Gang of Four Design Patterns in Machine Learning and Recommender Systems

Patrik Müller

University of Ostrava

patrik.muller@osu.cz

Abstract

This paper attempts to build a more solid bridge between the field of machine learning and software engineering. Although creating demonstration scripts of recommender systems examples became – especially in the era of generative artificial intelligence and coding assistance – significantly simpler and approachable, integrating a recommender solution into the comprehensive system enabling reacting to changes in the highly dynamic environment of recommender systems requires a completely different set of skills and knowledge. Much important information about the real-life scenarios and its practical implications for the implementation of the recommender systems and the integration to the other parts of the larger systems are often neglected in the lectures of recommender systems when being part of machine learning courses. Starting from the basics, we decided to show how the 18 design patterns of the famous "gang-of-four" group could be appropriate for usage in the recommender systems and some of the concepts are applicable even to machine learning in general. To provide an objective reason for using the patterns aside from a perspective of a clean code, better architecture providing more flexibility of integrating new features and re-usability, we also compared some of the patterns in cognitive complexity of the code and *time and space complexity* **(TODO: Determine whether this makes sense or not and whether measuring on particular device is meaningful or do the algorithm analysis).**

Keywords: design patterns, gang of four, recommender systems, machine learning, software engineering, software architecture

1. Introduction

Based on our research, a surprisingly low number of articles connecting the information systems subfields of machine learning and software (SW) engineering exists. Many internet articles and tutorials present code in notebooks (e.g., through the currently popular platform Jupyter) and half-developed solutions without proper structure. Thus, this article attempts to fill the gap between machine learning-oriented theory and the software engineering field, further enriching the theoretical backgrounds of computer science by connecting design patterns with machine learning and information retrieval, with special emphasis on recommender systems.

When more successful applications of machine learning are presented today to the public, there seems to be general optimism about machine learning (the general public often speaks about "artificial intelligence"; however, it often refers to either machine learning or natural language processing). The success of big technological companies is presented in the media. However, the general public might not realize these companies either incorporated machine learning later when resources and technological advancement progressed (e.g., social media that initially had only simple ranking algorithms showing the latest posts), had investors in advance providing needed resources (e.g., OpenAI), or used clever techniques, methods, and algorithms to scale initially from a "garage" environment while still gaining initial traction to later move into more computational power when resources became available (e.g., Google search that initially was not by any means able to perform what it does today but was good enough for users). The reality of starting companies (start-ups) is more difficult. Research indicates that approximately 90% of AI-focused startups fail to achieve sustainable business models, often due to failure to demonstrate product-market fit, ineffective monetization strategies, and organizational challenges (Umbrello, 2021; Crevier, 1993).

Besides business-oriented reasons like marketing or market environment, one reason is technical debt or lack of human and computational resources. Technical debt in machine learning systems—defined as the accumulation of shortcuts and suboptimal design choices that hinder future development—has become increasingly critical as developers leverage AI-assisted coding to accelerate development cycles (Sculley et al., 2015). Software architecture that might not be flexible enough to provide efficient computations, scale, and adapt to changes or security threats might be one of the important factors.

Through the 1960s, 1970s, up to the early 2000s, the field of software engineering often reported problems with the "software crisis," generally referring to the problem of inability of software to keep up with advancements in hardware. The term was formally introduced at the 1968 NATO Software Engineering Conference held in Garmisch, Germany, where leading computer scientists identified that programming projects were chronically late, over budget, and often delivered inadequate or unreliable software (Naur & Randell, 1969). Edsger Dijkstra's 1972 Turing Award Lecture further articulated the crisis, noting that explosive growth in computing power had outpaced programmers' ability to effectively utilize those capabilities (Dijkstra, 1972). In response, several advancements were made—the creation of object-oriented languages (Dahl & Nygaard, 1966; Kay, 1993) or creation of the Agile Manifesto (Beck et al., 2001) are among the most prominent examples of factors that contributed to the future golden era of software.

Both the crisis of AI and software in general were characterized by not fulfilling expectations of businesses and the general public, with hardware advancements outpacing software advancements. Around the same time, the era often referred to as an "AI winter" or "first AI winter" emerged. The term first appeared in 1984 as the topic of a public debate at the annual meeting of the American Association for Artificial Intelligence (AAAI), when leading AI researchers Roger Schank and Marvin Minsky warned the business community that enthusiasm for AI had spiraled out of control and that severe disappointment would certainly follow (Schank & Minsky, 1984; Umbrello, 2021). While technological progress in the software development field is often depicted as having more of an exponential trajectory, the AI crisis is depicted as having distinct periods of boom and bust, with peaks in the late 1960s (the success of Simon, Newell, McCarthy, Minsky, et al., in the field of heuristic search) and low points in the 1970s. Another positive spike was recorded in the era of expert systems (end of the 1980s), which was followed by another low point by the end of the 1990s when there was a need for methods being more flexible, able to learn from mistakes, learn new knowledge, and generalize knowledge. The problems in the AI winter and the software crisis somewhat differed, and each sector dealt with specific challenges—in the field of artificial intelligence, for example, the problem of combinatorial explosion and overwhelming complexity of historical AI programs became prohibitive (McCarthy, 1973).

However, we argue that while scientific progress around methods of artificial intelligence forms one significant dimension and the (un)success of advancements and technological progression of hardware is another significant dimension, software-oriented problems form another dimension and are influenced by factors such as the (in)ability to advance in programming languages, libraries, and software engineering-related tools, techniques, and methodologies in the field of software project management (e.g., positive factors such as the rise of version control systems like Git, more user-friendly project management software, and accumulated project management knowledge from predecessors' experiences).p

Since the majority of machine learning projects today are implemented in Python, and Python is a "hybrid" language, significantly functional and procedural in its nature, principles of clean code and decent software engineering practices are often neglected. Survey data from major machine learning platforms and competitions indicate that Python has dominated the field, with adoption rates exceeding 85% among ML practitioners (Kaggle, 2019; JetBrains, 2021). From this perspective, it does not help that often, popular libraries are used for creating notebooks in a procedural style when programmers work interactively with notebooks in the style of interactive, visually rich consoles where typing and executing a line with a variable leads to printing content to the notebook's output. While these notebooks have significant advantages due to interactivity and visualization, good practices of general software development are often neglected, and readers might be left without knowledge of how individually executed pieces can be put together to form a robust, secure, yet flexible system.

After our research and review of current literature and web articles coming from reliable websites of private sector companies, we suggest the following main software engineering factors that improved the field of artificial intelligence and helped contribute to the rise of AI in the second half of the first decade of the 2000s and 2011-present:

- **Development of version control systems.** These tools enhance collaboration of development and help provide a maintainable and reusable source code base. Distributed version control systems like Git have become essential infrastructure for collaborative software development and knowledge preservation.
- **Development of user-friendly Kanban tools.** These tools enhance collaboration and project management, providing simple ways to track (un)finished tasks.
- **Modern infrastructure of cloud computing, continuous integration pipelines, testing tools, or application containers** which provide stable ways to deploy applications, mitigating differences between local development and production environments.
- **Agile-inspired SW development** provides a way to develop software in iterations or cycles (Beck et al., 2001), which should be beneficial to scientific experimenting compared to the waterfall model.

- **Design patterns** enhance maintainability and reusability of software while using well-tested solutions to common problems (Gamma et al., 1994).

To bridge the gap between fields of machine learning (or applied artificial intelligence, more broadly), we present useful ways design patterns can be used in recommender systems. The reasons why we chose this application are primarily:

1. Design patterns have been well studied since the 1990s, tested, and accepted well both in academic environments and practice, providing one of the solid blocks of the computer science field, which is important as computer science is still a relatively new science field.
2. We propose the subfield of recommender systems as an example for being one of the most important uses of machine learning and information retrieval today, which will likely develop flexibly more further and faster than other areas of artificial intelligence. For example, we assume it is unlikely new internet search methods can beat Google's PageRank algorithm in the near future, and new search engines advance more in the area of privacy rather than making search more precise or faster. On the other hand, recommender systems have more "flavors" and more competing strategies and methods, where strategies of each application can be little different. Also, there is the fact that traditional keyword search is prone to replacement with "AI-powered" search in the form of chatbots able to crawl the internet, which is another reason we assume that the field of recommender systems might be scientifically more interesting in the near future.

The well-known "Gang of Four" design patterns often taught in university courses worldwide aim to standardize common structure and relationships between objects by providing well-tested solutions for common problems in software engineering in general. While other commonly emerging patterns could be identified in the field of machine learning as proposed by some resources (Lakshmanan et al., 2020), the "Gang of Four" should serve as the gold standard pattern providing solid building blocks for better insight into implementation and architecture.

In common examples of design pattern usage either in books or popular internet articles or tutorials for software development beginners, it is usually not specifically mentioned or shown how to use design patterns in the field of artificial intelligence or machine learning. Yet, while developing our recommender system for recommending Czech news articles to users, we noticed many opportunities arise.

Structure of the paper: This paper is structured as follows: Section 2 reviews the state of the art in design patterns and their applications in machine learning. Section 3 presents the methodology for identifying and implementing design patterns in recommender systems. Section 4 describes specific Gang of Four patterns applicable to recommender systems with practical examples. Section 5 presents empirical measurements comparing pattern implementations. Section 6 discusses results and implications, and Section 7 concludes with future research directions.

2. Related Work

The "gold standard" of design patterns and groundbreaking work was published in 1994 in the book *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma et al., 1994). The community refers to this book often as the "Gang of Four design pattern book" or just "GoF book." In this book, authors identified 23 common design patterns as solutions to commonly emerging problems in software engineering, inspired by design patterns from the field of urbanism and architecture. All these patterns help implement the SOLID principles, which should improve certain aspects of code repositories: Single responsibility, Open-closed principle, Liskov substitution principle, Interface segregation principle, and Dependency inversion principle.

In the development of machine learning systems, many other patterns can be identified or were directly defined as design patterns usable specifically for the field of machine learning.

Examples of keywords we used to search for work connecting machine learning and design patterns are: "design patterns in machine learning," "design patterns in recommender systems," "gang of four design patterns in/and machine learning."

Several online articles try to connect these topics (unfortunately containing some nuances and inability to identify the patterns), however not many scientific articles can be found on Google and Google Scholar. We also searched using the Perplexity.ai prompt: "Is there any work writing about a gang of four design patterns used in AI, only restricted to scientific papers, not online blog articles?" Many found papers and articles rather described searching for design patterns in code bases using AI rather than describing what design pattern can be used when implementing machine learning systems.

The systematic literature review (Heiland et al., 2023) finds that many implementation patterns in AI are adaptations of the original GoF patterns to an AI context. The paper cross-references standard GoF patterns (such as Strategy, Adapter, or Decorator) and discusses their application in modern ML and AI system design.

Some patterns they listed are mentioned below. However, even if mentioned, the description of how to use a particular design pattern in machine learning or systems incorporating artificial intelligence methods is usually not described in the paper or the web page tied to the paper. Thus, we identified appropriate use cases of design patterns in recommendation systems, trying to identify only those use cases truly practically useful and beneficial since use of design patterns is not required and sometimes might even have negative effects if used inappropriately, for example, due to increasing complexity and hindering code readability. Some of these use cases were identified after our experience with designing a recommender system for news articles that was deployed in production, used by testing users and also publicly available. However, we also tried to identify use cases for recommendation systems of bigger scale since our recommendation systems were not that complex yet.

Thus, we identified more design patterns from the Gang of Four (GoF) design patterns book (Gamma et al., 1994) and/or provided more specific descriptions of how various design patterns can be used in recommendation systems.

The book *Machine Learning Design Patterns* (Lakshmanan et al., 2020) is dedicated to design patterns but identifies novel or different design patterns. We found, however, even basic and well-tested design patterns of the GoF can be used for implementing parts of recommender systems or even more broadly, programs involving machine learning tasks.

An internet article presents Gang of Four design patterns which can be observed in popular machine learning libraries (however some are incorrectly identified) (Yan, 2021).

Below, we present a description of how particular design patterns can be used in machine learning or recommendation systems and, if possible, provide examples of usage in state-of-the-art machine learning libraries. While technologies can change in the future and some more concrete information may not be relevant, we tried to identify examples that are likely to be maintained several years from now or translate to other examples of usage. Nevertheless, we provided our own examples, some even coded in the repository (**TODO: CITE**) to avoid too much specificity, to generalize some concepts and provide more robust and trend-independent examples providing a balance between information about state-of-the-art tools and more broad, lifelong, and universal concepts.

3. Methodology

Our methodology for identifying and validating design pattern applications in recommender systems consists of four phases:

3.1 Pattern Identification Phase

We systematically reviewed all 23 Gang of Four design patterns (Gamma et al., 1994) to identify those with potential applications in recommender systems and machine learning. The identification process involved:

1. Analyzing the core problem each pattern addresses
2. Examining typical recommender system architectures and workflows
3. Identifying structural and behavioral similarities
4. Validating applicability through existing implementations in ML libraries

3.2 Implementation Phase

For patterns identified as applicable, we developed concrete implementations in the context of a recommender system for Czech news articles. The implementation focused on:

1. Demonstrating practical utility
2. Maintaining adherence to the original pattern definition
3. Ensuring code readability and maintainability
4. Providing both textbook examples and real-world adaptations

3.3 Evaluation Phase

We evaluated selected patterns using objective metrics:

1. **Time complexity:** Measuring execution time for recommendation computation
2. **Space complexity:** Measuring memory consumption during operation
3. **Code complexity:** Analyzing cognitive load using standard software metrics

3.4 Validation Phase

We validated our findings through:

1. Comparison with existing implementations in state-of-the-art ML libraries (PyTorch, scikit-learn, HuggingFace)
2. Review of academic literature on software patterns in AI
3. Practical deployment in production environment

4. Gang of Four Design Patterns in Recommender Systems

4.1 Creational Design Patterns

4.1.1 Factory and Abstract Factory Pattern

In the field of machine learning, Factory and Abstract Factory are often introduced as a first design pattern. Their ability is to simplify classes consisting of creations of long code responsible for creating certain objects of different kinds or to handle creation of objects of different kinds in another single class. By introducing students to using a common interface or abstract class, this also demonstrates how objects can be stored together even in statically typed languages, thus maintaining a variety of different characteristics and behavior of objects while preserving generalization of common actions. For example, we can loop through objects representing certain recommender methods while calling common actions on objects such as recommenderModel.preprocess() and recommenderModel.evaluate(). This can, for example, provide convenient ability to switch between models on evaluation for comparison of models or if performance of certain models decreases.

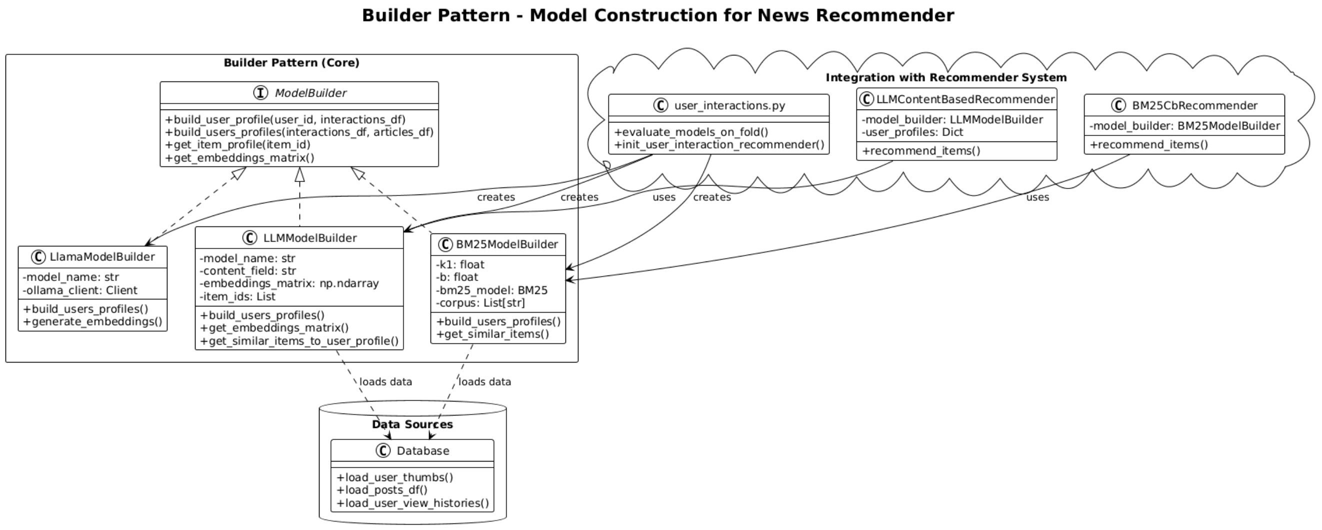
The state-of-the-art example of the Factory design pattern is the AutoModelForCausalLM class of the HuggingFace platform library (HuggingFace, 2024) where the from_pretrained() method is used for providing model names.

4.1.2 Builder Pattern

The Builder pattern can be used for constructing complex objects in steps and allows producing different types of objects with the same code. This pattern should avoid overcomplicated constructors or client code (the portion of the program where calling methods of other classes happens). This pattern is practically used often in cases when a usual symptom of an overcomplicated constructor emerges: the number of parameters in the constructor gets too large. After an object is created, then only necessary methods can be called (only necessary steps can be performed) specific to the need of the object. However, specific implementations could be needed; thus, it is also possible to provide different implementations for different classes. Usually two main parts are declared: the *director* responsible for defining steps (it is possible to just call it from client code; however, for cleaner code, it is often advised to implement it in a separate class) and the *builder* responsible for implementing steps. This pattern was mentioned in a related website of the research team (Heiland et al., 2023), however no description was provided.

In recommender systems or machine learning in general, this can be used for initializing complex recommender pipelines where initialization consists of several components such as machine learning model, feature extractor, data pre- or post-processors, etc. If a simpler recommendation mechanism is needed, it is possible to assemble a smaller object consisting only of simple parts.

We provided our own example of the Builder pattern in the repository (**TODO: CITE**) showing how the Builder pattern can be used for executing necessary methods specific to certain recommendation methods/strategies and how it can be extended to use of debugging class showing fundamental information about data, e.g., showing the number of items or parameters.



4.1.3 Prototype Pattern

Creates new objects by copying an existing object, known as the prototype. This can be beneficial if the object consists of many time-consuming parts, e.g., data loading or data preprocessing. In some languages or libraries, the solution for copying objects is provided by native built-in methods, e.g., Python's copy() method or the subsequent deepcopy() method, both from the default module *copy*. The paper (Heiland et al., 2023) did not include this pattern, but it is a GoF behavioral pattern (same with following patterns if not mentioned differently). The copy/deepcopy usage of the pattern was identified, for example, by the blog article of the author of the book (Shvets, 2024).

In recommender systems and machine learning, many small usages of this pattern can be found. For example, it can be used for copying original data before preprocessing was made, so later either preprocessed data or original data can be dynamically used based on actual performance.

4.2 Structural Patterns

4.2.1 Adapter Pattern

Makes objects with incompatible interfaces compatible by putting common methods in the interface. This pattern was mentioned in a related website of the research team (Heiland et al., 2023), however no description was provided.

This design pattern has very practical usage in recommender systems but can be extended also for machine learning in general. We might assume all machine learning models will consist of methods such as fit(), predict(), and in recommendation systems, methods used for recommending articles, such as recommend(). The Adapter pattern can be used to provide compatibility between different libraries.

```
class ModelAdapter:
    def preprocess(self, raw_data): pass
    def predict(self, preprocessed_data): pass
    def postprocess(self, predictions): pass

class SklearnAdapter(ModelAdapter):  # Adapts scikit-learn models
class TorchAdapter(ModelAdapter):    # Adapts PyTorch models
```

Some state-of-the-art machine learning libraries try to also provide compatibility between other libraries: tf.keras.wrappers.scikit_learn.

TODO: Really those?

- xgboost.XGBClassifier, lightgbm.LGBMClassifier, catboost.CatBoostClassifier), Keras (**TODO: CITE:** tensorflow.keras.wrappers.scikit_learn.KerasClassifier) and PyTorch (**TODO: CITE:** Trainer).
- **TODO:** Draw the hierarchy to the diagram.

In the recommender system developed by us, we found also usage of both the Adapter pattern for mocking item-to-item recommendations (query is one of the items and recommendations are other items) when the recommender system was designed initially by the customer's request to be implemented for providing results based on user data. Since many features of users and profiles were shared in both entities, we could create the User adapter from the item and provide item-to-item recommendations even though the method or function was designed to work with user data without the function or method being able to spot any difference. By using the ItemToUser adapter, we were quickly able to respond to changed user requirements. Although it initially looked like a major problem to the design, with usage of adapter, new features got quickly adapted and delivered. The ideal design would be to generalize and use abstract class as a parent with common attributes and methods from which individual cases can extend and specialize. However, to deliver products fast and keep customers satisfied, the Adapter pattern can often provide a suitable temporary solution for problems like this.

4.2.2 Bridge Pattern

Decouples an abstraction from its implementation. Practically it is used to split overly complex or large classes or several related classes into abstraction and implementation that becomes independent and can then be modified without affecting each other. In practice, the Bridge pattern can be used to split one large class with all related methods into an implementation interface and abstraction responsible for loading data and possibly other supporting classes handling one particular area of responsibility. The Abstraction interface would provide high-level control logic but rely on implementation to do actual work. We can observe resemblances of the pattern in various libraries (e.g., PyTorch, historically also in Keras), however it is not a textbook example and some existing code mixes several patterns together. However, an example would be to split data access in the implementation part of the bridge. The interface can define the set of common data access methods such as get_user_history(), get_items_data(), and concrete implementation can specify particular data access to offline or online repositories in different types of databases (e.g., different relational database systems, document databases, or offline data sources). The implementation is the recommender with the methods.

Potentially it can be a parent class, while particular recommender types can be children (but hierarchical implementation is not mandatory for the pattern to be classified as a Bridge pattern).

In summary the bridge pattern can provide the following benefits:

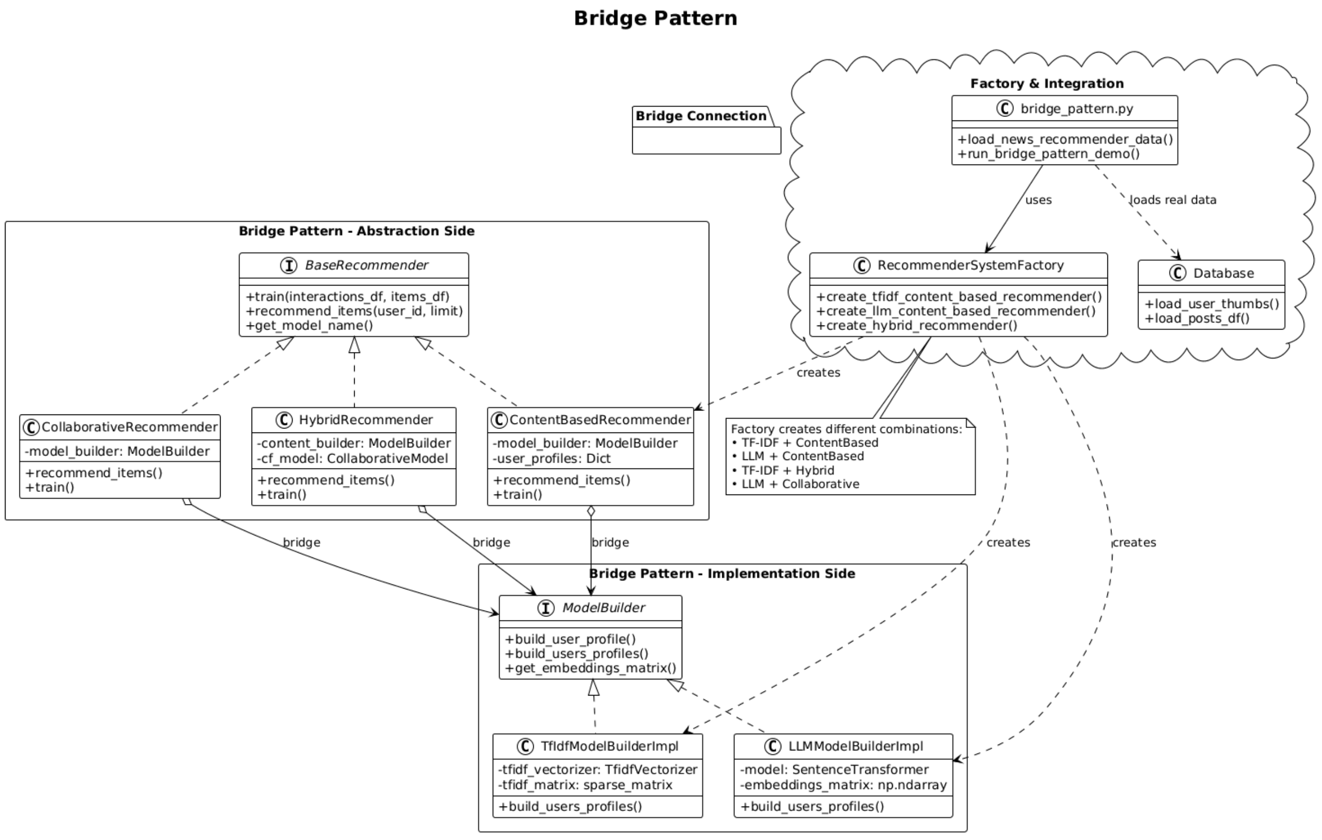
• Decouple abstraction from implementation. In the recommendation system environment, this can mean, there will be a bridge between the Recommendation as an abstraction and a Model as an implementation.

• Provides the ability to possibly switch implementations at runtime.

• Extend both sides independently.

• Uses REAL database data

We can even combine this with the builder pattern and have the ModelBuilder on the implementation side of the bridge. And even further, the factory can be also used to create the particular types of the recommendation methods as the recommendation methods are the refined implementations on the implementation side of the bridge.



TODO: Add Additional references: PyTorch distributed systems (PyTorch, 2024a; Gloo, 2024; NCCL, 2024).

4.2.3 Composite Pattern

Composes objects into tree structures to represent hierarchies. This design pattern can be used when constructing complex recommendation systems consisting of many individual recommendation methods. For example, the first level of the tree might consist of "cold-user" and "warm-user," and below these there can be child recommendation methods more appropriate for recommending items with distinguishing between the situation when we have a new user and we haven't gathered much user data. For example, for the "cold user" case, we might use the Popularity method (e.g., the most viewed or best-rated articles of all users) and Content-based methods (and its related classes and created objects). However, for the "warm user" case, we can rather use collaborative filtering or even the hybrid approach consisting of several methods including the collaborative one, e.g., the popular and successful two-tower recommendation method consisting of user vector embeddings and embeddings created from the content itself. Another part is the refined abstraction providing modalities of implementations. The client side of the program besides working with abstractions is responsible also for linking abstraction to particular implementation objects via aggregation.

We can imagine this pattern being useful in large e-commerce recommender systems. The Abstraction could be the RecommendationEngine, and then ConcreteAbstraction would be refined as: MovieRecommendationEngine and BookRecommendationEngine. The implementation interface could be named RecommendationAlgorithm, and its concrete implementations would follow: CollaborativeFilteringAlgorithm, ContentBasedAlgorithm, HybridAlgorithm, etc. The RecommendationEngine would hold reference to a RecommendationAlgorithm, delegating algorithm-specific behavior to it.

This design pattern or similar derivations are used in machine learning often for constructing preprocessing or training pipelines. Examples are the Sequential classes in PyTorch (PyTorch, 2024b) and Keras (Keras Team, 2024), or the well-known Pipeline class from scikit-learn (Pedregosa et al., 2011).

4.2.4 Decorator Pattern

Adds new functionality to an object dynamically, without altering its original structure. This pattern was mentioned in a related website of the research team (Heiland et al., 2023), however no description was provided.

We can use this to apply certain filters to the base recommendation model. For example, we might apply diversification of recommendations or business rules of certain companies or institutions. For example, in client code it could be wrapped such as:

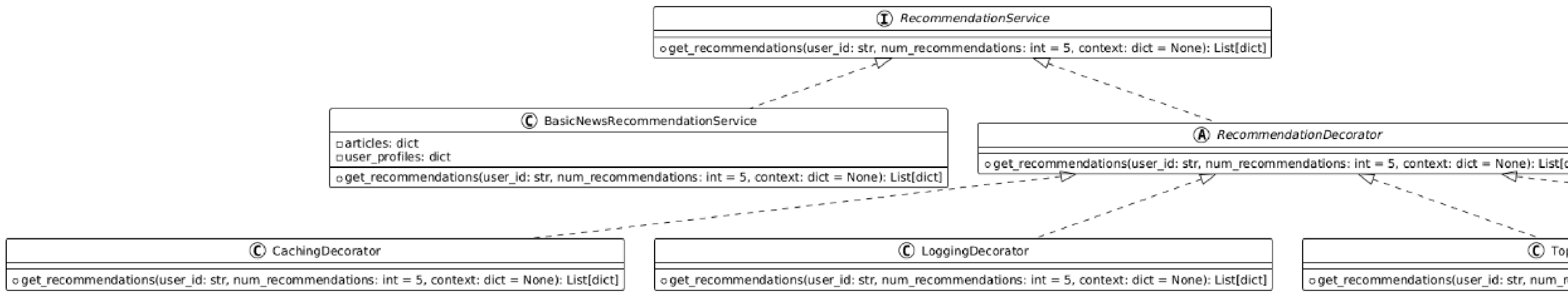
```
BusinessRuleDecorator(DiversityDecorator(BaseRecommender()))
```

The other example could be wrapping such as:

```
ExcludeViewedItemDecorator(DiversityDecorator(collaborative_algo))
```

which would handle user history and exclude already viewed items. Some languages denote decorators by the '@' symbol. This is not the same concept as the GoF design pattern; however, it shares the same concept of adding some functionality to existing functions. A typical practical decorator used often in machine learning, recommendation systems, but also any other more complex online information system requiring low latency is Python's `@lru_cache` decorator from default functools library, which we used on multiple places in our previous work (**TODO: CITE: JIMP paper News recommender**)—for example, for caching content-based recommendations based on queries consisting of user-entered keywords or single queried document. Decorators are also often used in Java or Python for denoting test fixtures or in Java as a well-known decorator for overriding base class methods. However, again, these decorators are not an example of the decorator design pattern.

Decorator Pattern - News Recommender System (simplifi


RecommendationService
get_recommendations(user_id: str, num_recommendations: int = 5, context: dict = None): List[dict]
BasicNewsRecommendationService
articles: dict
user_profiles: dict
get_recommendations(user_id: str, num_recommendations: int = 5, context: dict = None): List[dict]
RecommendationDecorator
get_recommendations(user_id: str, num_recommendations: int = 5, context: dict = None): List[
CachingDecorator
get_recommendations(user_id: str, num_recommendations: int = 5, context: dict = None): List[dict]
LoggingDecorator
get_recommendations(user_id: str, num_recommendations: int = 5, context: dict = None): List[dict]
To
get_recommendations(user_id: str, num_r

4.2.5 Flyweight Pattern

Minimizes memory usage by sharing as much data as possible with other similar objects.

Usage in recommender systems arises from the usual dilemma between obvious benefits of using big data that should enhance performance of retrieval and memory complexity. If we have many users (e.g., in the range of millions, each with a list of recommended movie IDs), instead of storing a separate full object representing Movie for each user's recommendation list, we might store shared state with metadata of movies (title, genre, year, etc.). This could be a single object per movie. Then we would have data unique per user: rating, whether purchased, time of interaction—stored outside the shared object. Users then reference the same flyweight Movie object instead of duplicating memory. In our repository (**TODO: CITE: GitHub repo with the codes**), we show how resemblance of Flyweight pattern (not exactly matching "textbook" examples but strongly resembling the core of the pattern) is used in the spaCy library, and we also provide an anti-pattern of how it would be used if implemented from scratch and using the non-sophisticated, naive, memory-consumptive solution which would create a new object on each iteration of the loop (e.g., through words in document).

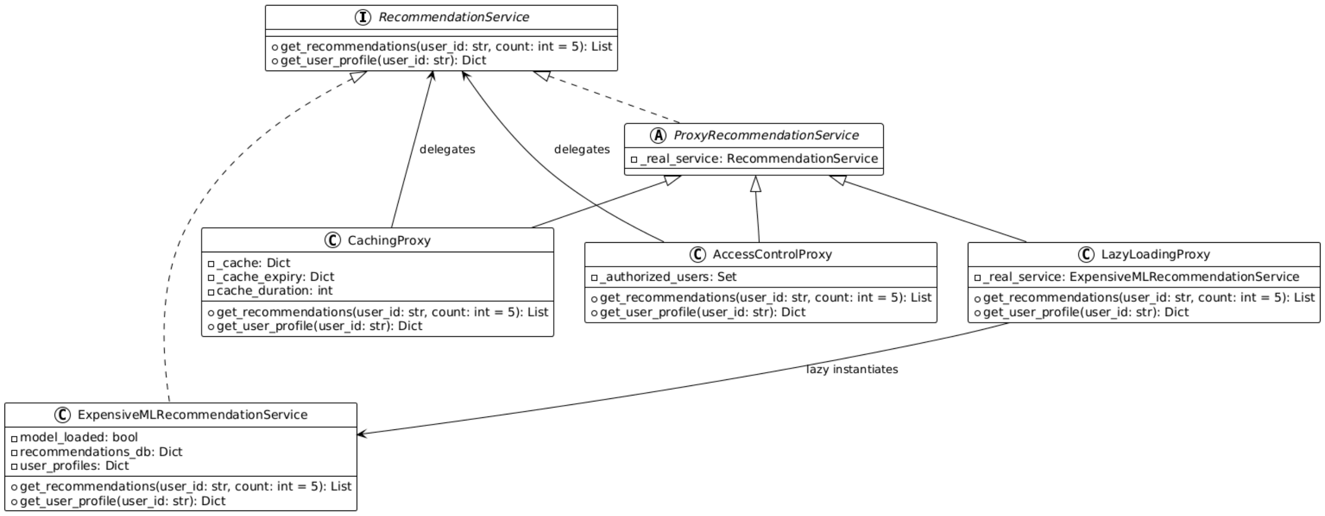
4.2.6 Proxy Pattern

Provides a surrogate or placeholder for another object to control access to it. Authors of the paper state on their related website (Heiland et al., 2023) that domain-specific usage of this pattern could be: "A substitute for the production database or service is needed. It should: 1) scale serving to multiple machines, but only provide one endpoint 2) reduce the needed amount of computational resources."

In PyTorch, proxy objects can be created; however, this should not be confused with the Proxy design pattern as it is a different, non-related concept which actually resembles more of a Builder pattern from Gang of Four design patterns by constructing step by step the layers:

```
a = torch.cat([y, z])  # Step 1: add concatenation
b = torch.tanh(a)      # Step 2: add tanh
c = torch.neg(b)       # Step 3: add negation
z = torch.add(b, c)    # Step 4: add addition
```

Proxy can, however, have usage whenever there is any security involved—recommendation systems also need to contain security handling; thus, the proxy can provide access control to recommendations as shown in the example in the repository (**TODO: CITE**). Similarly, it can facilitate caching or logging.


RecommendationService
get_recommendations(user_id: str, count: int = 5): List
get_user_profile(user_id: str): Dict
delegates
delegates
ProxyRecommendationService
_real_service: RecommendationService
CachingProxy
_cache: Dict
_cache_expiry: Dict
cache_duration: int
get_recommendations(user_id: str, count: int = 5): List
get_user_profile(user_id: str): Dict
AccessControlProxy
_authorized_users: Set
get_recommendations(user_id: str, count: int = 5): List
get_user_profile(user_id: str): Dict
LazyLoadingProxy
_real_service: ExpensiveMLRecommendationService
get_recommendations(user_id: str, count: int = 5): List
get_user_profile(user_id: str): Dict
lazy instantiates
ExpensiveMLRecommendationService
model_loaded: bool
recommendations_db: Dict
user_profiles: Dict
get_recommendations(user_id: str, count: int = 5): List
get_user_profile(user_id: str): Dict

4.3 Behavioral Patterns

4.3.1 Chain of Responsibility Pattern

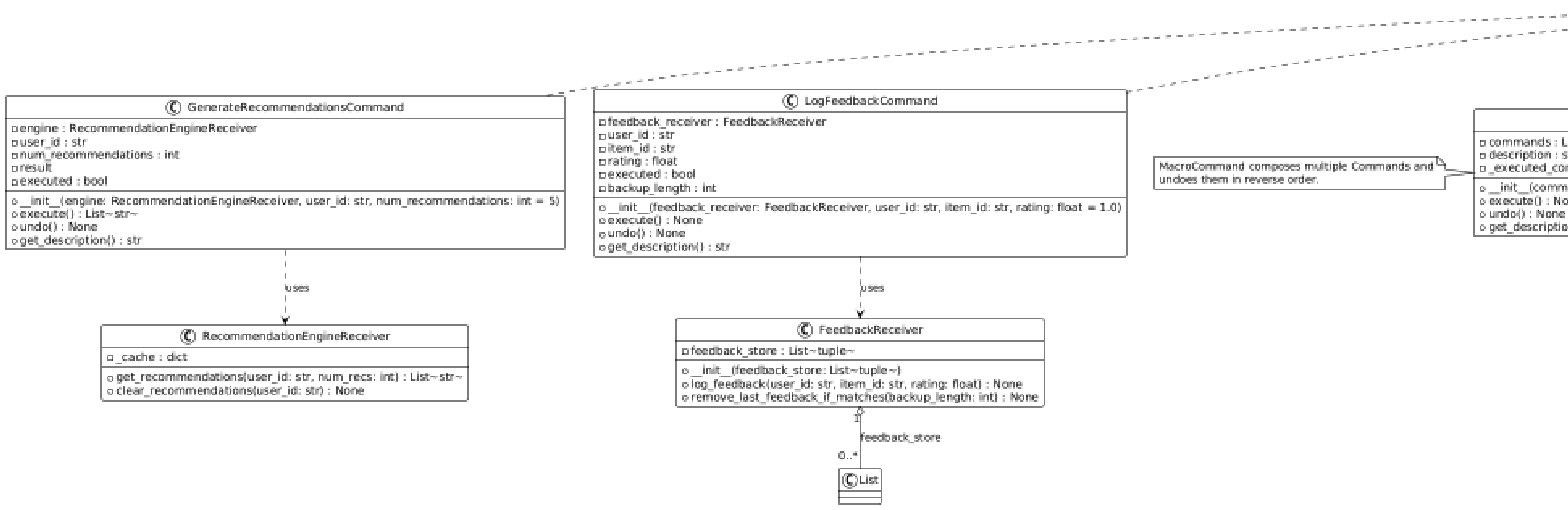
In this pattern, the request is passed in a chain of objects responsible for handling certain responsibilities. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain. There might be several usages of this pattern directly or indirectly in recommender systems. The Chain of Responsibility is often used in graphical user interfaces; thus, it can indirectly contribute to data recommender system performance by providing handlers for acquiring and processing user actions, thus more data. However, there is also an apparent possibility to use this pattern in preprocessing pipelines or post-processing pipelines—in a similar way to the Decorator pattern used for applying filters.

4.3.2 Command Pattern

In this pattern, the request is encapsulated as an object, thereby letting different requests queue or log requests, which could be also beneficial in many use cases in recommender systems.

The Command design pattern could be used to facilitate the recommendation itself but, even more importantly, used afterwards to log certain user actions and then perform necessary actions, such as retraining the model to gain advantage of newly discovered information about the user.

Command Pattern -


GenerateRecommendationsCommand
engine : RecommendationEngineReceiver
user_id : str
num_recommendations : int
result
executed : bool
__init__(engine: RecommendationEngineReceiver, user_id: str, num_recommendations: int = 5)
execute() : List~str~
undo() : None
get_description() : str
uses
RecommendationEngineReceiver
_cache : dict
get_recommendations(user_id: str, num_recs: int) : List~str~
clear_recommendations(user_id: str) : None
LogFeedbackCommand
feedback_receiver : FeedbackReceiver
user_id : str
item_id : str
rating : float
executed : bool
backup_length : int
__init__(feedback_receiver: FeedbackReceiver, user_id: str, item_id: str, rating: float = 1.0)
execute() : None
undo() : None
get_description() : str
uses
FeedbackReceiver
feedback_store : List~tuple~
__init__(feedback_store: List~tuple~)
log_feedback(user_id: str, item_id: str, rating: float) : None
remove_last_feedback_if_matches(backup_length: int) : None
1
feedback_store
0..*
List
MacroCommand composes multiple Commands and undoes them in reverse order.
commands : L
description : s
_executed_co
__init__(comm
execute() : No
undo() : None
get_descriptio

4.3.3 Iterator Pattern

Provides a way to access elements of an aggregate object sequentially without exposing its underlying representation. Iterator is often used in machine learning in supervised learning to iterate over individual examples. For example, it can be used for iterating over user interactions when training neural networks. The iterator itself can be named such as DataLoader. This object can prepare and output (mini)batches that would be fed into neural networks during training. In fact, the torch.utils.data.DataLoader in popular machine learning library PyTorch is an implementation of the Iterator pattern (technically it gives you an iterable object that returns an iterator) (PyTorch, 2024c).

4.3.4 Mediator Pattern

Defines an object that encapsulates how a set of objects interact, promoting loose coupling. "AI-based services are provided directly to downstream applications (e.g., e-commerce sites, trade orders), therefore coupling services and downstream applications tighter than needed" (Heiland et al., 2023).

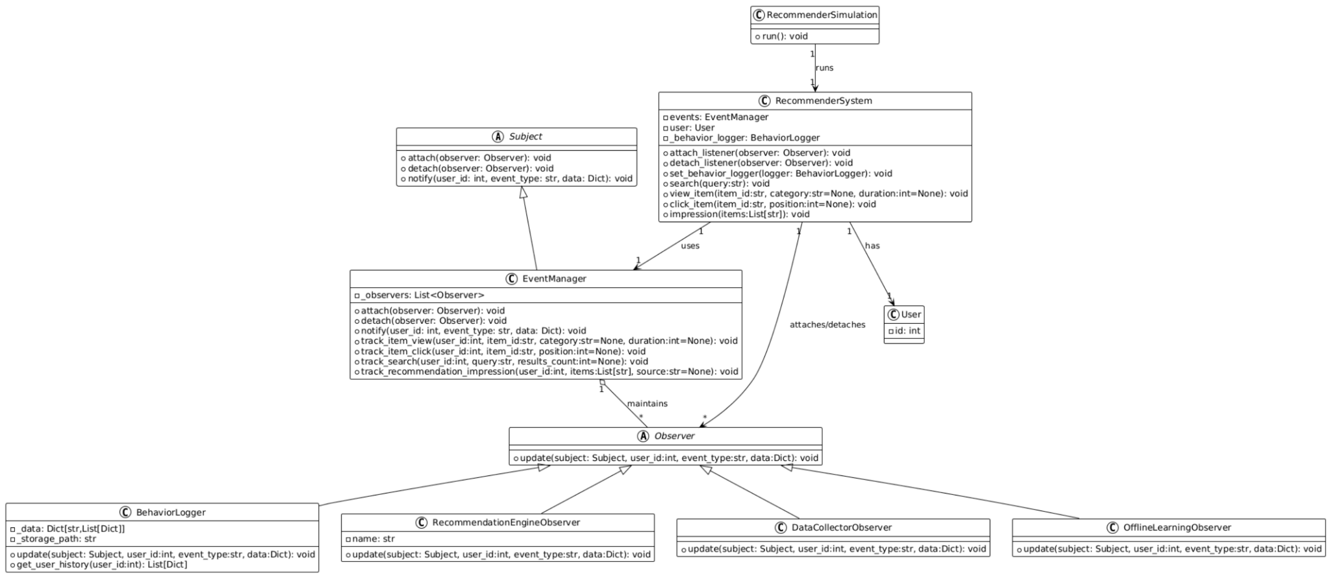
4.3.5 Memento Pattern

Captures and externalizes an object's internal state without violating encapsulation, so that the object can be restored to this state later. *The originator* could be the Model or training process. *Memento* is the Checkpoint file (weights, optimizer state, etc.). *Caretaker* is the Training manager that decides *when* to save and restore. Although not precisely (for example, it lacks greater separation of concerns compared to textbook examples), saving checkpoints using PyTorch library resembles the Memento design pattern and provides a similar solution.

TODO: [Code example for checkpoint creation and restoration]

4.3.6 Observer Pattern

Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. We propose that the Observer pattern can be used for facilitating execution of algorithms after user action. For example, user interacts with viewed item by clicking or rates it by thumbs up/down icon with button (or in some web applications it is denoted by the icon of heart or star), and the recommender system updates its knowledge, saves the new user action, and it may also update its models, execute computation of new recommendations based on performed user action, and prepare new results for the user.


RecommenderSimulation
run(): void
runs
RecommenderSystem
events: EventManager
user: User
_behavior_logger: BehaviorLogger
attach_listener(observer: Observer): void
detach_listener(observer: Observer): void
set_behavior_logger(logger: BehaviorLogger): void
search(query:str): void
view_item(item_id:str, category:str=None, duration:int=None): void
click_item(item_id:str, position:int=None): void
impression(items:List[str]): void
Subject
attach(observer: Observer): void
detach(observer: Observer): void
notify(user_id: int, event_type: str, data: Dict): void
uses
has
attaches/detaches
EventManager
_observers: List<Observer>
attach(observer: Observer): void
detach(observer: Observer): void
notify(user_id: int, event_type: str, data: Dict): void
track_item_view(user_id:int, item_id:str, category:str=None, duration:int=None): void
track_item_click(user_id:int, item_id:str, position:int=None): void
track_search(user_id:int, query:str, results_count:int=None): void
track_recommendation_impression(user_id:int, items:List[str], source:str=None): void
User
id: int
maintains
Observer
update(subject: Subject, user_id:int, event_type:str, data:Dict): void
BehaviorLogger
_data: Dict[str,List[Dict]]
_storage_path: str
update(subject: Subject, user_id:int, event_type:str, data:Dict): void
get_user_history(user_id:int): List[Dict]
RecommendationEngineObserver
name: str
update(subject: Subject, user_id:int, event_type:str, data:Dict): void
DataCollectorObserver
update(subject: Subject, user_id:int, event_type:str, data:Dict): void
OfflineLearningObserver
update(subject: Subject, user_id:int, event_type:str, data:Dict): void

4.3.7 State Pattern

Allows an object to alter its behavior when its internal state changes. This pattern was mentioned in a related website of the research team (Heiland et al., 2023), however no description was provided. In recommender systems, this can be used to represent the model's lifecycle management. A recommender system or ML model naturally moves through states: UntrainedState (model not yet trained, can't serve predictions), TrainingState (actively being trained, can log progress), TrainedState (ready to serve recommendations), ServingState (deployed and actively serving queries), DeprecatedState (old model, waiting to be archived or replaced).

We can also use this pattern, for example, to construct a class facilitating a neural network (either using a library or not) if we need different configurations of network for different states, e.g., we use configuration for a production state, evaluation state, or maybe even an adversarial state of neural network (Goodfellow et al., 2014) serving a similar function of a "safe mode" compared to terminology of operating systems if the neural network would be attacked.

[Code example for State pattern implementation provided in original document]

4.3.8 Strategy Pattern

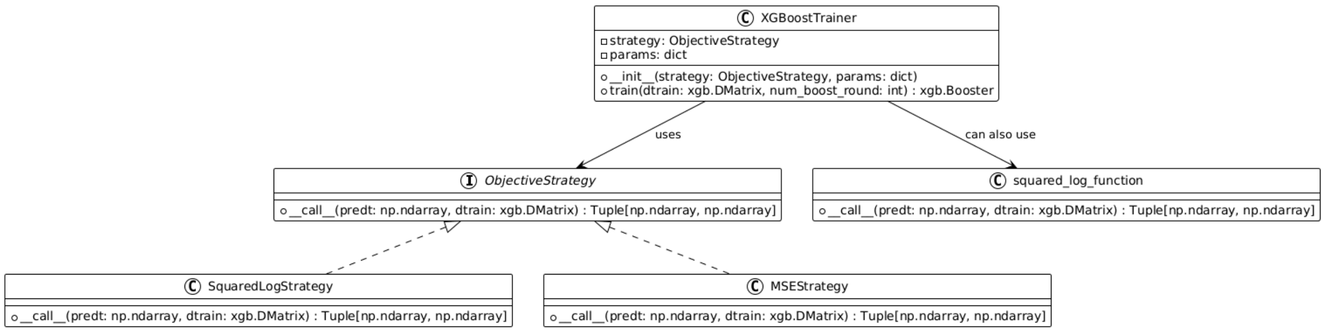
Defines a family of algorithms, encapsulates each one, and makes them interchangeable. The authors state the domain-specific usage is: "How can an ML model that performs a task in a given context be flexibly changed?" (Heiland et al., 2023). The solution is to define an interface (strategy) that different models implement.

This is also connected to different methods that are often switched in recommendation systems to provide better recommendations based on state of the system or amount of user data to provide more accurate recommendations and to avoid cold-start problems (the inability to recommend with user-based methods by collaborative methods if not enough data about user behavior is collected).

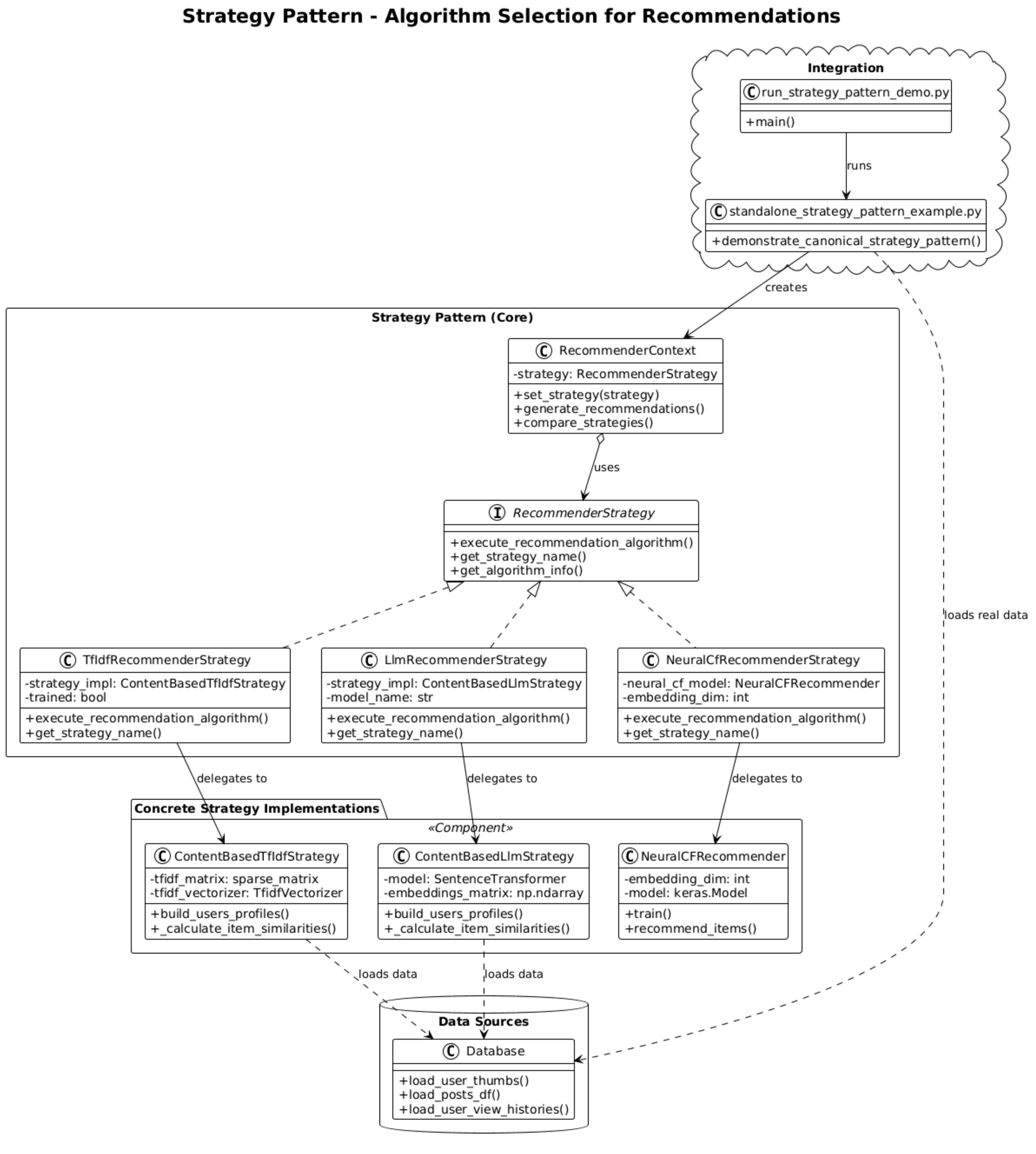
The context then calls methods exposed by the interface, and implemented models will behave differently based on contextual data. This should achieve higher flexibility with the downside of increased complexity of code. As an example of practical usage, they mentioned XGBoost's custom objective functions or HuggingFace's pipeline interface (Heiland et al., 2023).

In summary the benefits this design pattern can add are:

- Provides the ability of runtime algorithm switching.
- New strategies should be easier to integrate.
- Can encapsulate families of methods and algorithms.
- Client code can stay independent of the actual implementations of the methods and the algorithms.



XGBoostTrainer
strategy: ObjectiveStrategy
params: dict
__init__(strategy: ObjectiveStrategy, params: dict)
train(dtrain: xgb.DMatrix, num_boost_round: int) : xgb.Booster
uses
can also use
ObjectiveStrategy
__call__(predt: np.ndarray, dtrain: xgb.DMatrix) : Tuple[np.ndarray, np.ndarray]
squared_log_function
__call__(predt: np.ndarray, dtrain: xgb.DMatrix) : Tuple[np.ndarray, np.ndarray]
SquaredLogStrategy
__call__(predt: np.ndarray, dtrain: xgb.DMatrix) : Tuple[np.ndarray, np.ndarray]
MSEStrategy
__call__(predt: np.ndarray, dtrain: xgb.DMatrix) : Tuple[np.ndarray, np.ndarray]


Strategy Pattern - Algorithm Selection for Recommendations
Integration
run_strategy_pattern_demo.py
+main()
runs
standalone_strategy_pattern_example.py
+demonstrate_canonical_strategy_pattern()
creates
loads real data
Strategy Pattern (Core)
RecommenderContext
-strategy: RecommenderStrategy
+set_strategy(strategy)
+generate_recommendations()
+compare_strategies()
uses
RecommenderStrategy
+execute_recommendation_algorithm()
+get_strategy_name()
+get_algorithm_info()
TfIdfRecommenderStrategy
-strategy_impl: ContentBasedTfIdfStrategy
-trained: bool
+execute_recommendation_algorithm()
+get_strategy_name()
LlmRecommenderStrategy
-strategy_impl: ContentBasedLlmStrategy
-model_name: str
+execute_recommendation_algorithm()
+get_strategy_name()
NeuralCfRecommenderStrategy
-neural_cf_model: NeuralCFRecommender
-embedding_dim: int
+execute_recommendation_algorithm()
+get_strategy_name()
delegates to
delegates to
delegates to
Concrete Strategy Implementations
«Component»
ContentBasedTfIdfStrategy
-tfidf_matrix: sparse_matrix
-tfidf_vectorizer: TfidfVectorizer
+build_users_profiles()
+_calculate_item_similarities()
ContentBasedLlmStrategy
-model: SentenceTransformer
-embeddings_matrix: np.ndarray
+build_users_profiles()
+_calculate_item_similarities()
NeuralCFRecommender
-embedding_dim: int
-model: keras.Model
+train()
+recommend_items()
loads data
loads data
Data Sources
Database
+load_user_thumbs()
+load_posts_df()
+load_user_view_histories()

4.3.9 Template Method Pattern

Defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. While this pattern might remind some resemblances with the Builder design pattern, this method is focused on steps rather than object creation like the Builder pattern. The Template pattern can be used to define common steps in computing recommendations in the abstract base class, e.g., loading content, loading user-related data (e.g., their past interactions), data preprocessing, etc., until final steps such as ranking are done. These common steps can be performed by different types of recommendation methods (e.g., content-based vs. collaborative vs. hybrid), and different types of methods might choose to implement different methods specifically to their needs (e.g., content-based method would use content loading method rather than loading of user interactions). This can provide a flexibly extendable, clean, and well-readable interface code for incorporating new methods.

In the recommendation system thus, each concrete template thus customizes:

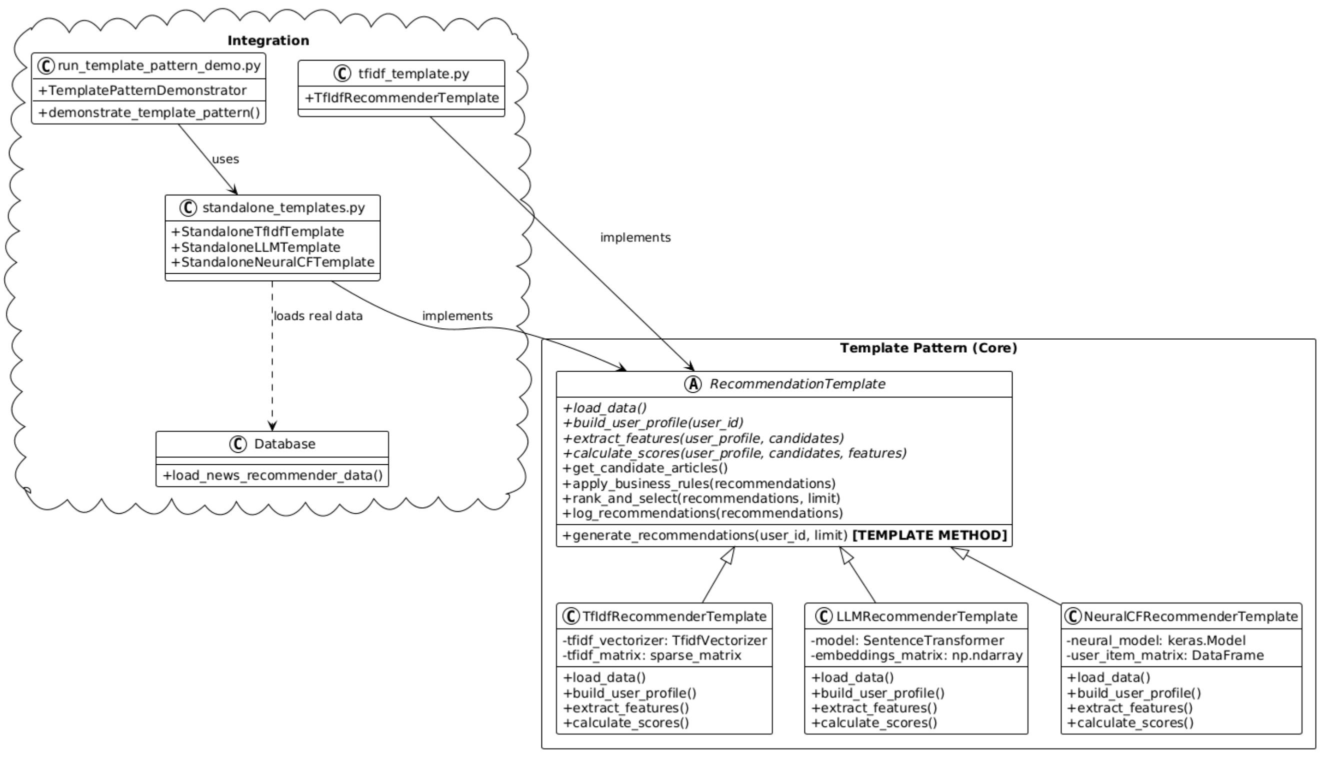
- How data is loaded.
- How user profiles are built.
- How features are extracted.
- How scores are calculated.

The existing example that resembles the Template Method pattern is, for example, using TextCorpus of natural language processing (NLP) library Gensim (Řehůřek & Sojka, 2010)—an abstract base class which can be used for overriding the get_texts() and `__len__()` to provide particular input of need, and the load() method is used for reading and processing a document and splitting it to sequence of words.

Template method benefits in summary are:

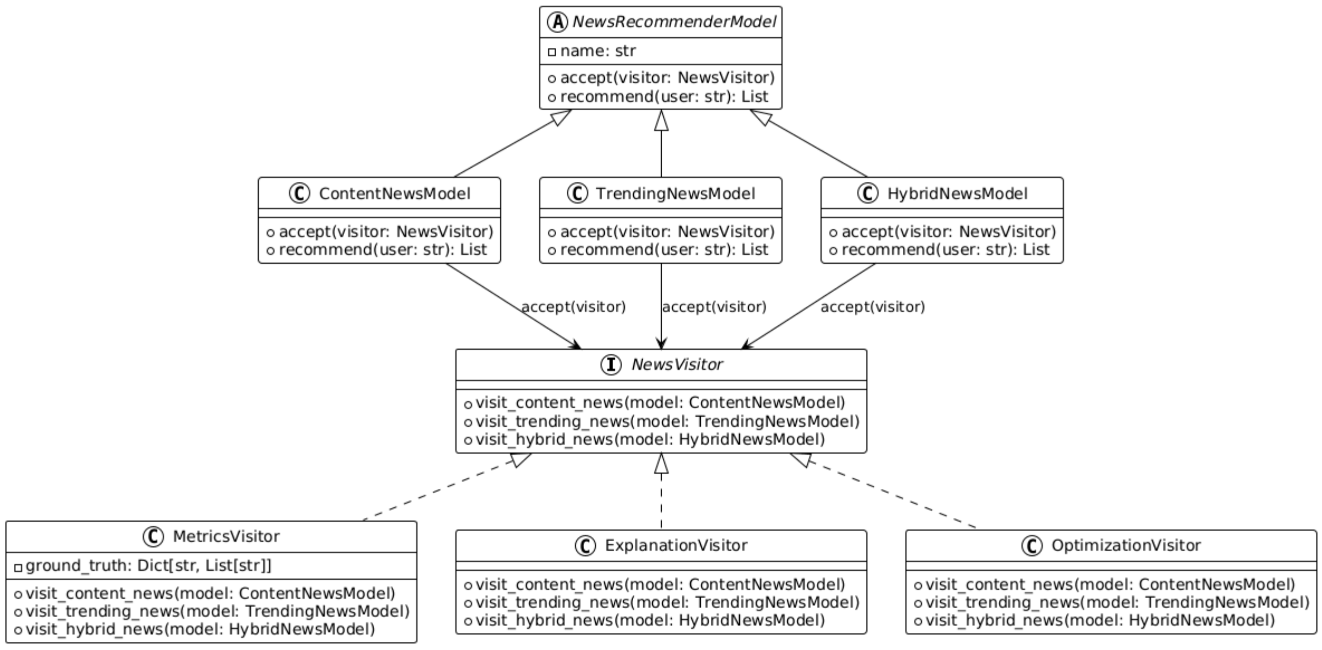
- Defining the skeleton of the algorithm, thus creating an abstraction of the algorithm.
- Common steps can be reused without repetition (following the "Don't repeat yourself", or DRY principle (**TODO:** CITE: The Pragmatic Programmer).
- We can use a consistent pipeline across different algorithms.

Template Method Pattern - Recommendation Pipeline


Integration
run_template_pattern_demo.py
+TemplatePatternDemonstrator
+demonstrate_template_pattern()
tfidf_template.py
+TfIdfRecommenderTemplate
uses
standalone_templates.py
+StandaloneTfIdfTemplate
+StandaloneLLMTemplate
+StandaloneNeuralCFTemplate
implements
loads real data
implements
Database
+load_news_recommender_data()
Template Pattern (Core)
RecommendationTemplate
+load_data()
+build_user_profile(user_id)
+extract_features(user_profile, candidates)
+calculate_scores(user_profile, candidates, features)
+get_candidate_articles()
+apply_business_rules(recommendations)
+rank_and_select(recommendations, limit)
+log_recommendations(recommendations)
+generate_recommendations(user_id, limit) [TEMPLATE METHOD]
TfIdfRecommenderTemplate
-tfidf_vectorizer: TfidfVectorizer
-tfidf_matrix: sparse_matrix
+load_data()
+build_user_profile()
+extract_features()
+calculate_scores()
LLMRecommenderTemplate
-model: SentenceTransformer
-embeddings_matrix: np.ndarray
+load_data()
+build_user_profile()
+extract_features()
+calculate_scores()
NeuralCFRecommenderTemplate
-neural_model: keras.Model
-user_item_matrix: DataFrame
+load_data()
+build_user_profile()
+extract_features()
+calculate_scores()

4.3.10 Visitor Pattern

Represents an operation to be performed on elements of an object structure. It lets you define a new operation without changing classes of elements on which it operates. Visitor pattern can be used in computation of metrics. For example, CollaborativeFiltering might require different metrics from ContentBased and Hybrid methods. We might define the `visitCollaborative()` method with calls of `calculatePrecision(model)` and `calculateRecall(model)`, but the content-based model might also have some diversity method like `calculateDiversity()`.


NewsRecommenderModel
name: str
accept(visitor: NewsVisitor)
recommend(user: str): List
ContentNewsModel
TrendingNewsModel
HybridNewsModel
accept(visitor: NewsVisitor)
recommend(user: str): List
accept(visitor)
NewsVisitor
visit_content_news(model: ContentNewsModel)
visit_trending_news(model: TrendingNewsModel)
visit_hybrid_news(model: HybridNewsModel)
MetricsVisitor
ground_truth: Dict[str, List[str]]
ExplanationVisitor
OptimizationVisitor

4.4 Patterns Left Out

We left out the Singleton pattern since this is a pattern commonly used for database connection and it has this usage also in recommendation systems when working with any database; however, we haven't found other special use cases.

Also, in addition to the Strategy design pattern, we need to point out another use case might emerge from graphical user interface. Recommender systems are often large decoupled systems consisting of many parts. GUIs often use the Chain of Responsibility design pattern. A rich user interface is needed in modern applications, and this front-end side of the application can then send requests to other parts of the system.

TODO:

- Read and mention: https://applyingml.com/resources/patterns/
- Check the examples if they are good examples according to textbook principles or not violating anything major from the pattern definition

5. Empirical Measurements

Because usage of a certain pattern for a particular example is highly opinion-based and subjective, we also used an objective metric—time of execution of recommendation computation using different GoF design patterns. In the field of machine learning, low execution time is always a highly important characteristic of the system since shorter execution time can provide shorter feedback loop for evaluation of precision of methods, thus potentially providing more opportunities to modify algorithms for better final results, thus economical advantages of the project. We also tested memory consumption since some patterns (e.g., Flyweight) directly aim to solve high memory consumption and obviously also provide economical benefits.

We should emphasize that the major goal of design patterns (at least most GoF design patterns) is usually not to provide improved time execution or less memory consumption but rather provide solutions to common problems through following SOLID principles (see above). Implementation of design patterns might bring even negative consequences to time and space complexity and even worsen performance; however, for the sake of better code maintainability, extendability, reusability, and other advantages implied by SOLID principles. Nevertheless, we believed it would be interesting to see whether avoidance of "spaghetti code" can lead to improvements not only in aspect of clean code but also in aspect of time of execution and memory consumption, which is always valuable in this domain where time and space are bottlenecks of many algorithms.

One exception of design patterns that is well known for actually improving memory consumption is Singleton, which by its nature restricts creation of multiple instances of the class and rather holds just one instance. On the other hand, an Observer might worsen time and space complexity since it requires notifications to subscribers (observers) — thus possibly adding complexity due to iterating over a list of subscribers and also consumes more memory (Nystrom, 2014).

5.1 Experimental Setup

Since we did not want to artificially implement use cases to any design pattern that would not fit the use case, we rather implemented and compared only those design patterns that were truly fit to being a solution specifically to the problem of computing and ensembling various different recommendation methods, e.g., hybrid, collaborative, etc., which emerged repeatedly as a good fit for many design pattern usages, and we implemented this to our recommender system for Czech news following our previous work (**TODO: CITE**).

Another metric used was cognitive load. **TODO:** [Complete this section]

The GoF patterns fitting this use case are:

- The Bridge pattern combined with the Factory pattern for initializing objects of different methods
- The Builder pattern
- The Strategy pattern
- The Template pattern
- ~~(The Flyweight pattern)~~

5.2 Results

TODO: [Add experimental results, performance comparisons, and statistical analysis]

TODO: Determine what metrics to choose. Possible options:

Actual measurement of time and memory consumption of execution on 1 or mean on more devices.

Lines of Code (LOC / NLOC) — overall size change.

Number of classes, interfaces, methods — reveals modularization.

Cyclomatic Complexity (per method / total) — decision complexity.

Cognitive Complexity — how hard it is to understand the code (IDE plugins, Sonar).

Maintainability Index (MI) — composite metric (Halstead + Cyclomatic + LOC).

Weighted Methods per Class (WMC), Response For Class (RFC), Coupling Between Objects (CBO) — OO metrics; patterns should reduce CBO and WMC in many cases.LCOM (cohesion) — higher cohesion is better; patterns often increase cohesion.

Number of code smells / rule violations — from SonarQube/PMD/ESLint.

Duplication (%) — patterns often remove duplication.API surface (public methods/types) — indicates exposure/complexity for clients.

Package/module dependency graph / number of cycles — fewer cycles is better. Tools: jdeps, depcheck, Sonar or ArchUnit.

6. Discussion

TODO: [Add discussion of findings, limitations, and implications]

6.1 Practical Implications

TODO: [Discuss practical implications for ML practitioners]

6.2 Theoretical Contributions

TODO: [Discuss contributions to software engineering and ML theory]

6.3 Limitations

In the new era of AI coding assistance, it is not known yet how much the cognitive complexity analysers made initially for estimating the complexity for the human programmers, correlates with the complexity for the AI coding coding assistance (whether the AI coding assistants "finds difficulty" in the code in similar ways to the humans).

TODO: [Discuss limitations of the study]

7. Conclusion

This paper presented a comprehensive analysis of how Gang of Four design patterns can be effectively applied to machine learning systems, with particular focus on recommender systems. We identified 18 applicable patterns from the original 23 GoF patterns and provided concrete examples of their implementation in real-world recommendation scenarios.

Our main contributions include:

1. Systematic mapping of GoF patterns to ML and recommender system use cases
2. Concrete implementation examples demonstrating practical applicability
3. Empirical evaluation of selected patterns in terms of time and space complexity
4. Identification of patterns particularly suited for addressing common ML challenges such as cold-start problems, model lifecycle management, and algorithm switching

While design patterns are primarily intended to improve code maintainability, extensibility, and adherence to SOLID principles rather than performance, our empirical measurements suggest that well-applied patterns can also contribute to computational efficiency in certain scenarios.

Future work should extend this analysis to other ML domains beyond recommender systems, investigate pattern combinations for complex ML pipelines, and develop domain-specific patterns that build upon the GoF foundation while addressing unique challenges in AI systems.

Acknowledgments

TODO: [Add acknowledgments and funding information]

References

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). Manifesto for Agile Software Development. Retrieved from http://agilemanifesto.org

Crevier, D. (1993). *AI: The tumultuous history of the search for artificial intelligence*. Basic Books.

Dahl, O. J., & Nygaard, K. (1966). SIMULA: An ALGOL-based simulation language. *Communications of the ACM*, 9(9), 671–678.

Dijkstra, E. W. (1972). The humble programmer. *Communications of the ACM*, 15(10), 859–866.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.

Gloo. (2024). *Gloo: Collective communication library*. GitHub. Retrieved from https://github.com/pytorch/gloo

Goodfellow, I. J., Shlens, J., & Szegedy, C. (2014). Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*.

Heiland, R., Ebert, F., & Brügmann, K. (2023). Design patterns for AI-based systems: A systematic literature review. *arXiv preprint arXiv:2303.13173*. Retrieved from https://arxiv.org/pdf/2303.13173

HuggingFace. (2024). *Auto classes documentation*. Retrieved from https://huggingface.co/docs/transformers/en/models?model-classes=AutoModel

JetBrains. (2021). *State of Developer Ecosystem 2021: ML and Data Science*. Retrieved from https://www.jetbrains.com/lp/devecosystem-2021/

Kaggle. (2019). *2019 Kaggle Machine Learning and Data Science Survey*. Retrieved from https://www.kaggle.com/kaggle-survey-2019

Kay, A. C. (1993). The early history of Smalltalk. *ACM SIGPLAN Notices*, 28(3), 69–95.

Keras Team. (2024). *Keras Sequential model*. Retrieved from https://keras.io/guides/sequential_model/

Lakshmanan, V., Robinson, S., & Munn, M. (2020). *Machine learning design patterns: Solutions to common challenges in data preparation, model building, and MLOps*. O'Reilly Media.

McCarthy, J. (1973). The philosophy of artificial intelligence. In M. A. Boden (Ed.), *The philosophy of artificial intelligence* (pp. 1–13). Oxford University Press.

NCCL. (2024). *NVIDIA Collective Communication Library*. GitHub. Retrieved from https://github.com/NVIDIA/nccl

Naur, P., & Randell, B. (Eds.). (1969). *Software engineering: Report on a conference sponsored by the NATO Science Committee*. NATO Scientific Affairs Division.

Nystrom, R. (2014). *Game programming patterns*. Genever Benning. Retrieved from https://gameprogrammingpatterns.com/observer.html

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.

PyTorch. (2024a). *Distributed communication package*. Retrieved from https://docs.pytorch.org/docs/stable/distributed.html

PyTorch. (2024b). *Sequential container*. Retrieved from https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html

PyTorch. (2024c). *DataLoader documentation*. Retrieved from https://pytorch.org/docs/stable/data.html

Řehůřek, R., & Sojka, P. (2010). Software framework for topic modelling with large corpora. In *Proceedings of the LREC 2010 workshop on new challenges for NLP frameworks* (pp. 45–50).

Schank, R. C., & Minsky, M. L. (1984). Warnings about artificial intelligence. In AAAI (Vol. 84, pp. 5–7).

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., ... & Dennison, D. (2015). Hidden technical debt in machine learning systems. *Advances in Neural Information Processing Systems*, 28, 2503–2511.

Shvets, A. (2024). *Prototype pattern in Python*. Refactoring.Guru. Retrieved from https://refactoring.guru/design-patterns/prototype/python/example

Umbrello, S. (2021). AI winter. In P. Frana & M. J. Klein (Eds.), *Encyclopedia of artificial intelligence: The past, present, and future of AI*. Springer.

Yan, E. (2021). *Design patterns in machine learning code and systems*. Retrieved from https://eugeneyan.com/writing/design-patterns/

TODO: Complete all references marked with (CITE) throughout the document