Explainer Pattern – A Design Pattern for the Integration of Explanations in Code

Journal:	*IEEE Software*
Manuscript ID	SW-2025-10-0161
Manuscript Type:	Regular
Date Submitted by the Author:	13-Oct-2025
Complete List of Authors:	Deters, Hannah; Leibniz Universitat Hannover Burgath, Lucas; Leibniz Universitat Hannover Schneider, Kurt; Leibniz Universitat Hannover
Keywords:	D.2.11.e Patterns < D.2.11 Software Architectures < D.2 Software Engineering < D Software/Software Engineering, D.2.13.b Reusable libraries < D.2.13 Reusable Software < D.2 Software Engineering < D Software/Software Engineering, D.2.17.i Programming paradigms < D.2.17 Software Construction < D.2 Software Engineering < D Software/Software Engineering, D.2.19.a Quality concepts < D.2.19 Software Quality/SQA < D.2 Software Engineering < D Software/Software Engineering, D.2.3.e Standards < D.2.3 Coding Tools and Techniques < D.2 Software Engineering < D Software/Software Engineering
Abstract:	*With the growing complexity of software systems, the quality aspect of explainability is gaining relevance. Explanations are implemented for AI systems to explain inner workings, but also for everyday software systems to explain further system aspects such as privacy, interactions or domain knowledge. To standardize and facilitate the development process of explanations, we present a design pattern for the integration of explanations using aspect-oriented programming (AOP). The pattern was exemplified for two programming languages, but is suitable for all AOP-enabled programming languages. As a preliminary evaluation of the pattern, we developed a plugin for the development environment Visual Studio Code that supports the implementation of the pattern. This plugin was tested in a preliminary evaluation, revealing benefits in terms of both development time and cognitive load.*

Note: The following files were submitted by the author for peer review, but cannot be converted to PDF. You must view these files (e.g. movies) online.

IEEE Software - Explanation Integration.zip

Article Type: Magazine

Explainer Pattern – A Design Pattern for the Integration of Explanations in Code

Hannah Deters, *Software Engineering Group, Leibniz University Hannover, Germany*
Lucas Burgath, *Leibniz University Hannover, Germany*
Kurt Schneider, *Software Engineering Group, Leibniz University Hannover, Germany*

Abstract—With the growing complexity of software systems, the quality aspect of explainability is gaining relevance. Explanations are implemented for AI systems to explain inner workings, but also for everyday software systems to explain further system aspects such as privacy, interactions or domain knowledge. To standardize and facilitate the development process of explanations, we present a design pattern for the integration of explanations using aspect-oriented programming (AOP). The pattern was exemplified for two programming languages, but is suitable for all AOP-enabled programming languages. As a preliminary evaluation of the pattern, we developed a plugin for the development environment Visual Studio Code that supports the implementation of the pattern. This plugin was tested in a preliminary evaluation, revealing benefits in terms of both development time and cognitive load.

Developing software systems is becoming increasingly complex [1], [2]. This is due to the fact that software systems themselves are becoming more complex and extensive, which in turn leads to software being developed by large, sometimes globally distributed teams [1]. Following standards and design patterns helps in the development of such large distributed projects [3]. Design patterns improve the reusability, comprehensibility, and maintainability of code [4]. There is currently no design pattern for implementing centrally organized explanations. Consequently, explanations are implemented in an undefined manner, which impairs the readability of the code. This makes it difficult for developers to familiarize themselves with new code sections, debug explanation-related errors, and modify or further develop the explainability of a system. To mitigate this problem, we propose a design pattern that handles the organization of explanations centrally. By storing the explanations in a central, globally accessible object, many classes can access the explanations in a reliable manner. We implemented the design pattern in Java and C# as a proof of concept and implemented a plugin for Visual Studio Code that helps to apply the pattern. We provide the Java and C# code in our supplementary material. To demonstrate the usefulness of the pattern, we conducted an initial user study with seven participants who were asked to integrate explanations into a sample project with vs. without the pattern. The results indicate that using the pattern (with the help of the plugin) reduces the cognitive load and the time required.

Digital Object Identifier 10.1109/XXX.0000.0000000

BACKGROUND

Explainability

Explainability is a non-functional requirement that deals with providing information about certain system aspects making these aspects more understandable to the user [5]. Explainability has gained relevance, particularly due to the rise of opaque AI systems [6]. The field of explainable AI (XAI) is primarily concerned with making the inner workings of these so-called black box systems explainable [7]. In recent years, the concept of explainability has also expanded to everyday software systems [8]. This involves explaining system aspects such as domain knowledge, privacy information, and interactions with the system [8].

A challenge in the field of explainability are the highly individual needs for explanations [5], [9]. Depending on the user and context of use, users may need explanations about different system aspects at

different points in the system [10]. Therefore, explanations cannot simply be displayed every time when a function that could require an explanation is executed, since too many explanations have a negative impact on the users' mental load [11], usage efficiency [5], and the overall user experience [12]. Therefore, we are currently researching how users can either actively request an explanation themselves, or how the system can automatically recognize the need for an explanation and then trigger an explanation [10].

Design Patterns

Design patterns are structural or behavioral solutions to design problems that frequently arise in similar forms in software projects [3], [4]. Design patterns build on practical experiences and make the resulting concepts applicable for other developers. Patterns assist in finding appropriate objects, determining their granularity, and their interfaces. They may even determine the appropriate implementation of the objects or how certain objects interact with each other [3]. Design patterns provide as much support as possible in solving recurring design problems, while remaining universally applicable to the design problem at hand.

A well-known design pattern is the *Singleton* pattern [3], [4]. Singleton ensures that a class is guaranteed to have only one instance created and that it has a global access point, so that it controls when and how a client accesses it. It also prevents the namespace from being polluted with global variables.

Aspect-oriented Programming

Aspect-oriented programming (AOP) is a paradigm that enables the modularization of cross-cutting concerns [13]. Functionalities that affect many different parts of an application but cannot be assigned to a single class can be outsourced to separate units (aspects). A frequently mentioned example for AOP is logging [14]. Using AOP avoids the repetition of logging statements in multiple methods of multiple classes.

AOP has a few important core concepts. *Aspects* are modules that encapsulate the cross-cutting functionality [13]. *Join points* are points in the program where an aspect can be inserted, with *pointcuts* determining where the aspects are actually inserted [14]. When an aspect is inserted into a point in the program, so-called *weaving* is performed, whereby the functionality of the aspect is woven into the code [13]. This means that at the pointcut, the functionality of the aspect is executed first before continuing with the rest of the code.

DESCRIPTION OF THE DESIGN PATTERN

We present the design pattern according to the guidelines of Gamma et al. [3]. They recommend reporting 13 items, which we list below.

1 – Pattern Name and Classification

Pattern Name: Explainer Pattern
Classification: Behavioral Pattern, Object Scope

2 – Intent

The design pattern is used to implement centrally organized explanations. Its implementation allows users to request an explanation at any point in the system. During program runtime, the pattern stores explanations at all points that could be unclear (in this case, when executing a complicated method) so that they can be retrieved by the user. The pattern ensures that the most recent explanation is displayed.

3 – Also Known As

Not yet applicable.

4 – Motivation

A system consists of several methods that are executed depending on user input. At various points during use, the user may wonder how the system arrived at a certain output. However, not every user needs an explanation at the same point, so displaying the same explanations for every user would not be beneficial. Our design pattern allows users to call up the explanation that belongs to the last executed method (to which an explanation was previously assigned) at any time during use. To do this, the pattern constantly stores the explanation of the last method executed.

Example scenario 1: A user uses the *COUNT2* function in Microsoft Excel and gets a result that they did not expect. The user does not know what went wrong and requests an explanation, whereupon the explanation for the *COUNT2* function is displayed. To this end, the following steps were taken using the design pattern: When calculating the cell, the system executed the *COUNT2* function, whereupon the explanation for the *COUNT2* was saved by the pattern. Upon request, a class could now request this most recently saved explanation and thus make it available to the user.

Example scenario 2: A navigation device suggests a different route than usual for the same journey. The user does not know why and requests an explanation, and the system shows that a route restriction has been

taken into account. To this end, the following steps were taken using the design pattern: When calculating the route, the system executed the *calculating route* method that calls the *route restriction* method if the systems detects route restrictions. When calling the *route restriction* method, the respective explanation was saved by the pattern and provided upon request.

5 – Applicability

This design pattern can always be applied when a system is to be equipped with centrally organized explanations, i.e., when a user should always be able to request an explanation for the last executed method.

6 – Structure

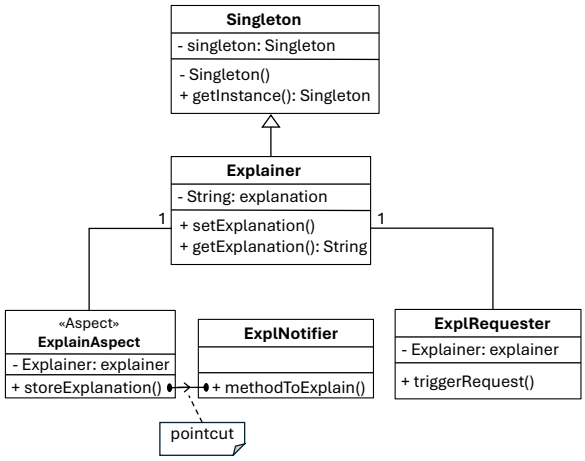

FIGURE 1. Class diagram of the *Explainer* pattern

The structure of the pattern is shown in Figure 1 as a class diagram. Note that we used a stereotype in the class diagram to represent an Aspect (AOP), as there is currently no standardized form for its representation. We also used a custom syntax for representing a pointcut. The illustration means that the aspect **ExplainAspect** is added to the method **methodToExplain()**. Adding an aspect to a method sets a pointcut ensuring that the code in the aspect is woven into the method.

7 – Participants

The following classes are involved in the pattern:

1) **Explainer**: The Explainer class is created at system start as a singleton object. This object is responsible for storing the explanation of the last executed method and for providing it upon request. Since the Explainer is created as a singleton object, it is ensured that only one object exists across the entire system and that every class references the same instance. This ensures that regardless of which class sets an explanation and which class retrieves an explanation, the result is always the same.
2) **ExplRequester**: The object requesting an explanation must have access to the instance of the Explainer class. This allows it to request the current explanation at any time. Usually, this class has a method that determines when a user needs an explanation, which then requests the explanation from the explainer (see method *triggerRequest()* in Fig 1).
3) **ExplainAspect**: This role is a so-called aspect from the AOP paradigm. This aspect is responsible for triggering the storing process of the correct explanation in the explainer object.
4) **ExplNotifier**: An object of this class is executing methods that may require explanations. It is responsible for ensuring that the ExplainAspect is triggered when a method that may require explanations is executed. This is ensured by adding the ExplainAspect to the respective method.

Note that a class can be both **ExplRequester** and **ExplNotifier**. This means that a class may both trigger the saving of an explanation and request an explanation.

8 – Collaborations

Methods of an **ExplNotifier** class that may require explanation must be marked using AOP so that they store the respective explanation in the **Explainer Class** before the method is executed. An **ExplRequester** class has access to the Singleton **Explainer** class and can therefore request the current explanation from the **Explainer** class at any time.

An exemplary collaboration could be as follows: A user input initiates the updating of some data. The **ExplNotifier** class then executes an explainable method, whereupon the pointcut causes the *storeExplanation()* method of the **ExplainAspect** to be executed (AOP). As a result, the corresponding explanation for the method is stored in the **Explainer** object prior to the method being executed normally. Shortly afterward, the **ExplRequester** class recognized that the user needs an explanation, causing a request to the **Explainer** object (*getExplanation*). The **Explainer** object then returns the respective explanation.

9 – Consequences

The advantage of this design pattern is the controlled, clear management of explanations due to the singleton pattern and the AOP paradigm. The use of the singleton pattern has the advantage of resource control and global access. This is particularly important because many classes need to access the same object. Using AOP makes the code easier to read, as all methods that have been assigned an explanation have an aspect. This allows developer and tester to see at a glance which method has been assigned which explanation.

10 – Implementation

The implementation of this design pattern is only feasible for object-oriented languages that support AOP. Ideally, the language should have an existing library for AOP to facilitate implementation. Due to space limitations, we only present a sample implementation of the ExplainAspect in Listing 1. We offer a sample implementation of the entire pattern in both Java and C# on GitLab (see supplementary material).

```
using Metalama.Framework.Aspects;
public class ExplainAspect:
    OverrideMethodAspect
{
  private readonly string _explanation = "";

  public ExplainAspect(string explanationId)
  {
    _explanation = readExplanation(
    explanationId);
  }

  public retreiveExplanation(string
    explanationId)
  {
    /* retrieve explanation based on the
    explanationId (e.g. from CSV file)*/
  }

  public override dynamic? OverrideMethod()
  {
    Explainer.SetExplanation(_explanation);
    return meta.Proceed();
  }
}
```

Listing 1. C# example for the ExplainAspect

Note that the method *retrieveExplanation(string explanationID)* may also contain complex code in which an explanation is actually generated. In our example implementation, we assume static explanations that are stored in a CSV file and can be accessed using IDs.

11 – Sample Code

The specific implementation of assigning aspects looks slightly different in each programming language. When using the *metalama* framework in C#, for example, aspects are added as attributes and are therefore written in front of the method with two square brackets (see Listing 2 line 4).

```
public class ExplNotifier
{
    ...
    [ExplainAspect("E001")]
    public void calculateTaxOfItem(...)
    {
        // Complicated calculations
    }
    ...
}

public class ExplRequester
{
    ...
    public void triggerRequest(...)
    {
        string explanation =
            Explainer.GetExplanation();
        display(explanation);
    }
    ...
}
```

Listing 2. C# example for setting and requesting an Explanation

To request an explanation, the ExplRequester class can send a request directly to the Explainer object (see Listing 2).

12 – Known Uses

Not yet applicable.

13 – Related Patterns

The design pattern uses Singleton for resource control and global access.

In addition, the structure of the pattern is similar to the observer pattern. The difference is that explanations are actively requested and are not automatically distributed each time a new explanation is saved. Furthermore, we used AOP to improve the readability of the code, as the structure is more complex than in the observer pattern. Within the explainer pattern, each class can both save and request explanations. In addition, compared to the observer pattern, many more methods store a explanations, making it more difficult to maintain an overview.

POSSIBLE EXTENSIONS OF THE DESIGN PATTERN

Datatypes for Explanations

In the above examples, we assume that explanations are provided in text form. However, it is also possible that explanations are provided in other data types. For example, in navigation systems, explanations may be provided in audio form. Explanations may also be provided in image or video form.

Types of Explanations

Different types of explanations exist (system behavior, privacy, interactions) [8]. The need for explanations of different types can be identified using different indicators [10]. It may therefore be useful to store explanations for each type of explanation in order to provide the right type of explanation when required. This would mean that the explainer object has a variable for each type of explanation.

Explanation Memory (List)

A user may want to see not only the most recently saved explanation, but also the ones before that. To do this, the explanations could be saved in a list so that the last *n* explanations can always be displayed.

FEASABILITY EVALUATION

As part of the bachelor's thesis of Burgath, a co-author of this paper, the design pattern was implemented as an C# package. We also implemented an IDE plugin to support the use of this package. The objective of the evaluation was twofold: First, we wanted to test the feasibility of the design pattern i.e., whether the pattern can be implemented properly. Second, we wanted to check whether the pattern helps developer to implement explanations when it is integrated into the development environment as a plugin.

Participants

Seven participants took part in the study. Participants were required to have a background in computer science, in particular participants had to have experience with object-oriented programming. Six participants were male and one participant was female.The average age of the participants was 23.3 years(min: 21, max 25, sd=1.4). Three participants were enrolled in a bachelor's degree program in natural sciences, three participants had a bachelor's degree in computer science, and one participant was in their third year of training as an IT specialist.

Study Design

The design pattern was implemented as a C# package. To make it directly usable via interface elements, we also implemented a plugin for *Visual Studio Code*. We prepared a small test application for the study, into which the participants were supposed to integrate explanations. Each participant received a brief introduction explaining the basic C# concepts that were important for the study, a description of the study, and then gave their informed consent to participate.

Procedure. After the introduction, participants were asked to integrate four explanations for defined methods of the sample application. The explanations that had to be integrated were stored in a CSV file. In the first step, participants did not have access to the package and the plugin. Immediately afterward, we measured the participants' cognitive load using NASA task load index (NASA-TLX) [15]. In the next step, the participants were again asked to integrate four explanations into defined methods of the sample application. In this step, the participants had access to the package and the plugin. Again, their cognitive load was measured immediately afterward.

Reasoning for study design We chose a within-subject design in order to have a direct comparison with and without the plugin. Since task completion times and cognitive load scores vary greatly among participants, a comparison in a between-subject design would not have been as meaningful. Note that since the participants were familiar with the structure of the design pattern after using the plugin, we were unable to randomize the task order. Therefore, the participants always completed the task without the plugin first, possibly introducing order bias.

Results

One participant (P5) was not able to complete the task without the plugin within the time limit of 10 minutes. Apart from that, all tasks were completed within the time limit. Table 1 shows the time required to complete the task (time) and the cognitive load of the participants (TLX score).

The results show that the plugin reduced the time needed to complete the task for every participant. Participants were at least 3 minutes and 24 seconds faster with the plugin. The biggest time difference was for participant 6, who was 5 minutes and 59 seconds faster with the plugin.

The plugin also reduced cognitive load in five out of seven participants. Participant 6 showed the greatest difference, with a 63.67 reduction in their NASA-TLX score. Participants 4 and 7, on the other hand, showed

TABLE 1. Results of the user study. The time codes are shown in mm:ss.

Partici-pants	Without plugin		With plugin	
	Time	TLX-Score	Time	TLX-Score
P1	07:49	50.00	04:25	21.00
P2	09:59	49.33	04:51	32.67
P3	06:40	48.33	01:40	34.33
P4	09:21	55.67	05:10	61.67
P5	10:00*	81.33	04:31	15.33
P6	09:57	74.00	03:58	10.33
P7	09:51	59.00	04:11	61.33
mean ($\bar{x}$)	**09:05**	**59.67**	**04:07**	**33.81**

*did not complete the task

a slightly higher NASA-TLX score when using the plugin (an increase of 2.33 to 6).

Threats to Validity

Due to the small sample size, we were not able to perform any significance tests. Therefore, the results of the study should be viewed as indicators rather than evidence. Furthermore, the time values and NASA-TLX scores may be subject to order bias. Since the plugin was always used for the second task, it is possible that participants were already familiar with the test application and were therefore able to complete the task more quickly or with less mental effort. However, due to the large differences, we remain confident that the use of the plugin had a positive impact on the values.

Discussion

The user study was intended to serve as proof of concept, demonstrating that the implementation of the design pattern is generally feasible. The results show that participants were able to use the pattern with the help of a plugin and that it even brings noticeable time savings and reduced cognitive load in most cases.

The results allow initial assumptions about the benefits of the pattern. A lower cognitive load and faster completion times indicate clearer, easier-to-manage code. However, in order to make reliable statements about the improvement in readability and structure of the code, we need to conduct a larger-scale study.

CONCLUSION

In this paper, we introduced a design pattern for the integration of explanations in code. The design pattern uses aspect-oriented programming and provides a structure and behavioral guidelines. We demonstrated the feasibility of the implementation of the pattern in Java and C#. We also implemented the pattern as a combination of a package and a plugin, making it directly usable, and tested this combination in a user study. The user study demonstrated the usability of the plugin and even indicated that the use of the plugin saves time and reduces cognitive load.

In future research, we plan to conduct a larger-scale study to evaluate the design pattern itself. We plan to have different developers implement the pattern in different projects and compile advantages and disadvantages based on their experiences.

SUPPLEMENTARY MATERIAL

We provide an example implementation of the pattern in C# and Java. The code is available on GitLab at: https://gitlab.com/LucasBurgath/explainer-pattern

ACKNOWLEDGMENTS

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Grant No.: 470146331, project softXplain (2022-2025).

REFERENCES

1. Holmstrom, H., Conchúir, E. Ó., Agerfalk, J., and Fitzgerald, B. (2006, October). Global software development challenges: A case study on temporal, geographical and socio-cultural distance. In 2006 IEEE International Conference on Global Software Engineering (ICGSE'06) (pp. 3-11). IEEE.
2. M. Kuhrmann, P. Tell, J. Klünder, R. Hebig, S. Licorish, S. MacDonell (Eds.): Complementing Materials for the HELENA Study (Stage 2). [online] DOI: 10.13140/RG.2.2.11032.65288, published: 2018-11-28
3. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). Design patterns: elements of reusable object-oriented software. Addison-Wesley Pearson Education.
4. Dooley, J. F., Dooley, K., and Green. (2017). Software development, design and coding. Apress.
5. L. Chazette, W. Brunotte, and T. Speith, "Exploring explainability: a definition, a model, and a knowledge catalogue," in 2021 IEEE 29th international requirements engineering conference (RE). IEEE, 2021, pp. 197–208.
6. U.-e. Habiba, M. K. Habib, J. Bogner, J. Fritzsch, and S. Wagner, "How do ml practitioners perceive explainability? an interview study of practices and challenges," Empirical Software Engineering, vol. 30, no. 1, p. 18, 2025.

7. A. Adadi and M. Berrada, “Peeking inside the black-box: a survey on explainable artificial intelligence (xai),” IEEE access, vol. 6, pp. 52 138– 52 160, 2018
8. J. Droste, H. Deters, M. Obaidi, and K. Schneider, “Explanations in everyday software systems: Towards a taxonomy for explainability needs,” in 2024 IEEE 32nd international requirements engineering conference (RE). IEEE, 2024
9. T. Miller, “Explanation in artificial intelligence: In- sights from the social sciences,” Artificial Intelli- gence, vol. 267, pp. 1–38, 2019.
10. Deters, H., Reinhardt, L., Droste, J., Obaidi, M., and Schneider, K. (2025). Identifying Explanation Needs: Towards a Catalog of User-based Indicators. in 2025 IEEE 33rd international requirements engineering conference (RE). IEEE, 2025
11. I. Nunes and D. Jannach, “A systematic review and taxonomy of expla- nations in decision support and recommender systems,” User Modeling and User-Adapted Interaction, vol. 27, pp. 393–444, 2017.
12. H. Deters, J. Droste, A. Hess, V. Kl¨os, K. Schneider, T. Speith, and A. Vogelsang, “The x factor: On the relationship between user experi- ence and explain-ability,” in Proceedings of the 13th Nordic Conference on Human-Computer Interaction, 2024, pp. 1–12.
13. G. Kiczales, J Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier and J. Irwin (1997). Aspect-oriented programming. In: ECOOP’97 — Object-Oriented Programming. ECOOP 1997. Lecture Notes in Computer Science, vol 1241. Springer
14. M. Groves, AOP in .NET: Practical Aspect-Oriented Programming, Manning, 2013.
15. Hart, S. G., and Staveland, L. E. (1988). Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In Advances in psychology (Vol. 52, pp. 139-183). North-Holland.

Hannah Deters is a PhD student at the Software Engineering Group at Leibniz Universität Hannover, Germany. Her research focuses on alternative elicitation techniques for explainability at runtime as part of the research project “softXplain”. Contact her at hannah.deters@inf.uni-hannover.de.

Lucas Burgath is a Master’s student at the Leibniz Universität Hannover. He is passionate about software quality, with a focus on testing methodologies and quality assurance, particularly in the context of C# development. Contact him at lucas.burgath@stud.uni-hannover.de.

Kurt Schneider is a full professor of software engineering at Leibniz Universität Hannover in Germany. He is interested in socio-technical aspects of software and requirements engineering. Contact him at kurt.schneider@inf.uni-hannover.de.