Registries in Machine Learning-Based Drug Discovery: A Shortcut to Code Reuse

Citation for the original published paper (version of record):
Hartog, P., Svensson, E., Mervin, L. et al (2025). Registries in Machine Learning-Based Drug Discovery: A Shortcut to Code Reuse. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 14894 LNCS: 98-115.
http://dx.doi.org/10.1007/978-3-031-72381-0_9

(article starts on next page)

Registries in Machine Learning-Based Drug Discovery: A Shortcut to Code Reuse

Peter B. R. Hartog[1,3](✉), Emma Svensson[2,3](✉), Lewis Mervin[4], Samuel Genheden[3], Ola Engkvist[3,5], and Igor V. Tetko[1]

[1] Institute of Structural Biology, Molecular Targets and Therapeutics Center, Helmholtz Munich-Deutsches Forschungszentrum für Gesundheit und Umwelt (GmbH), 58764 Neuherberg, Germany

[2] ELLIS Unit Linz, Institute for Machine Learning, Johannes Kepler University Linz, 4040 Linz, Austria

[3] Molecular AI, Discovery Sciences, R&D, AstraZeneca Gothenburg, 431 83 Gothenburg, Sweden
{peter.hartog,emma.svensson1}@astrazeneca.com

[4] Molecular AI, Discovery Sciences, R&D, AstraZeneca Cambridge, Cambridge CB2 0AA, UK

[5] Department of Computer Science and Engineering, Chalmers University of Technology, 412 96 Gothenburg, Sweden

Abstract. Computer-aided drug discovery gradually builds on previous work and requires reusable code to advance research. Currently, research code is mainly used to provide further insights into the original research whilst code reuse has a lower priority. Modularity, the segmentation of code for independent modules, promotes good coding practices and code reuse. The registry pattern has been proposed as a way to call functionalities dynamically, but it is currently overlooked as a shortcut to promote code reuse. In this work, we expand the registry pattern to better suit computer-aided drug discovery and achieve a unified, reusable, and interchangeable interface with optional meta information. Our reformulated pattern is particularly suitable for collaborative research with standardized frameworks where multiple internal and external modules are used interchangeably and coding is more focused on fast iteration over low-debt technical code, such as in machine learning-based research for drug discovery. In a workflow, we exemplify the usage of the design patterns. Additionally, we provide two case studies where we 1) showcase the effectiveness of registration in a larger collaborative research group, and 2) overview the potential of registration in currently available open-source tools. Finally, we empirically evaluate the registry pattern through previous implementations and indicate where additional functionality can improve its use.

Keywords: registration · design pattern · modularity · code reuse · drug discovery · machine learning

D.-A. Clevert et al. (Eds.): AIDD 2024, LNCS 14894, pp. 98–115, 2025.
https://doi.org/10.1007/978-3-031-72381-0_9

1 Introduction

The development of computer-aided drug discovery relies on previous research from multiple fields to bridge the knowledge gap between domain experts and computer scientists [41]. As such, software development in the field is often built up of a combination of open-source tools, collaborative developments, and independent research. Currently, research code is mainly used to provide further insights into the original research rather than to use in future research [6]. There is also a reproducibility problem [3] of code that stems from the low priority of code reuse, as noted by Nature Computational Science ("But is the code (re)usable?" [Editorial]. 2021, 23 July). Benureau and Rougier [5] proposed that research code should adhere to particular requirements for stable and reliable results. Code should be replicable, to obtain the same results as the original paper; be able to run without problems; repeatable (i.e., deterministic); reproducible (i.e., deterministic over multiple runs); and, finally, reusable. However, research code is different from production software because its goals are focused mostly on replication, where reuse is often an afterthought.

Code reuse is dependent on the concept of modularity [4]. Modular code is code that is grouped with related code and mostly independent of other parts of the code, named high cohesion and low coupling, respectively [33,46]. This results in code that is interchangeable, replaceable, and can be updated and used without issue [23,25,39]. Modular code also avoids the need to repeat code segments [17] and forces code modules to achieve single objectives over multiple responsibilities [26].

Machine learning (ML) workflows are inherently modular (Fig. 1). The workflow of ML is usually segmented into separate steps, such as data generation and model creation, regardless of implementation. ML approaches are becoming more prominent for research in drug discovery [7,11,42]. Computational research, like all scientific research, builds on the knowledge from previous discoveries and utilizes known methods and coding frameworks to create new tools, apply methods to new fields, or investigate new problems. ML researchers use the available tools to compare their novel models to previous approaches [9,22,45] and to streamline their pipeline [14,47]. There are also tools designed explicitly for ML in drug discovery, some of which focus on aiding new practitioners in quickly finding and comparing state-of-the-art approaches [10,24,28,34,40,44]. Other similar tools can function to bridge the knowledge gap between natural scientists and computer scientists. The latter can be achieved either by supplying domain-specific knowledge to ML workflows [16,21,27,31] or by making common ML frameworks [1,32] more accessible through higher-level abstractions [8,15,43]. Although tools are created to be used by others, and therefore surpass research code in reusability, tools are often semi-rigid, made to work out of the box for a fixed domain, purpose, or type of method. This means there is often a high bar to add new functionalities in open-source tools [4].

In this work, we identify registries [12] as a shortcut to code reuse for ML-based drug discovery. We introduce the registry design pattern to those unfamiliar and suggest additional capabilities. Furthermore, we identify situations where

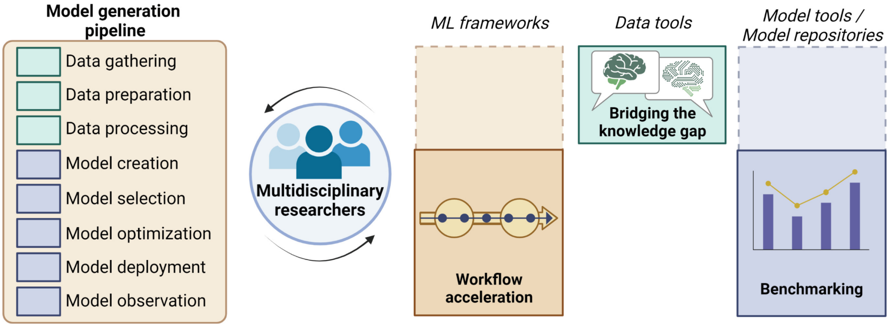

Fig. 1. Overview of model generation pipeline and applicability of tools for each step. Repository tools are primarily used for reproducibility and to benchmark new approaches against the existing state-of-the-art. **Data tools** bridge the knowledge gap between domain experts and ML practitioners. Finally, **ML frameworks** add workflow abstractions that accelerate the ML pipeline. Multidisciplinary collaborations rely on the use of a combination of tools from these categories, as illustrated in the leftmost panel.

the cheminformatics community can benefit from registries to easily make their code more reusable as well as more replicable [5]. We provide an open-source implementation of the generic registry design pattern through the Python Package Index. While the fast pace of drug discovery research can result in bad coding practices, the simplicity of our proposed registry is meant to encourage improved coding practices with minimal effort for the researchers. The use of registries during codebase design can additionally serve as an easy way to enforce desired behaviors, such as a factory pattern or test adherence, which in turn helps contributors adhere to the desired coding standards. Finally, we identify where previous implementations of registries have succeeded and failed, and discuss the reasons behind successful implementations in both research code and software tools. Our contributions are summarized as follows.

- We extend previous explanations of the registry with capabilities for use by researchers.
- We outline several use cases of where and when registries can be a shortcut for reuse in computer-aided drug discovery research.
- We identify previous implementations of registries and note their positive and negative implications.

2 Methods

The registry design pattern has been proposed as a tool for dynamic instance creation of object-oriented classes [12]. Best practices in object-oriented programming are often formalized as design patterns. A software design pattern provides a template for a general and reusable way that solves a recurring problem in software engineering [13]. We reformulate the registry as a method to retrieve similar modules with similar functionalities or uses. Consider a set of modules $\mathcal{M}$ supplying the same type of functionality but with different implications to a workflow. Given that the modules follow the Liskov substitution principle [23], i.e. that $f : \mathcal{X} \to \mathcal{Y}, \forall f \in \mathcal{M}$ where $\mathcal{X}$ and $\mathcal{Y}$ are the set of inputs and outputs respectively, it is commonly known that they can be used interchangeably through inheritance. A problem with inheritance is that each module still has to be initialized individually. The standard design pattern to interchange such modules is the factory pattern (Fig. 2 left). However, this approach requires strict inheritance from an abstract class.

The registry design pattern uses call and set functions, usually renamed to get and register, to dynamically set and retrieve objects from a unified storage location. Registry systems are often initialized at run-time and used in combination with alias-based retrieval. As such, a registry follows the factory design pattern in that it provides a common interface for categorically similar functionalities without the explicit need for concrete classes. Furthermore, the registry encapsulates each alternative module, hiding individual details behind a unified set of function calls.

Formally, we reformulate the existing registry design pattern [12] as a means to collect interchangeable modules with encapsulated functionalities retrievable using a unified command. Additionally, the registry pattern is dynamic in its application. We provide our proposed design pattern to the cheminformatics community through the Python Package Index as the registry-factory package under the MIT license [29], which specifies the implementation of a collection of registries. The open-source code is available at https://github.com/aidd-msca/registry-factory.

3 Results

3.1 Workflow: Creating a Registry and Registering Modules

An overview of the process of creating a registry is illustrated in Fig. 3. A new registry is either imported from the package or is instantiated from a factory class. No instance of a registry needs to be created to use it. First, a section of code is separated from the framework. This is then converted into a function or class and registered into the registry. All subsequent scripts and external collaborators are then able to call upon the registry for this code.

When a registry is outlined as above it promotes and allows specific actions to be performed more fluently: 1) A registry provides a framework to exchange modules with similar functions. This switch is also stable in execution and flexible in application, depending on how strictly the registry is set up. 2) The registry setup allows control over how modules should be set up. More strict setups will force standardized modularity and interfaces, whilst more flexible setups allow faster extension and broader application. 3) Due to the standardization, more internal and external modules can use the same execution framework to function.

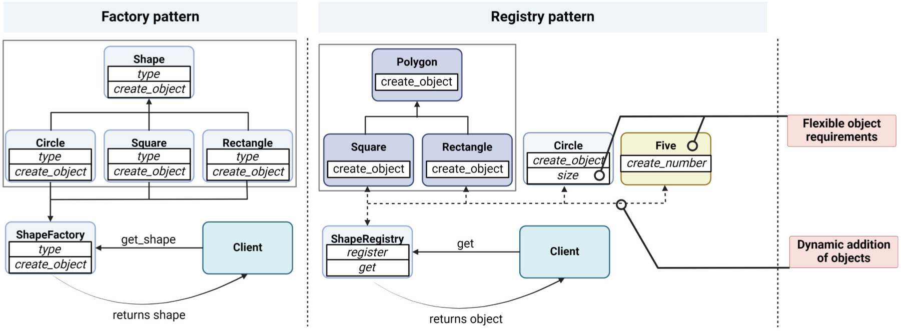

Fig. 2. Unified Modeling Language (UML) diagram of the factory design pattern and our proposed Registry design pattern. Left: UML of the factory pattern. The ShapeFactory functions as an interface where all subclasses of the Shape superclass can be called by the client. **Right:** UML of the registry pattern. The ShapeRegistry functions as an interface where any object no matter their superclass can be registered dynamically and called by the client.

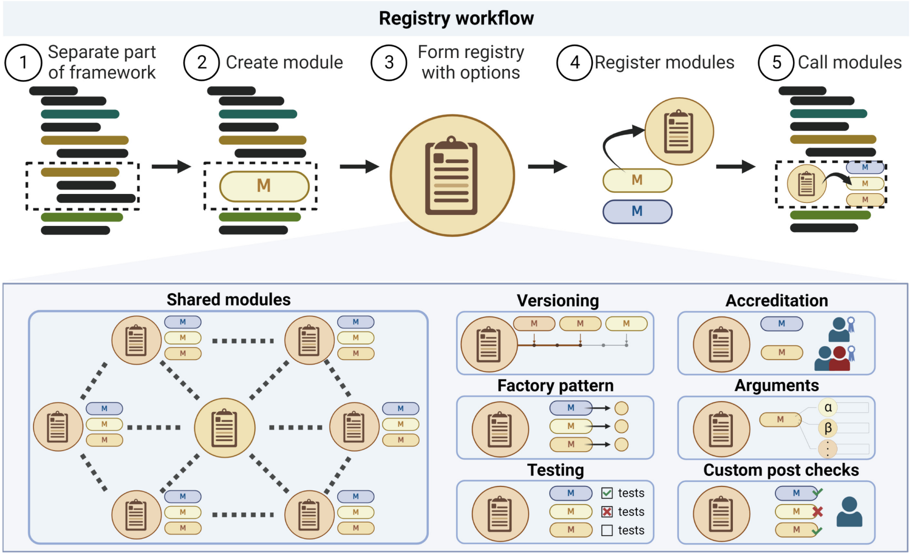

Fig. 3. Workflow of creating and using a registry. 1) Part of the code framework is identified which can be separated from the rest. 2) This section is modularized to be independent of the rest of the code. 3) A registry is created. Here, the choice of additional meta information such as versioning, accreditation, and arguments are set as well as the choice to share modules and force a specific class pattern. There is also the option to add post-registration checks, both custom and those that reference a testing script. 4) New modules are registered to the registry. 5) Finally, the modules are called in the main workflow using the registry.

3.2 Enhanced Functionality and Expanded Capabilities

Minimal usage of registries can be limited in their application. As a result, additional capabilities can increase the application options of registries, even in more advanced architecture designs. In this paper, we advocate for a non-exhaustive selection of additional capabilities and have implemented them into a separate package that will be made available upon acceptance (Fig. 3).

Upon creation, the specifications of the registry are instantiated and a selection of additional capabilities can be included. Firstly, to increase ease of use, inter-registry module sharing and argument registration are implemented. Secondly, the following features are implemented to allow for better control in a codebase setting, factory pattern forcing, versioning, and automatic testing upon registration. Finally, to accommodate the research community when contributing to packages, we have implemented accreditation which can be retrieved when calling registered objects.

3.3 A Shortcut to Modularity and Reusability

The main reason for the integration of registries is the passive enforcement of modularity. Experienced programmers will inherently shift to modular code and separate classes and functions into coherent modules and packages. However, research code and practitioners from other fields in the cheminformatics community will often focus on fast iteration over low-debt technical code. As such, the use of registries aims to passively enforce the usability of sections that the researcher will need repeatedly without the need for that code to be of high standard. Similarly, registries allow researchers to share code more easily, both internally during a project and externally after the research has been published. Contrary to common Python principles the use of registries gives a more flexible way of sharing code such that certain parts can be reused while others are updated or changed entirely. For example, this can be achieved by using the registries as hooks, allowing researchers to add functionalities without altering the code.

Case Study: Codebase Design and Collaboration. Registries are ideal for collaborative work, as they signify which part of the code is reusable to collaborators. The field of cheminformatics is multidisciplinary by definition and researchers often work collaboratively or using shared codebases. In these settings, low-quality code can prohibit collaboration.

By utilizing registries in collaborative work, researchers can specify beforehand what parts of code can be easily shared between collaborators. It also stops collaborators from having to dive into messy code and instead be able to just extract the segments of interest.

Furthermore, codebase designers and maintainers can use registries together with our added capabilities to automatically check and control suggested code submissions. Using automatic testing, registered modules are submitted to testing upon entering the registry, whereas custom controls can enforce desired behaviors, such as consistent input variables. This approach not only eliminates redundancy but also enhances code readability and maintainability.

Case Study: Registries in Cheminformatics Tools. The usage of cheminformatics tools can also benefit from registries. Table 1 gives a non-exhaustive list of available tools often used in ML for drug discovery. These tools are essential in their respective domains but they can be difficult to combine or use interchangeably. Many of them are internally built in a modular way but less so developed to be adjusted by the users. The registry design pattern can be used on different levels together with these tools to create adaptability and interchangeability, which in turn allows for code reuse.

Firstly, including registration can allow the individual tools to open up the possibility for users to contribute with their own functionalities or include other open-source packages. Users can test out new functionalities directly in the tool environment using registries, without the need to download and add to the package code. Model repository tools can benefit from registries by allowing users to register additional models in the collection. One can imagine entire libraries of models and data collections being allowed in ML pipeline tools or workflow systems, such that different collections can be used interchangeably.

Additionally, tools can open up internal capabilities using registries. Oftentimes, modules from cheminformatics tools are built with a specific functionality in mind. However, most modules contain multiple useful functionalities inside which can be used outside of that specific module. The use of registries allows users to easily extract internal capabilities and use them in their own code.

Finally, allowing users access to internal sections allows them to switch parts of the internal characteristics of tools. In the field of ML, this can include functionalities such as custom loss functions, weight initialization schemes, layers, or activation functions. In a more general sense, framework tools can create registries by specifying the sections that can be altered, and controlling how these sections operate in a unified interface.

Consequently, implementation of registries in open-source code allows for quick benchmarks, inherently supports contribution to tools, and promotes code reuse.

3.4 Empirical Evaluation: Application and Impact in Previous Implementations

Previous packages have been implemented with versions of the registry that we propose. Here, we assess their impacts and analyze possible limitations in the implementations. Here, we analyze two model repositories and one ML pipeline package that have internal registries, the graph-based TorchDrug [49], the model training GT4SD package [24] and the ML pipeline package MLFlow [47]. We also compare these packages with how the highly cited and often-used Hugging Face package [45] operates. The Hugging Face package uses an online repository system to collect machine learning models and benchmarking datasets. For this, the

Table 1. Overview of tools. Non-exhaustive overview of open-source tools used for ML and/or drug discovery.

Software	Description
Data tools	
CDK [38]	Chemistry development kit with methods for molecular informatics.
RDKit [21]	Extensive toolkit for cheminformatics logic and functionalities.
OpenBabel [31]	Toolbox with functionalities for chemical languages.
TDC [16]	Collection of benchmarks in several drug discovery applications.
DataMol [27]	Library for intuitive manipulation of molecules.
Datasets [22]	HuggingFace collection of natural language dataset.
...	
Model repositories	
OCHEM [40]	ML framework for the collection of QSAR models.
Transformers [45]	HuggingFace collection of language models.
`bio_embedding` [10]	State-of-the-art language models for protein encoding.
solo-learn [9]	Collection of self-supervised models for representation learning.
GT4SD [24]	Generative modeling environment for material discovery.
...	
ML pipelines	
ODDT [44]	Traditional ML methods applied to drug discovery.
TensorFlow [1]	General tool for deep learning logic.
MLFlow [47]	Standardized ML workflow.
PyTorch [32]	General tool for deep learning logic.
DeepChem [34]	High-level ML framework for drug discovery.
AMPL [28]	High-level ML framework for drug discovery.
MetaFlow [14]	Standardized ML workflow.
TorchDrug [49]	Geometric deep learning for drug discovery.
...	
Workflow systems	
KNIME [36]	Graphical user interface for data analytics with components for ML.
AZOrange [37]	Graphical environment for high-performance ML-based QSAR models.
...	

package uses an alias resolver similar to registry calls to map string names to model instances and classes. As such, it has no obvious relation to the registry design pattern but it does use many of the same functionalities. TorchDrug is a package that uses a registry as an alias resolver analog as well, but that also actively uses it for registration purposes. Here, models and datasets are registered to be easily retrieved by users using simple string representations. It then further supports changing models and datasets within their internal pipeline, opening up the interface to other users. In GT4SD, the available algorithms are stored in a registry that can be called upon to retrieve each algorithm. Here, the registry is used to combine the interfaces of different molecular representation and prediction models and collections, including those from TorchDrug and Hugging Face. This is a good example of how to use registries to combine models from different classes. Finally, the MLFlow package uses online registration of models with version tracking and aliasing. It uses registries, but its registry is online or saved to a local log file. This is primarily used for model training, versioning, and benchmarking.

Both TorchDrug and GT4SD use registries internally built in a modular way but are less useful for user adjustments. As mentioned, registries can be used on different levels to create adaptability and interchangeability. TorchDrug uses its registry to allow users to add datasets to their selection and then use these new datasets similarly to their ways, seamlessly integrating new data into the workflow. GT4SD uses its registries more to standardize the interface between the model libraries of other packages. While Hugging Face uses the basic alias call function, but not the registry function itself, it is clear that it values the capabilities of registries, though prefers the higher-level modularity that allows users to publish code in a GitHub fashion over code snippets.

4 Discussion

In this work, we have introduced the idea of using the registry design pattern to promote code reuse, as well as other good coding practices. In the following section, we discuss previous adoptions, specify important aspects to consider when employing registries, and outline the advantages and disadvantages of registries implemented for research.

4.1 Adoption in Previous Implementations

The three current implementations of registries in packages that we have analyzed, in TorchDrug [49], GT4SD [24] and MLFlow [47], indicate promise usage of registries, but these implementations lack the simple integration of opening up internal modules to changes and only use it to change external models. Hugging Face mentions as much in their description of the Transformers package [45], where models are exclusively used for comparison and simple optimization, not for further refinement. This means that the registries, or registry-like systems, are limited in their applicability. GT4SD does something more interesting, in that it uses its registry to combine the different registered objects from both Hugging Face and TorchDrug. They achieve a large library of models however, it means that the package is somewhat limited in its integration of further models by users. Both TorchDrug and GT4SD have the issue that registries are mostly used internally to resolve and gather different models, rather than a method of code reuse. This can limit external contributions to the package. We also note that models are more often submitted to Hugging Face compared to packages whose registry systems are more code-based rather than online. One of the reasons might be that Hugging Face has a very clear way and tutorials regarding how to contribute to the Transformers package, as well as allowing local integration. A second reason for the mismatch between TorchDrug, GT4SD, and Hugging Face is the broader view as well as the adoption of Hugging Face as a platform, meaning that a critical mass may have been achieved for the Hugging Face package that promotes registering models there over other systems. However, note that TorchDrug also has a significant amount of external contributions and users of more than the implemented models.

4.2 Registration as a Design Pattern

Design patterns are software generic solutions to problems that often arise [13]. When registration is implemented at the start of the project, it enforces modular code. If it is instead adopted into preexisting code, it allows users to use any standardized framework and switch a segment out to replace it with external code. There is an ongoing debate on the effectiveness of design patterns in general, criticizing the relative lack of empirical evidence of effectiveness [2,48]. However, meta-studies conclude that the original design patterns [13] are mostly correlated with system complexity [19], which in turn is positively correlated with system design quality [18]. This leads to the suggestion that registries as a design pattern might be best used in complex systems, or, in the case of ML for drug discovery, in highly consistent systems where the calls to the registry are sparse.

Jaspan et al. have found that coding speed depends on code visibility [20]. Due to the ability to register any object using registries, registration introduces some encapsulation and information hiding because it makes major module logic inaccessible from the execution program. Despite promoting good software design principles, including decoupling [30], this also introduces a layer of invisibility for the programmer. Moreover, the layer of invisibility can impede speed by making it harder to track source code. However, the main advantage of encapsulation is the ease of interchanging modules without knowledge about the encapsulated function. Therefore, the trade-off is the need to inspect the internal logic of encapsulated modules, and researchers should consider this when deciding which objects to keep in a registry.

4.3 Advantages and Considerations of the Registry

Our reformulation of the registry design pattern offers several advantages for software development, including increased modularity and reusability, improved interchangeability, enhanced code clarity, increased stability, and ease of future extensions. In the following section, these advantages and considerations are discussed in more detail.

Modularity. Registration allows researchers to easily list and call objects to standardize workflows and switch out modules in a structured and flexible manner. Researchers can register any object, from small modular functionalities to entire scripts. As a result, total flexibility in the scale of modularity is possible, which crucially also allows the integration of external tools. The dependency of registration on modularity is through their execution mechanism where modules require similar input-output regimens, which inherently pushes for modular design choices in research code. There are two things to consider when making modular design choices.

The first consideration for modular design is composability. The composability of a system refers to the relationships between modules in the execution. Following the classification described by Sarjoughian et al. [35], modules can be composed to follow one another hierarchically (*mono*) or be used within high-level modules (*super*). Additionally, there are the *meta* and *poly* composability options. These options are higher-level systems to translate the *mono* system, where the execution runs on the transformed modules in the higher-level system. Due to their complexity, these latter composability options are often avoided in ML. The use of registries helps to make the composability clear to anyone using the system and spells out what can and should be modified.

The second consideration is that modularity can be coded on different scales of abstraction. Module abstraction describes the scope that a module has. Code can be modular on a small scale of minimal functionalities or large scales, e.g., in the case of ML, it can be one step of data analysis or the entire data preparation. The scale of modularity and abstraction influences the amount of effort needed. Small-scale modularity allows programmers to use the individual parts of the model but requires a more specific framework to combine the code. In comparison, modularity on a larger scale allows for more flexibility in the framework but less reuse of specific code. Consequently, there is a time trade-off between code reuse and time spent on modularity. Registries often use meta-coding principles, the *meta* composability, to register and call modules. This is to limit computational overhead, but registries are still susceptible to computational overhead in frequent calls when the assignment is often overwritten or is not a pointer assignment.

Reusability. Starting from scratch or reconstructing previously written code is an inefficient way to build on previous research. Reusable code instead allows researchers to bypass this initial stage and directly build upon previous research.

When previous work contains an implemented registration system, new work can automatically use the entire system and easily adapt or exchange any registered module [24]. Furthermore, previously registered modules can be called from a new system. In contrast, previous research can also be refactored to support registration, either by modularizing the specific functionalities of interest or by modularizing and registering the section of the execution that requires change. Consequently, registered modules are easily reused in new research and new research can easily build upon old implementations. Both forward-implemented and backward-implemented registries bypass initial avoidable time commitments and allow researchers to focus on the new research immediately.

Interchangeability. Registries for models also help to streamline the process to benchmark various methods. Similarly, a registry of datasets aids the process of applying the same method to varying benchmarks. Easily switching between implementations can increase experimentation speed in writing code connected with new research. Modularity allows researchers to experiment more easily with various options for code applications. Registration, in turn, further allows researchers to variate, update, adapt, and modify modules because registered modules are inherently modular. This automatically forces code to adhere to the five R's of published work [5], as discussed in the introduction. Depending on the abstraction level of the modules, other researchers can variate and use the coding framework, as well as easily replace and reuse parts of it. Furthermore, it increases code longevity as outdated code can be easily updated. Similarly, due to increased reuse by other researchers, citations and longevity increase as the original author is no longer the only one with a vested interest in the original code.

Tools. Registration of modules inside the toolkit can allow users to retrieve modules and generate new applications following tool specifications. The latter decreases the threshold for new functionalities to be suggested to the main tool. Moreover, registries allow researchers to build functionalities in private repositories using the framework set out by the toolkit and then easily upload those, once the original work has been published. As previously stated, some available tools already provide versions of the registration feature [24,47,49]. However, their scope is mostly limited to the registration of ML models. The main advantage of our proposed generic registration, which is missing in previous implementations, is the flexibility to make specific design choices.

Clarity. Research collaborations in multidisciplinary fields, such as drug discovery, rely on integrating code from various sources and contributors, thus requiring deliberate forethought and coordination. As such, a standardized workflow is of particular importance and written code should be modular to avoid code instability when working on different interdependent code sections. The registration system creates a clear structure to use the registered modules, thus allowing researchers or project coordinators to standardize their workflows and call functionalities through registries where code can be varied. Consequently, registration increases the efficiency of research collaborations.

Stability. The stability of a codebase can be affected when new modules are added. As new functionality is introduced, the potential for interactions and conflicts with existing code increases. This can lead to defects and unexpected behavior. Additionally, adding new modules to a codebase can increase its complexity and make it more difficult to maintain and understand, which can lead to issues in the long term. To mitigate these risks, it is important to have a thorough testing process in place before new modules are added and to thoroughly review and test the entire codebase after the new modules are integrated. A registry can be used to increase the stability of code through specified points to integrate new modules as well as the ability to introduce post-registration checks. These post-checks can then be used to enforce adaptation of external code to the existing framework, such as passing a set of tests or adherence to meta information, such as versioning and factory patterns. As an example, registries can be set up to handle different versions of the same module such that modules are registered together with their version. This way users can more easily track the influence of changing modules. Additionally, the modularity of new functionality ensures minimal impact on the existing codebase.

4.4 Codebases for Efficient Coding in Research

Codebases focus more on the software development process. As such, a codebase should particularly ease continuous development, aid code stability, and allow for incremental addition of modules. Due to the multidisciplinary field, ML research focused on drug discovery uses multiple external tools. The usage of tools ranges from the curation and processing of data to the general setup and deployment of models. While the ML pipelines are primarily created for individual projects separately, the developed functionalities can often be helpful for other projects or researchers in the same field. Functionalities from individual projects are often presented with an irregularity in the level of modularity and, therefore, accessibility and usability.

For collaborations or big projects, a choice is made between keeping multiple single-application repositories or creating a bigger shared codebase. There is an open discussion on the advantages and disadvantages of codebases over multiple single-project repositories [20]. Codebases represent an opportunity for collaborators to standardize their workflows as well as share and reuse code with particular functionalities. In general, more standardized workflows allow for more specific coding criteria. Modular code or modular tools are parts of the workflow that can be easily updated with or interchanged for applications with similar functionality. More modular codebases increase collaborator contribution [4] and allow for more use of external tools within its framework. By using registries when designing a codebase, active choices can be made to promote modularity in the project. The benefits of choosing a high level of modularity include code longevity, reuse, and increased potential for collaboration and research speed. Therefore, using a codebase can widen the scope of single, multidisciplinary research projects, making them more modular and reusable for other projects or researchers.

On the other hand, there are disadvantages to coding in a codebase environment: 1) Even though the level of modularity is flexible, this level should be static during research development to prevent unnecessary overhead. It can otherwise be costly to uphold the modularity and maintain backward compatibility. 2) Codebases can restrict reproducibility. Reproducibility requires specific versioning, which can be more fluid in the continuous development within codebases. 3) Codebases can introduce irrelevant functionalities that obscure crucial functionalities in open-source publications. 4) Although coding in a codebase can speed up research long-term, setting up research using modular code is more time-consuming.

However, we advocate for a more general perspective on codebases where any published code can be considered a codebase. This interpretation of published code is less dependent on actual features of codebases, as most code in research often does not require active design, nor intricate design patterns to function properly. Instead, we view the act of writing code as actively assuming others will reuse parts of the code, which will then promote the idea of modularity and reusability, including the use of registries where warranted. This view is more flexible in its application and gives the scientist a base from which to work.

4.5 Future Work

The capabilities of registries can be further explored. For example, future research can further investigate how registration aids reproducibility in practice through experiments or surveys. As previously mentioned, continuous development and non-contributing code can impede reproducibility and clarity. Similarly to the accreditation system, versioning, and optional factory patterns, other modules could be attached to registered modules in the registry to aid stability and reproducibility. One can even imagine a generative functionality of the registry to produce a single repository of only the necessary modules from a codebase for a specific application.

A common issue that can occur when combining modules from different sources is that versions can be incompatible. Registries, as proposed here, would not immediately solve problems with dependency conflicts but one can imagine an extension where the registration of modules is accompanied by dependency requirements that are automatically checked and installed upon execution. This in turn will not deal with situations where different parts of the code use different versions of third-party dependencies. However, it would allow for the registration of modules that depend on conflicting versions into the same registry where only one at a time is used during execution.

Furthermore, future research might analyze how their effectiveness depends on using a generic registry. While higher usage promotes modularity, it also removes a level of visibility. An analysis of the overall usage and the usage focused on specific groups of modules will give a better understanding of the best practices of registries. Likewise, one can investigate the trade-off between code visibility and coding speed as noted by Jaspan et al. [20]. Ultimately, collaborating researchers should investigate whether using registries and codebases instead of multiple repositories is advantageous for their research goals and try to design their code to best suit their needs.

5 Conclusions

To conclude, we highlight the importance and promise of the registry design pattern, especially in the field of ML development for drug discovery. Registries can promote code reuse through their modular nature. Modularity is the independence of a module from the rest of the code and is crucial for reuse. A registry also promotes other important coding practices and includes the possibility to easily switch between custom functionalities and functionalities from open-source tools. We introduce a method to flexibly register objects and add additional functionalities such as accreditation and versioning. Additionally, we outline the advantages and considerations of registries and stress that registries clarify the usually concealed abstraction and composability of a system. Finally, registries promote clarity, experimentation speed, good coding practices, and code reuse.

Acknowledgements. We thank our colleagues and reviewers for their valuable feedback. Additionally, the authors thank BioRender for the publication of the figures generated with BioRender authorized under the subscription plan of Peter Hartog. This study was partially funded by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Actions grant agreement "Advanced machine learning for Innovative Drug Discovery (AIDD)" No. 956832.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Abadi, M., et al.: TensorFlow: a system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pp. 265–283. USENIX Association (2016)
2. Almadi, S.H., Hooshyar, D., Ahmad, R.B.: Bad smells of gang of four design patterns: a decade systematic literature review. Sustainability **13**(18), 10256 (2021)
3. Baker, M.: Reproducibility crisis. Nature **533**(26), 353–66 (2016)
4. Baldwin, C.Y., Clark, K.B.: The architecture of participation: does code architecture mitigate free riding in the open source development model? Manage. Sci. **52**(7), 1116–1127 (2006)
5. Benureau, F.C., Rougier, N.P.: Re-run, repeat, reproduce, reuse, replicate: transforming code into scientific contributions. Front. Neuroinform. **11**, 69 (2018)

6. Cadwallader, L., Hrynaszkiewicz, I.: A survey of researchers' code sharing and code reuse practices, and assessment of interactive notebook prototypes. PeerJ **10**, e13933 (2022)
7. Chen, H., Engkvist, O., Wang, Y., Olivecrona, M., Blaschke, T.: The rise of deep learning in drug discovery. Drug Discov. Today **23**(6), 1241–1250 (2018)
8. Chollet, F., et al.: Keras (2015). https://keras.io
9. da Costa, V.G.T., Fini, E., Nabi, M., Sebe, N., Ricci, E.: solo-learn: a library of self-supervised methods for visual representation learning. J. Mach. Learn. Res. **23**(56), 1–6 (2022)
10. Dallago, C., et al.: Learned embeddings from deep learning to visualize and predict protein sets. Curr. Protoc. **1**(5), e113 (2021)
11. Dara, S., Dhamercherla, S., Jadav, S.S., Babu, C., Ahsan, M.J.: Machine learning in drug discovery: a review. Artif. Intell. Rev. **55**(3), 1947–1999 (2022)
12. Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R., Stafford, R.: Patterns of Enterprise Application Architecture. Addison-Wesley Professional (2002)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Deutschland GmbH, Munich (1995)
14. Goyal, S.: More data science, less engineering: a Netflix original. In: 2020 USENIX Conference on Operational Machine Learning (2020)
15. Howard, J., Gugger, S.: FastAI: a layered API for deep learning. Information **11**(2), 108 (2020)
16. Huang, K., et al.: Therapeutics data commons: machine learning datasets and tasks for drug discovery and development. In: Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1) (2021)
17. Hunt, A., Thomas, D.: The Pragmatic Programmer. Addison-Wesley, Boston, United States (1999)
18. Hussain, S., Keung, J., Khan, A.A.: The effect of gang-of-four design patterns usage on design quality attributes. In: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 263–273. IEEE (2017)
19. Hussain, S., Keung, J., Khan, A.A., Bennin, K.E.: Correlation Between the Frequent Use of Gang-of-four Design Patterns and Structural Complexity. In: 2017 24th Asia-Pacific Software Engineering Conference (APSEC), pp. 189–198. IEEE (2017)
20. Jaspan, C., et al.: Advantages and disadvantages of a monolithic repository: a case study at Google. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, pp. 225–234 (2018)
21. Landrum, G.: RDKit: Open-Source Cheminformatics (2006). https://doi.org/10.5281/zenodo.6961488, http://www.rdkit.org
22. Lhoest, Q., et al.: Datasets: a community library for natural language processing. In: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pp. 175–184. Association for Computational Linguistics (2021)
23. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6), 1811–1841 (1994)
24. Manica, M., et al.: Accelerating material design with the generative toolkit for scientific discovery. NPJ Comput. Mater. **9**(1), 69 (2023)
25. Martin, R.C.: The dependency inversion principle. C++ Report **8**(6), 61–66 (1996)
26. Martin, R.C.: Design principles and design patterns. Object Mentor **1**(34), 597 (2000)
27. Mary, H., et al.: Datamol: molecular manipulation made easy (2022). https://doi.org/10.5281/zenodo.6856321, https://datamol.io/

28. Minnich, A.J., et al.: AMPL: a data-driven modeling pipeline for drug discovery. J. Chem. Inf. Model. **60**(4), 1955–1968 (2020)
29. Gnu general public license, version 3. https://opensource.org/licenses/MIT. Accessed 17 January 2022
30. Mo, R., Cai, Y., Kazman, R., Xiao, L., Feng, Q.: Decoupling level: a new metric for architectural maintenance complexity. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp. 499–510. IEEE (2016)
31. O'Boyle, N.M., Banck, M., James, C.A., Morley, C., Vandermeersch, T., Hutchison, G.R.: Open Babel: an open chemical toolbox. J. Cheminf. **3**(1), 1–14 (2011)
32. Paszke, A., et al.: PyTorch: an imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems, vol. 32 (2019)
33. Pressman, R.S.: Software Engineering: A Practitioner's Approach. Palgrave Macmillan, Gurgaon, India (2005)
34. Ramsundar, B., Eastman, P., Walters, P., Pande, V.: Deep Learning for the Life Sciences: Applying Deep Learning to Genomics, Microscopy, Drug Discovery, and More. O'Reilly Media, Sebastopol (2019)
35. Sarjoughian, H.S.: Model composability. In: Proceedings of the 2006 Winter Simulation Conference, pp. 149–158. IEEE (2006)
36. Sieb, C., Meinl, T., Berthold, M.R.: Parallel and distributed data pipelining with KNIME. Mediterr. J. Comput. Netw. **3**(2), 43–51 (2007)
37. Stålring, J.C., Carlsson, L.A., Almeida, P., Boyer, S.: AZOrange - high performance open source machine learning for QSAR modeling in a graphical programming environment. J. Cheminf. **3**(1), 1–10 (2011)
38. Steinbeck, C., Han, Y., Kuhn, S., Horlacher, O., Luttmann, E., Willighagen, E.: The Chemistry Development Kit (CDK): an open-source java library for chemo- and bioinformatics. J. Chem. Inf. Comput. Sci. **43**(2), 493–500 (2003)
39. Sullivan, K.J., Griswold, W.G., Cai, Y., Hallen, B.: The structure and value of modularity in software design. ACM SIGSOFT Softw. Eng. Notes **26**(5), 99–108 (2001)
40. Sushko, I., et al.: Online Chemical Modeling Environment (OCHEM): web platform for data storage, model development and publishing of chemical information. J. Comput. Aided Mol. Des. **25**(6), 533–554 (2011)
41. Tomar, V., Mazumder, M., Chandra, R., Yang, J., Sakharkar, M.K.: Small molecule drug design. In: Ranganathan, S., Gribskov, M., Nakai, K., Schönbach, C. (eds.) Encyclopedia of Bioinformatics and Computational Biology, pp. 741–760. Academic Press, Oxford (2019)
42. Vamathevan, J., et al.: Applications of machine learning in drug discovery and development. Nat. Rev. Drug Discov. **18**(6), 463–477 (2019)
43. William, F.: PyTorch Lightning (2019). https://doi.org/10.5281/zenodo.3828935, https://www.pytorchlightning.ai
44. Wójcikowski, M., Zielenkiewicz, P., Siedlecki, P.: Open Drug Discovery Toolkit (ODDT): a new open-source player in the drug discovery field. J. Cheminf. **7**(1), 1–6 (2015)
45. Wolf, T., et al.: Transformers: state-of-the-art natural language processing. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pp. 38–45. Association for Computational Linguistics (2020)
46. Xiang, Y., Pan, W., Jiang, H., Zhu, Y., Li, H.: Measuring software modularity based on software networks. Entropy **21**(4), 344 (2019)
47. Zaharia, M., et al.: Accelerating the machine learning lifecycle with MLflow. IEEE Data Eng. Bull. **41**(4), 39–45 (2018)

48. Zhang, C., Budgen, D.: What do we know about the effectiveness of software design patterns? IEEE Trans. Softw. Eng. **38**(5), 1213–1231 (2011)
49. Zhu, Z., et al.: TorchDrug: A powerful and flexible machine learning platform for drug discovery. arXiv preprint arXiv:2202.08320 (2022)