AMPLIACIÓN DE INGENIERÍA DEL SOFTWARE

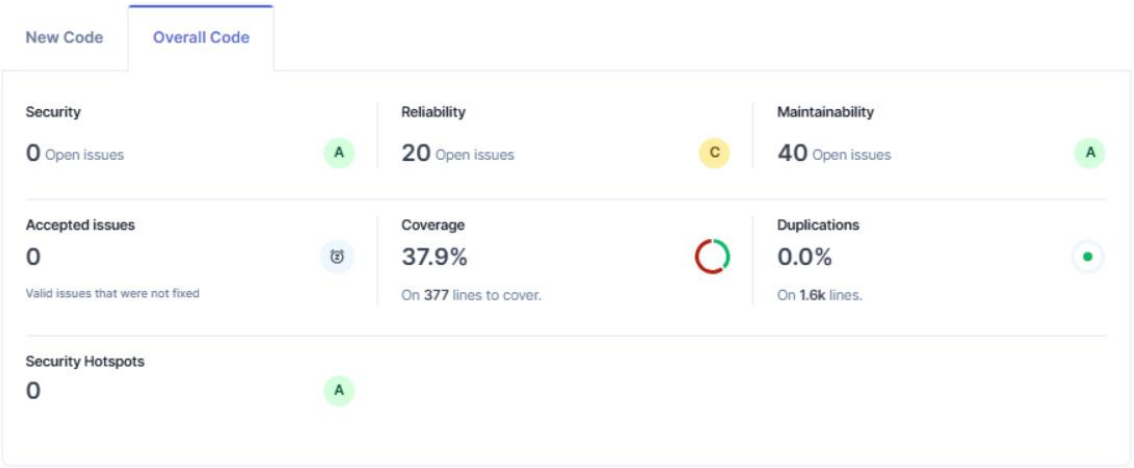
3º INGENIERÍA INFORMÁTICA

PRÁCTICA 2: PRUEBAS DE SISTEMA Y ANÁLISIS ESTÁTICO DE CÓDIGO

Por: Zaira Ruiz Fernández

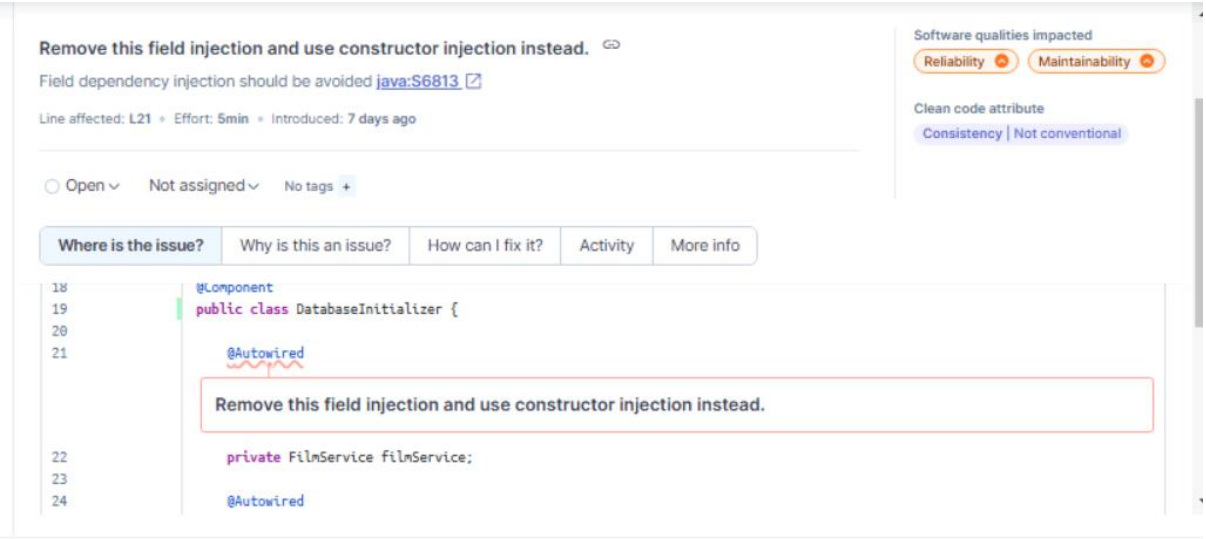
ÍNDICE

1.Overview inicial con las principales métricas.



2.Issues encontradas.

2.1.Inyección de dependencias por campo.



Este problema se debe a que, al inyectar dependencias directamente en un atributo, a la hora de programar algún test, te obliga a tener las dependencias implementadas, ya que no se puede hacer un mock de ellas. En cambio, si inyectas las dependencias mediante el constructor de la clase, se podría hacer un mock de ellas y pasarle estos mocks al constructor, permitiendo ahora sí tests solitarios.

Este problema se daba varias veces, pero, por claridad y legibilidad en el documento, no voy a poner todos los casos, sólo el primero. Sin embargo, las soluciones a continuación sí son las de todos los casos.

```java
private UserRepository userRepository;

@Autowired
public UserService(UserRepository userRepository){
    this.userRepository=userRepository;
}
```

```java
private FilmService filmService;

private ReviewService reviewService;

@Autowired
public FilmRestController(FilmService filmService, ReviewService reviewService) {
    this.filmService=filmService;
    this.reviewService=reviewService;
}
```

```java
private UserService userService;

private UserMapper userMapper;

@Autowired
public UserRestController(UserService userService, UserMapper userMapper){
    this.userService=userService;
    this.userMapper=userMapper;
}
```

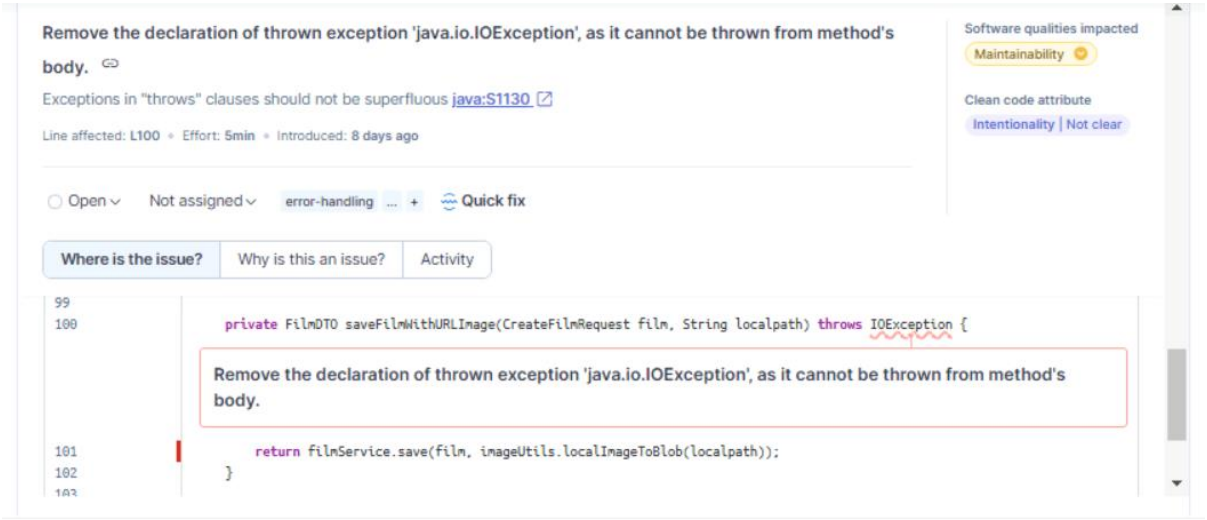
```java
private FilmService filmService;

private FavoriteFilmService favoriteFilmService;

private ReviewService reviewService;

@Autowired
public FilmWebController(FilmService filmService, FavoriteFilmService favoriteFilmService,
        ReviewService reviewService) {
    this.filmService = filmService;
    this.favoriteFilmService = favoriteFilmService;
    this.reviewService = reviewService;
}
```

```java
private UserComponent userComponent;

private UserService userService;

@Autowired
public UserWebController(UserComponent userComponent, UserService userService) {
    this.userComponent = userComponent;
    this.userService = userService;
}
```

```java
private UserComponent userComponent;

@Autowired
public UserModelAttributes(UserComponent userComponent){
    this.userComponent=userComponent;
}
```

```java
UserRepository userRepository;

@Autowired
public UserComponent(UserRepository userRepository){
    this.userRepository=userRepository;
}
```

2.2.Declaración de una excepción no generada nunca.



Dado que la excepción no se lanza nunca, se puede eliminar (hay otro caso más con este problema además de éste cuya solución se ve debajo).

```java
private FilmDTO saveFilmWithURLImage(CreateFilmRequest film, String localpath) {
    return filmService.save(film, imageUtils.localImageToBlob(localpath));
}
```

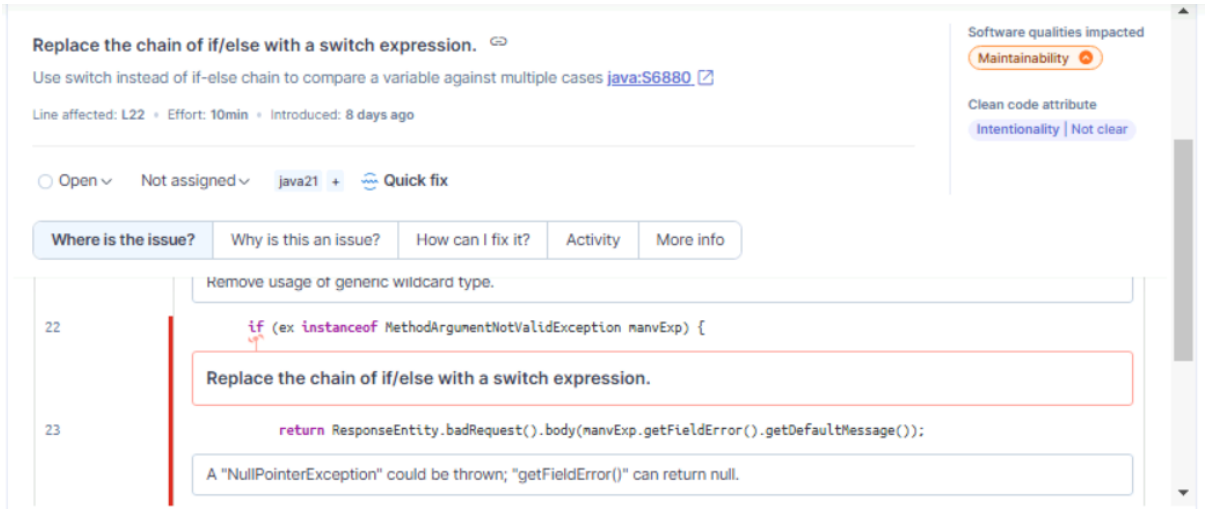
```java
@PostConstruct
public void init() {

    if (!isRunningTest()) {
        User michel = new User(name:"Michel", email:"michel.maes@urjc.es");
        User raul = new User(name:"Raúl", email:"raul@urjc.es");
```

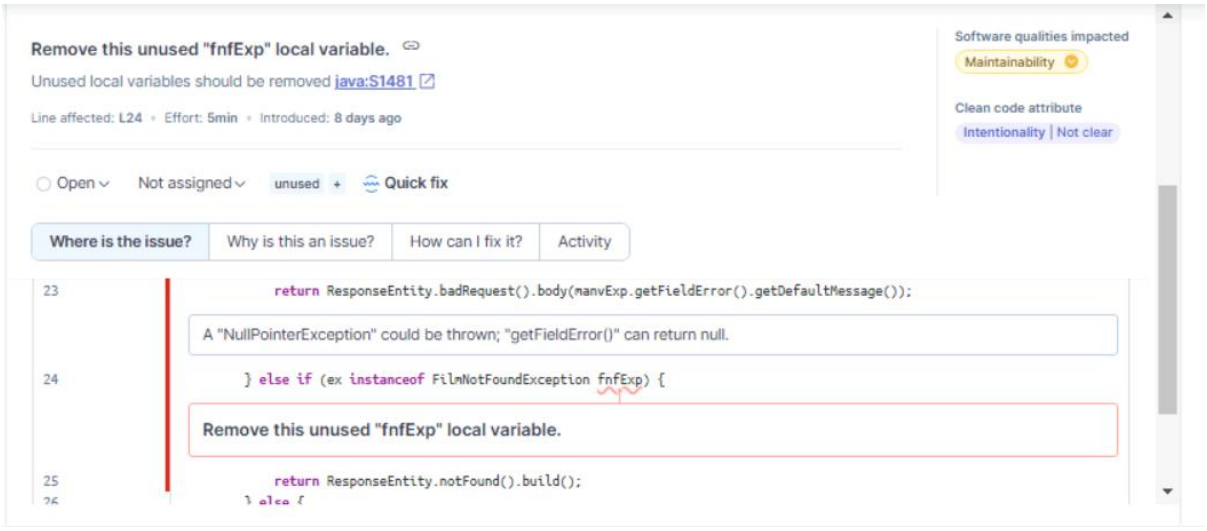
2.3.Uso de un tipo de retorno no especificado, ambiguo.

Al no especificar el tipo de retorno, el código es mucho menos claro y se vuelve mucho más difícil saber qué tipo de datos se espera exactamente como respuesta. En este caso se devuelve o un String o nada (en el segundo caso), así que se puede cambiar ? por String.

```
@ExceptionHandler({ FilmNotFoundException.class, IllegalArgumentException.class, BindException.class })
public ResponseEntity<String> handleException(Exception ex) {
    if (ex instanceof MethodArgumentNotValidException manvExp) {
        return ResponseEntity.badRequest().body(manvExp.getFieldError().getDefaultMessage());
    } else if (ex instanceof FilmNotFoundException fnfExp) {
        return ResponseEntity.notFound().build();
    } else {
        return ResponseEntity.badRequest().body(ex.getMessage());
    }
}
```

2.4.Cadena de if/else y variable no utilizada.



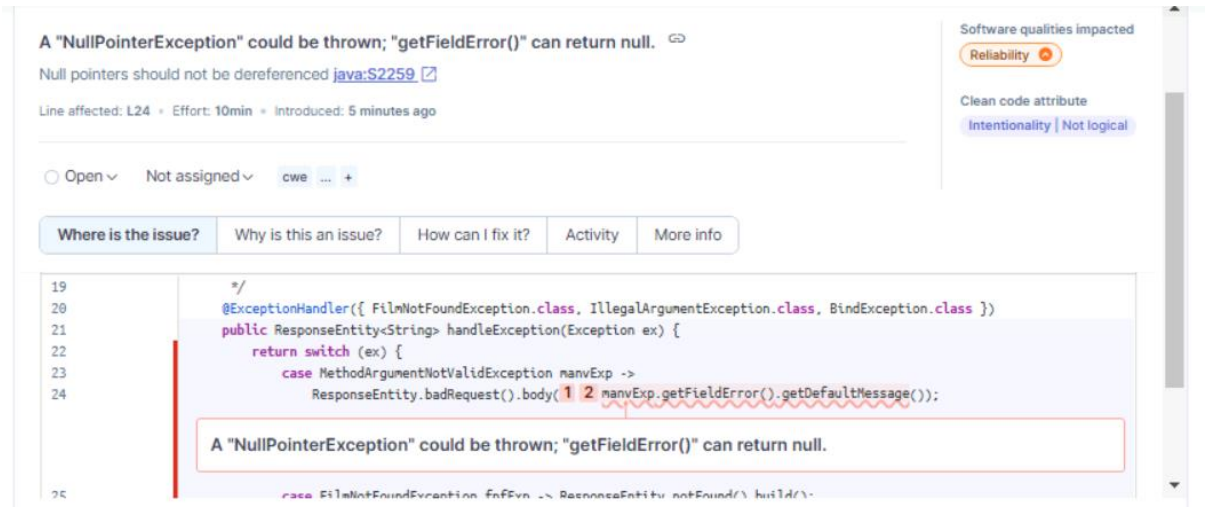


Hay una cadena larga de if/else, por lo que la claridad del código sería mayor al sustituirlo por una expresión switch (que va a permitir que el contenido de dichos if/else se visualice mejor).

```
@ExceptionHandler({ FilmNotFoundException.class, IllegalArgumentException.class, BindException.class })
public ResponseEntity<String> handleException(Exception ex) {
    return switch (ex) {
        case MethodArgumentNotValidException manvExp ->
            ResponseEntity.badRequest().body(manvExp.getFieldError().getDefaultMessage());
        case FilmNotFoundException fnfExp -> ResponseEntity.notFound().build();
        default -> ResponseEntity.badRequest().body(ex.getMessage());
    };
}
```

Al utilizar el switch, la variable pasa a ser necesaria aunque no se utilice, así que ya no se puede quitar.

2.5.getFieldError() puede devolver null.

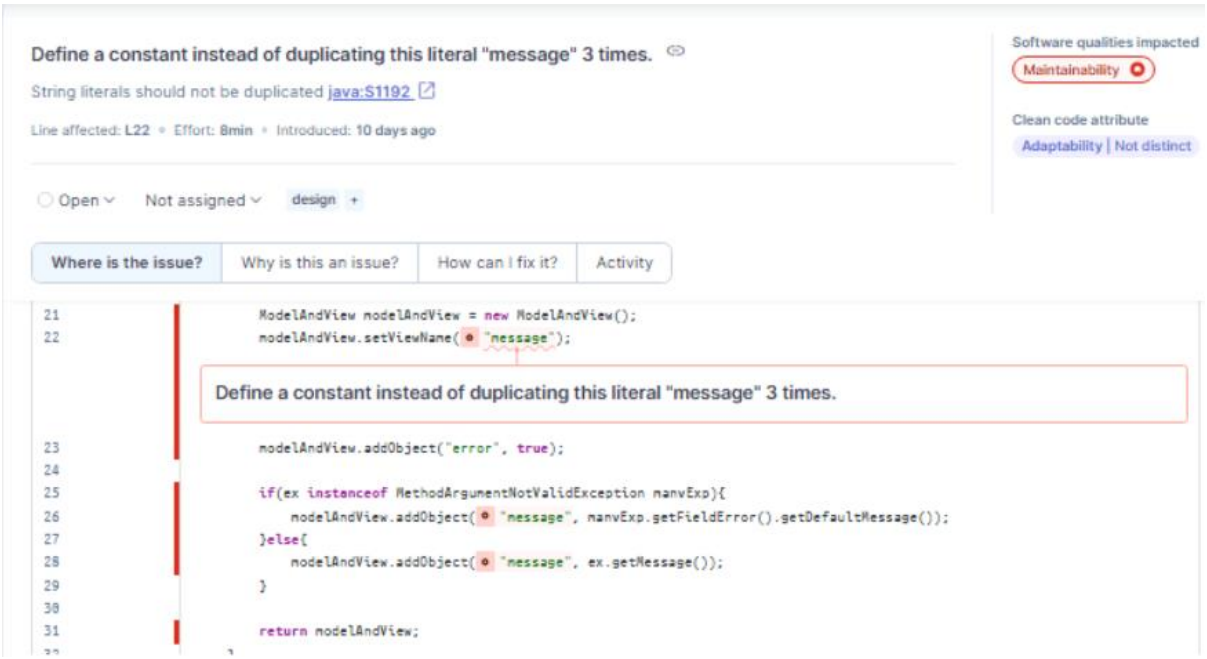


Como el método getFieldError() puede devolver null, intentar llamar al siguiente método (getDefaultMessage) podría provocar una excepción (ya que no se puede

llamar al método desde null). Para corregirlo, primero se hace una comprobación de si el objeto devuelto por getFieldError() es nulo o no y luego, en caso de no serlo, se llama al otro método. El aviso en sonar sigue saltando, pero como ahora es imposible que se lance la excepción, se puede clasificar el aviso como un falso positivo.

```
case MethodArgumentNotValidException manvExp ->
    ResponseEntity.badRequest()
            .body(manvExp.getFieldError() != null ? manvExp.getFieldError().getDefaultMessage() : "");
```

2.6.Literal repetido tres veces.



Tener ”message” tres veces va a aumentar el número de sitios en los que hay que modificar la cadena si se quiere cambiar por otra, lo que hace el código más largo y difícil de mantener (además es más fácil cometer un error al hacerlo).

Se soluciona añadiendo una variable que contenga “message” y que sea la que se use en lugar de la cadena directamente. Este problema también se daba en varias partes:

```java
@ExceptionHandler({FilmNotFoundException.class, IllegalArgumentException.class, BindException.class})
public ModelAndView handleException(Exception ex){
    String message="message";
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.setViewName(message);
    modelAndView.addObject("error", true);

    if(ex instanceof MethodArgumentNotValidException manvExp){
        modelAndView.addObject(message, manvExp.getFieldError().getDefaultMessage());
    }else{
        modelAndView.addObject(message, ex.getMessage());
    }
```

```java
private String filmNotFound="Film not found";
private String error="error";
private String action="action";
private String filmForm="filmForm";
private String redirectFilms="redirect:/films/";
```

```java
    return "film";
} else {
    throw new ResponseStatusException(HttpStatus.NOT_FOUND, filmNotFound);
}
```

```java
    FilmDTO removedFilm = op.get();
    model.addAttribute(error, false);
    model.addAttribute("message", "Film '" + removedFilm.title() + "' deleted");
    return "message";
} else {
    throw new ResponseStatusException(HttpStatus.NOT_FOUND, filmNotFound);
}
```

```java
@GetMapping("/films/new")
public String newFilm(Model model) {
    model.addAttribute(action, "/films/new");
    model.addAttribute("film", new Film());
    model.addAttribute("ageRatings", AgeRating.values());
    return filmForm;
}
```

```java
public String newFilmProcess(CreateFilmRequest film, MultipartFi

    FilmDTO newFilm = null;

    try {
        newFilm = filmService.save(film, imageField);
    } catch (IllegalArgumentException e) {
        model.addAttribute(error, true);
        model.addAttribute("errors", List.of(e.getMessage()));
        model.addAttribute(action, "/films/new");
        model.addAttribute("film", film);
        return filmForm;
    }

    return redirectFilms + newFilm.id();
}
```

```java
@GetMapping("/films/{id}/edit")
public String editFilm(Model model, @PathVariable long id) {

    Optional<FilmDTO> op = filmService.findOne(id);
    if (op.isPresent()) {
        FilmDTO film = op.get();
        model.addAttribute(action, "/films/" + id + "/edit");
        model.addAttribute("film", film);
        model.addAttribute("ageRatings", AgeRatingOptionsUtils.getAgeRatingOpti
        return filmForm;
    } else {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, filmNotFound);
    }
}
```

```java
public class FilmWebController {
    public String editFilmProcess(Model model, @PathVariable long i
        FilmDTO updatedFilm = null;

        try {
            updatedFilm = filmService.update(id, film, imageField);
        } catch (ResponseStatusException e) {
            model.addAttribute(error, true);
            model.addAttribute("errors", List.of(e.getReason()));
            model.addAttribute(action, "/films/" + id + "/edit");
            model.addAttribute("film", film);
            return filmForm;
        }

        model.addAttribute("film", updatedFilm);

        return redirectFilms + film.id();
    }
```

```java
public String addReview(@PathVariable
    FilmDTO film = reviewService.addRe
    return redirectFilms + film.id();
}

@PostMapping("/films/{filmId}/reviews/
public String removeReview(@PathVariab
    FilmDTO film = reviewService.delet
    return redirectFilms + film.id();
}

@PostMapping("/films/{filmId}/addFavor
public String addFavorite(@PathVariabl
    favoriteFilmService.addToFavorites
    return redirectFilms + filmId;
}
```

```java
@PostMapping("/films/{filmId}/removeF
public String removeFavorite(@PathVar
    favoriteFilmService.removeFromFav
    return redirectFilms + filmId;
}
```

```java
private String errorProcessingImage="Error at processing the image";
```

```
} catch (IOException e) {
    throw new ResponseStatusException(HttpStatus.BAD_REQUEST, errorProcessingImage);
}
```

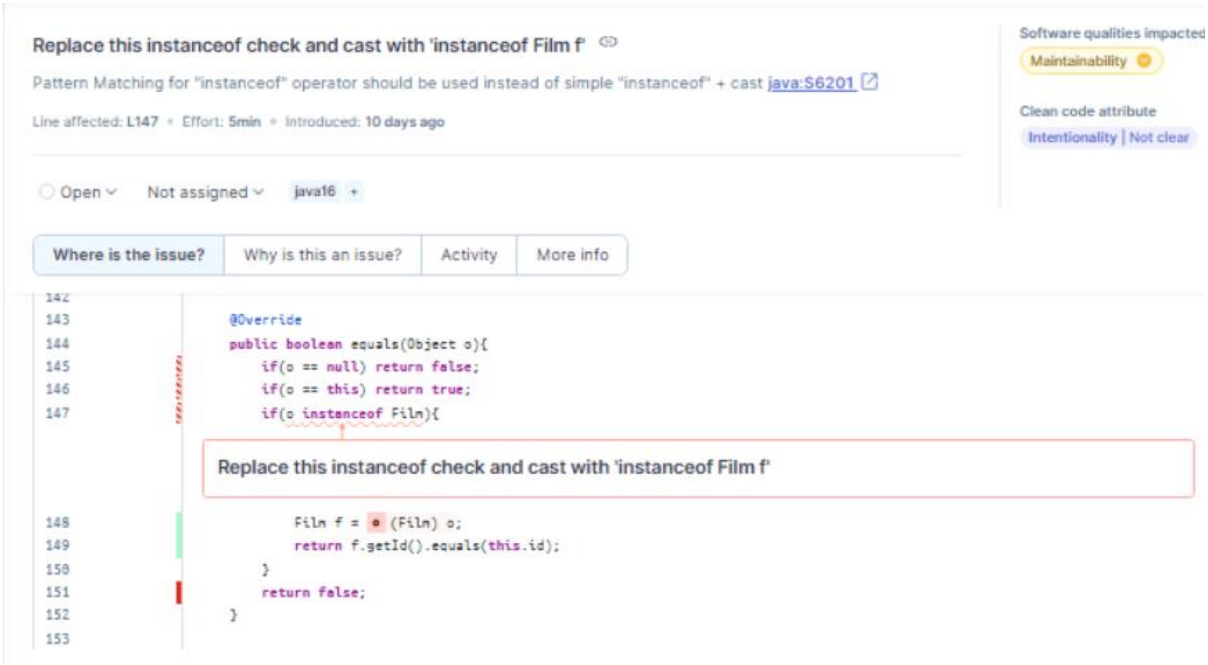
```
} catch (IOException e) {
    throw new ResponseStatusException(HttpStatus.BAD_REQUEST, errorProcessingImage);
}
```

```
} catch (IOException e) {
    throw new ResponseStatusException(HttpStatus.BAD_REQUEST, errorProcessingImage);
}
```

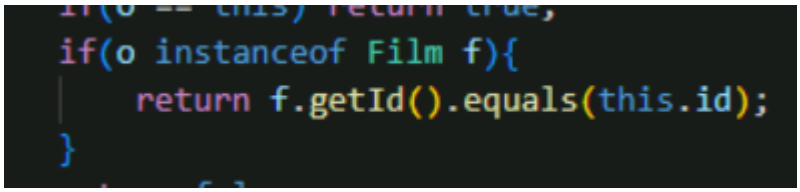
2.7.El código de estado devuelto no refleja el estado de la operación.

Aquí se devuelve siempre el código 200 OK, pero esto puede no ser correcto, porque la operación podría fallar al no encontrar lo que se busca. Sin embargo, aquí lo que se busca es una imagen que está cargada en una carpeta del proyecto, por lo que siempre la encontrará, devolviendo un código 200 OK. Debido a esto, lo podemos considerar un falso positivo.

2.8.Cast innecesario.

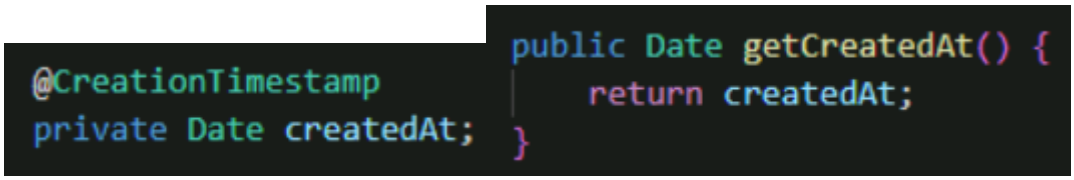


En este código el cast es innecesario y redundante porque ya se está haciendo en la comprobación instanceof, por lo que se puede declarar la variable Film f directamente ahí y usarla después.

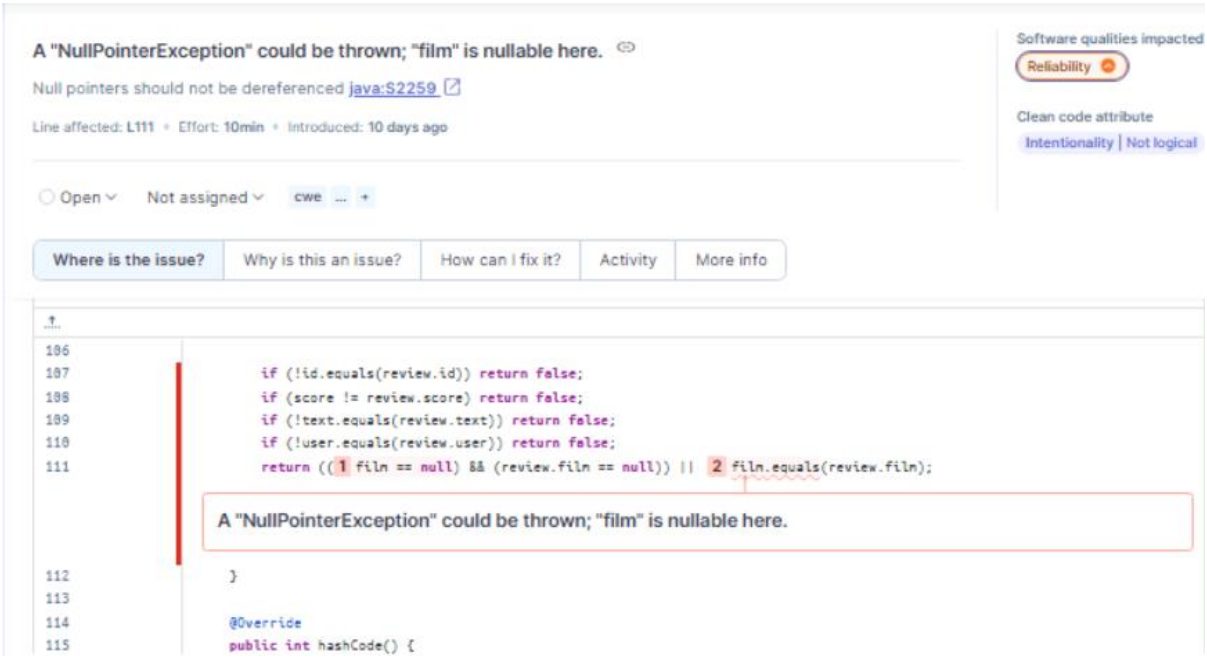


2.9.Nombres de campos y métodos que no siguen el formato adecuado (camelCase).

Para solucionarlo, simplemente se eliminan las barras bajas y se pone la primera letra de la segunda palabra en mayúscula, pasando el nombre a camelCase. Este problema también se da en el nombre de un método de la misma clase.



2.10.film puede ser null.



Como la variable film puede ser null, intentar llamar al método equals desde ella (o desde review.film, que también podría ser null), podría provocar una excepción (ya que no se puede llamar al método desde null). Para corregirlo, primero se hace una comprobación de si la variable film es nula o no y luego, en caso de no serlo, se llama al método equals (aquí se puede hacer mediante &&, ya que no pasa a la segunda condición si la primera no se cumple).

```
return ((film == null) && (review.film == null)) || ((film != null) && film.equals(review.film));
```

2.11.Excepción poco clara.

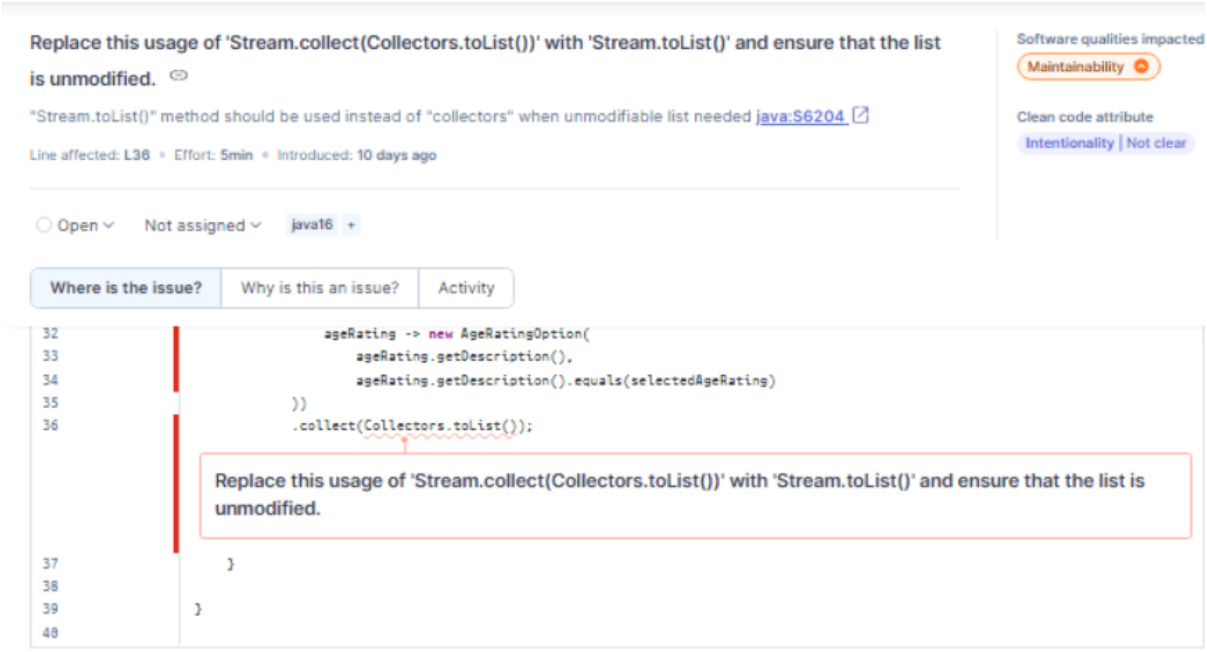
La excepción RuntimeException no es muy específica, lo que podría dificultar la

comprensión del código y de los problemas que surjan. Para solucionarlo, se cambia por una excepción más específica.

```
} catch (SQLException e) {
    throw new ImageFromDatabaseException(message:"Error getting image from database", e);
}
```

```
package es.codeurjc.web.nitflex.service.exceptions;

public class ImageFromDatabaseException extends RuntimeException {
    public ImageFromDatabaseException(String message, Throwable cause) {
        super(message, cause);
    }

}
```

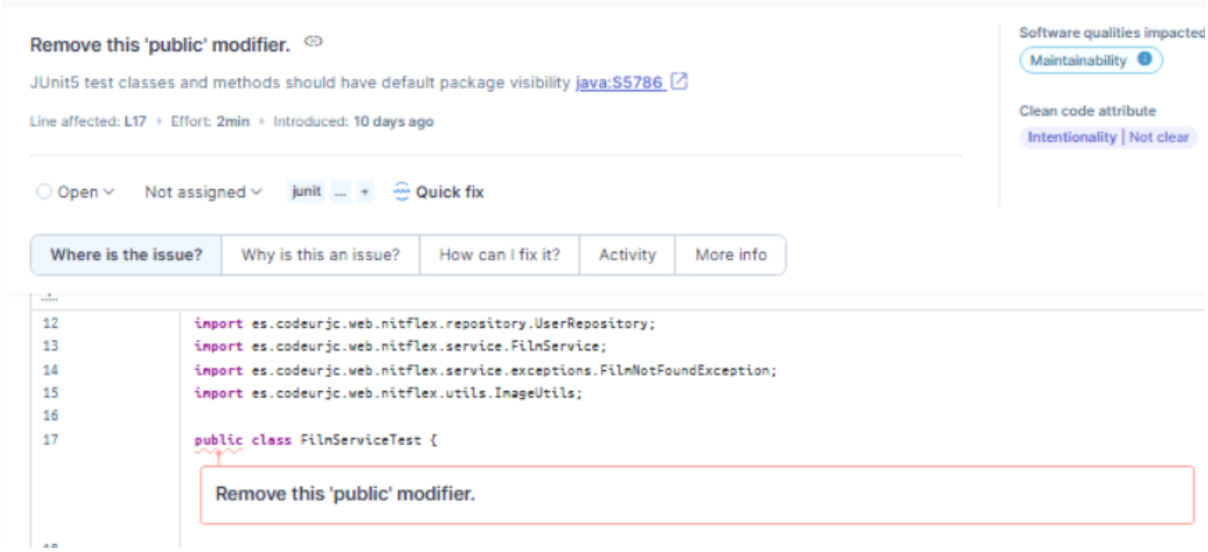
2.12.Lista mutable.



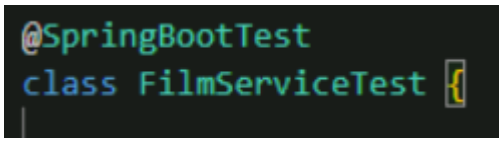
.collect(Collectors.toList()) devuelve una lista que se puede modificar, lo que es poco seguro y puede dar lugar a errores. Para hacerla inmutable, basta con utilizar .toList().

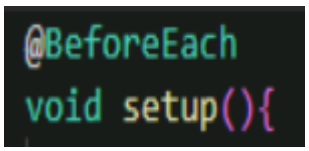
```
return Stream.of(AgeRating.values())
    .map(
        ageRating -> new AgeRatingOption(
            ageRating.getDescription(),
            ageRating.getDescription().equals(selectedAgeRating)
    ))
    .toList();
```

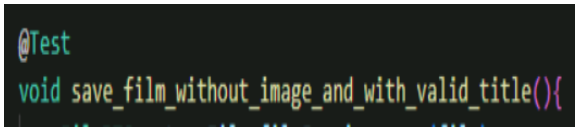
2.13.Visibilidad pública innecesaria en clases de test y sus métodos.

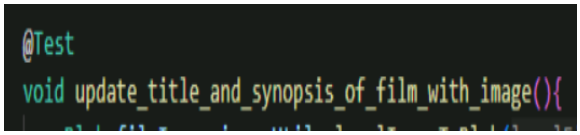


Al ser un test, se asume directamente la visibilidad pública, por lo que especificarlo no es necesario. Esto se da en las clases de test de integración y unitarios y no es realmente un problema, pero basta con quitar la visibilidad de las clases y métodos para solucionarlo.









Esto es sólo la solución en una de las clases, pero con la otra es exactamente igual (no está puesta por claridad de la memoria).

3.Overview final con las principales métricas y la evaluación del nuevo código.