Università di Catania

Make Me Resilient

Ingegneria dei sistemi distribuiti

Università di Catania

Anno 2021/2022

Aldo Fiorito

Simone Scionti

Prof. Emiliano Tramontana

27 agosto 2022

Sommario

Introduzione e scopo del progetto

MakeMeResilient nasce con l'esigenza di dover creare un automatismo per analizzare un applicazione e trovare, per poi correggere, i suoi punti critici. Introdurre dei miglioramenti relativi alla robustezza del codice permetterà di innestare nuove parti logiche che si occuperanno di fornire una maggiore resistenza ai guasti.

Solitamente, molte delle applicazioni odierne fanno utilizzo di chiamate e procedure remote che utilizzano meccanismi basati sulla comunicazione in rete. Questa tipologia, come è ben noto, per quanto sia utile, espone all'applicativo notevoli complicazioni e possibili punti di fallimento, poiché la rete di se è lenta e inaffidabile.
Inoltre, oggi, molti degli applicativi nascono con l'essere distribuiti, questo non permette di avere una totale visione e controllo di ci che succede dall'altra parte.
Per questo motivo, è utile avere dal proprio lato dei meccanismi di resilienza ai guasti che permettono di far funzionare il proprio applicativo anche quando qualcosa all'esterno non va.

Requisiti di M.M.R

(1) Maven

(2) javaParser

(3) Java 8+

(4) MavenXpp3Reader

Tecnologie utilizzate

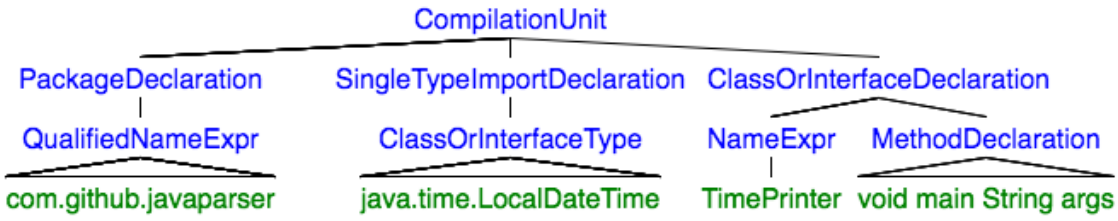
MakeMeResilient è stato sviluppato in java attraverso l'ausilio di javaParser per innestare codice aggiuntivo su applicativi distribuiti scritti in java. Per poterne mostrare l'utilizzo è stato costruito un progetto che simula un applicazione di rete distribuita.
JavaParser è un parser di codice realizzato adHoc che permette di analizzare, trasformare e generare dei costrutti ed istruzioni nuove o esistenti in base alle proprie esigenze.

Essendo un vero e proprio parser, verrà generato un Abstract Syntax Tree (AST) del codice analizzato per poter manipolare ogni singolo nodo dell'albero. Tramite la manipolazione di esso verrà implementata l'intera logica di immissione del nuovo codice e la modifica dei nodi esistenti che verranno marcati come criceti e quindi da dover proteggere.

Per fare un esempio di codice e AST generato

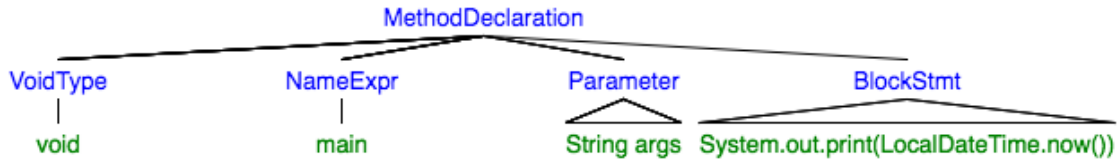
```java
package com.github.javaparser;

import java.time.LocalDateTime;

public class TimePrinter {

    public static void main(String args[]){
        System.out.print(LocalDateTime.now());
    }
}
```

AST generato da javaParser:

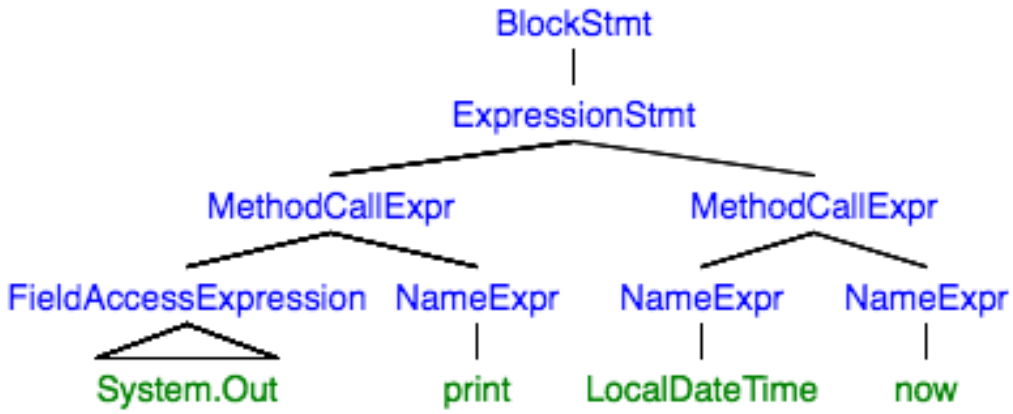


Come è possibile vedere, ogni radice di alto livello indica una declaration che sia a livello di import, package e classi.

Sull'ultimo ramo e in particolare sulle ClassOrInterfaceDeclaration cercheremo di manipolare le singole parti



E andando a scendere tra le varie branch della compilation unit troveremo le singole espressioni



Per quanto riguarda la comunicazione di rete è stata implementa attraverso l'uso di RMI (remote metodo invocation). Questa libreria per java semplifica all'utilizzatore i fini dettagli della comunicazione via rete, rendendo trasparente il sistema distribuito di oggetti e le chiamate che vi stanno sotto saranno notevolmente semplificate. Di base il funzionamento di RMI distribuito è basato su un'entità Server e una client.

Il Server RMI implementa un'interfaccia relativa ad un particolare oggetto RMI e registra tale oggetto nel Java RMI Registry.

Il Java RMI Registry è, semplicemente, un processo di tipo daemon che tiene traccia di tutti gli oggetti remoti disponibili su un dato server.

Il Client RMI effettua una serie di chiamate al registry RMI per ricercare gli oggetti remoti con cui interagire (attraverso l'operazione di Lookup all'interno della rete).

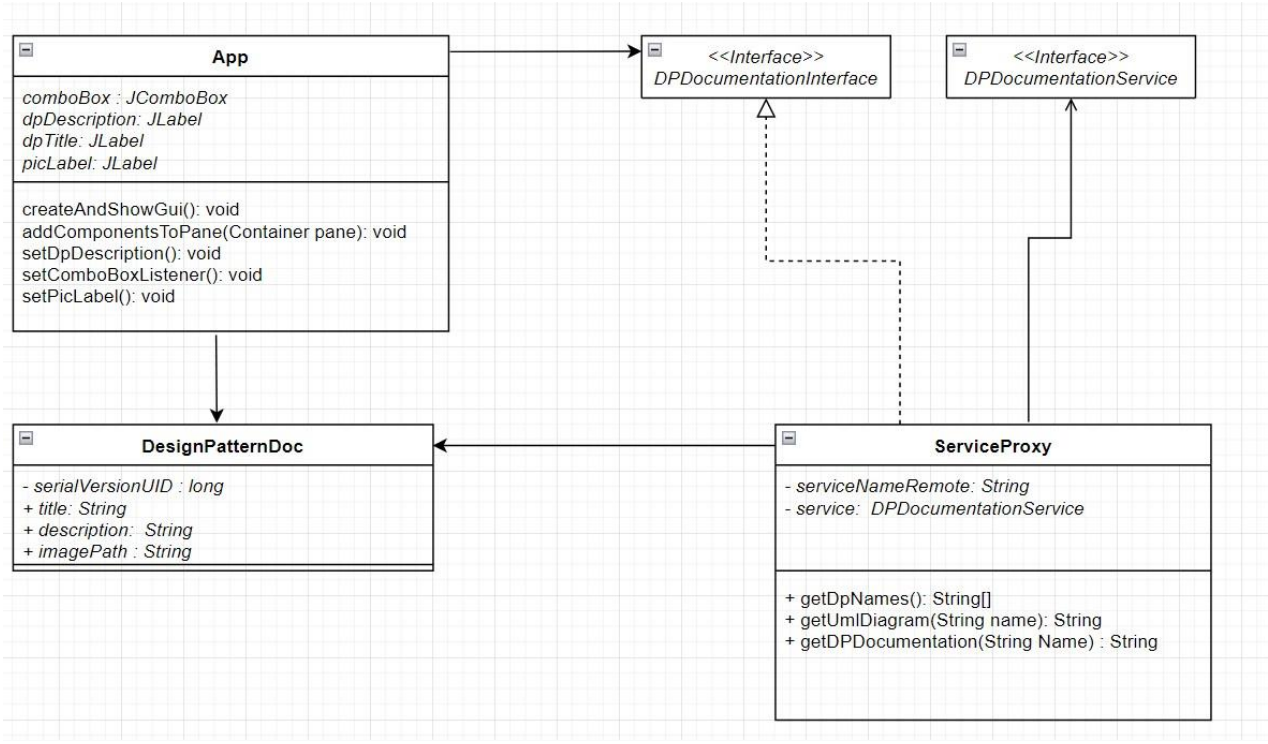
Altro tool utilizzato è stato quello di MavenXpp3Reader, un parser dedicato a maven e alla gestione del file pom.xml.

Questo ci tornerà utile per immettere le dipendenze utili e necessarie al nostro intento per poter integrare effettivamente le modifiche.

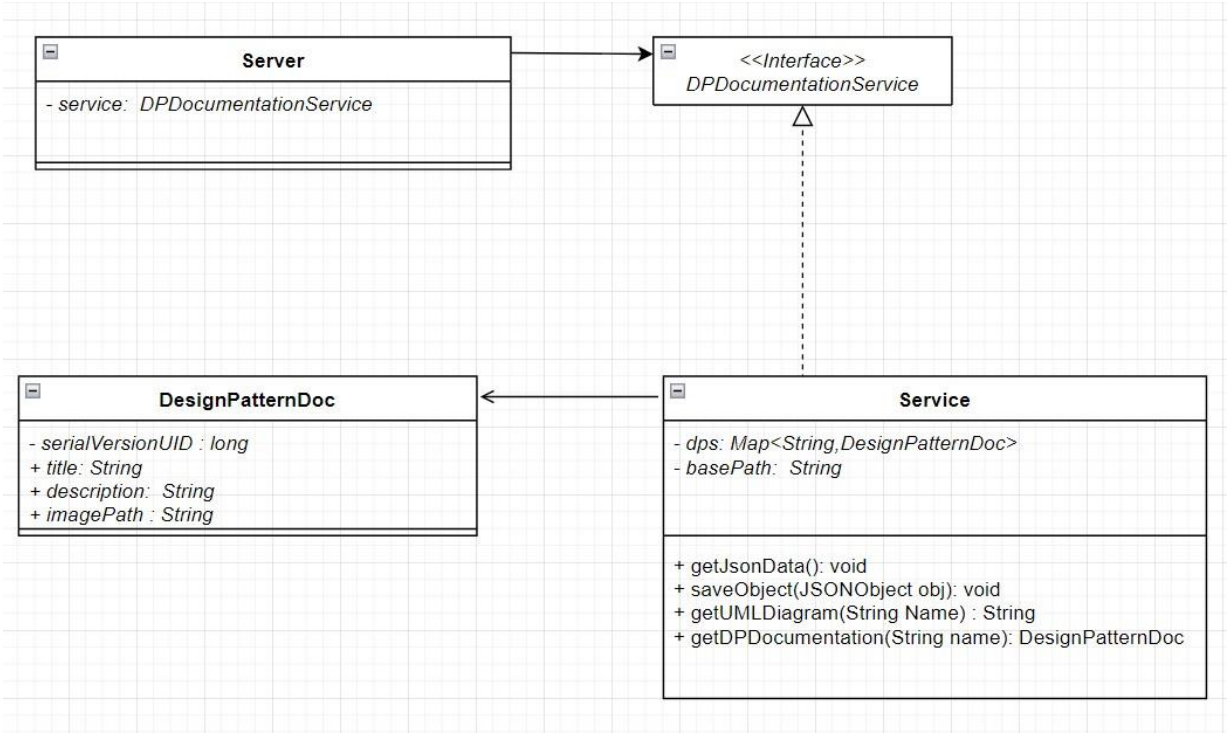
Struttura Applicativo

Diagramma UML delle classi

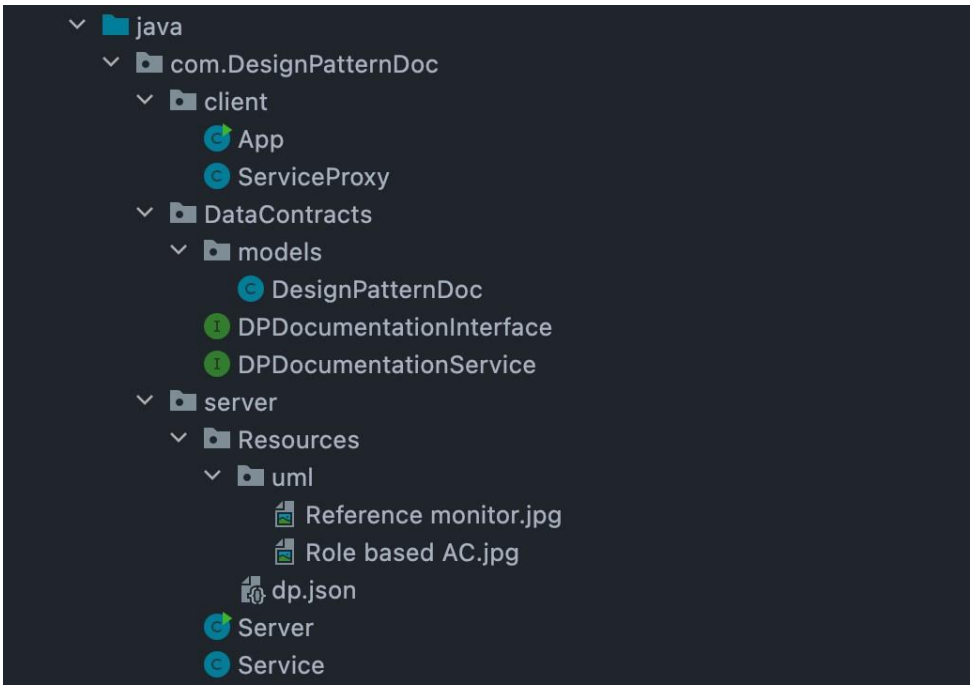
Client Side



Server Side


Server
- service: DPDocumentationService
<<Interface>>
DPDocumentationService
DesignPatternDoc
- serialVersionUID : long
+ title: String
+ description: String
+ imagePath : String
Service
- dps: Map<String,DesignPatternDoc>
- basePath: String
+ getJsonData(): void
+ saveObject(JSONObject obj): void
+ getUMLDiagram(String Name) : String
+ getDPDocumentation(String name): DesignPatternDoc

Project Folder



Il progetto è composto da tre packages principali: client, DataContracts e server.

Package client:

- App, emula il client come utilizzatore finale. Crea una interfaccia visuale attraverso Swing e utilizza i metodi offerti da Service attraverso ServiceProxy.
- ServiceProxy, fa da ponte tra App e la parte nel server, si occupa di cercare in rete il servizio da usare attraverso RMI.

Nasconde al client e gestisce le eventuali eccezioni lanciate Package DataContracts:

- DesignPatternDoc, è una classe nata con lo scopo di essere un DTO per l'applicativo. All'interno implementa Serializable.

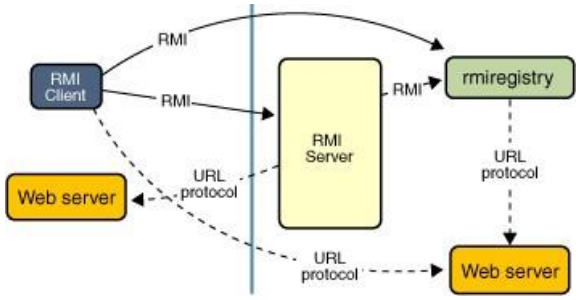
- DPDocumentationInterface.
- DPDocumentationService, estende l'interfaccia Remote. Conosciuta da entrambe le parti per permettere la comunicazione di rete

Package server:

- Server, definisce un url insieme ad una porta in cui avviare il servizio remoto. Su questo URL viene associato attraverso, un'operazione di binding, un istanza di tipo Service
- Service, implementa e offre alcune operazioni desiderate dal client.

Logica Applicativo

- Setup del server in rete di RMI



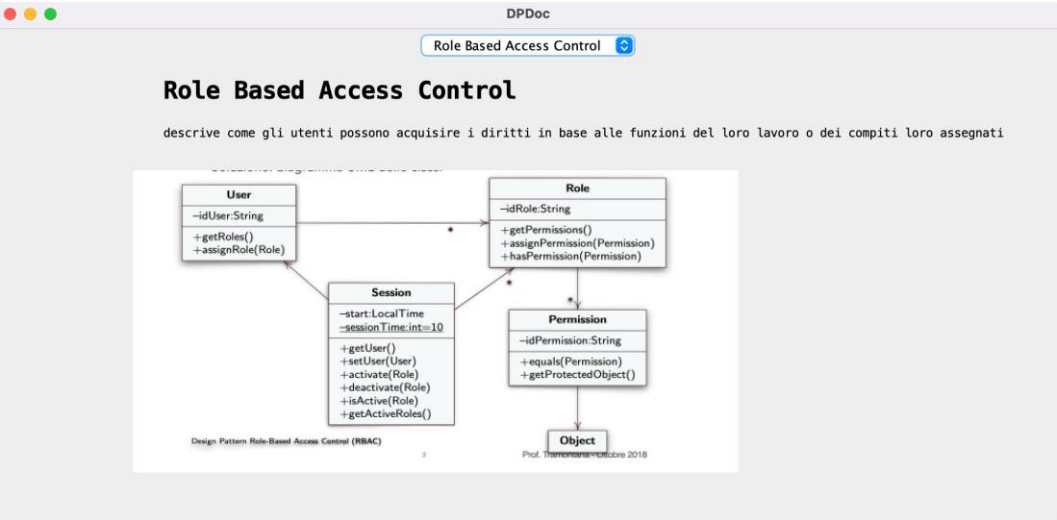
Il registry di RMI è raggiungibile sull'indirizzo 127.0.0.1 (localhost) ed in particolare, sulla porta 9100 viene fatto il binding con il l'istanza di Service creata.

Adesso il Server espone i metodi invocabili da remoto.

```
System.setProperty("java.rmi.server.hostname","127.0.0.1");
String URL = "rmi://127.0.0.1:9100//Server";

DPDocumentationService Service = new Service();

try{
    LocateRegistry.createRegistry( port: 9100);
    Naming.rebind(URL,Service);
```

- Client

Il client è una semplice applicazione scritta in Java con l'ausilio del framework Swing, utilizzato per fornire all'utente una UI minimale, che invoca i metodi offerti dall'oggetto remoto sulla base delle interazioni dell'utente.



L'applicativo offre la possibilità di consultare la documentazione relativa ai Design Patterns studiati nel corso di Ingegneria dei Sistemi Distribuiti in modo interattivo, attraverso la selezione del design pattern desiderato.

La selezione di uno specifico design pattern richiamerà alcuni metodi del serviceProxy (classe che in modo trasparente comunica con il server tramite design pattern RMI). Questa classe è l'unica che è esegue delle chiamate remote, per tanto è responsabile di gestire eventuali errori. Ci significa che questa è la classe che l'applicativo sviluppato con JavaParser dovrà individuare come quella su cui intervenire per aumentare la robustezza dell'applicativo.

Di seguito i metodi a disposizione del client.

```java
package com.DesignPatternDoc.DataContracts;
import com.DesignPatternDoc.DataContracts.models.DesignPatternDoc;

public interface DPDocumentationInterface{
    String[] getDPNames();

    String getUMLDiagram(String name);

    DesignPatternDoc getDPDocumentation(String name);
}
```

- Analisi e innesto codice con Javaparser

Prima di tutto, per capire quale codice e classe andare a modificare bisogna capire quale classe estende l'interfaccia RMI denominata Remote all'interno dell'applicativo. Per fare questo, una volta passati al parere tutti i file all'interno della cartella java, andremo a cercare dentro ogni CU (compilation unit del file analizzato) ogni denominazione della classe e di questa, qualora fossero presenti, andremo a prendere le classi che estende.

```
all_cu.forEach(cu->{
    if(!cu.getResult().isPresent()) return;
    cu.getResult().get().findAll(ClassOrInterfaceDeclaration.class).forEach(c -> {
        NodeList<ClassOrInterfaceType> interfaceTypes = c.getExtendedTypes();
        if(interfaceTypes.size() == 0) return;
        if(interfaceTypes.get(0).toString().equals("Remote")){
            Optional<String> qualifiedName = c.getFullyQualifiedName();
            qualifiedName.ifPresent(s -> interfaceToLookup.add(s.toString()));
        }
    });
});
```

Se questa è di tipo Remote, come da nostro intento abbiamo trovato il target interessato "*DPDocumentationService*"

```
FOUND REMOTE INTERFACE Optional[com.DesignPatternDoc.DataContracts.DPDocumentationService]
```

A questo punto, dato l'obiettivo di proteggere la parte client dai fallimenti di rete bisognerà cercare e capire come questa interfaccia venga usata.

Prendendo coscienza di come funziona Remote e di come il proxy cerca il servizio nel server attraverso l'operazione di Lookup l'operazione che cerchiamo all'interno delle nostre classi lato client è del tipo

```
service = (DPDocumentationService) Naming.lookup(serviceNameRemote);
```

Essa pu essere identificata come un espressione di cast o di assegnamento, ma per la corretta individuazione avremmo bisogno di un'altra componente del javaParser ovvero il symbolSolver in grado di risolvere a run-time i tipi delle istanze coinvolte.

Per poter utilizzare il plugin è necessario aggiungere alcune dipendenze su maven.

```xml
<dependency>
  <groupId>com.github.javaparser</groupId>
  <artifactId>javaparser-symbol-solver-core</artifactId>
  <version>3.24.4</version>
</dependency>
```

Aggiunta la dipendenza dovremmo configurare il risolutore con diverse tipologie di resolver. Quelle di nostre interesse saranno le "ReflectionTypeSolver" e "JavaParserTypeSolver". Quest'ultima prenderà in input il progetto per cercare le tipologie di dato non primitive di java, ma quelle user-defined.

Una volta individuata l'espressione, siamo interessati all'oggetto a cui è assegnata l'espressione di destra. Effettuando un cast dell'espressione a una

```
CombinedTypeSolver combinedSolver = new CombinedTypeSolver();
combinedSolver.add(reflection);
combinedSolver.add(typeSolver);

JavaSymbolSolver symbolSolver = new JavaSymbolSolver(combinedSolver);
StaticJavaParser
        .getConfiguration()
        .setSymbolResolver(symbolSolver);

return symbolSolver;
```

di assegnamento e possibile ricavarne il target, nel nostro caso "service"

A questo punto basterà individuare tutte le chiamate effettuate da service, il nostro oggetto remoto all'interno della classe Proxy.

Per fare questo visitiamo nuovamente AST relativo alla classe e individuiamo tutte le chiamate effettuate da l'oggetto remoto service attraverso lo scope della espressione. Su questi punti andremo a modificare e innestare nuovo codice.

```
localCu.get().findAll(MethodCallExpr.class).forEach(e -> {
    if(e.getScope().get().equals(target)) {
        System.out.println("need to be changed:"+e.toString());
        lines.add(e.getRange().get().begin.line);
    }
});
```

Partendo dalla stessa CU e dalle linee ricavate andremo ad esplorare nuovamente l'ast prendendo in considerazione tutti i metodi ed eventualmente il costruttore e di questi poi, il loro blocco di istruzione.

```
NodeList<BodyDeclaration> members = typeDec.getMembers();
Optional<BlockStmt> localBlock;
for(BodyDeclaration member: members){
    if (member instanceof ConstructorDeclaration) {
        localBlock = Optional.ofNullable(((ConstructorDeclaration) member).getBody());
    }
    else if (member instanceof MethodDeclaration) {
        localBlock = ((MethodDeclaration) member).getBody();
    }
    else continue;

    if(localBlock.isPresent()) {
        System.out.println("\tFound Block");
        List<Integer> linesToRemove = new ArrayList<>();
        InsertTimemout(lines,linesToRemove, localBlock);
        lines.removeAll(linesToRemove);
    }
}
```

Ogni blocco, qualunque esso sia è formato da vari statements, essi possono essere istruzioni semplici, while, for, blocchi try catch e cosi via.
Solitamente, le chiamate di service dovrebbero essere inserite all'interno di un blocco try catch per lanciare eventualmente eccezioni.

Scorrendo quindi i vari statements e prendendo di ognuno inizio e fine linea possiamo individuare i nostri confini e di qui confrontare con le linee interessate alla modifica. Individuato lo statement giusto e il corpo del body da modificare possiamo innestare il nuovo codice che usa Failsafe e politiche di Retry per correggere i possibili failure points e rendere resiliente l'applicativo client. Per aggiungere un nuovo blocco possiamo utilizzare il metodo offerto da javaParser: parseStatement, che, in maniera autonoma, potrà rendere

```
if(isReturnExp) bs.addStatement(parseStatement("return Failsafe.with(retryPolicy).get(() ->"+localExpr+");"));
else
    bs.addStatement(parseStatement("Failsafe.with(retryPolicy).get(() ->"+localExpr+");"));
```

compatibile e aggiungere codice attraverso una stringa.

Failsafe e RetryPolicy

Failsafe è una libreria messa a disposizione a partire da java 8 utile per gestire i failure case con diversi pattern e policy da comporre.
Questa libreria offre svariati tipi di policy, che sono in grado di gestire specifiche eccezioni lanciate durante la comunicazione di rete. L'operazione da eseguire al verificarsi di una o più eccezioni dipende dal tipo di policy adottata.

Per irrobustire l'applicativo client, è stata utilizzata una policy di Wait and Retry con exponential backoff. Ci significa che, al verficarsi di una eccezione durante la comunicazione, verrà rieseguita la stessa chiamata RMI, attendendo un intervallo di tempo che cresce esponenzialmente al crescere delle eccezioni consecutive verificatesi.

L'applicativo raggiunge quindi un livello di robustezza molto maggiore, essendo in grado di sopperire a problemi di comunicazione sporadici.

L’esecuzione del tool MakeMeResilient ha l’effetto seguentemente descritto su tutti i metodi che effettuano chiamate RMI:

Prima dell’esecuzione di MakeMeResilient:

```java
@Override
public String[] getDPNames() {
    try{
        return service.getDPNames();
    }catch(Exception e) {
        System.err.println(e);
        return new  String[] {"ciao"};
        //throw e; //add timer
    }
}
```

Dopo l’esecuzione:

```java
try {
    RetryPolicy<Object> retryPolicy = RetryPolicy.builder().handle(Exception.class).withBackoff(
    return Failsafe.with(retryPolicy).get(() -> service.getDPNames());
} catch (Exception e) {
    System.err.println(e);
    return new String[] { "ciao" };
    // throw e; //add timer
}
```

Robustezza e caratteristiche implementazione parser

La robustezza del parser è fondata su diverse euristiche necessarie al corretto funzionamento e implementazione di javaParser.

Questa vede in primis l’operazione di cast che lo sviluppatore è necessario che effettui quando chiama l’operazione di Lookup attraverso rmi.

Il casting è esplicito, ed è valorizzato con l’interfaccia che estende remote.

Questo ci permette di recuperare il target su cui l’assegnamento è effettuato, non importa se esso sia derivato da un return o meno, il singolo assegnamento e l’operazione di casting sono essenziali per individuare le chiamate che l’oggetto remote effettuerà.

Il metodo “CheckForCommonExpression” controllerà inoltre che i duplicati, riferiti alla stessa operazione e non saranno considerati durante l’analisi dei codice.
Qualsiasi altra operazione che non conterrà un cast o un assegnamento non porterà ad un riconoscimento del target rmi interessato.
Parlando invece delle chiamate critiche, o possibile failure points, essi sono stati individuati partendo da un principio di statement che parte dall’inserimento della chiamata a metodo dentro un costrutto try-catch, in maniera tale da gestire l’eccezione di rete, qualora questa dovrebbe riportare qualche errore.
Tuttavia, questo è a discapito del programmatore e del suo buon senso nell’inserite una chiamata possibile ad eccezioni dentro un valido blocco.
Attualmente non è prevista l’individuazione e la correzione su altri tipi di statements.
Un controllo incrociato per individuare il corretto punto si ha attraverso le linee da modificare e i range dello statement individuato. Attraverso la logica di “linea inclusa” si è sicuri di aver individuato lo statement (tra i multipli disponibili) necessario al cambiamento. Una volta visitato e modificato un costrutto, verrà marcato come “visited”, questo consentirà di modificare più failures points attraverso una sola passata e di non duplicare la costruzione delle policy della libreria failsafe.

PomInject

È una semplice classe nata per immettere le dipendenze necessarie al corretto funzionamento di Failsafe all’interno dell’applicativo. Per fare questo è necessario modificare dunque gli artefatti maven all’interno del file pom.xml.
La modifica della struttura del file xml è stata possibile grazie all’ausilio di un tool di terze parti chiamato “MavenXpp3Reader”, che ha il compito, così come javaParser di fornire una parserizzazione del file in un corrispettivo albero/modello.
Una volta generato la struttura, partendo da un path di riferimento, cercheremo di aggiungere il gruppo, l’artefatto e la versione richiesta al funzionamento di Failsafe.
Questo sarà possibile attraverso la creazione di un nuovo nodo di tipologia “Dependency” da innestare all’interno dell’albero come da codice sotto mostrato.

```java
public void addFailSafeDep(){
    if(!this.model.getDependencies().stream().filter(dep-> dep.getArtifactId().equals("failsafe")).collect(Collectors.toList()).isEmpty()) return;
    Dependency failSafe = new Dependency();
    failSafe.setGroupId("dev.failsafe");
    failSafe.setArtifactId("failsafe");
    failSafe.setVersion("3.2.4");
    this.model.addDependency(failSafe);
    System.out.println("ADDED failsafe");
}
```

Finita la modifica dell'albero potremmo riscrivere nuovamente il file pom.xml attraverso il writer messo a disposizione da MavenXpp3Reader, che ci permetterà di fare l'overwrite del vecchio pom con un nuovo secondo il nuovo modello precedentemente modificato.

La modifica del file pom.xml è elemento necessario per permettere alla classe di ContractDiscovery di importare la libreria di Failsafe e di aggiungerla alla CU modificata, affinché il compilatore possa riconoscere le nuove dipendenze.

```java
private static void AddImportDeclaration(Optional<CompilationUnit> localCu) {
    localCu.get().addImport(new ImportDeclaration( name: "dev.failsafe.Failsafe", isStatic: false, isAsterisk: false));
    localCu.get().addImport(new ImportDeclaration( name: "dev.failsafe.RetryPolicy", isStatic: false, isAsterisk: false));
    localCu.get().addImport(new ImportDeclaration( name: "java.time.temporal.ChronoUnit", isStatic: false, isAsterisk: false));
}
```

Contract discovery

Descrizione metodi della classe

SelectProjectsPath

Attraverso la libreria JFileChooser andremo a selezionare la cartella /java del progetto interessato da parserizzare e il relativo file pom.xml in cui voler immettere le dipendenze.

AddImportDeclaration

Aggiunge alla CU analizzata gli import di Failsafe, in particolare modo le classi

- dev.failsafe.Failsafe

- dev.failsafe.RetryPolicy
- java.time.temporal.ChronoUnit

reWriteClass

Finestra di dialogo in cui viene domandato all'utente se voglia fare l'overwrite o meno della classe vecchia con la classe nuova, creata dopo l'esecuzione del parser e dei nuovi innesti di codice.

CheckForCommonExpression

Metodo che effettua un controllo su tutte le possibili ed eventuali assegnazioni all'oggetto remote, in tal caso rimuove l'espressione doppione dai visitabili.

GetLinesUsage

Trovata l'assegnazione target con l'interfaccia desiderata, si va a riempire una struttura con tutti gli utilizzi (chiamate a metodo) dell'oggetto interessato (anche detto "scope")

LookupForRemoteInterface

Ciclo che scansione tutte le dichiarazioni di classe all'interno del progetto java e va ad estrarre le classi/interfacce che estendono Remote (rmi)

SetupSimbolSolver

Metodo che si occupa di fare un setup del symbol solver di javaParser. Esso tornerà utile quando, prese tutte le espressioni di assegnazione e di cast all'interno di una classe, sarà necessario risolverne il tipo per trovare, secondo quando tornato da *LookupForRemoteInterface*.

InnestCode

Punto d'ingresso per recuperare tutti i metodi e i costruttori di ogni classe e di questi prenderne il corpo (body). Le linee interessate alla modifica saranno sicuramente dentro uno o più di questi costrutti.

```
NodeList<BodyDeclaration> members = typeDec.getMembers();
Optional<BlockStmt> localBlock;
for(BodyDeclaration member: members){
    if (member instanceof ConstructorDeclaration) {
        localBlock = Optional.ofNullable(((ConstructorDeclaration) member).getBody());
    }
    else if (member instanceof MethodDeclaration) {
        localBlock = ((MethodDeclaration) member).getBody();
    }
    else continue;
```

InsertTimemout

Si occupa di recuperare tutti gli statements all'interno di un blocco presente dentro un body. Con blocco intendiamo qualsiasi istruzioni presente all'interno di una parentesi graffa aperta e chiusa { }.

Individuatone uno, recuperiamo tutti gli statements presenti in esso, in particolare quelli della tipologia "try-catch" secondo quanto premesso nelle pagine sopra.

Di questo, se la linea interessata al cambiamento ricade nel suo range di linee, allora dovremmo modificare il blocco dello statement analizzato.

Inserendo due parti nuove: la policy di failsafe e la chiamata dell'oggetto remote wrappata all'interno della stessa policy

```java
int beginLine = s.getRange().get().begin.line;
int endLine = s.getRange().get().end.line;
for (Integer line:lines){
    System.out.println("\t\t\tAnalyzing line:"+line);
    boolean lineIsInStatement = ((endLine >= line) && (line >= beginLine));
    if( lineIsInStatement ) {
        System.out.println("\t\t\t\tINSERTING POLICY");
        linesToRemove.add(line);
        BlockStmt bs = s.findFirst(BlockStmt.class).get();
        if(!alreadyVisited.containsKey(bs.getRange().get().begin.line)) {
            AddRetryPolicy(alreadyVisited, bs);
        }
```

AddRetryPolicy

Aggiunge al blocco dello statement la policy di failsafe, creando l'oggetto retryPolicy con attesa di retry che segue un esponential backoff.

I tentativi massimi sono impostati a 3.

L'intera costruzione della policy è inserita all'interno di una chiamata parseStatement, proprietaria di javaParser.

```java
bs.addStatement(parseStatement("RetryPolicy<Object> retryPolicy = RetryPolicy.builder()\n" +   You, Today • Unc
        "    .handle(Exception.class)\n" +
        "    .withBackoff(1, 30, ChronoUnit.SECONDS)\n" +
        "    .withMaxRetries(3)\n" +
        "    .onRetriesExceeded(e -> System.out.println(\"Failed to connect. Max retries exceeded.\"))\n" +
        "    .build();"));
alreadyVisited.put(bs.getRange().get().begin.line, bs.getRange().get().end.line);
```

Successivamente, per evitare di inserire due volte la stessa policy duplicata per lo stesso statement, viene memorizzata la posizione dello statement appena modificato.

AddRetryCode

Aggiunta la policy di Failsafe, è necessario modificare direttamente la linea della chiamata critica, essa rappresenta un possibile failure Point. Per rendere resiliente la chiamata sfruttiamo la policy sopra creata e la inseriamo all'interno di essa. Inoltre, la chiamata esistente verrà rimossa dall'albero della relativa CU.

```
if(isReturnExp) bs.addStatement(parseStatement("return Failsafe.with(retryPolicy).get(() ->"+localExpr+");"));
else
    bs.addStatement(parseStatement("Failsafe.with(retryPolicy).get(() ->"+localExpr+");"));
```

PrintAllStatements

Stampa tutti i blocchi presenti all'interno di tutti gli statements.