DepIn-O: An Ontology on Dependency Injection Software Frameworks

Cleisson S. Guterres , Camila Z. de Aguiar and Vítor E. S. Souza

[1]*Ontology & Conceptual Modeling Research Group (NEMO), Universidade Federal do Espirito Santo (UFES), Brasil Av. Fernando Ferrari, 514 - Goiabeiras - Vitória, ES - 29075-910*

Abstract

Dependency Injection (DI) is a state-of-practice design pattern utilized for implementing inversion of control, with various DI frameworks widely adopted in many object-oriented programming languages. However, to the best of our knowledge, a formal definition of the associated terminologies within these frameworks has yet to be established. This lack of standardization can make it challenging to achieve semantic interoperability goals, such as code migration between different frameworks or identification of architectural issues regardless of the specific framework in use. To tackle this challenge, we propose DepIn-O, a domain reference ontology designed to capture and express the semantic concepts associated with DI within software development.

Keywords

Dependency Injection, Ontology, Ontology Engineering, DepIn-O, SABiOx

1. Introduction

Dependency Injection (DI) frameworks are extremely useful and popular tools to facilitate the implementation of DI patterns across multiple object-oriented (OO) programming languages, and their usage is state-of-the-practice. Despite these facts, to the best of our knowledge, there are no proposals in the literature that provide a formalization of the DI concepts employed in frameworks, required to pave the way for the automation of semantic interoperability tasks, such as, e.g., detection of code smells — pieces of code that signal a bad programming practice and warrant further inspection [1] — or code migration.

To change that scenario, we introduce the Dependency Injection Ontology (DepIn-O), an ontology with the goal of representing the fundamental semantic concepts associated with the DI domain. Our ontology was developed following the SABiOx method [2], built upon the foundation laid out by UFO [3], specialized concepts provided by OOC-O [4] and modeled using OntoUML [5]. Its evaluation was performed through verification — answering the competency questions raised when eliciting the requirements — and validation — mapping the ontology concepts to instances within some of the most popular DI frameworks currently available.

The remaining sections of this paper are organized as follows: Section 2 summarizes the baseline on Dependency Injection and its frameworks, Ontology Engineering, and foundational

Proceedings of the XVI Seminar on Ontology Research in Brazil (ONTOBRAS 2023) and VII Doctoral and Masters Consortium on Ontologies (WTDO 2023), Brasilia, Brazil, August 28 - September 01, 2023.

cleissonsg@gmail.com (C. S. Guterres); camila.z.aguiar@ufes.br (C. Z. d. Aguiar); vitor.souza@ufes.br (V. E. S. Souza)

and domain ontologies used as a basis for the construction of DepIn-O; Section 3 presents our ontology; Section 4 describes the evaluation of DepIn-O through verification, validation and an application; Section 5 compares our work with related works; and Section 6 presents our final considerations and discusses future possibilities. All supplementary material mentioned in this paper is available at https://nemo.inf.ufes.br/en/projetos/sfwon/.

2. Baseline

In this section, we summarize the baseline of our work: Dependency Injection and its supporting frameworks, and the ontological foundations of DepIn-O.

2.1. Dependency Injection

Dependency Injection (DI) is a set of software design patterns that enable the development of loosely coupled code. It encompasses three main principles: *Composition*, *Lifetime Management*, and *Specification & Interception* [6].

Composition states that objects must receive their dependencies externally instead of creating them internally [7]. A *dependency* is a piece of code essential for a different part of the code to work [8]. The receiver of a dependency is referred to as its *consumer*. Composition provides the foundation of DI and paves the way for the other two principles. Listing 1 exemplifies a Composition.

Listing 1: DI Composition example in Java.

```java
public interface DependencyA {}

public class ConsumerCI {
    DependencyA dep;

    public ConsumerCI(DependencyA service) {    // receives external dependency
        dep = service;
    }
}
```

Composition removes from the consumer the knowledge of how its dependencies are implemented. To benefit from it, we should supply our consumers with abstract dependencies, as done in Listing 1, and later *specify* a concrete implementation of said dependency. That way, we gain flexibility, as dependencies can be easily swapped out — in case of migration — or mocked for unit testing. As a direct consequence, we can apply the *interception* pattern by employing decorators: a class that can use the original dependency as its consumer and be used by the original consumer as its dependency, as exemplified in Listing 2.

Listing 2: Decorator example in Java.

```java
// DecoratorA is an implementation of DependencyA
public class DecoratorA implements DependencyA {
    DependencyA dep;

    //  DecoratorA is also a consumer of DependencyA
    public DecoratorA(DependencyA service){
        dep = service;
    }
}

public class DependencyImpl implements DependencyA {}

// later usage and instantiation
```

```
DependencyImpl service = new DependencyImpl();
DecoratorA decorator = new DecoratorA(service);
ConsumerCI consumer = new ConsumerCI(decorator);
```

Composition also removes from the consumer the responsibility over the *lifetime management* of its dependencies, thus allowing the developer to define when to share the same instance of a dependency between different consumers and when to provide a different instance to a consumer. This is exemplified in Listing 3, as *serviceX* is shared between *consumerA* and *consumerB*, while *serviceY* is only provided to *consumerC*.

Listing 3: Dependency shared between different Consumers in Java.

```
DependencyImpl serviceX = new DependencyImpl();
DependencyImpl serviceY = new DependencyImpl();
ConsumerCI consumerA = new ConsumerCI(serviceX);
ConsumerCI consumerB = new ConsumerCI(serviceX);
ConsumerCI consumerC = new ConsumerCI(serviceY);
```

Assuming a good understanding of DI's underlying principles, *frameworks* become extremely valuable tools to streamline the implementation of DI patterns across multiple object-oriented (OO) programming languages. A software framework generally provides a skeletal abstraction of a solution to a number of problems that have some similarities [9]. A DI framework is a software library that either explicitly or implicitly provides an Injector that automates many of the tasks involved in Object Composition, Specification & Interception, and Lifetime Management, constructing and resolving object graphs [6].

2.2. Ontological Foundations

The fact that DI patterns are applicable to any OO language and are supported by numerous frameworks stumbles on a well-known issue: the problem of Semantic Interoperability, i.e., combining independently conceived information spaces and providing unified analytics over them. Ontologies tackle that challenge, as they aim to produce concrete representation models of conceptualizations of reality that are consistent and facilitate interoperability [10]. This is our motivation to build an ontology on DI.

The use of an Ontology Engineering method can help improve the quality of the process and, thus, the quality of the results. The SABiO method [11] has been successfully used for developing several ontologies in the Software Engineering domain (e.g., [12, 13]), given its focus on the development of domain ontologies. More recently, however, SABiOx [2] has been proposed as an extension of SABiO, incorporating agile principles and providing more details on the activities that compose its process in order to help guide the ontology engineer. For these reasons, it was chosen as our Ontology Engineering method for the construction of DepIn-O.

As proposed by SABiOx, we defined a Foundational Ontology to be the basis of our work. We chose the Unified Foundational Ontology (UFO) due to its notable capacity to provide conceptual clarification in complex domains. In particular, we employed UFO-A, an ontology of endurants that deals with aspects of structural conceptual modeling [3].

Another procedure of SABiOx is choosing a Modeling Language. We opted for OntoUML, an extension of the Unified Modeling Language (UML) based on UFO that incorporates principles from Ontology and Logic Based Modeling to provide a rigorous, expressive, and rich set of modeling constructs for representing different types of entities and relationships. [5] To ensure conformity to it, we utilized the OntoUML plugin for Visual Paradigm[1] to model our ontology.

SABiOx also suggests reusing related Domain Ontologies, if possible. Since Dependency Injection is a subdomain of Object-Oriented Programming, we found that the Object-Oriented Code Ontology (OOC-O) provides substantial support for the notions presented in DepIn-O. OOC-O is a Domain Ontology that identifies and represents the semantics of the entities present at compile time in object-oriented code [4]. Figure 1 presents a fragment of OOC-O referenced and reused by our ontology.

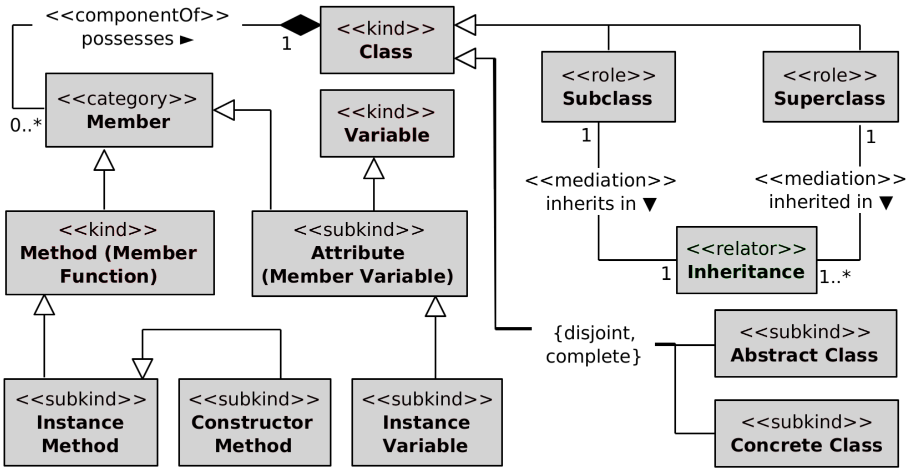


Figure 1: OOC-O Fragment for Reuse.

In OOC-O, every **Class** must be either a **Concrete Class**, which is fully implemented and can have instances, or an **Abstract Class**, which is an incompletely implemented class used as a basis for its descendants. A **Member** is a component of a class, such as an **Attribute (Member Variable)**, which is a **Variable** that is also a Member; or a **Method (Member Function)**, a function that defines the behavior of an object. An **Instance Method** is a specialization of Method that operates only on objects of its Class; the **Constructor Method** is a specialization of Instance Method that specifies the process of creating and initializing an object. An **Instance Variable** is an Attribute that delineates characteristics of an object in the context of a specific instance of its root Class. A Class assumes the role of a **Superclass** when its instance variables and methods are provided by **Inheritance** to another Class, defined as **Subclass**.

In the next sections, to make explicit the references to OOC-O concepts in the text, we use a prefix, e.g., "OOC-O Instance Method", rather than just "Instance Method".

3. DepIn-O: a Dependency Injection Ontology

The main purpose of DepIn-O is to represent the fundamental conceptual features associated with the Dependency Injection domain in a unified and consensual manner across various

[1]https://github.com/OntoUML/ontouml-vp-plugin

Table 1
DepIn-O: Competency Questions.

ID	Competency Question
CQ01	What defines Dependency Injection associations?
CQ02	What are the specializations of Injection Point?
CQ03	How to abstract and implement Dependencies?
CQ04	What defines a Decorator?
CQ05	What determines which implementation of a Dependency is provided to its Consumer?
CQ06	What are the main specializations of Dependency that define its lifetime?

programming languages and frameworks.

We conducted an analysis of frameworks developed for the top three most used OO Programming Languages according to the 2022 Stack Overflow Developer Survey: Java, C#, and Python [14]. This survey was chosen due to its popularity among the software development community. The selected frameworks were Dependency Injector for Python; CDI and Spring for Java; Autofac and Simple Injector for C#.

Following SABiOx, we delimited our focus in the vertical dimension to the level of design-time constructs, without delving into the perspective of runtime; and in the horizontal dimension to include concepts associated with the main DI principles listed in Section 2.1, provided that they were present in two or more of the examined frameworks. The result is documented as DepIn-O's **Catalog of Concepts** and is available as supplementary material.

SABiOx suggests the elaboration of Competency Questions (CQs) to elicit the functional requirements of an Ontology. Table 1 presents these questions for DepIn-O.

To facilitate comprehension, maintenance, and future reutilization, SABiOx proposes partitioning an Ontology into modules in conformity with previously identified subdomains. We divided DepIn-O into modules according to the main principles of DI previously discussed — Composition (*DepIn-O: Composition*), Specification & Interception (*DepIn-O: Specification*) and Lifetime Management (*DepIn-O: Scopes*). Their relationships to each other and to OOC-O are shown in Figure 2 and, in what follows, we detail each partition of the ontology. The original stereotype-based color scheme from the OntoUML plugin was altered in the figures in favor of a module-based color scheme to facilitate the visualization of the relationships between the modules. The complete (unpartitioned) view of DepIn-O is also available as supplementary material.

3.1. DepIn-O: Composition Module

Figure 3 presents the OntoUML diagram of the DepIn-O Composition Module. The concepts of **Injection Point**, **Consumer**, and **Dependency** are interlinked roles that cannot exist independently. An element is defined as an Injection Point when it composes a **Dependency Injection** with an OOC-O Class, which is then specialized as the Dependency, while the OOC-O Class that possesses the Injection Point is specialized as the Consumer.

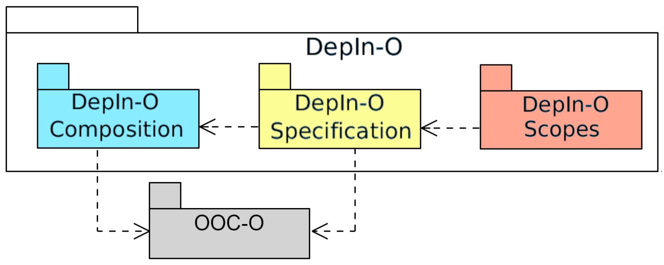


Figure 2: DepIn-O modules and their relationships with OOC-O.

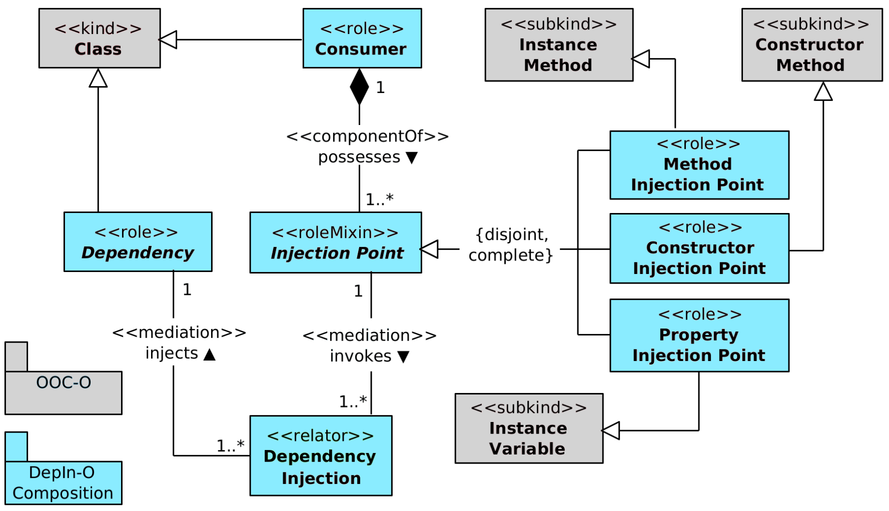
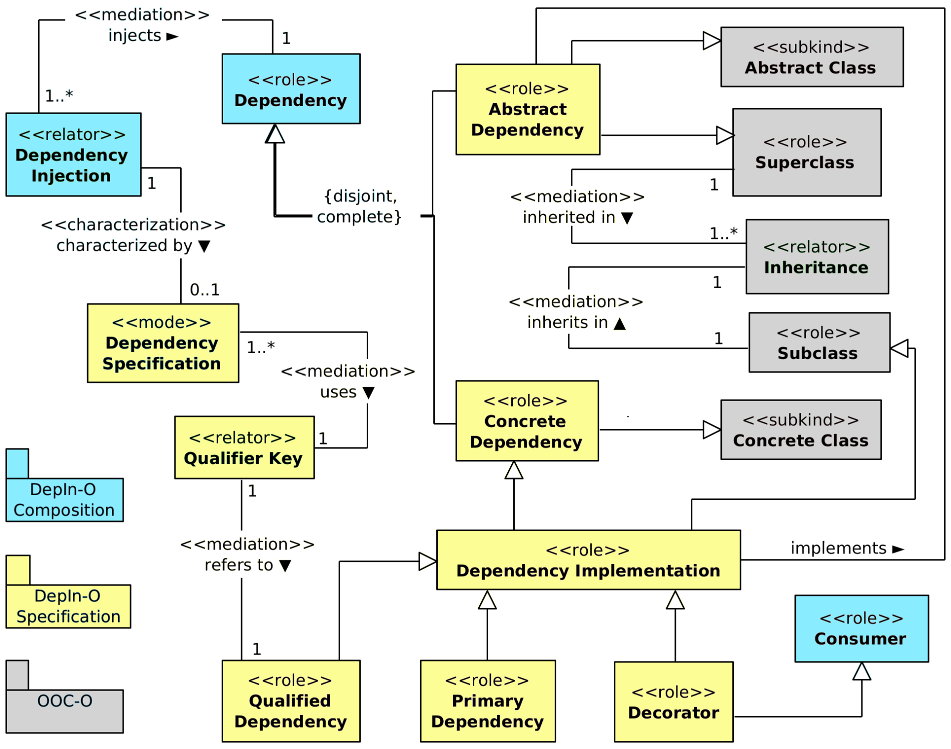

<<kind>> Class
<<role>> Consumer
<<subkind>> Instance Method
<<subkind>> Constructor Method
1
<<componentOf>> possesses ▼
1..*
<<role>> Method Injection Point
<<role>> Dependency
<<roleMixin>> Injection Point
{disjoint, complete}
<<role>> Constructor Injection Point
1
1
<<mediation>> injects ▲
<<mediation>> invokes ▼
<<role>> Property Injection Point
OOC-O
1..*
<<subkind>> Instance Variable
1..*
<<relator>> Dependency Injection
DepIn-O Composition

Figure 3: OntoUML diagram of the DepIn-O Composition Module.

When assuming the role of an Injection Point, an OOC-O Constructor Method is defined as a **Constructor Injection Point**, while a non-Constructor OOC-O Instance Method is defined as a **Method Injection Point** — this separation is particularly significant, as frameworks typically handle constructors differently from non-constructors when wiring DI compositions. When an OOC-O Instance Variable is an Injection Point, it is called a **Property Injection Point**. Since OOC-O Instance Method and OOC-O Instance Variable inherit different identity principles (cf. Figure 1) an **Injection Point** is a *RoleMixin*, an abstract generalization set that aggregates concrete elements of distinct identity principles that fit the same role. It is also important to note that the roles that specialize Injection Point are characterized by the relation of the latter with **Dependency Injection**, which they inherit.

As general rules, a Consumer can have multiple Injection Points and an Injection Point can set up multiple Dependency Injections; we employ axioms to describe the exceptions to these rules, namely: a Consumer can only have one Constructor Injection Point; a Property Injection Point can only invoke one Dependency Injection. As mentioned before, there is no restriction on how many Consumers can share the same Dependency, so a Dependency can also be injected by multiple Dependency Injections. Still, every Dependency Injection is unique between one



<<mediation>> injects ►
1..*
1
<<role>> Dependency
<<relator>> Dependency Injection
1
{disjoint, complete}
<<characterization>> characterized by ▼
0..1
<<mode>> Dependency Specification
1..*
<<mediation>> uses ▼
<<relator>> Qualifier Key
1
1
<<mediation>> refers to ▼
1
DepIn-O Composition
DepIn-O Specification
OOC-O
<<role>> Qualified Dependency
<<role>> Primary Dependency
<<role>> Decorator
<<role>> Consumer
<<role>> Abstract Dependency
<<subkind>> Abstract Class
<<role>> Superclass
<<mediation>> inherited in ▼
1
1..*
<<relator>> Inheritance
1
<<mediation>> inherits in ▲
1
<<role>> Subclass
<<role>> Concrete Dependency
<<subkind>> Concrete Class
<<role>> Dependency Implementation
implements ►

Figure 4: OntoUML diagram of the DepIn-O Specification Module.

Dependency and one Injection Point from one Consumer.

3.2. DepIn-O: Specification Module

Figure 4 presents the OntoUML diagram of the DepIn-O Specification Module. We define a Dependency that specializes OOC-O Concrete Class as a **Concrete Dependency**. In contrast, an **Abstract Dependency** specializes not only OOC-O Abstract Class but also OOC-O Superclass, since it requires at least one implementation to be provided to resolve it. This implementation must be a Concrete Dependency that also specializes OOC-O Subclass, as it is linked to the Abstract Dependency through the OOC-O Inheritance relator, and we refer to it as a **Dependency Implementation**. As with the previous module, the roles that specialize Dependency are characterized by the relation of the latter with **Dependency Injection**.

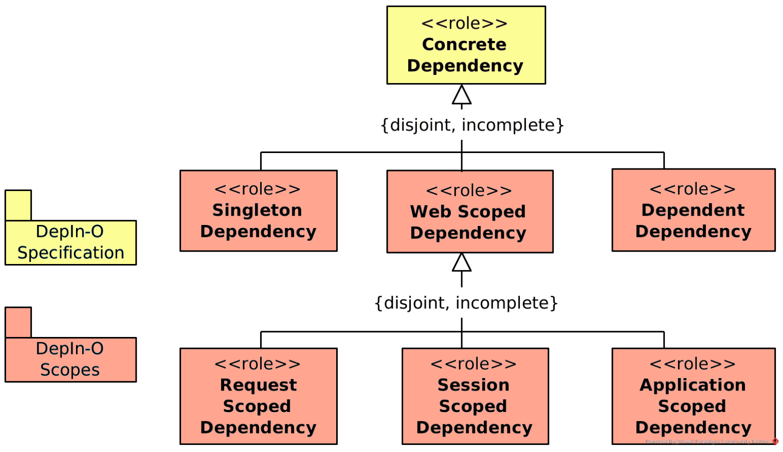
An important observation arises from the fact that while OOC-O supports the concept of multiple inheritance, DepIn-O does not, as a Dependency Implementation is restricted to inheriting from only one Abstract Dependency. However, there is no upper limit on the number of implementations that can be provided for one Abstract Dependency. Consequently, the existence of multiple Dependency Implementations for the same Abstract Dependency raises a question regarding the determination of the specific implementation that will be invoked to resolve a call to an Abstract Dependency.

A Dependency Implementation becomes a **Primary Dependency** when it resolves the

Abstract Dependency by default. On the other hand, when a Dependency Injection demands the existence of a **Dependency Specification** to call for a specific Dependency Implementation, then that Dependency Implementation becomes a **Qualified Dependency**. A Dependency Specification uses a **Qualifier Key** to reference the Qualified Dependency to be used. We define a **Decorator** as a Dependency Implementation that is also a Consumer of the same Dependency it implements. We employ an axiom to describe the rule regarding Abstract Dependencies, namely: there must be at least one Dependency Implementation that is not a Decorator for every Abstract Dependency.

3.3. DepIn-O: Scopes Module

Figure 5 presents the OntoUML diagram of the DepIn-O Scopes Module. Every Concrete Dependency is specialized by a scope that defines its intended lifetime. This aspect is crucial to determine the specific object instance to be provided within the context of a Dependency Injection.



<<role>> Concrete Dependency
{disjoint, incomplete}
<<role>> Singleton Dependency
<<role>> Web Scoped Dependency
<<role>> Dependent Dependency
DepIn-O Specification
{disjoint, incomplete}
DepIn-O Scopes
<<role>> Request Scoped Dependency
<<role>> Session Scoped Dependency
<<role>> Application Scoped Dependency

Figure 5: OntoUML diagram of the DepIn-O Scopes Module.

A **Singleton Dependency** means only one instance of the dependency will be created for the application, hence if multiple distinct Consumers request the same Dependency, all of them will be supplied with the same instance of said Dependency. In contrast, a **Dependent Dependency** means its instances follow the same lifetime as their respective Consumers, therefore if multiple distinct Consumers request the same Dependency, each one of them will be provided with a different instance of said Dependency. Whenever a Concrete Dependency is not explicitly specified in regards to its scope, virtually all frameworks or language compilers make it either singleton or dependent by default.

Most DI frameworks also provide support for Web-based applications, so we added the most common **Web Scoped Dependency** specifications to our Ontology. A **Request Scoped Dependency** implies that a new instance of the same dependency will be created for every new HTTP request, such as refreshing the page. A **Session Scoped Dependency** indicates that a new instance of the same Dependency will be created for every new HTTP session, but the same

Table 2
DepIn-O Verification.

ID	Answer to Competency Question
CQ01	**Consumer**, **Dependency** *subtype of* **Class**; **[Many] Injection Point** *component of* **[One] Consumer**; **[One] Injection Point** *invokes* **[Many] Dependency Injection** ; **[One] Dependency** *injected by* **[Many] Dependency Injection** ; **[One] Dependency Injection** *mediates* **[One] Injection Point**, **[One] Dependency**.
CQ02	**Constructor Injection Point** *subtype of* **Injection Point**, **Constructor Method**; **Method Injection Point** *subtype of* **Injection Point**, **Instance Method**; **Property Injection Point** *subtype of* **Injection Point**, **Instance Variable**.
CQ03	**Abstract Dependency** *subtype of* **Abstract Class**, **Superclass**, **Dependency**; **Concrete Dependency** *subtype of* **Concrete Class**, **Dependency**; **Dependency Implementation** *subtype of* **Concrete Dependency**, **Subclass**.
CQ04	**Decorator** *subtype of* **Dependency Implementation**, **Consumer**.
CQ05	**Primary Dependency**, **Qualified Dependency** *subtype of* **Dependency Implementation**; **Dependency Injection** *characterized by* **Dependency Specification**. **[Many] Dependency Specification** *uses* **[One] Qualifier Key** **[One] Qualifier Key** *refers to* **[One] Qualified Dependency**
CQ06	**Singleton Dependency**, **Dependent Dependency**, **Web Scoped Dependency** *subtype of* **Concrete Dependency**.

instance will be maintained throughout the session. The **Application Scoped Dependency** denotes that a single instance of the same dependency will exist for the entirety of the Web application.

There are countless other scope-related specializations for dependencies provided by DI frameworks which we did not include in our ontology due to our delimitations. Furthermore, many DI frameworks offer not only predefined scope specializations but also the ability for users to create customized scopes to suit their specific needs.

4. Evaluation

The evaluation of the ontology was performed by verification and validation activities, according to SABiOx. For **Verification**, it is necessary to identify if the elements that make up the ontology are able to answer the competency questions raised as its requirements. Table 2 shows the result of DepIn-O's verification.

For **Validation**, it is necessary to determine if the ontology matches the reality, that is, if

Listing 4: Dependency Injection Compositions in Spring.

```
public interface DependencyA {}

@Component
public class ConsumerCI {
    DependencyA dep;

    public ConsumerCI(DependencyA da) {      // Constructor Injection Point
        dep = da;
    }
}

@Component
public class ConsumerMI {
    public void someMethod(DependencyA da) {}     // Method Injection Point
}

@Component
public class ConsumerPI{
    @Autowired DependencyA dep;     // Property Injection Point
}
```

DepIn-O's concepts can provide a consensual understanding of the domain across different programming languages and frameworks as intended. Therefore, we mapped DepIn-O's Catalog of Concepts to instances in all of our chosen frameworks. For brevity, we only present here an instantiation employing the Java framework Spring, and its corresponding diagram is presented as supplementary material. The full validation of DepIn-O on the other frameworks is also available as supplementary material. Note that the code examples from Section 2.1 employed a "Pure DI" approach, not using resources from any framework.

As previously stated, for the composition of a **Dependency Injection**, we need the **Consumer**, its **Injection Point** and the **Dependency**. In Spring, the *@Component* annotation is added to Consumers and Concrete Dependencies to automate their instantiation and to set up injections when using Constructor and Method Injection Points. However, setting up an injection with a Property Injection Point also requires the *@Autowired* annotation. This is presented in Listing 4.

Next, we show **Dependency Implementations**. To instantiate a **Primary Dependency**, we use *@Primary* alongside *@Component* (however, when there is only one implementation provided to the abstraction, *@Primary* is optional). For a **Qualified Dependency**, besides *@Component*, we need to use the *@Qualifier("label")* annotation at the Injection Point — where "label" is the **Qualified Key** and can be a name of our choice — and add to the **Dependency Specification** a matching *@Qualifier("label")*. One of the possible ways to implement a **Decorator** in Spring is to set it up as a Primary Dependency and as a Consumer of a Qualified Dependency, as shown in Listing 5.

Listing 5: Decorator as Primary Dependency and Consumer of Qualified Dependency in Spring.

```
@Component @Primary       // is the primary dependency
public class DecoratorA implements DependencyA {
    @Autowired @Qualifier("Impl") DependencyA dep;   // invokes a qualified dependency
}

@Component @Qualifier("Impl") // qualified dependency
public class DependencyImpl implements DependencyA {}
```

In regards to specializing our Concrete Dependencies by their scopes, Spring uses **Singleton Dependencies** by default, though we can add the *@Scope("singleton")* annotation to make it explicit. For **Dependent Dependencies**, the annotation *@Scope("prototype")* must be added.

In a Web Application context, we can declare **Web Scoped Dependencies**, such as **Request Scoped Dependencies**, **Session Scoped Dependencies**, and **Application Scoped Dependencies** — in Spring, these specializations can be respectively indicated by the class annotations *@RequestScope*, *@SessionScope* and *@ApplicationScope*. All are shown in Listing 6.

Listing 6: Singleton, Dependent and Web Scoped Dependencies in Spring.

```
@Component @Scope("singleton")
public class DependencySing {}

@Component @Scope("prototype")
public class DependencyDep {}

@Component @RequestScope
public class DependencyReq {}

@Component @SessionScope
public class DependencySes {}

@Component @ApplicationScope
public class DependencyApp {}
```

We also demonstrate a practical application of our Ontology, capturing DI *code smells* such as *Over-Injection* — any combination of Constructor and Property Injection Points that results in over four dependencies being requested at initialization, a sign of violation of the Single Responsibility Principle — and *Concrete Class Injection* — a direct request to a Concrete Dependency by a Consumer Class, as it causes the loss of the flexibility brought by using abstractions [6]. In order to do that, we extended the operational version of OOC-O written in OWL by adding DepIn-O concepts, employed the ontology editor *Protégé*[2] and used DL Queries to capture the code smells. Such application is available as supplementary material, and, although simple, demonstrates that DepIn-O could be used to detect DI smells in real code bases in combination with, e.g., the OSCIN method [15].

5. Related Works

We searched the literature for works within the domain of Dependency Injection and ontology construction. However, to our knowledge, there are no works that instigate an intersection between DI and ontologies. We then broadened the scope of our search for works that explore the DI domain itself or the ontological representation involving the domain of software frameworks.

Concerning the fundamental concepts from the DI domain and their application on DI frameworks, [7] and [6] explain and explore the same subject as we did. The contrast between their works and ours is that their approach is mostly didactic, while we anchored our work on ontological foundations.

The Object/Relational Mapping Ontology (ORM-O) [16] shares similarities to our work regarding the ontological foundation — e.g., both use UFO as Foundational Ontology and add specializations to the OOC-O — and the methods employed — e.g., ORM-O was built following SABiO and DepIn-O was built following SABiO's newer extension SABiOx; also, both had their concepts presented in a Catalog of Concepts and instantiated in a select group of popular frameworks. The major difference between the two works comes to the domain: ORM-O deals

[2]https://protege.stanford.edu/

with Object/Relational Mapping whereas DepIn-O is concerned with Dependency Injection, still, both are contained within the larger Objected-Oriented Programming domain.

Related to the practical application of DepIn-O, the Dependency Injection For Web Services (DI4WS) [17] is a development model implemented in Java for the construction of Web Service applications. This approach involves mining an already existing code to find a list of candidate services for DI and refactoring the code by having the services (selected from the list by the developer) injected into the application. On the other hand, we are more interested in following an inverse path: with a solid domain ontology on DI, we aim to create tools that will support the developer to create applications with DI patterns implemented from the start.

The catalog of DI anti-patterns from [18] was compiled after the development of a static code analyzer tool that revealed the large presence of a number of DI anti-patterns in open-source projects and is meant to be used as a reference for developers to avoid them. Notably, this reference aligns with our research objectives, albeit through a distinct approach, as employing DepIn-O to provide tools for detecting code smells and anti-patterns is one of many practical applications of our ontology.

Another work that shares a degree of similarity to ours is [19], since its first step is to describe the Model-View-Controller (MVC) architectural pattern formally by using the description language in order to form the MVC architectural pattern ontology of the conceptual layer. Again, we differ in focus, since our work does not dive into the MVC pattern, but into the Dependency Injection domain, though there is a connection, as DI patterns can be implemented into MVC architecture in OO Programming Languages. Another difference is that our primary focus is not describing our model with a description language but with an ontological-level language and only as a later step to have it translated into a description language.

6. Conclusions

In this paper, we presented DepIn-O, an ontology on Dependency Injection software frameworks. DepIn-O was constructed in compliance with the phases laid out in a detailed ontology engineering method [2], including extensive research on sources of knowledge from the DI domain and analysis of some of its most used supporting frameworks from the most popular object-oriented (OO) languages.

The evaluation of the ontology through verification and validation was successful, as the requirements elicited through competency questions were answered and full coverage of the fundamental concepts of our Ontology was provided by instantiation. A simple practical application of DepIn-O was also illustrated.

The integration of DepIn-O with the Object-Oriented Code Ontology (OOC-O) also aims to add our ontology to a wider ongoing effort on the construction of a network of ontologies on various different subdomains of software development, contributing to it on the DI area.

For future works, we intend to further evaluate DepIn-O in practice, implementing a tool that automates tasks, e.g., code migration or smells detection, over source code bases that use DI frameworks. We also intend to use the ontology to propose improvements to the modeling language of FrameWeb [20], a Web Engineering method that incorporates the concepts of frameworks (including DI frameworks) to architectural models.

Acknowledgments

This research was conducted with the financial support of the Foundation for Support to Research and Innovation of Espírito Santo (Fundação de Amparo à Pesquisa e Inovação do Espírito Santo — FAPES).

References

[1] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, Boston, MA, USA, 1999.

[2] C. Z. de Aguiar, Interoperabilidade Semântica entre Códigos-Fonte baseada em Ontologia, Technical Report, Tese de Doutorado, Programa de Pós-Graduação em Informática. Universidade Federal do Espírito Santo, Vitória, ES, Brasil, 2021.

[3] G. Guizzardi, G. Wagner, J. Almeida, R. Guizzardi, Towards Ontological Foundations for Conceptual Modeling: The Unified Foundational Ontology (UFO) Story, Applied ontology 10 (2015). doi:10.3233/AO-150157.

[4] C. Z. de Aguiar, R. de Almeida Falbo, V. E. S. Souza, OOC-O: A Reference Ontology on Object-Oriented Code, in: A. H. F. Laender, B. Pernici, E. Lim, J. P. M. de Oliveira (Eds.), Conceptual Modeling - 38th International Conference, ER 2019, Salvador, Brazil, November 4-7, 2019, Proceedings, volume 11788 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 13–27. URL: https://doi.org/10.1007/978-3-030-33223-5_3. doi:10.1007/978-3-030-33223-5_3.

[5] G. Guizzardi, Ontological Foundations for Structural Conceptual Models, Ph.D. thesis, University of Twente, Enschede, The Netherlands, 2005.

[6] S. van Deursen, M. Seemann, Dependency Injection: Principles, Practices and Patterns, Manning, Shelter Island, NY, USA, 2019.

[7] M. Bojkic, D. Przulj, M. Stefanovc, S. Ristic, Usage of Dependency Injection within different frameworks, in: 19th International Symposium INFOTEH-JAHORINA (INFOTEH 2020), 2020, pp. 119–124.

[8] Airfocus, What is a dependency?, https://airfocus.com/glossary/what-is-a-dependency/, 2020. Accessed in October 1, 2021.

[9] E. Mnkandla, About Software Engineering Frameworks and Methodologies, in: Proc. of AFRICON 2009, 2009, pp. 1 – 5. doi:10.1109/AFRCON.2009.5308117.

[10] G. Guizzardi, Ontology, Ontologies and the "I" of FAIR, Data Intelligence 2 (2019) 181–191. doi:10.1162/dint_a_00040.

[11] R. A. Falbo, SABiO: Systematic Approach for Building Ontologies, in: Proc. of the 1st Joint Workshop ONTO.COM / ODISE on Ontologies in Conceptual Modeling and Information Systems Engineering, volume 1, CEUR, 2014, pp. 17–31.

[12] F. B. Ruy, R. d. A. Falbo, M. P. Barcellos, S. D. Costa, G. Guizzardi, SEON: A Software Engineering Ontology Network, in: Proc. of the 20th International Conference on Knowledge Engineering and Knowledge Management, Springer, 2016, pp. 527–542.

[13] B. B. Duarte, R. d. A. Falbo, G. Guizzardi, R. Guizzardi, V. E. S. Souza, An Ontological

Analysis of Software System Anomalies and their Associated Risks, Data & Knowledge Engineering 134 (2021) 101892. doi:10.1016/j.datak.2021.101892.

[14] Stack-Overflow, Stack Overflow Developer Survey 2022, https://survey.stackoverflow.co/2022/, 2022.

[15] C. Z. de Aguiar, F. L. Zanetti, V. E. S. Souza, Source Code Interoperability based on Ontology, in: Proc. of the 17th Brazilian Symposium on Information Systems, Uberlândia, MG, Brasil, 2021, pp. 1–8. doi:10.1145/3466933.3466951.

[16] F. L. Zanetti, C. Z. de Aguiar, V. E. S. Souza, Representação Ontológica de Frameworks de Mapeamento Objeto/Relacional, in: Proc. of the 12th Seminar on Ontology Research in Brazil (ONTOBRAS 2019), CEUR, Porto Alegre, RS, Brasil, 2019, pp. 1–12.

[17] M. Crasso, C. Mateos, A. Zunino, M. Campo, Empirically Assessing the Impact of DI on the Development of Web Service Applications, Journal of Web Engineering 9 (2010) 66–94.

[18] R. Laigner, D. Mendonça, A. Garcia, M. Kalinowski, Cataloging Dependency Injection Anti-Patterns in Software Systems, Technical Report, Version accepted at The Journal of Systems & Software, 2021. doi:10.48550/ARXIV.2109.04256.

[19] Y. Qiang, W. Lulu, L. Bixin, Identify MVC Architectural Pattern Based on Ontology, in: Proc. of the 31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019, 2019, pp. 612–617. doi:10.18293/SEKE2019-163.

[20] V. E. S. Souza, The FrameWeb Approach to Web Engineering: Past, Present and Future, in: J. P. A. Almeida, G. Guizzardi (Eds.), Engineering Ontologies and Ontologies for Engineering, 1 ed., NEMO, Vitória, ES, Brazil, 2020, pp. 100–124. URL: http://purl.org/nemo/celebratingfalbo.