it uses annotation processing as a bootstrapping mechanism to include itself into the compilation process and modify the AST via some internal compiler APIs (not the intended purpose of annotations processing).
- Actively used by many Java libraries, to generate metaclasses in QueryDSL and JPA, to augment classes with boilerplate code in Lombok, etc.

Compile-time Annotation Processing (2)



Process annotations in generated new files



Compile-time Annotation Processing (3)

- Package javax.annotation.processing - Facilities for declaring annotation processors and for allowing annotation processors to communicate with an annotation processing tool environment:
- Completion - A suggested completion for an annotation.
- Filer - This interface supports the creation of new files by an annotation processor.
- Messager - A Messager provides the way for an annotation processor to report error messages, warnings, and other notices.

Compile-time Annotation Processing (4)

- ProcessingEnvironment - An annotation processing tool framework will provide an annotation processor with an object implementing this interface so the processor can use facilities provided by the framework to write new files, report error messages, and find other utilities.
- Processor- The interface for an annotation processor.
- RoundEnvironment - An annotation processing tool framework will provide an annotation processor with an object implementing this interface so that the processor can query for information about a round of annotation processing.

Defining Custom Annotations

-
  ```java
  @Retention(RUNTIME)
  @Target(TYPE)
  @Inherited
  public @interface Repository {
        String value() default "";
  }
  ```

-
  ```java
  @Retention(RUNTIME)
  @Target(TYPE)
  @Inherited
  public @interface Scope {
        BeanScope value() default BeanScope.SINGLETON;
  }
  ```

- `public enum BeanScope { SINGLETON, PROTOTYPE }`

Processing Annotations at Runtime (1)

- An annotation A is **directly present** on an element E if E has a RuntimeVisibleAnnotations or RuntimeVisibleParameterAnnotations or RuntimeVisibleTypeAnnotations attribute, and the attribute contains A.
- An annotation A is **indirectly present** on an element E if E has a RuntimeVisibleAnnotations or RuntimeVisibleParameterAnnotations or RuntimeVisibleTypeAnnotations attribute, and A 's type is repeatable, and the attribute contains exactly one annotation whose value element contains A and whose type is the containing annotation type of A 's type.

Processing Annotations at Runtime (2)

- An annotation A is **present** on an element E if either:
  - A is directly present on E; or
  - No annotation of A 's type is directly present on E, and E is a class, and A 's type is inheritable, and A is present on the superclass of E.
- An annotation A is **associated** with an element **E** if either:
  - A is directly or indirectly present on E; or
  - No annotation of A 's type is directly or indirectly present on E, and E is a class, and A's type is inheritable, and A is associated with the superclass of E.

Processing Annotations at Runtime (3)

Overview of kind of presence detected by different AnnotatedElement methods

Method		Kind of Presence			
		Directly Present	Indirectly Present	Present	Associated
T	getAnnotation(Class<T>)			X	
Annotation[]	getAnnotations()			X	
T[]	getAnnotationsByType(Class<T>)				X
T	getDeclaredAnnotation(Class<T>)	X			
Annotation[]	getDeclaredAnnotations()	X			
T[]	getDeclaredAnnotationsByType(Class<T>)	X	X		

Inversion of Control (IoC) & Dependency Injection (DI) (Recap.)

- Components provide necessary metadata (specification) about their supported interfaces (contracts) which allows their dynamic discovery and assembly at runtime.
- There is no dependency on concrete component implementations (only on supported interfaces), which allows to be developed in parallel with their clients, and to be swapped with improved implementations without changing the existing code of client components.
- This is usually done using the principle of Inversion of Control (IoC) implemented either as Dependency Injection (DI - e.g. CDI) or a programmatic lookup (e.g. JNDI).

Exercise: Building Custom DI Framework

- **Goal:** To build an initial prototype of custom Dependency Injection (DI) Framework using JSR 330: Dependency injection for Java, and custom annotations:
  - for bean (component) stereotypes: @Component, @Repository, @Service, following the principles of Domain Driven Design (DDD)
  - for bean scope declaration – initially only two bean scopes will be supported: SINGLETON and PROTOTYPE
- The framework should be implemented using Java Dynamic Proxy implementation, and using reflection mechanisms at runtime to discover annotations and inject dependencies.

Domain Driven Design (DDD)

We need tools to cope with the complexity inherent in most real-world design problems.

Simple solutions are needed – cope with problems through divide and concur on different levels of abstraction:

Domain Driven Design (DDD) – back to basics: domain objects, data and logic.

Described by Eric Evans in his book:
Domain Driven Design: Tackling Complexity in the Heart of Software, 2004

Domain Driven Design (DDD)

Main concepts:

- Entities, value objects and modules
- Aggregates and Aggregate Roots [Haywood]:
  value < entity < aggregate < module < BC
- Repositories, Factories and Services:
  application services <-> **domain services**
- Separating interface from implementation

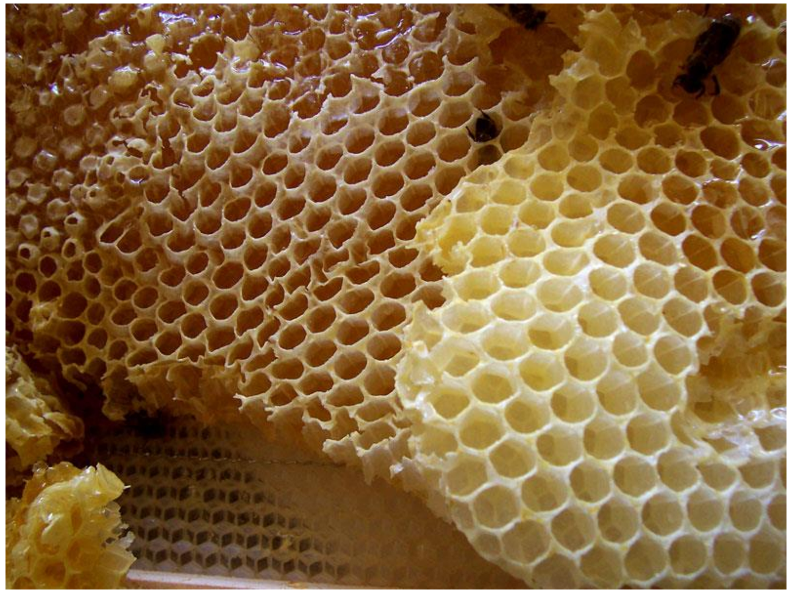
Domain Driven Design (DDD) & Microservices

- Ubiquitous language and Bounded Contexts
- DDD Application Layers:
- Infrastructure, Domain, Application, Presentation
- Hexagonal architecture :

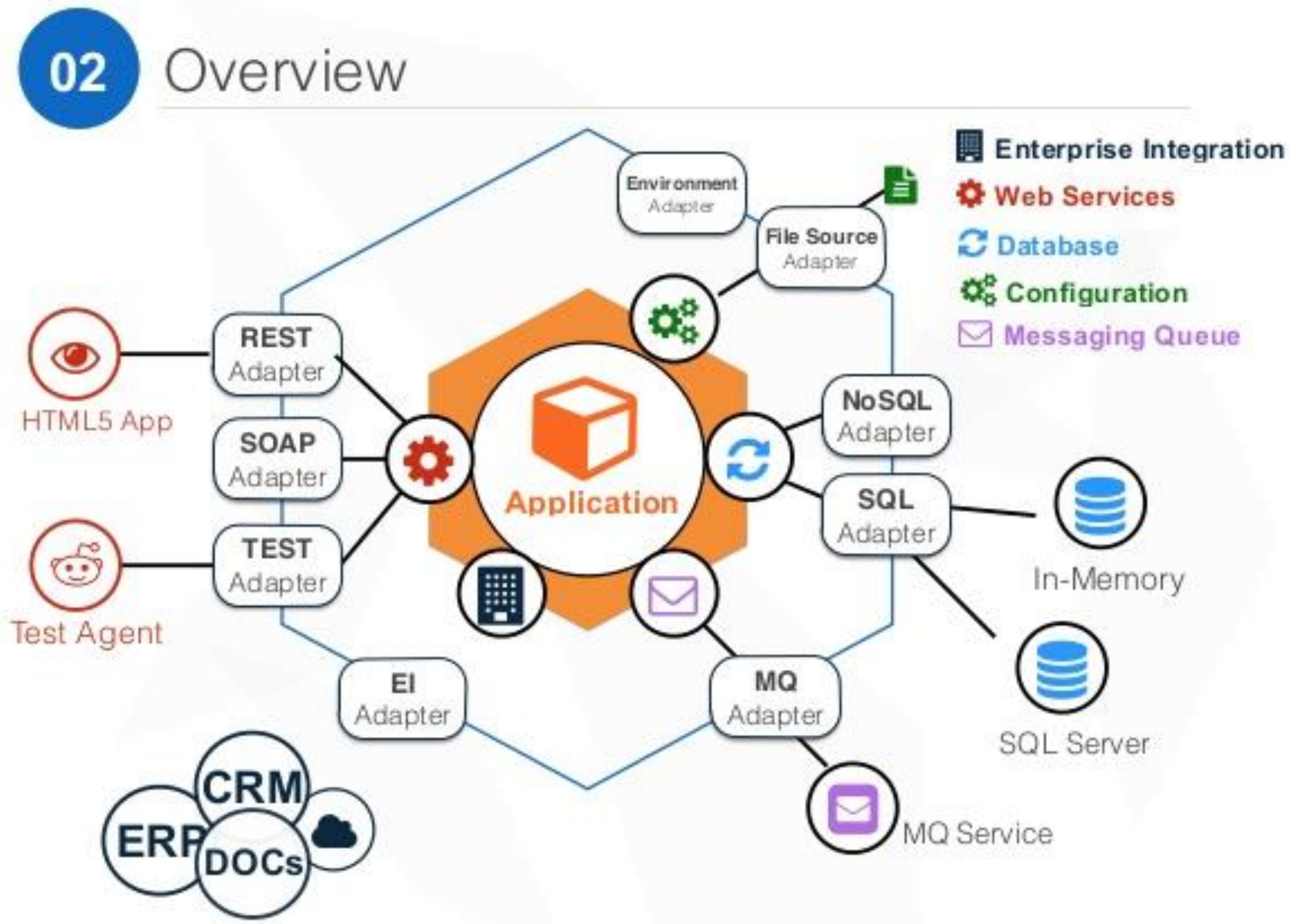
  OUTSIDE <-> transformer <->

  (application <-> domain)

  [A. Cockburn]



Hexagonal Architecture



Source: https://www.slideshare.net/fabricioepa/hexagonal-architecture-for-java-applications

Fully featured examples: CDI & Bean Validation APIs

- **Contexts and Dependency Injection for Java (CDI)** – building custom bean and event qualifiers,
- **Bean Validation** – building custom validation annotations.

Thank's for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

http://robolearn.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD