|  |  |  |
| --- | --- | --- |
|  | UNIVERSITY OF CRAIOVA  FACULTY OF AUTOMATION, COMPUTERS AND ELECTRONICS  COMPUTERS AND INFORMATION TECHNOLOGY DEPARTMENT |  |

DEGREE PROJECT

**Stoenica Robert-Bogdan**

|  |
| --- |
| SCIENTIFIC COORDINATOR  **S.l. Dr. Ing. Stefan Udristoiu.**  Department of Computer Science and Information Technoligies,  University of Craiova (Romania). |

**September 2019**

**CRAIOVA**

|  |  |  |
| --- | --- | --- |
|  | UNIVERSITY OF CRAIOVA  FACULTY OF AUTOMATION, COMPUTERS AND ELECTRONICS  COMPUTERS AND INFORMATION TECHNOLOGY DEPARTMENT |  |

**Eclipse plug-in for Object Oriented Programming**

Stoenica Robert-Bogdan

|  |
| --- |
| SCIENTIFIC COORDINATOR  **S.l. Dr. Ing. Stefan Udristoiu.** |

**September 2019**

**CRAIOVA**

*“It always seems impossible until it’s done.”*

Nelson Mandela

**DECLARAȚIE DE ORIGINALITATE**

Subsemnatul Stoenica Robert-Bogdan, student la specializarea Calculatoare Engleza din cadrul Facultății de Automatică, Calculatoare și Electronică a Universității din Craiova, certific prin prezenta că am luat la cunoștință de cele prezentate mai jos şi că îmi asum, în acest context, originalitatea proiectului meu de licenţă:

* cu titlul „Eclipse plug-in for Object Oriented Programming”
* coordonată de S.l. Dr. Ing. Stefan Udristoiu,
* prezentată în sesiunea Septembrie 2019.

La elaborarea proiectului de licenţă, se consideră plagiat una dintre următoarele acţiuni:

* reproducerea exactă a cuvintelor unui alt autor, dintr-o altă lucrare, în limba română sau prin traducere dintr-o altă limbă, dacă se omit ghilimele şi referinţa precisă,
* redarea cu alte cuvinte, reformularea prin cuvinte proprii sau rezumarea ideilor din alte lucrări, dacă nu se indică sursa bibliografică,
* prezentarea unor date experimentale obţinute sau a unor aplicaţii realizate de alţi autori fără menţionarea corectă a acestor surse,
* însuşirea totală sau parţială a unei lucrări în care regulile de mai sus sunt respectate, dar care are alt autor.

Pentru evitarea acestor situaţii neplăcute se recomandă:

* plasarea între ghilimele a citatelor directe şi indicarea referinţei într-o listă corespunzătoare la sfărşitul lucrării,
* indicarea în text a reformulării unei idei, opinii sau teorii şi corespunzător în lista de referinţe a sursei originale de la care s-a făcut preluarea,
* precizarea sursei de la care s-au preluat date experimentale, descrieri tehnice, figuri, imagini, statistici, tabele et caetera,
* precizarea referinţelor poate fi omisă dacă se folosesc informaţii sau teorii arhicunoscute, a căror paternitate este unanim cunoscută și acceptată.

Data, Semnătura candidatului,

|  |  |  |
| --- | --- | --- |
|  | UNIVERSITATEA DIN CRAIOVA  Facultatea de Automatică, Calculatoare şi Electronică  Departamentul de Calculatoare și Tehnologia Informației | Aprobat la data de  …………………  Şef de departament,  Prof. dr. ing.  Marius BREZOVAN |

**PROIECTUL DE DIPLOMĂ**

|  |  |
| --- | --- |
| Numele și prenumele studentului: | Stoenica Robert-Bogdan |
| Enunțul temei: | Eclipse plug-in for Object Oriented Programming |
| Datele de pornire: |  |
| Conținutul proiectului: |  |
| Material grafic obligatoriu: |  |
| Consultații: | Periodice |
| Conducătorul științific  (titlul, nume și prenume, semnătura): | S.l. Dr. Ing. Stefan Udristoiu. |
| Data eliberării temei: |  |
| Termenul estimat de predare a proiectului: |  |
| Data predării proiectului de către student și semnătura acestuia: |  |

|  |  |  |
| --- | --- | --- |
|  | UNIVERSITATEA DIN CRAIOVA  Facultatea de Automatică, Calculatoare şi Electronică  Departamentul de Calculatoare și Tehnologia Informației |  |

**REFERATUL CONDUCĂTORULUI ȘTIINȚIFIC**

|  |  |
| --- | --- |
| Numele și prenumele candidatului: | Stoenica Robert-Bogdan |
| Specializarea: | Calculatoare Engleza |
| Titlul proiectului: | Eclipse plug-in for Object Oriented Programming |
| Locația în care s-a realizat practica de documentare (se bifează una sau mai multe din opțiunile din dreapta): | În facultate □ |
| În producție □ |
| În cercetare □ |
| Altă locație: [*se detaliază*] |

În urma analizei lucrării candidatului au fost constatate următoarele:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Nivelul documentării | | Insuficient  □ | Satisfăcător □ | Bine  □ | Foarte bine  □ |
| Tipul proiectului | | Cercetare  □ | Proiectare  □ | Realizare practică □ | Altul  [*se detaliază*] |
| Aparatul matematic utilizat | | Simplu  □ | Mediu  □ | Complex □ | Absent  □ |
| Utilitate | | Contract de cercetare □ | Cercetare internă □ | Utilare  □ | Altul  [*se detaliază*] |
| Redactarea lucrării | | Insuficient  □ | Satisfăcător □ | Bine  □ | Foarte bine  □ |
| Partea grafică, desene | | Insuficientă  □ | Satisfăcătoare □ | Bună  □ | Foarte bună  □ |
| Realizarea practică | Contribuția autorului | Insuficientă  □ | Satisfăcătoare □ | Mare  □ | Foarte mare  □ |
| Complexitatea  temei | Simplă  □ | Medie  □ | Mare  □ | Complexă  □ |
| Analiza cerințelor | Insuficient  □ | Satisfăcător □ | Bine  □ | Foarte bine  □ |
| Arhitectura | Simplă  □ | Medie  □ | Mare  □ | Complexă  □ |
| Întocmirea specificațiilor funcționale | Insuficientă  □ | Satisfăcătoare □ | Bună  □ | Foarte bună  □ |
| Implementarea | Insuficientă  □ | Satisfăcătoare □ | Bună  □ | Foarte bună  □ |
| Testarea | Insuficientă  □ | Satisfăcătoare □ | Bună  □ | Foarte bună  □ |
| Funcționarea | Da  □ | Parțială  □ | Nu  □ | |
| Rezultate experimentale | | Experiment propriu  □ | | Preluare din bibliografie  □ | |
| Bibliografie | | Cărți | Reviste | Articole | Referințe web |
| Comentarii  și  observații | |  | | | |

În concluzie, se propune:

|  |  |
| --- | --- |
| ADMITEREA PROIECTULUI  □ | RESPINGEREA PROIECTULUI  □ |

Data, Semnătura conducătorului științific,

**Project Summary**

This project aims to ease the writing of Object Oriented Programming code by automatically generating a factory template for a chosen class, or adding a creation method for said class in the already existing one assuming such a factory does already exist by making use of a parser.

***Key Words****: Object Oriented Programming · Factory · Parser*

***Mulțumiri***

Pe această cale aș dori să le mulțumesc persoanelor care m-au susținut si călăuzit pe parcursul elaborării lucrării de licență, în special conducătorului științific care m-a îndrumat și sprijinit pentru a putea realiza această lucrare, cât și familiei care mi-a oferit un enorm suport moral.

Mulțumesc în special domnului S.l. Dr. Ing. Stefan Udristoiu pentru îndrumare, dar și altor cadre didactice.

Table of Contents

[**Project Summary** 9](#_Toc19150923)

[Tables List 12](#_Toc19150924)

[Figure List 13](#_Toc19150925)

[1. Introduction 14](#_Toc19150926)

[2. Domain Knowledge and Tools 15](#_Toc19150927)

[2.1 Object Oriented Programming 15](#_Toc19150928)

[2.1.1 Introduction to OOP 15](#_Toc19150929)

[2.1.2 The SOLID principle 17](#_Toc19150930)

[2.1.2.1 The single responsibility principle (SRP) 18](#_Toc19150931)

[2.1.2.2 The open closed principle (OCP) 18](#_Toc19150932)

[2.1.2.3 The Liskov substitution principle (LSP) 19](#_Toc19150933)

[2.1.2.4 The interface segregation principle (ISP) 20](#_Toc19150934)

[2.1.2.5 The dependency inversion principle (DIP) 21](#_Toc19150935)

[2.1.3 The factory design pattern 23](#_Toc19150936)

[2.2 Eclipse 25](#_Toc19150937)

[2.2.1 The Plug-in Development Environment (PDE) 25](#_Toc19150938)

[2.3 Parsing 27](#_Toc19150939)

[3. Implementation 31](#_Toc19150940)

[3.1 Plug-in side 31](#_Toc19150941)

[3.1.1 Creating the UI button for the action 31](#_Toc19150942)

[3.1.2 Integrating JavaParser 32](#_Toc19150943)

[3.1.3 The handler 33](#_Toc19150944)

[3.2 Parser side 35](#_Toc19150945)

[3.2.1 ClassOrInterface 35](#_Toc19150946)

[3.2.2 FileManager 37](#_Toc19150947)

[3.2.3 Factorizer 39](#_Toc19150948)

[4. Conclusions 42](#_Toc19150949)

[5. Future Works 42](#_Toc19150950)

[References 43](#_Toc19150951)

# Tables List

[Table 1. ISP example 2](#_Toc12647039)1

# Figure List

[Figure 1. The basic OOP concepts 16](#_Toc12921922)

[Figure 2. SOLID Principles 17](#_Toc12921923)

[Figure 3. DIP Principle 22](#_Toc12921924)

[Figure 4. Factory UML 24](#_Toc12921925)

[Figure 5. Overview of a parser 30](#_Toc12921926)

[Figure 6. Adding an extension to a plug-in project](#_Toc12921927) 31

[Figure 7. The UI extensions, menu and command 32](#_Toc12921928)

[Figure 8. Creating JavaParser plug-in 33](#_Toc12921929)

# Introduction

For as long as Object Oriented Programming and the factory design pattern has been around, writing factories has never been an engaging or particularly intensive activity. Most would call it nothing more than a chore, so in order to make something rather trivial but bothersome much easier I decided to create a plug-in for a widespread IDE, Eclipse which generates factory templates by parsing your project and creating your factory class by itself.

# Domain Knowledge and Tools

# Object Oriented Programming

# Introduction to OOP

“Object-Oriented Programming (OOP) is the term used to describe a programming approach based on objects and classes. The object-oriented paradigm allows us to organize software as a collection of objects that consist of both data and behaviour. This is in contrast to conventional functional programming practice that only loosely connects data and behaviour.

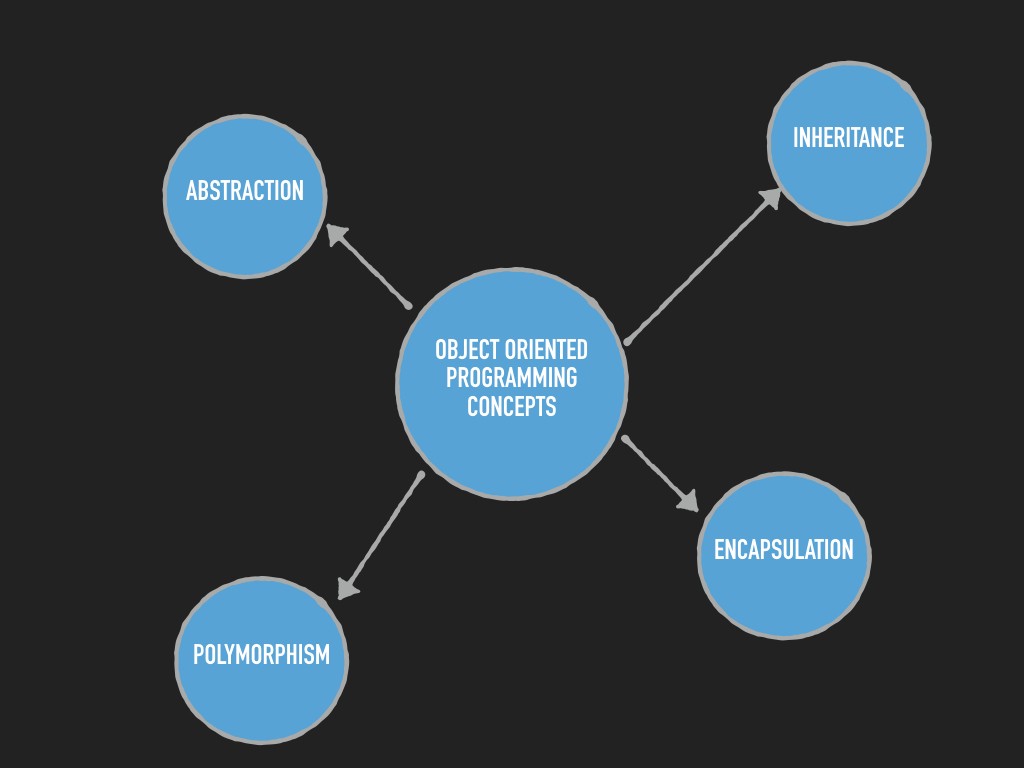
Since the 1980s the word 'object' has appeared in relation to programming languages, with almost all languages developed since 1990 having object-oriented features. Some languages have even had object-oriented features retro-fitted. It is widely accepted that object-oriented programming is the most important and powerful way of creating software.

The object-oriented programming approach encourages:

* Modularization: where the application can be decomposed into modules.
* Software re-use: where an application can be composed from existing and new modules.

An object-oriented programming language generally supports five main features:

* Classes
* Objects
* Classification
* Polymorphism
* Inheritance” (Molloy, n.d.)

 Figure 1 – The basic OOP concepts

# The SOLID principle

**“SOLID** is short for basic five principles of OOP, which was introduced in the early 2000s and adopted widely in the software industry. When these principles are combined together, a programmer can create an application that will be easy to maintain and can be extended over time.

The SOLID abbreviation is defined as follows:

* **S**: Single responsibility principle
* **O**:Open closed principle
* **L**: Liskov substitution principle
* **I**: Interface segregation principle
* **D**: Dependency inversion principle

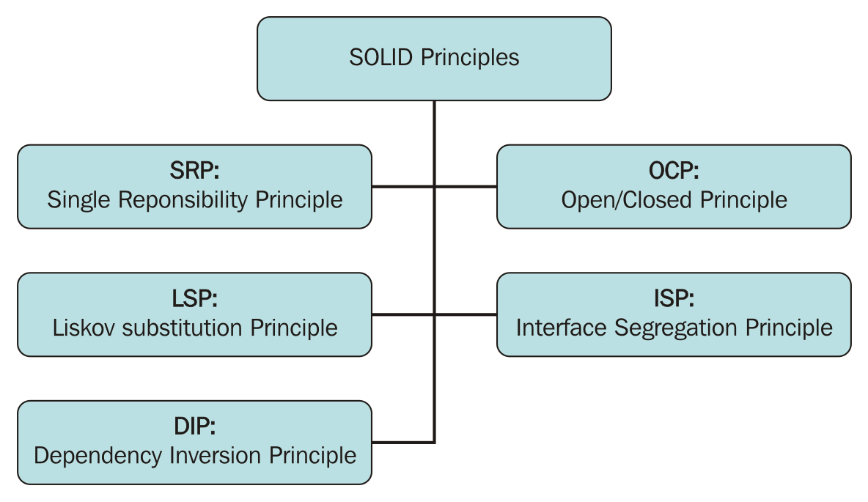


Figure 2 – SOLID Principles

# The single responsibility principle (SRP)

This states that a class should have only one reason to change it, and this means, it should have a single job.

If we can write code for multiple functionalities in a class, it doesn't mean that we should. Smaller classes and smaller methods will give us more flexibility, and we don't have to write a lot of extra code. It saves us from over complicating classes and helps in achieving high cohesion.

For example, the Person class has the code to show the available balance and deduct it from Account. This is a clear violation of SRP. This class has two reasons to change: if any attribute of Person changes or any information about Account changes.

The advantages of SRP are as follows:

* It makes code as easy as possible to reuse
* Small classes can be changed easily
* Small classes are more readable

Splitting classes is a way to implement SRP. Another example of the SRP violation is God classes, which we will discuss in the next chapter.

# The open closed principle (OCP)

This states that entities of software, such as classes and methods, should be open for extension but closed for modification. This means that classes and methods should be allowed to be extended without modification.

For example, a class returns report data in the string and XML formats. In future, we may want to return data in the JSON or CSV format. We should not modify the existing class as it may have an impact on all the other classes using it. It would be a violation of OCP.

The importance of OCP lies in the following scenarios:

* Any changes made in any existing code can potentially impact the entire system
* In some conditions, we cannot change code (the managed package in Apex), so OCP is implied

We can implement OCP using design patterns, such as the strategy pattern. In the preceding scenario, we can create an interface of the ReportData type and different classes implementing that interface to return different report formats. We will discuss this in more detail in the upcoming chapters.

# The Liskov substitution principle (LSP)

This states that if class B is a child of class A, then A can be replaced by B, without changing anything in a program. In other words, the LSP principle states that you should not encounter unexpected results if child (derived) classes are used instead of parent classes.

This principle is also known as **Substitutability** and was introduced by **Barbara Liskov** in 1987. This is one of the most widely used principles in programming. You might be already using this, but may not know that it is called LSP.

For example, let's say that we have a  Customer\_Ticket class defined to close a case using the close() method. A  Customet\_Ticket\_Escalated child class is defined as well to handle an escalated case; however, it cannot close a case by a normal process because the customer was not happy. If we substitute a parent class by this child class and call the close() method, it will throw an exception, which is a clear violation of LSP.

The following code snippet explains this scenario:

public virual class Customer\_Ticket{

String status ;

public virtual void close(){

status = 'close';

}

//other code

}

public class Customet\_Ticket\_Escalated extends Customer\_Ticket{

public override void close(){

throw new Exception('As this is escalated case therefore

cannot be closed by normal process');

}

//other code

}

The anonymous Apex code for testing is as follows:

Customer\_Ticket issue = new Customet\_Ticket\_Escalated();

issue.close();//runtime exception, violation of LSP

To implement LSP, a proper use of inheritance with a protected access specifier is needed, and a parent class should not have any attributes, which may not apply to every child class.

# The interface segregation principle (ISP)

This states that do not force a child class to depend on a method that is not used for them. This principle suggests that you break interfaces into smaller ones so that a client can only implement an interface that is of interest. This principle is very similar to the high cohesive principle, as discussed earlier.

One way to identify the ISP violation is if we implement any interface or derive a base class where we need to throw an exception for an unsupported operation.

The ISP are as follows:

* It enforces the single responsibility principle for interfaces and base classes
* Any changes made in the interface may affect child classes even though they are not using unused methods

For example, Product is an interface and contains the Name and Author attributes. Two child classes named Movie and Book are derived from Product. However, Movie is a Product but does not have an author, and therefore a runtime exception would be thrown if it's used.

The following example shows the valid and invalid code according to the ISP:

|  |  |
| --- | --- |
| **Violation of ISP** | **Adheres ISP** |
| Public interface Product{  Public String getName();  Public String getAuthor();  }  Public Class Movie implements Product{  private String movieName;  private String author;  Public String getName(){  return movieName;  }  Public String getAuthor(){  return new CustomException('Method not Supported');  }  }  Anonymous apex code for testing is as follows:  Product m = new Movie();  m.getAuthor();//runtime exception | Public interface Product{  Public String getName();  Public String getAuthor();  }  Public Class Book implements Product{  private String bookName;  private String author;  Public String getName(){  return bookName;  }  Public String getAuthor(){  return author;  }  }  Anonymous apex code for testing is as follows:  Product p = new Book();  p.getAuthor(); //works |

Table 1. ISP example

# The dependency inversion principle (DIP)

This states that modules should not depend on each other directly and should depend via an interface (abstraction).

In other words, two classes should not be tightly coupled. Tightly coupled classes cannot work independently of each other, and if a change is required, then it creates a wave of changes throughout the application.

One way to identify a DIP violation is the use of a new keyword in the same class. If we are using a new keyword, then this means that we are trying to instantiate a class directly. We can create a container class to delegate the creation of a new object. This class will know how to instantiate another class on the basis of the interface type. This approach is also known as **dependency injection**or**Inversion of Control**(**IoC**). If you know about the trigger factory pattern that is widely used in Apex, then you may be able to relate with it, else we will discuss this in the upcoming chapters.

For example, in the real world you would not want to solder a lamp directly to the electrical wiring; we would rather use a plug so that the lamp can be used in any electric outlet. In this case, the lamp and electric outlet are the class and the plug is the interface.

Class A should not know any details about how class B is implemented. An interface should be used for communication. As discussed earlier, if needed we can always create a new child class from the interface and use it as per the LSP principle.

The following screenshot shows a scenario before and after DIP. In the first case, the Apex scheduler directly uses classes to calculate sharing and assigns a record to the user. All three classes are tightly coupled in this case. As per DIP, we need to introduce interfaces between them so that classes do not depend on implementation, but they will depend on the abstraction (interface).” (Apex Design Patterns, 2016)

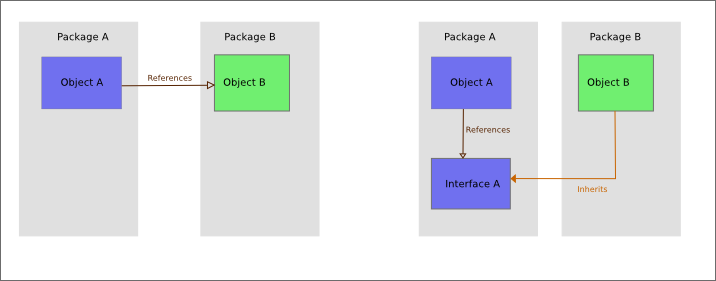


Figure 3 – DIP principle

# The factory design pattern

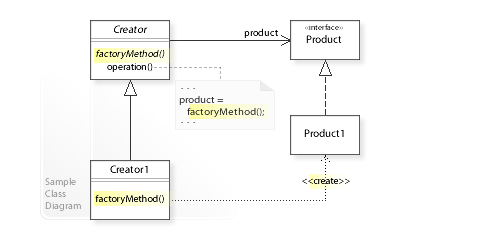
“The Factory Method design pattern is one of the "Gang of Four" design patterns that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

The Factory Method design pattern is used instead of the regular class constructor for keeping within the SOLID principle of programming, decoupling the construction of objects from the objects themselves. This has the following advantages and is useful for the following cases, among others:

* Allows construction of classes with a component of a type that has not been predetermined, but only defined in an "interface", or which is defined as a dynamic type. Thus, for example, a class Vehicle that has a member Motor of interface IMotor, but no concrete type of Motor defined in advance, can be constructed by telling the Vehicle constructor to use an ElectricMotor or a GasolineMotor. The Vehicle constructor code then calls a Motor factory method, to create the desired Motor that complies with the IMotor interface.
* Allows construction of subclasses to a parent whose component type has not been predetermined, but only defined in an interface, or which is defined as a dynamic type. For example, a class Vehicle with a member Motor defined with a dynamic type, can have subclasses of type ElectricPlane and OldCar each constructed with a different type of Motor. This can be accomplished by constructing the subclasses with a Vehicle factory method, while supplying the motor type. In cases like this the constructor may be hidden.
* Allows for more readable code in cases where multiple constructors exist, each for a different reason. For example if there are two constructors Vehicle(make:string, motor:number) and Vehicle(make:string, owner:string, license:number, purchased:date) a more readable construction of the classes would be to use Vehicle.CreateOwnership(make:string, owner:string, license:number, purchased: date) vs Vehicle.Create(make:string, motor:number)
* Allows a class to defer instantiation to subclasses, and to prevent direct instantiation of an object of the parent class type. For example, a Vehicle can be prevented from being instantiated directly since it has no constructor, and only subclasses like ElectricPlane or OldCar can be created by calling the Vehicle (static) factory method in the subclass constructor or initializer.

Creating an object directly within the class that requires or uses the object is inflexible because it commits the class to a particular object and makes it impossible to change the instantiation independently of the class. A change to the instantiator would require a change to the class code which we would rather not touch. This is referred to as code coupling and the Factory method pattern assists in decoupling the code.

The Factory Method design pattern is used by first defining a separate operation, a factory method, for creating an object, and then using this factory method by calling it to create the object. This enables writing of subclasses that decide how a parent object is created and what type of objects the parent contains.” (Factory method pattern, n.d.)

Figure 4 – Factory UML

# Eclipse

“Eclipse is an integrated development environment (IDE) used in computer programming, and in 2014 was the most widely used Java IDE in one website's poll. It contains a base workspace and an extensible plug-in system for customizing the environment. Eclipse is written mostly in Java and its primary use is for developing Java applications, but it may also be used to develop applications in other programming languages via plug-ins.” (Eclipse (software), n.d.)

# The Plug-in Development Environment (PDE)

“The Plug-in Development Environment (PDE) provides tools to create, develop, test, debug, build and deploy Eclipse plug-ins, fragments, features, update sites and RCP products.

PDE also provides comprehensive OSGi tooling, which makes it an ideal environment for component programming, not just Eclipse plug-in development.

PDE Components

The PDE subproject consists out of UI, API Tools and Build. We also have an Incubator component where we can develop non-SDK features.

|  |  |
| --- | --- |
| **Name** | **Description** |
| PDE UI | Models, builders, editors and more to faciliate plug-in development in the Eclipse IDE. |
| PDE API Tools | Eclipse IDE and build process integrated tooling to maintain API. |
| PDE Build | Ant based tools and scripts to automate build processes. |
| PDE Incubator | Development of new tools that are not ready to be added to the Eclipse SDK |

PDE UI

The PDE UI component provides a comprehensive set of tools to create, develop, test, debug and deploy Eclipse plug-ins, fragments, features, update sites and RCP products.

PDE UI also provides comprehensive OSGi tooling, which makes it an ideal environment for component programming, not just Eclipse plug-in development.

Here is a small list of what PDE UI provides to the Eclipse SDK:

* Form-Based Manifest Editors - multi-page editors that centrally manage all manifest files of a plug-in or feature.
* RCP Tools - wizards and a form-based editor that allow you to define, brand, test and export products to multiple platforms.
* New Project Creation Wizards - create a new plug-in, fragment, feature, feature patch and update sites.
* Import Wizards - import plug-ins and features from the file system.
* Export Wizards - wizards that build, package and export plug-ins, fragments and products with a single click.
* Launchers - test and debug Eclipse applications and OSGi bundles.
* Views - PDE provides views that help plug-in developers inspect different aspects of their development environment.
* Miscellaneous Tools - wizards to externalize and clean up manifest files.
* Conversion Tools - wizard to convert a plain Java project or plain JARs into a plug-in project.
* Integration with JDT - plug-in manifest files participate in Java search and refactoring.
* User Assistance Tools - Editors and tools for developing user help and other UA documents.
* Declarative Services Tools - Editors and validation for OSGi declarative services.

PDE Build

The goal of PDE Build is to facilitate the automation of plug-in build processes. Essentially, PDE Build produces Ant scripts based on development-time information provided by, for example, the plugin.xml and build.properties files. The generated Ant scripts, can fetch the relevant projects from a CVS repository, build jars, Javadoc, source zips, put everything together in a format ready to ship and send it out to a remote location (e.g., a local network or a downloads server).

While PDE Build is still being maintained, it is not actively enhanced. For new builds, you can also consider other build systems like Maven Tycho or Gradle.

PDE API Tools

API tooling will assist developers in API maintenance by reporting API defects such as binary incompatibilities, incorrect plug-in version numbers, missing or incorrect @since tags, and usage of non-API code between plug-ins. The tooling will be integrated in the Eclipse SDK and will be used in the automated build process.

Specifically, the tooling is designed to do the following:

* Identify binary compatibility issues between two versions of a software component or product.
* Update version numbers for plug-ins (bundles) based on the Eclipse versioning scheme.
* Update @since tags for newly added classes, interfaces, methods, etc.
* Provide new javadoc tags and code assist to annotate types with special restrictions.
* Leverage existing information (in MANIFEST.MF) to define the visibility of packages between bundles.
* Identify usage of non-API code between plug-ins.
* Identity leakage of non-API types into API.” (PDE, n.d.)

# Parsing

“Parsing, syntax analysis, or syntactic analysis is the process of analysing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar. The term parsing comes from Latin pars (orationis), meaning part (of speech).

The term has slightly different meanings in different branches of linguistics and computer science. Traditional sentence parsing is often performed as a method of understanding the exact meaning of a sentence or word, sometimes with the aid of devices such as sentence diagrams. It usually emphasizes the importance of grammatical divisions such as subject and predicate.

Within computational linguistics the term is used to refer to the formal analysis by a computer of a sentence or other string of words into its constituents, resulting in a parse tree showing their syntactic relation to each other, which may also contain semantic and other information. Some parsing algorithms may generate a parse forest or list of parse trees for a syntactically ambiguous input.

The term is also used in psycholinguistics when describing language comprehension. In this context, parsing refers to the way that human beings analyze a sentence or phrase (in spoken language or text) "in terms of grammatical constituents, identifying the parts of speech, syntactic relations, etc." This term is especially common when discussing what linguistic cues help speakers to interpret garden-path sentences.

Within computer science, the term is used in the analysis of computer languages, referring to the syntactic analysis of the input code into its component parts in order to facilitate the writing of compilers and interpreters. The term may also be used to describe a split or separation.

**A parser** is a software component that takes input data (frequently text) and builds a data structure – often some kind of parse tree, abstract syntax tree or other hierarchical structure, giving a structural representation of the input while checking for correct syntax. The parsing may be preceded or followed by other steps, or these may be combined into a single step. The parser is often preceded by a separate lexical analyser, which creates tokens from the sequence of input characters; alternatively, these can be combined in scannerless parsing. Parsers may be programmed by hand or may be automatically or semi-automatically generated by a parser generator. Parsing is complementary to templating, which produces formatted output. These may be applied to different domains, but often appear together, such as the scanf/printf pair, or the input (front end parsing) and output (back end code generation) stages of a compiler.

The input to a parser is often text in some computer language, but may also be text in a natural language or less structured textual data, in which case generally only certain parts of the text are extracted, rather than a parse tree being constructed. Parsers range from very simple functions such as scanf, to complex programs such as the frontend of a C++ compiler or the HTML parser of a web browser. An important class of simple parsing is done using regular expressions, in which a group of regular expressions defines a regular language and a regular expression engine automatically generating a parser for that language, allowing pattern matching and extraction of text. In other contexts regular expressions are instead used prior to parsing, as the lexing step whose output is then used by the parser.

The use of parsers varies by input. In the case of data languages, a parser is often found as the file reading facility of a program, such as reading in HTML or XML text; these examples are markup languages. In the case of programming languages, a parser is a component of a compiler or interpreter, which parses the source code of a computer programming language to create some form of internal representation; the parser is a key step in the compiler frontend. Programming languages tend to be specified in terms of a deterministic context-free grammar because fast and efficient parsers can be written for them. For compilers, the parsing itself can be done in one pass or multiple passes – see one-pass compiler and multi-pass compiler.

The implied disadvantages of a one-pass compiler can largely be overcome by adding fix-ups, where provision is made for code relocation during the forward pass, and the fix-ups are applied backwards when the current program segment has been recognized as having been completed. An example where such a fix-up mechanism would be useful would be a forward GOTO statement, where the target of the GOTO is unknown until the program segment is completed. In this case, the application of the fix-up would be delayed until the target of the GOTO was recognized. Conversely, a backward GOTO does not require a fix-up, as the location will already be known.

Context-free grammars are limited in the extent to which they can express all of the requirements of a language. Informally, the reason is that the memory of such a language is limited. The grammar cannot remember the presence of a construct over an arbitrarily long input; this is necessary for a language in which, for example, a name must be declared before it may be referenced. More powerful grammars that can express this constraint, however, cannot be parsed efficiently. Thus, it is a common strategy to create a relaxed parser for a context-free grammar which accepts a superset of the desired language constructs (that is, it accepts some invalid constructs); later, the unwanted constructs can be filtered out at the semantic analysis (contextual analysis) step.

For example, in Python the following is syntactically valid code:

x = 1

**print**(x)

The following code, however, is syntactically valid in terms of the context-free grammar, yielding a syntax tree with the same structure as the previous, but is syntactically invalid in terms of the context-sensitive grammar, which requires that variables be initialized before use:

x = 1

**print**(y)

Rather than being analyzed at the parsing stage, this is caught by checking the values in the syntax tree, hence as part of semantic analysis: context-sensitive syntax is in practice often more easily analyzed as semantics.” (Parsing, n.d.)

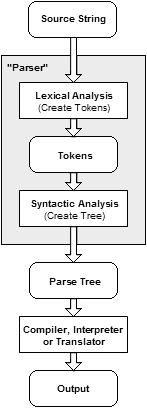


Figure 5 – Overview of a parser

# Implementation

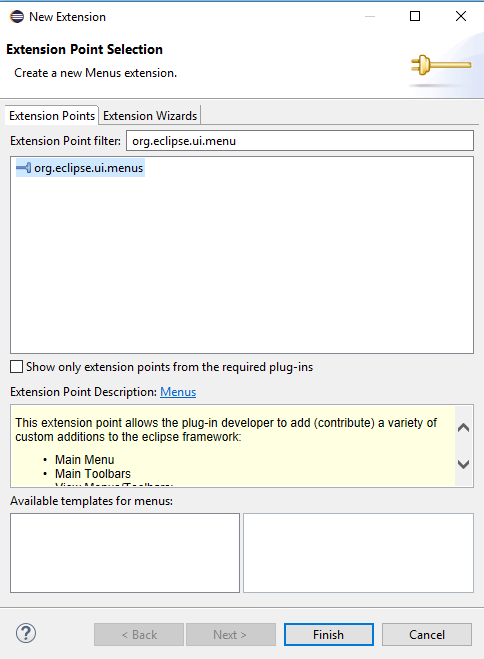
The implementation can be divide into two different sections, the plug-in side, with all that that requires for it to be implemented as a plug-in, and the parser side, with implementing the parsing and generation of the code.

# Plug-in side

# Creating the UI button for the action

In order to access the plug-in action we will need an option from which to do so.

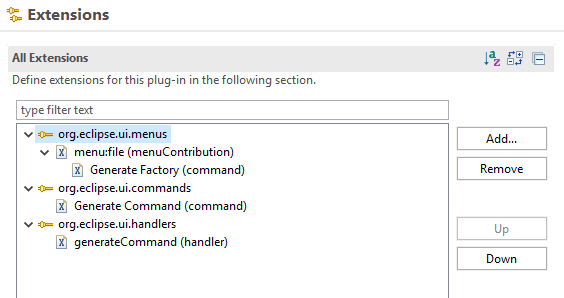
We create an option in the File menu of the Eclipse interface.

First we’ll need to add the extensions:

*Figure 6 – Adding an extension to a plug-in project*

Rinse and repeat two more times for *org.eclipse.ui.commands* and *org.eclipse.ui.handlers*.

Now with our extensions in place, we start creating the menu and commands for it.

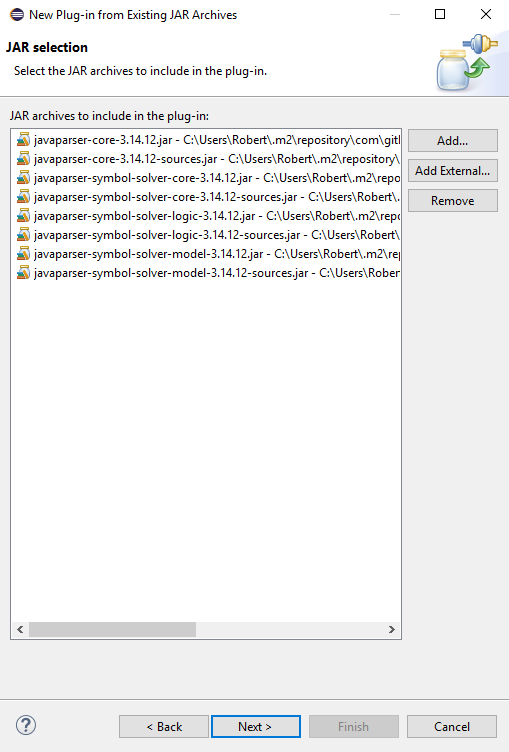


*Figure 7 – The UI extensions, menu and command*

# Integrating JavaParser

For us to integrate a third party library like JavaParser, we need to first turn it into a plug-in of its own.

In order to do that, we simply create a new plug-in project using the Jar files, which we then add as a dependency to our plug-in project.



*Figure 8 – Creating JavaParser plug-in*

# The handler

The first step is getting the file path and machine it into the components we need, which are the Class name (without the extension) and the project path.

The next step is creating a parser with our project path which we will use to create a Factorizer object which generate the factory itself based on the class.

1. **public** **class** GenerationHandler **extends** AbstractHandler {
3. String filePath;
5. @Override
6. **public** Object execute(ExecutionEvent event) **throws** ExecutionException {
8. IWorkbench workbench = PlatformUI.getWorkbench();
9. IWorkbenchWindow window =  workbench == **null** ? **null** : workbench.getActiveWorkbenchWindow();
10. IWorkbenchPage activePage = window == **null** ? **null** : window.getActivePage();
12. IEditorPart editor =  activePage == **null** ? **null** : activePage.getActiveEditor();
13. IEditorInput input =  editor == **null** ? **null** : editor.getEditorInput();
14. IPath path = input **instanceof** FileEditorInput ? ((FileEditorInput)input).getPath() : **null**;
15. **if** (path != **null**)
16. {
17. filePath = path.toString();
18. }
20. filePath = filePath.substring(0, filePath.lastIndexOf('.'));
22. filePath = filePath.replaceAll("/", "\\\\");
24. String className = filePath.substring(filePath.lastIndexOf('\\') + 1);
26. String projectPath = filePath.substring(0,filePath.lastIndexOf('\\'));
28. FileManager parser = **new** FileManager(projectPath);
30. Factorizer factorizer = **new** Factorizer(parser);
32. **try** {
33. ClassOrInterface classOrInterface = factorizer.factorize(className);
35. parser.write(classOrInterface);
37. } **catch** (Throwable e) {
38. // TODO Auto-generated catch block
39. e.printStackTrace();
40. }
42. **return** **null**;
43. }
44. }

# Parser side

We use 3 main classes to complete the process of generating the factory: *FileManager*, *ClassOrInterface* and *Factorizer*.

# ClassOrInterface

This class contains methods for validating whether a file is a class or an interface (like the name suggests) but it also has methods to machine the eventual methods in the factory for creating the object.

1. **public** **class** ClassOrInterface{

4. **private** ClassOrInterfaceDeclaration \_classOrInterface;

7. **public** ClassOrInterface(String name)
8. {
9. CompilationUnit compilationUnit = **new** CompilationUnit();
11. \_classOrInterface = compilationUnit.addClass(name);
12. }
14. **public** ClassOrInterface(ClassOrInterfaceDeclaration classOrInterface)
15. {
16. \_classOrInterface = classOrInterface;
17. }
19. **public** String getName()
20. {
21. **return** \_classOrInterface.getName().getIdentifier();
22. }
24. **public** ClassOrInterfaceType getType()
25. {
26. **return** StaticJavaParser.parseClassOrInterfaceType(\_classOrInterface.asTypeDeclaration().getNameAsString()).asClassOrInterfaceType();
27. }
29. **public** ArrayList<FieldDeclaration> getMembers()
30. {
31. ArrayList<FieldDeclaration> fieldDeclarations = **new** ArrayList<FieldDeclaration>();
33. **for**(BodyDeclaration<?> bodyDeclaration: \_classOrInterface.getMembers())
34. {
35. **if**(bodyDeclaration **instanceof** FieldDeclaration)
36. {
37. fieldDeclarations.add((FieldDeclaration)bodyDeclaration);
38. }
39. }
41. **return** fieldDeclarations;
42. }
44. **public** ArrayList<ConstructorDeclaration> getConstructors()
45. {
46. ArrayList<ConstructorDeclaration> constructorDeclarations = **new** ArrayList <ConstructorDeclaration>();
48. **for**(BodyDeclaration<?> bodyDeclaration: \_classOrInterface.getMembers())
49. {
50. **if**(bodyDeclaration **instanceof** ConstructorDeclaration)
51. {
52. constructorDeclarations.add((ConstructorDeclaration)bodyDeclaration);
53. }
54. }
56. **return** constructorDeclarations;
57. }

60. **public** ArrayList<MethodDeclaration> getMethods()
61. {
62. ArrayList<MethodDeclaration> methodDeclarations = **new** ArrayList <MethodDeclaration>();
64. **for**(BodyDeclaration<?> bodyDeclaration: \_classOrInterface.getMembers())
65. {
66. **if**(bodyDeclaration **instanceof** MethodDeclaration)
67. {
68. methodDeclarations.add((MethodDeclaration)bodyDeclaration);
69. }
70. }
72. **return** methodDeclarations;
73. }
75. **public** **void** AddMethod(**final** NodeList<Modifier> modifiers, **final** String name, **final** Type type, **final** NodeList<Parameter> parameters, BlockStmt body)
76. {
77. \_classOrInterface.addMember( **new** MethodDeclaration(modifiers,**new** NodeList<AnnotationExpr>() , **new** NodeList<TypeParameter>(),type,**new** SimpleName(name), parameters, **new** NodeList<ReferenceType>(),body));
78. }
80. **public** **void** AddEnum(**final** NodeList<Modifier> modifiers, **final** String name)
81. {
82. \_classOrInterface.addMember(**new** EnumDeclaration(modifiers, name));
83. }
85. **public** ArrayList<EnumDeclaration> getEnums()
86. {
87. ArrayList<EnumDeclaration> enumDeclarations = **new** ArrayList <EnumDeclaration>();
89. **for**(BodyDeclaration<?> bodyDeclaration: \_classOrInterface.getMembers())
90. {
91. **if**(bodyDeclaration **instanceof** EnumDeclaration)
92. {
93. enumDeclarations.add((EnumDeclaration)bodyDeclaration);
94. }
95. }
97. **return** enumDeclarations;
98. }
100. **public** ArrayList<ClassOrInterfaceType> getImplements()
101. {
102. ArrayList<ClassOrInterfaceType> implementsList = **new** ArrayList<ClassOrInterfaceType>();
103. **for**(ClassOrInterfaceType implement: \_classOrInterface.getImplementedTypes())
104. {
105. implementsList.add(implement);
106. }
108. **return** implementsList;
109. }
111. @Override
112. **public** String toString()
113. {
114. **return** \_classOrInterface.toString();
115. }
117. **public** Boolean isInterface()
118. {
119. **return** \_classOrInterface.isInterface();
120. }
122. }

# FileManager

This class accesses the file in question, takes it through the necessary steps using *ClassOrInterface* and then outputs the result by creating the factory file.

1. **public** **class** FileManager {
3. **private** String \_workingDirectory;
5. **private** HashMap<String, ClassOrInterface> \_classesOrInterfaces;
7. **public** FileManager(String workingDirectory)
8. {
9. \_workingDirectory = workingDirectory;
11. \_classesOrInterfaces = **new** HashMap<String, ClassOrInterface>();
12. }

15. **public** Boolean canParse(String className)
16. {
17. File file = **new** File(\_workingDirectory + "\\" + className + ".java");
18. **return** file.exists() && !file.isDirectory();
19. }
21. **public** ClassOrInterface parse(String className)
22. {
24. **if**(\_classesOrInterfaces.containsKey(className))
25. {
26. **return** \_classesOrInterfaces.get(className);
27. }
29. ArrayList<ClassOrInterface> classInfo = **new** ArrayList<ClassOrInterface>();
31. **try** {
32. **new** VoidVisitorAdapter<ArrayList<ClassOrInterface> >()
33. {
35. @Override
36. **public** **void** visit(ClassOrInterfaceDeclaration classOrInterface, ArrayList<ClassOrInterface> arg) {
37. **super**.visit(classOrInterface, arg);
39. arg.add(**new** ClassOrInterface(classOrInterface));
40. }
42. }.visit(StaticJavaParser.parse(**new** File(\_workingDirectory + "\\" + className + ".java")), classInfo);
43. } **catch** (FileNotFoundException e) {
44. // TODO Auto-generated catch block
45. e.printStackTrace();
46. }
48. ClassOrInterface desiredClass = **null**;
50. **for**(ClassOrInterface classOrInterface: classInfo)
51. {
52. **if**(\_classesOrInterfaces.containsKey(classOrInterface.getName()) == **false**)
53. {
54. \_classesOrInterfaces.put(classOrInterface.getName(), classOrInterface);
55. }
57. **if**(classOrInterface.getName().equals(className))
58. {
59. desiredClass = classOrInterface;
60. }
61. }
63. **return** desiredClass;
64. }

67. **public** **void** write(ClassOrInterface classOrInterface)
68. {
69. **try** {
70. FileWriter writer = **new** FileWriter(\_workingDirectory + "\\" + classOrInterface.getName() + ".java");
72. writer.write(classOrInterface.toString());
74. writer.close();
76. } **catch** (IOException e) {
77. // TODO Auto-generated catch block
78. e.printStackTrace();
79. }
80. }
81. }

# Factorizer

This is our main class, making use of both the previous ones to not only create the factory file but also add in the necessary creation methods for the eventual objects that are going to be manufactured by the factory.

1. **public** **class** Factorizer {
3. **private** FileManager \_parser;
5. **private** HashMap<String, ClassOrInterface> \_factoryClasses;
7. **public** Factorizer(FileManager parser)
8. {
9. \_parser = parser;
11. \_factoryClasses = **new** HashMap<String, ClassOrInterface>();
12. }
14. **public** ClassOrInterface factorize(String className) **throws** Throwable
15. {
16. ClassOrInterface  currentCos = \_parser.parse(className);
18. **if**(currentCos.isInterface() == **false**)
19. {
20. ArrayList<ClassOrInterfaceType> interfaces = currentCos.getImplements();
22. **if**(interfaces.size() == 1)
23. {
24. ClassOrInterfaceType root = interfaces.get(0);
26. ArrayList<ConstructorDeclaration> constructors = currentCos.getConstructors();
28. **if**(constructors.size() == 0)
29. {
30. constructors.add(**new** ConstructorDeclaration(currentCos.getName().toString()));
31. }
33. **if**(constructors.size() == 1)
34. {
35. NodeList<Parameter> constructorParameters = constructors.get(0).getParameters();
37. String name  = root.getName().getIdentifier();
39. name = name.substring(1);
41. ClassOrInterface factoryClass = **null**;
43. **if**(\_factoryClasses.containsKey(name))
44. {
45. factoryClass = \_factoryClasses.get(name);
46. }
47. **else**
48. {
49. **if**(\_parser.canParse(name + "Factory") == **false**)
50. {
51. factoryClass = **new** ClassOrInterface(name + "Factory");
53. factoryClass.AddEnum(NodeList.nodeList(**new** Modifier(Modifier.Keyword.PUBLIC)), "Enum" + name);
55. Parameter enumParameter = **new** Parameter(**new** TypeParameter("Enum" + name), **new** SimpleName("type"));
57. SwitchStmt stmt = **new** SwitchStmt();
59. stmt.setSelector(**new** NameExpr("type"));
61. BlockStmt block = **new** BlockStmt();
63. block.setStatements(NodeList.nodeList((Statement)stmt));
65. NodeList<Parameter> parameters = **new** NodeList<Parameter>();
67. parameters.add(enumParameter);
69. parameters.addAll(constructorParameters);
71. factoryClass.AddMethod(NodeList.nodeList(**new** Modifier(Modifier.Keyword.PUBLIC), **new** Modifier(Modifier.Keyword.STATIC)),
72. "create" +name, root, parameters , block);
74. \_factoryClasses.put(name, factoryClass);
75. }
76. **else**
77. {
78. factoryClass = \_parser.parse(name + "Factory");
80. \_factoryClasses.put(name, factoryClass);
81. }
82. }
84. EnumDeclaration enums = factoryClass.getEnums().get(0);
86. **for**(EnumConstantDeclaration cstDecl: enums.getEntries())
87. {
88. **if**(cstDecl.getName().getIdentifier().equals((currentCos.getName().toUpperCase())))
89. **return** currentCos;
90. }
92. enums.addEntry(**new** EnumConstantDeclaration(currentCos.getName().toUpperCase()));
94. MethodDeclaration createMethod = factoryClass.getMethods().get(0);
96. SwitchStmt switchStatement = createMethod.getBody().get().getStatements().get(0).asSwitchStmt();
98. SwitchEntry switchEntry = **new** SwitchEntry();
100. switchEntry.setLabels(NodeList.nodeList((Expression)**new** NameExpr("Enum" + name  + "." + currentCos.getName().toUpperCase())));
102. ReturnStmt returnStmt = **new** ReturnStmt();
104. NodeList<Expression> constructorArguments = **new** NodeList<Expression>();
106. NodeList<Parameter> createMethodParameters = createMethod.getParameters();
108. **if**(constructorParameters.size() != createMethodParameters.size() -1)
109. **throw** **new** ConstructorParametersLengthFactorizerError();
111. **for**(**int** i=0;i<createMethodParameters.size() - 1;++i)
112. {
113. **if**(createMethodParameters.get(i+1).getType().toString().equals(constructorParameters.get(i).getType().toString()) == **false**)
114. {
115. **throw** **new** ConstructorParametersFactorizerError();
116. }
117. }
119. **for**(**int** i = 1 ;i < createMethodParameters.size();++i)
120. {
121. constructorArguments.add(**new** NameExpr(createMethodParameters.get(i).getName().getIdentifier()));
122. }
124. returnStmt.setExpression(**new** ObjectCreationExpr(**null**, currentCos.getType(), constructorArguments));
126. switchEntry.setStatements(NodeList.nodeList((Statement)returnStmt));
128. NodeList<SwitchEntry> switchEntries = switchStatement.getEntries();
130. switchEntries.add(switchEntry);
132. switchStatement.setEntries(switchEntries);
134. **return** factoryClass;
135. }
136. **else**
137. {
138. **throw** **new** ManyConstructorsFactorizerError();
139. }
141. }
142. **else** **if**(interfaces.size() == 0)
143. {
144. **throw** **new** NoInterfaceFactorizerError();
145. }
147. }
148. **else**
149. {
150. **throw** **new** NotInterfaceFactorizerError();
151. }
153. **return** **null**;
154. }
155. }

# Conclusions

In this paper we have implemented a plug-in to help with OOP, in particular easing the process of creating factories for objects by generating a template which can then be customized to whatever particular need the user may have.

# Future Works

The plug-in is still fairly simplistic and only helps with a particular task thus it can be improved to further customization and ease of writing code.

Alternatively it can also branch out into other areas of helping with OOP, like warning the user where they do not respect the SOLID principles we have talked about.

# References

(2016). In V. Anshul, & Z. Jitendra, *Apex Design Patterns.*

*Factory method pattern*. (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Factory\_method\_pattern

Molloy, D. (n.d.). *EE402*. Retrieved from http://ee402.eeng.dcu.ie/introduction/chapter-1---introduction-to-object-oriented-programming

*Parsing*. (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Parsing

*PDE*. (n.d.). Retrieved from Eclipse: https://www.eclipse.org/pde/