

SUPSI

2D image editor

TABLE OF CONTENTS

- 01 CONTEXT AND MOTIVATION
- 02 PROBLEM
- 03 STATE OF THE ART
- 04 APPROACH
- 05 DEMO
- 06 CONCLUSIONS

CONTEXT AND MOTIVATION



ACADEMIC PROJECT

Academic software engineering project focusing on implementing software design patterns and testing best practices



ANALYZE REQUIREMENTS

Creation of a 2D image editor with support for applying multiple filters in a pipeline architecture



CREATE A 2D EDITOR

Development of an extensible application enabling filter chaining, comprehensive testing coverage, and appropriate design pattern usage

PROBLEM

Create a modular 2D image editor capable of loading different image formats (PBM, PPM, PGM) and applying a customizable pipeline of filters. The application must support:

- Loading and saving images in PNM formats
- Dynamic filter pipeline management
- Cross-format image conversion
- Preferences management
- Localization in multiple languages

STATE OF THE ART



ADOBE PHOTOSHOP



LUMINAR NEO

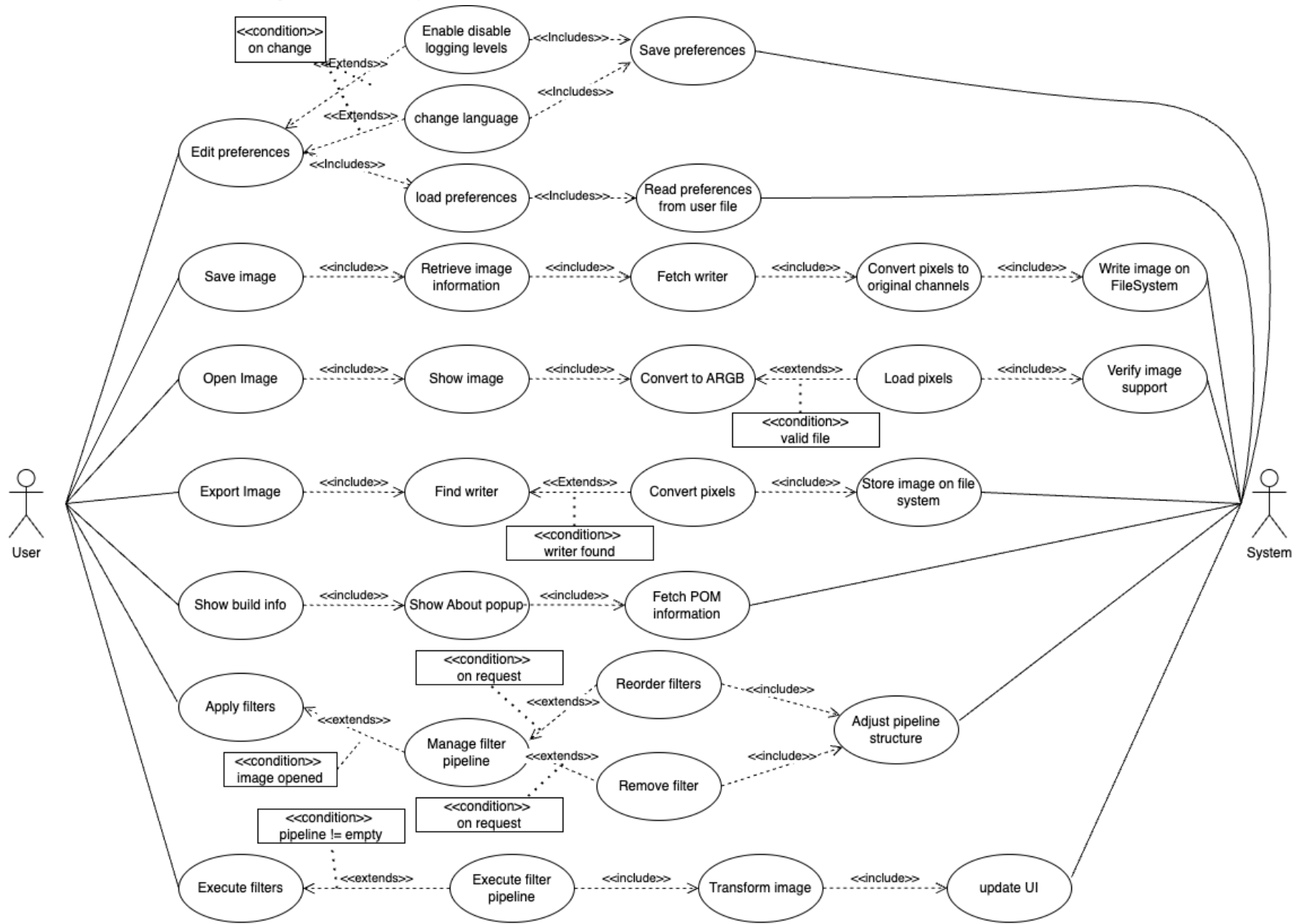


PHOTO DIRECTOR



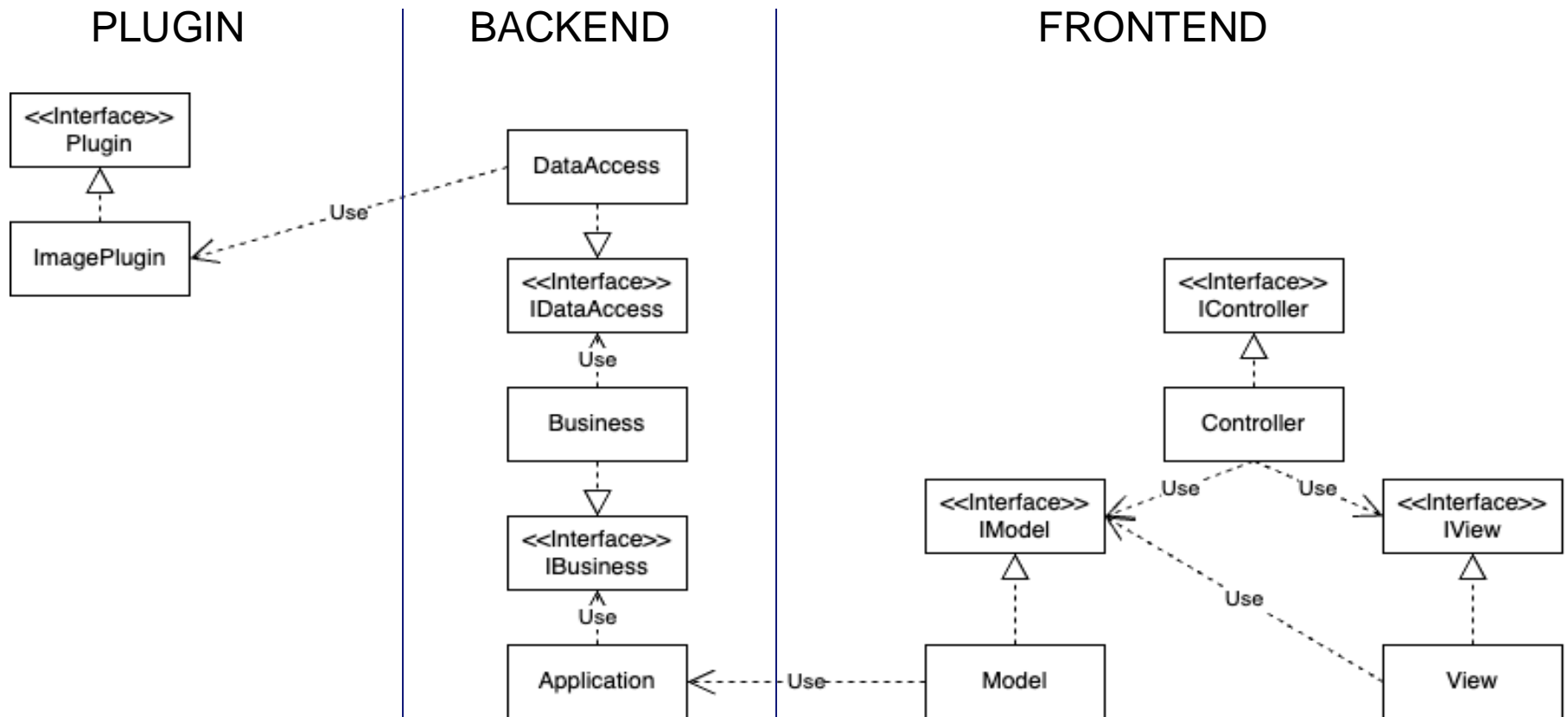
CAPTURE ONE PRO

USE CASE (1 of 1)



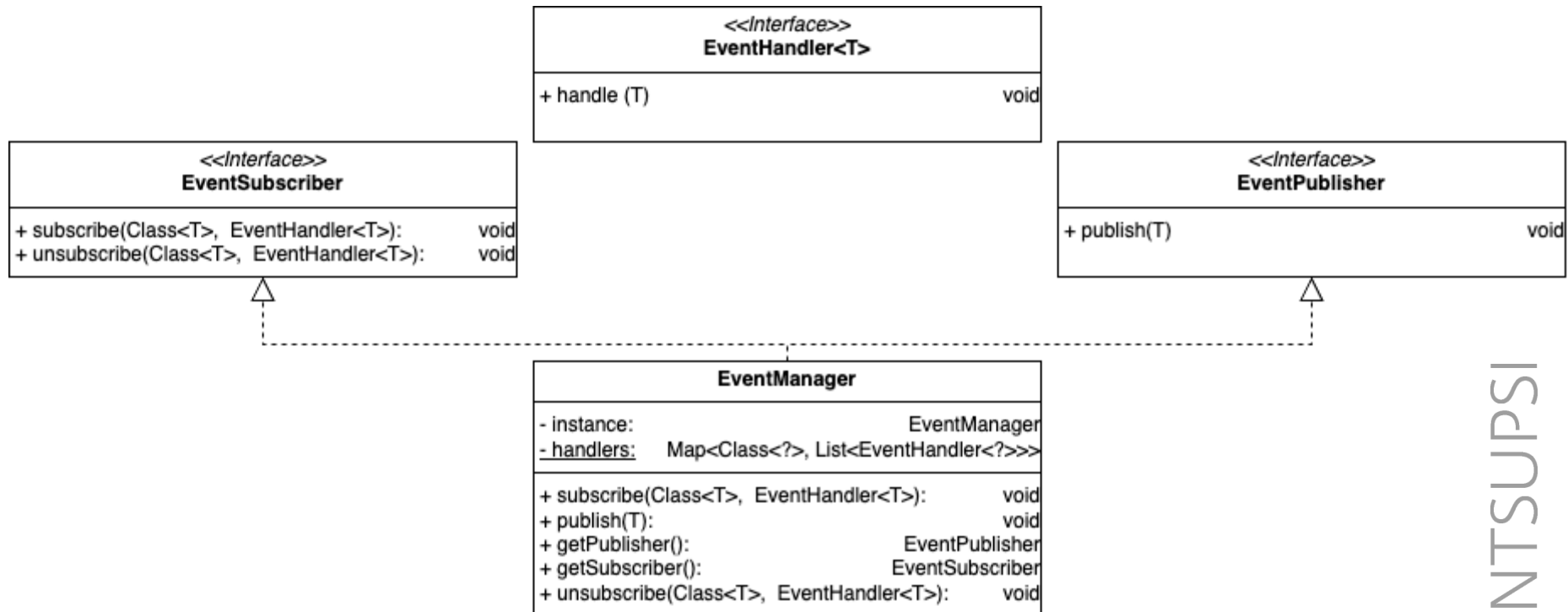
APPROACH (1 of 15)

Architecture



APPROACH (2 of 15)

Patterns - Observer



APPROACH (3 of 15)

Patterns - Observer

```
public sealed interface FilterEvent {  
    record FilterAddRequested(String filterName) implements FilterEvent {}  
    record FilterRemoveRequested(int index) implements FilterEvent {}  
    record FilterMoveRequested(int fromIndex, int toIndex) implements FilterEvent {}  
    record FilterExecutionRequested() implements FilterEvent {}  
}
```

Controller:

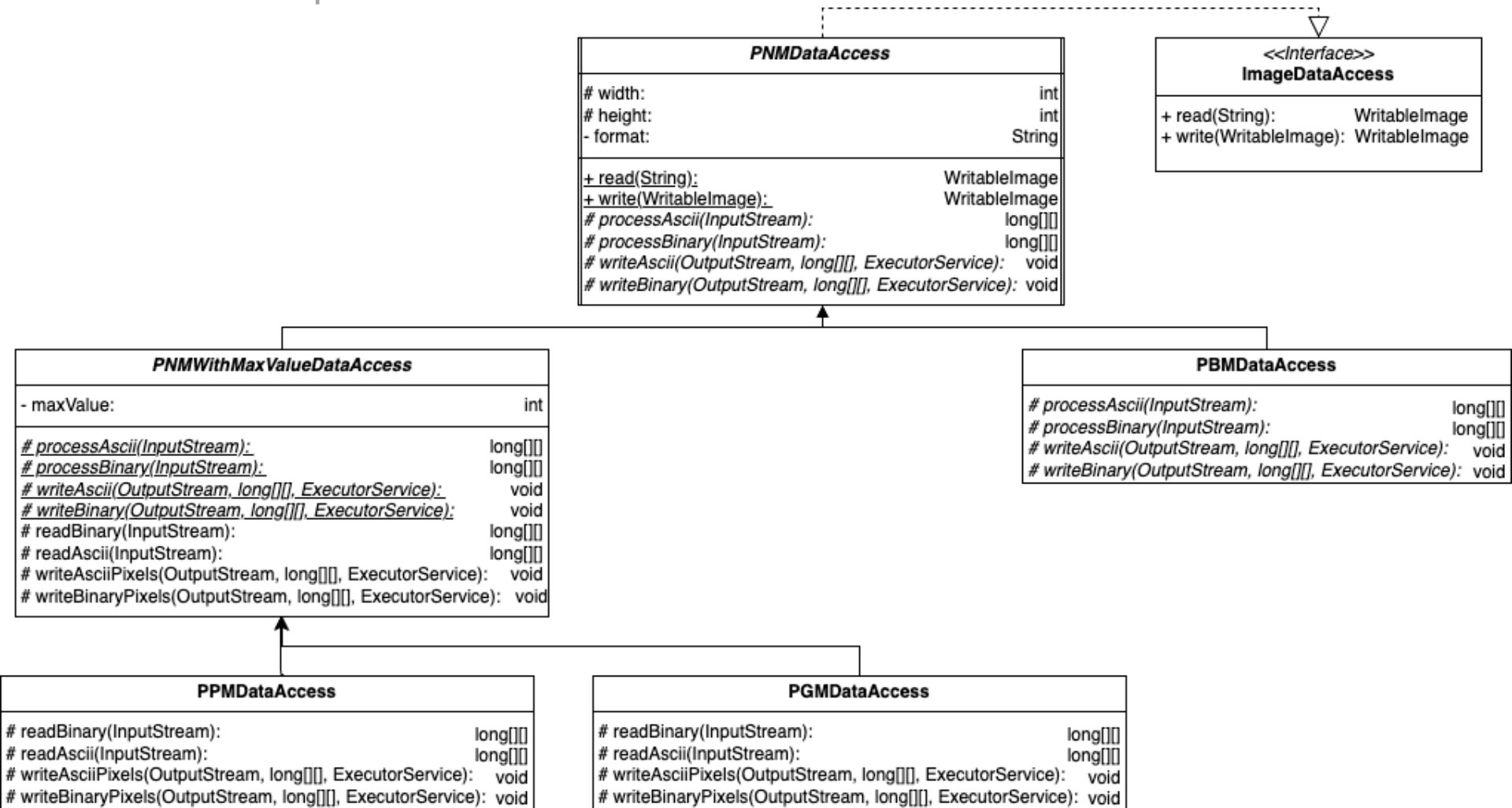
```
EventSubscriber subscriber = EventManager.getSubscriber();  
subscriber.subscribe(FilterEvent.FilterAddRequested.class, this::onFilterAdded);
```

View:

```
EventPublisher publisher = EventManager.getPublisher();  
publisher.publish(new FilterEvent.FilterAddRequested(filterKey))
```

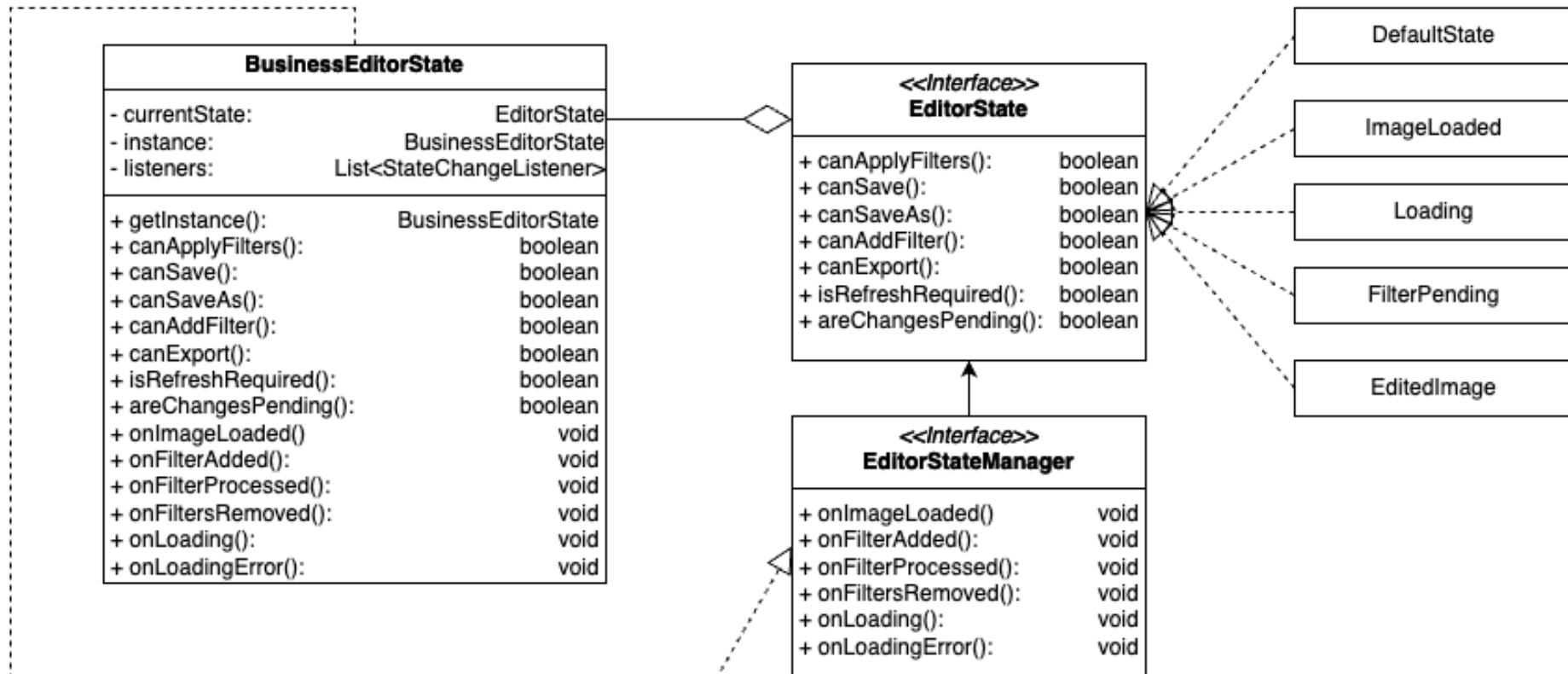
APPROACH (4 of 15)

Patterns - Template



APPROACH (5 of 15)

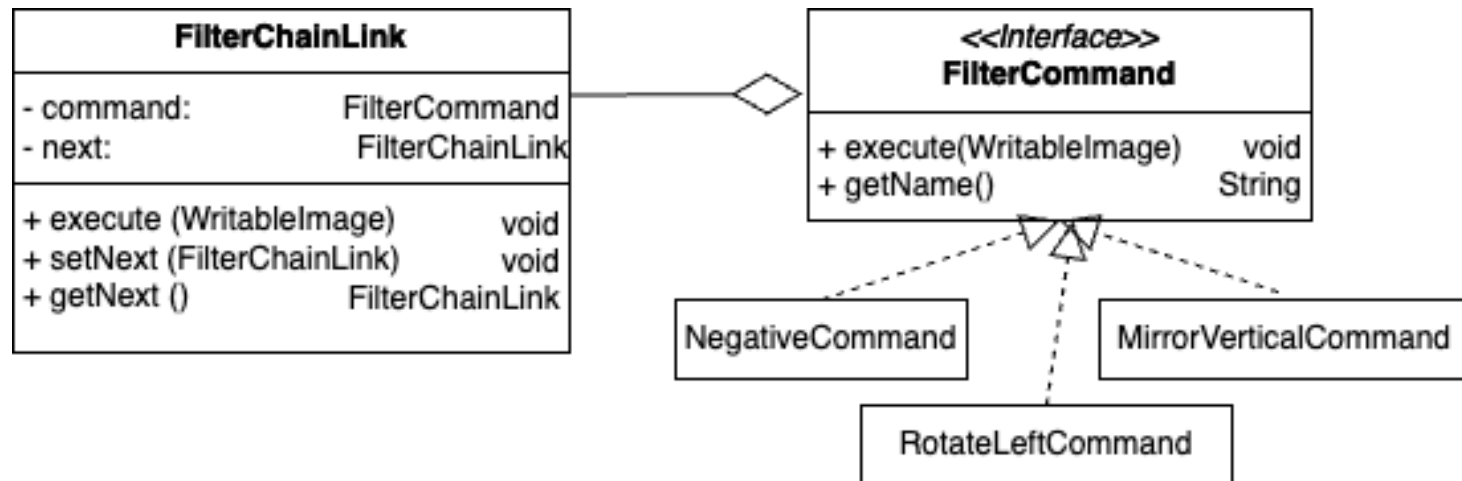
Patterns - State



APPROACH (6 of 15)

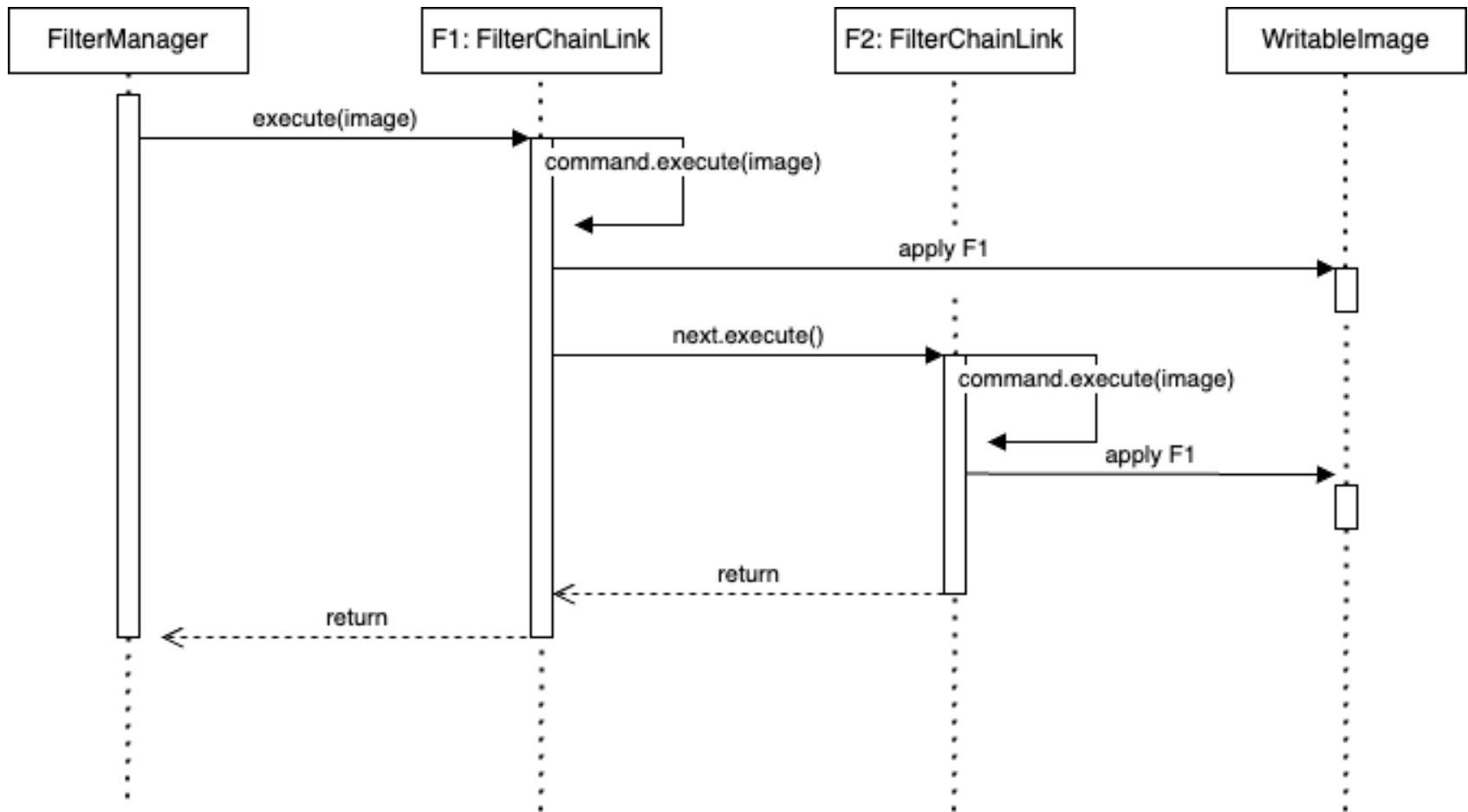
Patterns – Chain of responsibility

- Order matters
- If a command fails, others should not be executed
- Every component processes the image.
- It provides support for single filter execution
- Commands are wrapped, in this way no component 'knows' other components on the same layer
- Commands are stateless, so they can be cached and used multiple times, even in the same pipeline, without side effects or circular dependencies.



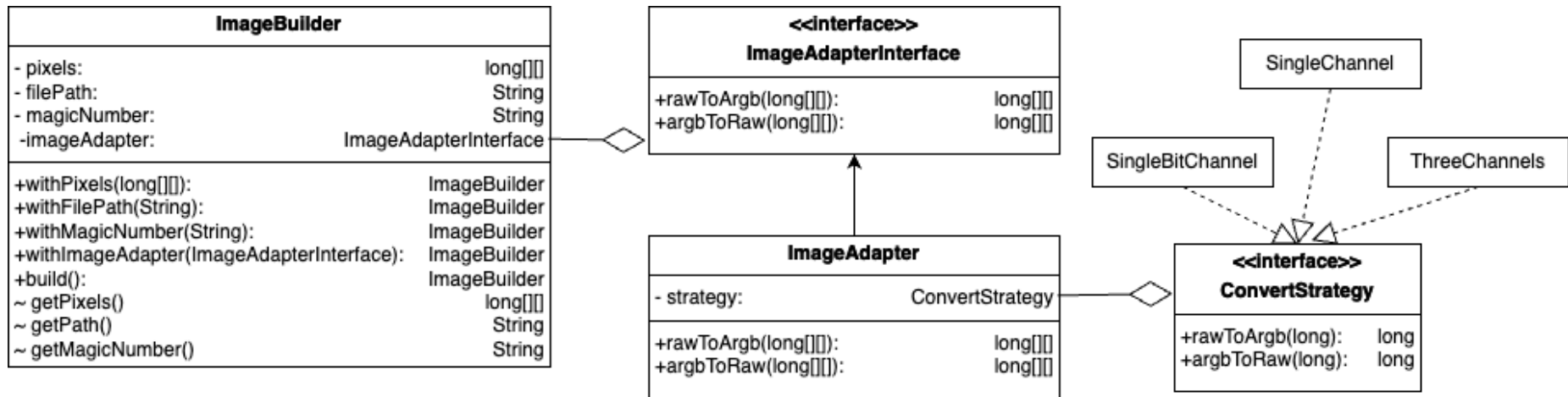
APPROACH (7 of 15)

Patterns – Chain of responsibility



APPROACH (8 of 15)

Patterns - Strategy



APPROACH (9 of 15)

Patterns - Strategy

```
@Override
public long ArgbToOriginal(long pixel) {
    int r = (int) ((pixel >> 16) & 0xFF);
    int g = (int) ((pixel >> 8) & 0xFF);
    int b = (int) (pixel & 0xFF);

    if (r != g || g != b) {
        return (long) (((0.299 * r + 0.587 * g + 0.114 * b) / 255.0) * maxValue);
    } else {
        return (long) ((b / 255.0) * maxValue);
    }
}
```

APPROACH (10 of 15)

Patterns – Dependency Injection



Chain of Responsibility

- Performance overhead due to runtime chain traversal
- High coupling: components require knowledge of other components at same level
- Poor extensibility: adding new components requires modifying existing code
- Error propagation: failures in one component can cascade through the chain
- Complex debugging of chain execution flow

Chain of Responsibility with Configuration

- Violates Java's 'convention over configuration' principle
- Runtime overhead for configuration parsing
- Additional complexity from reflection-based validation
- Same coupling and separation of concerns issues as basic chain
- Configuration maintenance becomes an extra burden

APPROACH (11 of 15)

Patterns – Dependency Injection

Pure Reflection Approach

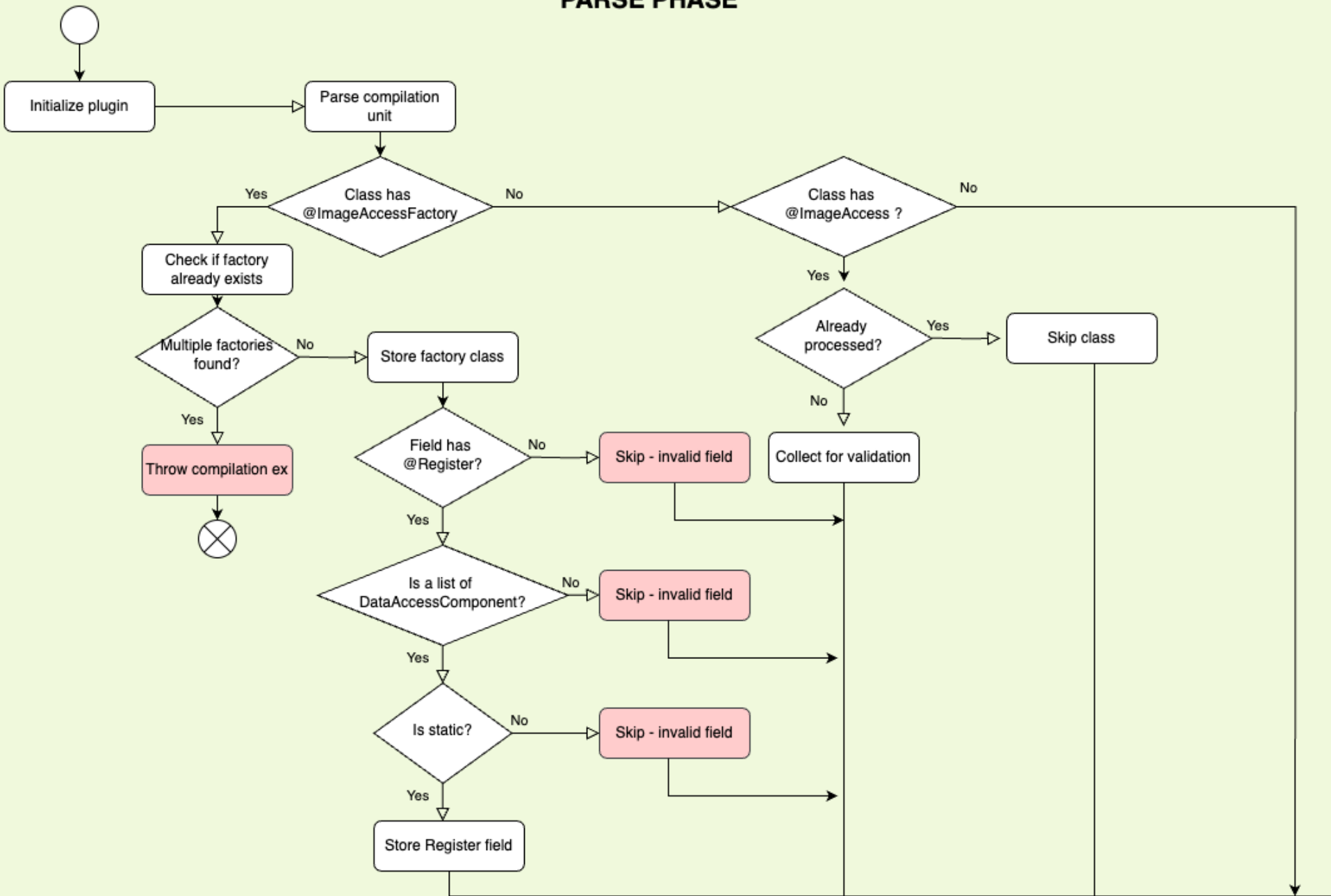
- More flexible and decoupled design
- Significant runtime performance overhead
- Defers critical validation to runtime
- Potential runtime failures from incorrect implementations (e.g., malformed singletons)
- Complex error handling and recovery

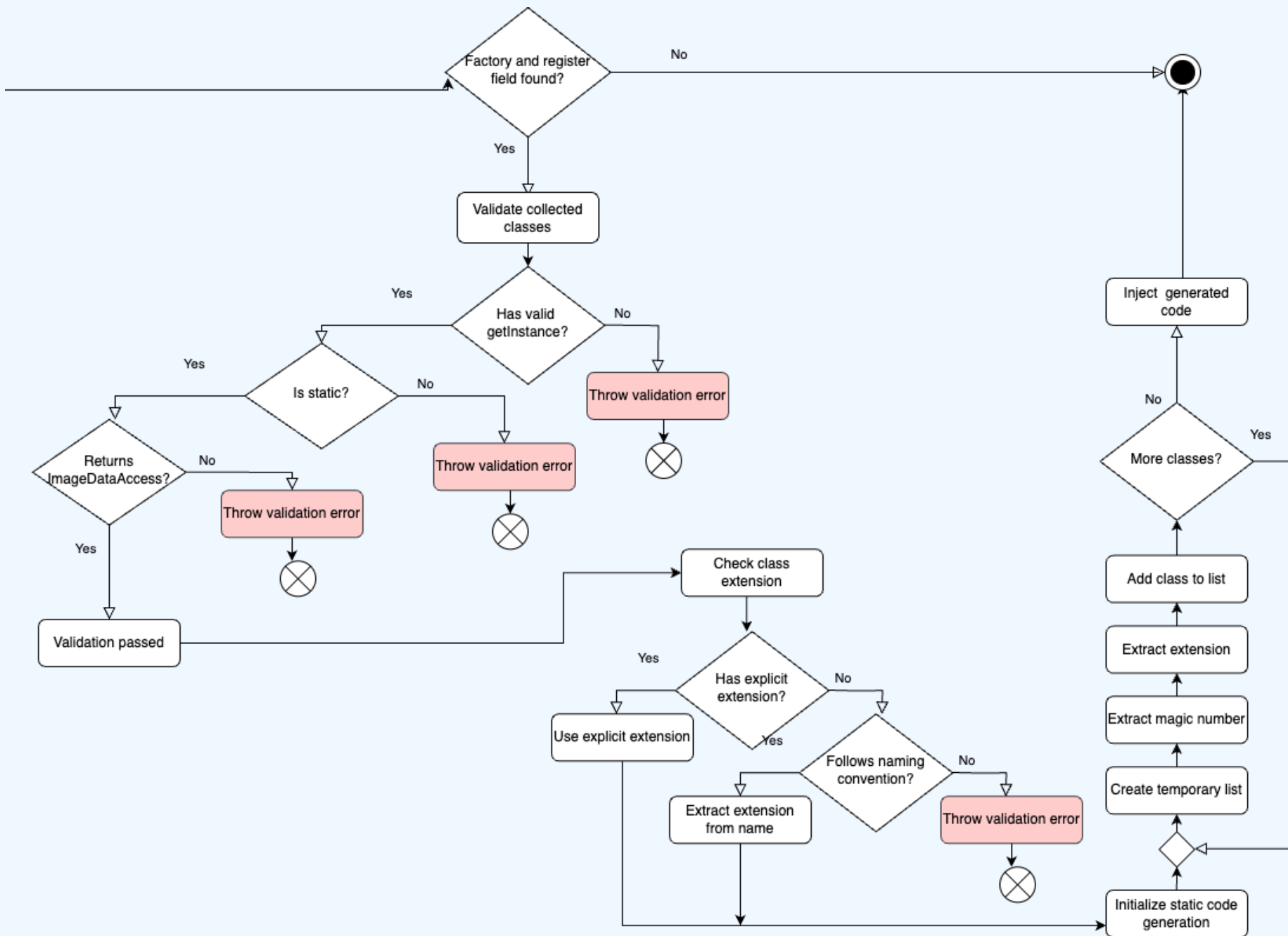
Compiler plugin (Selected Solution)

- Compile-time validation and code generation
- No runtime overhead
- Early detection of implementation errors
- Better separation of concerns
- Follows Java conventions for extensibility
- Simplified maintenance and debugging
- Flawless extensibility

—————→ Native javac plugin

PARSE PHASE





APPROACH (14 of 15)

Patterns – Dependency Injection

```
@ImageAccessFactory
public class DataAccessFactory {

    /** {@link List} of registered data access components. */
    @Register 4 usages
    private static List<DataAccessComponent> dataAccessComponents;
```

Decompiled .class file, bytecode version: 61.0 (Java 17)

```
22      public class DataAccessFactory { no usages
146          static {
147              List<DataAccessComponent> templist = new ArrayList();
148              DataAccessComponent component0 = new DataAccessComponent();
149              component0.magicNumber = new String[]{"P3", "P6"};
150              component0.extension = "ppm";
151              component0.clazz = PPMDDataAccess.class;
152              templist.add(component0);
153              DataAccessComponent component1 = new DataAccessComponent();
154              component1.magicNumber = new String[]{"P1", "P4"};
155              component1.extension = "pbm";
156              component1.clazz = PBMDDataAccess.class;
157              templist.add(component1);
158              DataAccessComponent component2 = new DataAccessComponent();
159              component2.magicNumber = new String[]{"P2", "P5"};
160              component2.extension = "pgm";
161              component2.clazz = PGMDDataAccess.class;
162              templist.add(component2);
163              dataAccessComponents = templist;
164          }
165      }
166  }
```

APPROACH (15 of 15)

Optional – Exceptions – Modules

Optional vs Null:

- Using Optional makes it explicit that a value might not be present
- Forces conscious handling of null cases through methods like `orElse/orElseThrow`
- Improves code readability and maintainability by reducing `NullPointerException`s

Exception Handling:

- Exceptions propagated from the backend are caught in the frontend
- Frontend translates them into user-friendly messages through `ErrorController`

Java Modules:

- Clear separation between backend and frontend through `module-info.java`
- Translation bundles are encapsulated in their respective modules (better organization)
- Precise control over dependencies and package access
- Strong encapsulation: only intentionally exposed APIs are accessible
- Better architectural design

TESTS (1 of 2)

Coverage

	CLASSES	COV	LINES	COV	METHODS	COV	BRANCHES	COV
BACKEND	37	100%	4.425	100%	277	100%	426	100%
FRONTEND	45	100%	3.626	100%	266	100%	168	100%

	TESTS
BACKEND	326
FRONTEND	166
PLUGIN	20

512 tests

TESTS (1 of 2)

Strategies

JUNIT 5

- Test framework

TESTFX

- End-to-end tests

MOCKITO

- Mock
- Spy
- Static mock
- Mocked construction

JAVAASSIST

- Runtime dynamic bytecode generation

JAVA TOOLS

- Runtime code compilation

JUNIT Standalone

- Plugin testing

JACOCO

- Test coverage

ADDITIONAL DOCUMENTATION (1 of 1)

TEST REPORTS

CODE DOCUMENTATION

MAINTENANCE GUIDE

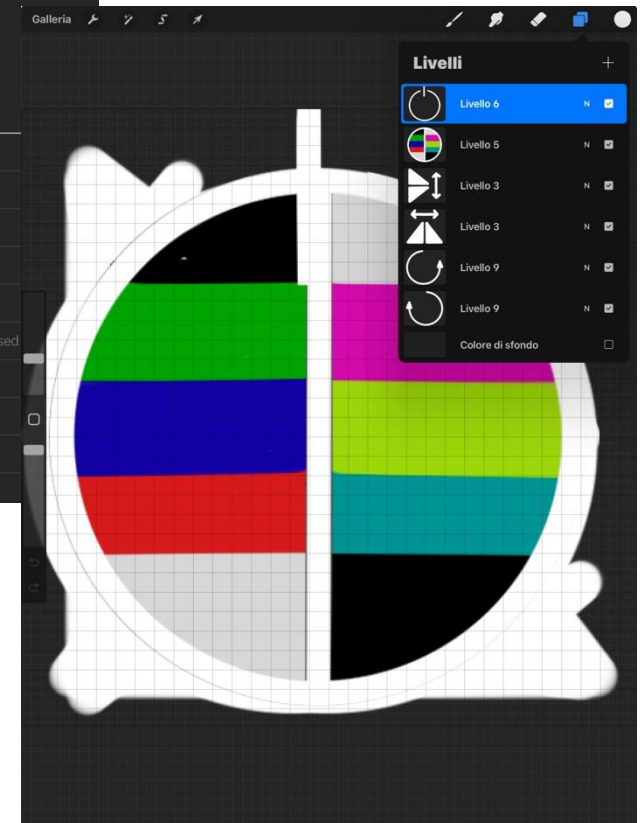
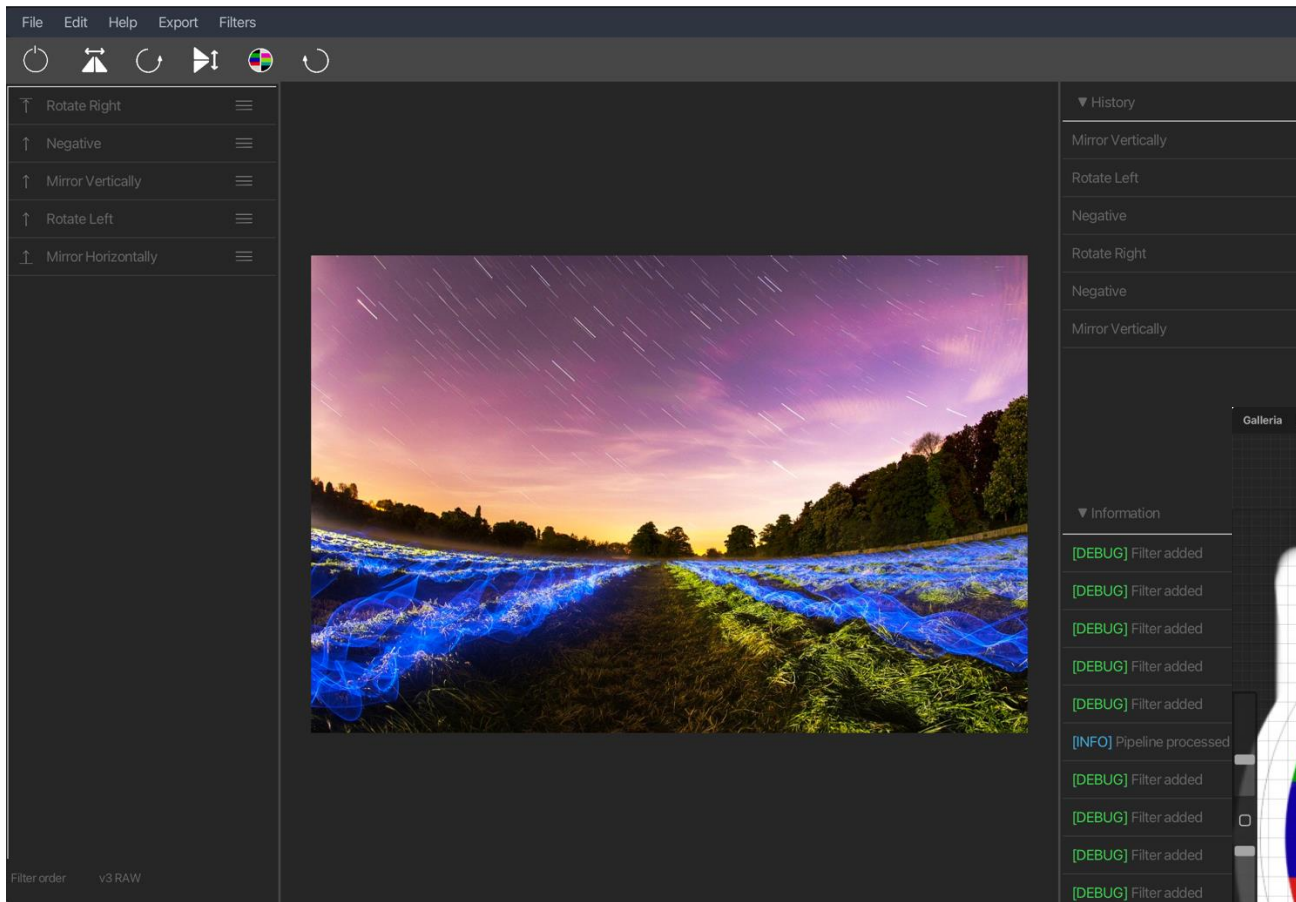
BUILD SCRIPTS

DIAGRAMS

README

```
marti@macbook-pro-di-martina ~/Documents/a/Ingegneria_software_2/progetto_os/os (main) % ./build.sh
```

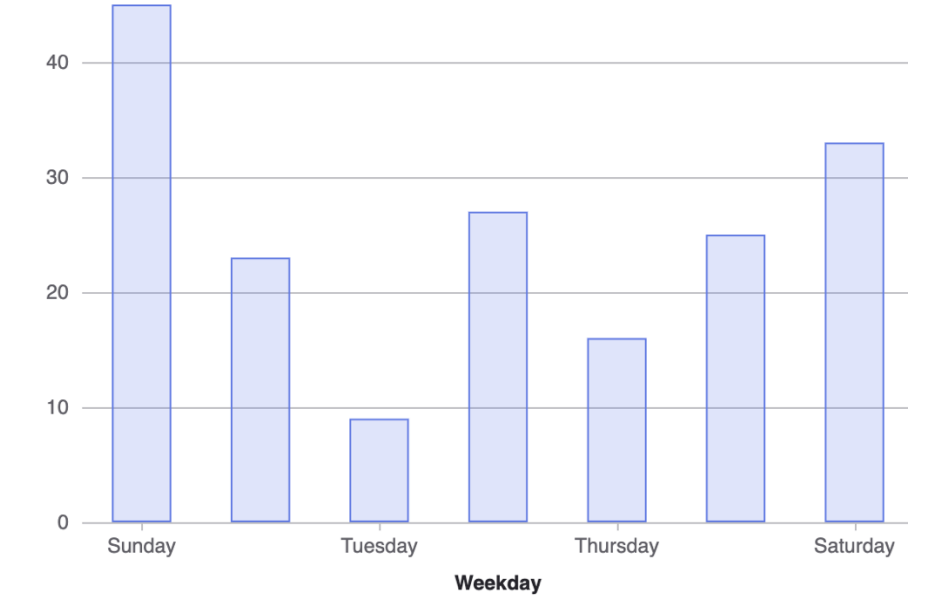
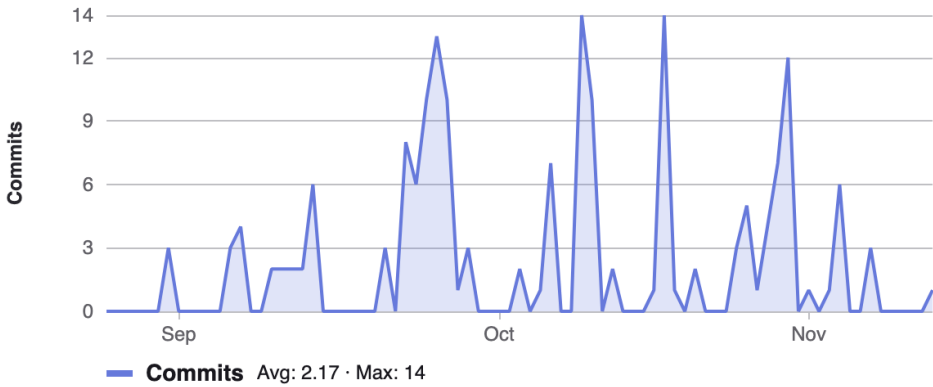

USER INTERFACE (1 of 1)



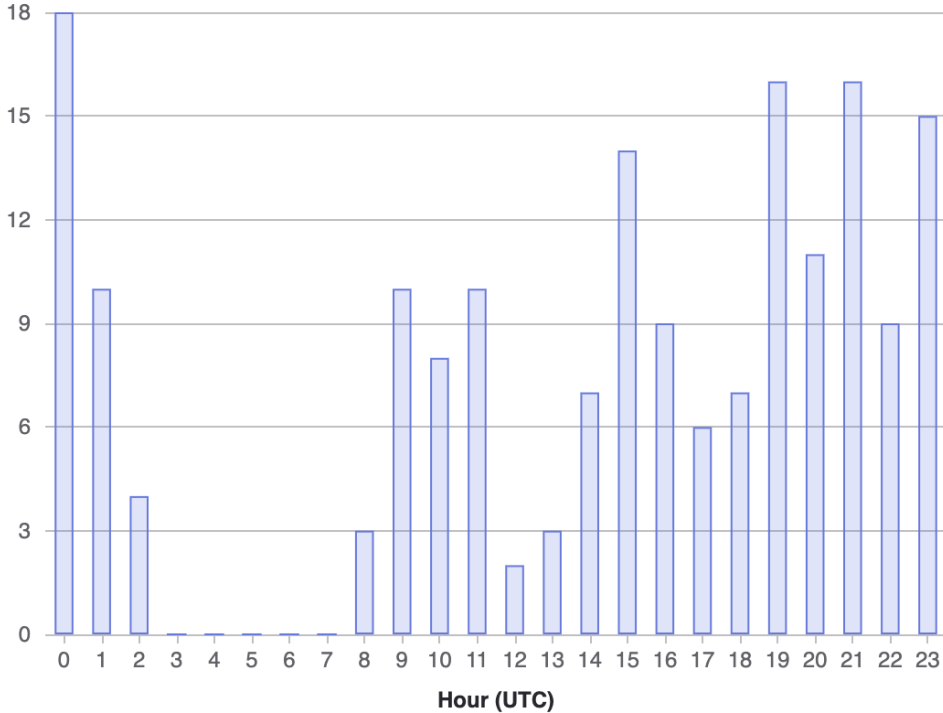
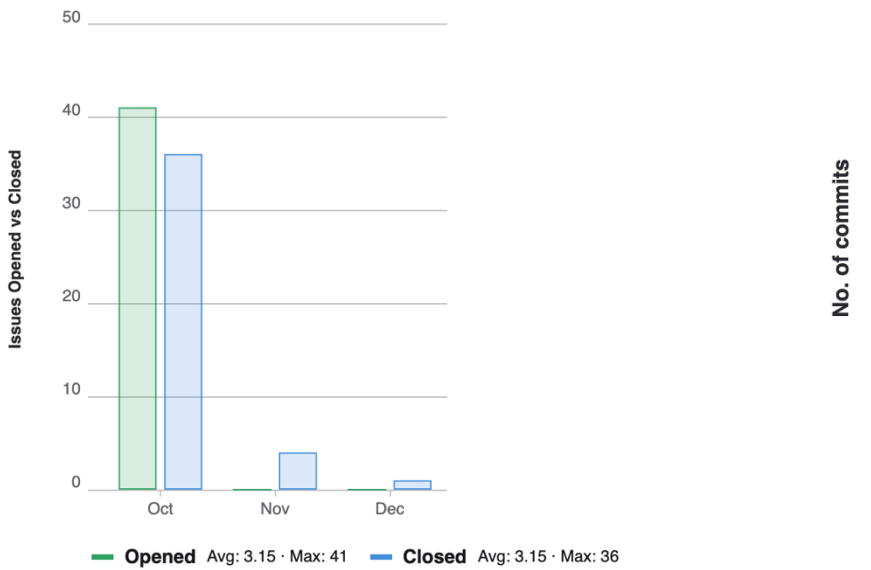
DEMO

Galasso Martina

176 commits (martina.galasso@student.supsi.ch)



Overview



CONCLUSIONS