FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Bringing Liveness to Design Patterns

Filipe Oliveira e Sousa Ferreira de Lemos

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Prof. Filipe Alexandre Pais de Figueiredo Correia

Co-Supervisor: Prof. Ademar Manuel Teixeira de Aguiar

July 24, 2020

Bringing Liveness to Design Patterns

Filipe Oliveira e Sousa Ferreira de Lemos

Mestrado Integrado em Engenharia Informática e Computação

July 24, 2020

Abstract

During the development of the software and, eventually, its maintenance, each element of a software team should be able to understand what patterns are being used to design the system, and why. This highlights the need to adequately document software. In fact, a good documentation practice is to describe each pattern instance that composes the software. When documenting one, we should give a quick visual overview of the design context, enhance the reason why it was instantiated, describe the design context in more detail, outline its role, illustrate the possible interactions between it and the client, provide the benefits and consequences of using it and, finally, identify special features of its implementation by referencing the source code. Additionally, since the software is constantly evolving, regular documentation updates are required to keep it consistent since existing instances might be affected by it and should be updated accordingly.

Approaches to software documentation with design patterns can either be based on documenting pattern instances or by an automatic extraction of the pattern instances from the code. Code annotations, pattern instance validators and external documentation, such as hypertext or graphical representations, are some examples for the first category. However, none of these solutions encompass all the aspects that we consider important to tackle this problem. Some do not maintain close proximity between code and documentation, others do not update the documentation after changes in the code are introduced and, all present a very low liveness level.

The goal of this work is to address the challenge of understanding software in terms of its design patterns, by streamlining the creation and documentation of software, making it easier to switch between these phases. This was achieved by increasing the amount of feedback about the pattern instances presented in the code, while the developers are editing it, granting a higher degree of liveness. Moreover, our approach maintains close proximity between the artifacts and provides means to visualize the pattern instances under the same environment. A prototype — the Design-Pattern-Doc — was designed, as a plugin for Intellij, that identifies pattern instances in the source code, live-suggests and generates pattern instance documentation, and provides live editing and visualization of the pattern instances.

To validate this approach, we carried out a controlled experiment with Informatics master students, where they had to complete several tasks, for which they had to understand and evolve a software system. The participants were divided in two groups, where one of the groups had access to the developed prototype. During these experimental sessions, different sorts of data were collected, such as task duration, number of context switches, among others, which were subject to a statistical analysis. The results support answering this thesis' research questions. In fact, they helped confirming that, with our approach, developers spend less time understanding and documenting a software system, based on design patterns. Additionally, the context switching, due to the transition between the programming and documenting phases, is reduced by embedding the latter in the IDE and by maintaining it close to the source code. Our approach does not solve the inconsistency problem, but we provide the means to reduce their occurrence.

Keywords: liveness, design patterns, documentation, consistency

Resumo

Durante o desenvolvimento do *software* (e eventualmente na sua manutenção), cada elemento duma equipa de software deve ser capaz de entender que padrões estão inseridos no sistema e por quê. Isso demonstra a necessidade de documentar adequadamente software. Uma boa prática na documentação de software é a descrição de cada instância de padrão que compoe o sistema. Quando se documenta uma, devemos fornecer uma rápida referência visual do contexto do design, especificar o motivo pelo qual foi instanciado, descrever o contexto do design com mais detalhes, explicar o seu papel, ilustrar as possíveis interacções entre o cliente e o padrão, fornecer os benefícios e as consequências do seu uso e, finalmente, identificar características especiais da sua implementação, fazendo referência ao código-fonte. Além disso, como o *software* está em constante evolução, é necessário recorrer a actualizações regulares da documentação de modo a mantê-la consistente, pois instâncias de padrões que já existam, poderão ser afectadas e devem ser actualizadas de acordo.

A documentação de software, com base em padrões de *design*, pode ser baseada na documentação das instâncias de padrões ou através da extracção automática de instâncias de padrões do código. Anotações no código, validadores de instâncias de padrões e documentação externa, são alguns exemplos da primeira categoria. No entanto, não se conhece nenhuma solução que contemple todos os aspectos que consideramos importantes para resolver este problema. Alguns não mantêm proximidade entre o código e a documentação, outros não actualizam a documentação após introdução de alterações no código e, todos apresentam um nível de *liveness* muito baixo.

O objectivo deste trabalho é endereçar o desafio da compreensão de *software* em termos dos seus padrões de design, através da simplificação da criação e documentação de *software*, facilitando a transição entre estas fases. Isto foi atingido através do aumento da quantidade de *feedback* sobre as instancias de padrões inseridas no código, enquanto os programadores o editam, garantindo um maior nivel de *liveness*. Para além disso, a nossa abordagem mantém proximidade entre os artefactos e promove os meios para visualizar as instâncias de padrões dentro do mesmo ambiente. Um protótipo — Design-Pattern-Doc — foi desenvolvido, como um plugin para o Intellij, capaz de identificar instâncias de padrões no código, sugerir e gerar documentação para as instâncias de uma forma *live* e, permite a edição e visualização das instâncias de padrão em *real-time*.

Para validar esta abordagem, foi realizada uma experiência controlado com alunos do mestrado integrado em informática, onde tiveram de concluir diversas tarefas, para as quais foi necessário que percebessem e evoluíssem sistemas de software. Os participantes foram divididos em dois grupos, sendo que um deles tinha acesso ao protótipo desenvolvido. Durante as sessões experimentais, diferentes tipos de dados foram recolhidos, como a duração das tarefas, o número de trocas de contexto, entre outros, os quais foram submetidos a uma análise estatística. Os resultados permitem responder as perguntas de investigação desta tese. Na verdade, estes ajudaram a confirmar que, com a nossa abordagem, os programadores gastam menos tempo a perceber e documentar sistemas de software, baseados em padrões. Adicionalmente, reduz-se a necessidade de troca de contexto, entre as fases de programação e documentação, através da inclusão da documentação e sua edição no próprio IDE e por estar próxima do código. A nossa abordagem não resolve o problema de inconsistências mas fornece os meios para reduzir a sua ocorrência.

Keywords: liveness, design patterns, documentação, consistência

Acknowledgements

First I'd like to thank my supervisors Professor Filipe Alexandre Pais de Figueiredo Correia and Professor Ademar Manuel Teixeira de Aguiar for all their help, availability and enthusiasm. I could always count on them whenever I ran into a trouble spot or had a question about my research or writing.

Thanks to all my friends and colleagues that helped and encouraged me along the way. Special thanks to my friend Diogo Torres for his help during the debugging sessions. I'd also like to express my sincere gratitude to all the wonderful faculty at FEUP that have helped me learn and advance my education over the years.

Finally, I must express my very profound gratitude to my parents and my other family members for providing me with unfailing support throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you all.

Filipe Lemos

“Success is the sum of small efforts,
repeated day in and day out.”

Robert Collier

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
ARG	Attributed Relational Graph
ASG	Abstract Syntax Graph
AST	Abstract Syntax Tree
BDD	Binary Decision Diagram
CG	Control Group
ED	External Documentation
EDP	Elemental Design Patterns
EG	Experimental Group
FAMIX	Famoos Information Exchange
FEUP	Faculty of Engineering of the University of Porto
GoF	Gang of Four
HTML	HyperText Markup Language
ID	Internal Documentation
IDE	Integrated Development Environment
LP	Literate Programming
MESW	Master in Software Engineering
MIEIC	Master in Informatics and Computing Engineering
MSc	Master of Science
UML	Unified Modeling Language
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Context

Throughout the entire life cycle of a software project, developers store knowledge in software artifacts. These artifacts, which are often part of the system itself, help them or other developers understand what has been done, how does it work and why it was made that way. Note that, this is not only important for new developers that have never interacted with the project before. Even the original developers, after some time without tackling a certain design detail (e.g. class, method, module, among others), tend to lose some of the knowledge they once had, making these artifacts particularly useful when they are trying to reconnect with it.

However, software is constantly evolving. As a result, these artifacts must be updated when new knowledge is acquired. This may not be a trivial task due to the structure of these artifacts. Some can have a richer and predefined structure, whose form must be respected, making the update of its contents harder due to their strict nature. Others may be free of structure, like free-text documents, which provide flexibility for evolving their contents but, due to their abstract structure nature, knowledge may be harder to retrieve [10, 9]. Moreover, the latest are harder to keep consistency among the rest of the artifacts, in other words, to make sure that two related artifacts do not express contradictory ideas.

Actually, there are several types of documentation artifacts. Use cases, sequence diagrams, class diagrams and other Unified Modeling Language (UML) diagrams help describing the functionality, architectural and design details of software products. Project work plans, risk management reports, among others, focus on the actual development process of the project. User manuals and API documentation provide guidance for new users to use the software. Even source code can be seen as a documentation artifact. In fact, all these artifacts have one thing in common: they are all affected by the constant evolution of software.

Aguiar stated that "*Before reusing a piece of software, one first has to invest time on understanding and learning how to use it. The more complex a reusable piece of software is, the more difficult and time consuming will probably be these tasks*" [1]. This denotes the importance of documenting software. Without having a clear notion of what it does and how to use it, we may end up breaking its original design purpose and benefits, when trying to increment functionalities or modify existing ones. This outcome is not that uncommon due to the lack of good practices and documentation standards, used by most developers, when documenting software design.

Design patterns offer reusable solutions to recurrent design problems, being particularly useful when designing and describing complex object-oriented software systems. In fact, a good way to document software is by describing the patterns instances that compose it. Pattern instances provide a detailed description of the design context and state the reason why those design patterns were instantiated, in the first place. Due to the patterns globally known vocabulary, sharing knowledge and design experience with other developers, becomes an easier task. Together with some good documentation practices, to enforce consistency during software evolution and maintenance, documentation using design patterns could lead to a standard for software documentation.

1.2 Motivation

During the process of software development, the developer switches, essentially, between two phases: programming and documenting. Since the output of one phase serves as input for the other, we have a mutual feedback loop. This loop enforces a constant swap of context, which may lead to the loss of important knowledge. In fact, the human brain has a limited capacity of working memory resources, also known as cognitive load. Thus, when heavily loaded, it may produce negative effects on task completion, such as the one mentioned above.

Another common consequence is the constant postponing of one of those two phases, usually the documentation phase, in order to remain longer in the same context. This may lead to inconsistencies since one of the artifacts is not evolving with the other. We may try to sync them back together later but, the more we increase the volume of changes before updating them, the harder it gets. This will have a critical impact in the reuse and the ability to understand the software, since we will be dealing with outdated or inconsistent documents.

1.3 Objectives

The purpose of this thesis is to streamline the creation and documentation of software, making it easier to switch between:

- The design of software with design patterns
- The documentation of software through design patterns

Achieving this requires tackling some concerns, such as finding out how to maintain consistency between code and its documentation, how to display the pattern instances to the developer and how to support the identification of design patterns in the source code. Perhaps, increasing the liveness level would help solving these concerns, since we would have a more clear perception of the information that transits between those phases. Here, by liveness we refer to the feedback provided to the developer when the software system is being designed or executed. In this particular case, the feedback could be about the pattern instances that should be documented, while the system is being designed.

For that reason, the research questions explore whether it is possible to reduce the inconsistencies between code and documentation and to improve the overall perception of the system's design by reducing the mutual feedback loop.

The final product should allow analyzing design patterns by verifying if they respect their rules, generating the required documentation and, alerting the developer every time an inconsistency is found between the documentation and its code (e.g. missing the implementation of a pattern participant). Thus, we expect that developers that use the tool respond faster to errors or inconsistencies, the total number of inconsistencies decreases, and the identification of pattern instances and the update of their documentation is made easier to perform.

1.4 Contributions

The outputs of this thesis are:

- **State of the art Review** — During which we compare the approaches and/or tools that try to solve the challenge of understanding and documenting a software system, based on design patterns.

- **Approach** — Our solution covers some of the flaws from the explored approaches, by maintaining proximity between the code and its documentation, proving the means required to ease the update of the documentation, and by increasing the liveness level.

- **Design-Pattern-Doc** — A prototype was developed, in the form of a plugin, to provide a concrete implementation of our approach to the participants of the controlled experiment.

- **Controlled User Experiment** — A controlled experiment with MSc students was performed to validate our approach.

1.5 Thesis Overview

In the following chapter, background regarding documentation good practices, visualisation and user interaction in development environments and, the concept of liveness will be given. In Chapter 3, current approaches on documentation with design patterns will be analysed to the very detail, exploring how they document pattern instances and how they they extract design patterns from application' source code.

Chapter 4 briefly describes the scope of the problem and the validation methodology used. Additionally, it presents the hypothesis and research questions that support this thesis.

Chapter 5 presents our approach to the challenge of understanding software in term of its design patterns. The developed prototype — **Design-Pattern-Doc** — is also presented, more specifically its architectural design and its main features.

In Chapter-6 we explore the controlled user experiment used to validate our approach, by presenting its goals, design and results. At the end of this chapter, the research questions presented in Chapter 4 are answered.

Chapter 7 provides a quick overview of this thesis and explores possible future steps.

Chapter 2

Background

In this chapter, we describe and contextualize the topics required to fully comprehend this document. We begin by exploring a set of best practices for documenting software. These will be an important criteria for comparing the different approaches and tools, in the State of the Art (Chapter 3). Then we will examine how development environments display information and interact with the user in a non-intrusive way. This will be followed by a brief analyze of the definition of Liveness and Literate Programming.

2.1 Documentation Good Practices

Software documentation is the collection of knowledge, captured in several artifacts, that may evolve throughout the entire software life cycle. Some of these artifacts may address related information, meaning that when updating one, the related artifacts should also be updated to avoid inconsistencies.

On the following subsections, we will be examining some good practices, that have been presented in the literature, to consistently document software [11]. Note that, there are several other documentation good practices. However, we will only be covering those that are required to understand the following chapters.

2.1.1 Information Proximity

The Information Proximity is a pattern that tries to solve the problem of preserving documentation consistency when related information is scattered across documents/artifacts. Easing the access to artifacts that contain related information, by using links, a single source, transclusion or views, is their solution to this problem.

Links should be used when related contents are not meant to be in the same document. On the other hand, single source is meant to capture related contents on the same document, making them easier to maintain. Transclusions are used when we want to import information that exists on other artifacts to another document. This is done by referring them. Finally, views are documents that only contain information presented on other artifacts.

2.1.2 Co-Evolution

The Co-Evolution is a pattern that tries to solve the problem of updating related information scattered across artifacts, more precisely, when should it be updated. Based on their experience and existing literature, they've realized that every time a change is introduced, all the related pieces of information should be updated.

The more we increase the number and volume of changes, before updating the corresponding artifacts, the harder it gets to sync them. If the cost for recovering consistency later is high or when documentation must always be consistent, we should use synchronous co-evolution (update immediately after changes occur). When this is not the case, we may use the time-shifted co-evolution. Here, pending related changes are tracked and they are only updated when needed.

2.1.3 Domain-Structured Information

The Domain-Structured Information is a pattern that tries to solve the problem of structuring the information in documentation, in order to automatically assess consistency among the related artifacts. This is achieved by formalizing contents according to their domain concepts, which avoids multiple interpretations of the same information.

2.1.4 Integrated Environments

Integrated Environments is a pattern that tries to solve the problem of maintaining consistency among independent but content related artifacts. These environments do not only allow handling several types of artifacts but, also provide a structure for them to inter operate with each other. During Section 2.3, we will be analyzing two of the most popular integrated development environments.

2.2 Literate Programming

Literate programming (LP), which is an example of the documentation good practices discussed above, was invented by Knuth [30] to help humans comprehend software programs. LP achieves this by switching the focus from instructing the computer what to do, to explain to humans what the computer is designed to do. It has three main features:

- **Verisimilitude** - Source code and documentation are written and placed together in the same source document (Information proximity). This technique helps preserving consistency between the source code and its documentation.

- **Psychological arrangement** - Unlike traditional approaches, a literate program is organized in the most appropriate way to enhance human comprehension, not being restricted to respect the structure accepted by the compiler.

- **Good readability** - Programs are considered to be works of literature. Therefore, they should be easy to read and comprehend.

Literate programs are unified source-documents composed by interleaves of chunks of source code and explanatory text, with a psychological arrangement, to help describing the program to human beings. Since these programs are not source code nor documentation, they are commonly known as web documents. This term was derived from the notion of web of information.

Unfortunately, there are some problems that make LP not widely used by programmers. Firstly, it provides too much overhead for small programs (programming language, formatting language, language for relating the previous two languages). Secondly, most tools do not support the documentation of existing software, only new. Finally, their program arrangement makes it hard for the source code to be manipulated by some tools, such as refactoring tools or debuggers.

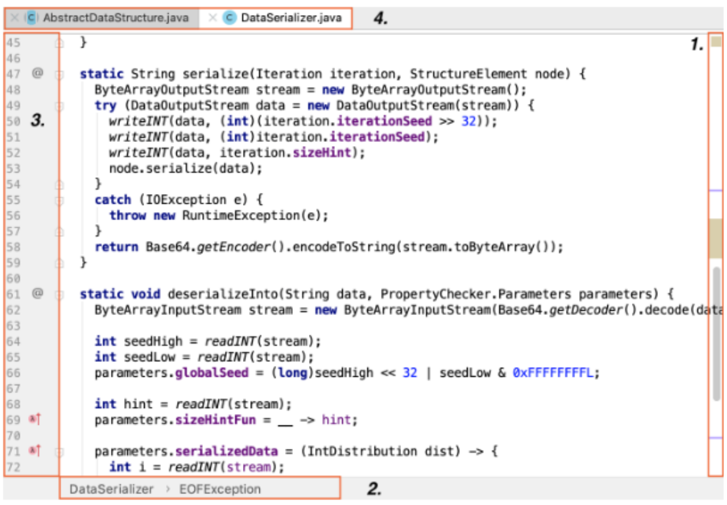
2.3 Integrated Development Environment

Integrated Development Environments (IDE) are software products designed to maximize developers productivity, during the software development process. Their offer a set of functions and tools such as debuggers, refactors, code inspectors, different perspectives to view the source code, among others, that help manipulating the source code. In this section, we will be analysing how two of the most popular IDEs, Intellij[1] and Visual Studio Code[2], provide software visualization and user interaction.

Intellij was created by JetBrains[3] to maximize productivity when programming in Java. They have developed several interesting user interaction features, such as smart code completion, framework specific assistance (understands and provides intelligent coding for several other languages, even if they are injected in a string literal in the Java code), and productivity boosters (predicts needs and automates repetitive tasks). Unlike many IDEs, Intellij is context aware and, automatically, displays the tools that make sense in the current context.

Intellij's editor is represented in Figure 2.1. On top of the active file, there is a tab layout with all the previously opened files. To its left, there is a gutter that displays the line numbers, annotations and breakpoints. Breadcrumbs, which help the user navigate inside the code, are displayed at the bottom of the file. Finally, the file scrollbar, also known as error stripe, highlights the errors and warning, which can be differentiated through their colour.

[1]More information on `https://www.jetbrains.com/help/idea`
[2]More information on `https://code.visualstudio.com/docs/`
[3]More information on `https://www.jetbrains.com/`



```java
  }

  static String serialize(Iteration iteration, StructureElement node) {
    ByteArrayOutputStream stream = new ByteArrayOutputStream();
    try (DataOutputStream data = new DataOutputStream(stream)) {
      writeINT(data, (int)(iteration.iterationSeed >> 32));
      writeINT(data, (int)iteration.iterationSeed);
      writeINT(data, iteration.sizeHint);
      node.serialize(data);
    }
    catch (IOException e) {
      throw new RuntimeException(e);
    }
    return Base64.getEncoder().encodeToString(stream.toByteArray());
  }

  static void deserializeInto(String data, PropertyChecker.Parameters parameters) {
    ByteArrayInputStream stream = new ByteArrayInputStream(Base64.getDecoder().decode(data

    int seedHigh = readINT(stream);
    int seedLow = readINT(stream);
    parameters.globalSeed = (long)seedHigh << 32 | seedLow & 0xFFFFFFFFL;

    int hint = readINT(stream);
    parameters.sizeHintFun = __ -> hint;

    parameters.serializedData = (IntDistribution dist) -> {
      int i = readINT(stream);
```

Figure 2.1: Intellij editor

Furthermore, by right clicking on variables, expressions, methods and so on, the user is presented with a list of possible refactoring processes. Live templates can also be used to insert common code structures like loops, conditions. These contain only plain text that when expanded insert the entire block of code. For example, *sout* is a live template for "`System.out.println();`".

Another relevant feature, particularly when dealing with design patterns, is their UML class diagram generation, which allows reflecting the structure of the actual source code classes (Figure 2.2).

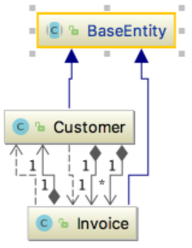
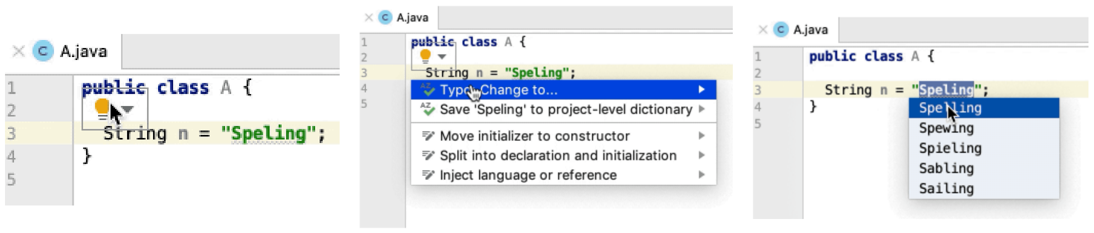


Figure 2.2: Intellij UML class diagrams

Regarding liveness, their most interesting feature is the code inspection, which detects, suggests and corrects wrongly written code, before the source code is compiled. These allow detecting unreachable code, spelling problems and bugs, by highlighting the respective blocks of code. For example, unreachable code is normally greyed out, while spelling mistakes are underlined. When hovering on top of these pieces of code, a yellow light bulb pops up, which by clicking on it, offers a set of suggestions to fix those problems. Figure 2.3 represents this process for a spelling mistake.



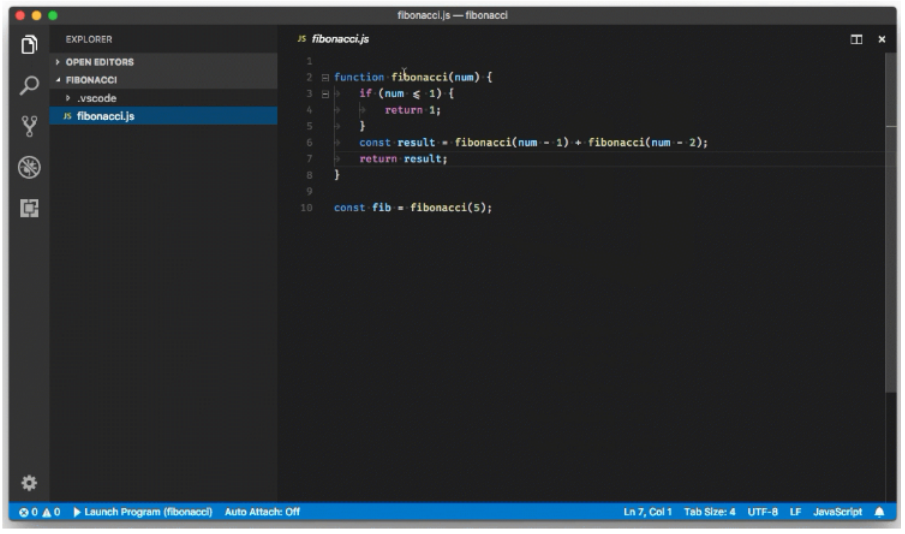
```java
public class A {
    String n = "Speling";
}
```

A.java
Type: Change to...
Save 'Speling' to project-level dictionary
Move initializer to constructor
Split into declaration and initialization
Inject language or reference
Spelling
Spewing
Spieling
Sabling
Sailing

Figure 2.3: Using code inspections to fix spelling mistake

Visual Studio Code was created by Microsoft[4] as a lightweight multi platform source code editor. Intelligence code completion and powerful editing like linting, multi-cursor editing or parameter hints, are some of the features that can be found within this IDE.

Visual Studio Code editor can be seen in Figure 2.4. On its left side we have a sidebar with six icons: 1) File explorer, 2) Search across files, 3) Source code management, 4) Program launch or debug, 5) Marketplace and extensions manager and 6) Settings. On the bottom of the window, we can find the status bar, where, among other details, the errors and warnings are displayed. Just like Intellij, on the top of the active source file, there is also a tab layout with all the previously opened files.

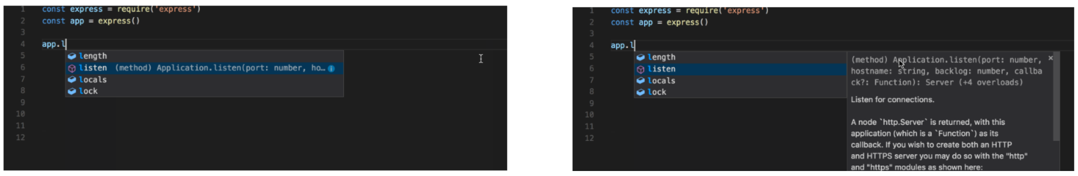


```javascript

function fibonacci(num) {
    if (num <= 1) {
        return 1;
    }
    const result = fibonacci(num - 1) + fibonacci(num - 2);
    return result;
}

const fib = fibonacci(5);
```

Figure 2.4: Visual Studio Code editor

IntelliSense is the name they give to their code completion features. Through source code analysis and language semantics, they are able to suggest possible completions while the user is typing. As the user types, the hint list gets filtered. By pressing the info icon or selecting "*quick info*" in any of those list elements, its documentation gets expanded (Figure 2.5). Moreover, after choosing a method, the user is provided with the parameter info. Another thing that appears when using IntelliSense are code snippets. These are the equivalent to the live templates discussed in Intellij.

[4]More information on https://www.microsoft.com



```
const express = require('express')
const app = express()

app.l
```

```
const express = require('express')
const app = express()

app.l
```

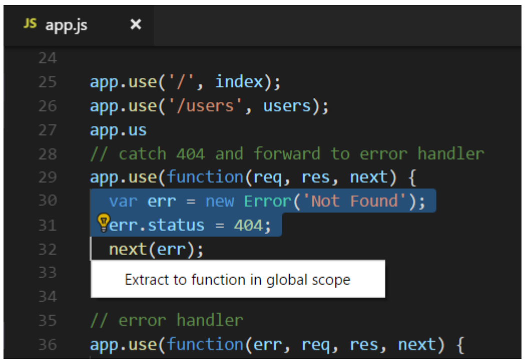
```
(method) Application.listen(port: number,
hostname: string, backlog: number, callba
ck?: Function): Server (+4 overloads)
```

Listen for connections.

A node `http.Server` is returned, with this application (which is a `Function`) as its callback. If you wish to create both an HTTP and HTTPS server you may do so with the "http" and "https" modules as shown here:

Figure 2.5: Documentation pop up with code completion

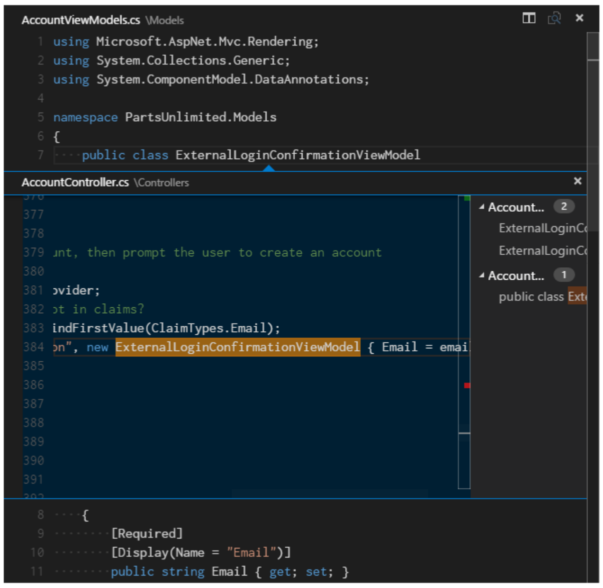
Visual Studio Code, through code inspection, can detect and highlight issues for quick fixes or refactoring purposes. Just like Intellij, a light bulb pops up when the cursor is on top of these highlighted issues (Figure 2.6).



```

app.use('/', index);
app.use('/users', users);
app.us
// catch 404 and forward to error handler
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);


// error handler
app.use(function(err, req, res, next) {
```

Figure 2.6: Visual Studio Code refactoring suggestion

Additionally, it has two particularly interesting features. The first one is the integrated terminal, which allows performing a quick command-line task (like starting a pattern detection tool) without having to switch windows. The second one is their code navigation feature peek. This feature allows embedding the search result inline, avoiding having to change context (Figure 2.7).



AccountViewModels.cs \Models

```
using Microsoft.AspNet.Mvc.Rendering;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace PartsUnlimited.Models
{
    public class ExternalLoginConfirmationViewModel
```

AccountController.cs \Controllers

```


unt, then prompt the user to create an account

ovider;
ot in claims?
indFirstValue(ClaimTypes.Email);
on", new ExternalLoginConfirmationViewModel { Email = email
```

```
    {
        [Required]
        [Display(Name = "Email")]
        public string Email { get; set; }
```

Figure 2.7: Visual Studio Code Peek

2.4 Liveness

Live programming is a way to support *technical agility* — it is the application of the *inspect and adapt* principle used in agile methods to programming activities. This constant awareness of the program state helps reducing the edit-compile-run cycle and, therefore, reduces the the total time required to build the system. Note that, it can also be particularly useful during program maintenance or for debugging purposes.

Tanimoto [47] proposes a hierarchy of six levels to define liveness:

1. Informative
2. Directly requested feedback
3. Delayed non requested feedback
4. Instantaneous feedback
5. Predictive and suggestive feedback
6. Strategically predictive

Level one is merely informative, like providing which design patterns were detected, in text format or in a UML diagram. Level two provides feedback when the user asks for it (e.g. user requests feedback by pressing a button designed for that purpose and, an informative box is displayed on the screen) . In level three, the feedback is provided some time after an event (delayed automatic response). Warning the developer that some method is missing, a small number of seconds after saving the program, without direct request by the user, is an example for this level.

Level four provides feedback in real-time. The feedback is not requested by the user but can be linked to an event (e.g. running the program in background to search for design patterns and, after a small number of seconds without typing, results are displayed on the screen). In level five, while running the program in background, the system predicts future actions and suggests them to the developer. Finally, level six system's automatically foresee the future actions and implement them, without asking the user for permission.

Chapter 3

State of the Art

Information about the pattern instances of a code base can be made available by either creating documentation that describes such instances, or by extracting patterns from the source code. In this chapter, we will explore the approaches and/or tools, developed so far, to tackle these problems.

3.1 Documenting pattern instances

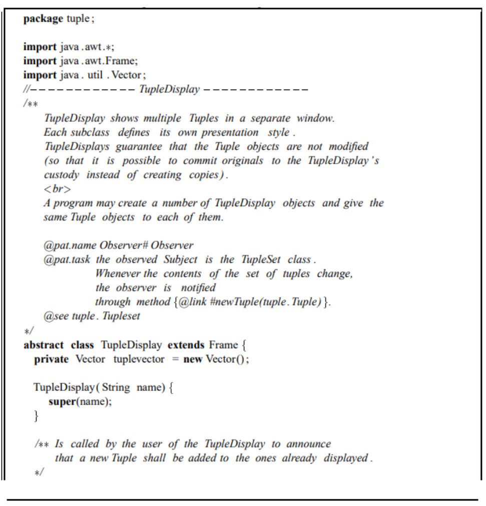
Pattern instances have been documented through text-based approaches, such as code annotations or HTML pages, graphical representations like UML diagrams or by filling out pattern code templates. In the following sections, approaches related to each one of these categories will be analyzed.

3.1.1 Text-based

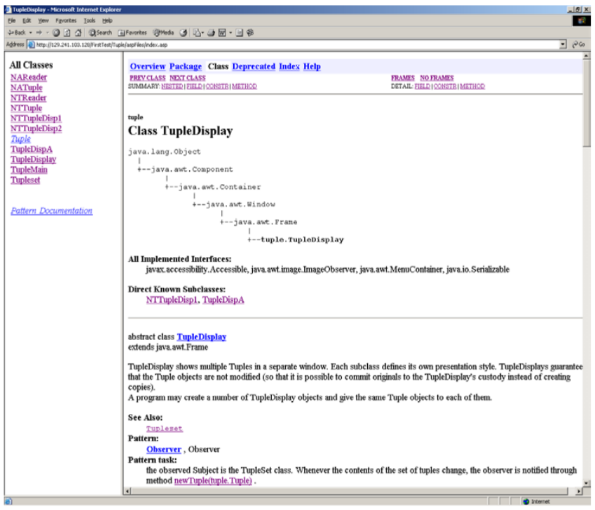
This is the most commonly used approach. It can be achieved by annotating the source code with comments or by providing external documentation pages [41, 49, 27, 37, 26]. A good example of a tool that belongs to this category is the Javadoc [32]. Javadoc is a tool designed for the automatic generation of documentation, from Java source code, with the goal of displaying and distributing on the Web. Through a documentation syntax for comments, which are embedded in the source code, and the help of the Java compiler, it produces web pages describing classes, interfaces, methods, etc. for an online, hypertext-based documentation.

Even though most of these approaches assure information proximity, co-evolution is hardly supported. For instance, Sametinger et al. [41] uses links to automatically update the documents when the code annotations change. However, these annotations are not updated when the source code is modified, which may lead to inconsistencies.

These approaches normally start by defining new tags for documenting patterns and a doclet, that processes these tags, producing the corresponding documentation output. A doclet is a program created with the doclet API, which specifies the format and content of the data to be provided. An example of these source code tags and documentation output are shown in Figure 3.1.



```
package tuple;

import java.awt.*;
import java.awt.Frame;
import java.util.Vector;
//------------- TupleDisplay -------------
/**
    TupleDisplay shows multiple Tuples in a separate window.
    Each subclass defines its own presentation style.
    TupleDisplays guarantee that the Tuple objects are not modified
    (so that it is possible to commit originals to the TupleDisplay's
    custody instead of creating copies).
    <br>
    A program may create a number of TupleDisplay objects and give the
    same Tuple objects to each of them.

    @pat.name Observer# Observer
    @pat.task the observed Subject is the TupleSet class.
              Whenever the contents of the set of tuples change,
              the observer is notified
              through method {@link #newTuple(tuple.Tuple)}.
    @see tuple.Tupleset
*/
abstract class TupleDisplay extends Frame {
  private Vector tuplevector = new Vector();

  TupleDisplay(String name) {
     super(name);
  }

  /** Is called by the user of the TupleDisplay to announce
      that a new Tuple shall be added to the ones already displayed.
  */
```



TupleDisplay - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://129.241.103.128/FirstTest/Tuple/aspFiles/index.asp
Go
All Classes
NAReader
NATuple
NTReader
NTTuple
NTTupleDisp1
NTTupleDisp2
Tuple
TupleDispA
TupleDisplay
TupleMain
Tupleset
Pattern Documentation
Overview Package Class Deprecated Index Help
PREV CLASS NEXT CLASS
FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD
Internet

tuple

Class TupleDisplay

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--java.awt.Window
                    |
                    +--java.awt.Frame
                          |
                          +--tuple.TupleDisplay
```

All Implemented Interfaces:
javax.accessibility.Accessible, java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable

Direct Known Subclasses:
NTTupleDisp1, TupleDispA

abstract class TupleDisplay
extends java.awt.Frame

TupleDisplay shows multiple Tuples in a separate window. Each subclass defines its own presentation style. TupleDisplays guarantee that the Tuple objects are not modified (so that it is possible to commit originals to the TupleDisplay's custody instead of creating copies).
A program may create a number of TupleDisplay objects and give the same Tuple objects to each of them.

See Also:
Tupleset

Pattern:
Observer , Observer

Pattern task:
the observed Subject is the TupleSet class. Whenever the contents of the set of tuples change, the observer is notified through method newTuple(tuple.Tuple) .

Figure 3.1: A class with patterns documented (Left) and an external documentation example of the same class (Right) as presented by Norges et al. [26]

What usually differentiates these approaches is how they describe their pattern instances. Odenthal et al. [37] presents the pattern instance structure shown in Table 3.1.

Table 3.1: Pattern instance structure by Odenthal et al. [37]

Label	Description
Overview	Reference to a class diagram or other visualization mechanism to give a first glimpse of the design context
Intent	Reason for instantiating the pattern
Motivation	Describe the context and the design actions that led to this pattern instantiation
Roles	Label pattern instance classes with "Pattern:Role". Briefly explain how each role contribute to the pattern
Collaborations	Interaction between the client and the pattern instance. Here, the developer may provide the methods that allow interaction among the different pattern participants
Consequences	Pros and Cons
Implementation	Special features of the implementation with references to the source code
Hot spots vs Frozen Spots*	Detailed description of the parts not to be changed and the parts to be extended by the user
Recipe*	A ready-to-use example
Integration*	Which assumptions the environment of the framework has to fulfill
Known uses*	Reference all sucessufull uses and common misuses
Structural extensions*	Aspects that are limiting the framework design and implementation

The labels marked with an asterisk are a later extension of their pattern instance description to contemplate flexibility aspects of patterns. Unfortunately, the cost/benefit ratio associated with pattern instances captured with such an extensive structure would not be positive. In fact, Hallum [26] uses only three labels to describe each pattern: 1) name, 2) task and 3) use. Torchiano [49] adds one more tag to those defined by Hallum which is the role of the class on that pattern instance. Finally, Sametinger [41] also uses three labels (pattern name, instance name and role) plus an optional text tag. The instance name tag allows multiple uses of the same pattern on the same source code.

3.1.2 Graphical representation

Design patterns are usually represented by standard notation UML diagrams, making this sort of representation familiar to developers. As a result, some approaches believe that the ability to visualize the system's design patterns in a graphical way, such as UML diagrams, can ease the comprehension of the software [48, 42, 19]. However, if the system is composed by a high number of classes, this sort of representation can get very hard to understand (Figure 3.2).

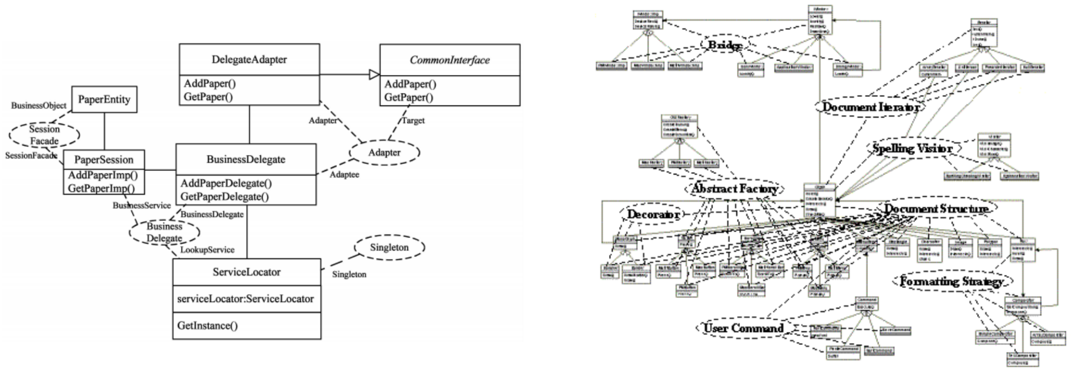


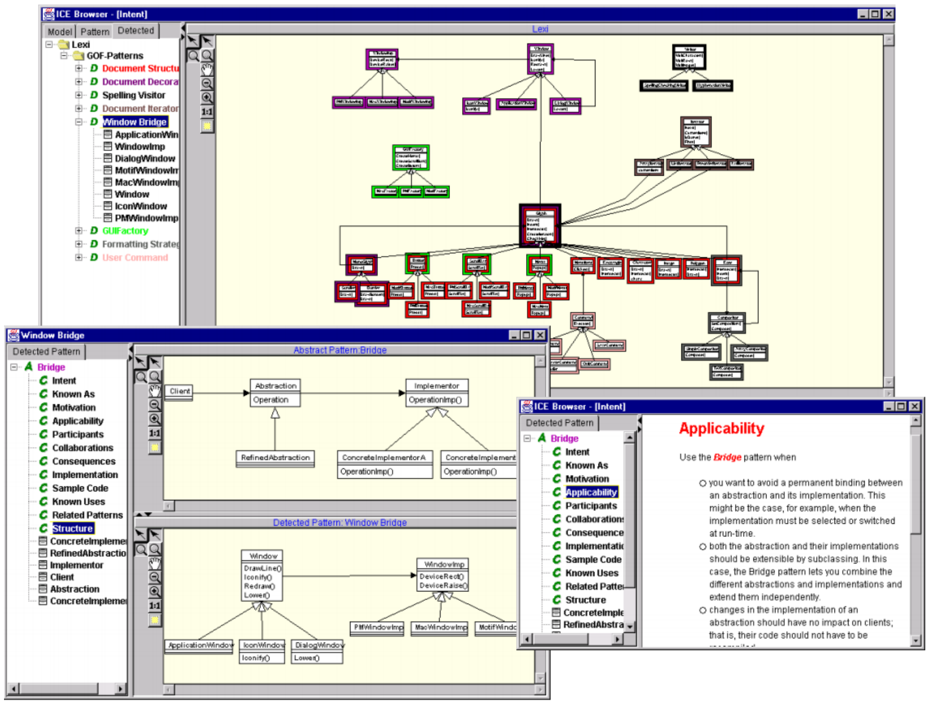
Figure 3.2: A simple design pattern instance diagram using the standard UML notation (Left) and a more complex diagram using the same notation (Right) as presented by Schauer et al. [42]

Schauer et al. [42] has implemented a few functionalities to the Lexi tool [23], which eases the comprehension of these type of complex system diagrams:

1. The ability to distinguish different patterns by applying a colorful border around each pattern;

2. The ability to zoom in/out in a pattern by highlighting the classes that play a role in that pattern, while opening a new window with more detailed information about it (Figure 3.3);

3. The system's pattern collaboration diagram which allows users to collapse some patterns in order to focus their attention on others, being the dominant class (normally the abstract class) the only one visible (Figure 3.4).



Applicability

Use the *Bridge* pattern when

- you want to avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when the implementation must be selected or switched at run-time.
- both the abstraction and their implementations should be extensible by subclassing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently.
- changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be

Figure 3.3: Overall system's design, with design patterns highlighted with different coloured borders (upper window), individual view of specific design pattern (lower, left window), design pattern instance (lower, right window) as presented by Schauer et al. [42]

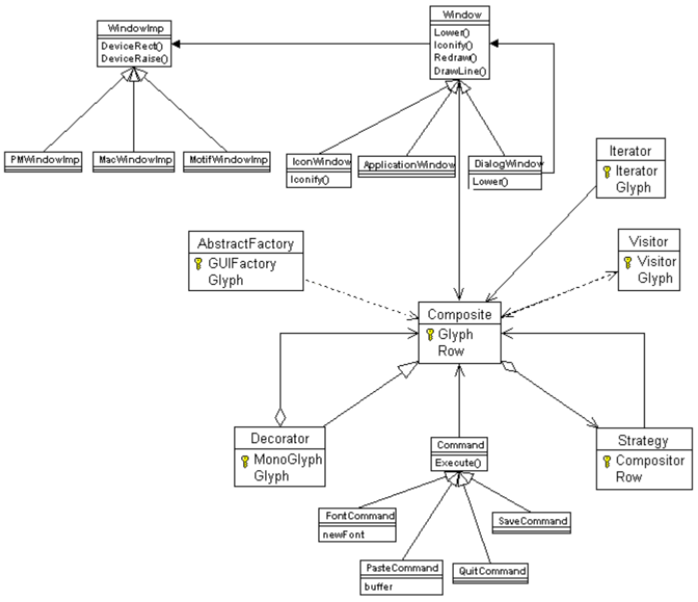


Figure 3.4: Collapsed design pattern UML notation as presented by Schauer et al. [42]

Trese et al. [48] describe a way to display design patterns implementation, with the participating classes in the same structure as they appear in their UML canonical form. Moreover, they suggest that *"showing only selected information specified by the canonical representation, leverages parallelism and established visual convention, thereby facilitating more rapid program* *understanding"*. Their technique tries to solve of the challenges regarding graphical representation of design patterns, such as dealing with multiple patterns in the same visualization and, using the design patterns standard representation.

The authors start by dividing the patterns into the Gang of Four (GoF) categories — creational, structural, behavioral, among others — and each group is placed in a different row on the patterns layout document (Figure 3.5). Each row is delimited by a thick black line to visually separate the different groups and, the thickness makes it distinguishable from relation arcs or class rectangle borders. Close space proximity of patterns is achieved by placing them on the same row with a small margin between them. Note that, these pattern instances are associated to their code implementation through their class name.

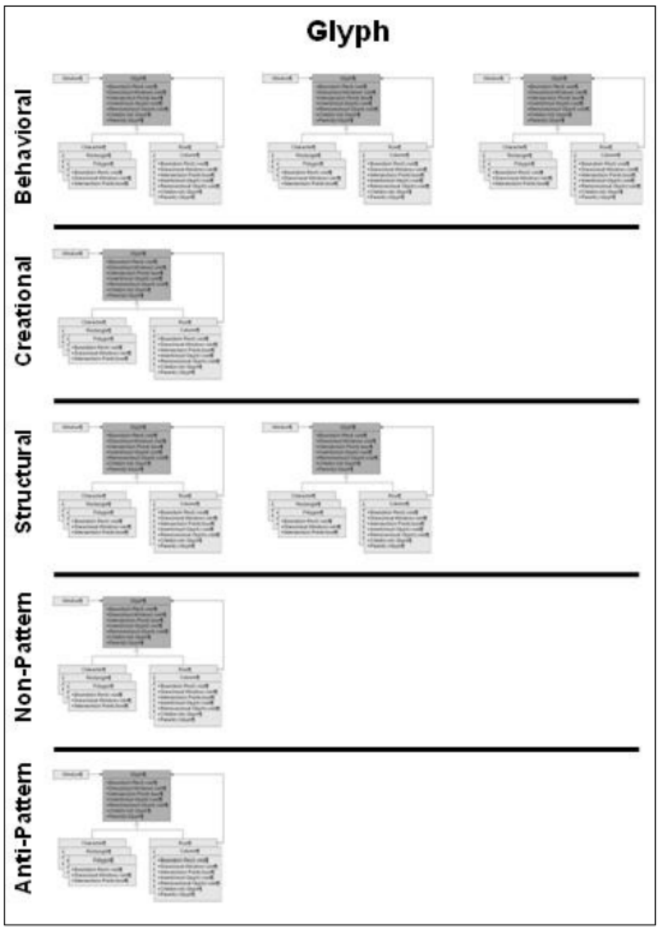


Figure 3.5: Layout of class design pattern information as presented by Trese et al. [48]

Dong et al. [19] used a technique to represent the roles that a class, method or attribute play in a design pattern, by attaching tagged values to the UML diagrams (Figure 3.6). This approach allows dealing with multiple instance of design patterns but introduces too much text, making it harder to read and increasing, significantly, the size of the diagrams.

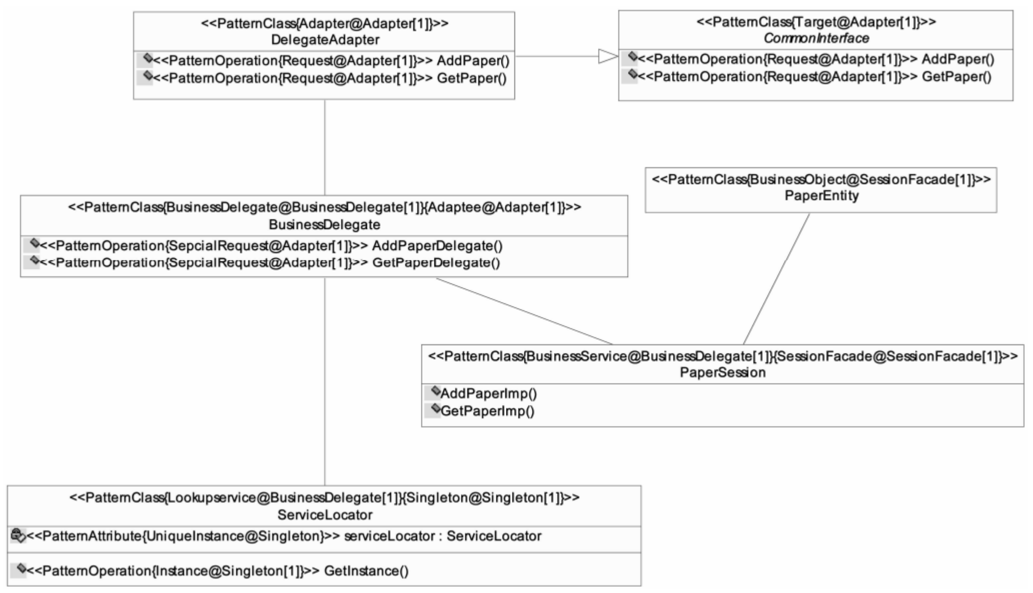

<<PatternClass{Adapter@Adapter[1]}>>
DelegateAdapter
<<PatternOperation{Request@Adapter[1]}>> AddPaper()
<<PatternOperation{Request@Adapter[1]}>> GetPaper()
<<PatternClass{Target@Adapter[1]}>>
CommonInterface
<<PatternOperation{Request@Adapter[1]}>> AddPaper()
<<PatternOperation{Request@Adapter[1]}>> GetPaper()
<<PatternClass{BusinessObject@SessionFacade[1]}>>
PaperEntity
<<PatternClass{BusinessDelegate@BusinessDelegate[1]}{Adaptee@Adapter[1]}>>
BusinessDelegate
<<PatternOperation{SepcialRequest@Adapter[1]}>> AddPaperDelegate()
<<PatternOperation{SepcialRequest@Adapter[1]}>> GetPaperDelegate()
<<PatternClass{BusinessService@BusinessDelegate[1]}{SessionFacade@SessionFacade[1]}>>
PaperSession
AddPaperImp()
GetPaperImp()
<<PatternClass{Lookupservice@BusinessDelegate[1]}{Singleton@Singleton[1]}>>
ServiceLocator
<<PatternAttribute{UniqueInstance@Singleton}>> serviceLocator : ServiceLocator
<<PatternOperation{Instance@Singleton[1]}>> GetInstance()

Figure 3.6: UML with attached tagged values by Dong et al. [19]

3.1.3 Pattern instance validators

Some approaches require the developer to write design patterns instances by filling out their pattern' code templates. These approaches use code checkers, similar to compilers, which allow validating the design pattern described by a specific pattern instance (if it respects its specification, such as intent, rules, participants, among others).

Cornils et al. [8] implemented the DPDOC tool to help documenting systems with design patterns. The tool's work space is separated in two: the program editor and the pattern applicator (special-purpose language to document patterns instances used in the program). The latter has a list of supported design patterns templates, which are meant to be filled by the developer, every time a design pattern is introduced in the system. The template's placeholders are class names, methods or variables that play specific pattern roles. After filling out these placeholders, the roles are tied to the program elements, allowing the developer to receive feedback via error boxes, when some method is missing or is not correctly implemented. This feedback is provided without an explicit request by the developer. Unfortunately, there isn't enough information to conclude when is the feedback displayed, whether it is when the user saves the project, compiles it or if it is linked to other sort of event.

Lovatt et al. [33] extended the Java conventional compiler, creating the Pattern Enforcing Compiler (PEC), to include the extra checks need to enforce design patterns. Each pattern has to follow a specific template structure, in order for the compilation using PEC to verify if the class is following the constraints and rules of the specified pattern. A list of interfaces for several design patterns has been developed. By making a certain class implement one of those interfaces (follow a specific pattern), the developer is letting PEC know that this class should be validated, during compilation time (Figure 3.7). This may be limiting, since developers may have reasons for naming their methods with different names, from those expected by the interface.

```
import pec.compile.singleton.*;

public final class
SingletonClass implements
Singleton {

  private final static
  SingletonClass instance = new
  SingletonClass();

  private SingletonClass() {
       if ( instance != null ) {
           throw new
           IllegalStateException(
           "Attempt to create a
           second Singleton" );
       }
  }

  public static SingletonClass
  instance() {
       return instance;
  }

  // other methods

}
```

```
package pec.compile.singleton;

import pec.compile.*;
import pec.compile.creational.*;
import
pec.compile.staticfactory.*;

public interface Singleton
extends Creational,
StaticFactory, CompilerEnforced
{
  /* Empty - no instance methods
  or static fields required by a
  Singleton */
}
```

Figure 3.7: Template for the Singleton pattern (Left) and the Singleton interface for the PEC (Right) as presented by Lovatt et al. [33]

Florijn et al. [22] implemented an object-oriented design environment prototype for design patterns, in Smalltalk. It uses fragments, which are design elements of different types such as classes, methods, associations, among others. These are used to represent pattern data (smalltalk objects, references to other fragments) and a template of patterns to generate program elements for a new pattern instance. These pattern instances are integrated into the program by binding the program elements to a role in the pattern. Like the other approaches, it checks if modified occurrences of patters still meet the patterns rules/constraints. Regarding visualization, the environment provides: a fragment browser with a global view on the fragment structure (Figure 3.8), displaying the relationship between fragments; the ability to collapse/expand nodes; the ability to mark nodes with different colors and an inspector window that allows focusing on a specific fragment, providing its details.

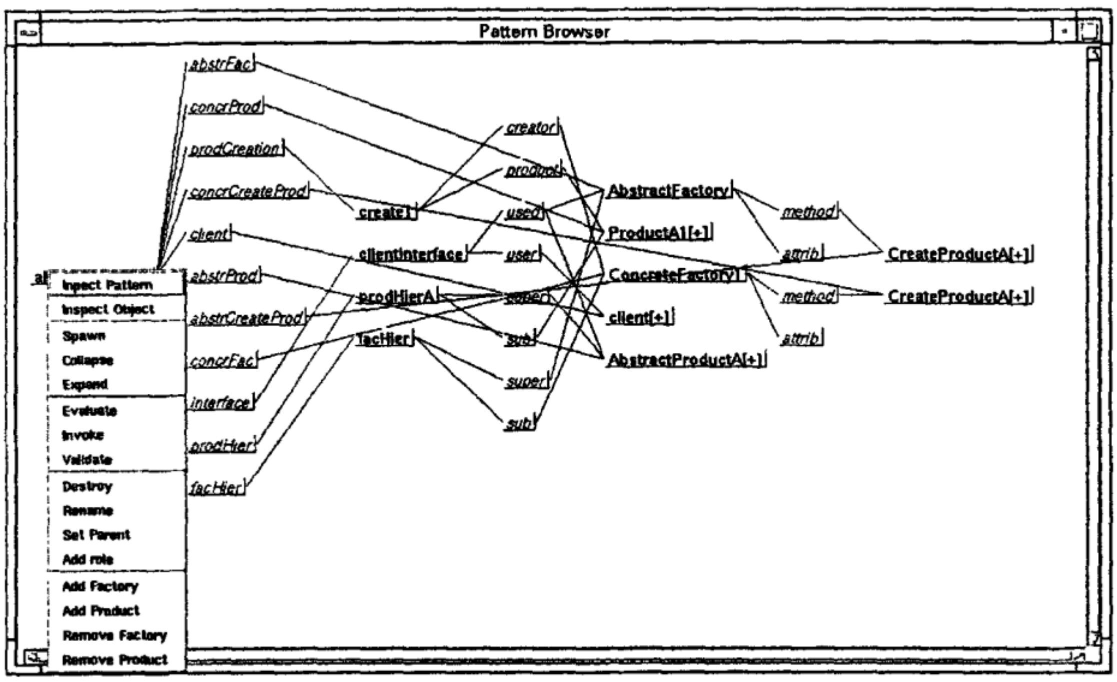


Figure 3.8: Fragment browser window as presented by Florijn et al. [22]

3.1.4 Discussion

The approaches for documenting pattern instances were categorized according to the good documentation practices introduced in Chapter 2 (information proximity and co-evolution), their liveness level, how they present the pattern instances descriptions and, whether they can validate the implemented patterns or not. Table 3.2 provides an overview of the analyzed approaches.

Table 3.2: Overview of the approaches for documenting pattern instances

Strategy	Approach	Information Proximity	Co-Evolution	Validates Patterns	Domain-Structured Information	Liveness	Visualization
Text-based	Sametinger et al. [41]	Yes	Yes	No	Yes	1	Code annotations, html pages
	Torchiano [49]	Yes	Yes	No	Yes	1	Code annotations, html pages
	Hallum [26]	Yes	Yes	No	Yes	1	Code annotations, html pages
	Odenthal et al. [37]	Yes	Yes	No	No	1	Code annotations
Graphical	Trese et al. [48]	No	No	No	No	1	UML
	Schauer et al. [42]	No	No	No	No	1	UML
	Dong et al. [19]	No	No	No	No	1	Javadoc html pages
Template	Florijn et al. [22]	No	No	Yes	No	N/a[1]	UML
	Lovatt et al. [33]	No	No	Yes	No	3	N/a
	Cornils et al. [8]	Yes	Yes	Yes	No	3	Boxes next to editor

[1]Not available.

As we can see from Table 3.2 none of the presented approaches will simultaneously 1) maintains proximity between documentation and source code, 2) updates this documentation when changes to the source code are introduced and 3) provides a liveness level above 3:

- Every text-based strategy assures information proximity. Due to this fact, they also support co-evolution, since information proximity helps to quickly notice when two pieces of related content stop being consistent with each other. Moreover, the approaches that extend Javadoc also support domain-structured information, since they introduce an increased level of structure to the source code annotations;
- Graphical representation strategies are merely informative and do not support any of the three features described above;
- The pattern instance validators strategies are the only ones that verify if the code is following the specified pattern structure and its roles. However, the information that describes the pattern instance is not easily consumed in this format;
- Regarding liveness, text-based or graphical approaches belong to level one since their documentation is only informative. On the other hand, Lovatt et al. [33] and Cornils et al. [8], since their approaches provide feedback after compiling the source code (after an event), without direct request from the developer, they belong to level three of the liveness hierarchy, discussed in Chapter 2;
- The integrated environment pattern was also explored but, due to the lack of information on the analyzed approaches, we could not conclude anything.

The closest approach, to fulfill the three features mentioned above, that we found was Cornils' approach but, it only warns developers that the code is no longer respecting the specified design pattern, it does not support any type of automatic co-evolution. Moreover, after contacting the authors, we have found out that the tool is no longer available.

3.2 Extracting patterns from code

Recovering design patterns from the source code has been a topic of interest for the last two decades. This has led to the development of several approaches and tools to detect and visualize design patterns. Some require the source code to be compilable, while others also need it to be executable. Known as *design pattern detection tools*, they can be categorized in several ways, depending on the level of automation, the level of liveness, the supported language, the way they display the discovered patterns, their type of analysis, among others.

Even though they can effectively recognize several patterns, they still have some flaws. Firstly, many do not scale when analyzing large systems, becoming significantly slower while processing. Secondly, those that require the source code to be compiled force the developer to solve the dependencies, required to use the tool, before being able to use them. Thirdly, since many are black boxes, it is hard for the developer to control the pattern detection process, being compelled to only interpret the final output. Additionally, several are not up-to-date regarding the current programming language' versions because, their creators stopped developing or maintaining them. Finally, the results found contain a large set of false positive pattern instances, requiring a manual check by the developer. This last one can be particularly tricky since not all developers are experts in design patterns and, may be induced in error by the tool.

In this section, we will explore some of the most prominent tools and approaches that have been developed so far. Since there is a clear distinction on the type of analysis approach, we will use that to catalogue them. In Section 3.2.1 , we will analyze those that only focus on the structural characteristics of the system, and in Section 3.2.2, we will examine those that contemplate some sort of dynamic analysis.

3.2.1 Static Analysis

PINOT (Pattern INference recOvery Tool) [43] is a fully automated tool, designed by Shin and Olsson, that performs structural and behaviour analysis of the source code. Being a modification of IBMs Java compiler, Jikes, embedded with pattern analysis code, it takes advantage of symbol tables and semantic checks to detect patterns in systems written in Java. Their approach starts by reclassifying the GoF design patterns into five new categories, based on the patterns intent: 1) provided by the programming language, 2) structure driven, 3) behaviour driven, 4) domain specific and 5) generic concepts. Figure 3.9 shows which patterns belong to each category. They claim to recognize all the GoF structure or behaviour driven design patterns.

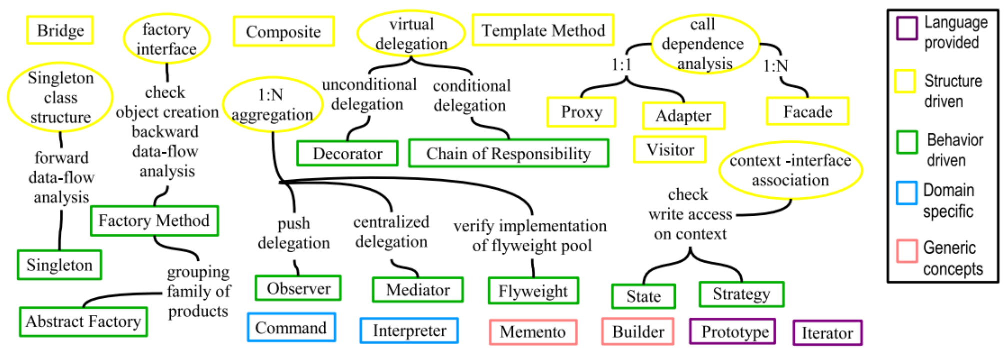


Figure 3.9: Reclassification of the 23 GoF Patterns as presented by Shi et al. [43]

The tool extracts inter-class relationships, such as abstract delegations, aggregations, associations, among others, from the source code generated AST, allowing them to identify the structural aspect of a pattern. When dealing with behaviour driven patterns, this last step helps narrowing down the search space of specific methods required for data flow analysis. Through this analysis, methods are represented as control flow graphs and used to determine the pattern instance behaviour.

Unfortunately, it does not provide any sort of visualization technique for displaying the detected patterns. The output is given by the command line as plain text.

The **SPOOL (Spreading desirable Properties into the design of Object-Oriented, Large-scale software systems)** environment was conceived in a collaborative work between the University of Montreal and Bell Canada "to help structure parts of class diagrams to resemble pattern diagrams" [29]. It supports several functionalities for design pattern engineering, being the most important for this thesis, its design pattern recovery approach for systems written in C++.

This approach envisions three operation modes being them manual, semi-automated and fully automated pattern recovery. Manual recovery operation allows developers to express their patterns instances or pattern variants by grouping design elements such as classes, attributes, methods, among others. The automatic recovery mode extracts pattern instances through queries to the source code models. Initially, they generated a repository to store design pattern's template descriptions which has, for each pattern instance variant, a query that is used to search the source code models for that pattern' structure. Therefore, each pattern instance variant has to be firstly stored in the repository before being able to be detected by the tool. Unfortunately, the semi-automated mode, whose goal is to mix both modes, hasn't yet been implemented. Nevertheless, the authors have envision it as a multi-phase recovery process, where during the first phase the general core of pattern descriptions is automatically detected and, during the subsequent phases these instances are matched against more specific implementation details, which may be provided manually by the developer.

The environment also provides interactive visual representations of the source code models as can be seen in Figure 3.10.

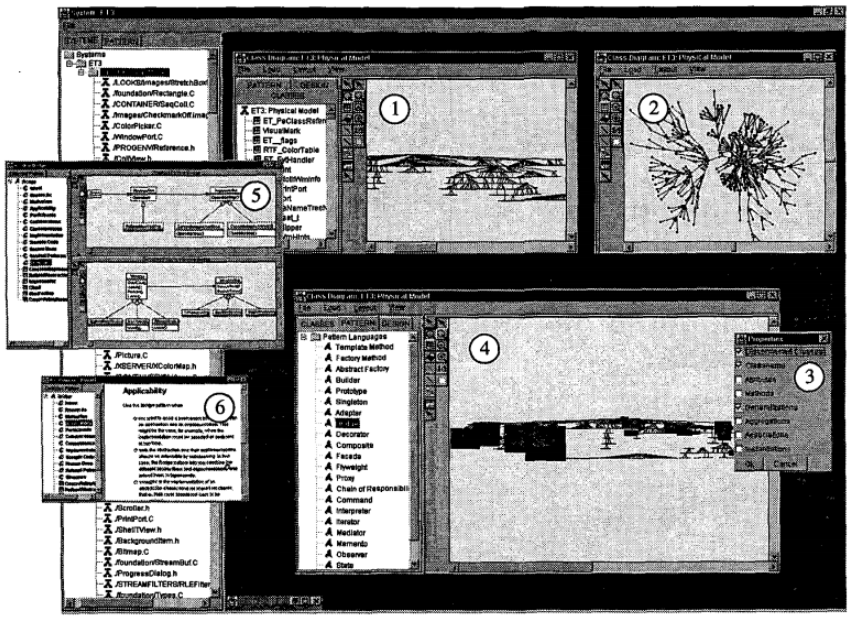
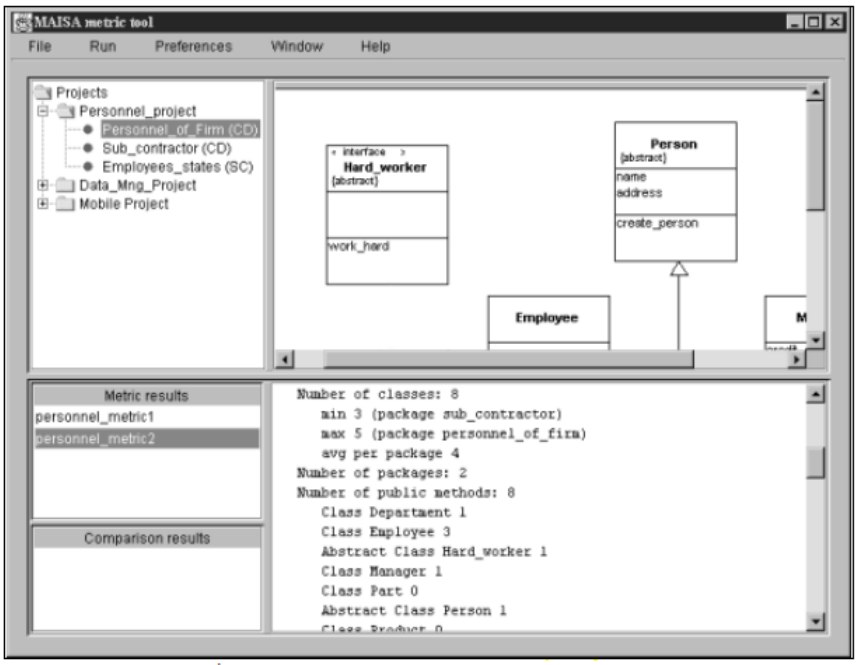


Figure 3.10: SPOOL's user interface with detected design patterns (window 5) and pattern instance documentation (window 6) as presented by Keller et al. [29]

MAISA [34] is a dedicated metric tool intended to analyze the quality of system' software architecture. Apart from providing metric results, it is able to identify and display the pattern instances used in the system (Figure 3.11). The pattern detection is achieved by solving a constraint satisfaction problem, where sets of variables representing roles, methods, constrains and other structural pattern details, are used. These variables are stored in a library of patterns. Pattern variants can be detected by relaxing or even removing some constraints.

The system source code is converted into UML descriptions, in Prolog and FAMIX files, which are the actual input of the tool. Since the MAISA tool doesn't perform this last step, the creators of the tool have extended the MetaEdit CASE tool [45] in order to achieve that type of input format. A complete overview of the tool is shown by Figure 3.12.

Ferenc et al. [21] have developed an exporter plug-in for Columbus [20], which transforms the Abstract Semantic Graph (ASG) into Prolog facts, the input format of MAISA tool. Since the Columbus' ASG is generated from C/C++ systems, this allows detecting C/C++ design patterns with the MAISA tool.



```
Number of classes: 8
   min 3 (package sub_contractor)
   max 5 (package personnel_of_firm)
   avg per package 4
Number of packages: 2
Number of public methods: 8
   Class Department 1
   Class Employee 3
   Abstract Class Hard_worker 1
   Class Manager 1
   Class Part 0
   Abstract Class Person 1
   Class Product 0
```

Figure 3.11: Detected design patterns in the Maisa tool as presented by Nenonen et al. [35]

FUJABA (From UML to Java And Back Again) [36] is a tool suite, developed by the University of Paderborn, that supports semi-automatic design pattern detection. It uses an incremental algorithm that combines Abstract Syntax Graphs (ASG) parsing techniques with fuzzy logic. The use of fuzzy set rules allows handling implementation variations of the same pattern, by providing a degree of uncertainty associated to a certain fuzzy value.

The tool's process overview can be seen in Figure 3.13. FUJABA starts by generating an ASG from the system source code and loading the pattern description catalogue, which can be modified

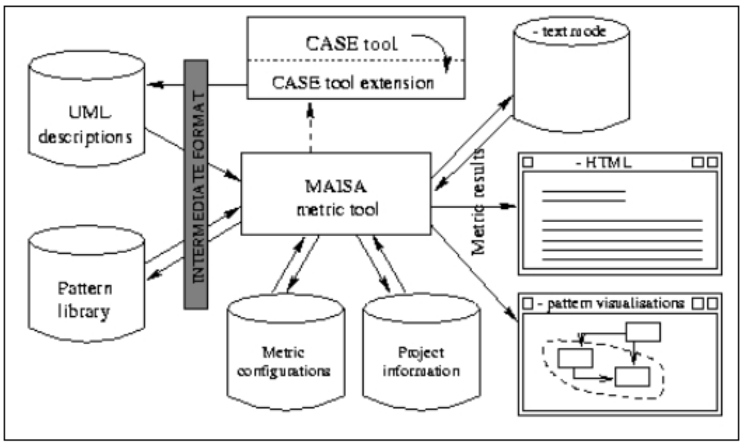


Figure 3.12: Maisa tool's architecture as presented by Nenonen et al. [35]

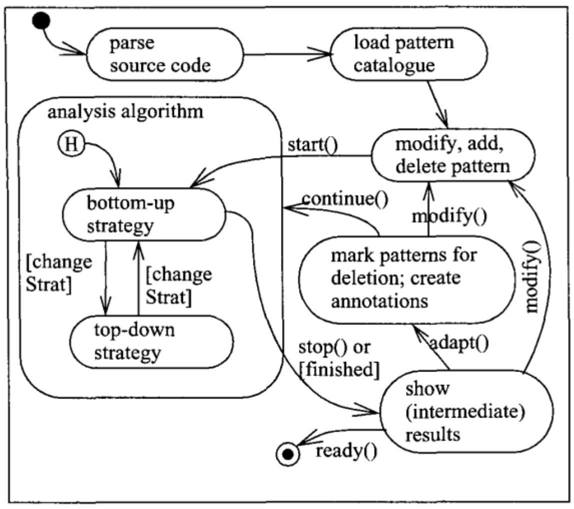


Figure 3.13: FUJABA tool's process statechart as presented by Niere et al. [36]

by the developer. Then, the developer has to give order to start the recognition algorithm. This algorithm applies a bottom-up-top-down searching approach to speed up the detection. How? Every time a pattern instance is identified (or partially identified, sub-pattern) through the bottom-up search, it annotates that knowledge on the graph and, switches to the top-down search, to finish the rest of the search and confirm the existence of such a pattern. This methodology also helps reducing the number of false positive detection. An instance is identified by applying transformation rules to the generated graph. Once the algorithm finishes, it displays the detected pattern instances in UML format (Figure 3.14). Since it is an iterative tool, it takes advantage of the newly created annotations on the graph to reduce the top-down graph search, on the following iterations. At the end of each iteration, if the developer isn't satisfied with the results, he can request the tool for another iteration. This can be particularly useful when the algorithm is not detecting the patterns or is providing too many false positives, allowing the developer to vary the fuzzy set values accordingly, and redo the search with the new fuzzy rules.

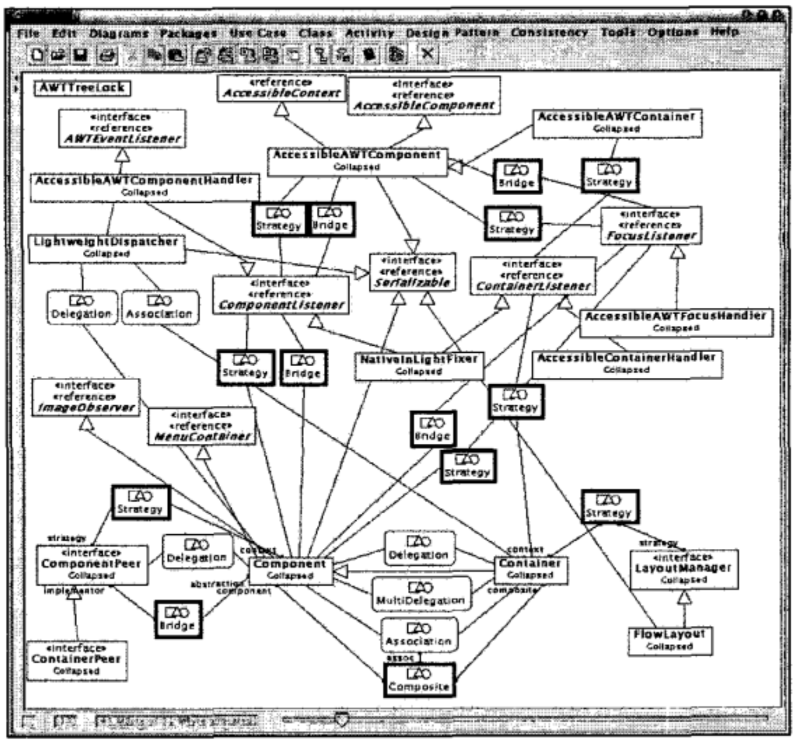
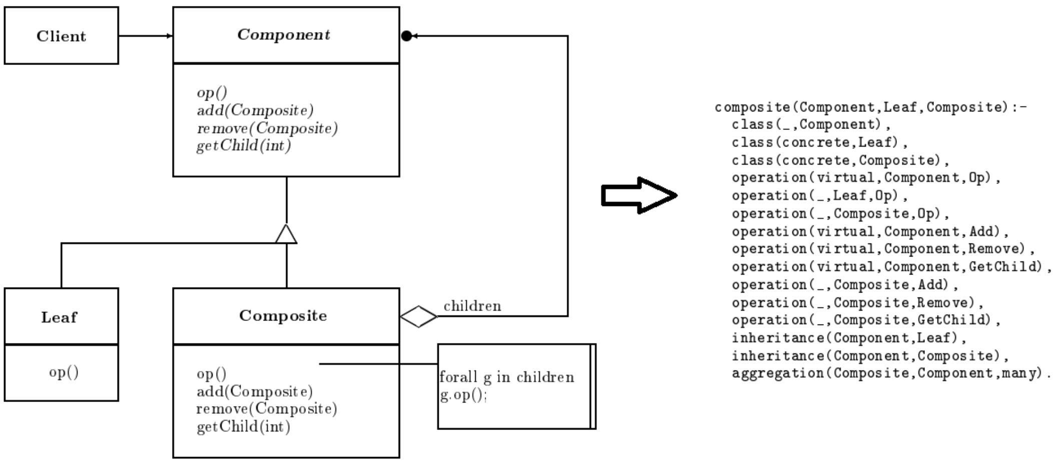

File Edit Diagrams Packages Use Case Class Activity Design Pattern Consistency Tools Options Help
AWTTreeLock
«reference» AccessibleContext
«interface» «reference» AccessibleComponent
AccessibleAWTContainer Collapsed
«interface» «reference» AWTEventListener
AccessibleAWTComponent Collapsed
AccessibleAWTComponentHandler Collapsed
LightweightDispatcher Collapsed
«interface» «reference» Serializable
«interface» «reference» ContainerListener
«interface» «reference» FocusListener
«interface» «reference» ComponentListener
AccessibleAWTFocusHandler Collapsed
NativeInLightFixer Collapsed
AccessibleContainerHandler Collapsed
«interface» «reference» ImageObserver
«interface» «reference» MenuContainer
«interface» ComponentPeer Collapsed
Component Collapsed
Container Collapsed
«interface» LayoutManager Collapsed
FlowLayout Collapsed
«interface» ContainerPeer Collapsed
Bridge
Strategy
Delegation
Association
MultiDelegation
Composite

Figure 3.14: Example of FUJABA results view as presented by Niere et al. [36]

PAT [38] is an automatic design pattern recovery tool for C++ software systems. It requires two inputs: 1) a library with the pattern UML diagrams and 2) the system source code. Through a structural analysis on the source code, the tool converts it into PROLOG facts. Simultaneously, the pattern diagrams are transformed into PROLOG rules (Figure 3.15). The pattern instances are, therefore, identified through queries on the PROLOG engine. Once the tool finishes, it provides a textual output with the detected patterns.



```
composite(Component,Leaf,Composite):-
  class(_,Component),
  class(concrete,Leaf),
  class(concrete,Composite),
  operation(virtual,Component,Op),
  operation(_,Leaf,Op),
  operation(_,Composite,Op),
  operation(virtual,Component,Add),
  operation(virtual,Component,Remove),
  operation(virtual,Component,GetChild),
  operation(_,Composite,Add),
  operation(_,Composite,Remove),
  operation(_,Composite,GetChild),
  inheritance(Component,Leaf),
  inheritance(Component,Composite),
  aggregation(Composite,Component,many).
```

Figure 3.15: Diagram of the design pattern "Composite" and its generated PROLOG rules as presented by Prechelt et al. [38]

Unfortunately, since only the class header files are examined, behavioral data is not available, making it only useful to detect structural design patterns. Moreover, false positives have to be manually removed by the developer.

MARPLE (Metrics and Architecture Reconstruction PLug-in for Eclipse) [2] was designed as a software architecture reconstruction tool that also provides design pattern detection functionalities. Unlike other tools, MARPLE requires knowing which pattern should it be searching for.

The tool starts by extracting elemental design patterns or micro patterns and collecting metric details, from the source code AST. This data is stored in XML files, which are transferred along each tool phase. During the second phase, instead of dealing directly with the results from the AST, it generates an Attributed Relational Graph (ARG) from it, where vertexes correspond to the classes, interfaces and other types and, the edges are the basic elements that connect those types to each other. Through a graph matching algorithm, potential design pattern candidates are recovered. These candidates are submitted to a machine learning approach, composed by neural networks and WEKA data mining algorithms, in order to classify their degree of similarity with the design pattern to be detected. This phase also helps eliminating false positives. Finally, on a view provided by the tool, the design pattern diagrams are presented to the user.

They claim to detect any kind of old and new partners. However, it requires a lot of test data to train the machine learning algorithm.

SPQR (System for Pattern Query and Recognition) [44] is a fully automated tool for detecting elemental design patterns in C++ systems. They use elemental design patterns (EDP) because being smaller makes them easier to identify. These EDP are used to define the GoF design patterns. Additionally, it makes the tool language independent since the design patterns are represented by these abstractions (EDP).

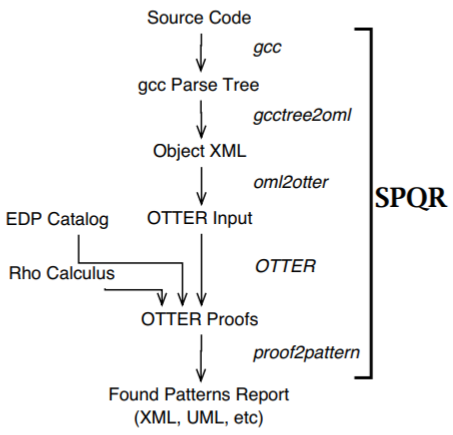


Figure 3.16: SPQR Outline as presented by Smith et al. [44]

The system's abstract syntax tree is generated with the help of the gcc compiler, which is subsequently parsed in order to build a XML description of the object structure features. This output is used to create the system's feature-rule input file for the logical inference engine, based on Rho-calculus, OTTER. The engine uses reliance operators to combine the feature-rules, forming the design patterns, and the Rho-calculus to encode the relationships between classes. Finally, OTTER proofs are analyzed in order to generate a XML file with the identified pattern descriptions. Figures 3.16 and 3.17 represent the tool's architecture and an example of output, respectively.

```
<pattern name="Decorator">
    <role name="Component"> "File" </role>
    <role name="Decorator"> "FilePile" </role>
    <role name="ConcreteComponent">
        "FileFAT" </role>
    <role name="ConcreteDecorator">
        "FilePileFixed" </role>
    <role name="operation"> "op1" </role>
</pattern>
```

Figure 3.17: Example of the SPQR tool's output as presented by Smith et al. [44]

CrocoPat [5] is an automatic tool, developed by the Technical University of Cottbus, to detect design patterns in Java systems. It uses an analysis tool named Sotograph to represent the system source code in terms of relations, which are stored in a file. In addition, design patterns definitions, which are represented using relational algebra expressions, are created using pattern specification language and, used as the tool's second input. Then, it transforms the relations, in the relation file, into binary decision diagrams (BDD), applies all the expressions on the pattern definitions file and, proceeds to a sub-graph search in order to obtain the pattern instances. Note that, the tool uses BDD because they help improving performance, which is the authors main motivation.

3.2.2 Dynamic Analysis

PTIDEJ (Pattern Trace Identification, Detection and Enhancement in Java) [24] is a complete reverse engineering tool for Java, C++ and AOL software systems. Developed by the University of Montreal, they formulate the detection of design patterns as a constraint satisfaction problem, more specifically, explanation-based constraint. Instead of dealing directly with design patterns, they extract micro patterns (patters at a lower level of abstraction), which are then used to define the design patterns.

Initially, the source code is parsed into PADL (Pattern and Abstract level Description Language) meta-model to describe the structure of the system parts in pattern definitions. In parallel, the source code is also compiled and injected into Caffeine, a dynamic analyser for Java, which is used to identify class inter-relationships while the system is running. By combining this output with the system model, they retrieve an enhanced program model, which is then used to identify the design patterns. For this last step, they use their explanation constraint solver that tries to find pattern instances by matching design pattern definitions and, if they do not match, constraints get relaxed in order to find variants of those patterns. This process is shown by Figure 3.18.

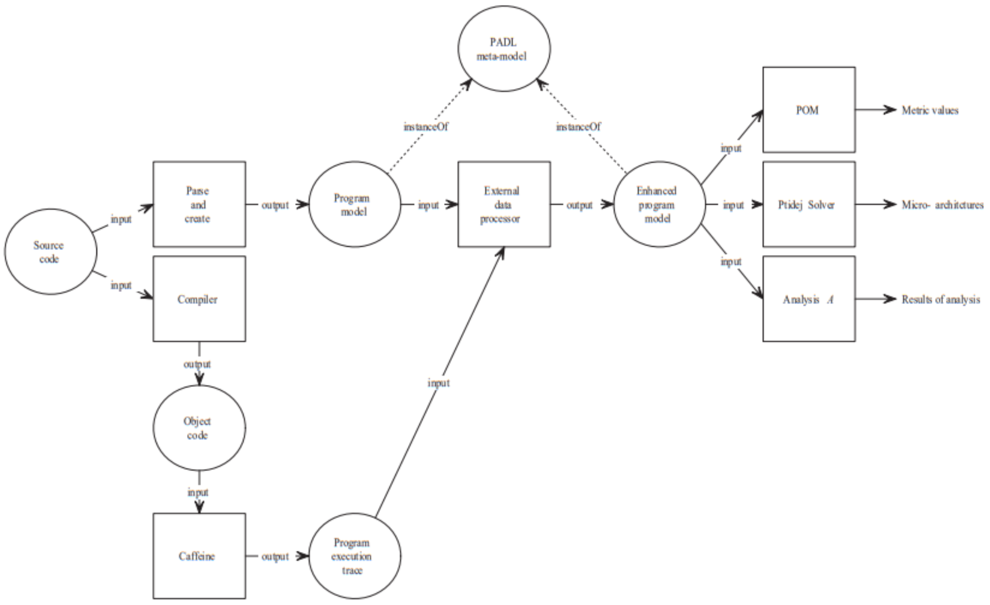
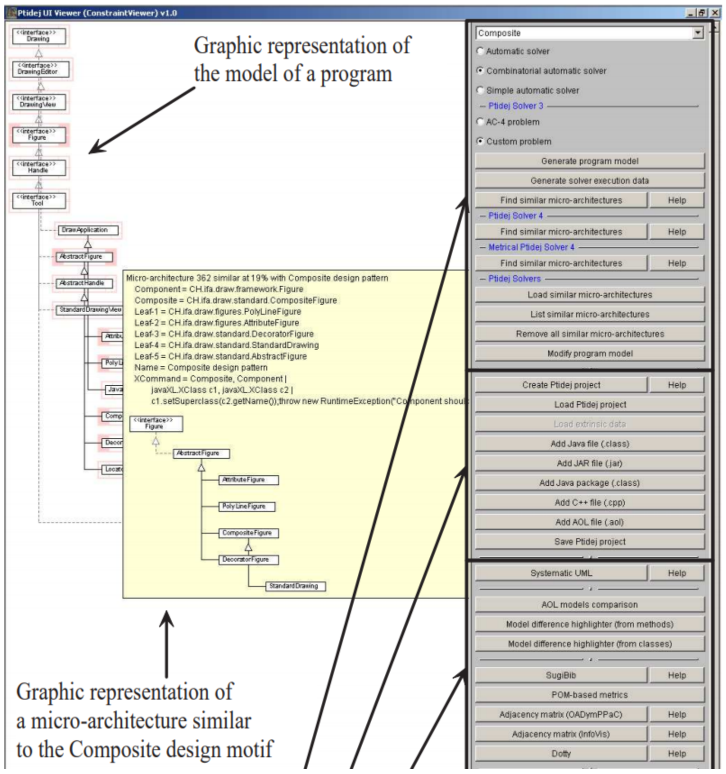


Figure 3.18: Use of the Ptidej tool suite as presented by Gueheneuc et al. [24]

Regarding visualization, the tool provides a user interface that displays the system model and the identified design patterns (Figure 3.19). The tool suite has a library containing design pattern models for this exact functionality.



Composite

( ) Automatic solver [✓ Combinatorial automatic solver] ( ) Simple automatic solver

( ) AC-4 problem [✓ Custom problem]

```
Micro-architecture 362 similar at 19% with Composite design pattern
   Component = CH.ifa.draw.framework.Figure
   Composite = CH.ifa.draw.standard.CompositeFigure
   Leaf-1 = CH.ifa.draw.figures.PolyLineFigure
   Leaf-2 = CH.ifa.draw.figures.AttributeFigure
   Leaf-3 = CH.ifa.draw.standard.DecoratorFigure
   Leaf-4 = CH.ifa.draw.standard.StandardDrawing
   Leaf-5 = CH.ifa.draw.standard.AbstractFigure
   Name = Composite design pattern
   XCommand = Composite, Component |
         javaXL.XClass c1, javaXL.XClass c2 |
         c1.setSuperclass(c2.getName());throw new RuntimeException("Component shoul
```

Figure 3.19: Part of Ptidej interface as presented by Gueheneuc et al. [24]

ePAD (eclipse plug-in for design Pattern Analysis and Detection) [14] detects design pattern instances from Java source code by performing static and dynamic analysis, being the last one used to validate the detected pattern instances (Figure 3.20). This is achieved by exploring their run-time behaviour.

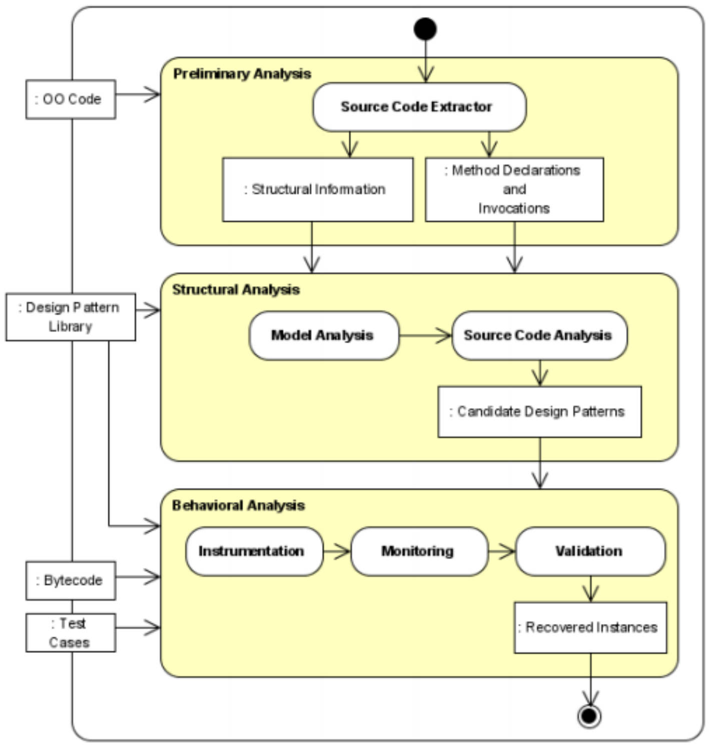
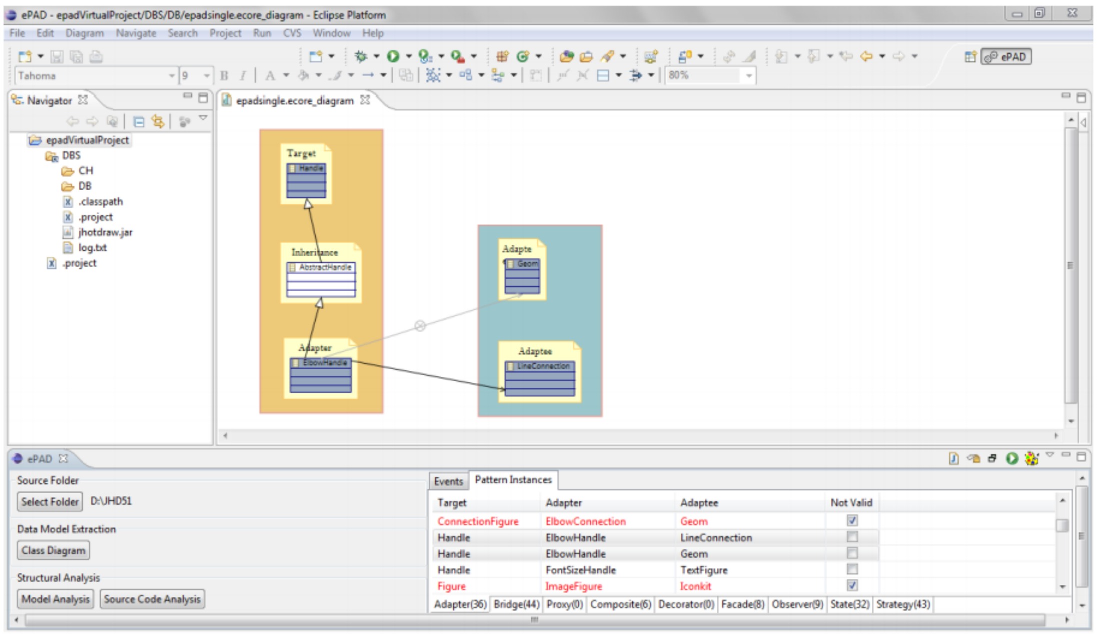


Figure 3.20: The recovery process of ePAD as presented by De Lucia et al. [14]

The tool extracts the source code structural information in order to build the class diagram's UML model. Through comparison between design pattern UML models, stored in a previously created library, and the generate source code UML model, they identify the pattern instances. Since this leads to a coarse-grained solution, the tool subsequently filters the false positives candidates by verifying their classes declarations and invocation methods. These candidates are analyzed during program execution to make sure that they are behaving according to that specific design pattern behaviour. The inputs for this phase are the set of candidates, the design patterns behaviour descriptions and, a list of test cases for monitoring and guiding the program execution.

Recovered design patterns are displayed as UML models, in an appropriate view (Figure 3.21). Unfortunately, the tool does not provide a way to execute all the steps, in their respective order, automatically, forcing the user to manually select each and every step.

The same authors have added **EVOsuite** to **ePAD** [16] in order to refine the initial set of test cases according to the detected candidate' instances, avoiding having to run unnecessary tests.



Target	Adapter	Adaptee	Not Valid
ConnectionFigure	ElbowConnection	Geom	✓
Handle	ElbowHandle	LineConnection	☐
Handle	ElbowHandle	Geom	☐
Handle	FontSizeHandle	TextFigure	☐
Figure	ImageFigure	Iconkit	✓

Adapter(36) Bridge(44) Proxy(0) Composite(6) Decorator(0) Facade(8) Observer(9) State(32) Strategy(43)

Figure 3.21: Visualization of a detected design pattern as presented by De Lucia et al. [14]

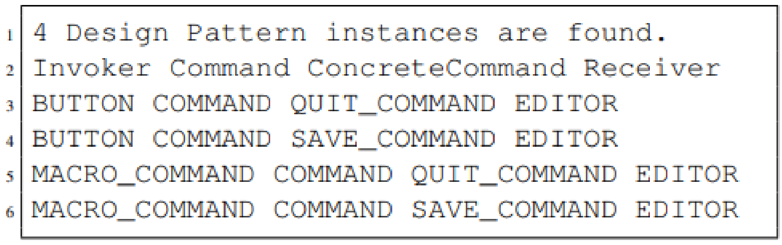
Li et al. [32] presented an approach to retrieve pattern instance behavioral data by analyzing its UML sequence diagram. The tool starts by parsing a C++ system source code into an Abstract Syntax Graph (ASG) with the help of an open compiler tool, OpenC++. Their static analysis algorithm retrieves pattern instances candidates (classes and sub classes), which are mapped to their suspected pattern through a name binding process. Binding accuracy is increased by taking into account the method body analysis as well as a naming convention.

When the system is running, they collect run-time data by tracing pattern instances classes behaviour in a XML file. This file is used to recreate a UML sequence diagram, which is compared to the sequence diagram of the mapped suspected pattern. Lothar Wendehals [52], also compares sequence diagrams to remove false positives.

DPVK (Design Pattern Verification toolKit) [51] is a semi-automatic tool for detecting pattern instances in Eiffel systems. They use a repository with two definitions for each design pattern, one based on their static structure and the other on their dynamic behaviour. Thus, the system's design patterns are found by comparing its static and dynamic fact files with those definitions.

Their approach has four stages: 1) Static fact extraction, 2) Candidate instance discovery, 3) False positive elimination and 4) Manual evaluation. Ignoring the last stage makes the tool fully automatic. The tool starts by generating the classes file descriptions, with the help of the Eiffel compiler, which are used to retrieve static information (relationships between classes). Then, with that data, they are able to extract the pattern instance candidates.

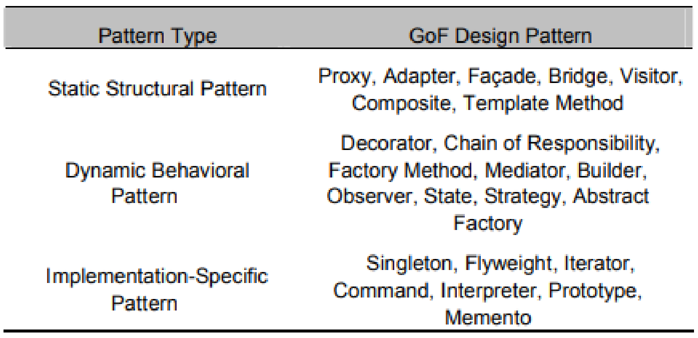
The third stage requires running the actual program. During this stage, calls to the classes that represent the pattern candidates are observed and their dynamic behaviour is collected to form the dynamic facts. These facts are used to compare with the respective pattern behaviour description in order to remove false positives. An example of a final output is shown in Figure 3.22. The final step is a manual validation by developers.



```
4 Design Pattern instances are found.
Invoker Command ConcreteCommand Receiver
BUTTON COMMAND QUIT_COMMAND EDITOR
BUTTON COMMAND SAVE_COMMAND EDITOR
MACRO_COMMAND COMMAND QUIT_COMMAND EDITOR
MACRO_COMMAND COMMAND SAVE_COMMAND EDITOR
```

Figure 3.22: Example of detected design patterns by DPVK tool as presented by Wang et al. [51]

Lee et al. [31] developed a design pattern detection technique based on their taxonomy: 1) Static Structural Pattern such as Bridge or Composite, 2) Dynamic Behavioral Pattern such as Decorator or Observer and 3) Implementation-Specific Pattern such as Singleton or Iterator. The taxonomy used for the GoF patterns is represented in Figure 3.23.



Pattern Type	GoF Design Pattern
Static Structural Pattern	Proxy, Adapter, Façade, Bridge, Visitor, Composite, Template Method
Dynamic Behavioral Pattern	Decorator, Chain of Responsibility, Factory Method, Mediator, Builder, Observer, State, Strategy, Abstract Factory
Implementation-Specific Pattern	Singleton, Flyweight, Iterator, Command, Interpreter, Prototype, Memento

Figure 3.23: Pattern Taxonomy for Reverse Engineering as presented by Lee et al. [31]

Initially, they start by representing the system's source code as an AST or ASG. Then, the high-level parts of the code are analyzed with a static algorithm, for each pattern, to identify pattern instance candidates. These candidates are retrieved by successfully matching the pattern definitions, on their pattern catalogue, with the high-level code parts. If the pattern candidates belong to the Dynamic taxonomy class, their method call tracing have to be monitored, while the program is running. During this step, candidate's methods are checked against their suspected pattern behaviour rules, being discarded those who do not. Finally, they use information of specific keywords and coding styles, of each pattern, to identify implementation-specific patterns.

3.2.3 Discussion

The design pattern extraction tools were categorized according to the level of automation, the target's system language, how they display the detected patterns, if they are restricted to an Integrated Development Environment (IDE) and their liveness level. Just to clarify, by automation level we refer to how independent, from developer's interaction, they are, which can range from manual to automatic. Table 3.3 provides an overview of the 33 analyzed tools.

Table 3.3: Overview of the approaches/tools for extracting design patterns from the source code

Technique	Tool	Authors	Level of Automation	Target Language	Visualization	IDE Restricted
Static	SPOOL	Keller et al. [29]	Automatic	C++	UML	No
	SPQR	Smith et al. [44]	Automatic	C++	XML	No
	Columbus	Ferenc et al. [20]	Automatic	C++	N/a[2]	No
	PAT	Prechelt et al. [38]	Automatic	C++	Text	No
	HEDGEHOG	Blewitt et al. [6]	Automatic	Java	Text	No
	DP-CORE	Diamantopoulos et al. [17]	Automatic	Java	Text	No
	PINOT	Shi et al. [43]	Automatic	Java	Text	No
	N/a	Rasool et al. [39]	Automatic	Java, C#	Text, UML	No
	CrocoPat	Beyer et al. [5]	Automatic	Java	N/a	No
	N/a	Chihada et al. [7]	Automatic	N/a	N/a	No
	DPF	Bernardi et al. [4]	Automatic	N/a	N/a	Eclipse
	FUJABA	Niere et al. [36]	Semi-automatic	Java	UML	N/a
	DeMIMA	Guéhéneuc et al. [25]	Semi-automatic	Java, C++	Text, UML	No
	N/a	Tsantalis et al. [50]	Semi-automatic	Java	Text	No
	WoP	Dietrich et al. [18]	N/a	Java	XML	Eclipse
	N/a	De Lucia et al. [12]	N/a	Java	HTML	No
	D^3	Stencel et al. [46]	N/a	Java	N/a	No
	N/a	Rasool et al. [40]	N/a	Any EA sup. lang.	N/a	Visual Studio.NET
	N/a	Ferenc et al. [21]	N/a	C++	UML, Text	No
	MAISA	Nenonen et al. [34]	N/a	Prolog	UML, Text	No
	MARPLE	Fontana et al. [2]	N/a	Independent	UML	Eclipse
	N/a	Mayvan et al. [3]	N/a	Independent	UML	Eclipse
Dynamic	ePAD	De Lucia et al. [14]	Manual	Java	UML	Eclipse
	ePADevo	De Lucia et al. [16]	Manual	Java	UML	Eclipse
	N/a	De Lucia et al. [13]	Automatic	Java	UML	Eclipse
	N/a	De Lucia et al. [15]	Automatic	Java	N/a	N/a
	DPAD	Zhang et al. [53]	Automatic	Java	Text	Eclipse
	N/a	Heuzeroth et al. [28]	Automatic	Java	N/a	No
	N/a	Wendehals [52]	Semi-automatic	Independent	N/a	No
	DPVK	Wang et al. [51]	Semi-automatic	Eiffel	Text	Eclipse
	PTIDEJ	Guéhéneuc [24]	N/a	AOL, Java, C++	UML	No
	N/a	Li et al. [32]	N/a	C++	UML	No
	N/a	Lee et al. [31]	N/a	Java	N/a	N/a

Through the analysis of the previous table, it is possible to see that two thirds of the tools use a static analysis strategy. In other words, most of the tools do not need to run the target systems to detect design patterns. Nevertheless, there are some patterns that they can't differentiate, since they focus only on the static structure of the classes, ignoring their run-time behaviour. An example of this, is how most of these tools can not differentiate the State from the Strategy patterns, which are structurally identical. Still, there are a few number of tools that by searching for specific methods, required during data flow analysis, manage to represent these methods as control flow graphs and determine their behaviour, such as PINOT [43].

[2]Not available.

Regarding their automation level, we can not draw many conclusions due to the large amount of not-available (N/a) results. There are quite a few tools that require constant user interaction, throughout the detection process like ePAD [14, 16] or FUJABA [36], and several that are completely autonomous such as PAT, DPAD, Columbus [38, 53, 21]. Note that, by completely autonomous we mean maximum one click, which is the execution of the tool on a given source code.

The majority of these tools were designed to identify design patterns in C++ or Java. The reason why this occurs might be because the design patterns they are trying to detect are the ones specified by the Gang of Four (GoF), who wrote the design patterns book in a C++ context [23]. Moreover, several of these design patterns described in GoF are not useful or are defined in a significantly different way, when outside of this context.

In terms of visualization, detected design patterns are displayed in several formats, being them textual (plain or structured), Hypertext Markup Language (HTML), Extensible Markup Language (XML) or even in a graphical representation format like UML. One thing we have noticed is that, most of the Eclipse plugins use UML representation, which might be explained by its ability to display design patterns in that format. Moreover, the tools that provide a graphic user interface like SPOOL, MAISA, FUJABA [29, 34, 36], among others, also use UML representation for the detected patterns. As we have seen in Section 3.1.2, the ability to visualize design patterns in a graphical way can ease the comprehension of the software, which might be the reason why these approaches tend to use this type of representation. Textual representations were mostly associated to command-line like tools [17, 43].

Regarding IDEs, we have often encountered Eclipse. However, the amount of tools that are not restricted on any IDE is higher. Also, there are some tools that can be used in different ways. For example, PTIDEJ [24] can be imported as a library for a Java, C++ or AOL project or used as a plugin for the Eclipse.

Considering that feedback is only provided through a direct request by the user, these tools belong to the second liveness level. In fact, design patterns are only displayed after we execute the tools on a certain software system.

Finally, we would like to point out one interesting feature that is not directly contemplated in the above Table 3.3. Not all tools search the software systems for each and every design pattern they may find. As a matter of fact, there are some tools, such as PAT or DP-CORE [38, 17], which allow checking the source code for a specific design pattern. This could be interesting if, after the developer specified the design pattern he was implementing, the tool could constantly check if that design pattern was found. For this purpose, the pattern detection tool would be running in the background. A negative result by the tool could be used to inform the developer that the design pattern was not correctly implemented.

Chapter 4

Problem Statement

In this chapter we start by describing, with more detail, the problem that exists within the current solutions related to documentation with design patterns to promote consistency between code and its documentation. Then, we introduce the thesis hypothesis and its research questions. Finally, we will analyze our validation methodology.

4.1 Scope

Design patterns are commonly defined as reusable solutions to recurrent design problems. They help designing complex object-oriented software systems, by capturing design experience from their original designers and, by providing a common vocabulary that facilitates interaction with other designers.

Software systems are often designed by different software development teams, composed by people with different cultures and skills, where knowledge is not evenly distributed among the team members. Good communication is important to make sure everyone understands what the developed software does and how to manipulate it. New developers may join the team and, in order to maximize the team productivity, the effort these newcomers have to apply, to quickly absorb everything that has be done, should be minimum. This highlights the need to adequately document software.

A good way to document software is by describing the design pattern instances that compose it. According to Odenthal and Quibeldey-Cirkel [37], a good practice when documenting a pattern instance is to give a quick visual reference (overview) of the design context, enhance the reason why it was instantiated, describe the design context in more detail, outline its role, illustrate the possible interactions between it and the client, provide the benefits and consequences of using it and, finally, identify special features of its implementation by referencing the source code. This makes understanding each pattern instance and its reuse a trivial task.

However, this does not solve everything. Since software is constantly evolving, regular documentation updates are required to keep it consistent since existing instances might be affected, by this evolution, and should be updated accordingly. Not performing this task while changes in the software keep being introduced, may lead to inconsistencies between the software and its documentation. By inconsistencies we mean conflicting ideas in related artifacts. In fact, during the process of software development, the developers are constantly switching between the programming and the documentation phase. These phases are firmly related to each other, since the output of one phase serves as input to the other, creating a mutual feedback loop. Considering that the act of programming and documenting are firmly different, usually even involve different types of artifacts, the loop enforces a constant swap of context. As a result, there are several negative consequences that may occur, including the loss of important knowledge and the constant postponing of one of those two phases, usually the documenting one, in order to remain longer within the same context. Once again, these problems lead to inconsistencies among the artifacts and, make the learning and reuse process of the software, a hard and troublesome task.

Throughout the previous chapter, we have explored several solutions that are intended to facilitate documentation with design patterns. However, none of the solutions found encompass all the aspects mentioned in Section 3.1.4, which we consider important to tackle this problem. Some do not maintain close proximity between code and documentation, others do not update the documentation after changes in the code are introduced and, all present a very low liveness level.

4.2 Hypothesis

This thesis is based on the following hypothesis:

"By increasing the level of liveness of documentation based on pattern instances, we will streamline the process of switching between creating and using documentation, promoting updated and consistent documentation, and improving the overall understanding of the system's design."

Since this hypothesis has some terms that can be ambiguous, we will try to clarify them:

Liveness - The feedback provided to the developer while the software system is being designed or executed (Section 2.4).

Documentation - As Correia stated *"Any form of captured information about [a] system that may help its developers and users understand it."* [9]. We are particularly interested in documentation based on pattern instances.

Consistency - When two pieces of related information do not express contradictory ideas.

4.3 Research Questions

We aim to tackle this problem by reducing the feedback loop between programming and documenting. Given that, this thesis addresses the following research questions:

> **RQ1** *"Can we reduce the inconsistencies between code and documentation?"* We aim to evaluate how liveness can help reducing conflicting ideas in related artifacts, while developing and maintaining the software system.
>
> **RQ2** *"Can we improve the comprehension of the software's design?"* We aim to evaluate how liveness can help converging faster to the solution.
>
> **RQ3** *"Can documentation be kept updated more easily?"* We aim to evaluate how liveness can help to reduce the effort (and, therefore, time) required to keep documentation up to date.

4.4 Methodology

With this work we aim to improve the understanding of software, in terms of its design patterns, by reducing the feedback loop between the programming and documenting phases. For that reason, we start with a **state of the art review** in order to understand the problems that exist with the act of documenting the pattern instances found in software systems. During this review, we compare the different approaches and highlight their major problems, since these would help us decide which problems we would like to solve. From this analysis, an hypothesis is created, which helps designing an **approach** to tackle those problems. Since our **approach** is mostly focused on providing benefits to the user and not to the actual program itself, even though it does indirectly, we carry out a **controlled experiment with user** to validate it.

In this study, the participants are master students in Informatics and Computing engineering, who are familiar with design patterns. The exposure of these subjects to our approach is achieved by creating a **prototype**, in the form of a plugin, that identifies pattern instances in the source code and live-suggests them to the developer, automatic generation of accepted pattern instance suggestions, and live editing and visualization of those pattern instances.

For the **controlled user experiment**, the test subjects are divided in two groups: 1) the control group and 2) the experimental group with access to our tool. While both groups complete several tasks, for which they needed to understand and evolve a software system, we evaluate some metrics, such as task completion speed and number of accesses to external documentation. These metrics and the qualitative data, collected via a Google form during the experiment, help answering the research questions selected for this work and support the discussion of the validity of the hypothesis.

Chapter 5

Approach and Design of the Design-Pattern-Doc Plugin

As we have previously covered, the goal of this thesis is to streamline the creation and documentation of software, making it easier to switch between these two software development phases. Nevertheless, there are a few concerns that must be dealt with to achieve this goal, such as how to maintain consistency between code and documentation, how to present the pattern instances to the developer and how to support the identification of design patterns in the source code.

Our approach uses liveness to tackle these problems. We believe that by increasing the amount of feedback about the pattern instances represented in the source code, anytime a pattern participant reference is being inspected, while the developers are editing the source code, we will achieve a higher degree of liveness. This will not only allow close proximity between the artifacts, it will make some of the tasks, like editing/creating documentation or comprehending the design of the software systems, easier to perform. Ultimately, since the increase in the liveness level makes it easier to maintain and consume documentation, it should also reduce inconsistencies between the artifacts.

During the rest of the chapter, we will explore the principles used as guidelines for the design of the approach, the prototype's implementation/architecture and the features that cover that architecture.

5.1 Approach

The goal of this thesis is to address the challenge of **understanding** software in terms of its design patterns, by reducing the mutual feedback loop between programming and documenting, easing the transition between these phases. By doing this, we will also ease the **creation, consistency maintenance and use** of pattern-based software documentation. We believe that, the most likely outcome, will be an increase in the liveness level and a design based on some documentation good practices, such as information proximity, co-evolution, domain-structured information and integrated environment. In other words, by maintaining close proximity between the code and its documentation and, by constantly providing visual awareness of the pattern instance descriptions and the program state to the developer (like displaying missing pattern participants for that specific pattern instance).

For the design of our approach, we followed some of the good practices mentioned in Chapter 2. These good practices form the list of principles in which our solution is based on:

1. **Information Proximity** - Using links and transclusion to ease the access to the pattern instances documentation, directly from the source code.

2. **Co-Evolution** - Updating all the related artifacts, every time a change is introduced. For example, after renaming a class that plays a role in a pattern instance, the documentation for that pattern instance must be updated. This also helps avoiding obsolete documentation.

3. **Domain-Structured Information** - Structuring the pattern instances according to a data model, capable of representing them in a rich format, instead of using plain text to represent them.

4. **Integrated Environments** - Maintaining all the related artifacts (eg. source code and its documentation) under the same environment, reducing the need to constantly switch context, when alternating between the software development phases.

5. **Liveness** - Increasing the liveness level to provide almost instantaneous awareness of the program's state. Instantaneous access to pattern instances documentation when editing the source code, alerting the developer when the documentation is incomplete, easier awareness of the effects of editing the documentation are some of its benefits.

With this approach we expect to decrease the response time of developers when the code and documentation of its patterns diverges or is not complete, mitigating these inconsistencies and reducing the overall required time to develop and document the system.

A prototype for our approach was developed, in the form of a plugin for an Integrated Development Environment such as Visual Studio Code or IntelliJ IDEA, able to analyze source code, live-suggest and generate corresponding pattern-based documentation for that specific pattern instance, and display inconsistencies.

5.2 Prototype

Based on the principles presented in the previous section, we have created some more concrete desideratum for the design of the prototype. They are the following:

1. Identifying pattern instances in the source code. At least liveness level three should be achieved here. For example, running a detection tool, few seconds after the developer stops typing.

2. Suggest detected pattern instances to the developer. The developer should be able to decide whether to accept or reject those suggestions. This is important since we do not want to be too intrusive. The liveness level here should be level four. If there are any pattern instance suggestions, they should be instantaneously displayed to the user.

3. Create and display documentation in the IDE. This should reduce the context switching between software creation and documentation, since both artifacts will be created and viewed on the same environment. Here, the visualization of documentation should correspond to, at least, a liveness level three. Nevertheless, we aim for level four.

This desiderata was used as aid to the design of the prototype's architecture, which will be explored next.

5.2.1 Architecture

Our prototype's conceptual overall architecture is depicted by Figure 5.1. Achieving our solution's guidelines can be done by linking the five design components and the two data storage represented in the figure.

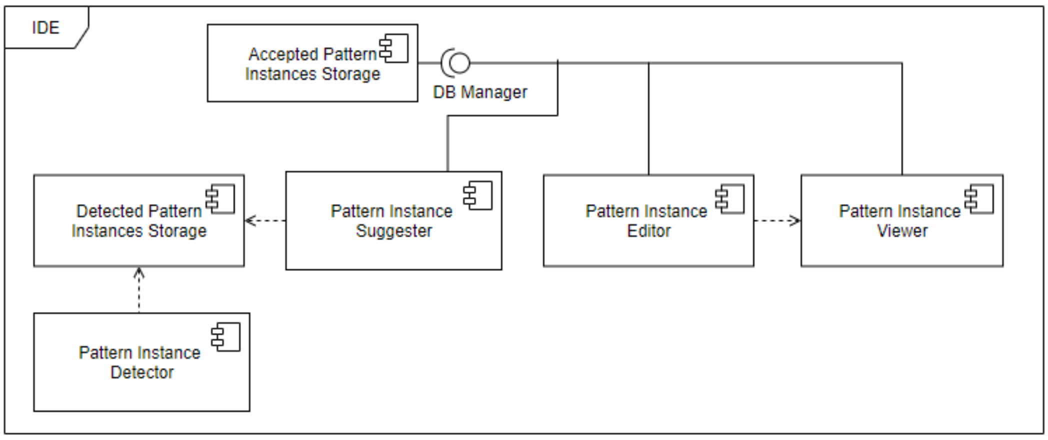

IDE
Accepted Pattern Instances Storage
DB Manager
Detected Pattern Instances Storage
Pattern Instance Suggester
Pattern Instance Editor
Pattern Instance Viewer
Pattern Instance Detector

Figure 5.1: Prototype's conceptual overall architecture

Let's discuss these components' main responsibilities. There are two types of data storage: 1) the *Accepted Pattern Instances Storage* and 2) the *Detected Pattern Instances Storage*. The first one is responsible for persisting the pattern instance's documentation that were either manually created or the outcome of accepting a pattern instance suggestion. The second one is responsible for managing a collection of all the pattern instances detected by the selected tool (view the first desideratum in Section 5.2). The latter is also used as input for the component described in the second desideratum — *the Pattern Instance Suggester*. Note that, it would be possible to merge the two data storage. Nevertheless, having them separated, does not only provide separation of responsibilities, it also allows one type of data to be persisted (to the disk), while the other one remains volatile. There is no need to persist the pattern instances detected by the detection component if it will be scanning the source code every few seconds. Since there will be multiple read/write requests to the *Accepted Pattern Instances Storage*, from several different components, we have defined an interface, that implements concurrency, to interact with it — the *DB Manager*. This component is the only entry to the persisted documentation.

Shifting the focus to the other four components, the *Pattern Instance Detector* component runs in the background, every few seconds or a couple of seconds after the developer stops modifying the source code. As we have explained before, its output is stored in volatile memory. The selected detection algorithm/tool was designed to be easily replaced, as long as the output matches the same format — Factory Design Pattern. For our prototype, we used one of the tools explored during the literature review. This will be discussed in the following section.

Related to the pattern instance detector component, the *Pattern Instance Suggester* component informs the developer every time a pattern instance is detected. The feedback provided by it is instantaneous and the least intrusive as possible. Moreover, it doesn't modify the persisted documentation without a direct command, given by the developer. For that reason, there is the possibility to ignore the suggestions. On the other hand, accepting the suggestion, automatically generates the pattern instance documentation for that detected design pattern.

Regarding the last desideratum, we may find the *Pattern Instance Viewer* and the *Pattern Instance Editor*. Recall that the goal of this thesis is to reduce the feedback loop between programming and documenting. This means that the documentation should be kept with close proximity to the source code, avoiding too many changes in the context when consuming it. Given this, we have designed a component responsible for providing the persisted documentation, in the IDE, where the source code is being edited. By inspecting the class references from the pattern participants of the persisted documentation, we allow the developer to view the pattern instances where they play at least one role. This is the role of the *Pattern Instance Viewer*. Additionally, the representation of these pattern instances was decided based on the literature review, presented in Chapter 3. We will focus this design detail in the following section.

Directly dependent of the *Pattern Instance Viewer*, the *Pattern Instance Editor* component does not only allow the editing of a persisted pattern instance or the manual creation of the documentation for one, it does it in a lively manner. The developer is able to (almost instantaneously) see the changes in the pattern instances when editing them. This is the reason why it is dependent on the *Pattern Instance Viewer* component. Pattern instances edited or created by it are persisted via the *DB Manager* component.

Finally, since we want all these components to co-exist on the same context, they should all belong to the same environment. Given that nowadays, source code is largely created in IDE's, and we want the documentation to be created and consumed in the same environment, our approach implements all these components in an IDE.

5.2.2 Design decisions

As we concluded from the Chapter 3, most of the design pattern detection approaches found, were designed to deal with Java systems. Additionally, a large part of these approaches were theoretical or their implementation wasn't public. This conclusion came from the fact that those papers do not reference the tool, nor were we able to find them by searching on the internet. We also filtered out the tools that were made for a specific IDE since reusing these wouldn't be an easy task.

Another important decision that had to be done, was regarding the type of source code analysis technique, used by the detection tool. We decided to go for a static analysis tool. The reason for this was that, since these tools do not require to run the code being developed, they grant less computational effort for providing live feedback. Certainly there are some drawbacks, as we have previously discussed in Section 3.2.3, but their overall performance is very satisfactory.

We tried a couple of the tools from the remaining set, which provided public access to their source code. Unfortunately, we only managed to successfully run one of them — the DP-CORE tool. This tool does not support the detection of many design patterns but, for the sake of this thesis, its performance was good enough to test our hypothesis. Moreover, having access to its source code, allows extending the number of supported design patterns. This tool will play the *Pattern Instance Detector* component role in our architecture.

Also, given that we chose Java as our target system language, we chose Intellij IDE for the plugin environment.

Regarding the pattern instance visualization, we have seen in Section 3.1 that there are two different approaches: 1) text based and 2) graphical representation. Since pattern instances are usually represented by UML diagrams, the second one would be the most appropriate approach. However, we have seen that most of the graphical approaches require external documentation to display the UML diagrams. Nevertheless, after exploring some of the existing tools for Intellij, we found Plantuml. This tool allows visual representation of UML diagrams directly in the IDE environment. Given that it covers the mentioned disadvantage of using graphical representations, we have decided to append this library as a dependency for the *Pattern Instance Viewer* component, in our approach.

Finally, by combining all these design decisions, we turn our prototype's conceptual architecture into a concrete architecture. This is represented by Figure 5.2.

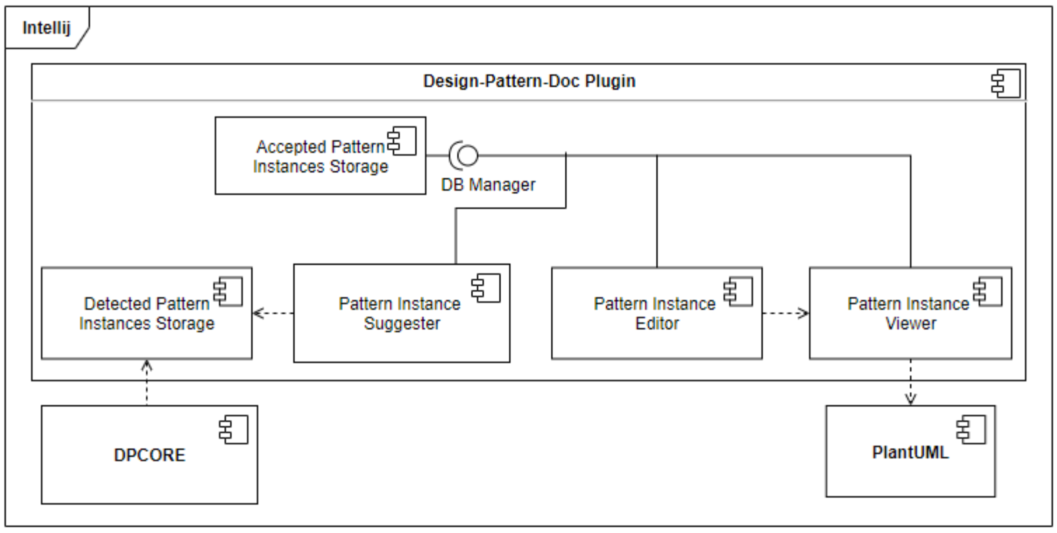

Intellij
Design-Pattern-Doc Plugin
Accepted Pattern Instances Storage
DB Manager
Detected Pattern Instances Storage
Pattern Instance Suggester
Pattern Instance Editor
Pattern Instance Viewer
DPCORE
PlantUML

Figure 5.2: Prototype's overall architecture

5.2.3 Domain Model

Our prototype resides on the domain model represented in Figure 5.3 to describe its sphere of knowledge. The main concept of our approach is the documentation of the pattern instances that compose the software systems. Apart from a field where the intention of the developer for instantiating a design pattern should be stored, each pattern instance is composed by one or more pattern participants. These pattern participants are the objects that play at least one role of the design pattern associated to that specific pattern instance.

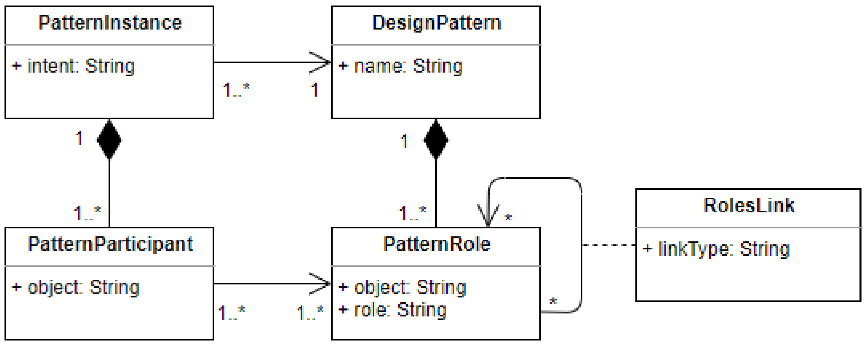


Figure 5.3: Class diagram of the Domain Model

5.3 Source Code Repository

The prototype was published to Intellij's marketplace with the MIT License. Currently, it is still pending JetBrains approval. Nevertheless, the source code can be accessed via the author's Github public repository [1].

5.4 Plugin Features

Throughout this section, we will introduce the main features of our tool. It is composed by ten major services who run in separate threads.

5.4.1 Pattern Instance Detector

The pattern instance detection service uses DP-CORE as a library to scan the project folder, for the design patterns supported by it. This service is executed periodically and, in each execution, exports a set with all the design patterns (name, roles, participants) found. The output may be an empty set. The contents of this set are used to suggest pattern instance documentation to the user.

[1]https://github.com/FilipeFLemos/Design-Pattern-Doc

5.4.2 Accepted Pattern Instances Storage

Intellij supports an application service for persisting the plugin state. In our case, we use it to store a map with all the pattern instances documented. Accepted suggestions or manual creation of pattern instances are persisted into this map.

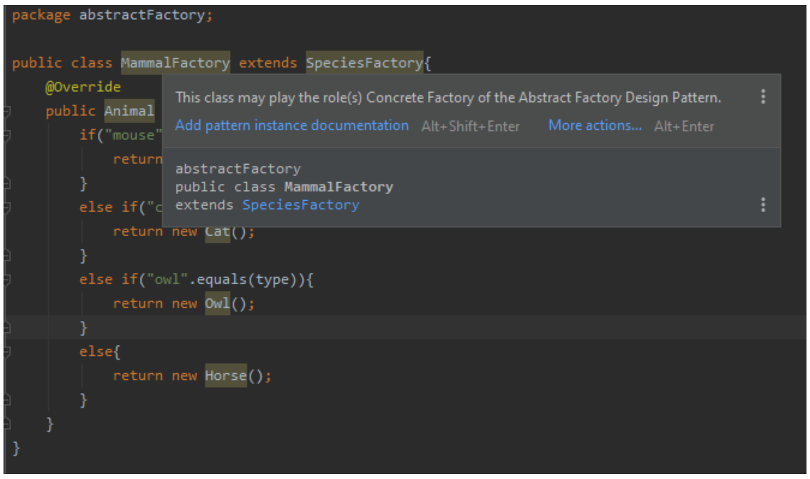
Another data structure that is stored here is a set with all the supported design patterns. Each set element contains a design pattern model composed by a name, a set of roles and a list representing how the roles relate with each other.

The class *PluginState* is responsible for saving and loading the plugin state data from a xml file.

5.4.3 Inspections

Our plugin contains three different types of code inspections. One is responsible for highlighting the objects that we believe to play a role in a pattern instance. The second one is responsible for updating the persisted pattern instances data after renaming an object. The third and last one is responsible for alerting the user when the source code is incomplete/inconsistent regarding its documentation (eg. missing some role).

Pattern Instance Suggester Information regarding pattern participants, collected by the design pattern detection service, is used to highlight those objects on the source code. If an object plays a role in a design pattern that hasn't yet been documented, a yellow light bulb (quick fix suggestion) will appear next to it, indicating a pattern instance suggestion. Figure 5.4 illustrates this highlighting mechanism. It provides two types of quick fixes: 1) accept the documentation suggestion or 2) ignore it. By accepting the suggestion, the pattern instance is automatically persisted to the local storage, making it accessible by the visualization features (which will be discussed later on). By ignoring it, the object stops being highlighted by the tool.



```
package abstractFactory;

public class MammalFactory extends SpeciesFactory{
    @Override
    public Animal
        if("mouse"
            return
        }
        else if("c
            return new Cat();
        }
        else if("owl".equals(type)){
            return new Owl();
        }
        else{
            return new Horse();
        }
    }
}
```

This class may play the role(s) Concrete Factory of the Abstract Factory Design Pattern.

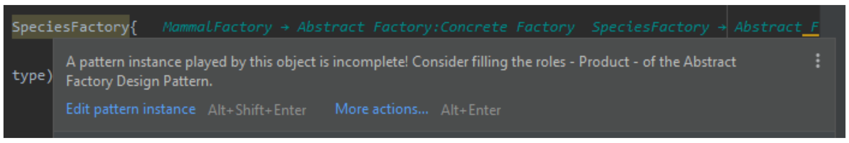
Add pattern instance documentation Alt+Shift+Enter
More actions... Alt+Enter

```
abstractFactory
public class MammalFactory
extends SpeciesFactory
```

Figure 5.4: Example of a pattern instance suggestion

Renaming Inspection The renaming code inspection is self explanatory. Instead of turning the documentation obsolete after renaming a certain object, it makes sure the documentation is kept updated. This code inspection supports the undo operation after renaming an object.

Incomplete Documentation Inspection The incomplete documentation inspection highlights all the pattern participants from a pattern instance, where at least one role is not played by any object. Figure 5.5 illustrates this situation. This helps the user spotting which objects should be created to complete the design pattern. If the missing pattern participants were already implemented, it warns the developer that the documentation hasn't been updated yet. This highlighting provides two types of quick fix: 1) Edit the documentation or 2) Delete the pattern instance. Choosing the first option opens the live pattern instance editor for that specific instance.



```
SpeciesFactory{   MammalFactory → Abstract Factory:Concrete Factory   SpeciesFactory → Abstract_F
type)
```

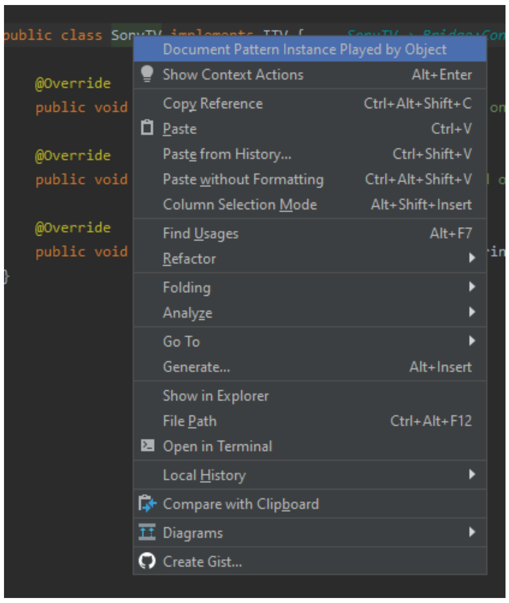
A pattern instance played by this object is incomplete! Consider filling the roles - Product - of the Abstract Factory Design Pattern.

Edit pattern instance Alt+Shift+Enter   More actions... Alt+Enter

Figure 5.5: Example of an incomplete documentation warning

5.4.4 Manual Pattern Instance Documentation

As we have explained before, pattern instances can be persisted by accepting the tool's pattern instance suggestion. However, manual documentation of a pattern instance is also possible. By right clicking on a class name, the developer will be presented with that option (view Figure 5.6 for more details). This will automatically place that class as one of the pattern participants of a default design pattern. At the same time, the pattern instance live editor will display the new pattern instance for further editing.



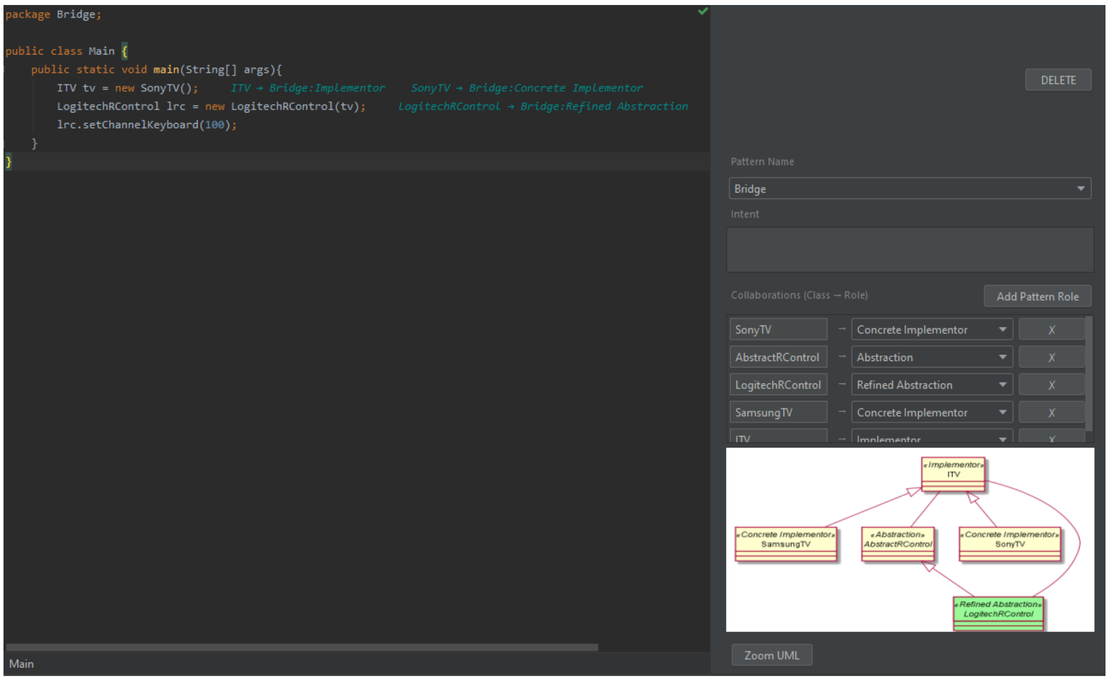
```
public class SonyTV implements ITV {

    @Override
    public void

    @Override
    public void

    @Override
    public void
}
```

Document Pattern Instance Played by Object
Show Context Actions Alt+Enter
Copy Reference Ctrl+Alt+Shift+C
Paste Ctrl+V
Paste from History... Ctrl+Shift+V
Paste without Formatting Ctrl+Alt+Shift+V
Column Selection Mode Alt+Shift+Insert
Find Usages Alt+F7
Refactor
Folding
Analyze
Go To
Generate... Alt+Insert
Show in Explorer
File Path Ctrl+Alt+F12
Open in Terminal
Local History
Compare with Clipboard
Diagrams
Create Gist...

Figure 5.6: Example of manual documentation

5.4.5 Pattern Instance Live Editor

Persisted pattern instances can be edited via the pattern instance live editor, located on the right side of the Intellij's file editor. The content on this window is displayed if the mouse's cursor is on top of a class, that plays a role in at least one persisted pattern instance. Changing the cursor's position among the different class names, contained in each file, updates the window's content, accordingly.



```
package Bridge;

public class Main {
    public static void main(String[] args){
        ITV tv = new SonyTV();     ITV → Bridge:Implementor    SonyTV → Bridge:Concrete Implementor
        LogitechRControl lrc = new LogitechRControl(tv);    LogitechRControl → Bridge:Refined Abstraction
        lrc.setChannelKeyboard(100);
    }
}
```

Pattern Name: Bridge

Intent:

Collaborations (Class → Role)

SonyTV	→	Concrete Implementor
AbstractRControl	→	Abstraction
LogitechRControl	→	Refined Abstraction
SamsungTV	→	Concrete Implementor
ITV	→	Implementor

DELETE
Add Pattern Role
X
«Implementor» ITV
«Concrete Implementor» SamsungTV
«Abstraction» AbstractRControl
«Concrete Implementor» SonyTV
«Refined Abstraction» LogitechRControl
Zoom UML
Main

Figure 5.7: Example of the live pattern instance editor

This editor uses the plantuml tool to display a live representation of the current pattern instance. Any change to a pattern participant (name, role), will update the UML preview after a small amount of seconds. Since large images have to be resized to fit the dialog, we provide a "Zoom UML" button, which can be accessed to display the real sized UML in a new window.

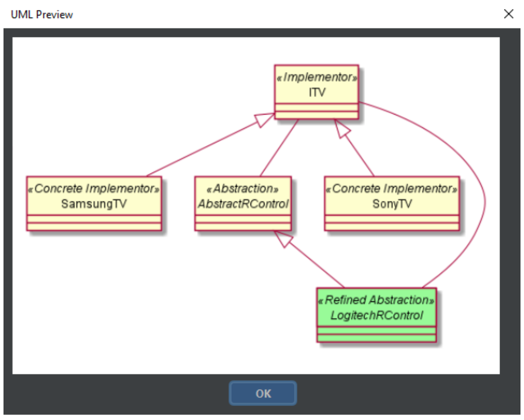

UML Preview
×
«Implementor» ITV
«Concrete Implementor» SamsungTV
«Abstraction» AbstractRControl
«Concrete Implementor» SonyTV
«Refined Abstraction» LogitechRControl
OK

Figure 5.8: Example of a real sized UML

5.4.6 Design Patterns Definition

The plugin offers a couple of supported design patterns, which are the ones supported by the detection tool. Nevertheless, it is also possible to define new design patterns. This feature can be accessed by right clicking on the file's editor. It is a two step dialog, where on the first screen the developer must specify the pattern name and roles and, on the second screen, the relationship between the specified roles.

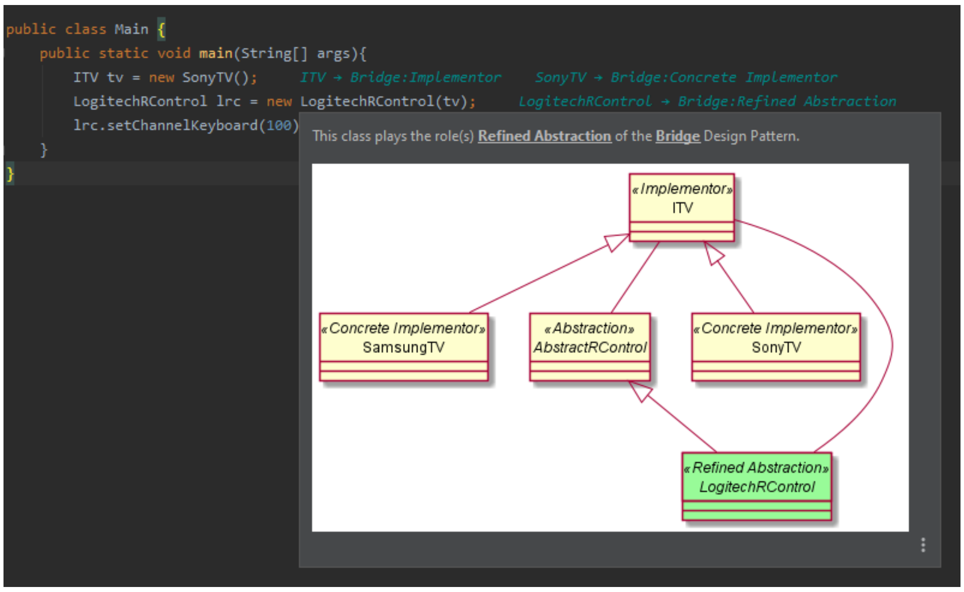
Note that, these new design patterns will not be used by the detection tool. However, all the other visualization techniques (pattern hints, documentation hover) and the incomplete documentation feature, will still work accordingly.

5.4.7 Pattern Instance Viewer

Apart from the live pattern instance editor, there are a two features, whose purpose is merely visual, that also provide details regarding persisted pattern instances. These details can be observed by hovering on top of the pattern participants or by reading the pattern hint text in front of them.

Pattern Instance documentation When hovering on top of the class name of pattern participants, that have played at least one role in a persisted pattern instance, a window appears below it. This window contains all the pattern instances played by that object in a UML representation (also created by plantuml tool). Just like in the pattern instance live editor, the class which was hovered, is highlighted in a different colour from the rest of the pattern participants.

In order to take full advantage of the UML class diagram (eg. inheritance links), Graphviz should be installed in the client's machine. To ease the readability of the diagrams, we decided to use UML stereotypes to represent the pattern roles, played by the objects.

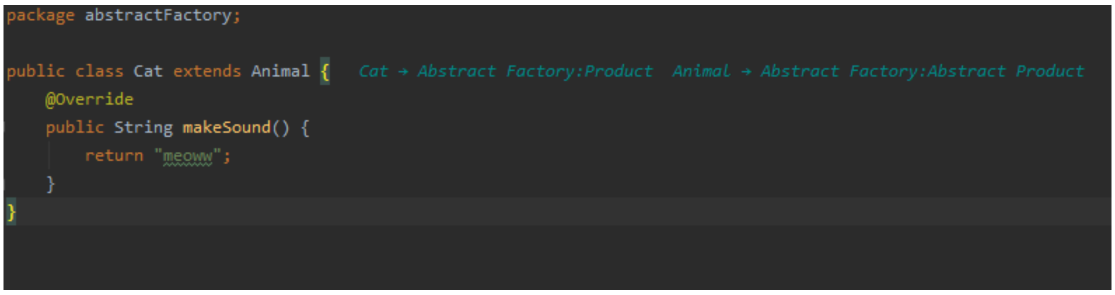


```
public class Main {
    public static void main(String[] args){
        ITV tv = new SonyTV();     ITV → Bridge:Implementor    SonyTV → Bridge:Concrete Implementor
        LogitechRControl lrc = new LogitechRControl(tv);     LogitechRControl → Bridge:Refined Abstraction
        lrc.setChannelKeyboard(100)
    }
}
```

This class plays the role(s) Refined Abstraction of the Bridge Design Pattern.

Figure 5.9: Example of documentation when hovering on top of a pattern participant

Pattern hints Pattern hints are text extensions, which are appended in front of the lines, where the class name of pattern participants are found. These extensions provide details on the roles played by that object in each pattern instance. The style and colour were selected to mimic the text extensions displayed by Intellij debugging mode.



```
package abstractFactory;

public class Cat extends Animal {   Cat → Abstract Factory:Product  Animal → Abstract Factory:Abstract Product
    @Override
    public String makeSound() {
        return "meoww";
    }
}
```

Figure 5.10: Example of pattern hints

Chapter 6

Empirical Study

Throughout this chapter, we will describe the controlled experiment used to validate our approach, introduced in the previous chapter, whose goal is to reduce the feedback loop between programming and documenting. This controlled experiment consisted in the execution of a few programming tasks by MSc informatics engineering students. It was designed with two different treatments — the experimental treatment, where participants use the Intellij IDE with the solution's prototype detailed in Chapter 5, and the control treatment, where participants do not have access to the developed prototype. The data collected during the experiment was subject to a statistical analysis, whose results were important to answer the research questions introduced in Chapter 4.

All the documents mentioned throughout this chapter are available in a GitHub repository[1].

6.1 Goals

The goal of this experiment is to validate our approach regarding the three research questions presented in Chapter 4. These research questions were answered with the help of some metrics, as denoted below:

RQ1 *"Can we reduce the inconsistencies between code and documentation?"* — This research question might be the hardest to answer since it is hard to measure it quantitatively. Nevertheless, we investigated if 1) by **reducing context switching**, between writing code and creating the documentation for it, and 2) **spending less time documenting** a software system, we reduce the probability of the developer to postpone the documenting phase and, consequently, avoid losing information between those two phases.

RQ2 *"Can we improve the comprehension of the software's design?"* — We aim to investigate whether liveness would help improving the comprehension of the software's design. Exploring if the participants **spend less time understanding and documenting** a software system, should help answering this research question.

[1] https://github.com/FilipeFLemos/Design-Pattern-DocUserStudy

RQ3 *"Can documentation be kept updated more easily?"* — We want to find out if we can reduce the effort (time) to keep documentation updated by increasing the level of liveness. For this particular research question, analysing if the participants **spend less time documenting** a software system should be enough to present some conclusions.

6.2 Design

Throughout this section, we will be introducing several aspects that helped designing a fine-tuned experimental design. Questions such as "How were the participants recruited?", "What data source was collected?", "How was data analysed?" and some others, should be answered during the following subsections.

Participants Participants were recruited via a dynamic e-mail to MSc students in FEUP, more specifically, to MIEIC (2nd to 5th year) and MESW students. From the entire community of students in FEUP, we decided to filter those that had background on Design Patterns, since it was a requirement for performing the experiment.

After recruiting the participants, we randomly distributed them among the control and the experimental group. However, to ensure that both groups had the same set of skills and that this random attribution wouldn't be the cause for statistical deviation in the results, we provided a background assessment questionnaire, which was used to evaluate their skills.

Data Sources For the purpose of the experiment, we collected five difference sources of data: 1) the questionnaires answered before and after the tasks execution, 2) the source code that was modified, 3) the produced pattern instances documentation (.png), 4) the tasks' completion times, and 5) the number of times that the participant switched context (between the Intellij and external documentation).

Environment Due to the world's pandemic circumstances, the experiment could not be ran in one of the faculty's laboratories. For that reason, and given that we required a controlled environment, we have decided to perform the experiment through a remote computer control. Each participant would have access to the Intellij IDE, with or without the developed plugin, depending on the treatment group. For some of the tasks, the plugin would already have some pattern instances documentation pre-configured, where for the control group, a pdf with the same documentation was provided.

Procedure The participants were randomly assigned to two groups, corresponding to the two different treatments —- the control group (CG) used a regular Intellij IDE, and the experimental group (EG) used the Intellij IDE with the developed Design-Pattern-Doc Plugin. Additionally, each participant had also access to a GoF design pattern cheat sheet, available during the entire experiment. Throughout the rest of this chapter, this cheat sheet will be referred as the external documentation. The session took around 50 min for each group and was structured the following way:

- **Background Questionnaire (1 min)** — Before starting the tasks, the participants were submitted to a small questionnaire, with the aim of determining how comfortable they were with the tools, technologies and concepts, required for the experiment. This was used to confirm that any statistical deviation in the results wasn't cause by skill dissimilarities between the two groups.
- **Plugin Walkthrough - EG only (2 min)** — The experimental group is also submitted to a quick overview of the developed tool to make them slightly familiar with it. Each tool's feature is presented with a screenshot and a small description. We didn't want the time required to learn the main plugin's features to somehow affect the performance of the experimental group, during their tasks. This was the main reason for providing the experimental group' participants with a quick overview of the developed tool.
- **Tasks (45 min)** — Identifying the pattern instances presented in a software system (comprehension tasks), documenting a software system in terms of its pattern instances, completing a system's implementation by exploring the provided documentation and, finally, the complete cycle — understanding, expanding and documenting a system, are the four tasks that should be completed by each participant.
- **Final Assessment (2 min)** — The participants had to answer a couple of questions designed to assess some effects that cannot be measured objectively by the environment, such as tool features' usefulness.

Data Collection The majority of the data was collected via a google form provided during the experiment: task's total duration, modified source code, produced pattern instance documentation and the answers to the questionnaires. Regarding these questionnaires, we used Likert items with the format: 1) strongly disagree, 2) disagree, 3) neutral, 4) agree and 5) strongly agree.

Some data was collected manually during the experiment by a spectator, who had access to the screen sharing process. This spectator measured the time of each individual task (including sub-tasks), using a stopwatch. Simultaneously, during the execution of each sub task, he counted the number of context switches, in other words, every time a participant left the IDE to access the external documentation (or the other way around). Specifically during EG treatments, he also counted the times that a participant used the documentation features, such as viewing the generated pattern instances when hovering or via the live editor. Even though the participants were asked to submit their answers every time they complete each sub task, we have also asked them to "think aloud" to help predicting and comprehending the participants' intentions, while performing their tasks. This helped the spectator to follow changes in context, like moving to the next question or accessing the external documentation. Additionally, note that some tasks consisted in providing UML class diagrams for representing the pattern instances. Since these diagrams were submitted as images, making the automatic validation of these diagrams too hard to even consider, the spectator would also warn the participant, when they want to move to the next task, if the task was not correct. This ensured that the total time spent during a task was the required to achieve the solution, reducing quick yet wrong answers, that could affect the experiment results.

Data Analysis The data analysis was performed with the help of IBM SPSS Statistics software. All the collected data — task's duration, number of accesses to internal or external documentation, questionnaire answers — where all aggregated in a single SAV file. Running the created syntax SPS file with that data file, results in some descriptive statistics and test results for the t-tests and Mann-Whitney U tests. Additionally, some box plots for the collected data are also generated.

Pilot Experiments The experimental design was validated after recurring to three pilots. In each pilot we tested the procedure and tried to identify unsuspected issues in the developed tool, experiment instructions or in the sort of data that we would like to collect. Given that these pilots followed the same protocol and environment that would be used during the actual controlled experiment, the early detection of problems and post solution, helped achieving a polished design.

6.3 Data Analysis

The focus of the statistical analysis resided on the quantitative data, collected during the experimental sessions, such as tasks duration and number of times the developer switched between the IDE and the external documentation (context switching). The majority of the conclusions came from the use of significance tests, more specifically, Mann-Whitney U tests and standard T-Tests. We tried to use T-Tests by default, but since it assumes normal distribution of the variables being evaluated, we had to resort to Mann-Whitney U test for those variables that did not respect that assumption.

Throughout this section, the used notation denotes CG and EG as the control and experimental groups, u as the u-statistic of Mann-Whitney U tests, and t as the t-statistic of the T-Tests. Additionally, a 95% confidence level was used when interpreting the significance of all statistical test results.

6.3.1 Background

The participants' background was evaluated on the basis of their responses to a form, composed by questions regarding how comfortable the participants were with the concepts and technologies used in the experiment. This would ensure that the two groups were similar and that the possible differences, in the performance of each group, would be a direct consequence of the tools provided during the experiment.

Let us assume the following Hypothesis (H_1): CG $\neq$ EG. Then, we run an independent-samples Mann-Whitney U test to compare the set of skills between the two groups. The results from this test can be seen in Table 6.1.

Table 6.1: Summary of the answers to the background questionnaire

	CG		EG		MW-U		
	$\bar{x}$	σ	$\bar{x}$	σ	H_1	u	ρ
BG1	3.5	0.401	3.64	0.411	$\neq$	49.5	0.714
BG2	3.7	0.335	3.64	0.364	$\neq$	53.5	0.950
BG3	2.6	0.400	2.82	0.325	$\neq$	50.5	0.743
BG4	2.7	0.423	2.82	0.325	$\neq$	54.5	1
BG5	2.6	0.400	2.91	0.343	$\neq$	47.5	0.656

As seen in Table 6.1, there was no significant difference between the scores of the control and experimental groups ($\rho > 0.05$ for every question). This allows us to reject the hypothesis H_1, in other words, the existence of differences between the two groups.

6.3.2 Task's Duration

We would like to investigate if the time spent, during the execution of the tasks, by the control group was significantly greater than the time spent by the experimental group. This can be represented by the following hypothesis:

$$H_2 : \text{CG} > \text{EG} \tag{6.1}$$

We started by inspecting the total time spent by the participants, during the execution of the programming tasks. The box plot presented in Figure 6.1 shows that the control group spent more time than the experimental group. However, this analysis is not enough for the evaluation of the metrics required to answer the research questions presented in Section 6.1. Our experiment was designed to help evaluating those metrics, meaning that some tasks were more focused on the documenting act, while others were more focused on understanding the software systems provided. For this reason, we wanted to confirm the Hypothesis H_2 for every task.

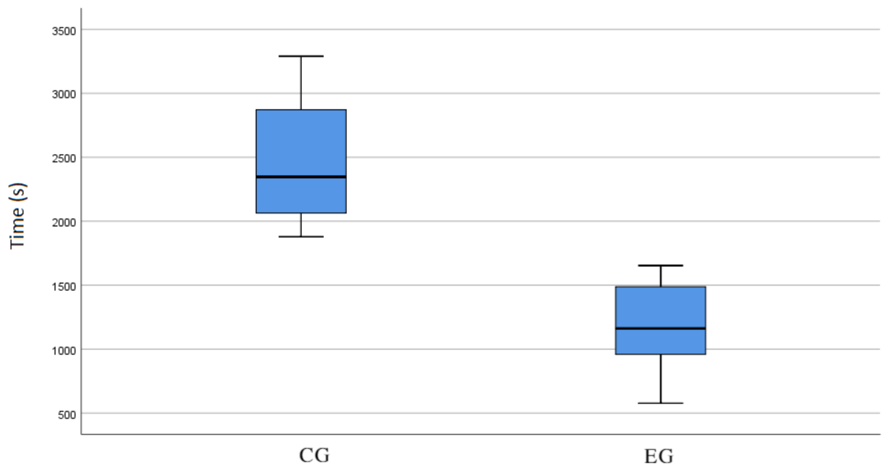


Figure 6.1: Box plot of the total time spent by the participants, in the execution of the tasks

After taking a good look at the task's individual collected data, by observing the box plot presented in Figure 6.2, it is possible to see a clear difference on the task's completion speed between the two treatments, confirming our hypothesis. Nevertheless, we have decided to submit the data to a few of statistical tests, in order to statistically confirm that, with our approach, developers do actually spend less time understanding and documenting a software system.

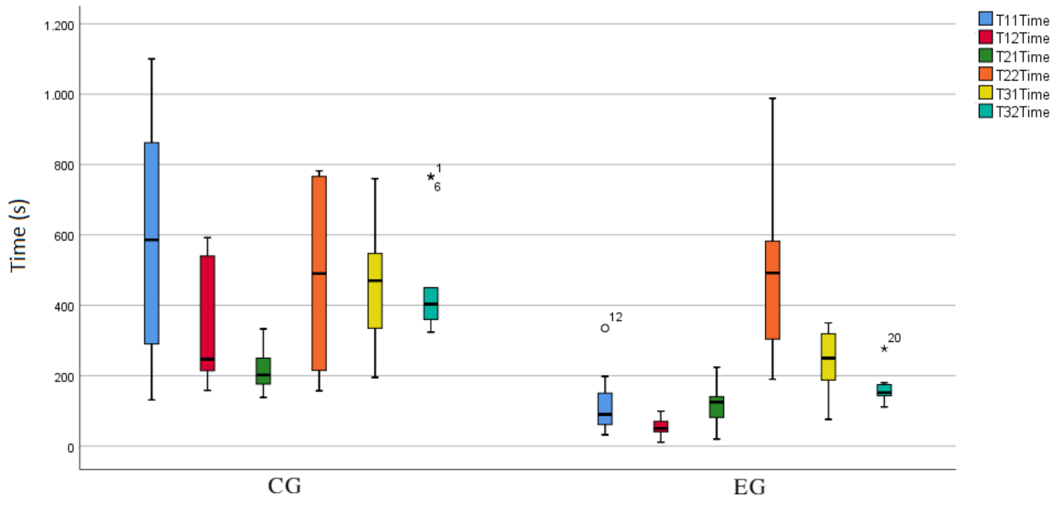


Figure 6.2: Box plot of the task's duration variable in both treatments

The time spent during tasks T11, T22 and T32 does not follow a normal distribution (See Appendix A). Since normality is a requirement for applying t-tests, for these tasks we have opted to perform MW-U tests. The test results can be seen in Table 6.2.

Table 6.2: Summary of the statistical tests' results for the tasks duration variable. The Levene test used to select the correct t-test values can be seen in Appendix B.

	CG		EG			MW-U		T-Test	
	$\bar{x}$	σ	$\bar{x}$	σ	H_2	u	ρ	t	ρ
T11	585.8	104.662	117.3	26.799	✓	6	< 0.001	-	-
T12	325.5	52.102	54.6	7.895	✓	0	< 0.001	-	-
T21	211.8	18.700	114.9	17.237	✓	-	-	3.816	< 0.001
T22	475.3	80.407	483.9	70.574	✗	-	-	-0.081	0.468
T31	468.5	51.428	243.1	26.636	✓	-	-	4	< 0.001
T32	462.2	52.405	163.3	12.929	✓	0	< 0.001	-	-

According to these results, all tasks except T22 have confirmed the hypothesis H_2 (since $\rho < 0.05$). On the other hand, task T22 rejected the hypothesis, meaning that for that specific task the control group was not significantly slower than the experimental group, while solving this task. This could come from the fact that it was a code completion task only, which couldn't be measured as expected since the plugin was not designed to support, directly, code completion tasks. Regarding this type of tasks, the only thing the plugin could do to provide assistance to the developer, would be informing him of which pattern participants were missing to complete the design pattern. For this reason, we ended up measuring something to which it wasn't expected any difference between the control and the experimental groups.

To sum up, we managed to conclude that for those tasks whose purpose was not only to complete code, but also to understand which patterns instances were being implemented by the system or, even the act of documenting the system using pattern instances, the time spent by the control group was significantly greater than the time spent by the experimental group.

6.3.3 Context Switching

As presented in Section 6.1, one of the goals of this experiment was to analyze if by embedding documentation in the IDE, we could reduce the context switching between programming and documenting. We can evaluate this goal by comparing the access to external documentation by the control group versus the experimental group. We believe that the context switching will be higher in the control group. For that reason, let's assume the following Hypothesis H_3:

$$H_3 : \text{CG} > \text{EG} \tag{6.2}$$

We investigated this hypothesis by running an independent-samples Mann-Whitney U test. The results from this test can be seen in Table 6.3.

Table 6.3: Summary of the MW-U statistic results for the context switching variable

	CG		EG		MW-U		
	$\bar{x}$	σ	$\bar{x}$	σ	H_3	u	ρ
T11	4.6	0.792	0.45	0.207	✓	1.5	< 0.001
T12	4.9	1.016	0.18	0.122	✓	2	< 0.001
T21	3.3	0.367	1.09	0.251	✓	6	< 0.001
T22	1.5	0.563	1.91	0.530	✗	46.5	0.289
T31	4.3	0.895	0.91	0.285	✓	11	< 0.001
T32	5.5	0.543	0	0	✓	0	< 0.001

As seen in Table 6.3, the hypothesis was confirmed ($\rho < 0.05$) for every task except task T22. Note that, during task T21 and T22, we provide an incomplete pattern instance documentation of the source code and ask the participants to find which pattern participants are missing (T21) and to complete the implementation (T22). During T21, the participants discover the design pattern that is being partly implemented. Since the external documentation had a complete UML diagram of that design pattern (unlike the one that was given at the beginning of the task T21, which was incomplete), we found out that participants tend to lean on that UML diagram to complete task T22. This also happened with the participants from the experimental group, since the pattern instance documentation displayed by the tool is incomplete and most of them jump to the second sub task without completing it. This is probably the reason why task T22 rejected the hypothesis. In fact, we believe this could have been an issue with the experimental design given that, with our approach, we are trying to reduce the mutual feedback loop between programming and documenting, and in this particular task we are only evaluating the programming outcome. Probably, if we had asked the participants to document the software system, at the end of the task T22, the results would have been different. Actually, task T32, which consisted exactly in this correction proposal — implementation and post documentation of the implemented system — confirmed the hypothesis.

Nevertheless, it was possible to confirm that, for the majority of the tasks, participants that used embedded documentation in the IDE, switched context significantly less times than those that only had access to the external documentation.

During the experiments, the spectator collected data regarding the internal documentation (provided by the plugin) consumption. This was done by counting the access to the documentation displayed when hovering on top of the class name of pattern participants, the pattern instance live editor and the plugin' suggestions. Figure 6.3 illustrates the number of access to internal (ID) and external (ED) documentation, during each software comprehension task. We have seen, with the statistical tests, that the participants from the control group switched context significantly more times. This figure does not only confirm this, it also shows that a possible cause for it was the availability of internal documentation. This embedded documentation allowed the participants to rely on it, decreasing the need for moving out of the IDE.

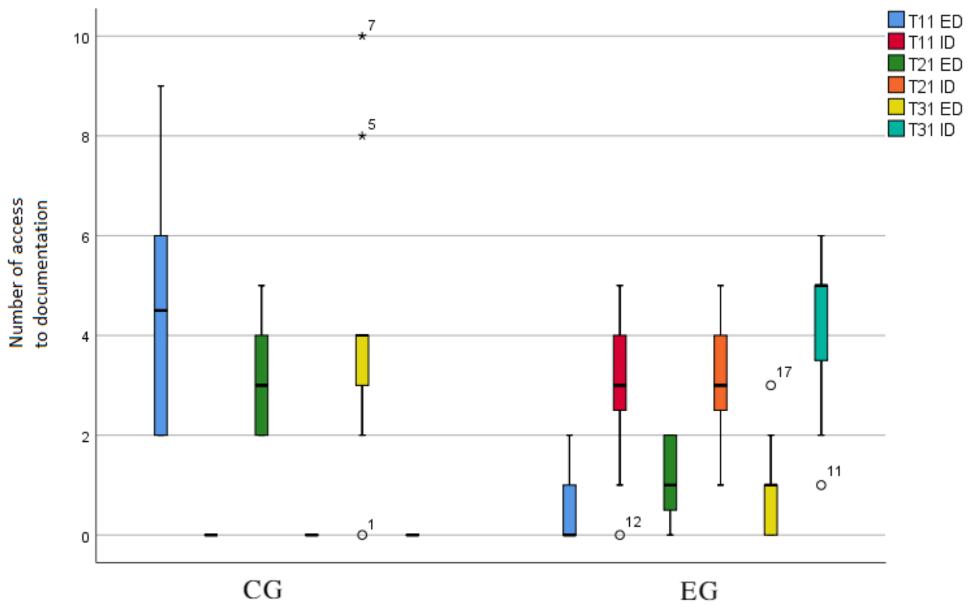


Figure 6.3: Box plot of the number of access to the internal and external documentation, related to the software comprehension activity

Regarding the documenting tasks, which required the participants to represent the pattern instances found in the software systems, in a UML class diagram format, we found out that the participants did not need the help of the external documentation to aid them. This is represented in Figure 6.4. The participants found out that the developed tool provided all the required information and means to perform this type of tasks.

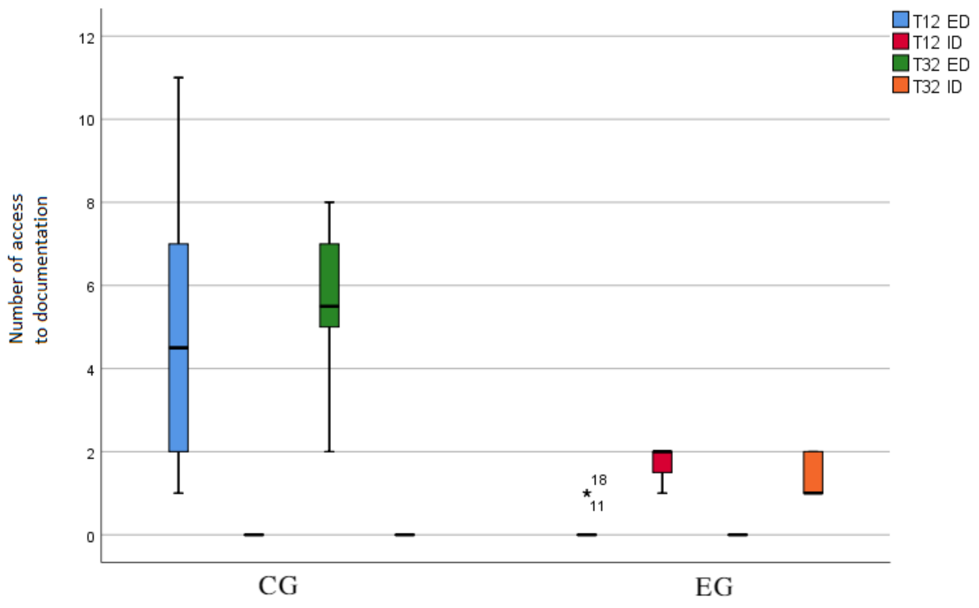


Figure 6.4: Box plot of the number of access to the internal and external documentation, related to documenting tasks

6.3.4 Assessment Questionnaire

In order to understand the participants' perception on the difficulty of solving the requested tasks, we have asked them to answer a final questionnaire. This questionnaire was composed by three questions, more specifically, 1) if the participants found it easy to identify design patterns on the source code, 2) if it was easy to document the code using pattern instances in UML format, and 3) if the communication environment (remote computer) had a negative impact in their performance. We expect that, for the first two questions, the tasks were harder for the CG than they were for the EG. Additionally, we believe that the last question should provide similar responses in both groups.

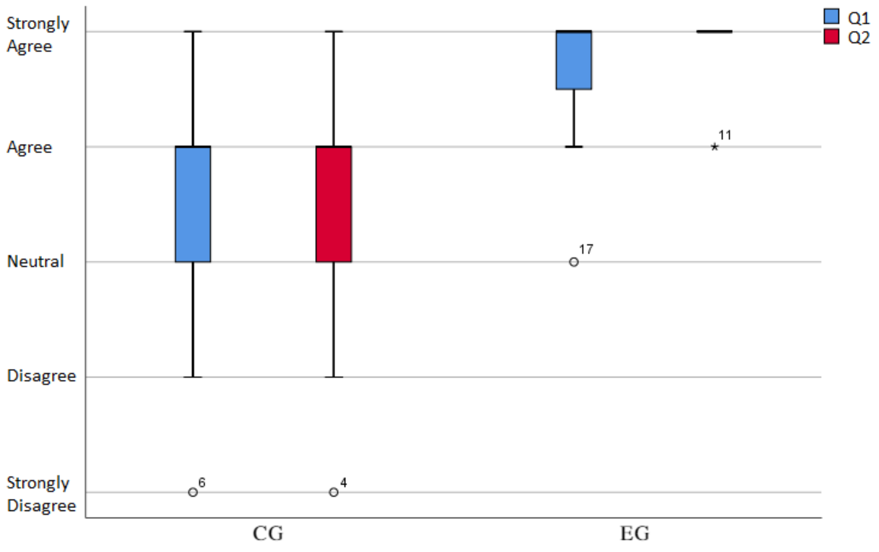


Figure 6.5: Box plot of the Q1 and Q2 in both treatments

We have started by plotting the data collected for the first two questions in order to retrieve some descriptive statistics. Just by looking at the box plot in Figure 6.5, we can see that the experimental group found the identification and documentation tasks easier than the control group did. Nevertheless, we want to investigate if this difference is statistically significant or not. Thus, similarly to the previous analysis, let's assume the following hypothesis:

$$H_4 : \text{CG} > \text{EG} \tag{6.3}$$

$$H_5 : \text{CG} \neq \text{EG} \tag{6.4}$$

The mathematical expression denoted by the hypothesis H_4 was used to investigate the first two questions and H_5 for the last one. Just like we did for the background questionnaire, we also run a Mann-Whitney U test to validate these hypothesis. The results can be seen in Table 6.4.

Table 6.4: Summary of the MW-U statistic results for the final questionnaire

	CG		EG		MW-U			
	$\bar{x}$	σ	$\bar{x}$	σ	H_4	H_5	u	ρ
Q1	3.500	0.373	4.640	0.203	✓	-	19.500	0.003
Q2	3.400	0.371	4.910	0.091	✓	-	8.500	0.000
Q3	1.300	0.213	1.820	0.325	-	✗	40.000	0.268

By looking at the results from the statistical test (Table 6.4), we can confirm the hypothesis H_4 ($\rho < 0.05$). This means that our participants also agree that our approach does indeed help improving software comprehension and eases the documenting process. Moreover, the hypothesis H_5 was rejected by the last question, confirming that there wasn't a group that was more affected by the communication environment than the other. Actually, the participants do not believe that their performance was negatively affected by the remote environment. The bar chart presented by Figure 6.6 shows that the majority of the participants (14 out of 21) chose the lowest Likert scale value, which helps corroborating that conclusion.

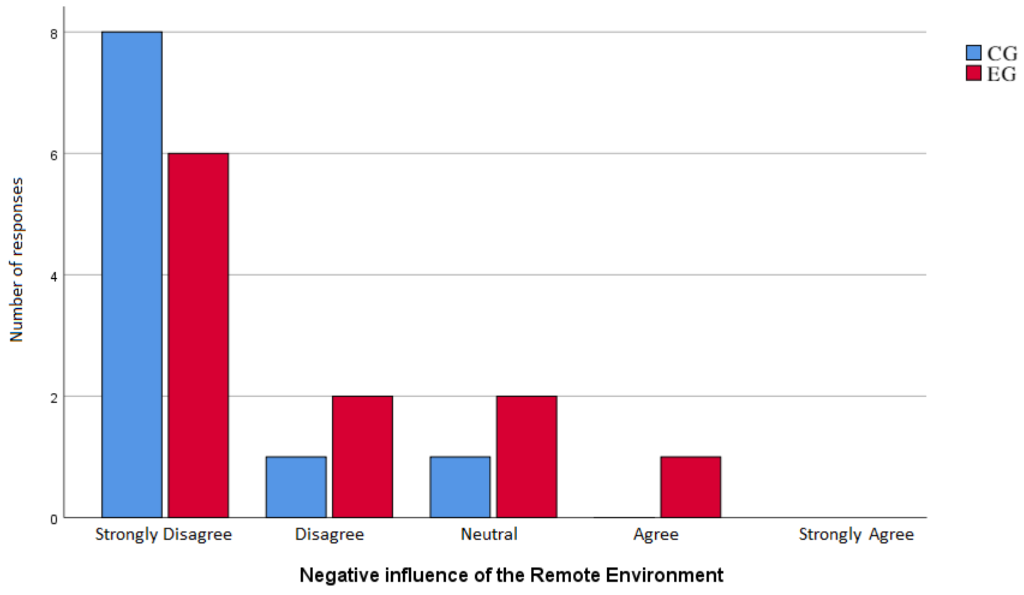


Figure 6.6: Bar chart representing how participants felt for doing the experiment on a remote environment

6.4 Threats to Validity

During the design of the experiment, we thought about the experimental conditions that could cause deviations in the results and, therefore, compromise the validity of our approach. Since we want to obtain sound answers to our research questions, some precautions were taken to discard these threats.

1. **Different Skills** — The design of the experiment assumes that the participants in the control and the experimental group have similar skills and knowledge. Otherwise, we couldn't be certain that the differences in the results, between the two groups, were entirely due to the different treatments used, and not because there could be dissimilarities in the groups. Therefore, we decided to select participants with the same background (all MSc students from MIEIC and MESW), who had the same level of contact with the thematic explored during the experiment. Additionally, we have asked all the participants to answer a background questionnaire, which was used to statistically prove that there were no significant differences between the two groups.

2. **External Factors** — Due to the world's pandemic circumstances in which we were living, when this experiment was conducted, the experiment could not be performed physically in

a university's laboratory, as we had planned. That left us with a remote experiment, which could be a threat to our validity since the results could be affected by the communication environment, such as latency or network breakdowns. Moreover, being remote makes it harder to observe the subject being tested, which may be crucial to understand the entire process towards the task's solution. The first issue was discarded by submitting the participants to a final questionnaire, where they were asked if the communication environment affected their performance. For the second one, we have decided to place a spectator that would watch the entire experiment, by having access (via share screen) to the computer where the experiment was taking place.

3. **Non-Generalizability to Professionals** — Given that recruiting students is easier than recruiting professionals software developers, we used the first ones as the subjects for validating our approach. Additionally, it helps making sure that every participant has a similar skill level. However, we can't help but wonder if the conclusions retrieved, by using them as subjects, can be totally generalized to professionals, due to the lack of experience of the former. In order to tackle this concern, we would have to replicate the experiment, using professionals as the subjects.

6.5 Discussion

During this chapter, we evaluated some of the metrics that shall be used to answer the research questions, presented in Section 6.1. These metrics consisted in 1) **the time spent understanding** and 2) **documenting** a software system, and 3) **the number of context switches**, between the IDE and the external documentation, which we expect to be reduced by embedding the latter in the IDE.

The experiment was designed with the intention of making these metrics easy to evaluate. During the tasks T11, T21 and T31, the participants were asked which pattern instances were being represented in the source code. These tasks were used to evaluate the first metric, more specifically, whether our approach does or not reduce the time required to understand a software system. In Section 6.3.2, with the help of descriptive statistics and the statistical test' results, we found out that the time spent, by participants without the access to the developed tool, completing these tasks was significantly higher the the time spent by those with access to the tool. Given that these three tasks were directly related to the first metric, we managed to confirm that, with our approach we reduce the time required to understand a software system.

Just like the previously mentioned tasks, the tasks T12 and T32, were explicitly designed to evaluate if the act of documenting a software system, could be done with less effort (time) with the aid of our approach. In fact, we have discovered, also during Section 6.3.2, that the participants that used the developed tool were significantly faster than those from the control group. Therefore, we have confirmed that our approach does also reduce the time required to document a software system.

Finally, we evaluated the last metric — the number of context switches. Instead of using some specific tasks, which could bias the outcome, for measuring this metric, we have decided to use the information collected in every task. In Section 6.3.3, we have confirmed that the participants that used embedded documentation in the IDE (provided by the developed tool), tend to switch context significantly less times that those that only had access to the external documentation. Additionally, we have noticed that the participants relied, exclusively, in the developed tool to perform the documenting tasks. During the software comprehension tasks, the participants have also reduced their need to use the external documentation, by inspecting the embedded documentation more often.

Now that we have collected the results for these metrics, we can try to answer the thesis' research questions.

RQ1 *"Can we reduce the inconsistencies between code and documentation?"* — As we have said in the beginning of this chapter, in order to answer this research question, we would have to use the last two metrics. Usually, inconsistencies occur when the act of documenting does not follow the evolution of the software. One of the reasons why this happens is because the programming and development phases aren't always performed simultaneously, making it easy for information to get lost in the process. Additionally, since these phases are firmly related to each other, and involve different types of artifacts, the transition between these phases enforce a constant swap of context. With our approach, we tried to solve this by placing these different types of artifact within the same environment, embedding the documentation artifacts in the same place where the code is edited (IDE), and by simplifying the act of documenting. We have concluded that we do manage to reduce the time required to document a software system, which should reduce the need for the developer to postpone this task, since the effort is much lower. Moreover, given that the creation, edition and consumption of the documentation is close to the source code, it should be easier for the developer to notice when the artifacts are no longer consistent, allowing them to act much quicker. To sum up, we do not solve the inconsistencies problem but we do believe that we provide the means required to reduce their occurrence.

RQ2 *"Can we improve the comprehension of the software's design?"* — This particular research question can be mapped directly to the first and second metrics. Since those two metrics were validated by our approach, we can confirm that our approach does indeed improve the comprehension of the software's design.

RQ3 *"Can documentation be kept updated more easily?"* — Reducing the time required to edit/update the documentation, should reduce the effort for keeping it updated. Since we managed to confirm that, with our approach, developers require less time to document a software system, we believe that we do provide the means to keep the documentation updated more easily.

Chapter 7

Conclusions

This chapter provides a quick overview of this thesis. We start by summarising the conclusions of this work, then we present its main contributions and, finally, we explore the possible paths that could be followed as future work.

7.1 Summary

After doing a literature review on the the main concepts of this dissertation (Chapter 3), specifically, how patterns instances are documented and how are design patterns extracted from the source code, analyzing the several scientific approaches that already exist, it was possible to enumerate the common issues (Chapter 4). As a result, we formulated the following hypothesis:

"By increasing the level of liveness of documentation based on pattern instances, we will streamline the process of switching between creating and using documentation, promoting updated and consistent documentation, and improving the overall understanding of the system's design."

Software development and the creation of its documentation are often treated as different phases, in the overall software development process. Developers had to constantly switch context, which most of the times, led to loss of important knowledge, inconsistencies among artifacts and, subsequently, had a critical impact in the learning process and reuse of software systems.

This is where our approach comes in. Merging the fundamental ideas of literate and live programming, our approach reduces the mutual feedback loop between the programming and documenting phases, helping the overall understanding of the software's design. This is achieved by adding live feedback to the documentation based on pattern instances. Our prototype (Chapter 5) — **Design Pattern Doc** — provides feedback to the developer, while the system is being implemented, regarding which pattern instances should be documented. Furthermore, it generates the required documentation for a specific pattern instance, allows live editing of the persisted pattern instances, and alerts the developer when a pattern instance is incomplete (i.e., when one of the pattern roles isn't being played by an object). This identification of pattern instances is performed with the help of a design pattern detection tool — **DPCORE** — which searches the source code for a list of design patterns. Regarding the visualization of the pattern instances, our tool uses UML diagrams, generated by **PlantUML**, to present them when hovering on top of the class' name of pattern participants.

During Chapter 4, we have presented the research questions that should be addressed by this work. These research questions, were answered in Chapter 6, after performing a controlled experiment with MSc students from FEUP, with the goal of validating our approach. With this experiment we managed to confirm that, with our approach, developers **spend less time understanding** and **documenting** a software system, and the **context switching**, between the IDE and the external documentation, is **reduced**, when embedding the latter in the IDE. These metrics helped answering the research questions.

Can we reduce the inconsistencies between code and documentation? — Yes. Even though we do not solve the inconsistencies problem, we do provide the means required to reduce their occurrence.

Can we improve the comprehension of the software's design? — Yes. By reducing the effort (time) required to understand and document a software system, we improve its comprehension.

Can documentation be kept updated more easily? — Yes. Similarly to the previous research question, by reducing the time required to edit/update the documentation, we reduce the effort for keeping it updated.

7.2 Main Contributions

This dissertation had the following main contributions:

State of the art Review — We explored and compared the different tools/approaches for documenting pattern instances and for extracting pattern instances directly from the source code.

Approach — A solution to the challenge of understanding software systems in terms of its design patterns. We increased the level of liveness of the system's documentation to reduce the mutual feedback loop between the programming and documenting phases.

Design-Pattern-Doc — A prototype for our approach was developed, in the form of a plugin for IntelliJ IDEA, able to analyze source code, live-suggest and generate corresponding pattern-based documentation for that specific pattern instance, and live editing of the persisted documentation.

Controlled Experiment - A group of 21 MSc students completed a set of tasks, for which they had to understand and evolve a software system. This controlled experiment helped concluding that, with our approach, we reduce the time required to understand and document a software system, and we reduce the need for context switching, by embedding the documentation in the IDE.

7.3 Future Work

There are three main aspects that can help discussing this work's future steps: 1) the approach, 2) the prototype and 3) the empirical validation.

Approach

- **Increase the liveness level** — After defining that a class plays a role in a design pattern, the environment could suggest the implementation of the classes that would play the rest of the roles from that design pattern. This would allow achieving level 5.

Prototype

- **Detection Tool Expansion** — **DPCORE** only detects a few design patterns in Java source code systems. It would be interesting to expand the set of supported design patterns, which in the case of the DPCORE, would be easy if the new design patterns were detected by static analysis. This could also be achieved by replacing the chosen pattern instance detector (DPCORE).

- **Dynamic Analysis** — As discussed in Chapter 3, some design patterns can be identified with just a static analysis tool, while others require exploring their dynamic behaviour. As a result, it would be interesting to identify what changes would have to be done, in order to dynamically analyse the source code. Would the liveness level reduce with such changes? Would it be required to actually run the source code? Or just running the tests would be enough? With the static analysis detection, we have managed to achieve level 4 of liveness. If the tool (DPCORE) were to be replaced by a dynamic analysis tool, it would be a challenge to maintain the current liveness level. We are assuming that for this new type of detection tool, we would have to run the system to detect the pattern instances, which would delay the feedback to the developer. If it was possible to detect the majority of the behavioral design patterns just by running tests, then it would be possible to remain with level 4.

- **Pattern Instances View** — Just like there is a class and package view for the JAVA files in Intellij, the existence of a pattern instances view would be helpful for the developer to have quick and easy access to all the persisted documentation. Moreover, it would be interesting to jump to the corresponding source code by clicking, somewhere in that view, on a pattern participant.

Empirical Validation

- **Controlled Experiment with Professionals** — It would be interesting to perform a controlled experiment with professional software developers, in order to discard the threat to validation, presented in Chapter 6. Note that, the presented experimental package can be, with minor changes, directly applied to professional developers.

- **Case Study** — Providing the prototype to a team that is in the field developing a software product, in order to understand, in a more real scenario, the approach' trade-offs. This would help investigating how the approach behaves in a real product with multiple classes and lines of code (eg. if the developers get overwhelmed by the number of pattern instance suggestions).

Appendix A

Shapiro-Wilk Test

This appendix includes the statistical test — Shapiro-Wilk Test — used to evaluate if the task's duration variables were normally distributed in the collected data. The results are presented in Table A.1. As we can see, the tasks T11, T22 and T32 do not follow a normal distribution (ρ<0.05).

Table A.1: Shapiro-Wilk Test results

	Statistic	df	ρ
T11	0.831	21	0.002
T12	0.806	21	0.001
T21	0.984	21	0.966
T22	0.942	21	0.238
T31	0.953	21	0.391
T32	0.829	21	0.002

Appendix B

Levene Test

This appendix includes the statistical test — Levene Test — used to verify if the assumption of equal variances holds for the variables being evaluated with the T-Test.

Table B.1: Summary of the standard t-test statistic results for the tasks duration variable

		Levene Test for equal variances		T-Test
	Equal Variances	Z	ρ	ρ
T21	Assumed	0.037	0.850	0.0005
	Not Assumed			0.0005
T22	Assumed	0.518	0.481	0.4680
	Not Assumed			0.4685
T31	Assumed	1.444	0.244	0.0005
	Not Assumed			0.0010

As we can see from the Levene Test column, in Table B.1, all the tasks have $\rho > 0.05$, which means that we can conclude that the assumption of equal variances holds. From this conclusion, we only have to focus on the first row of T-test results for each task.

Appendix C

Experimental Guide

This appendix includes the questionnaires used during the experimental sessions for the control and experimental groups.

C.1 Control Group Questionnaire

Design Pattern Doc Questionnaire

*Obrigatório

Background Check

This work is centered around the Design patterns from the "Gang of Four", being them Factory Method, Builder, Strategy, Observer, Command, and so on.

1. At this point ... *

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
... I'm comfortable working with Intellij	◯	◯	◯	◯	◯
... I'm comfortable programming in Java	◯	◯	◯	◯	◯
... I'm comfortable working with draw.io	◯	◯	◯	◯	◯
... I know well the GoF design patterns	◯	◯	◯	◯	◯
... I can recognize GoF design patterns in code	◯	◯	◯	◯	◯
... I can implement GoF design patterns	◯	◯	◯	◯	◯

2. Gender *

Marcar apenas uma oval.

◯ Female

◯ Male

◯ Prefer not to say

3. What is the highest degree or level of education you have completed? *

Marcar apenas uma oval.

- ⬭ High School
- ⬭ Bachelor's degree
- ⬭ Master's degree
- ⬭ Doctoral degree

4. If you are a student from MIEIC, what is your current year?

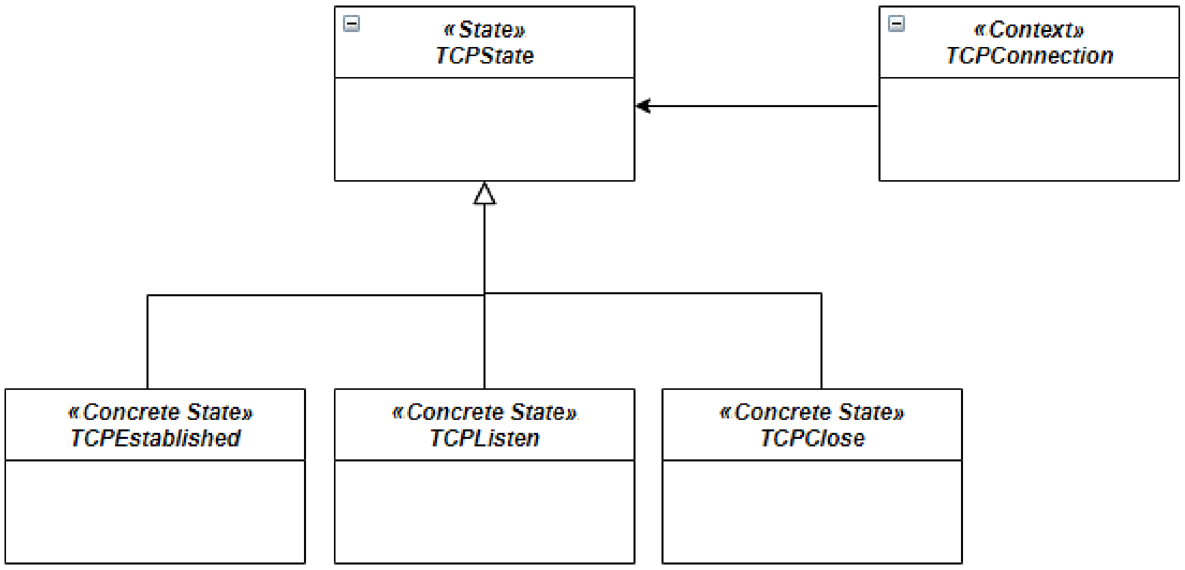
Marcar apenas uma oval.

- ⬭ 1st
- ⬭ 2nd
- ⬭ 3rd
- ⬭ 4th
- ⬭ 5th

Tasks

During this part of the experiment, you will be submitted to a couple of tasks. For each task, you will be given a different source code folder. Note that, at the end of the third task, you will be asked to submit your solutions (more details on how to do this will be discussed later). The experiment will take approximately 40-50 minutes. It is important to do it in a single run, without interruptions.

Some tasks involve submitting class diagrams showing the implementation of pattern instances. An example for this type of task can be seen in the figure below. Note that, we are using stereotypes (<<Role>>), which are not the standard format, for representing the pattern roles. For this type of tasks, we would like you to use: https://draw.io



We suggest to take a quick look at the State design pattern, on the cheat sheet provided, in order to easily comprehend how both diagrams relate to each other. As part of the experiment, you will be given access to the development environment Intellij.

Task 1

5. 1.1. Before starting this task, please write down the current time: *

Exemplo: 08:30

6. 1.2. This source code contains one or more pattern instances. Which design pattern(s) are represented in the system? *

7. 1.3. Document the pattern instances that you have found as a UML class diagram. Do it as was instructed previously, using draw.io, and specifying pattern roles as class stereotypes

   Ficheiros enviados:

8. 1.4. After you have finished this task, please write down the current time: *

   Exemplo: 08:30

Task 2

9. 2.1. Before starting this task, please write down the current time: *

   Exemplo: 08:30

10. 2.2. John is trying to implement a simple system for controlling the light of a lightbulb, in his house. Unfortunatelly, he can't get it to work. Which pattern participant(s) are missing? *

2.3 Create new objects and/or modify those already provided to complete the system.

11. 2.4. After you have finished this task, please write down the current time: *

Exemplo: 08:30

Task 3

The zookeepers from Maia's Zoo have designed a system to easily manage their animal's diet and daily feeding times. Upon the arrival of a new specie to the zoo, the system must be updated accordingly. Today, a specie of Giraffe has arrived to the zoo. Can you help the zookeepers introducing it to the system?

12. 3.1. Before starting this task, please write down the current time: *

Exemplo: 08:30

13. 3.2. Identify the main GoF pattern in this code and explain what changes (objects, pattern roles) would you need to apply to the system to contemplate these requirements *

14. 3.3. Implement those changes and add the pattern instances documentation required to understand the extended system. (Here submit only the class diagrams)

Ficheiros enviados:

15. 3.4. After you have finished this task, please write down the current time: *

Exemplo: 08:30

16. Create a folder containing the folders: 1) task2 and 2) task3. Inside those folders place the respective source code that resulted from your solution to the tasks. Zip the folder and submit it below:

    Ficheiros enviados:

Final feedback

17. Select your opinion towards the following sentences: *

    Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
I found it easy to identify design patterns in the source code	◯	◯	◯	◯	◯
I found it easy to document the code using pattern instances in UML format	◯	◯	◯	◯	◯
The communication environment (remote computer) had a negative impact in the experiment	◯	◯	◯	◯	◯

Este conteúdo não foi criado nem aprovado pela Google.

Google Formulários

C.2 Experimental Group Questionnaire

Design Pattern Doc Questionnaire

*Obrigatório

Background Check

This work is centered around the Design patterns from the "Gang of Four", being them Factory Method, Builder, Strategy, Observer, Command, and so on.

1. At this point ... *

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
... I'm comfortable working with Intellij					
... I'm comfortable programming in Java					
... I know well the GoF design patterns					
... I can recognize GoF design patterns in code					
... I can implement GoF design patterns					

2. Gender *

Marcar apenas uma oval.

- Female
- Male
- Prefer not to say

3. What is the highest degree or level of education you have completed? *

Marcar apenas uma oval.

- High School
- Bachelor's degree
- Master's degree
- Doctoral degree

4. If you are a student from MIEIC, what is your current year?

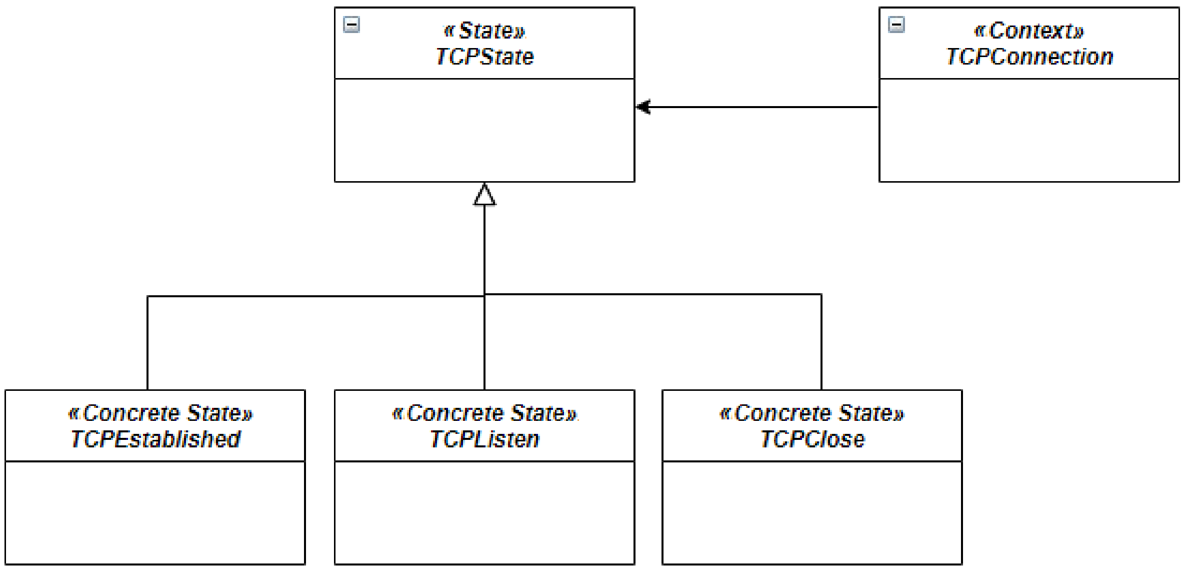
Marcar apenas uma oval.

- 1st
- 2nd
- 3rd
- 4th
- 5th

Tasks

During this part of the experiment, you will be submitted to a couple of tasks. For each task, you will be given a different source code folder. Note that, at the end of the third task, you will be asked to submit your solutions (more details on how to do this will be discussed later). The experiment will take approximately 40-50 minutes. It is important to do it in a single run, without interruptions.

Some tasks involve submitting class diagrams showing the implementation of pattern instances. An example for this type of task can be seen in the figure below. Note that, we are using stereotypes (<<Role>>), which are not the standard format, for representing the pattern roles. For this type of tasks, we would like you to use the plugin discussed on the following section.

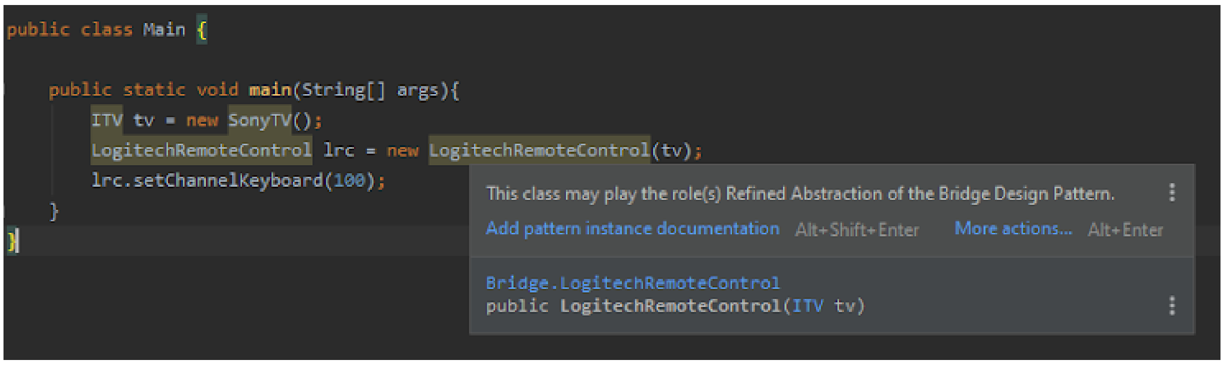


We suggest to take a quick look at the State design pattern, on the cheat sheet provided, in order to easily comprehend how both diagrams relate to each other.

Plugin Walkthrough

As part of the experiment, you will be given access to the development environment Intellij with a plugin for documenting pattern instances. For that reason, we would like to highlight its main features and explain how to take advantage of them. We will start by introducing those that allow creating/updating the documentation:

Feature 1 - Pattern instance documentation suggestion. This plugin uses a design pattern detection tool to scan the project for design patterns. Those detected by it are highlighted as can be seen in the figure below (background yellow-brownish color). Accepting the suggestion will persist the pattern instance documentation for that design pattern.



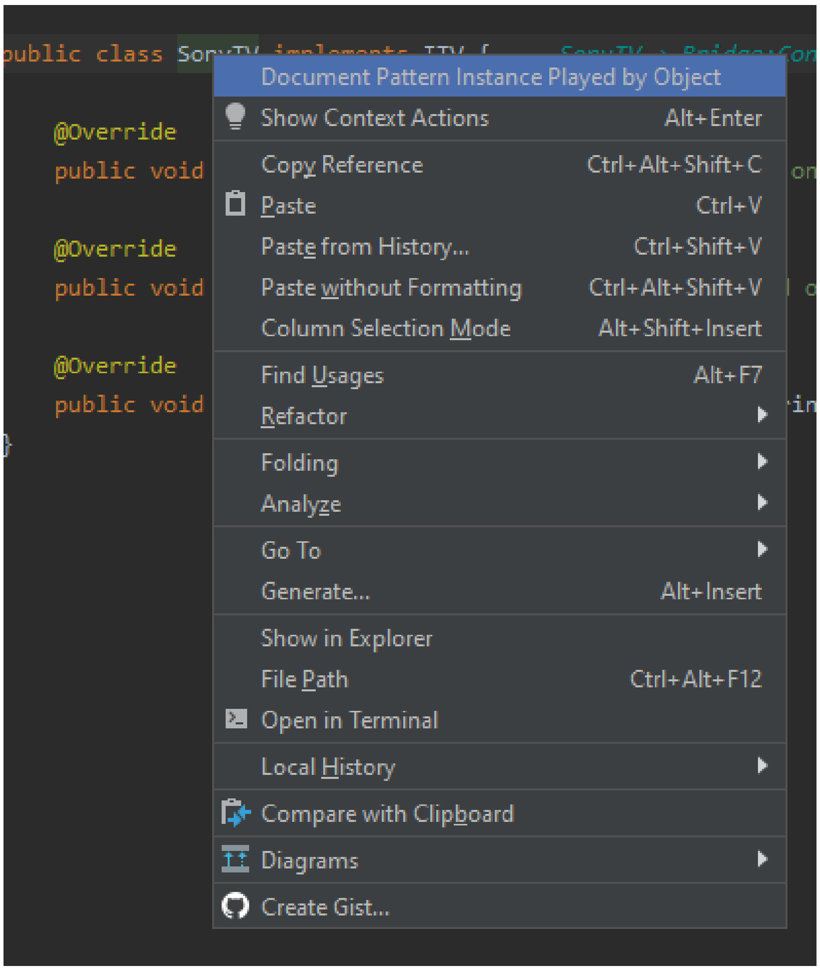
```
public class Main {

    public static void main(String[] args){
        ITV tv = new SonyTV();
        LogitechRemoteControl lrc = new LogitechRemoteControl(tv);
        lrc.setChannelKeyboard(100);
    }
}
```

This class may play the role(s) Refined Abstraction of the Bridge Design Pattern.

Add pattern instance documentation Alt+Shift+Enter
More actions... Alt+Enter

```
Bridge.LogitechRemoteControl
public LogitechRemoteControl(ITV tv)
```

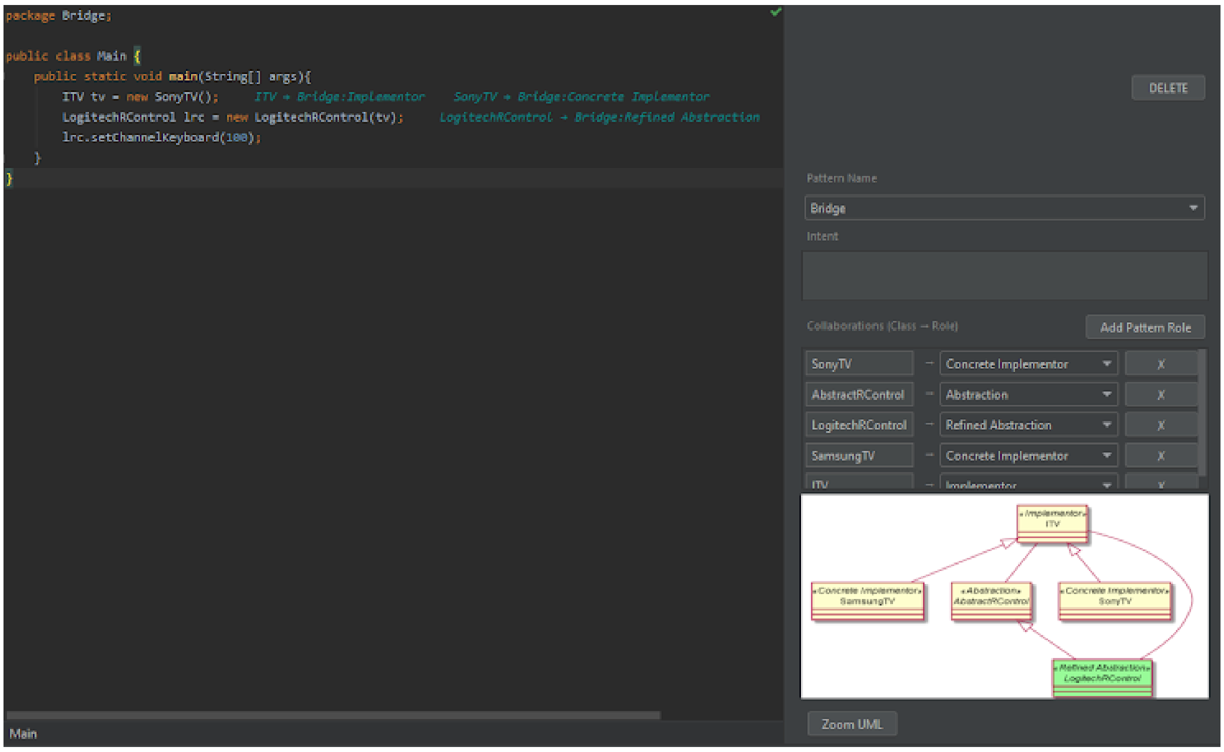
Feature 2 - Manually document pattern instance. By right clicking on a object, a default pattern instance is created for that object.



```
public class SonyTV implements ITV {

    @Override
    public void

    @Override
    public void

    @Override
    public void
}
```

Document Pattern Instance Played by Object
Show Context Actions Alt+Enter
Copy Reference Ctrl+Alt+Shift+C
Paste Ctrl+V
Paste from History... Ctrl+Shift+V
Paste without Formatting Ctrl+Alt+Shift+V
Column Selection Mode Alt+Shift+Insert
Find Usages Alt+F7
Refactor
Folding
Analyze
Go To
Generate... Alt+Insert
Show in Explorer
File Path Ctrl+Alt+F12
Open in Terminal
Local History
Compare with Clipboard
Diagrams
Create Gist...

Feature 3 - Manually edit documentation. When the cursor is on top of an object, its documentation is displayed on the pattern instance documentation editor (window on the right side of intellij editor). This is a live editor of the pattern instances played by that object.



```
package Bridge;

public class Main {
    public static void main(String[] args){
        ITV tv = new SonyTV();      ITV → Bridge:Implementor    SonyTV → Bridge:Concrete Implementor
        LogitechRControl lrc = new LogitechRControl(tv);     LogitechRControl → Bridge:Refined Abstraction
        lrc.setChannelKeyboard(100);
    }
}
```

Pattern Name: Bridge

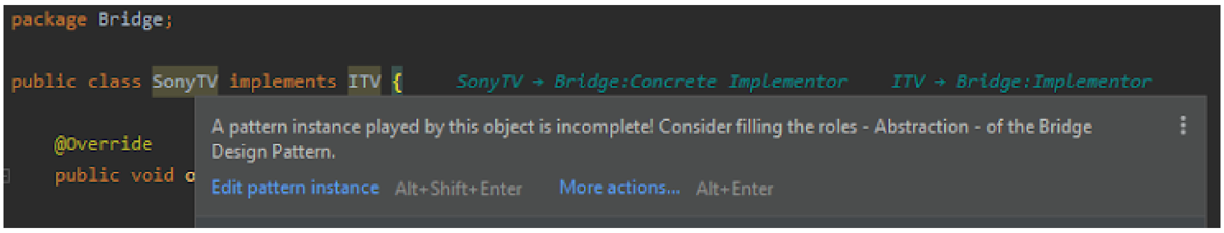
Intent:

Collaborations (Class → Role)

SonyTV	→	Concrete Implementor
AbstractRControl	→	Abstraction
LogitechRControl	→	Refined Abstraction
SamsungTV	→	Concrete Implementor
ITV	→	Implementor

DELETE
Add Pattern Role
X
«Implementor» ITV
«Concrete Implementor» SamsungTV
«Abstraction» AbstractRControl
«Concrete Implementor» SonyTV
«Refined Abstraction» LogitechRControl
Zoom UML
Main

Feature 4 - Incomplete implementation warning. When the code is not consistent with its pattern instance documentation (eg. pattern instance role is missing), objects that play a part in that pattern instance are highlighted. Users can decide whether they would like to edit the documentation or delete it.



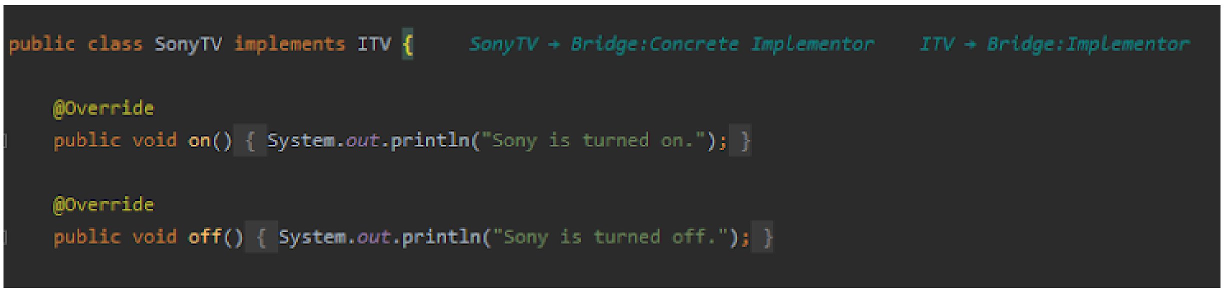
```
package Bridge;

public class SonyTV implements ITV {     SonyTV → Bridge:Concrete Implementor     ITV → Bridge:Implementor

    @Override
    public void o
```

A pattern instance played by this object is incomplete! Consider filling the roles - Abstraction - of the Bridge Design Pattern.

Edit pattern instance Alt+Shift+Enter
More actions... Alt+Enter

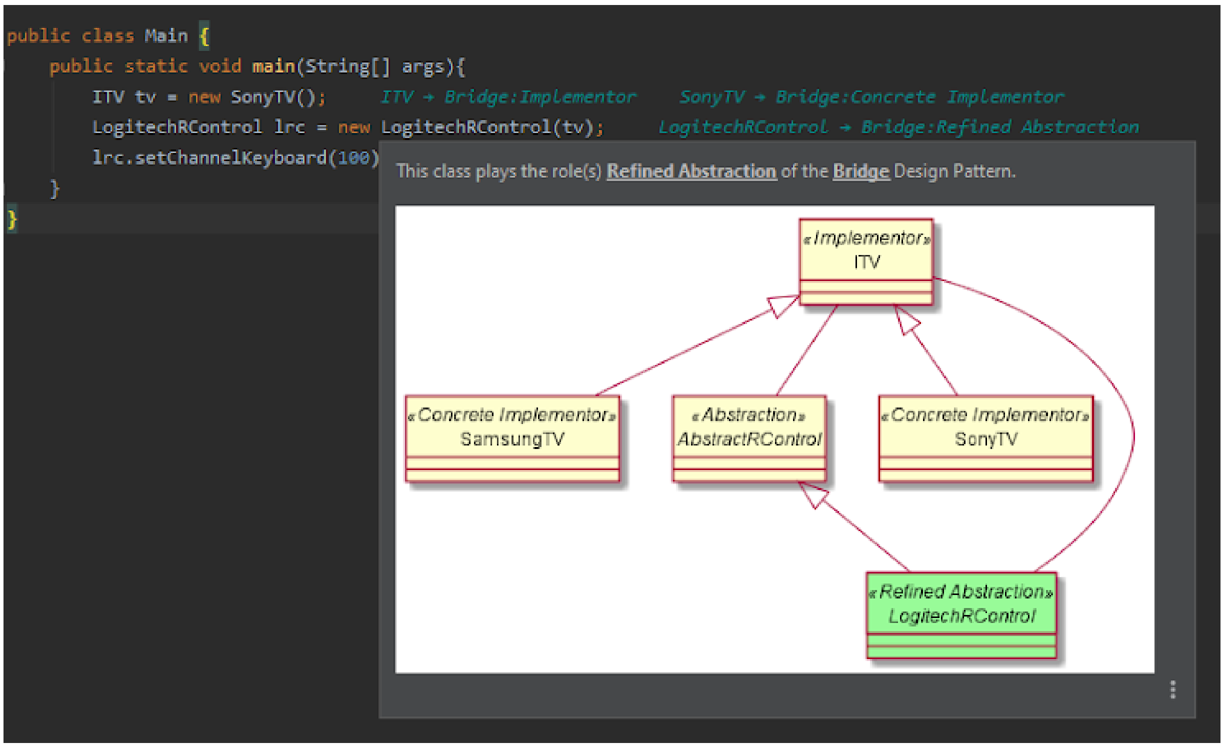
The following features are merely visual. Their purpose is to display the persisted pattern instance documentation:

Feature 5 - Pattern Hints (text at the end of the line where pattern participants are found). These provide information regarding played roles in persisted pattern instances.



```
public class SonyTV implements ITV {     SonyTV → Bridge:Concrete Implementor    ITV → Bridge:Implementor

    @Override
    public void on() { System.out.println("Sony is turned on."); }

    @Override
    public void off() { System.out.println("Sony is turned off."); }
```

Feature 6 - Pattern documentation. When hovering a pattern participant, a pop up dialog appears with the UML pattern instances documentation, played by that object.



```
public class Main {
    public static void main(String[] args){
        ITV tv = new SonyTV();      ITV → Bridge:Implementor    SonyTV → Bridge:Concrete Implementor
        LogitechRControl lrc = new LogitechRControl(tv);      LogitechRControl → Bridge:Refined Abstraction
        lrc.setChannelKeyboard(100)
    }
}
```

This class plays the role(s) Refined Abstraction of the Bridge Design Pattern.

«Implementor» ITV
«Concrete Implementor» SamsungTV
«Abstraction» AbstractRControl
«Concrete Implementor» SonyTV
«Refined Abstraction» LogitechRControl

Task 1

5. 1.1. Before starting this task, please write down the current time: *

Exemplo: 08:30

6. 1.2. This source code contains one or more pattern instances. Which design pattern(s) are represented in the system? *

7. 1.3. Document the pattern instances that you have found as a UML class diagram. Do it as was instructed previously, using the plugin, and specifying pattern roles as class stereotypes.

   Ficheiros enviados:

8. 1.4. After you have finished this task, please write down the current time: *

   Exemplo: 08:30

Task 2

9. 2.1. Before starting this task, please write down the current time: *

   Exemplo: 08:30

10. 2.2. John is trying to implement a simple system for controlling the light of a lightbulb, in his house. Unfortunatelly, he can't get it to work. Which pattern participant(s) are missing? *

2.3 Create new objects and/or modify those already provided to complete the system.

11. 2.4. After you have finished this task, please write down the current time: *

Exemplo: 08:30

Task 3

The zookeepers from Maia's Zoo have designed a system to easily manage their animal's diet and daily feeding times. Upon the arrival of a new specie to the zoo, the system must be updated accordingly. Today, a specie of Giraffe has arrived to the zoo. Can you help the zookeepers introducing it to the system?

12. 3.1. Before starting this task, please write down the current time: *

Exemplo: 08:30

13. 3.2. Identify the main GoF pattern in this code and explain what changes (objects, pattern roles) would you need to apply to the system to contemplate these requirements *

14. 3.3. Implement those changes and add the pattern instances documentation required to understand the extended system. (Here submit only the documentation)

Ficheiros enviados:

15. 3.4. After you have finished this task, please write down the current time: *

Exemplo: 08:30

16. Create a folder containing the folders: 1) task2 and 2) task3. Inside those folders place the respective source code that resulted from your solution to the tasks. Zip the folder and submit it below:

    Ficheiros enviados:

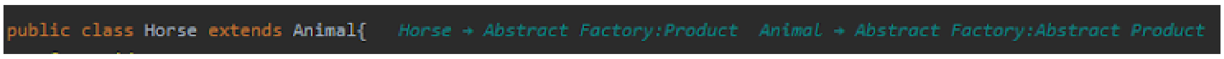
Final feedback

17. Select your opinion towards the following sentences: *

    Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
I found it easy to identify design patterns in the source code	⬭	⬭	⬭	⬭	⬭
I found it easy to document the code using pattern instances in UML format	⬭	⬭	⬭	⬭	⬭
The communication environment (remote computer) had a negative impact in the experiment	⬭	⬭	⬭	⬭	⬭

18. The pattern hints (text in front of each pattern participant line) helped ... *

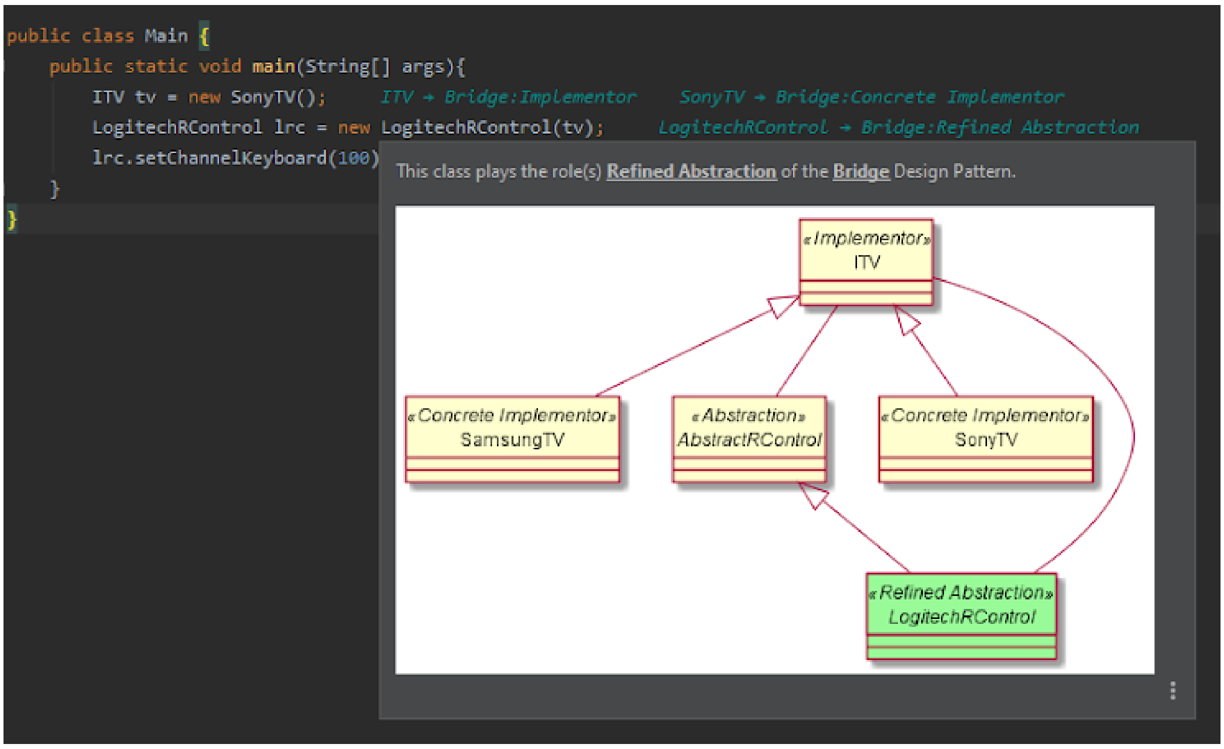


```
public class Horse extends Animal{   Horse → Abstract Factory:Product  Animal → Abstract Factory:Abstract Product
```

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
... understanding the code	⬭	⬭	⬭	⬭	⬭
... solving the tasks quicker	⬭	⬭	⬭	⬭	⬭
... registering and keeping the pattern instance information updated	⬭	⬭	⬭	⬭	⬭

19. The documentation provided when hovering on top of the pattern participants helped ... *



```
public class Main {
    public static void main(String[] args){
        ITV tv = new SonyTV();      ITV → Bridge:Implementor    SonyTV → Bridge:Concrete Implementor
        LogitechRControl lrc = new LogitechRControl(tv);      LogitechRControl → Bridge:Refined Abstraction
        lrc.setChannelKeyboard(100)
    }
}
```

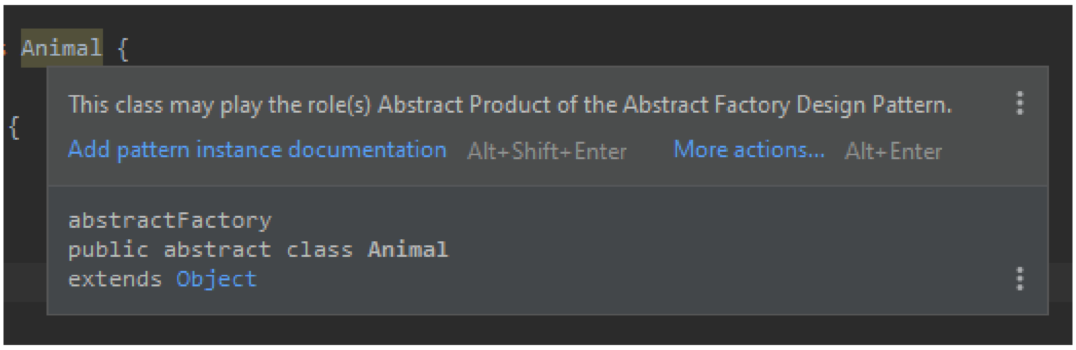
This class plays the role(s) Refined Abstraction of the Bridge Design Pattern.

«Implementor» ITV
«Concrete Implementor» SamsungTV
«Abstraction» AbstractRControl
«Concrete Implementor» SonyTV
«Refined Abstraction» LogitechRControl

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
... understanding the code	◯	◯	◯	◯	◯
... solving the tasks quicker	◯	◯	◯	◯	◯
... registering and keeping the pattern instance information updated	◯	◯	◯	◯	◯

20. The highlighting of pattern participants when suggesting design patterns helped ... *



```
Animal {
{
```

This class may play the role(s) Abstract Product of the Abstract Factory Design Pattern.

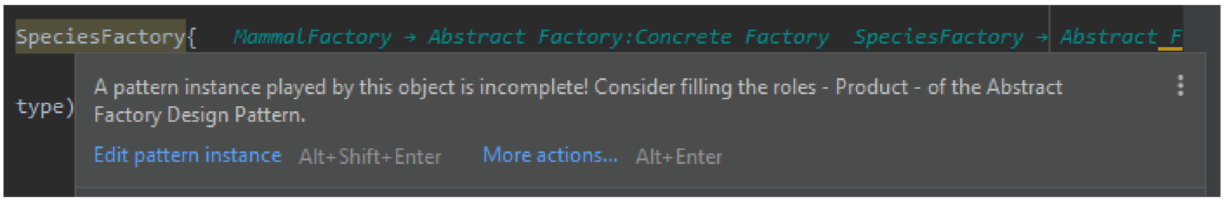
Add pattern instance documentation Alt+Shift+Enter   More actions... Alt+Enter

```
abstractFactory
public abstract class Animal
extends Object
```

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
... understanding the code	⬭	⬭	⬭	⬭	⬭
... solving the tasks quicker	⬭	⬭	⬭	⬭	⬭
... registering and keeping the pattern instance information updated	⬭	⬭	⬭	⬭	⬭

21. The highlighting of pattern participants, when some roles are missing regarding the documentation, helped ... *



```
SpeciesFactory{    MammalFactory → Abstract Factory:Concrete Factory  SpeciesFactory → Abstract_F
type)
```

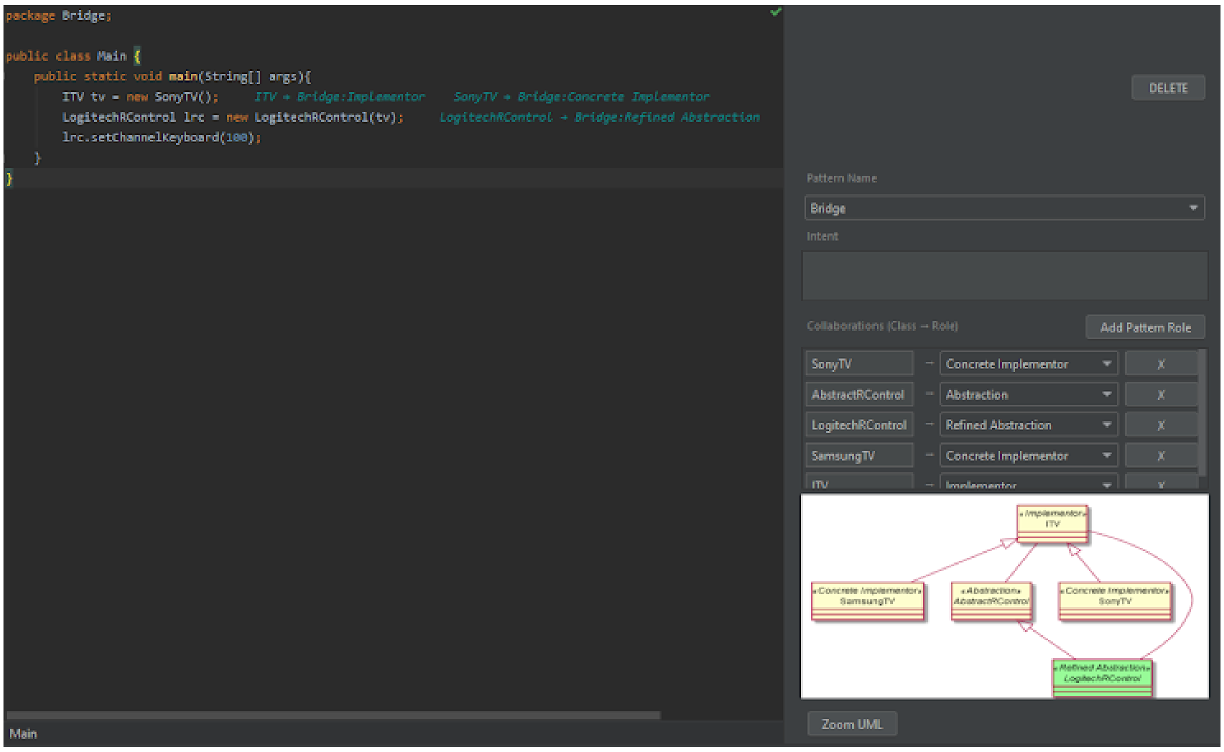
A pattern instance played by this object is incomplete! Consider filling the roles - Product - of the Abstract Factory Design Pattern.

Edit pattern instance Alt+Shift+Enter
More actions... Alt+Enter

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
... understanding the code	⬭	⬭	⬭	⬭	⬭
... solving the tasks quicker	⬭	⬭	⬭	⬭	⬭
... registering and keeping the pattern instance information updated	⬭	⬭	⬭	⬭	⬭

22. The live update of documentation helped ... *



```
package Bridge;

public class Main {
    public static void main(String[] args){
        ITV tv = new SonyTV();     ITV → Bridge:Implementor    SonyTV → Bridge:Concrete Implementor
        LogitechRControl lrc = new LogitechRControl(tv);     LogitechRControl → Bridge:Refined Abstraction
        lrc.setChannelKeyboard(100);
    }
}
```

Pattern Name: Bridge

Intent:

Collaborations (Class → Role)

SonyTV	→	Concrete Implementor
AbstractRControl	→	Abstraction
LogitechRControl	→	Refined Abstraction
SamsungTV	→	Concrete Implementor
ITV	→	Implementor

DELETE
Add Pattern Role
X
«Implementor» ITV
«Concrete Implementor» SamsungTV
«Abstraction» AbstractRControl
«Concrete Implementor» SonyTV
«Refined Abstraction» LogitechRControl
Zoom UML
Main

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
... understanding the code	◯	◯	◯	◯	◯
... solving the tasks quicker	◯	◯	◯	◯	◯
... registering and keeping the pattern instance information updated	◯	◯	◯	◯	◯

Este conteúdo não foi criado nem aprovado pela Google.

Google Formulários

References

[1] Ademar Aguiar. *Framework Documentation - A Minimalist Approach*. Phd thesis, 2003.

[2] Francesca Arcelli Fontana and Marco Zanoni. A tool for design pattern detection and software architecture reconstruction. *Information Sciences*, 181(7):1306–1324, apr 2011.

[3] Bahareh Bafandeh Mayvan and Abbas Rasoolzadegan. Design pattern detection based on the graph theory. *Knowledge-Based Systems*, 120:211–225, mar 2017.

[4] Mario Luca Bernardi, Marta Cimitile, and Giuseppe Antonio Di Lucca. A model-driven graph-matching approach for design pattern detection. *Proceedings - Working Conference on Reverse Engineering, WCRE*, pages 172–181, 2013.

[5] Dirk Beyer and Claus Lewerentz. CrocoPat: A Tool for Efficient Pattern Recognition in Large Object-Oriented Programs. (May), 2003.

[6] Alan Bundy and Alex Blewitt. Automatic Verification of Java Design P. pages 324–327. 2001.

[7] Abdullah Chihada, Saeed Jalili, Seyed Mohammad Hossein Hasheminejad, and Mohammad Hossein Zangooei. Source code and design conformance, design pattern detection from source code by classification approach. *Applied Soft Computing Journal*, 26:357–367, 2015.

[8] Aino Cornils and Gorel Hedin. Statically checked documentation with design patterns. In *Proceedings of the Conference on Technology of Object-Oriented Languages and Systems, TOOLS*, pages 419–430. IEEE Comp Soc, 2000.

[9] Filipe Alexandre Pais de Figueiredo Correia. *Documenting Software With Adaptive Software Artifacts*. Phd thesis, Universidade do Porto, 2015.

[10] Filipe Figueiredo Correia. Supporting the evolution of software knowledge with adaptive software artifacts. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 231–232, 2010.

[11] Filipe Figueiredo Correia, Hugo Sereno Ferreira, Ademar Aguiar, and Nuno Flores. Patterns for consistent software documentation. In *ACM International Conference Proceeding Series*, 2010.

[12] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. A two phase approach to design pattern recovery. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pages 297–306, 2007.

[13] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Behavioral pattern identification through visual language parsing and code instrumentation. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pages 99–108, 2009.

[14] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. An Eclipse plug-in for the detection of design pattern instances through static and dynamic analysis. *IEEE International Conference on Software Maintenance, ICSM*, pages 1–6, 2010.

[15] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Improving behavioral design pattern detection through model checking. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pages 176–185, 2010.

[16] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, Michele Risi, and Ciro Pirolli. EPadEvo: A tool for the detection of behavioral design patterns. *2015 IEEE 31st International Conference on Software Maintenance and Evolution, ICSME 2015 - Proceedings*, pages 327–329, 2015.

[17] Themistoklis Diamantopoulos, Antonis Noutsos, and Andreas Symeonidis. DP-CORE: T design pattern detection tool for code reuse. In *BMSD 2016 - Proceedings of the 6th International Symposium on Business Modeling and Software Design*, pages 160–169, 2016.

[18] Jens Dietrich and Chris Elgar. Towards a web of patterns. *Web Semantics*, 5(2):108–116, jun 2007.

[19] Jing Dong, Sheng Yang, and Kang Zhang. Visualizing design patterns in their applications and compositions. *IEEE Transactions on Software Engineering*, 33(7):433–453, jul 2007.

[20] Rudolf Ferenc, Árpád Beszédes, Lajos Fülöp, and János Lele. Design pattern mining enhanced by machine learning. *IEEE International Conference on Software Maintenance, ICSM*, 2005:295–304, 2005.

[21] Rudolf Ferenc, Juha Gustafsson, László Müller, and Jukka Paakki. Recognizing design patterns in C++ programs with the integration of Columbus and Maisa. *Acta Cybernetica*, 15(4):669–682, 2002.

[22] Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool support for object-oriented patterns. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 1241, pages 472–495. Springer Verlag, 1997.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1995.

[24] Yann-Gaël Guéhéneuc. Ptidej: Promoting patterns with patterns Software Quality and Security View project Software Processes for Video Games Development View project Ptidej: Promoting Patterns with Patterns. Technical report, 2015.

[25] Yann Gaël Guéhéneuc and Giuliano Antoniol. DeMIMA: A multilayered approach for design pattern identification. *IEEE Transactions on Software Engineering*, 34(5):667–684, 2008.

[26] Arne Marius Hallum. *Documenting Patterns.* PhD thesis, Norges Teknisk-Naturvitenskapelige Universitet Fakultet for Matematikk, Informasjonsteknologi og Elektroteknikk, 2002.

[27] Görel Hedin. Language support for design patterns using attribute extension. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 1357, pages 137–140. Springer Verlag, 1998.

[28] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe. Automatic design pattern detection. *Proceedings - IEEE Workshop on Program Comprehension*, 2003-May:94–103, 2003.

[29] Rudolf K. Keller, Reinhard Schauer, Sebastien Robitaille, and Patrick Page. Pattern-based reverse-engineering of design components. *Proceedings - International Conference on Software Engineering*, pages 226–235, 1999.

[30] Donald E. Knuth. Literate Programming. *Computer Journal*, 27(2):97–111, feb 1984.

[31] Hakjin Lee, Hyunsang Youn, and Eunseok Lee. A design pattern detection technique that Aids reverse engineering. *International Journal of Security and its Applications*, 2(1):1–12, 2008.

[32] Fan Li, Qing-Shan Li, S U Yang, and Ping Chen. Detection of design patterns by combining static and dynamic analyses. *Digital Object Identifier*, 11(2):156–162, 2007.

[33] Howard C Lovatt, Anthony M Sloane, and Dominic R Verity. A Pattern Enforcing Compiler (PEC) for Java: Using the compiler. In *Conferences in Research and Practice in Information Technology Series*, volume 43, pages 69–78, 2005.

[34] Lilli Nenonen and Juha Gustafsson. Pattern recognition in the MAISA tool. *Science*, pages 1–12, 2000.

[35] Lilli Nenonen, Juha Gustafsson, Jukka Paakki, and A. Inkeri Verkamo. Measuring object-oriented software architectures from UML diagrams. In *Proceedings of the Fourth International ECOOP Workshop on Quantitative Approaches in Object-oriented Software Engineering*, Cannes, France, 2000.

[36] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendenhals, and Jim Welsh. Towards pattern-based design recovery. *Proceedings-International Conference on Software Engineering*, pages 338–348, 2002.

[37] Georg Odenthal and Klaus Quibeldey-Cirkel. Using patterns for design and documentation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 1241, pages 511–529. Springer Verlag, 1997.

[38] Lutz Prechelt and Christian Krämer. Functionality versus practicality: Employing existing tools for recovering structural design patterns. *Journal of Universal Computer Science*, 4(12):866–882, 1998.

[39] Ghulam Rasool and Patrick Mäder. Flexible design pattern detection based on feature types. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Proceedings*, pages 243–252, 2011.

[40] Ghulam Rasool, Ilka Philippow, and Patrick Mäder. Design pattern recovery based on annotations. *Advances in Engineering Software*, 41(4):519–526, 2010.

[41] J. Sametinger and M. Riebisch. Evolution support by homogeneously documenting patterns, aspects and traces. In *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pages 134–140, 2002.

[42] Reinhard Schauer and Rudolf K. Keller. Pattern visualization for software comprehension. *Program Comprehension, Workshop Proceedings*, pages 4–12, 1998.

[43] Nija Shi and Ronald A. Olsson. Reverse engineering of design patterns from Java source code. In *Proceedings - 21st IEEE/ACM International Conference on Automated Software Engineering, ASE 2006*, pages 123–132, 2006.

[44] J.M. Smith and D. Stotts. SPQR: flexible automated design pattern extraction from source code. pages 215–224, 2004.

[45] Kari Smolander, Kalle Lyytinen, Veli-Pekka Tahvanainen, and Pentti Marttiin. MetaEdit—A flexible graphical environment for methodology modelling. In Rudolf Andersen, Janis A Bubenko, and Arne Sølvberg, editors, *Advanced Information Systems Engineering*, pages 168–193, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[46] Krzysztof Stencel and Patrycja Wegrzynowicz. Detection of diverse design pattern variants. *Neonatal, Paediatric and Child Health Nursing*, pages 25–32, 2008.

[47] Steven L Tanimoto. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming, LIVE 2013 - Proceedings*, pages 31–34, 2013.

[48] Tim Tøese and Scott Tilley. Documenting software systems with views V: Towards visual documentation of design patterns as an aid to program understanding. In *SIGDOC'07: Proceedings of the 25th ACM International Conference on Design of Communication*, pages 103–111, 2007.

[49] Marco Torchiano. Documenting pattern use in Java programs. In *Conference on Software Maintenance*, pages 230–233, 2002.

[50] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, 2006.

[51] Wei Wang and Vassilios Tzerpos. Design pattern detection in eiffel systems. *Proceedings - Working Conference on Reverse Engineering, WCRE*, 2005:165–174, 2005.

[52] Lothar Wendehals. Specifying patterns for dynamic pattern instance recognition with UML 2.0 sequence diagrams. *Proc. of the 6th Workshop Software*, pages 0–1, 2004.

[53] Haotian Zhang and Shu Liu. Java source code static check eclipse plug-in based on common design pattern. In *Proceedings - 2013 4th World Congress on Software Engineering, WCSE 2013*, pages 165–170. IEEE Computer Society, 2013.