Thesis Title

Politecnico di Milano

Carlos Blasco Andrés

June 8, 2020

Abstract

Handling Big Data appears as a new problem for our society that aims at extracting useful information from them.

To this purpose, complex platforms have been conceived that support the development of so-called Data Intensive Applications (DIAs). Creating a DIA, though, still requires a complex design process that, besides the problems related to ensuring high performance and availability, includes also the need to provide guarantees in terms of data privacy and to allow the users to define and update privacy policies based on their own preferences.

Thus, StreamGen is a model-driven approach to support the design of DIAs and the automated code generation for two target platforms, Flink and Spark. Moreover, StreamGen is designed with the goal to be easy to handle by users with few knowledge about DIA.

The purpose of this thesis is to extend StreamGen to make it privacy policies-aware. In order to reach such purpose: a) the definition of a language, as part of a UML profile, will be required in order to allow users to define for a DIA two types of privacy policies: View Creation Policies (VCP) and Data Subject Eviction Policies (DSEP). VCPs modify the incoming data taking into account the data subject and some predefined conditions which have to be specified. DSEPs remove the data from the stream when the data comes from a data subject and it satisfies some predefined conditions. Such language is defined taking into account the dataflow model of DIAs and its sources, transformations and sinks approach, making the definition sufficiently simple to be handled by users who are not familiar with DIAs. b) from the defined language and by means of Acceleo, an implementation of a model-to-code transformation will be developed allowing users to automate the generation of the privacy-enhanced DIA code, targeting Flink as final platform. c) the evaluation of the approach will be done by exploiting two case studies. In the first case, a DIA will be developed in order to preserve the privacy of the users who make different transactions among a fixed shop stock. Secondly, a DIA will be modified. This DIA takes the temperatures from two rooms and it computes two statistics, the maximum and the average temperatures, and the prediction of the temperature of the rooms in a given time. This application will be modified in order to avoid the computation of the prediction when the temperatures come from the second room.

Resumen

El manejo de Big Data aparece como un nuevo problema para nuestra sociedad que tiene como objetivo extraer información útil de ellos.

Para este propósito, se han concebido plataformas complejas que soportan el desarrollo de las llamadas Aplicaciones Intensivas en Datos (AID). Sin embargo, crear un AID todavía requiere un proceso de diseño complejo que, además de los problemas relacionados con garantizar un alto rendimiento y disponibilidad, también incluye la necesidad de proporcionar garantías en términos de privacidad de datos y permitir a los usuarios definir y actualizar políticas de privacidad basadas en sus propias preferencias.

En este sentido, StreamGen es un enfoque basado en modelos para soportar el diseño de AID y la generación automática de código para dos plataformas de destino, Flink y Spark. Además, StreamGen está diseñado con el objetivo de que sea fácil de manejar por usuarios con pocos conocimientos sobre AID.

El propósito general de la presente propuesta de Trabajo de Final de Máster (TFM) es extender StreamGen para que tenga en cuenta las políticas de privacidad. Para alcanzar tal propósito: a) se va a requirir la definición de un idioma, como parte de un perfil UML, para permitir a los usuarios definir para un AID dos tipos de políticas de privacidad: políticas de visión del creador (View Creation Policies, VCP) y políticas de eviccón del sujeto del dato (Data Subject Eviction Policies, DSEP). Las VCPs modifican los datos teniendo en cuenta quien es el propietario del dato y algunas condiciones predefinidas que se deben satisfacer sobre el mismo. Las DSEPs extraen los datos de un flujo dado cuando el dato es propiedad de un cierto sujeto y, además, satisface unas condiciones predefinidas. Dicho lenguaje se define teniendo en cuenta el modelo de flujo de datos de las AID y su enfoque en fuentes, transformaciones y sumideros, lo que hace que la definición sea lo suficientemente simple como para ser manejada por usuarios que no están familiarizados con las AID. b) A partir del lenguaje definido y por medio de Acceleo, se desarrollá una implementación de una transformación de modelo a código que permitirá a los usuarios automatizar la generación del código AID con privacidad mejorada, apuntando a Flink como plataforma final. c) Se va a realizar la evaluación del enfoque mediante la explotación de dos casos de estudio. En el primer caso, se desarrollará una AID para preservar la privacidad de los usuarios que realizan diferentes transacciones sobre un stock fijo de una tienda. En segundo lugar, se modificará una AID que toma las temperaturas de dos habitaciones y que calcula un estudio estadístico de las temperaturas máximas y medias, y la predicción de la temperatura de las habitaciones para un tiempo dado y se añadirán políticas de privacidad para que no se calcule la predicción de la temperatura de la segunda habitación.

Sommario

La gestione dei Big Data appare come un nuovo problema per la nostra società che mira a estrarre informazioni utili da essi.

A tal fine, sono state concepite piattaforme complesse che supportano lo sviluppo delle cosiddette applicazioni ad alta intensità di dati (DIA). La creazione di una DIA, tuttavia, richiede ancora un processo di progettazione complesso che, oltre ai problemi relativi a garantire prestazioni e disponibilità elevate, include anche la necessità di fornire garanzie in termini di privacy dei dati e di consentire agli utenti di definire e aggiornare le politiche sulla privacy in base a le proprie preferenze.

Pertanto, StreamGen è un approccio basato su modelli per supportare la progettazione di DIA e la generazione di codice automatizzata per due piattaforme target, Flink e Spark. Inoltre, StreamGen è progettato con l'obiettivo di essere facile da gestire da parte di utenti con poche conoscenze su DIA.

Lo scopo di questa tesi è estendere StreamGen per renderlo consapevole delle politiche sulla privacy. Per raggiungere tale scopo: a) la definizione di una lingua, come parte di un profilo UML, sarà richiesta per consentire agli utenti di definire per una DIA due tipi di politiche sulla privacy: Visualizza politiche di creazione (VCP) e politiche di sfratto degli interessati (DSEP). I VCP modificano i dati in entrata tenendo conto dell'interessato e di alcune condizioni predefinite che devono essere specificate. I DSEP rimuovono i dati dal flusso quando provengono da una persona interessata e soddisfano alcune condizioni predefinite. Tale linguaggio viene definito tenendo conto del modello di flusso di dati delle DIA e delle sue fonti, trasformazioni e approccio dei pozzi, rendendo la definizione sufficientemente semplice per essere gestita da utenti che non hanno familiarità con le DIA. b) dal linguaggio definito e tramite Acceleo, verrà sviluppata un'implementazione di una trasformazione da modello a codice che consente agli utenti di automatizzare la generazione del codice DIA ottimizzato per la privacy, indirizzando Flink come piattaforma finale. c) la valutazione dell'approccio sarà effettuata sfruttando due casi studio. Nel primo caso, verrà sviluppata una DIA al fine di preservare la privacy degli utenti che effettuano transazioni diverse tra un magazzino fisso. In secondo luogo, verrà modificata una DIA. Questa DIA prende le temperature da due stanze e calcola due statistiche, la temperatura massima e quella media, e la previsione della temperatura delle stanze in un dato tempo. Questa applicazione verrà modificata per evitare il calcolo della previsione quando le temperature provengono dalla seconda stanza.

Acknowledgements

I want to thank...

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This is a test.

1.1 Section 1

This a test for the first section.

1.2 Section 2

And this is a test for the second section.

Chapter 2

State Of The Art

Big Data has transformed our society in the last years. The capability to develop efficient applications which were able to handle large amount of data at run time was the first step. In section Data-Intensive Applications, first of all, a classification of these kind of application is made. After that, the characteristics of data-intensive applications are explained. Finally, in this section some frameworks as Hadoop, Spark or Flink are explained giving some characteristics and what makes them different. In section Privacy Policies in IT some concepts relative to computer security and how such concepts impact on data privacy are developed. Some different approaches to generate private data are exposed and, after that, it is emphasized how privacy policies can be applied in the already known data-intensive applications. Finally, in section Modeling & Metamodeling, first of all the main concepts of metamodeling are exposed. After that the Unified Model Language is explained and, at the end of the section, a model-driven approach to support data-intensive application design called StreamGen is introduced.

2.1 Data-Intensive Applications

Data-Intensive Applications (DIAs), according to [1], are a class of parallel computing applications which handle high amount of datasets and that devote the most of their processing time to I/O and manipulate such datasets. These kind of applications are able to manage datasets of several terabytes and petabytes which are available in a wide variety of formats and that are distributed among several locations. This capability to process large volumes of data is possible due to the fact that DIAs use multi-step analytical pipelines with transformations and fusion stages in order to parallelize the execution of the data.

Intensive applications can be classified ([2], [3] and [4]) according to the parallel processing approaches commonly known as compute-intensive and data-intensive. On the one hand, compute-intensive is used to describe application programs that are compute bound. Such applications devote most of their execution time to computational requirements and they process small volumes of data. In this kind of applications, processes must be parallelized by means of individual algorithms and then they must be decomposed into separate tasks. Later, such tasks are executed in a pipeline reaching higher performance that in serial processing. Finally, compute-intensive applications are able to perform multiple operations simultaneously as they allow task parallelism.

On the other hand, data-intensive is used to describe application programs that are I/O bound ([5]). Such applications devote most of their execution time to I/O, move and manipulate the datasets. This second parallel processing approach, unlike compute-intensive, processes large volumes of data. Data-intensive applications require the partition or subdivision of the initial dataset in some smaller datasets due to the fact that the large volume of data that they are composed. These partitions must be processed independently by the application allowing the parallelization of the initial dataset. Once the partitions are executed, they must be reassembled in an output dataset.

As any type of application which takes advantage of data-parallelism, DIAs need to select an algorithm not only to distribute data among the processing nodes of the cluster but also to execute the data in such processing nodes ([6]). Moreover, they need to adopt a strategy to decompose the datasets, find a trade-off in terms of working loads in the processing nodes by means of load balancing systems and to communicate the nodes of the cluster.

2.1.1 Characteristics

Data-intensive applications are modeled in a different way than any other computing application due to the property of DIAs to be processed in a cluster infrastructure. Moreover, as any other system, DIAs need to reach a high performance in order to make efficient applications; this is possible by means of the minimization of the number of movements of the data. Other important properties required by DIAs are that they must be available when they are called, and that they must be reliable to the number of failures; both properties are accomplished due to data-intensive applications are fault resilient. Last but not least, data-intensive applications must be able to change the number of required nodes taking into account the workload. Then, four characteristics, according to [7] and [8], can be defined in order to distinguish data-intensive applications from any other kind of computing system:

Dataflow Model

Data-intensive applications are modeled by means of nodes where some transformations take place and data or programs flow between such nodes. Moreover, such model allows to control how to schedule, to execute, to balance the load and to communicate programs and data during runtime using a cluster computing infrastructure.

Data Movement Minimization

Minimize the number of movements required by each data along the different computations in order to achieve high performance is crucial in data-intensive applications. Then, processing algorithms are executed on the nodes of the dataflow model where the data is located, reducing the number of movements and therefore increasing performance.

Fault Resilient

Due to the fact that DIAs work with high amounts of data, the probability for the hardware to fail is greater than in other computing systems but the same happens with the probability of communications errors and software bugs. This is why, DIAs are designed to be fault resilient. In order to make these applications available and reliable, on the storage disk of such systems redundant copies of all data files or intermediate processing results can be found. But also they are provided with systems capable to detect the failure of any node or processing failures and, once such failures are detected, then re-compute the results.

Node Variability

Depending on the hardware and software architecture of the data-intensive application, the required number of nodes and processing tasks and its variability along the runtime can be fix or variable. This is due to the fact that it is very important to achieve a comfortable processing independently of the amount of data and, also, to reduce the critical time of computation by adding some nodes.

2.1.2 Frameworks

2.1.2.1 Apache Hadoop

Apache Hadoop is an open source software framework that supports distributed applications. As any other DIA, Hadoop allows to work with thousands of nodes and petabytes of data. This framework is composed by a processing layer called MapReduce which was inspired by Google documentation.

MapReduce works as a DIA processing large volumes of data. First of all, this programming model divides the incoming works into a set of independent tasks and, after that, such tasks are executed. Then, the user sends the complete job to a master which divides it into independent tasks and submits them to the slaves in order to process the works in parallel. This process allows Hadoop to reach speed and reliability of cluster.

In summary, Hadoop MapReduce works breaking the incoming datasets into independent sets that are executed in parallel by means of two phases ([9]): map phase and reduce phase. Firstly, in the map phase, the business logic is specified by putting the custom code in the way MapReduce works. And, secondly, in the reduce phase, operations as summations and aggregations are processed.

2.1.2.2 Spark

Spark is an open source platform that allows general-purpose and fast cluster computing. It is an engine that allows to process large volumes of data.

Spark enables users to access repeatedly to datasets in order to perform streaming, machine learning or SQL workloads by means of high-level APIs in Java, Scala, Python and R. It also allows users to accomplish batch processing or stream processing. Stream processing can be defined, according to [11], as to compute on data just when it is received or produced.

Moreover, Spark can be integrated with any other data-intensive tool due to its design. Spark is compatible with Hadoop as it is able to access Hadoop data

sources and run on Hadoop clusters. This platform extends MapReduce, the engine of Hadoop, by including iterative queries and stream processing.

Some features make Spark more efficient than Hadoop. The following features, that are presented in [10], make Spark being the 3G of Big Data:

- Fault tolerance.
- Dynamic in nature.
- Reusability.
- Advanced analytics.
- Lazy evaluation.
- Real-time stream processing.
- In-memory computing.

The in memory computing feature reduces the number of read-write to disk operations what achieves a data processing performance 100 times faster in memory and 10 times faster on the disk. Due to the high amount of operators that Spark provides, to achieve parallel applications is easier with Spark. Regarding to fault tolerance, Sparks handles the failure of the nodes in the cluster by means of Resilient Distributed Datasets (RDDs) which its specific design makes to reduce the number of failures to zero.

In conclusion, Spark can be differentiated from Hadoop in its cluster management system. On the other hand, regarding to the similarities between Spark and Hadoop, Spark takes advantage of Hadoop for storage.

2.1.2.3 Apache Flink

Apache Flink is an open source platform which allows data streaming computation because of its data flow engine. Moreover, as happens with Spark, Apache Flink is able to execute stream processing and batch processing and it is also compatible with Hadoop.

Apache Flink can undertake efficiently some different types of processing, becoming in the most potent open source platform in the market to handle them. Such processing types, according to [12], are:

- Batch Processing.
- Interactive processing.
- Real-time stream processing.
- Graph Processing.
- Iterative Processing.
- In-memory processing.

Apache Flink reduces the complexity that other platforms as Spark faced by means of some improvements. Such improvements include the integration of query optimization in MapReduce processing layer, some concepts from database systems and efficient parallel in-memory and out-of-core algorithms. These optimizations are given due to the fact that Flink architecture is designed taking into account the streaming model. Such design improves the micro-batch approach taken in Spark for data streaming processing making Flink faster and with lower latency for such kind of processing.

Following the example of Spark, Apache Flink is provided with three APIs (DataStream, DataSet and Table APIs) in order to use its engine. DataStream and DataSet APIs are two regular programs used by Flink in order to perform transformations on data streams (filtering, aggregating, update state...) and data sets (joining, grouping, mapping...) respectively. On the other hand Table API is used for relational operations and it can be integrated into DataStream API for relational stream processing and into DataSet API for relational batch processing.

Finally, it is important to remark the five features that can be found in [12] about Apacha Flink:

1. Low latency and high performance.
2. Fault tolerance.
3. Memory management.
4. Iterations.
5. Integration.

All these characteristics are what make Flink the 4G of big data and the most powerful tool when dealing with data stream processing.

2.2 Privacy Policies in IT

The appearance of Big Data and DIA platforms dealing with large amount of datasets has led the society to question about the privacy of those data that are processed. Due to the fact that such data are not visible for the data producers but these data are bought and handled by many companies in order to compute some statistics, data privacy has become in a must for the development and commercialization of new DIA in order to give the option to the user to decide on which data can be seen by other agents.

On the other hand, large amount of data flowing through the net and the existence of black hat hackers forces DIA designers to take into account some concepts about computer security ([13]). Computer security, also known as cybersecurity or information technology security (IT security), is the protection of computer systems and networks from people who is interested in stealing or damaging their hardware, software or the data with which they are working. It is at this last point where Big Data applications have to pay attention in order to be designed.

Every computer security system accomplish the well-known CIA paradigm, as it is explained in [13]. "C" stands for Confidentiality, "I" for Integrity and "A" for

Availability and they are the three basic requirements that compose such paradigm. Any computer security system has to grant that any piece of data has to be accessed only by those who are authorized, accomplishing such requirement, information is confidential. Moreover, such systems have to be able to allow data modifications only to those how are authorized for it and only in the way that they are entitled to modify them, this requirement makes information to be integrated. Finally, data must be available to everyone who has a right over them within some time constraints, this makes information to be available. These three requirements raise an engineering problem and it is that availability conflicts with confidentiality and integrity due to the fact that if a data is available then it could not be confidential as anybody may modify it and it could not be integrated as anybody could modify then. This is the main problem that has to be faced when designing computer security problems. In order to handle this complex engineering problem, designers can use legal instruments but also technical tools.

Due to the fact that the information of many entities (persons, companies, etc.) are involve in such complex problem, there are some legal instruments to handle the design of security systems. This is the case of privacy policies. Privacy policies are statements or legal documents that reveal how an entity can collect, use or handle data from another entity. Privacy policies can differ from one country to another this is why there are some agreements in order to handle them. In the European Union (EU) the General Data Protection Regulation (GDPR), according to its web page [14], is in charge of harmonizing data privacy laws across all member states of the EU. In spite of these regulations, there are many ways in order to apply privacy policies.

Due to the importance of such privacy policies, some researchers haven been studying about the development of formal languages in order to specify them in a logical way. This is the case of some literature [15] where XACML is used to specify privacy policies by means of some logic rules in order to reach a set of policies. This formal language (XACML) also can be used for other reasons as it is the case of user authentication in distributed systems [16].

On the other hand, when a computer security system is designed, the designer has to put himself in the place of the cyberthief, or threat agent, in order to achieve a secure system. Such cyberthieves try to find vulnerabilities and, then, exploit them in order to steal some information. Usually, vulnerabilities are a bug that allows to violate the CIA paradigm but the exploit can be a wide variety of vulnerability uses. Despite this design technique, unfortunately, a great amount of vulnerabilities are found when users accidentally deal with the applications once the applications are in the market.

Another remarkable point is the importance of threats, that are potential violations of the CIA paradigm in computer security systems. This means that an information system can be seen its security system broken due to the fact that a circumstance or an event could impact on it adversely by means of the break of any of the three basic requirements of a computer security system. A threat is composed by three layers. The attacker or threat agent that is the person or the information system that performs the attack action, the countermeasure layer that is all the systems that try to identify an attack and to stop it and, finally, the target of the attack that is on which resides the vulnerability and on the threat consequences take place when the exploit arrives to the vulnerability. By means of such vulnerabili-

ties, private data can be stolen by cyberthieves. This is why, system dealing with confidential data must hash them as soon as possible in order to handle private data even during the execution phase of the pipeline.

In conclusion, the concept of computer security involves many fields but when dealing with the design of DIAs, it is very important to take into account not only the CIA paradigm but also the design techniques from the threat agent perspective in order to reach secure DIAs, always under the legal framework of privacy policies. Furthermore, what makes security on DIAs reliable is the balance on its conflicting requirements what is a complex engineering problem. Finally, systems with a high amount of vulnerabilities but with no threats, are secure systems. However, systems with many threats and only a few vulnerabilities become in the most attacked systems and, then, the ones that need more security measures.

2.2.1 Privacy Policies in DIAs

Due to the fact that DIAs are constantly managing data from different entities, it is very important the way in which privacy policies are applied on them. The final goal of applying them is that, in case of threats, threat agents cannot obtain such data. This reduces the probability of threats as there is no sensitive data that can be obtained and, then, the CIA paradigm cannot be broken.

When dealing with DIAs, sensitive data are sent in data streams. Such data streams are composed of tuples which can contain a high variety of data. Each tuple of a data stream contains a finite set of data that are always ordered in the same way. Due to the fact that all the tuples contained in a data stream have the same structure, when speaking about data streams, each of the data of the tuple is considered as a field of the data stream. Due to this distinction on the fields of the data streams, some literature [17] classifies data streams into three different sets:

- Subject-specific streams.
- Subject-generic streams.
- Non-personal streams.

A subject-specific stream is a stream composed of a field referring to the data subject. This means that each tuple of the stream contains a data which is the owner of the tuple. A subject-generic stream is a stream without data subject. However, subject-generic streams are produced by an operator whose input is a subject-specific stream. Finally, a non-personal stream is a stream without data subject and that it has not been produced by an operator whose input is a subject-specific stream.

The context in which a tuple is conveyed is also important. Depending of the context in which data are transferred, an entity could want that the data of the tuple are private or public. This context is specified by means of three variables in [17]:

- Observer.
- Role.
- Purpose.

The first variable is the observer of the data, the entity who demands the usability of the data. The second variable is the role that the requesting entity is playing. The requesting entity can be a person but also a company. In the case of the company, many persons working for such company can be demanding the data of the tuple and each person has a role inside the company, this is why the role is an important variable in order to define the context in which a privacy policy can be applied. Finally, the third variable that defines the context is the purpose. The purpose for which the requesting entity demands the data of the tuple. Then, the context in which a privacy policy has to be applied is defined by means of three variables that are called static context variables (SCV) as, once they are defined, they are not going to change and because of this they are static.

Another important distinction is the one given by the data that compose a tuple. Each of these data can be strings or numbers. Because of this, a classification on the type of each data of the tuple, and then on the type of the fields of the streams, has to be defined. Moreover, following the approach given by the SCVs, each of these values are considered as a variable as the values of a field of a given stream can change depending on the tuple that is referred. Then, the variables that compose a field of a stream, according to [17], are:

- Categorical variables.
- Numerical variables.

Categorical variables are those that correspond to a string in any programming language. They can be words but also links or any set of letters with special characters and numbers. The second type of variables correspond to numbers but also to a range which corresponds to a pair of numbers. For example, a data subject can be a categorical variable when it is the name of the user who uses the DIA but a data subject can be also a numerical variable if it is an ID which is compose only by numbers. Furthermore, the set of categorical and numerical variables used by a DIA are called contextual variables.

Taking into account these three classifications, privacy policies can be defined by means of some rules. These rules have to be defined making a relationship between the contextual variable used by the DIA and some values that the user selects in order to encrypt the original fields of the stream. These relationships are made by means of logical operators. In the case of categorical variables, the operators that can be used are: equal to ($=$) and not equal to ($\neq$). On the other hand, in the case of numerical variables, the relation operators that can be used are: equal to ($=$), not equal to ($\neq$), is greater than ($>$), is less than ($<$), is greater than or equal to ($\geq$) and is less than or equal to ($\leq$)

Finally, due to the fact that privacy policies encourage the anonymity of the entity who owns the data of the tuple, depending on the data stream classification explained above and the rules defined by each user, privacy policies can differ. Then, another classification is made in [17], in this case about the privacy policies in DIAs:

- View Creation Policies (VCPs).
- Data Subject Eviction Policies (DSEPs).

2.2.1.1 View Creation Policies (VCPs)

View Creation Policies (VCPs) are a type of privacy policies for DIAs which are applied to subject-specific streams. VCPs are applied when the context is precisely specified by means of the SCVs and the information contained in all the fields of the stream is observable, which means that it has not been modified before by another privacy policy.

Furthermore, VCPs are defined by means of a set of rules which are relationships between the contextual variables used by the DIA and some values that the user selects in order to encrypt the original fields of the stream. In this kind of privacy policies, the encrypted final values of the stream are defined by means of generalization vectors. A generalization vector defines the value to be used in order to generalize each of the fields that compose a subject-specific stream.

In conclusion, given a tuple belonging to a subject-specific stream, if at any instant such tuple satisfies all the rules defined by means of logical operators relating some contextual variables with a value defined by the user of the DIA, a generalization vector is applied on the tuple in order to generalize each of the fields of the streams and obtaining an encrypted tuple. Then in [17], a VCP is defined by means of four items:

1. Subject-specific stream.
2. Data subject.
3. Set of relational rules.
4. Generalization vector.

2.2.1.2 Data Subject Eviction Policies (DSEPs)

Data Subject Eviction Policies (DSEPs) are applied when given an operator, which input is a subject-specific stream, produces a subject-generic stream. Sometimes, the entity who owns the data which are flowing along the DIA does not want that an operator takes some information from the input stream in spite of the output stream is not showing any information directly pointing to such owner. This is the case in which DSEPs are applied.

Unlike what happens with VCPs, DSEPs do not encrypt the fields of a given data stream. In this case, DSEPs prevent the tuples flowing in the subject-specific stream that inputs into a given operator from being processed by such operator. In this case, the output from the operator is computed without taking into account the data of the data subject who defines the DSEP, preserving the anonymity of the owner of the data for the computation of a certain statistic.

In conclusion, in order to define a DSEP, first of all, a set of rules have to be specified by the user relating some contextual variables with some values that such user has to define. Then, given an instant when all the defined rules are satisfied on a subject-generic stream which outputs from a given operator, all the tuples referring to the data subject who defined such rules have to be evicted from the subject-specific stream that inputs into the given operator. Then in [17], a DSEP is defined by means of three items:

1. Subject-generic stream.

2. Data subject.

3. Set of relational rules.

2.3 Modeling & Metamodeling

Data Intensive Applications are used when large amount of data have to be handle with high performance. In order to reach such high performance, DIAs require complex designs and a lot of resources to achieve their goal. One of these resources is the number of software developers who are needed to write the application codes. On the other hand, when some specific requirements are added to the DIAs, as it is the case of privacy policies managing the privacy of the users, developing DIAs becomes in a really difficult task which cannot be faced writing codes from scratch. At this point is where other approaches to write application codes take advantage. Such is the case of modeling and metamodeling techniques applied to software development.

A model is a representation of a reality in order to explore, to redesign or to transform it. This representation can be externalized by means of a diagram in a paper or in a computer software. Modeling is the method that has to be developed in order to reach a model. A modeling method consists of a modeling technique and a mechanism or algorithm working on the model. Moreover, a modeling techniques consists of a modeling language and a modeling procedure. A modeling procedure is a sequential process composed of several steps but also an iterative process since the procedure can be repeated many times until the final model is reached. In spite of several modeling procedures can be accomplish in order to create a model, as it can be found in [18], all these approaches share some common steps:

- Purpose definition.
- Boundaries definition.
- Model elements definition.
- Relationships definition.

The first step that has to be accomplished in order to create a model is to define its purpose. With such purpose, the questions that the model will have to answer, once it is implemented, should be specified. Later, the boundaries of the model must be specified. Such boundaries must focus on the reality that has to be modeled discarding any information that is not able to reach the predefined purpose. Once these boundaries are defined, everything that is going to take part of the model is called the model domain. Due to the fact that usually this domain is still very big, each concept belonging to the model domain must be filtered taking into account the relevance of each of them and grouping those concepts that are identical in order to reach the purpose. After that, each of the groups with identical concepts can be transformed into a model element, which is an abstraction of all the concepts that are part of the group. Finally, the relevant relationships among the model elements have to be identified and represented in the model.

Furthermore, a model must be complete in order to be able to answer the question specified in the purpose but also, it must be consistent which entails the lack of

contradictions in its representation. In order to reach such characteristics, modeling methods take advantage of modeling techniques in addition to a mechanism, an algorithm, working on the model.

The first component of a modeling technique that is a modeling language is a set of well-known elements and relationships by means of which a model can be represented. A modeling languages is described by a syntax, semantics and a formal notation. A syntax is the set of elements, relationships and rules that can be written in order to represent a model by means of a grammar. Moreover, a syntax can be described by two different types of grammars: graph grammars and metamodels. According to [20], a metamodel is a model of a modeling language. A metamodel is written by means of a modeling language that is known as the metamodeling language. The metamodeling language has a model which defines it. This is why exists a modeling hierarchy which is not limited and that ends given a useful abstraction level of the model. In the table 2.1 there is an example of a modeling hierarchy with five levels.

Language Level	Models	Language Name
Level 1	Model	Modeling Language
Level 2	Metamodel	Metamodeling Language
Level 3	$Meta^2$-Model	$Meta^2$-Modeling Language
Level 4	$Meta^3$-Model	$Meta^3$-Modeling Language
Level 5	$Meta^4$-Model	$Meta^4$-Modeling Language

Table 2.1: Modeling Hierarchy

The semantics of a modeling language describe the meaning of the language by means of a semantic domain and a semantic mapping.

Finally, the formal notation describes how the modeling language is visualized. Some symbols are used in order to represent the represent the syntax of the language in a diagram.

On the other hand, the finality of the mechanism is to check that the model written with the modeling language by means of the modeling method is correct and satisfies the purpose of the model. There are three main types of mechanisms and algorithms: generic, specific and hybrid.

In the figure 2.1 o model of a modeling method is shown.

2.3.1 Unified Modeling Language (UML)

Unified Modeling Language (UML) is a standardized modeling language whose objective is to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes.

UML is able to achieve such goal by means of a wide variety of options to build modeling diagrams. Among the fourteen possibilities that UML provides to construct a diagram, a distinction between two main classes is made in [19]: behavior diagrams and structural diagrams.

Behavior diagrams represent the dynamics of a system, this means the representation of all the possible changes that a system can experience over time. This class

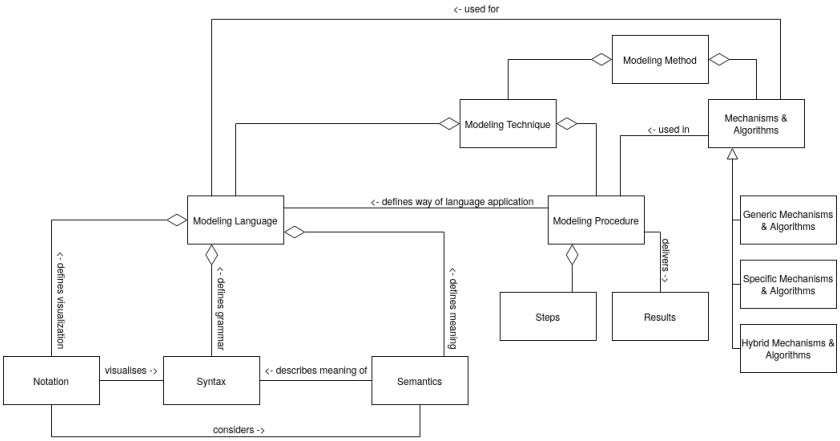

Figure 2.1: Modeling Method Components

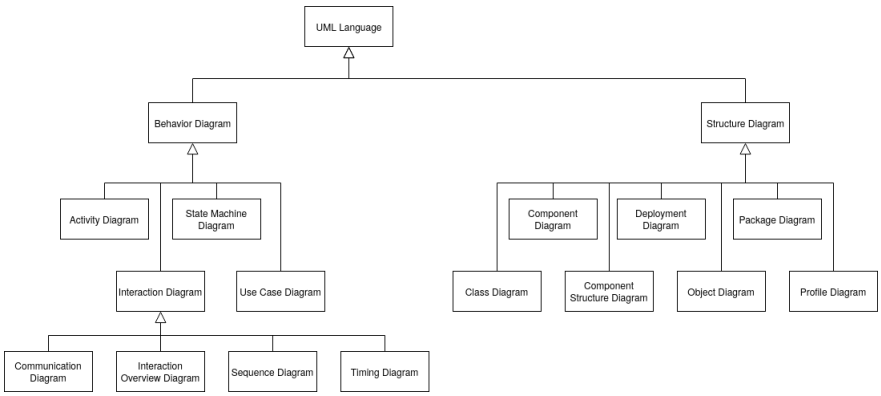

Figure 2.2: UML Metamodel

is composed of seven diagrams: activity diagram, communication diagram, interaction overview diagram, sequence diagram, state machine diagram, timing diagram and use case diagram.

On the other hand, structural diagrams represent the different static parts of the system, that ones which do not change over time. This representation can be done from different abstraction and implementation levels and making the corresponding relationships between the parts of the system. This second class is composed of other seven possibilities for diagram construction: class diagram, component diagram, component structure diagram, deployment diagram, object diagram, package diagram and profile diagram.

A UML diagram making a description of all the UML diagrams can be seen in the figure 2.2.

2.3.1.1 UML Diagrams

In spite of UML allows developers to use fourteen diagrams for modeling, some of such diagrams are the most commonly used. According to [19], this is the case of the following ones:

Class Diagram

This kind of diagram represents the model elements and the relationships that exist between such elements. The model elements, which are represented by means of nodes, can be: class, component, data type, interface, model or package among others. Regarding to the relationships between the nodes, they can be: association, association class, dependency, generalization or information flow among others.

Class diagrams are commonly used to represent the metamodel of the syntax of a modeling languages. Due to the fact that this kind of diagrams is not able to fully express syntactical rules, usually they take advantage of constraint languages as Object Constraint Language (OCL).

Profile Diagram

A profile diagram allows to create the model domain by adding model elements called stereotypes and making the corresponding relationships between them. The most commonly used relationships in this kind of diagrams are the composition and the generalization.

2.3.1.2 UML Frameworks

Due to the UML popularity among software developers, there are several frameworks working with it in order to make software development easier. Different platforms can be used for UML modeling but also platforms for converting a model to a source code can be found.

Papyrus

UML can be found in a wide amalgam of platforms due to its utility for modeling. Moreover, there are some platforms that are the most commonly used for software development. This is the case of Eclipse. Eclipse is the most popular Java Integrated Development Environment which allows a large amount of plugins in order to increase its functionalities.

One of the plugins provided by Eclipse is Papyrus, an open-source UML tool. In spite of Papyrus was developed by the French Alternative Energies and Atomic Energy Commission (CEA-List), currently it is one of the most used UML tools for software development as it can be expanded with UML profile diagrams giving a wide functionality for modeling software applications.

Acceleo

Regarding to platforms for converting an UML model into source code, Acceleo is an open-source generator from Eclipse Foundation. Acceleo is one of the Eclipse plugins that allows to transform UML models into source code files. It is written in Java and it is available for Linux, Windows and Mac OS.

Acceleo uses its own language in order to generate the source codes. Such language is based on MOF Model to Text Transformation Language (MOFM2T) and on template focusing. A template contains some text which describes the data that should be extracted from the elements of the model and such data are extracted iteratively from each component of the model. Moreover, the descriptions of the

information that should be taken from the model elements are written in OCL language.

Finally, it is import to remark that Acceleo can extract the information from different types of models, such as EMF, UML or DSL and it is able to generate different source languages such as C, Java or Python.

2.3.2 Object Constraint Language (OCL)

Object Constraint Language (OCL) is a language to describe formal expressions in UML models. These expressions represent invariants, preconditions, postconditions, initializations, guards, derivation rules, as well as queries to objects in order to determine its state conditions.

Initially, OCL was developed by IBM. But, then, in 2003, OCL was adopted as part of UML 2.0 by the group OMG.

This language does not cause side effects, so the verification of a condition, which presupposes an instantaneous operation, never alters the objects of the model. Its main role is to complete the different artifacts of the UML notation with formally expressed requirements.

In conclusion, OCL associates boolean expressions with the elements of the model and these expressions are evaluated as true every time the program is executed. In addition, they can also be associated with methods. In this case, the Boolean expressions are evaluated as true at the moment before or after the execution of the program.

2.3.3 StreamGen

StreamGen is a model-driven approach to support the design of DIAs and the automated code generation for two target platforms, Flink and Spark. This approach is developed focusing on two requirements:

- To provide a platform-independent and graphical modeling language for streaming applications.
- To enable the fast prototyping of distributed streaming applications over different platforms.

The first requirement is needed in order to develop any kind of DIA for different target platforms and, independently for which platform it is targeted, provide a complete language for all of them. This is possible due to the DIA dataflow model characteristic that supports the development of them by means of sources, transformations and sinks. The second requirement is satisfied by using UML class diagrams for such purpose. This kind of diagrams allow to represent sources, transformations and sinks but also the data flows really fast, just introducing the predefined stereotypes.

2.3.3.1 DIA Generation Process

The different steps that have to be followed in order to generate a DIA with StreamGen are the following:

- UML model creation.
- Application code generation.
- Application execution.

UML model creation

First of all, the user must create a UML model by exploiting a predefined language in a UML profile diagram. This language consists of four metaclasses:

- Class
- InformationFlow
- Model
- Package

The class metaclass represents the streaming operators stereotypes which are data sources, data sinks and data stream transformation. Moreover, this three stereotypes can be represented by means of several generalization. In the table 2.2 can be shown such stereotypes.

Sources	Transformations	Sinks
Text File	Reduce	Text File
Socket	Map	CSV File
Collection	Window	Socket
Kafka	Flatmap	Cassandra
-	NFlatmap	Kafka
-	NMap	-
-	Count	-
-	Fold	-
-	Sum	-
-	Filter	-
-	Join	-
-	CoGroup	-

Table 2.2: StreamGen Classes

The InformationFlow metaclass represents the data streams. There are three main data streams stereotypes: non-parallel stream, windowed stream and partitioned stream. Moreover, the partitioned stream stereotype can be generalized to broadcasted stream, keyed stream, randomly partitioned stream and round robin stream.

The model metaclass is extended by a distributed streaming application stereotype which is generalized to Flink application, Spark application and Apex application.

The last metaclass, the package, is extended by the stream data types stereotype which will contain all the data types conveyed by the different information flows.

Application code generation

Once the user has defined the UML model, the java application configuration must be run in order to generate the application codes. This is possible because of the development of an Acceleo code. This Acceleo program is able to take as inputs the generated UML model and, depending on the different stereotypes contained in such model, generate a DIA code for Flink or Spark.

Application execution

Currently, StreamGen allows code generation for Spark and Flink. As can be seen in its model stereotypes, also there is an intention to develop Apex codes in the future.

2.4 Discussion

The appearance of Big Data has generated a growth in IT research. Firstly, with the irruption of Hadoop framework, the firsts DIAs were developed. Since then, more development frameworks, as Spark or Flink, have been improving Hadoop features until arriving to the targeted stream processing where data is computed just when it is received or produced.

Due to the fact that currently data-intensive applications provide speed besides performance, usually we can find them in our day to day and, with the generalization of DIAs in our lives, two new challenges in IT research appeared.

The first one is relative to the privacy of the large amount of data that DIAs are handling. Since DIAs are used in several fields and some of these fields, as it is the case of the bank industry, work with sensitive data, data must be modified in order to keep the privacy of the data owner what means to preserve the data privacy. Some regulations in this regard have been legislated in the last years but they are not enough. In addition to the legislation some technical approaches in order to apply the regulations have been studying. This is the case of privacy policies as View Creation Policies and Data Subject Eviction Policies.

Regarding to the second challenge, coding data-intensive applications from scratch is no longer efficient. At this point is where techniques as metamodeling appear to have an important role in the new software development scenario. The well-known UML language used by software developers to avoid programming from scratch can be used for such purpose. Moreover, there are already some approaches in this regard. This is the case of StreamGen, a model-driven approach to support the design of DIAs and the automated code generation for two target platforms, Flink and Spark. However, StreamGen does not provide a way to automate code generation for privacy-aware DIAs.

Chapter 3

Requirements and Design

In the last years the interest among data-intensive technologies and applications in order to extract useful information from data has increased in our day to day. These data-intensive applications and their execution environments allow users not only to handle Big Data but also to do it efficiently. This is why Big Data has become in a common technology in our lives.

Furthermore, the appearance of Big Data has generated some needs in our society. One of these needs is privacy policies. DIAs require some tools which are able to preserve the privacy of users which are generating such data. These tools can vary from international laws that legislate in this regard to technical approaches that allow to encrypt the flowing data.

Due to the development of dataflow applications and the wide range of fields where Big Data is still getting involved, writing DIA codes from scratch became in an inefficient task. At this point was where modeling techniques for software development became important for DIA development. StreamGen is an example of a platform used to develop DIAs from UML class diagrams by means of Papyrus. However, StreamGen does not allow to specify privacy policies.

Some approaches can be found in the literature about privacy policies applied to dataflow applications as it is the case of [17]. This approach allows to specify two different types of privacy policies: View Creation Policies and Data Subject Eviction Policies. Each of these privacy policies are determined by means of a transformation in charge of each of the policies. Moreover, a past condition checker operator is considered in order to check not only the static context variables conditions specified by the user but also some conditions about the variables flowing through the streams of the DIA which could be given in the past. Moreover, this article introduces a source generating the viewer tuples (SCV tuples) to feed the privacy operators.

In [17] a dataflow application based on an e-commerce example is presented to explain how privacy policies could be applied to StreamGen. Such application consists of three operators to compute some real time statistics about the transactions generated by the consumers of the e-commerce company. The statistics are sent to external companies in order to provide useful information to both parts, the clients and the external company that buys the statistics.

In order to develop this, a source, three transformations and three sinks are connected by means of streams as can be seen in the figure 3.1. This figure represents the dataflow model of the application without any privacy policy.

Once the application is developed, in [17], an algorithm is presented in order to

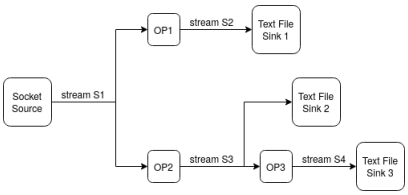

Figure 3.1: Great Seller Dataflow Model

generate the privacy-aware dataflow application. Such algorithm is able to generate two privacy policy types, VCPs and DSEPs. Furthermore, the article develops one privacy policy for each of the types which are applied to the e-commerce example:

- VCP: the client avoids that the employees of a market consultancy (MarketConsult employees) know if the client spent amount in the last 30 minutes is bigger than €100 and if the number of issued transactions by the client is bigger than 3, in such case, the client spent amount should be modified to the biggest allowed (€100).
- DSEP: the client does not allow that data goes to one of the operators (to operator 3) when MarketConsult observes the application.

In order to do this, some specific privacy policy operators are introduced. Such privacy policy operators are:

- StaticContextSource.
- PastConditionChecker.
- ViewBuilder.
- DataSubjectEvictor.

The proposed algorithm in [17] works with these operators as follows. First of all, the algorithm takes as input the application which is not privacy-aware. Moreover, when the algorithm is called, a StaticContextSource is directly added to the input application. Such source introduces the external companies which apply for the statistical information. After that, depending on the stream, if it is VCP-aware or DSEP-aware, a ViewBuilder operator (VCP-aware stream) or a DataSubjectEvictor operator (DSEP-aware) are added to the input application. Finally, if any past condition must be taken, a PastConditionChecker operator is applied over the corresponding stream. In the e-commerce example that is specified above, the stream S2 is VCP-aware and DSEP-aware then a ViewBuilder operator and a DataSubjectEvictor operator are applied on such streams and added to the non-privacy-aware application. Moreover, a past condition is applied for the stream S2 then a PastConditionChecker is applied on the stream S2 and added to the input application. In the figure 3.2 can be the seen the privacy-aware dataflow model of the e-commerce application after applying the algorithm presented in [17].

All the Java files developed to implement the privacy-aware e-commerce application are uploaded in two GitHub repositories (`https://github.com/MicheleGuerriero/dataflow-privacy-library/tree/master/src/main/java/it/deib/polimi/diaprivacy/`

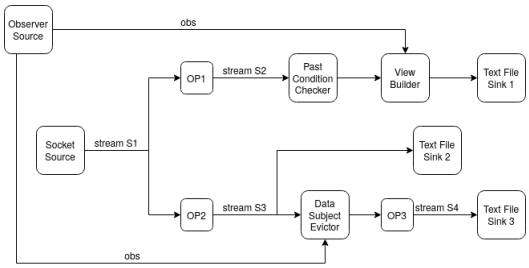

Figure 3.2: Great Seller Privacy Dataflow Model

`library` and `https://github.com/MicheleGuerriero/common/tree/master/src/main/java/it/deib/polimi/diaprivacy/model`). Such files are stored there with the goal to provide a library to future developers to apply VCP and DSEP privacy policies in dataflow applications.

As it can be seen, this approach requires that dataflow application developers have some knowledge about the privacy policy language but also about the targeted platform language (Hadoop, Spark, Flink...) which supposes a loss of speed to design DIAs. These inefficiencies are given due to the fact that, after implementing the application without privacy policies, the privacy policy operators are not developed as StreamGen language and neither the translation from the UML language to the target language. Then, there is no possibility to apply the presented algorithm in an easy way. Because of this, the current document faces the following objectives:

- To extend the StreamGen high level modeling approach to enable the modeling of privacy-aware streaming applications.
- To allow software developers with few knowledge about privacy policies to handle the design of privacy-aware streaming applications.
- From a non-privacy-aware application reach a privacy-aware application easily.

Thus, first of all, the StreamGen high level modeling approach, which is defined by means of a UML profile diagram, must be modified in order to add all the elements which are necessary to design privacy-aware streaming applications. Moreover, the reached profile must be elegantly integrated with the current profile. In order to achieve this goal, a new language that slightly differs from the language of the four operators presented in [17] is defined. The new language identifies which are the streams that must be protected by means of a stereotype that also has two properties to identify if the stream is VCP-protected or DSEP-protected. The extension of the language is made by means of the Eclipse plugin Papyrus. Further information about the versions used for Eclipse or Papyrus can be found in the table 3.1.

Secondly, the designer of the data-intensive application must be able to generate automatically all the Flink codes that are necessary taking into account that such designer could have few knowledge about privacy policies. In order to reach this goal, all the Acceleo codes to compile the previously defined language are written calling to the already developed library in [17]. This allows to have the same target Flink codes independently of the DIA and, then, the Flink privacy code modification by the developer is not required. For further information about the Acceleo version see the table 3.1.

Finally, the new language must be easily applicable over non-privacy-aware applications in order to reach Flink codes without errors. This goal is achieved by means of an intuitive language based on the datatype package already implemented in StreamGen.

In summary, in this document StreamGen language is extended in order to easily generate by DIA developers privacy-aware applications from non-privacy-aware dataflow application. In the table 3.2 can be found an abstract with all the added stereotypes and their metaclasses for the extension of the StreamGen language. Moreover, the Acceleo code to generate the privacy Flink codes based on the already existing privacy library presented in [17] are written. Such generated codes are defined in a way to be always the sames independently of the DIA.

Program	Version
Eclipse	2019-03 (4.11.0)
Papyrus	SysML 1.4 Feature 1.3.0
Acceleo	3.7.8.201902261618
m2e-Maven Integration for Eclipse	1.11.0.20190220-2119
Apache Flink	1.4.0

Table 3.1: Program Versions Used

Stereotype	Metaclass
PrivacyProtectingStream	Information Flow
PrivacyPolicyPackage	Package
PrivacyContextSource	Class
PrivacyPolicySource	Class

Table 3.2: Extended Language Abstract

Chapter 4

Solution

4.1 Privacy Policy Modeling Language

StreamGen uses its own language in order to develop DIAs. In this document, StreamGen language is expanded in order to allow users to protect the streams with privacy policies. This expansion consists of a new data stream stereotype in order to specify which streams are protected and with which privacy policy type (VCP or DSEP). Moreover, the sources that generate the Static Context Variables (SCV) and the privacy rules written by the users are also added to the StreamGen language. SCVs are possibly supplied by different sources whilst privacy policies will be supplied only by one data source type. This sources are going to be inserted in a package which represents the external sources of the DIA following the already existing approach of the StreamGen data types.

Thus, when a stream is protected, the PrivacyProtectingStream stereotype is added to the stream and some properties are filled in order to configure the type of protection. However, defining how the streams must be protected is not enough. Privacy policies require some external data, SCVs and privacy rules, in order to be applied. This two external data are introduced into the DIA by means of two sources, SCV source and privacy rules source. Regarding to the SCVs sources, four different types can be found in this approach. All these types are specified following the already existing approaches of data sources in StreamGen. On the other hand, for the privacy rules source, only one type is defined because the privacy rules file is supposed to be always the same. This two different sources, SCVs source and privacy rules source, are introduced in the application inside of a package stereotype called PrivacyPolicyPackage in order to make more intuitive the approach to the DIA developers.

4.1.1 PrivacyProtectingStream

StreamGen represents the flows of data flowing in DIAs by means of the UML metaclass Information Flow. These flows are known as streams and StreamGen allows to generate some different types as Keyed Stream or Windowed Stream. Each of these types have different characteristics and allow users to generate different behaviors. Moreover, these different streams can be applied all together generating a stream which can be, as in this case, windowed and keyed.

Following this approach, a new stereotype has been added to the UML profile

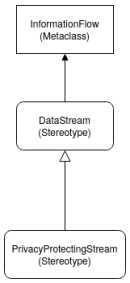

Figure 4.1: PrivacyProtectingStream UML Profile

of StreamGen. This stereotype has been called PrivacyProtectingStream and it is a generalization of the DataStream stereotype, which in turn is an extension of the InformationFlow metaclass. In the figure 4.1, the UML profile for the PrivacyProtectingStream is shown.

This new stereotype is going to have some properties in addition to the already existing property of the DataStream stereotype, isObservable. These properties are:

- ProtectedByVCP.
- ProtectedByDSEP.
- ProtectedStreamConf.

ProtectedByVCP and protectedByDSEP properties are added in order to specify how the data stream is protected; if it is protected with a DSEP, with a VCP or if it is protected with both privacy policy types. This is why these two properties are boolean values. Moreover, the configuration of the generated protected stream, which is a stream class imported from the privacy library developed in [17], has to be specified. In order to define such configuration, a new data type has been added to the UML profile. This data type is called ProtectedStreamConfiguration and it is composed of seven properties. In the tables 4.1 and 4.2 can be seen a summary of the properties of the PrivacyProtectingStream and the properties of the ProtectedStreamConfiguration data type.

Property Name	Property Type	Multiplicity
protectedByVCP	Boolean	1
protectedByDSEP	Boolean	1
protectedStreamConf	ProtectedStreamConfiguration	1

Table 4.1: PrivacyProtectingStream Properties

The PrivacyProtectingStream allows to easily specify which are the streams that must be protected and with which privacy policy types. This approach does not require any knowledge about the privacy policy types. Moreover, the ProtectedStreamConfiguration can be filled always with the same values, then with an example using such configuration, all the other DIAs could be defined with the same values. The unique value that can vary is the logDir property which is a path and it is not difficult to define a path by a DIA developer. Finally, this approach allows to define the non-privacy-aware dataflow application and, from that application, to

Property Name	Property Type	Multiplicity
monitoringActive	Boolean	1
timestampServerIp	String	1
timeStampServerPort	Integer	1
topologyParallelism	Integer	1
simulateRealisticScenario	Boolean	1
allowedLateness	Integer	1
logDir	String	1

Table 4.2: ProtectedStreamConfiguration Data Type

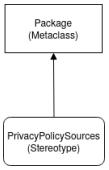

Figure 4.2: PrivacyPolicyPackage UML Profile

add the corresponding privacy configurations over the streams. However, there is no way to specify the SCVs and from where the privacy rules must be read. In order to specify such sources, the PrivacyPolicyPackage is developed.

4.1.2 PrivacyPolicyPackage

StreamGen uses a package stereotype called StreamDatatypes in order to collect all the data types that are flowing through the streams. The data types that are specified inside the package allow users to specify the different values that are conveyed in a stream but inside the same tuple. Following these approach, a new package called PrivacyPolicyPackage is added to the UML profile in order to collect all the privacy sources involved in the privacy-aware dataflow applications, SCV sources and privacy rules source.

These sources are not directly connected to the streams or to the transformations of the DIA but they allow users to specify from where the different external variables required to protected the streams are taken. As they are not directly connected to any operator of the DIA, a package to put all of them together is added to the StreamGen language in order to make the approach more intuitive for the dataflow application developers.

In the figure 4.2 can be seen the UML profile of this package.

Moreover, in the figure 4.3 can be seen the language of the PrivacyContextSource and PrivacyPolicySource represented in a UML profile diagram.

In summary, independently of the developed DIA, two external sources are always required. A SCV source that specifies the privacy context of the application and a privacy rules source that introduces the privacy policies defined by the users. Moreover, in order to make the approach more intuitive, a package called PrivacyPolicyPackage containing both external sources is added to the StreamGen language.

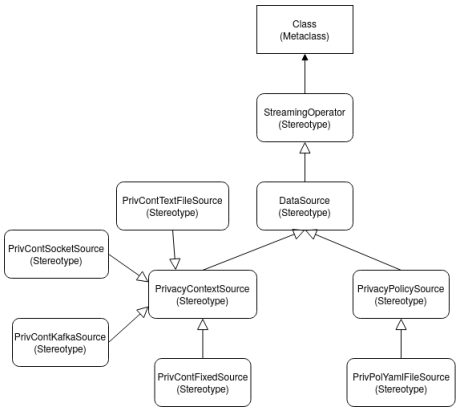

Figure 4.3: PrivacyPolicyDataSources Classes UML Profile

4.1.2.1 PrivacyContextSource

Static Context Variables (SCVs) are required in order to specify under what conditions privacy policies must be applied. According to [17], SCVs are always three variables: purpose, role and observer identifier. PrivacyContextSource stereotype provides who is observing the results of the dataflow application. Then, these SCVs can be just a tuple with the three context variables or many tuples with the three variables. This is why a wide variety of possibilities have been developed in order to allow the users to choose the most appropriate SCV source depending on the DIA that is developed. Such possibilities are:

- PrivContTextFileSource.
- PrivContSocketSource.
- PrivContKafkaSource.
- PrivContFixedSource.

Following the approach of StreamGen with data sources, a stereotype called PrivacyContextSource has been created as a generalization of the already existing DataSource stereotype. This new stereotype has four generalizations, the four possibilities among which the user can choose which is the best source for the observer depending on the application. Each of these four sources have its own properties. These properties have been defined taking into account the already existing data sources:

- TextFileSource.
- SocketSource.
- KafkaSource.

TextFileSource stereotype reads a file which contains all the input values of the DIA. When the user selects this stereotype the path, (pathToFile) from where the DIA must read the file, is specified. PrivContTextFileSource works exactly in the same way, this stereotype contains a property called pathToFile (String) where the

user specifies the path where the file with all the SCV tuples is located. SocketSource stereotype is connected to a port and to a host by means of the properties host (String) and port (Integer) that the user specifies when the stereotype is defined. Following this approach, PrivContSocketSource is connected by a port (Integer) and a host (String) to a server which supplies the SCV tuples.Finally, the KafkaSource is connected by means of kafkaBrokerIp (String) and kafkaBrokerPort (Integer) to a Kafka server. PrivContKafkaSource connects a DIA to a Kafka server that provides the SCV tuples by means of two properties as in the case of the KafkaSource.

In case of PrivContTextFileSource and PrivContSocketSource, when the code is generated a NonParallelStream called contextString which contains the SCV tuples (Strings) is introduced into the DIA. This stream is automatically sent to a map transformation which is imported from the library avoiding that the DIA developer writes any Flink code. This map transformation parses contextString stream to a stream composed with the data type specified by the user in the corresponding package. This data type should contain the same data that each of the strings but the values are split by the comma in order to have each of the values independently. Moreover, this stream is called contextStream and it will be used to apply the privacy policies.

Finally, PrivContFixedSource is a source imported from the library which allows the users to specify only one SCV source. In order to do this, when the user is defining the UML model of the DIA, three properties must be specified. This three properties are fixedUser (String), fixedRole (String) and fixesPuporse (String) which allow to specify the static or fixed SCV source.

In case of PrivContKafkaSource and PrivContFixedSource the contextStream stream with the user specified data type is directly generated without applying any transformation.

In the tables 4.3 can be seen an abstract with all the privacy context sources possibilities, their properties and the multiplicity of each of the properties.

Privacy Context Source	Property Name	Property Type	Property Multiplicity
PrivContTextFileSource	pathToFile	String	1
PrivContSocketSource	host	String	1
	port	Integer	1
PrivContKafkaSource	kafkaBrokerIp	String	1
	kafkaBrokerPort	Integer	1
PrivContFixedSource	fixedUser	String	1
	fixedRole	String	1
	fixesPuporse	String	1

Table 4.3: Privacy Context Source Properties Abstract

4.1.2.2 PrivacyPolicySource

In addition to SCVs, StreamGen requires some user specified rules to guarantee privacy. Such privacy rules can be only some simple conditions that have to be satisfied over the stream where the privacy policy is applied or, also, some past conditions satisfied in any stream of the application, both the stream where the

privacy policy is applied and any other stream of the application. These rules are supposed to be written in a YAML file that must be inputed to the DIA in order to feed the privacy library Flink codes. In order to handle the YAML file, the library snakeyaml (org.yaml.snakeyaml.Yaml) is used. This is the unique type of sources for such purpose as the rules are supposed to be specified by the user and then stored in a YAML file.

Following the approach of DataSource stereotype, a new stereotype called PrivacyPolicySource is created as a generalization of the DataSource stereotype (figure 4.3). The goal with this stereotype is to have the possibility to add different type of sources as currently is happening with the SCV source in spite of the approach developed in this document only considers one of them. Moreover, this stereotype makes the language more understandable. From the PrivacyPolicySource stereotype, a new generalization is created in order to make the relationship with a new stereotype called PrivPolYamlFileSource. This stereotype represents the insertion of the YAML file with all the privacy policies specified by the users whose data is handled to the DIA. In order to specify the path where the YAML file is located, a new property (pathToFile, String) is added to the stereotype. With this property, the user is able to define the location where the file is stored and from where the DIA must read the privacy policy rules written by the user. In the table 4.4 can be seen an abstract of such stereotype.

Privacy Context Source	Property Name	Property Type	Property Multiplicity
PrivPolYamlFileSource	pathToFile	String	1

Table 4.4: Privacy Policy Source Property Abstract

This stereotype follows the example of the TextFileSource stereotype as happened with the PrivContTextFileSource stereotype. However, it is supposed to introduce a YAML file instead of a text file to the dataflow application.

4.2 Privacy Policy Code Generation

StreamGen exploits Acceleo in order to compile StreamGen language into Spark or Flink languages, depending on the target platform of the dataflow application. In this document, Acceleo is used in order to compile the StreamGen language explained above only into Flink code. Moreover, the developed privacy policy language must be translated in a general way allowing that, independently of the DIA, Flink code must not be modified by the application developer.

The above UML metamodel that specifies the language for the privacy policies must be used by the user in order to create the UML class diagram of the DIA in Papyrus. After that, this diagram is used by Acceleo in order to generate the final code which runs in Flink. In this section all the developed Acceleo codes are explained. Furthermore, all along this section an assumption is taken. This assumption is that the UML file generated by Papyrus is going to contain all the packaged classes in the following order:

1. Sources.

2. Transformations.

3. Sinks.

First of all, all the sources introducing data are packaged as elements in the UML file that is generated by Papyrus. After them, all the transformations are packaged. Finally, all the sinks where the data must be stored are packaged. This assumption is similar to say that the non-privacy-aware DIA is built in the above order and without adding transformations, sinks or sources after generating the codes of the non-privacy-aware DIA. This is important because due to the approach used for the compilation of the StreamGen language, if this order is not followed, some errors appear when the code is generated. This assumption is taken in the non-privacy-aware StreamGen version too.

In addition to this assumption, the target code generated by means of StreamGen follows a structure that generates the same pieces of code from the UML language specified before.

4.2.1 Privacy Policy Codes Structure

The generated codes for the designed privacy-aware DIA follow a predefined structure in order to use correctly the privacy policy libraries used in [17]. Each of these parts correspond to a different piece of code that is generated taken into account different conditions. Moreover, as specified before, these pieces of code allow to generate the same Flink codes only with slightly modification taken into account the streams of the dataflow application allowing that the approach can be used independetly of the designed DIA. This structure is defined by four parts:

1. Library Import.

2. Privacy Policy Initialization.

3. Streams Declaration.

4. Streams Protection.

In the Library Import part all the required libraries for the correct code generation operation are called in the corresponding file. Secondly, in the Privacy Policy Initialization part, all the Flink codes for the privacy sources (SCV source and privacy rules source) are generated. After that, in the third part Streams Declaration, all the streams that are not protected are declared in the YAML file in order to follow the approach of [17]. Finally, the streams that must be protected are protected with the privacy library.

4.2.1.1 Library Import

First of all, a condition is required in order to import the libraries of [17] only when they are required. Then, libraries are imported when in the application model exists at least one PrivacyProtectingStream stereotype. If the DIA is non-privacy-aware, there is no PrivacyProtectingStream stereotype and the different libraries are not imported. Such libraries are imported in the file where they are going to be used (generateFlinkApplication.mtl) and the generated piece of code can be seen in the figure 4.4. The imported libraries are:

```
[if (aModel.eAllContents(DirectedRelationship) -> exists(stream | hasStereotype(stream, 'PrivacyProtectingStream')))]
import org.apache.commons.io.FileUtils;
import java.io.File;
import org.yaml.snakeyaml.Yaml;
import it.deib.polimi.diaprivacy.library.GeneralizationFunction;
import it.deib.polimi.diaprivacy.library.ProtectedStream;
import it.deib.polimi.diaprivacy.model.ApplicationDataStream;
import it.deib.polimi.diaprivacy.model.ApplicationPrivacy;
import it.deib.polimi.diaprivacy.model.DSEP;
import it.deib.polimi.diaprivacy.model.PrivacyContext;
import it.deib.polimi.diaprivacy.library.PrivacyContextParser;
import it.deib.polimi.diaprivacy.model.VCP;
import it.deib.polimi.diaprivacy.library.PrivacyContextFixedSource;
[/if]
```

Figure 4.4: Triggering Imports Privacy Libraries

```
[if (aModel.eAllContents(DirectedRelationship) -> exists(stream | hasStereotype(stream, 'PrivacyProtectingStream')))]
<dependency>
  <groupId>org.yaml</groupId>
  <artifactId>snakeyaml</artifactId>
  <version>1.17</version>
</dependency>
[/if]
```

Figure 4.5: Yaml Dependency POM File

- import org.apache.commons.io.FileUtils;
- import java.io.File;
- import org.yaml.snakeyaml.Yaml;
- import it.deib.polimi.diaprivacy.library.GeneralizationFunction;
- import it.deib.polimi.diaprivacy.library.ProtectedStream;
- import it.deib.polimi.diaprivacy.model.ApplicationDataStream;
- import it.deib.polimi.diaprivacy.model.ApplicationPrivacy;
- import it.deib.polimi.diaprivacy.model.DSEP;
- import it.deib.polimi.diaprivacy.model.PrivacyContext;
- import it.deib.polimi.diaprivacy.library.PrivacyContextParser;
- import it.deib.polimi.diaprivacy.model.VCP;
- import it.deib.polimi.diaprivacy.library.PrivacyContextFixedSource;

The library it.deib.polimi.diaprivacy.library can be found in the repository `https://github.com/MicheleGuerriero/dataflow-privacy-library/tree/master/src/main/java/it/deib/polimi/diaprivacy/library`.

Regarding to the library it.deib.polimi.diaprivacy.model, it can be found in the repository `https://github.com/MicheleGuerriero/common/tree/master/src/main/java/it/deib/polimi/diaprivacy/model`.

Finally, the org.yaml.snakeyaml.Yaml library must be imported from the POM file. In order to do this, in the POM file generation Acceleo file (generateFlinkPom.mtl), a dependency is triggled when a PrivacyProtectingStream exists in the application model. In the figure 4.5 can be seen such generated piece of code.

4.2.1.2 Privacy Policy Initialization

At this part, StreamGen extracts the information supplied by the user in the PrivacyPolicyPackage stereotype. StreamGen generates the corresponding SCV source Flink code and the privacy rules source Flink code. In order to do this, each of the packages contained in the application model are taken and if the package is the PrivacyPolicyPackage, for each of the classes contained in the package a different action is applied taking into account the class type. In the figure 4.6 can be seen the written generation algorithm for the privacy sources. The class types that can be contained in the PrivacyPolicyPackage are:

- PrivPolYamlFileSource.
- PrivContFixedSource.
- PrivContKafkaSource.
- PrivContTextFileSource.
- PrivContSocketSource.

In the table 4.5 can be seen an abstract with the actions performed depending on the class type that is included in the PrivacyPolicyPackage. In such table, each of the rows represents: for each class stereotype that can be contained in the PrivacyPolicySources package, the action that is applied.

Class Stereotype	Applied Action
PrivPolYamlFileSource	generateFlinkPrivacyPolicyYamlFileSource
PrivContFixedSource	generateFlinkPrivacyContextFixedSource
PrivContKafkaSource	generateFlinkPrivacyContextKafkaSource
PrivContTextFileSource	generateFlinkPrivacyContextTextFileSource
PrivContSocketSource	generateFlinkPrivacyContextSocketSource

Table 4.5: Applied Actions for Class Stereotypes in PrivacyPolicySources Package

All the actions are applied when there is any of the class stereotypes specified in the table 4.5 are written in the generateFlinkSources.mtl file of the StreamGen Acceleo project as public templates.

In the figures 4.7 and 4.8 can be seen the examples of such generation actions for the class stereotypes PrivPolYamlFileSource and PrivContTextFileSource.

4.2.1.3 Streams Declaration

This document works with the already defined YAML file structure in the repository of [17]. This YAML file owns a field called concreteStream that is null when the YAML file is introduced. This field must be filled for all the streams flowing through the application because later the field is used by the library to protect the streams in the Streams Protection part. Then, after generating and parsing the SCVs and the privacy policy YAML file in the Privacy Policy Initialization part, StreamGen declares all those streams that are introduced to a transformation and

```
[comment StreamGen Privacy Policies Sources/]
[if (aModel.eAllContents(DirectedRelationship) -> exists(stream | hasStereotype(stream, 'PrivacyProtectingStream')))]
            // privacy init
[for (p:Package | aModel.eContents(Package)) ]
[if hasStereotype(p, 'PrivacyPolicyPackage')]
    [generateFlinkPrivacyPolicyYamlFile(p)/]
    [for (c:Class | p.eContents(Class)) ]
        [if hasStereotype(c, 'PrivPolYamlFileSource')]
        [generateFlinkPrivacyPolicyYamlFileSource(c)/]
        [elseif hasStereotype(c, 'PrivContFixedSource')]
        [generateFlinkPrivacyContextFixedSource(c)/]
        [elseif hasStereotype(c, 'PrivContKafkaSource')]
        [generateFlinkPrivacyContextKafkaSource(c)/]
        [elseif hasStereotype(c, 'PrivContTextFileSource')]
        [generateFlinkPrivacyContextTextFileSource(c)/]
        [elseif hasStereotype(c, 'PrivContSocketSource')]
        [generateFlinkPrivacyContextSocketSource(c)/]
        [/if]
    [/for]
[/if]
[/for]
            // finish privacy init
[/if]
```

Figure 4.6: Privacy Sources Generation Algorithm

```
[template public generateFlinkPrivacyPolicyYamlFileSource(aClass : Class)]

    Yaml yaml = new Yaml();

    String content = FileUtils.readFileToString(new File("[getStereotypeProperty(aClass, 'PrivPolYamlFileSource', 'pathToFile')/]"), "UTF-8");

    ApplicationPrivacy app = yaml.loadAs(content, ApplicationPrivacy.class);

[/template]
```

Figure 4.7: GenerateFlinkPrivacyPolicyYamlFileSource Acceleo Template

that the PrivacyProtectingStream stereotype is not applied on them. Regardless of what stream is considered, they are non-protected once in the application. If the application does not protected a stream, the stream exists only without privacy protection. However, if the stream is protected, first of all the non-protected version of the stream is generated and, then, the stream is protected. This is why, every generated stream by a transformation that is not protected, it is declared in the YAML file. In order to do this, a condition must be taken in account every time that a stream is generated by means of a transformation. Such condition is the same condition considered in order to import the libraries; when in the application model exists at least one PrivacyProtectingStream stereotype, then the the output stream of the transformation must be declared in the YAML file. In the figure 4.9 can be seen the piece of code written for such purpose. This piece of code is replicated in all the existing transformations of StreamGen (generateFlinkTransformations.mtl).

Thus, any time that a transformation is generated, after generating the piece of code corresponding to the transformation, one conditions is taken into account. This condition checks that exists at least one PrivacyProtectingStream stereotype in the application model. If such condition is satisfied, then, the stream is set into the privacy policy YAML file.

4.2.1.4 Streams Protection

StreamGen consists of three different types of operators: sources, transformations and sinks. Sources introduce some strings that must be parsed by a transformation

```
[template public generateFlinkPrivacyContextTextFileSource(aClass : Class)]
    // begin privacy source definition
    DataStream<String> contextString = env.readTextFile("[getStereotypeProperty(aClass, 'PrivContTextFileSource', 'pathToFile')/]")
        [if getParallelism(aClass, 'PrivContTextFileSource').toString().toInteger() > 1 ]
        .setParallelism([getParallelism(aClass, 'PrivContTextFileSource')/]);
        [else]
        ;
        [/if]

    DataStream<PrivacyContext> contextStream = contextString.map(new PrivacyContextParser());
    // finish privacy source definition
[/template]
```

Figure 4.8: GenerateFlinkPrivacyContextTextFileSource Acceleo Template

```
[if (aClass.getModel().eAllContents(DirectedRelationship) -> exists(stream | hasStereotype(stream, 'PrivacyProtectingStream')))]
app.getStreamByID("[getOutputNames(aClass)->first()/]").setConcreteStream([getOutputNames(aClass)->first()/]);
[/if]
```

Figure 4.9: Stream Declaration

```
[if hasStereotype(c, 'SocketSource')]
    [generateFlinkSocketSource(c)/]
```

Figure 4.10: Triggering SocketSource Privacy StreamGen Version

in order to generate a data type. This happens independently of the implemented data source. Transformations make some computation modifying and generating new data types. Finally, some of the generated data types are stored in a sink.

On the other hand, a transformation can introduce a pair of streams (CoFlatMap and CoMap transformation) and, also, it can generate several streams. StreamGen is implemented in a way that assumes that all the generated streams are the same stream. Then, in spite of several streams must be specified in the UML diagram, Acceleo takes only one of such outputs in order to generate the transformation. Following this approach, privacy policies are going to be generated only once assuming that all the privacy output streams of a transformation are the same stream.

Then, the Acceleo algorithm developed in StreamGen is extended. Before extending the algorithm, StreamGen checks one by one all the transformations that are contained in the application model and depending on the applied stereotype to the classes, generates different actions. If this is not modified, it allows to generate all the streams, regardless of whether they are protected. Once the non-protected stream generated by the transformation is created, an extension of the algorithm is made. A condition is taken into account in order to check if the output stream must be protected or if it must not. In order to do this, if exists an output stream from the transformation that contains the stereotype PrivacyProtectingStream, an action called generateFlinkPrivPolicies is executed. An example for the SocketSource, MapTransformation and TextFileSink stereotypes of the extended algorithm is presented in the figures 4.10, 4.11 and 4.12. As it can be seen there, the protection over the streams is only applied on the output streams of the transformations. This ensures that any stream is protected only once as it takes only one of all the replicas of the output streams. Moreover, two assumptions are taken. The first one is that any source requires a parser to transform the input stream into a DIA data type. The second one is that all the output streams generated by a transformation are protected with the same privacy policy types (VCP, DSEP or both of them).

This extended algorithm triggers the generateFlinkPrivPolicies action. Such action consists of four steps and it is written in the generateFlinkTransformations.mtl file of the StreamGen project. Moreover, this action takes advantage of the privacy library.

First of all, the stream declared in the Streams Declaration part is called. In the figure 4.13 can be seen how the declared stream is called. After that, a protected stream with the values specified in the ProtectedStreamConfiguration data type

```
[elseif (hasStereotype(c, 'MapTransformation'))]
    [generateFlinkMapTransformation(c)/]
    [if (c.getOutputs() -> exists(output | hasStereotype(output, 'PrivacyProtectingStream')))]
    [generateFlinkPrivPolicies(c.getOutputs() -> select(output | hasStereotype(output, 'PrivacyProtectingStream')) -> first())/]
    [/if]
```

Figure 4.11: Triggering MapTransformation Privacy StreamGen Version

```
[elseif (hasStereotype(c, 'TextFileSink'))]
    [generateFlinkTextFileSink(c)/]
[/if]
```

Figure 4.12: Triggering TextFileSink Privacy StreamGen Version

```
ApplicationDataStream app_[stream.eGet('name')/] = app.getStreamByID("[stream.eGet('name')/]");
```

Figure 4.13: Calling Declared Stream

of the property protectedStreamConf of the PrivacyProtectingStream stereotype is generated. Then, taking into account if the stream is VCP or DSEP protected with the properties: protectedByVCP and protectedByDSEP, the VCPs and DSEPs of the YAML file are added to the protected stream previously declared. Such step can be seen how is developed in the figure 4.14. Finally, all the privacy policies are applied on the stream. This steps takes advantage of the finalize method developed in the privacy library. The step is specified in the figure 4.15.

```
[if isVCPprotected(stream)]

    for (VCP vcp : app.getVCPs(app_[stream.eGet('name')/].getId())) {
        [stream.eGet('name')/]_p.addVCP(app_[stream.eGet('name')/], (VCP) vcp, app);
    }
[/if]
[if (isDSEPprotected(stream))]

    for (DSEP dsep : app.getDSEPs(app_[stream.eGet('name')/].getId())) {
        [stream.eGet('name')/]_p.addDSEP(app_[stream.eGet('name')/], (DSEP) dsep, app);
    }
[/if]
```

Figure 4.14: Adding Privacy Policies

```
DataStream<[stream.eGet('conveyed').name/]> [stream.eGet('name')/]_f = [stream.eGet('name')/]_p.finalize(env, contextStream);
```

Figure 4.15: Applying Privacy Policies

Chapter 5

Evaluation

5.1 Great Seller Privacy Aware DIA

In this section a DIA for an e-commerce company called Great Seller is implemented. This DIA is able to compute real time statistics about the transactions generated by the consumers. Such statistics can be observed by some other companies that pay in order to have access to the information making that Great Seller acts as a data broker. In order to simplify the implementation, Great Seller is going to sell three different types of statistics:

- Statistic 1: the total amount of money that each consumer of Great Seller spend in the last 10 minutes.
- Statistic 2: the number of transactions issued by each user in the last 10 minutes.
- Statistic 3: the number of users who spent more than €1000 in the last hour.

As any DIA, Great Seller generates such statistics by some transformations which are fed from a source and the generated information is stored into a sink which is accessible by the observer companies. Moreover, as Great Seller is producing three different types of statistics, its DIA requires three different sinks where the information should be stored. Due to the fact that Great Seller DIA is computing real time statistics about the transactions that are generated by the consumers of the company, only one source is required for the DIA model. Finally, one transformation is necessary for the computation of each statistic. In summary, Great Seller DIA requires one source, three transformations and three sinks for the design of its model.

The first step, after generating the Papyrus project (appendix A), is to define the model in the class diagram in order to specify that the application is going to be a Flink application. Then, first of all, we have to create a model node in the greatSellerApp.di file. This node is going to be called greatSeller as it is the object representing the whole application. Once the node is inserted, the properties of such node must be specified. More in detail, the Flink application stereotype has to be applied. In this case, the properties of the model has to be the default ones then nothing else must be done with this node. In the figure 5.1 can be seen how the stereotype has been applied to the model node.

5.1.1 Great Seller Source

In the implementation described along this section, Great Seller DIA is fed from a socket that reads from a text file where all the transactions generated by the consumers of Great Seller are stored. All the transactions are going to have the same predefined tuple structure 'transactionId,dataSubject,spentAmount,purchasedProduct'. This predefined structure is represented in the Great Seller DIA model by means of a data type called InputTransaction.

Regarding to the fields of such tuple, the transactionId field is an integer which varies from 1 to the number of generated transactions, such number can be 10, 100 or 1000. The dataSubject is the user who generates such tuple and it is one of the following: Bob, Carlos, Elisabetta or Michele. The spentAmount is the price paid for the product that has been bought with the transaction and it is an integer with a low boundary of €1 and an upper boundary of €200. Finally, the recipientId is the product bought with such transaction.

The steps that must be followed in order to add the data types that are requited to the application are the followings. First of all, a package node has to be inserted inside the Flink application model node. This package is called greatSellerDataTypes and a stereotype has to be applied as in the previous case. First of all, inside the properties window, in the profile field, the applied stereotype is defined. Inside this package all the datatypes that the application requires have to be inserted. The first data type is the InputTransactions which is composed of a transaction id (integer), a data subject (string), an amount (double) and a recipient id (string). The second data type is the IssuedTransactions which contains the data subject (string) and the number of transactions performed by such data subject which can be seen by means of the variable NTransactions (integer). The third data type is the SpentAmount which is composed of the data subject (string) and the total amount of money spent by the data subject which can be seen by means of the variable TotalAmount (double). Finally, the last data type is the NumberUsers which contains the number of users who spent more than 1000 dollars and it is represented by means of the variable NTopUsers (integer). Then, we need to insert one DataType node inside the package created in the previous step with each of these data types. Each data type can be seen as a tuple which is composed by several values. The name of the DataType node is going to be the same that the one corresponding to the tuple and each of the values that compose the tuple are going to be a property inside of the owned attributes that are specified in the UML field of the properties of each DataType node.In the table 5.1 can be seen an abstract with all the data types required for this example. Moreover, in the figure 5.1 can be seen the package with all the data types of the application inserted in the application model.

It is important to remark that socket source introduces in the DIA a data stream of strings and the InputTransaction data type must be generated by means of a transformation which parses the string into the data type. This transformation is called TupleParser as it splits the incoming tuples, strings, into the InputTransaction data type.

The Great Seller stock is composed by means of 25 products. In order to simplify the implementation, each product is named by the word 'product' immediately followed by a number between 1 and 25 in order to specify the product referred in the stock. In the table 5.2 can be seen such stock.

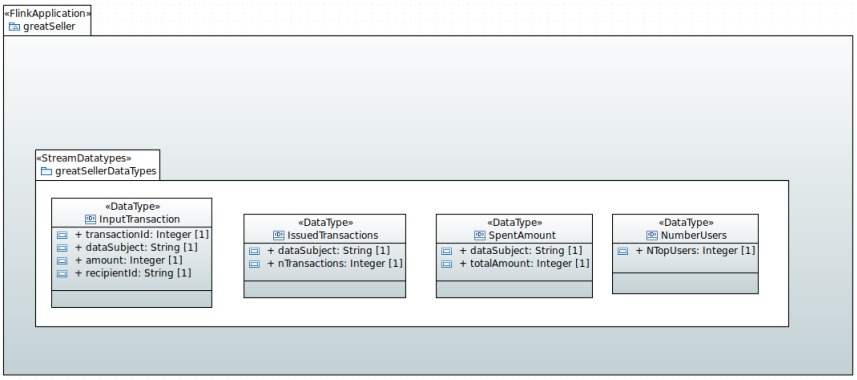

Figure 5.1: Great Seller Data Types Package

Data Type	Property Name	Property Type
InputTransactions	transactionId	integer
	dataSubject	string
	amount	integer
	recipientId	string
IssuedTransactions	dataSubject	string
	nTransactions	integer
SpentAmount	dataSubject	string
	totalAmount	integer
NumberUsers	NTopUsers	integer

Table 5.1: Great Seller Data Types

In order to generate this stock that feeds the Great Seller DIA, two Python codes have been developed. Furthermore, a Python server has been developed to input the transaction tuples to the dataflow application.

The first Python code (appendix B.1) builds a Python list which represents the Great Seller stock of 25 products and it assigns to each product a price. Each product is represented with a dictionary variable that contains a name variable and a price variable. The name variable is a string with the word 'product' immediately followed by an integer number from 1 to 25 which points to the product of the stock. The price is an integer number between $1 and $200 which is assigned randomly. Once the name and the price are assigned to the dictionary, the product is added to the stock list. Finally, the Great Seller stock list is saved in a binary shelf file in order to be accessible from the other Python code.

The second Python code (appendix B.2) generates the strings that represent the tuples produced by each consumer. In order to generate such tuples an integer number for the transactionId is assigned following an increasing numerical order from 1 to the maximum number of generated transactions which is input by command line to the code, it can be 10, 100 or 1000. After that, randomly, one of the four possible data subjects (Bob, Carlos, Elisabetta and Michele) and a product from the stock saved in the binary shelf file are assigned. Finally, each value is added to a string where each of these values are separated by a comma. The generated string represents the tuple produced by each consumer and it is written in the text file which is called from the Great Seller DIA in order to feed it.

Product	Price (€)
product1	196
product2	36
product3	179
product4	17
product5	120
product6	187
product7	139
product8	52
product9	160
product10	110
product11	113
product12	67
product13	100
product14	125
product15	192
product16	115
product17	113
product18	98
product19	113
product20	185
product21	143
product22	18
product23	194
product24	41
product25	26

Table 5.2: Great Seller Stock

Finally, the DIA inputs each of these strings into the DIA by means of a Python server (appendix B.3). This Python server introduces a non-parallel stream with the generated tuples (strings) and the stream is sent to the first transformation of the dataflow application.

5.1.2 Great Seller Transformations

The first transformation implemented in the application is the TupleParser. This transformation is a Map Transformation that is implemented in order to split the input strings supplied by the Python servers into the InputTransaction data type. This transformation takes advantage of the split Java method and it splits the strings by the comma generating the InputTransaction data type. The generated stream is sent to the OP1 and OP2 operators.

In order to compute each of the three statistics that Great Seller sells as data broker, the Great Seller DIA needs three transformations. Each of this transformations is the operator of each statistic. Thus, the operator one (OP1) computes the total amount of money spent by each user in the last 10 minutes. The second operator (OP2) computes the number of transactions issued by each user in the last

10 minutes. Finally, the operator three (OP3) computes the number of users who spent more than €1000 in the last hour.

These transformations input a data stream with a set of tuples that all of them have identical structure. Thus, the first stream (S1) is composed of tuples of the kind InputTransaction. This stream is duplicated and it is sent to the operators OP1 and OP2. Moreover, as each operator works taking into account the data subject of each tuple, the stream S1 must be keyed by the data subject field of this first stream. Finally, as the operators have to compute the statistics taking into account only the tuples generated in the last 10 minutes, the stream S1 is windowed by a time window of 10 minutes.

The new data generated in the OP1 are represented in a new data type called SpentAmount and that is composed by two fields following the structure: 'dataSubject, totalAmount'. The SpentAmount tuples are collected in the stream S2 and they are keyed by the data subject field of such stream.

Finally, the data generated in the OP2, is collected in a new data type called IssuedTransaction whose structure is 'dataSubject, nTransactions'. This new stream (stream S3) is sent to the OP3. Moreover, this third stream is keyed by the data subject field of the stream and it is also windowed with a time window of one hour. The OP3 generates a randomly partitioned stream with an uniform distribution called stream S4 whose structure is: 'nTopUsers'.

5.1.3 Great Seller Sinks

Each of the tuples flowing through the streams S2, S3 and S4 are stored in a file. Such values are stored because the observer companies are able to access to the tuples. This is why, after each stream is generated (streams S2, S3 and S4), a file text sink is implemented. These file are going to be stored in the same location but with different names. The file resultsS2.txt stores the generated tuples in the stream S2; the file resultsS3.txt stores the values flowing through the stream S3 and the file resultsS4.txt stores the values of the stream S4.

This implementation is given due to the fact that each of the tuples generated by the operators are observable for the observer companies who buy them. Furthermore, such sinks allow to check if the transformations are well implemented by the designer of the DIA. In the figure 5.2 the developed model with StreamGen for the non-privacy-aware Great Seller DIA can be seen.

5.1.4 Great Seller Privacy Enforcement

At this point, the non-privacy-aware dataflow application for the e-commerce company Great Seller is completed and the approach for the privacy-aware DIAs that is developed in this document must be applied in order to see if the previously stated objectives are met. The enforcement of the privacy model is made by means of two steps:

1. Protected Streams Definition

2. Privacy External Sources Definition

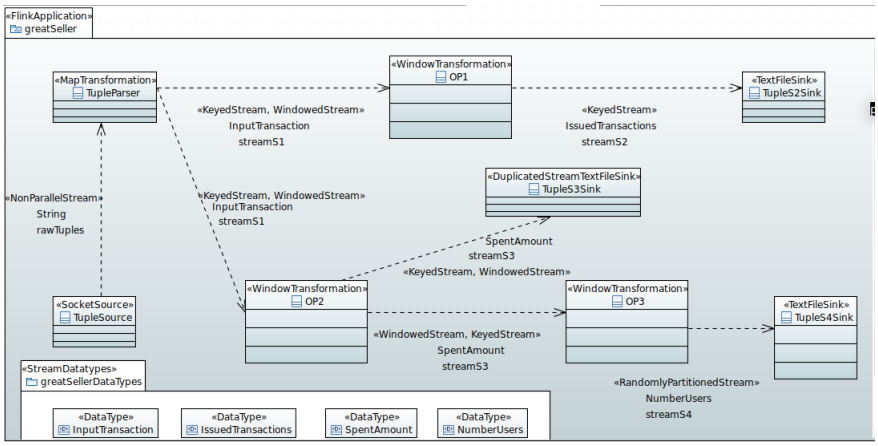

Figure 5.2: Great Seller StreamGen DIA

5.1.4.1 Protected Streams Definition

Once the non-privacy-aware application is implemented, the privacy metamodel is applied. First of all, the streams that should be protected are specified. In the Great Seller application, such streams are streamS2 and streamS3. Then, the PrivacyProtectingStream stereotype is applied on both streams. However, streamS2 is protected by a VCP whilst streamS3 is protected by a DSEP. This means that the properties of the stereotypes are differents.

In the case of the streamS2 stereotype, as this stream is protected by a VCP, only such property must be true. In addition to this, the protectedStreamConf is specified according to the values that can be shown in the table 5.3. The timestampServerIp field of the protectedStreamConf is empty, this is why nothing is specified.

Property	Field Name	Field Value
protectedByVCP	-	true
protectedByDSEP	-	false
protectedStreamConf	monitoringActive	false
	timestampServerIp	
	timeStampServerPort	-1
	topologyParallelism	1
	simulateRealisticScenario	false
	allowedLateness	0
	logDir	/home/cablan/Desktop/thesis/conf/

Table 5.3: PrivacyProtectingStream Great Seller StreamS2

StreamS3 is specified in a similar way, in this case the protectedStreamConf is exactly the same but the protectedByVCP property is false and the protectedByDSEP is true. In the table 5.4 can be seen the specified. As in the other case, the timestampServerIp field is empty because no value is specified there.

In the figure 5.3 can be seen how looks the application after defining the streams that are protected.

Property	Field Name	Field Value
protectedByVCP	-	false
protectedByDSEP	-	true
protectedStreamConf	monitoringActive	false
	timestampServerIp	
	timeStampServerPort	-1
	topologyParallelism	1
	simulateRealisticScenario	false
	allowedLateness	0
	logDir	/home/cablan/Desktop/thesis/conf/

Table 5.4: PrivacyProtectingStream Great Seller StreamS3

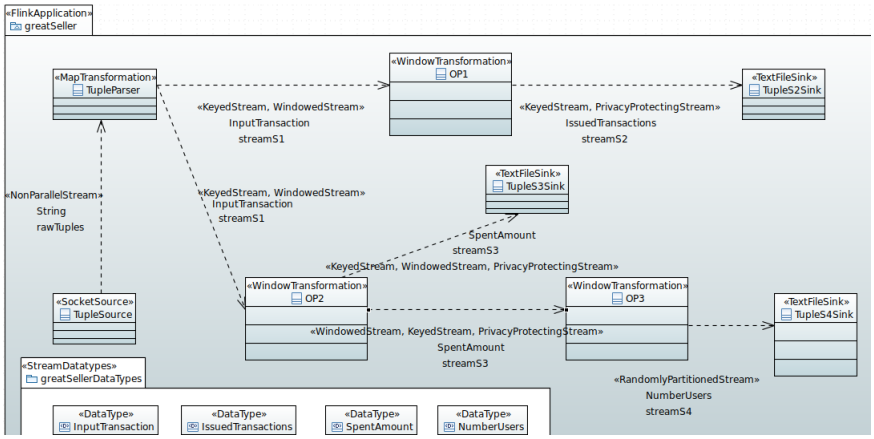

Figure 5.3: Great Seller Privacy Aware StreamGen DIA

5.1.4.2 Privacy External Sources Definition

Once the privacy streams have been defined, the second step is to define the SCV source and the privacy rules source. The privacy rules source have only one choice; however, the SCV source can be defined in several ways. In this example, as the input is working with sockets, the socket source is defined for this purpose.

But before introducing the classes of the sources, the PrivacyPolicyPackage stereotype must be defined in a new package in order to make more intuitive the approach to the DIA developer. In order to do this, a new package node is introduced in the application model node and the stereotype PrivacyPolicyPackage is applied in such new package that is called PrivacyPolicyInputs. Once the package is introduced into the model, the SCV source is defined. In order to do this, a new class is introduced inside the package called PrivacyPolicyInputs and the PrivContSocketSource stereotype is applied on it. Such new class is called StaticContextVariablesSource and their properties are filled with the values localhost (host, string) and 9998 (port, integer). Finally, another new class is introduced in the package. In this case, it is called PrivacyPolicySource and the PrivPolYamlFileSource stereotype is applied on it. This stereotype contains only one property (pathToFile) that must be filled. In this case, it is filled with the value /home/cablan/Desktop/thesisFiles/config/privacy-config.yml that is the path where the YAML file is located. In the figure 5.4 can be seen the introduced package

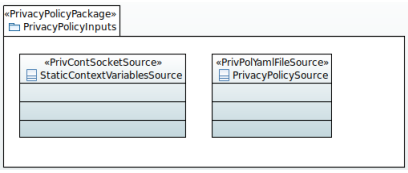

Figure 5.4: Great Seller Privacy Policy Package

and classes with their applied stereotypes.

At this point, the privacy-aware DIA is implemented and the configuration is run in order to generate the Java codes of the application. In the appendix C can be seen how to run such configuration.

5.1.5 Great Seller DIA Limitations

After developing this DIA, some limitations are identified relative to the current development of StreamGen. The first limitation is the number of servers that can be connected directly to a Map Transformation. In this example, it is supposed that the users send the transactions by means of a server to the target DIA. This server is the same for all the users who are using Great Seller DIA. At this point, a casuistry is proposed. This is the case of a DIA involved in an industry 4.0 application where the owner of the factory cannot afford to invest in a server for the DIA and all the machines generating datasets must be connected directly to the DIA by means of the factory net.

The second limitation that is extracted is that StreamGen does not allow to generate float or double data types. In this example, Great Seller is working with a stock where all the products have integer prices but should be necessary a DIA which allows, among the language, to declare float or double values.

In the following example, both casuistry are fixed and the privacy model is applied to check its behavior.

5.2 Cool Analyst Privacy Aware DIA

In addition to Great Seller, another DIA is developed with StreamGen in order to check more in detail the developed approach for the privacy-aware DIAs. This is the case of Cool Analyst, a DIA which is able to compute some statistics from the temperatures measured in two different refrigeration chambers of a panettone factory called Panettone 4.0. These measurements are going to be stored in a CSV file in order to be accessible by the manager of the industrial plant to see if any problem has been produced in the yeast fermentation. This application is going to compute four statistics in two different operators:

- Operator 1: the maximum, the minimum and the average temperature of each chamber during the last 5 minutes.
- Operator 2: the prediction of the temperature for the next 10 minutes.
- Operator 3: the filtered data between a range which have no null value and no null id.

Cool Analyst DIA is going to require two socket sources, one for each of the industrial chambers, two transformations in order to compute each of the operators specified before and two sinks where the CSV files with each of the results of the operators can be stored. For the development of the prediction operator, as its development is out of this document, it is taken from an already working transformation for such purpose.

Following the Great Seller approach, before generating the dataflow application, the first step is to create the Papyrus project according to the appendix A. After that, the application model is introduced in the Papyrus project. In order to do this, a model node is defined in Papyrus and the steotype FlinkApplication is applied on it. As in the previous example, the default properties for such stereotype are the final ones, then, they must not be modified. In the figure 5.5 can be seen the model and the applied stereotype.

5.2.1 Cool Analyst Sources

Panettone 4.0 is using temperature digital sensors in each of its two chambers where yeasts are fermented. These sensors are connected to the net of the factory and are sending the measured values directly to the Cool Analyst DIA with a 1 second sample frequency.

This two chambers need to control the temperature of the room in order to allow the correct fermentation of the yeasts. The role of the first chamber is to make the primary fermentation in the production process of the Panettone 4.0 factory. This fermentation is also known as bulk fermentation. On the other hand, the role of the second camera is to make the secondary fermentation which is also known as proofing. More information can be read about the fermentation of the years and how to control the temperature during its production in [21]. The conclusions that are extracted from [21], and that are useful for our purpose, are that the bulk fermentation must have the temperatures of the room in the range [20, 24] Celsius whilst the secondary fermentation must have an environment with temperatures in the range [22, 29] Celsius. This is why Cool Analyst requires two socket sources connected to the port and to the host of each of the chambers.

The two sockets send a tuple with the chamber identifier (room1 or room2) and the temperature of the chamber in a string separating both values by a comma. This string, as happens in the Great Seller example, is sent to a parser which generates a data type and, then, Cool Analyst works with several data types that must be defined in a package according to the StreamGen behavior. In order to do this, a package node is introduced into the model node and it is called coolAnalyst-DataTypes. After that, the StreamDatatypes stereotype is applied on the package and all the data types required by the DIA are defined. The first data type is the roomTemp data type. It is composed of two different values: roomId (string) that is the identifier of the chamber and roomTemp (double) which is the current temperature of the chamber. The second required data type is the roomStatistics data type which is composed of four values: roomId (string), maxTemp (double), minTemp (double) and avgTemp (double). The three last values of the roomStatistics data type (maxTemp, minTemp and avgTemp) are the maximum, minimum and average computed values of the chamber identified with the roomId string. Finally, the last requited data type for the Cool Analyst DIA is the tempPred which is a prediction

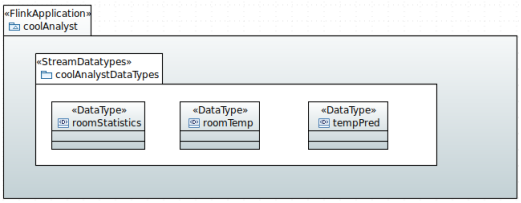

Figure 5.5: Cool Analyst Data Types Package

of the chambers for the following ten seconds. This data type is composed of the values roomId (string) and predTemp (double) which are the predicted temperature of the room identified by the roomId string. For each of the three data types required by the Cool Analyst DIA, a Data Type node is inserted in the package called coolAnalystDataTypes and a property for each of the values that compose the data type are inserted as an owned attribute of the data type. In the table 5.5 can be seen an abstract with all the data types required by the dataflow application. Furthermore, in the figure 5.5 can be seen the package with all the data types of the application inserted in the application model.

Data Type	Property Name	Property Type
roomTemp	roomId roomTemp	string double
roomStatistics	roomId maxTemp minTemp avgTemp	string double double double
tempPred	roomId predTemp	string double

Table 5.5: Cool Analyst Data Types

Also it is important to remark how the temperatures of the chambers are simulated. For such purpose a python code has been developed for each of the chambers (appendices B.4 and B.5). These models take as target temperature, the average temperature of each of the ranges. This temperature is supposed to be maintained during ten seconds. Then, randomly, the program generates an increasing or decreasing sine function with a frequency of 1/16 Hz for the lower range and with a frequency of 1/28 Hz for the upper range. The lower range is considered from the lower boundary of each of the ranges of temperature that the chambers can admit ([22, 29] Celsius in the chamber one and [20, 24] Celsius in the chamber two) until the average temperature and the upper range from the average temperature until the upper boundary of the ranges. In the figures 5.6 and 5.7 can be seen both models. The simulation is built for one hundred temperature values.

Finally, these two models are saved in a TXT file which is going to be read from two Python servers (appendices sec:Appendix2Sec6 and sec:Appendix2Sec7) that are developed in order to simulate the chambers working independently. These servers read one value each second during the period that the simulation lasts. This generates that the socket sending the values from the chamber one stops earlier than the other socket. This model design allows to proof the right behavior of the

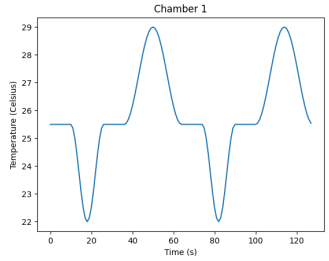

Figure 5.6: Temperature Chamber 1 Model

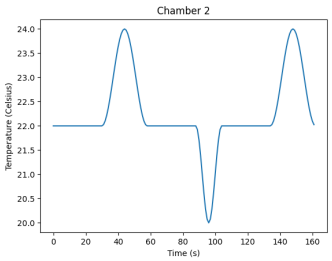

Figure 5.7: Temperature Chamber 2 Model

whole application, allowing to see what would happen if, suddenly, the sensors are disconnected.

5.2.2 Cool Analyst Transformations

The values read from the TXT file are sent in a string which has to be parsed in order to generate the data types that are used in the DIA. The strings contain the roomId followed by the temperature and both values are separated by means of a comma. This is why, first of all a NMap transformation called generateFlinkCoMapTransformation is developed. In order to do this, in the generateFlinkTransformations.mtl file, the existing but not developed generateFlinkCoMapTransformation is written taking as approach the transformations called generateFlinkCoFlatmapTransformation and generateFlinkMapTransformation. In the figure 5.8 can be seen the written Acceleo code. This NMap transformation inputs the two streams with the strings sent by the room servers and the incoming strings are parsed into a roomTemp data type which is composed of two properties: roomId (string) and roomTemp(double). Furthermore, in order to introduce double values from the Papyrus model by means of StreamGen, the generateDataTypes.mtl file is slightly modified. The modification is made considering the code previously developed for the Long values. Then, every time that a property name is EDouble, it is substitute by Double. In the figure 5.9 can be seen the piece of code added to the file and how this approach follows the Long approach. Moreover, this piece of code is added several times, as many times as necessary.

Once the roomTemp data type is generated, a stream is sent to a window transformation called RoomStatistics. In this transformation all the computations explained previously in the operator one are calculated (maximum temperature, minimum temperature and average temperature). This transformation generates a new data type called roomStatistics. Then, the stream with the roomStatistics data

```
[template public generateFlinkCoMapTransformation(aClass : Class)
    {first : DirectedRelationship = getInputs(aClass)->at(1);
     second : DirectedRelationship = getInputs(aClass)->at(2);}]

    // begin stream definition
    DataStream<[getOutputsConveyed(aClass) ->first()/]> [getOutputNames(aClass)->first()/] = [first.eGet('name')/]
        [if first.hasStereotype('KeyedStream') ]
        .keyBy("[getStereotypeProperty(first, 'KeyedStream', 'key').eGet('name')/]")
        [elseif first.hasStereotype('BroadcastedStream')]
        .broadcast()
        [/if]
        .connect(
        [second.eGet('name')/]
        [if second.hasStereotype('KeyedStream') ]
        .keyBy("[getStereotypeProperty(second, 'KeyedStream', 'key').eGet('name')/]")
        [elseif second.hasStereotype('BroadcastedStream')]
        .broadcast()
        [/if]
        )
        .map(new [aClass.name/]())
        [if getParallelism(aClass, 'NMapTransformation').toString().toInteger() > 1 ]
        .setParallelism([getParallelism(aClass, 'NMapTransformation')/]);
        [else]
        ;
        [/if]

        [if (aClass.getModel().eAllContents(DirectedRelationship) -> exists(stream | hasStereotype(stream, 'PrivacyProtectingStream')))]
        app.getStreamByID("[getOutputNames(aClass)->first()/]").setConcreteStream([getOutputNames(aClass)->first()/]);
        [/if]
        // finish stream definition
[/template]
```

Figure 5.8: NMap Transformation StreamGen

```
[for (p: Property | aClass.attribute) separator('\n')]
   private [p.type.name.substitute('ELong', 'Long').substitute('EDouble', 'Double')/] [p.name/];
[/for]
```

Figure 5.9: Double StreamGen Type

types is sent to the TemperaturePredictor transformation. This window transformation computes the operations described in the operator two and generates the tempPred data type. The last operator takes the stream with the roomTemp data type from the RoomStatistics transformation and, by means of a filter transformation called CleanRawData, checks that there is no temperature with a value below -9999 neither 9999. Moreover, this transformation checks that the temperature has a value and that value is referenced to a room identifier.

5.2.3 Cool Analyst Sinks

Finally, the data types generates in the RoomStatistics, in the TemperaturePredictor and in the CleanRawData transformations are stored each one in its sink. Cool Analyst uses CSV sinks as this kind of files are commonly used by engineers who work in the factories. In order to do this, the output stream from the RoomStatistics transformation is sent to StatisticsSink where is specified the path of the generated file, the output stream from the TemperaturePredictor transformation is sent to PredictorSink where the path of the second output file is specified and the same proceadure is done with the third operator CleanRawData and its sink CleanDataSink.

In the figure 5.10 can be seen the Cool Analyst StreamGen DIA explained above which is the non-privacy-aware application of the Cool Analyst DIA.

5.2.4 Cool Analyst Privacy Enforcement

At this points, the privacy approach for dataflow applications is applied to the already developed non-privacy-aware DIA model by defining the two already known steps. This example allows to check what happens with the developed approach for privacy-aware dataflow applications when a transformation generates a stream that is required to be protected partially, depending on the destination transformation.

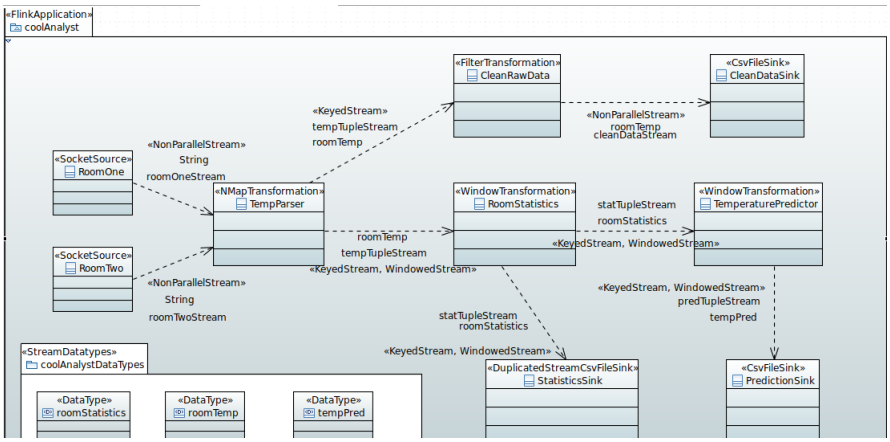

Figure 5.10: Cool Analyst StreamGen DIA

5.2.4.1 Protected Streams Definition

In this DIA a DSEP is required in order to avoid tuples coming from the first chamber to be considered in the computation of the RoomStatistics transformation. From the designer perspective, no more information is required in order to define the characteristics of the protected stream. This is why only one stream requires to be protected by privacy policies.

In order to do this, the PrivacyProtectingStream stereotype is applied in the stream known as tempTupleStream flowing from the TempParser transformation to the RoomStatistics transformation. Once the stereotype is applied, the properties for such stereotype are defined. In this case, the stream is protected by a DSEP (protectedByDSEP is true) but it is not protected by any VCP (protectedByVCP is false). Moreover, protectedStreamConf data type is defined with the same values that the protected streams in the Great Seller example were designed. In the table 5.6 can be seen how are defined all these values.

Property	Field Name	Field Value
protectedByVCP	-	false
protectedByDSEP	-	true
protectedStreamConf	monitoringActive	false
	timestampServerIp	
	timeStampServerPort	-1
	topologyParallelism	1
	simulateRealisticScenario	false
	allowedLateness	0
	logDir	/home/cablan/Desktop/thesis/conf/

Table 5.6: PrivacyProtectingStream Cool Analist TempTupleStream

The value for the timestampServerIp is empty because, as in the previous example, it is defined to nothing. All these values can be modified but they are considered as the default values for any protected stream.

In the figure 5.11 can be seen the applied stereotype.

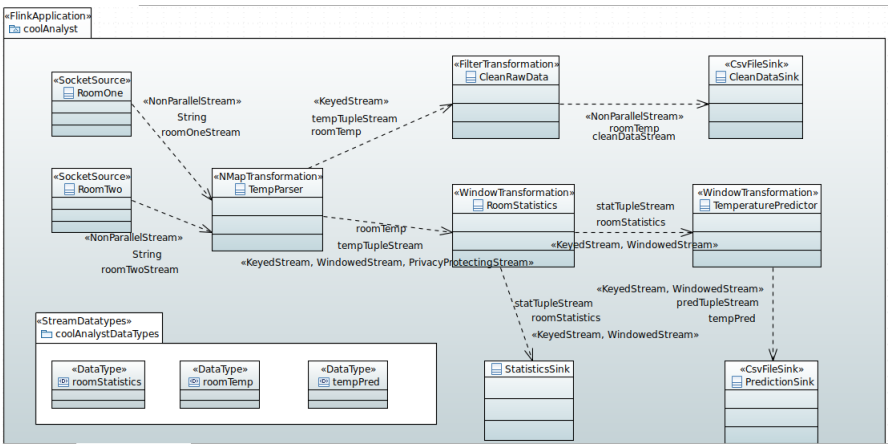

Figure 5.11: Cool Analyst Privacy Aware StreamGen DIA

5.2.4.2 Privacy External Sources Definition

After defining the protected streams, the external privacy sources are introduced. In this case, the predefined DSEP must be applied every time that an employee who is not the manager of the Panettone 4.0 factory tries to see the statistics of the first chamber. Then, a TXT file with all the employee identifiers and assuming that they try to access with analytical purposes is created and accessed by the SCV source.

In order to implement this, first of all, a package node called ExternalPrivacySources is introduced in the application model and the PrivacyPolicyPackage stereotype is applied on it in order to make more intuitive how to introduce the SCV source and the privacy rules source.

After that, a class node called SCVSource is introduced in the ExternalPrivacySources package and the PrivContTextFileSource stereotype is applied on it. This stereotype requires to define the pathToFile (string) property, in the table 5.7 can be seen such path. Moreover, in the table **??**an be seen the content of the text file.

Once the SCV source is completely defined, the privacy rules source is introduced. For such purpose, a class node called PrivacyRulesSource is introduced in the ExternalPrivacySources package. In this class the PrivPolYamlFileSource is applied and its property, pathToFile (string), is defined according to the table 5.7.

Class	Property Name	Property Value
SCVSource	pathToFile	/home/cablan/Desktop/thesis/conf/
PrivacyRulesSource	pathToFile	/home/cablan/Desktop/thesis/conf/

Table 5.7: Cool Analist External Sources Properties

Observer Identifier	Purpose	Role
employee1	analytics	employee
employee2	analytics	employee
employee3	analytics	employee

Table 5.8: Cool Analist SCV Text File

Chapter 6

Conclusions

Bibliography

[1] A.M. Middleton. *Data-Intensive Technologies for Cloud Computing*. Handbook of Cloud Computing. Springer, 2010.

[2] D.B. Skillicorn and D. Talia. *Models and languages for parallel computation*. ACM Computing Surveys, Vol. 30, No. 2, 1998, pp. 123-169.

[3] I. Gorton, P. Greenfield, A. Szalay and R. Williams. *Computing in the 21st Century*. IEEE Computer, Vol. 41, No. 4, 2008, pp. 30-32.

[4] W.E. Johnston. *High-Speed, Wide Area, Data Intensive Computing: A Ten Year Retrospective*. IEEE Computer Society, 1998.

[5] M. Gokhale, J. Cohen, A. Yoo and W.M. Miller. *IEEE: Hardware Technologies for High-Performance Data-Intensive Computing*. IEEE Computer, Vol. 41, No. 4, 2008, pp. 60-68.

[6] D. Ravichandran, P. Pantel and E. Hovy. *The terascale challenge*. Proceedings of the KDD Workshop on Mining for and from the Semantic Web, 2004.

[7] J. Gray. *Distributed Computing Economics*. ACM Queue, Vol. 6, No. 3, 2008, pp. 63-68.

[8] R.E. Bryant. *Data Intensive Scalable Computing*. 2008.

[9] Hadoop Tutorial
`https://data-flair.training/blogs/hadoop-tutorial/`

[10] Spark Tutorial
`https://data-flair.training/blogs/spark-tutorial/`

[11] Ververica Web Page
`https://www.ververica.com/what-is-stream-processing`

[12] Apache Flink Tutorial
`https://data-flair.training/blogs/apache-flink-tutorial/`

[13] Ken Thompson. *Reflections on Trusting Trust*. Communications of the ACM, 1984.

[14] General Data Protection Regulation Web Page
`https://gdpr-info.eu/`

[15] Carroline Dewi Puspa Kencana Ramli, Hanne Riis Nielson, Flemming Nielson. *The logic of XACML*.

[16] Markus Lorch, Seth Proctor, Rebekah Lepro, Dennis Kafura, Sumit Shah. *First Experiences Using XACML for Access Control in Distributed Systems.*

[17] Michele Guerriero, Damian Andrew Tamburri, Elisabetta Di Nitto. *Defining, Enforcing and Checking Privacy Policies In Data-Intensive Applications.* 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, 2018.

[18] Modeling Tutorial
`https://www.transentis.com/methods-techniques/models-and-metamodels/`

[19] Unified Modeling Language Tutorial
`https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/`

[20] Dimitris Karagiannis, Harald Kühn *Metamodeling Platforms* Springer-Verlag, 2002

[21] Temperature Control For Yeast Fermentation
`http://www.theartisan.net/temperature_control_baking_1.htm`

Appendices

Appendix A

Papyrus Project Creation

In order to implement the application, first of all, a new project has to be created in Eclipse. This project is going to be a Papyrus project (figure A.1).

The first field that has to be specified is the Architecture Context. Here, the default values are the right ones as an UML model is what is required to implement with Papyrus (figure A.2).

After that, the name of the project has to be defined. Due to the fact that it is going to implement an application called Great Seller, the name of the Papyrus project is going to be greatSellerApp (figure A.3).

The next step is to select the representation kind and to choose the StreamGen profile. StreamGen requires Papyrus to make a class diagram representation and, as the StreamGen project is already in the Eclipse workspace, the profile can be found browsing in the workspace (streamgen/profile/StreamUML.profile.uml). As it can be seen in the figure A.4.

Finally, we can finish with the new Papyrus project initialization (figure A.5).

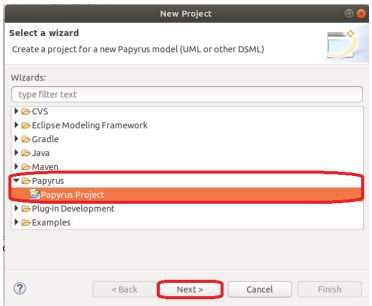

Figure A.1: New Papyrus Project

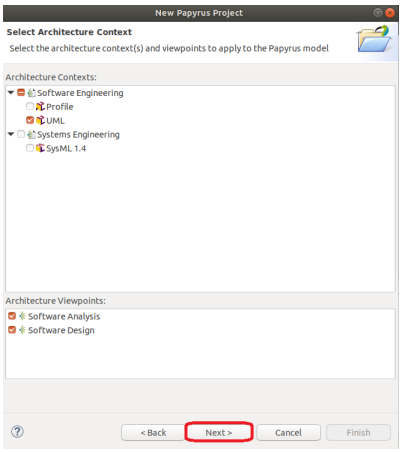

Figure A.2: Architecture Context

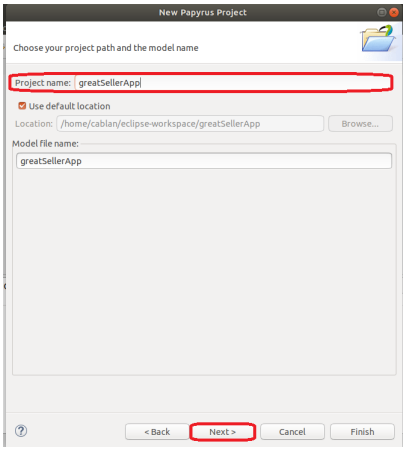

Figure A.3: Project Name Definition

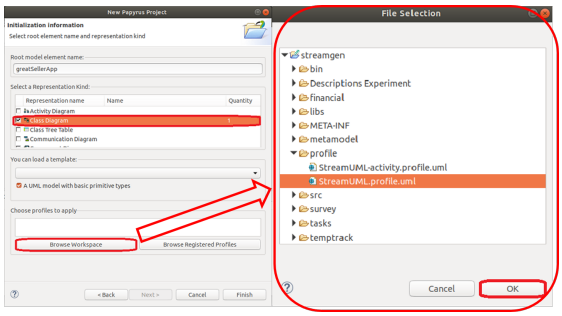

Figure A.4: Representation Kind

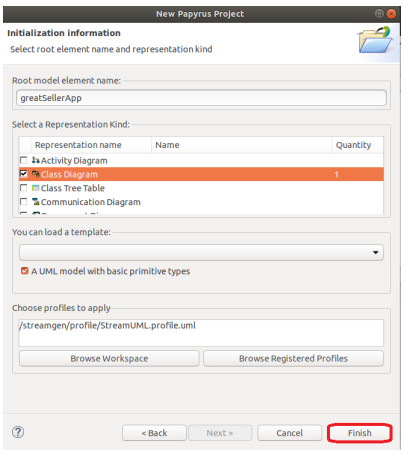

Figure A.5: Papyrus Project Definition

Appendix B

Support Python Codes

B.1 Great Seller Stock Creation

```
# Libraries
import random, shelve

# Predefined variables
numberProducts = 25 # Great Seller number of products available
priceLowerLimit = 1
priceUpperLimit = 200
greatSellerStock = []

# Binary shelf file creation
shelfFile = shelve.open('stocks')

# Creating the stock
for eachProduct in range(numberProducts):
    name = 'product' + str(eachProduct + 1)
    price = random.randint(priceLowerLimit, priceUpperLimit)
    product = {'name': name, 'price': price}
    greatSellerStock = greatSellerStock + [product]

# Writing the stock in the binary shelf file
shelfFile['greatSellerStock'] = greatSellerStock

# Closing the binary shelf file
shelfFile.close()

# Printing the generated stock in Latex table format
string = ''

for eachProduct in greatSellerStock:
    string = string + eachProduct['name'] + ' & ' + str(eachProduct['price
    string = string + '\hline' + '\n'

print(string)
```

B.2 Great Seller Input Tuples Creation

```
# Libraries
import random, sys, shelve

# Predefined variables
dataSubjectList = ['Bob', 'Carlos', 'Elisabetta', 'Michele']

if len(sys.argv) < 2:
    print('Usage: python3 tupleCreator.py [fileLength]')
    sys.exit()

# Running defined variables
fileLength = int(sys.argv[1])

# Open the source binary shelf file
shelfFile = shelve.open('stocks')
greatSellerStock = shelfFile['greatSellerStock']

# Open the destination file in write mode
fileName = 'tuples' + str(fileLength) +'File.txt'
tuplesFile = open(fileName, 'w')

# In each iteration we create a random tuple
for eachValue in range(fileLength):

    # Tuple values creation
    transactionId = str(eachValue + 1)
    dataSubject = dataSubjectList[random.randint(0, len(dataSubjectList)
    product = greatSellerStock[random.randint(0, len(greatSellerStock)-
    amount = str(product['price'])
    recipientId = product['name']

    # Tuple List creation
    tupleValues = [transactionId, dataSubject, amount, recipientId]

    # Tuple joined in a string separated by a comma
    builtTuple = ','.join(tupleValues) + '\n'

    # Tuple written in the file
    tuplesFile.write(builtTuple)

# Close the destination file
tuplesFile.close()
```

B.3 Great Seller Input Server

```
import socket, time
```

```
# Predefined variables
waitingTime = 1
fileLength = 1000

# Open the source file in read mode
fileName = 'tuples' + str(fileLength) +'File.txt'
fileName = '/home/cablan/Desktop/thesisFiles/' + fileName
tuplesFile = open(fileName, 'r')

# Create a socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Define the host and the port and listen into them
host = 'localhost'
port = 9999
s.bind((host, port))
s.listen(5)

while True:
    # Accept the requested connection
    clientSocket, addr = s.accept()
    print("Connection accepted from " + repr(addr[1]))

    # Send each tuple contained in the file
    for eachTuple in tuplesFile:
        time.sleep(waitingTime)
        print('Sending tuple: ' + '(' + eachTuple[0:-1] + ')')
        clientSocket.send(bytes(eachTuple, 'utf-8'))

    # Close the connection
    clientSocket.close()
```

B.4 Cool Analyst Temperature Chamber 1 Model

```
import random, math

import matplotlib.pyplot as plt

def sin(upperBoundary, lowerBoundary, freq, initialDelta):
    # Fix Variables
    midValue = lowerBoundary + (upperBoundary - lowerBoundary) / 2
    angularSpeed = 2 * math.pi * freq
    y = []

    for time in range(int(1/freq)):
        temp = midValue + ((upperBoundary - lowerBoundary) / 2) * math.sin
```

```
        y += [temp]

    return y

roomId = 'room1'
timeLength = 100

# Creating Plotting Lists
x = []
y = []

# Temperature Limits and Average
lowerLimit = 22
upperLimit = 29
avgTemp = lowerLimit + (upperLimit - lowerLimit) / 2


# Initial Time
time = 0

# Generating the function
while True:

    for staticValues in range(10):
        y += [avgTemp]

    randValue = random.randint(0,1)

    if randValue == 0:
        ysin = sin(avgTemp, lowerLimit, 1/16, math.pi / 2)
    else:
        ysin = sin(upperLimit, avgTemp, 1/28, math.pi * (-1) / 2)

    y += ysin
    time += 1
    staticValues = 0

    if len(y) > timeLength:
        break

for i in range(len(y)):
    x += [i]

fileObject = open('/home/cablan/Desktop/thesisFiles/inputs/room1Sin.txt'
for eachTemp in y:
    fileObject.write(roomId + ',' + str(eachTemp) + '\n')

# Creating the plot
```

```
# plotting the points
plt.plot(x, y)

# naming the x axis
plt.xlabel('Time (s)')
# naming the y axis
plt.ylabel('Temperature (Celsius)')

# giving a title to my graph
plt.title('Chamber 1')

# function to show the plot
plt.show()
```

B.5 Cool Analyst Temperature Chamber 2 Model

```
import random, math

import matplotlib.pyplot as plt

def sin(upperBoundary, lowerBoundary, freq, initialDelta):
    # Variables
    midValue = lowerBoundary + (upperBoundary - lowerBoundary) / 2
    angularSpeed = 2 * math.pi * freq
    y = []

    for time in range(int(1/freq)):
        temp = midValue + ((upperBoundary - lowerBoundary) / 2) * math.sin

        y += [temp]

    return y

roomId = 'room2'
timeLength = 120

# Creating Plotting Lists
x = []
y = []

# Temperature Limits and Average
lowerLimit = 20
upperLimit = 24
avgTemp = lowerLimit + (upperLimit - lowerLimit) / 2


# Initial Time
time = 0
```

```
# Generating the function
while True:

    for staticValues in range(30):
        y += [avgTemp]

    randValue = random.randint(0,1)

    if randValue == 0:
        ysin = sin(avgTemp, lowerLimit, 1/16, math.pi / 2)
    else:
        ysin = sin(upperLimit, avgTemp, 1/28, math.pi * (-1) / 2)

    y += ysin
    time += 1
    staticValues = 0

    if len(y) > timeLength:
        break

for i in range(len(y)):
    x += [i]


fileObject = open('/home/cablan/Desktop/thesisFiles/inputs/room2Sin.txt'
for eachTemp in y:
    fileObject.write(roomId + ',' + str(eachTemp) + '\n')

# Creating the plot
# plotting the points
plt.plot(x, y)

# naming the x axis
plt.xlabel('Time (s)')
# naming the y axis
plt.ylabel('Temperature (Celsius)')

# giving a title to my graph
plt.title('Chamber 2')

# function to show the plot
plt.show()
```

B.6 Cool Analyst Chamber 1 Server

```
import socket, time, sys
```

```
# Predefined variables
waitingTime = 1

# Choose fileLength
if len(sys.argv) == 2:
    fileLength = sys.argv[1]
else:
    fileLength = 10

# Open the source file in read mode
fileName = 'room1.txt'
fileName = '/home/cablan/Desktop/thesisFiles/inputs/' + fileName
tuplesFile = open(fileName, 'r')

# Create a socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Define the host and the port and listen into them
host = 'localhost'
port = 8888
s.bind((host, port))
s.listen(5)

# Accept the requested connection
clientSocket, addr = s.accept()
print("Connection accepted from " + repr(addr[1]))

time.sleep(5)

# Send each tuple contained in the file
for eachTuple in tuplesFile:
    print('Sending tuple: ' + '(' + eachTuple[0:-1] + ')')
    clientSocket.send(bytes(eachTuple, 'utf-8'))
    time.sleep(waitingTime)

# Close the connection
clientSocket.close()

# Close the file
tuplesFile.close()
```

B.7 Cool Analyst Chamber 2 Server

```
import socket, time, sys

# Predefined variables
waitingTime = 1
```

```
# Choose fileLength
if len(sys.argv) == 2:
    fileLength = sys.argv[1]
else:
    fileLength = 10

# Open the source file in read mode
fileName = 'room2.txt'
fileName = '/home/cablan/Desktop/thesisFiles/inputs/' + fileName
tuplesFile = open(fileName, 'r')

# Create a socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Define the host and the port and listen into them
host = 'localhost'
port = 8887
s.bind((host, port))
s.listen(5)

# Accept the requested connection
clientSocket, addr = s.accept()
print("Connection accepted from " + repr(addr[1]))

time.sleep(5)

# Send each tuple contained in the file
for eachTuple in tuplesFile:
    print('Sending tuple: ' + '(' + eachTuple[0:-1] + ')')
    clientSocket.send(bytes(eachTuple, 'utf-8'))
    time.sleep(waitingTime)

# Close the connection
clientSocket.close()

# Close the file
tuplesFile.close()
```

Appendix C

Run Configuration

In order to generate the Java codes from the Papyrus model, first of all we need to create a new Java project in Eclipse (File -> New -> Java Project). This project is going to be called 'greatSellerCodes' as it can be seen in the figure C.1.

Once the Java project is created, its structure has to be modified in order to allow to be compatible with Maven. As the figure C.2 shows, firstly the Java project is composed of a JRE System Library and a src folder. This src folder has to be deleted, leaving the Java project only with the JRE System Library as it can be seen in the figure C.3.

The next step is to create a source folder with a predefined structure as it is going to be specified now. Firstly, in the properties for the project, in the Java Build Path field, a folder has to added as the figure C.4 specifies. In the default output folder a specific structure has to be written, greatSellerCodes/src/main/java. After this is written, the changes have to be applied and then click on 'Add Folder...'.

In order to create the source folder, the java folder has to be selected and then click on 'OK' as it is shown in figure C.5. Then the changes have to be applied and, then, the window Properties for greatSellerCodes has to be closed (figure C.6).

After these steps, the project folder should look like it is shown in the figure C.7 at the Package Explorer. And then, the project is ready to convert it into a Maven project. In order to do this, going to the 'Configure' option of the project and then clicking on 'Convert to Maven Project'. Then, a POM file has to be created and it

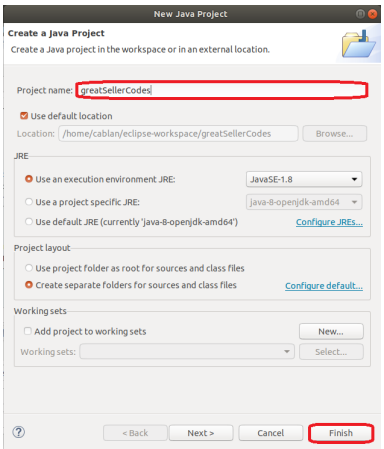

Figure C.1: Java Project Creation

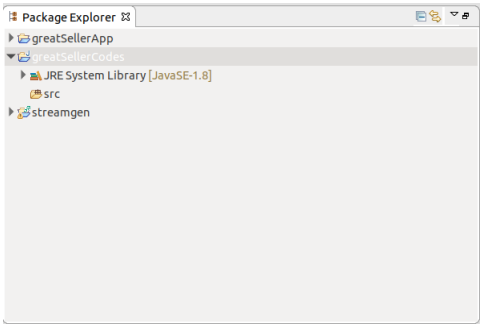

Figure C.2: Java Project Initial Structure

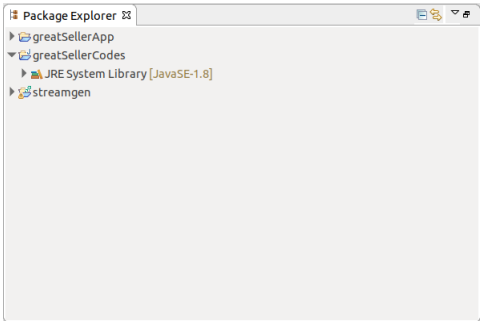

Figure C.3: Java Project Final Structure

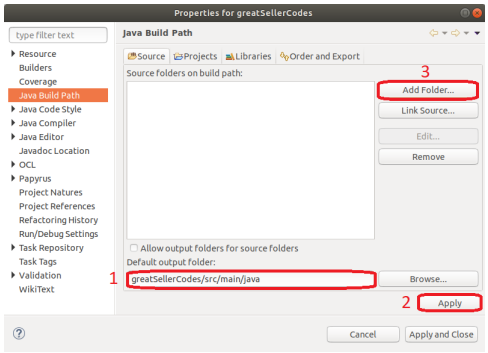

Figure C.4: Maven Project Structure Creation

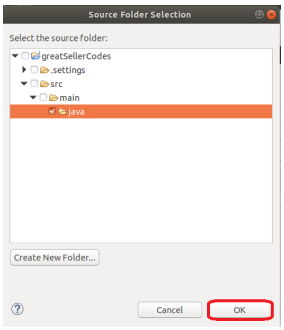

Figure C.5: Maven Project Source Folder Creation

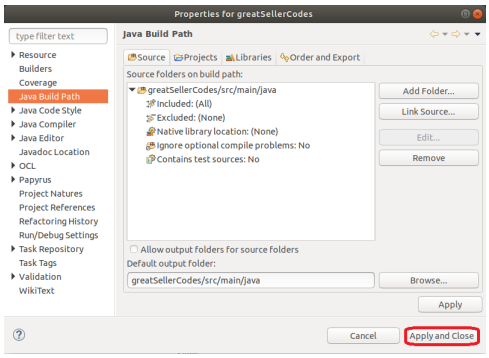

Figure C.6: Maven Project Properties

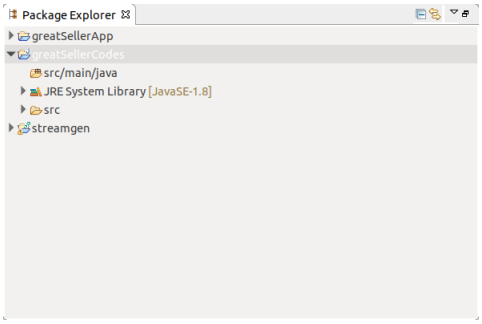

Figure C.7: Maven Project Structure

is going to be created the default one as it can be seen in the figure C.8. At this point, the structure of the project has to be the same that the one shown in the figure C.9.

Now it is time to run the configuration (Run -> Run Configurations...). In the figure C.10 is shown how this window has to be filled.

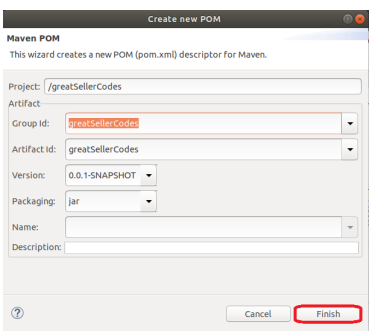

Figure C.8: POM File Creation

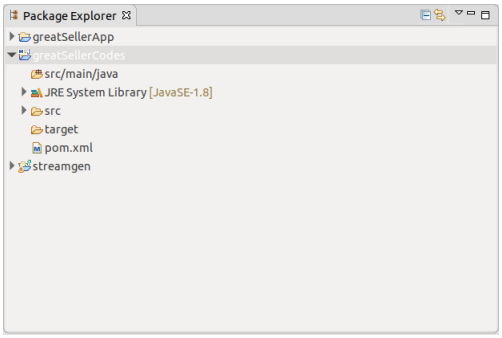

Figure C.9: Final Maven Project

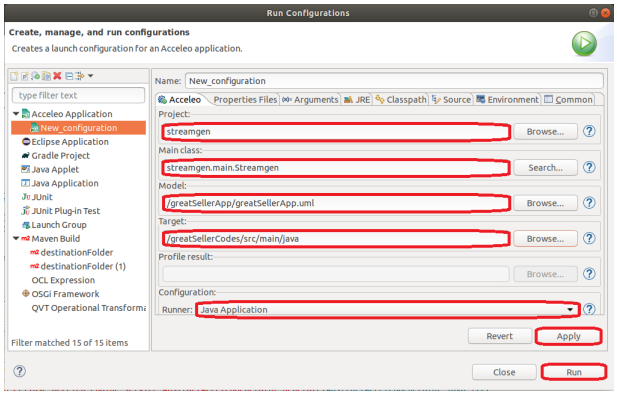

Figure C.10: Run Configuration