UNIVERSIDAD CATÓLICA BOLIVIANA "SAN PABLO"

UNIDAD ACADÉMICA REGIONAL COCHABAMBA

Departamento de Ingeniería y Ciencias Exactas

Carrera de Ingeniería de Sistemas

Estudio empírico sobre el uso de la librería *Java Stream Collections*

Perfil de Tesis de Licenciatura en Ingeniería de Sistemas

Joshua Abraham Nostas Valdivia

Cochabamba – Bolivia

Abril 2019

ÍNDICE GENERAL

1. INTRODUCCIÓN……...……..5

2. ANTECEDENTES……..5

2.1. Colecciones Stream…….……....6

2.2. Colecciones Stream y las operaciones en paralelo….……..7

2.3. Colecciones Stream y su rendimiento…….....8

3. PROBLEMA……...10

3.1. Situación Problemática……..10

3.2. Formulación del Problema……10

4. OBJETIVOS……...10

4.1. Objetivo General……10

4.2. Objetivos Específicos……..11

5. ALCANCES……11

6. LÍMITES……11

7. JUSTIFICACIÓN……..12

8. FUNDAMENTO TEÓRICO……13

8.1. Java……..13

8.2. Expresiones lambda ……..13

8.3. Java Stream Collection……14

9. PROPUESTA METODOLÓGICA……16

10. CRONOGRAMA……..17

11. BIBLIOGRAFÍA……..20

12. ANEXOS

ANEXO 1. Cronograma…….1

ÍNDICE DE TABLAS

Tabla 1 Objetivos específicos y sus respectivas acciones…….17

ÍNDICE DE FIGURAS

Figura 1 Ejemplo de función usando un for-loop……...6
Figura 2 Ejemplo de función usando la librería *stream* ……....7
Figura 3 Ejemplo de función usando *ParallelStream* …….....8
Figura 4 Ejemplo de un uso incorrecto de la librería *stream* ……....8
Figura 5 Ejemplo del uso correcto de la librería *stream* ……...9
Figura 6 Ejemplo de implementación de una expresión *lambda* …….....14
Figura 7 Cronograma de actividades …….....19

1. INTRODUCCIÓN

En la actualidad el uso de la librería *Java Stream Collections* es bastante usada entre los desarrolladores, esta librería proporciona la opción de programar de forma híbrida entre programación orientada a objetos y programación funcional. Permitiendo así realizar operaciones de orden superior en colecciones de objetos, por ejemplo, buscar un elemento en una colección, agrupar los elementos o filtrarlos por un criterio dado. Sin embargo, a pesar de las varias ventajas que puede tener el uso de estas librerías, poco se sabe de los efectos que esta puede tener al rendimiento de los programas y a la utilización de recursos.

Esta tesis de grado propone analizar el uso de esta librería en más de diez mil proyectos java almacenados en github. Este estudio empírico pretende revelar resultados del qué y cómo se utilizan las operaciones proporcionadas por *Java Stream Collections*. Se espera que los resultados sirvan de guía para que los usuarios puedan hacer un mejor uso de la librería, del mismo modo, guiar a los desarrolladores de lenguajes y herramientas de software para dar un mejor soporte a la librería en base a como esta se usa en la práctica.

2. ANTECEDENTES

En la actualidad existen más de un millón de proyectos desarrollados en el lenguaje de programación *JAVA*, solo en *GitHub* están registrados más de 611.678 repositorios desarrollados con este lenguaje. *JAVA* es un lenguaje de programación de propósito general, concurrente, orientado a objetos que fue creado con la intención de que los desarrolladores escriban el programa una vez y este pueda ser ejecutado en cualquier dispositivo, esto se conoce como: "*WORA*" lo que significa: "*Write once, Run Anywhere"* [5][6].

JAVA fue creado el año 1991 como una herramienta de programación, y en el año 1996 fue publicada la primera versión de *JAVA* como lenguaje de programación bajo el

nombre de JDK (Java Development Kit) 1.0. Desde entonces el lenguaje sufrió múltiples actualizaciones. Uno de los cambios más grandes fue implementado en la versión Java SE 8 publicada el año 2014, en esta versión se introdujo el manejo de expresiones lambda junto con la librería *Stream Collections*, esto aumentó las opciones para los desarrolladores.

2.1. Colecciones *Stream*

Los diseñadores de la interfaz API de Java han incorporado en su actualización una nueva abstracción denominada *Stream*, que permite procesar datos de modo declarativo. Más aún, los *streams* permiten aprovechar las arquitecturas de núcleos múltiples sin necesidad de programar líneas de código multiproceso. Por ejemplo, considere el siguiente código desarrollado de forma tradicional con for-loops donde se quiere sacar el promedio de los estudiantes que tienen una nota mayor a 51.

Figura 1

Ejemplo de función usando un for-loop

```
public int promedio(Estudiante[] estudiantes){
   int sumaNotas = 0;
   int estudiantesNotaMayor51 = 0;
   for(estudiante :: estudiantes){
         if(estudiante.nota() > 51){
               sumaNotas += estudiante.nota();
               estudiantesNotaMayor51 ++;
         }
   }
return sumaNotas/estudiantesNotaMayor51;
}
```

Fuente: Elaboración propia 2019

Haciendo uso de la librería *Stream*, se puede escribir el código anterior de la siguiente forma:

Figura 2

Ejemplo de función usando la libreria *stream*

```
public int promedio(Estudiante[] estudiantes){
      return promedio = estudiantes.stream()
                  .filter(estudiante -> estudiante.nota() > 51)
                  .mapToInt(estudiante -> estudiante.nota())
                  .average();
}
```

Fuente: Elaboración propia 2019

Haciendo una comparación a simple vista de ambos códigos podemos apreciar que el uso de *Streams* ayuda a que el código sea más corto y más fácil de leer pero no sabemos qué código tiene un mejor rendimiento. En el ejemplo anterior se puede ver que las operaciones pueden estar asociadas a *closures* (funciones creadas en un contexto), si bien el ejemplo solo muestra tres operaciones de orden superior (*filter*, *mapToInt*, *average*), la librería proporciona muchas otras más que pueden ser combinadas entre sí.

2.2. Colecciones *Stream* y las operaciones en paralelo

La librería *Stream* de java fue creada también con el propósito de hacer más sencillo el uso de programación en paralelo y multihilo en el lenguaje Java. Todas las operaciones de la librería pueden ejecutarse tanto de forma secuencial como en paralelo. La implementación de *Streams* en el JDK ejecuta las operaciones de manera secuencial a menos que se pida explícitamente que se ejecute en paralelo. Existen varias formas de ejecutar los *Streams* en paralelo pero la más utilizada es usando *parallelStream* en lugar de *Stream* en la creación del *stream* como se puede ver en el siguiente ejemplo:

Figura 3

Ejemplo de función usando *ParallelStream*

```
int promedio = estudiantes.parallelStream()
                .filter(estudiante ->estudiante.nota()> 51)
                .mapToInt(estudiante -> estudiante.nota())
                .average();
```

Fuente: Elaboración propia 2019

2.3. Colecciones *Stream* y su rendimiento

Desde su creación los desarrolladores de la librería *Stream* han ido mejorando el rendimiento de la misma, así también se agregaron más operaciones. Sin embargo, debido a que esta librería es nueva no se tiene mucho conocimiento sobre su uso y cómo afecta al rendimiento de los programas. Es sabido que el mal uso de las librerías de colecciones se puede pagar muy caro en términos de rendimiento. Por ejemplo, considere el siguiente ejemplo, estudiado el 2018 [34, 2,1,30,57,].

Figura 4

Ejemplo de un uso incorrecto de la librería *stream*

```
Collection<Widget> unorderedWidgest =new HashSet<>();
List<Widget> sortedWidget =
            unorderedWidget.stream()
            .sorted(Comparator.comparing(Widget::getWeight()))
            .collect(toList()).
```

Fuente: Khatchadourian, R. T., Tang, Y., Bagherzadeh, M. y Ahmed, S. (2018)

EL mismo ordena un conjunto de objetos *Widget* (*HashSet*) en base a su ancho (*weight*). Note que los conjuntos no consideran conservar el orden de inserción de los datos. Si

bien el código no tiene ningún error semántico o sintáctico, el mismo es ineficiente. En este caso particular, se sabe que cuando una colección no conserva el orden es posible ordenarla de una mejor forma en paralelo. Por lo que el código anterior se podría re-escribir como se ve a continuación:

Figura 5

Ejemplo del uso correcto de la librería *stream*

```
Collection<Widget> unorderedWidgest =new HashSet<>();
List<Widget> sortedWidget =
            unorderedWidget.parallelStream()
            .sorted(Comparator.comparing(Widget::getWeight()))
            .collect(toList()).
```

Fuente: Khatchadourian, R. T., Tang, Y., Bagherzadeh, M. y Ahmed, S. (2018)

El 2018, se propusieron algunas herramientas para detectar malos usos de la librería *Stream*, y sugerir cómo se podría realizar las mismas operaciones de forma más eficiente [1][2]. Si bien, la comunidad científica, está tratando de crear herramientas o mejorar las actuales para dar soporte a la librería *Stream*. No se sabe cuál será el impacto de estas herramientas en la práctica, porque se desconoce cómo la personas están usando la misma. Para poder medir el impacto, existen muchas preguntas que se deben responder primero, por ejemplo, ¿qué operaciones son las que se usan más? ¿cómo se combinan estas operaciones? ¿ sobre qué tipo de colecciones se aplican las operaciones?, entre otras.

Después de realizar una búsqueda en la librería digital de la ACM y IEEE usando el keyword "*Java Stream*", se encontró que hay varios artículos que proponen herramientas y/o mejoras a la librería *Java Stream*. Sin embargo, no se encontró ningún estudio respecto a cómo se utiliza esta librería en la práctica. Por otro lado, los desarrolladores tampoco saben si la forma en la que usan la librería es la más adecuada o si es la óptima.

Sin ir muy lejos, en *StackOverflow*[1] se tiene casi 6 mil preguntas relacionadas al uso de *Stream collections,* de las cuales un 2% no tienen respuesta.

3. PROBLEMA

3.1. Situación problemática

Poco se sabe del cómo los desarrolladores de Java utilizan la nueva librería de colecciones "*Java Stream Collections*", como consecuencia, los investigadores y desarrolladores desconocen como:

- Reducir el consumo de recursos con el uso de la librería.
- Agregar o mejorar las operaciones que ofrece en base a la necesidad de los desarrolladores.
- Mejorar las herramientas de programación que ayudan a analizar el mal uso de la nueva librería de colecciones.

3.2. Formulación del problema:

El escaso conocimiento del uso de la librería *Stream* limita una discusión racional acerca de cómo la librería, las herramientas y sus operaciones deberían mejorarse.

4. OBJETIVOS

4.1. Objetivo general

Elaborar un estudio empírico sobre el uso de la librería *Java Stream Collections* en la práctica.

[1] https://stackoverflow.com/questions/tagged/java-stream Foro *StackOverflow*.

4.2. Objetivos específicos

- Rastrear de manera automática secciones de código donde la libreria *Stream* es usada en una muestra de al menos 10.000 proyectos Java.
- Recolectar información de cada sección detectada para analizar el tipo de colección, las operaciones y los argumentos que se utilizaron.
- Clasificar la información del uso de la librería de colecciones en base a su rendimiento y sus operaciones.

5. ALCANCES

- Recolectar una muestra de los 10.000 proyectos mejor ranqueados de GitHub desarrollados en el lenguaje Java.
- Analizar cada proyecto y filtrar los que usan la librería *Stream* utilizando análisis estático e inferencia de tipos.
- Ordenar y categorizar las secciones de código bajo los siguientes criterios:
    - Operaciones y combinaciones
    - Uso de *Stream* y *parallelStream*
    - Tipo de dato de la fuente del *stream*

6. LÍMITES

- Solo se analizaron las secciones de código que contengan el origen del stream y las operaciones en la misma sección. Por lo que no se considerará operaciones de ejecutadas sobre el mismo *stream* en diferentes secciones de código.
- Solo se analizaran secciones de codigo de las cuales se puedan inferir el tipo del origen del *stream* automática. Ya que la detección de tipos no es posible en todos los casos.

- La categorización será semi-manual en base a la documentación proporcionada por Java. Por lo que, dicha categorización puede estar altamente sujeta a la percepción de los revisores. Para minimizar este riesgo se planea hacer categorizar cada sección con dos personas.

7. JUSTIFICACIÓN

Gran parte de la computación hoy en día realizan operaciones sobre colecciones de objetos. Los lenguajes de programación ofrecen una gran variedad de estructuras y operaciones sobre estas con diferentes propósitos. En los últimos años los lenguajes de programación han ido agregando nuevas maneras de almacenar y procesar objetos, como ser Java. En 2014, en la versión 1.8 de Java se introdujo la librería *Stream Collections* y con esta diferentes operaciones de orden superior cada una con diferentes propósitos.
Diversos *tools* han sido propuestos para el uso y manejo de *streams* en el desarrollo de software, *Tools* que soportan la migración de las colecciones tradicionales a *stream* o que sugieran una refactorización de algunas operaciones stream que se ejecutan mejor en paralelo. Surgieron discusiones sobre el rendimiento que tiene el uso de la libreria contra no usar la misma, pero muchos de estos *tools* y discusiones están basadas en suposiciones del uso de la librería.
Esto indica que el conocimiento sobre cómo los desarrolladores usan estas colecciones y operaciones, esta falta de conocimiento afecta a diferentes personas en el área:

- Investigadores, quienes no saben cómo los desarrolladores usan la librería y con esto se pierde la oportunidad de ayudarlos con problemas que puedan llegar a tener;

- Desarrolladores de herramientas de software o *tools*, ya que están publicando herramientas basados en suposiciones y no en las necesidades de los desarrolladores;
- Desarrolladores de software, quienes no saben si están usando de manera correcta o incorrecta la librería.

8. FUNDAMENTO TEÓRICO

8.1 Java.

Java es un lenguaje de programación de propósito general y orientado a objetos, que fue diseñado para correr en cualquier dispositivo por lo que sus creadores lo llamaron "WORA" que significa "write once, run anywhere"[6].

Java fue desarrollado por James Gosling y fue publicado en 1995 como un componente fundamental de la plataforma Java de Sun Microsystems [6]. En 1996 salió el JDK 1.0 desde entonces sufrió bastantes cambios y hoy en día nos encontramos con la versión JDK 11.

Un cambio interesante que sufrió fue cuando salió la versión de JDK 8.0, en esta versión agregaron la opción de programación con estilo funcional con la inclusión de las expresiones lambda y los *Streams*.

8.2 Expresiones *lambda* en el lenguaje Java.

Una expresión *lambda* es una subrutina definida que no está ligada a un identificador, estas expresiones por lo general son argumentos pasados a funciones de orden superior o son parte del resultado devuelto por una función de orden superior cuando esta necesita devolver una función. Las expresiones *lambda* están basadas en los principios del cálculo *Lambda* [11] propuestos en el trabajo de Alonzo Church.

Varios lenguajes de programación imperativos como ser: Java, c# o c++, están adaptando a las expresiones *lambda* para otorgar a los usuarios un estilo funcional de programación. En varios de estos lenguajes las expresiones lambda son tomadas como funciones anónimas, estas pueden pasarse o ser retornadas por otras funciones de orden superior.

En Java una expresión *lambda* consiste de una lista de parámetros separados por comas encerrados entre paréntesis, el símbolo de la flecha (->) y el cuerpo de la función [8], como se puede ver a continuación:

Figura 6

Ejemplo de implementación de una expresión *lambda*

```
Foo foo = parameter -> parameter + " from lambda";
```

Fuente: Elaboración propia 2019

Se pueden omitir los tipos de los datos, así como los paréntesis de la lista de parámetros cuando solo se tiene un parámetro.

8.3 *Java Stream Collections*

Java Stream Collections es un conjunto de clases que soportan la programación estilo funcional con *streams* de elementos [12, 14].

Los *streams*, a diferencia de las colecciones, no almacenan ya que un *stream* no es una estructura que almacena elementos, tiene naturaleza funcional ya que cada operación produce un resultado, ya sea otro *stream* o bien los datos. Estas operaciones se dividen en dos: operaciones intermedias y operaciones finales, la combinación de las distintas operaciones forma un *stream pipeline* [12, 14].

Un *Stream pipeline* consta de una fuente que puede ser una colección, un array, una función, entre otros, seguido de cero o más operaciones intermedias y por último una operación final [12, 14].

- **Operaciones intermedias.-**

  Las operaciones intermedias siempre retornan un *stream* por lo que es posible encadenar dos o más operaciones en los *pipelines*. Estas se dividen en operaciones con estado y sin estado:

  - Operaciones sin estado. Estas operaciones no retienen ningún estado de ningún elemento visto anteriormente, lo que permite que cada elemento sea procesado independientemente de los demás elementos.
  - Operaciones con estado. Las operaciones con estado deben incorporar el estado del elemento visto anteriormente cuando se procesan nuevos elementos, lo que quiere decir que estas operaciones deben procesar todos los datos antes de devolver un resultado [12].

- **Operaciones terminales.-**

  Las operaciones finales procesan todas las operaciones del *stream pipeline* para devolver un resultado, luego que la operación final es ejecutada el *stream pipeline* es considerado como consumido lo que quiere decir que ya no se puede usar.

  Únicamente las operaciones finales: *iterator*() y *spliterator*() otorgan una salida que permite recorridos arbitrarios cuando las operaciones intermedias no son suficientes para cumplir con la tarea [12].

9. PROPUESTA METODOLÓGICA

La presente tesis propone realizar un estudio tanto cuantitativo como cualitativo del uso de la librería *Java stream API*. Para lo cual se definió una metodología de cinco pasos basados en estudios empíricos anteriores [7, 8, 9, 10].

- **Paso 1:** Se descargara una muestra inicial de 10.000 proyectos java de github.
- **Paso 2:** Rastrear las secciones de código donde se utilice la librería *Stream*.
- **Paso 3:** Analizar manualmente una muestra de las secciones encontradas en el paso anterior, para detectar posibles falsos positivos y descartar secciones de código que no son relevantes para el estudio.
- **Paso 4:** Recolectar información de cada sección de código encontrada, principalmente, el tipo de colección, las operaciones y los argumentos que se utilizaron.
- **Paso 5:** Categorizar la información recolectada del uso de la librería en base a las operaciones que se utilizaron, el origen de los *streams* y los argumentos que se utilizaron.

El presente estudio pretende describir:

i) cómo los desarrolladores usan la librería *Java Stream* en sus proyectos

ii) como los desarrolladores introdujeron o modificaron el código relacionado con esta libreria en sus proyectos.

10. CRONOGRAMA

Tabla 1

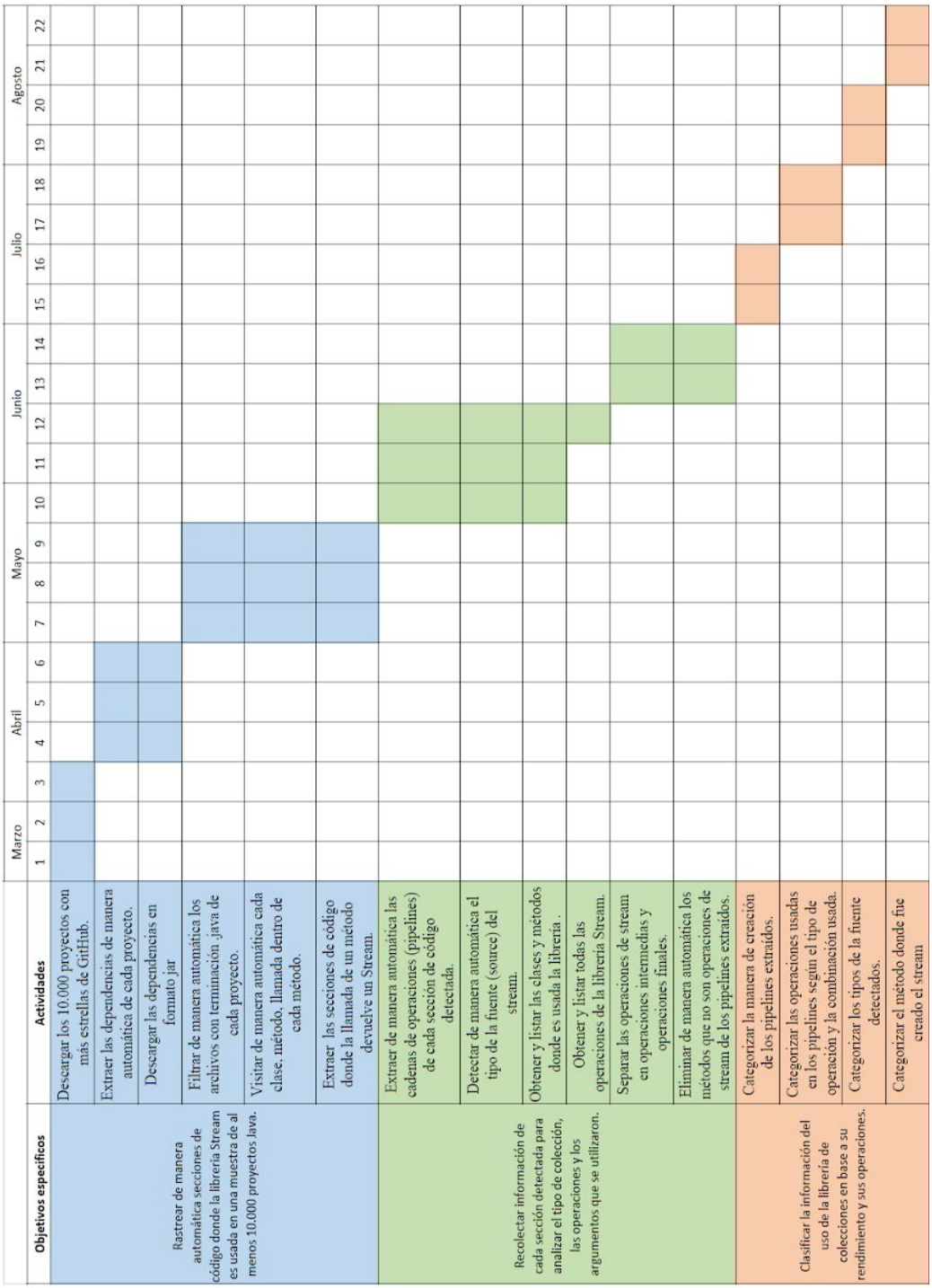
Objetivos específicos y sus respectivas acciones

Objetivos específicos	Actividades
Rastrear de manera automática secciones de código donde la libreria *Stream* es usada en una muestra de al menos 10.000 proyectos Java.	● Descargar los 10.000 proyectos con más estrellas de *GitHub*. ● Extraer las dependencias de manera automática de cada proyecto. ● Descargar las dependencias en formato *jar* ● Filtrar de manera automática los archivos con terminación .java de cada proyecto. ● Analizar de manera automática cada clase, método y llamada dentro de cada método. ● Extraer las secciones de código donde la llamada de un método devuelve un *Stream.*
Recolectar información de cada sección detectada para analizar el tipo de colección, las operaciones y los argumentos que se utilizaron.	● Extraer de manera automática las cadenas de operaciones (*pipelines*) de cada sección de código detectada. ● Detectar de manera automática el tipo de la fuente (*source*) del *stream*. ● Obtener y listar todas las operaciones de la librería *Stream*. ● Obtener y listar las clases y métodos donde es usada la librería . ● Separar las operaciones de *stream* en operaciones intermedias y operaciones finales. ● Eliminar de manera automática los métodos que no son operaciones de *stream* de los pipelines extraídos.

Clasificar la información del uso de la librería de colecciones en base a la fuente del *stream,* sus operaciones y su manera de creación.	● Categorizar la manera de creación de los *pipelines* extraídos. ● Categorizar las operaciones usadas en los *pipelines* según el tipo de operación y la combinación usada. ● Categorizar los tipos de la fuente detectados. ● Categorizar el método de creación del *stream*

Figura 7

Cronograma de actividades



Fuente: Elaboración propia 2019

11. BIBLIOGRAFÍA

[1] Khatchadourian, R. T., Tang, Y., Bagherzadeh, M. y Ahmed, S. (2018) A Tool for Optimizing Java 8 Stream Software via Automated Refactoring. *City University of New York (CUNY) CUNY Academic Works*

[2] Khatchadourian, R. T., Tang, Y., Bagherzadeh, M. y Ahmed, S. (2018) Poster: Towards safe refactoring for intelligent parallelization of Java 8 streams. *City University of New York (CUNY) CUNY Academic Works*

[3]Langer, Angelika (2015) Java performance tutorial - How fast are the Java 8 streams?. *JAX Magazine* https://jaxenter.com/java-performance-tutorial-how-fast-are-the-java-8-streams-118830.html

[4] Müller, Michael (2016) Java Lambdas and Parallel Streams. *Apress*

[5] Wong, William (2002) "Write Once, Debug Everywhere" electronicdesign.com https://www.electronicdesign.com/embedded/write-once-debug-everywhere

[6]SUN MICROSYSTEMS (1996) JavaSoft ships Java 1.0 https://tech-insider.org/java/research/1996/0123.html

[7] CALLAU, Oscar, ROBBES, Romain, TANTER, Eric y ROTHLISBERGER, David (2011). *How (and Why) Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk.* (s.l.)(s.e)

[8] MANZINANIAN, Davood, KETKAR, Ameya, TSANTALIS, Nikolaos y DIG, Danny (2017). *Understanding the use of lambda expressions in Java.* (s.l.)(s.e)

[9] SANDOVAL ALCOCER, Juan Pablo y BERGEL, Alexandre (2015). *Tracking down performance variation against source code evolution.* (s.l.)(s.e)

[10] ROBBES, Romain, ROTHLISBERGER, David y TANTER, Eric (2014). *Object-oriented software extensions in practice.* Nueva York: Springer Science+Business Media.

[11] CHURCH, Alonzo (1932). *A Set of Postulates for the Foundation of Logic. Annals of Mathematics.* Princeton: Mathematics Department, Princeton University.

[12] ORACLE (2018). *Package. Java.util.stream.* https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html

[13] SANDOVAL ALCOCER, Juan Pablo, DENKER, Marcus, BERGEL, Alexandre y ACURANA, Yasett (2016). *Dinamically Composing Collection Operations through Collections Promises.* Proceedings of International Workshop on Smalltalk Technologies.
[14] MULLER, Michael (2016). *Java lambdas and parallel Streams.* Apress.

ANEXO 1.
Cronograma

		Marzo			Abril			Mayo				Junio				Julio				Agosto			
Objetivos especificos	**Actividades**	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Rastrear de manera automática secciones de código donde la librería Stream es usada en una muestra de al menos 10.000 proyectos Java.	Descargar los 10.000 proyectos con más estrellas de GitHub.																						
	Extraer las dependencias de manera automática de cada proyecto.																						
	Descargar las dependencias en formato jar																						
	Filtrar de manera automática los archivos con terminación .java de cada proyecto.																						
	Visitar de manera automática cada clase, método, llamada dentro de cada método.																						
	Extraer las secciones de código donde la llamada de un método devuelve un Stream.																						
Recolectar información de cada sección detectada para analizar el tipo de colección, las operaciones y los argumentos que se utilizaron.	Extraer de manera automática las cadenas de operaciones (pipelines) de cada sección de código detectada.																						
	Detectar de manera automática el tipo de la fuente (source) del stream.																						
	Obtener y listar las clases y métodos donde es usada la librería .																						
	Obtener y listar todas las operaciones de la librería Stream.																						
	Separar las operaciones de stream en operaciones intermedias y operaciones finales.																						
	Eliminar de manera automática los métodos que no son operaciones de stream de los pipelines extraídos.																						
Clasificar la información del uso de la librería de colecciones en base a su rendimiento y sus operaciones.	Categorizar la manera de creación de los pipelines extraídos.																						
	Categorizar las operaciones usadas en los pipelines según el tipo de operación y la combinación usada.																						
	Categorizar los tipos de la fuente detectados.																						
	Categorizar el método donde fue creado el stream																						