Department of Computer Science & Applied Physics

The Java Module System – Stronger Encapsulation

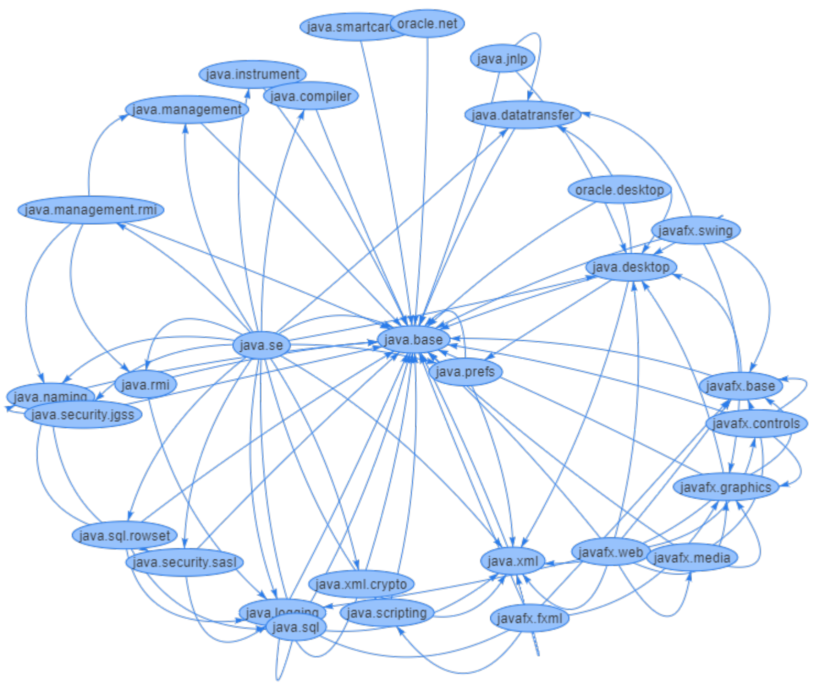
As a software application grows in size and complexity, some high-level mechanism for structuring and packaging resources becomes necessary. A good design decomposes and distributed its state and behaviour to a suite of highly cohesive and loosely coupled components. The granularity of these components can be fine, such as methods and classes, or coarse, such as packages and libraries. Regardless of the granularity of a component, the basic pillars and design principles of the object-oriented paradigm can still be applied effectively to promote usability, flexibility and maintainability.

At a high level, the combination of abstraction and encapsulation can be viewed as a form of *indirection*, where a layer of abstraction is used to hide the internal details of a component. This is not just an OOP concept, as it is has been described by Andrew Koenig, a C++ guru at Bell Labs, as the ***fundamental theorem of software engineering***: *any problem can be solved by introducing an extra level of indirection*. The "theorem" is sometimes qualified with the phrase "*except for the problem of too many levels of indirection*" to indicate that too many layers of abstraction conflate a design by creating intrinsic and unnecessary complexity. Indeed, the over-zealous application of abstraction and encapsulation is regarded as an anti-pattern and has been accorded the acronym YAFL.

An additional level of indirection was added to the Java language in 2017 with the integration of ***Project Jigsaw*** into the Java 9 release. ***Project Jigsaw*** is a mechanism for organising packages and applications into ***modules***. The main motivation for this addition to the language was to manage increasing complexity and scalability by promoting stronger encapsulation of packages and their dependencies. A Java application or library now exposes its behaviour through modules instead of packages, i.e. can be decomposed as *module→package→type→feature→statement*. At the root of an application, a **module descriptor**, called *module-info.java*, must **explicitly** declare the contained packages that are exported and accessible. This superimposes another layer of encapsulation on the existing set of visibility modifiers (public, private, protected). Internal packages that are not exported are not visible from outside the module, even if their features are public! The new modular system enables the detection of missing modules at start-up, allowing the JVM to shutdown gracefully. In contrast, the existing system dynamically loads classes on demand and will crash, with a *ClassNotFoundException*, if the class-loader fails to find a resource at run-time.

In addition to strengthening encapsulation, the new module system upholds the basic principles of package cohesion and coupling. A module exemplifies the ***Reuse-Release Equivalence Principle***, as a module replaces a package as the granularity of release and therefore re-use. By aggregating together cohesive packages into modules, the system also promotes the ***Common Reuse Principle (CRP)***, as a module packages its dependencies together. Even **transitive dependencies** are accommodated in the module system. The indirection of the module system allows even greater control of the visibility of types and features, coupling high cohesion with strong encapsulation. This is a manifestation of the ***Common Closure Principle (CCP)***, as the module system allows whole packages to be closed against the same kinds of change.

An important feature of the module system is that ***circular dependencies are not allowed*** between modules and will be detected and reported by the compiler. The modules used by a Java application form a directed acyclic graph, thus satisfying the ***Acyclic Dependencies Principle***. Loose-coupling between modules is further promoted by the declaration of service interface consumers and providers. Assuming that the existing packages that have been aggregated into modules are designed correctly and communicate with each other through polymorphic interactions, the layer of abstraction provided by the module system will decouple system components even further and allow for greater flexibility.



<u>Exercises</u>

- Open a version of Eclipse that supports Java modules and create a new Java Project called **CryptoLab**. When prompted for the name for the default module, provide **ie.atu.sw**.

- Download the Zip archive *Source Code for Composition and Reuse Lab (Finished)* from Moodle and extract the contents of the archive into the ***src*** directory of the project **Crypto-Lab**.

- Open the file module-info.java. It should be declared as follows:

```
open module atu.software {
}
```

- Create the following new packages in the application:

  ie.atu.sw.crypto
  ie.atu.sw.crypto.symmetric
  ie.atu.sw.crypto.asymmetric

- Using the Eclipse Explorer, drag and drop the existing set of Java classes into the new package structure as shown below. Eclipse will automatically update all the package names and import statements:

Package Name	Contents
ie.atu.sw	Runner.java
ie.atu.sw.crypto	Algorithm.java Cypherable.java AbstractCypher.java CypherFactory.java

ie.atu.sw.crypto.symmetric	AESCypher.java DESCypher.java VigenereCypher.java
ie.atu.sw.crypto.asymmetric	RSACypher.java

- Open a command prompt and navigate to the ***bin*** directory of the Eclipse project. Execute the following command: *java ie.atu.sw.Runner*. The application should function properly using the traditional mechanism for launching a JVM.

- Each class should be packaged in a **module** and made available through a module. Therefore, it is better to execute the programme as follows:

```
java --module-path ./ --module atu.software/ie.atu.sw.Runner
```

  Note the *module-path* and *module* switches. The *module-path* effectively replaces the traditional *–classpath* / *-cp* switch. The main class is referred to using the *moduleName/fullClassName* notation.

- Modules are packaged and deployed in JAR archives, but **only one module can be packaged in a single JAR**. Create a JAR archive from inside the ***bin*** directory as follows: jar -cf crypto.jar *

  Verify that the contents are packaged correctly as follows: jar -tf crypto.jar

  Note that building the JAR like this upholds the *Reuse-Release Equivalence Principle* - the granule of re-use is the full *atu.software* module and all its packages. This is also the granule of release – a single module per JAR.

Create a Dependency

- Create a new Eclipse Project called *CryptoConsumer* with the module name **atu.consumer**. Right click over the project name and select *Properties→Java Build Path→Libraries*. You should see two options: *Modulepath* and *Classpath*. Select *Modulepath* and click the *"Add External JAR"* button. Navigate to the directory containing the JAR file created above and select **crypto.jar**. The library has now been added to your project's *Modulepath*.

- Create a new package called **ie.atu.consumer.crypto** and add to it a new class called *ConsumerRunner* as shown below. As an alternative, you can copy and paste the contents of the class Runner in the **Crypto-Lab** project:

```
package ie.atu.consumer.crypto;

import ie.atu.sw.crypto.*;
public class ConsumerRunner {
  public static void main(String[] args) throws Throwable{
    CypherFactory cf = CypherFactory.getInstance();
    Cypherable cypher = cf.getCypherable(Algorithm.AES);

    byte[] s = new String("HAPPY DAYS").getBytes("UTF-8");
    byte[] t = cypher.encrypt(s);

    System.out.println(new String(t));
    System.out.println(new String(cypher.decrypt(t)));
  }
}
```

- Expand the ***Referenced Libraries*** folder in the ***Eclipse Project Explorer*** and check that ***crypto.jar*** is listed and available. You should have the following error message in the import statement of *ConsumerRunner*: *The import ie.atu.sw cannot be resolved*. Even though the required classes are in the referenced JAR library, we do not have access to it because the module layer of indirection masks the visibility of the public classes and features.

- Open the class *module-info.java* in the root of the ***CryptoConsumer*** project and add the following **requires** declaration:

```
module atu.consumer {
  requires atu.software;
}
```

- Move your mouse over the import statement in the class ***ConsumerRunner*** again. You should now see the following error: *The package ie.atu.sw.crypto is not accessible*. This means that the JVM can see the module but its contents are **fully encapsulated**, even the public features of the contained classes.

- Open the *module-info.java* file of the **Crypto-Lab** project and add the following **exports** declaration to it:

```
module atu.software {
    exports ie.atu.sw.crypto;
}
```

- **Rebuild the JAR** from a command prompt using the jar -cf crypto.jar * command. The errors messages in the class ***ConsumerRunner*** should now be gone. Execute the *main()* method of ***ConsumerRunner*** to test that the imported module works okay.

- In the class ***ConsumerRunner***, change the statement Cypherable cypher = cf.getCypherable(Algorithm.AES) to the following and add the import statement import ie.atu.sw.crypto.symmetric.AESCypher to the top of the class:

```
Cypherable cypher = new AESCypher();
```

  You should see the error "The type ie.atu.sw.crypto.symmetric.AESCypher is not accessible". Even though the class ***AESCypher*** contains a public constructor and public methods, the visibility of these features is **encapsulated by the indirection of the module** system.

- Open the *module-info.java* file of the crypto project and add a second exports declaration to it:

```
module atu.software {
  exports ie.atu.sw.crypto;
  exports ie.atu.sw.crypto.symmetric;
}
```

- All of the non-private resources in the *ie.atu.sw.crypto.**symmetric*** package are now available, but not those from the *ie.atu.sw.crypto.**asymmetric*** package. Verify this by attempting to reference the ***RSACypher*** class.

- Open a command prompt and navigate to the **bin** directory of the ***CryptoConsumer*** project. Execute the command jar -cf **myapp**.jar * to build a JAR library for the consumer application.

Custom JRE Configurations

- Among the benefits of a modular system design is the potential of reducing the size of binary distributions and runtime environments. The **jlink** tool is used to link a set of modules and their ***transitive dependencies*** to create a custom modular run-time deployment. Execute the **jlink** command from the directory containing **myapp**.jar:

```
jlink --module-path ./myapp.jar:../../CryptoLab/bin/crypto.jar --output ./out/ --add-modules atu.consumer
```

You will need to ensure that the module-path parameter is configured with the correct system path on your computer. This builds a JRE specific for the application and links together all the module dependencies, including the *ie.atu.sw.crypto* module, into a compact structure. The size of the JRE is circa 38MB compared to the 162.4 MB JRE for Java 1.8.0_05 on OSX, i.e. >75% less and. The **jlink** tool is designed to work with the Java module system to create a custom JRE with a small memory footprint for use on "smart" IoT devices, i.e. what the Java language was originally designed to do back in 1993, when it was called **Oak**…

- Change to the "**out**" directory and execute the following command to run the application. Note that we are using the **module-path** switch instead of **-classpath**:

  ```
  bin/java --module-path ./ --module
  atu.consumer/ie.atu.consumer.crypto.ConsumerRunner
  ```

- Execute the following command to see the set of modules and their version strings available to your default JRE: `java --list-modules`. You should see ~100 different module names. Note that these are aggregated into the following families:

- Execute the following command to see the set of modules and their version strings available to the JRE created with **jlink**: `bin/java --list-modules`. You should only see the following modules listed:

  - atu.consumer
  - atu.software
  - java.base@10.0.2 [The version string will vary depending on your SDK]

Transitive Dependencies

- Create a new project called **Vigenere** with a default module name of *atu.classic*. This module represents the suite of classical ciphers from antiquity until the advent of electro-mechanical cryptography after WW1.

- Add a default package of *ie.atu.classic.vigenere* to the project and then create a new class called **Vigenere**. With the exception of the *encrypt(byte[] plainText)* and *decrypt(byte[] cypherText)* methods, copy and paste all the source code from the class *VigenereCypher* into the new class ***Vigenere***. *Change the visibility of the get/setKey() methods to public.*

- Edit the file *module-info.java* and use the **exports...to** declaration to completely encapsulate the module to all modules *except atu.software* as follows:

  ```
  module atu.classic {
    exports ie.atu.classic.vigenere to atu.software;
  }
  ```

  Try adding the declaration **requires atu.software;** to the module descriptor. You should see the following error: "*Cycle exists in module dependencies, Module atu.classic requires itself via atu.software*". The module dependency structure must be a directed acyclic graph (DAG).

- Using a command prompt, build a JAR archive of the *atu.classic* module from the **bin** directory of the *Vigenere* project as follows: jar -cf vigenere.jar *.

- In the **CryptoLab** project, right click over the project name and select *Properties→Java Build Path→Libraries* and add the JAR archive ***vigenere.jar*** to the *Modulepath* variable. You should then change the class VigenereCypher to the following (note that we are using composition to re-use the class Vigenere from the external module):

```
package ie.atu.sw.crypto.symmetric;

import ie.atu.sw.crypto.*;
import ie.atu.classic.vigenere.*;

public class VigenereCypher extends AbstractCypher{
  private Vigenere v = new Vigenere(); //Compose

  public void setKey(String key){
    v.setKey(key); //Delegate
  }

  public byte[] encrypt(byte[] plainText) throws Throwable {
    return v.doCypher(plainText, true); //Delegate
  }

  public byte[] decrypt(byte[] cypherText) throws Throwable {
    return v.doCypher(cypherText, false); //Delegate
  }
}
```

- Rebuild the JAR archive **crypto.jar** and execute *ConsumerRunner* from Eclipse. You will be presented with the following error: "*Error occurred during initialization of boot layer java.lang.module.FindException: Module atu.classic not found, required by atu.software*". Add the transitive dependency to *module-info.java* as follows:

```
module atu.software {
  exports ie.atu.sw.crypto;
  exports ie.atu.sw.crypto.symmetric;

  requires transitive atu.classic;
}
```

- Delete the "**out**" directory and rebuild the JAR archive **crypto.jar** and then rebuild the full JRE using the existing configuration of **jlink** (use up arrow on the command prompt). You should get the following error again: *Error: Module atu.classic not found, required by atu.software*, i.e. **jlink** needs to know the location of the implementation of the module *atu.classic*.

- Add **vigenere.jar** to the *module-path* switch as follows and then rebuild the JRE:

```
jlink --module-path
./myapp.jar:../../ModuleLab/bin/crypto.jar:../../Vigenere/bin/vigenere.jar --output
./out/ --add-modules atu.consumer --add-modules atu.classic
```

Note that you may need to change the path delimiters on Windows to a semi-colon (;) instead of a colon (:).

- Execute the following command to see the set of modules and their version strings available to the JRE created with jlink: ***bin/java --list-modules***. You should now see the module *atu.classic*. Ensure that the distribution works correctly by executing the application as follows:

```
bin/java --module-path ./ --module
atu.consumer/ie.atu.consumer.crypto.ConsumerRunner
```

Command Line Options

The Java Platform Module System (JPMS) also introduces several command line options for the ***javac*** and ***java*** tools to work around compatibility issues. The most important of these are:

Switch	Description
--add-exports	Exports a package, making its public types and members accessible.
--add-opens	Opens a package, making all of its members accessible to *java*.
--add-modules	Adds the listed modules and their transitive dependencies to the module graph
--add-reads	Enables one module to read another
--patch-module	Adds the specified class to a specific module

For example, prior to Java 9, the JD-GUI decompiler was executed using the command **java -jar jd-gui-1.4.0.jar.** However, JD-GUI, which relies on reflection to dynamically examine compiled bytecodes and generate inferred source code, has not been modularised. Running the programme as shown above generates the following error:

```
WARNING: An illegal reflective access operation has occurred
Exception in thread "main" java.lang.reflect.InaccessibleObjectException: Unable to make
jdk.internal.loader.ClassLoaders$AppClassLoader(jdk.internal.loader.ClassLoaders$PlatformClassLoade
r,jdk.internal.loader.URLClassPath) accessible: module java.base does not "opens jdk.internal.loader" to
unnamed module @3cef309d
```

The JVM will not let the ***unnamed module*** (it's referred to by a *version string* instead) read a dependent package. The workaround for this is to use the --add-opens switch as follows:

```
java --add-opens java.base/jdk.internal.loader=ALL-UNNAMED jd-gui-1.4.0.jar
```

This allows the unnamed module to read the resources in the package *jdk.internal.loader* in the module *java.base*. However, executing this will still generate the error "*module jdk.zipfs does not "opens jdk.nio.zipfs" to unnamed module @1efed156*" because the transitive dependency has not been resolved. The solution is to open the jdk.nio.zipfs package to the ***unnamed module*** as follows:

```
java --add-opens java.base/jdk.internal.loader=ALL-UNNAMED --add-opens
jdk.zipfs/jdk.nio.zipfs=ALL-UNNAMED -jar jd-gui-1.4.0.jar
```

These workarounds are ugly and are intended to cajole developers into refactoring legacy code into modules, while allowing a cumbersome mechanism for forward compatibility.