

Co-Evolution of Java and Data-Oriented Programming: Practice Aspect

Felix Desyatirikov
Voronezh State University
Voronezh, Russia
felixdesyatirikov@gmail.com

Abstract—The article shows in practice that changes in Java and data-oriented programming (DOP) adhere to the principles of evolution. The development of DOP, from the foundational ideas introduced by Brian Goetz in version 1.0 to the refined mental model in version 1.1, is explored. A graph model of Java features within the DOP paradigm is proposed. Practical applications of DOP show how it makes code concise and secure.

Keywords—Java, Java-features, evolution, data-oriented programming (DOP)

I. INTRODUCTION

A programming paradigm is a fundamental approach to programming that shapes how solutions to problems are formulated and how programs are structured. It provides the concepts and techniques used to design, organize, and control the execution flow of programs. The emergence and development of any programming paradigms is driven by the need to address new types of problems, simplify programming tasks, and enhance software efficiency. Each paradigm introduces its own set of concepts and methods, with newer paradigms often incorporating elements of existing ones, either expanding upon or shifting the focus of earlier approaches.

Programming languages are also changing, evolve and adapt over time [1]. Researchers note that this change has much in common with the change in natural languages, which develop and evolve [2-5]. Evolution is understood as a natural process of change, while development is a controlled process of change. Table 1 shows the composition of the characteristics of the main processes of development and evolution. The difference between language evolution and language development lies in their direction, causes, mechanisms of changes [4, 6-8].

TABLE I. MAIN CHARACTERISTICS OF THE LANGUAGE CHANGE PROCESSES

Characteristics	Development	Evolution
Nature of Change	Refers to intentional, structured, targeted changes. It is a top-down process.	Refers to organic, often unplanned changes caused by external factors. It is a bottom-up process.
Directionality	Always goal-oriented, with a clear roadmap.	Irreversible, but may not have a clear direction. Adapts to current circumstances.
Timeframe	More immediate and happens within defined cycles.	A long-term process.
Participation	Directed by an authoritative group.	Driven by the collective actions of users.
Stability	Seeks stability and coherence.	Inherently unstable and adaptive process

The purpose of this article is to demonstrate in practice that the processes of changes in the Java language and data-oriented programming (DOP) should be viewed as evolution. For clarity, we chose DOP as a modern programming paradigm in Java that reveals the evolutionary potential of Java [9]. The object of our research is the evolutionary processes within programming. The subject focuses on the correlation between this evolution and the DOP paradigm evolution in the Java programming language, specifically through Java features that form the foundation of DOP. To achieve our objectives, we applied methods from System Analysis, structural modeling, and programmatic illustration.

II. DOP AS EVOLVING PARADIGM

Each new paradigm does not entirely replace its predecessors but often builds upon their principles. For instance, Object-Oriented Programming (OOP) incorporates concepts from Procedural Programming, while Reactive Programming draws from both Event-Driven, Object-Oriented and Functional approaches.

We intuitively define any paradigm as a set of principles and practices for solving specific tasks. Therefore a programming paradigm is a relatively high-level way of structuring and conceptualizing the implementation of a computer program. So, the main goal of any programming paradigm is to manage the growing complexity level. For Java, a relevant solution is OOP, which is one of the main approaches for solving complex tasks. OOP is effective in systems with extensive logic, numerous entities, and multiple integrations concentrated in one place—in short, in environments where clear boundaries are essential to prevent code from becoming unmanageable. However, large monolithic applications are increasingly becoming irrelevant, with a long-standing trend toward smaller, microservice-based architectures. In these more compact systems, the strict boundaries that OOP provides are often unnecessary. Instead, it would be beneficial to replace some of these boundaries, with more advanced functionality. And this is the starting point for the development of the new paradigm in Java – DOP [7, 10].

To identify this advanced functionality, let's consider a typical data processing flow in common application. At the beginning we deserialize our raw data into a data transfer object (DTO), then we map it onto a domain model, apply some business logic, and finally return the resulting DTO to the client. In sum, at every stage of processing, we're just converting data from one state to another. The key question, then, is how we transform the data at each stage.

In general, there are actually two different approaches for working with state: one for fixed behaviors and one for fixed data. These are known as the interface-oriented and dispatch-oriented approaches. At first time, the principles of the interface-oriented approach can be derived from its definition: the client interacts with the object through a contract defined by the interface. This approach is familiar to most developers; it is native to OOP and was the dominant option in Java until the release of Java 21.

However, there is an alternative mechanism for working with data (see Table 2). In the dispatch-oriented approach, the contract is defined simply by the type. Consequently, the client infers how to interact with an object primarily based on its type information. This approach can, of course, be combined with traditional polymorphism. This is precisely how DOP functions.

TABLE II. INTERFACE-BASED AND DISPATCH-BASED APPROACHES COMPARED

	Interface-based approach	Dispatch-based approach
Contract	Contract defines the ways how data can be processed	Contract defines type
Processing	Actor works with the data using it's contract	Actor works with type

Let's consider snippet of pseudocode:

```
interface Type
class TxData(txId: Long) extends Type
class IpData(ip: UInt) extends Type
fun process(dto: Type) {
  match (dto) {
    instanceof TxData -> {
      print("Tx: " + dto.txId)
    }
    instanceof IpData -> {
      print("Ip: " + dto.ip)
    }
    default -> print("???")
  }
}
```

This code represents common DOP-like implementation of data processing:

- We have a hierarchy of DTOs: the root is the Type and TxData and IpData are the inheritors.
- An object of unknown type (or with an optionally known upper bound) arrives to the function.
- We try to determine what logic should be performed on it using the operator that derives type information from it.

At first glance, it seems that this code violates several widely accepted design principles. Primarily, OOP is generally against using `instanceof` in any form. It's considered an antipattern because, from a pure OOP perspective, we should work only with the contract, not with its specific implementation. So this approach violates the principle of encapsulation and also impacts polymorphism. Additionally, such a function requires changes every time a new subclass is added. This aspect contradicts the Open-Closed Principle from SOLID. But, as it's often the case, first impressions can be deceiving. These "bugs" are actually features of DOP implementation in Java.

III. DOP IN EVOLVING JAVA

Brian Goetz defines DOP through four rules [10]:

1. Data is immutable.
2. Model the data, the whole data, and nothing but the data.
3. Make illegal states unrepresentable.
4. Validate at the boundary.

However, DOP has also a set of features that transfer these ideas into practice and characterize this approach as genuinely data-oriented. In practice, the DOP paradigm is built on two key concepts: algebraic data types and pattern matching. These concepts are implemented through the following features:

1. Algebraic Data Types
 - Records
 - Sealed Classes
2. Pattern Matching
 - Pattern Matching for instanceof
 - Pattern Matching for switch
 - Record Patterns
 - Unnamed Variables & Patterns
 - Primitive Types in Patterns, instanceof, and switch
 - Derived Record Creation

All of these features came from the Project Amber [11], which was originally created as a project for small features aimed at improving quality of life.

The four principles of DOP mentioned above were modified by Nikolay Parlog for DOP v.1.1 as follows [12]:

1. Model data immutably and transparently.
2. Model the data, the whole data, and nothing but the data.
3. Make illegal states unrepresentable.
4. Separate operations from data.

Segregating logic from data has long been considered as a best practice. And it is a valuable addition to the DOP paradigm too.

IV. SOFTWARE REFACTORING DUE JAVA-FEATURES OF DOP

Let's explore the possibilities of the DOP approach using the following basic example (see Fig. 1).

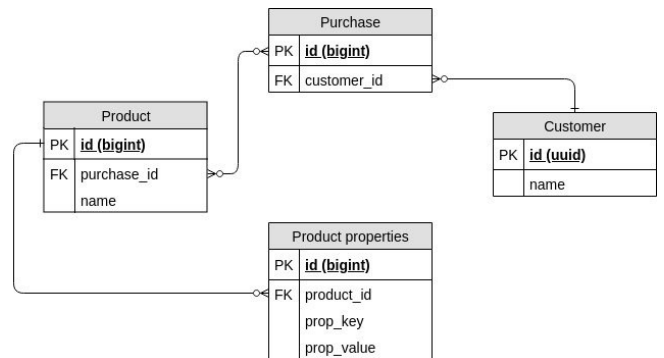


Fig. 1. Domain data structure.

In this domain, purchases are associated with products and customers, with each product having its own set of

properties. To see the principles of DOP in practice, let's implement an access checker algorithm for this domain. The logic is the follows: when a request for access to a specific object is received, we must verify access for all related entities. If access is granted throughout the entire hierarchy, then access to the root entity is also granted. For example, if a request to read a purchase is made, we need to check access for the purchase itself, the customer associated with that purchase, and the relevant products. Additionally, for each product, we verify property access, modeling domain relationships as a tree rooted in Purchase (Fig. 2).

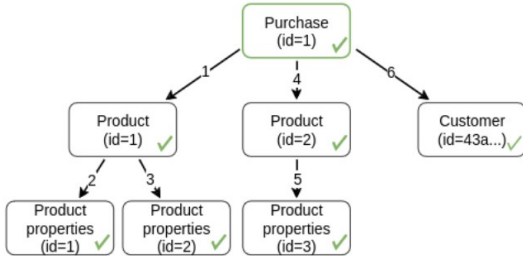


Fig. 2. Access check resulting with true.

In this example, the Purchase with id 1 has two associated products, and one customer. The first product has two properties, and the second product has one. So, on Fig. 2, we can see a successful access check. The numbers on the edges indicate the order in which nodes are traversed. On Fig. 3, there is access check process resulting in false. The actor has access to the root purchase, and to the product 2, but not to its properties. So, there is no access to the root.

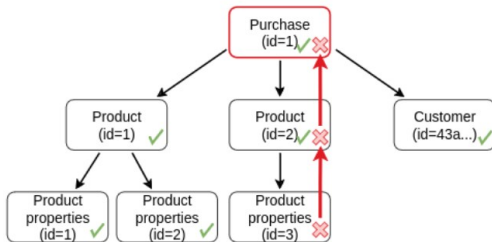


Fig. 3. Access check resulting with false.

Accordingly, the API consists of a single endpoint (see Fig. 4). The input includes a composite ID of the root entity and an action for which we need to grant or deny permission. This composite ID, referred to as the entity definition, includes a qualifier and an ID. For example, a product has an integer ID, while a customer has a UUID:

```
###
POST /composite-access-resolution/check
Content-Type: application/json
{
  "entityDefinition": {
    "qualifier": "PRODUCT",
    "id": 1
  },
  "actionType": "DELETE"
}
```

```
###
POST /composite-access-resolution/check
Content-Type: application/json
{
  "entityDefinition": {
    "qualifier": "CUSTOMER",
```

```
"id": "3fa85f64-5717-4562-b3fc-2c963f66afa6"
  },
  "actionType": "READ"
}
```

Fig. 4. Request listings.

We'll begin the design with the data, specifically using DTOs, as recommended by Brian Goetz. In this case, it's relatively straightforward: the input DTO contains two fields: the entity definition and the action type. The definition consists of a qualifier and an ID. In the domain model, we have four types of entities and two types of IDs: bigint and uuid.

So, we'll setup contracts for both entity definition types – LongId and Uuid and define classes that represent each domain entity (see Fig 5). Red interfaces here represent contracts that can be implemented by anyone and green are closed for inheritance.

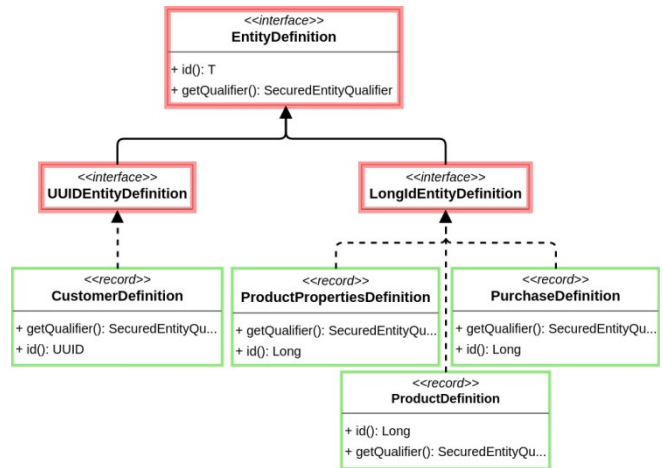


Fig. 5. Domain entity definition class diagram in Java 11.

With this hierarchy of DTOs, we adhere to the least principles of DOP:

1. Data is immutable.
2. Validate at the boundary.
3. Make illegal states unrepresentable.

Validation is performed through the constructor, and illegal states are prevented through polymorphism in the DTOs.

Now let's implement the permission service that performs the key functionality of the system. It is Fig.6 in Java 11:

```
public boolean checkAccess(AccessCheckRequest request) {
  // Build mapper
  var accessType = request.actionType();
  var authorities = SecurityContextHolder
    .getContext()
    .getAuthentication()
    .getAuthorities();
  var mapper = mapperResolver
    .getMapperByAccessTypeAndRoleSet(
      accessType,
      authorities
    );

  // Fetch entity
  var entityDefinition = request
    .entityDefinition();
```

```

var rootEntity = entityFetcher
    .findEntityByDefinition(
        entityDefinition
    )
    .orElseThrow();

// Process the entity tree
return entityWalkerService
    .processEntityTree(rootEntity, mapper)
    .stream()
    .allMatch(Predicate.isEqual(true));
}

```

Fig. 6. Access checker in Java 11.

Firstly, we build a mapper that determines access to a specific entity, based on the type of request and roles. Additionally, we fetch the root entity from the datasource. Then, we pass the root entity and the mapper to a service that traverses the tree and applies the mapper to each entity.

The task of fetching an entity can be solved in several ways. One of the most convenient of them is based on the registry pattern. To implement it, we need to define the contract of a strategy: the fetch method and a getter for a key. And the logic for fetching the entity will be delegated to the repository.

Commonly, strategies should be registered manually. But this may lead us to some troubles. By the way, using Spring, we can simply perform a self registration algorithm. Autowired annotation can mark any method for setter injection. So we pass our registry to this method and call the register method. The registry is a map with overridden 'put' and 'get' methods. The key is the subclass of EntityDefinition, and the value is the strategy.

Thus, in the access check service, we fetch the strategy from registry. And then call its fetch method with the ID of an entity. So we got the solution. It is shown below:

```

@Service
public class EntityByDefinitionRegistryFetcherImpl
{
    private final EntityFetcherStrategyRegistry
reg;
    @Override
    public <T> Optional<...>
findEntityByDefinition(
        EntityDefinition<T> definition
    ) {
        var fetchStrategy =
reg.getStrategy(definition);
        var id = definition.getId();
        return fetchStrategy.fetch(id);
    }
}

```

There are some demerits of this solution. The main issue is quite evident - we've created a lot of boilerplate for a simple task. We now have a map, an interface for the strategy, and some strategies, each of which is a delegate for repository. There are also security issues. New strategies will be added through copy-pasting. While we can guard against copying keys, we can't avoid accidental copying of logic. Since there will be many strategy files, there's a real chance of missing something. Tests can help us with this issue, but we now need to write tests for all of these abstractions. To summarize the classic implementation, it is

verbose. And the dispatch by entity type is implicit because our chosen approach is an interface-based one.

Lets' answer the question: why is it exactly so verbose? The answer is pretty simple. We have registry and a bunch of strategies. Yet do we really need whole of this code to solve a problem? And the answer is both "yes" and "no" at the same time. On one hand, yes, because the code works—it solves the problem at hand. On the other hand, no, because it also addresses hypothetical issues that are irrelevant if the goal is to keep the solution as simple as possible.

The main problem is dealing with possibly unlimited number of strategies. Those strategies can **be** loaded and actually **are** loaded at the runtime. And from this perspective all of this code is redundant because we have predefined set of strategies. So, for that task, we probably should try another approach - the DOP-based approach. The DOP implementation of entity fetcher is presented on Fig. 7. The logic is the follows: we are trying to determine specific type of an object by traversing the entire hierarchy, casting at each new level. If we do not find a match, we throw an exception.

```

Optional<? extends SecuredEntity> result;
if (entityDef instanceof LongIdEntityDefinition) {
    var longDef = (LongIdEntityDefinition)
entityDef;
    if (longDef instanceof ProductDefinition) {
        result =
productRepo.findById(longDef.getId());
    } else if (longDef instanceof
PurchaseDefinition) {
        result =
purchaseRepo.findById(longDef.getId());
    } else if (longDef instanceof
ProductPropertiesDef) {
        result = productPropertiesIntegration
        .findById(longDef.getId());
    } else throw new DomainException("Not
expected");
    } else if (entityDef instanceof
UUIDEntityDefinition) {
        var uuidDef = (UUIDEntityDefinition) entityDef;
        if (uuidDef instanceof CustomerDefinition) {
            result = customerRepo
        .findById(
            ((CustomerDefinition)
uuidDef).getId()
        );
        } else throw new DomainException("Not
expected");
    } else {
        throw new DomainException("Not expected");
    }
}
return result;

```

Fig. 7. Entity fetcher in Java 11.

This code has some issues. The primary issue is the 'instanceof'. We already discussed its drawbacks. The second issue is the use of exceptions at each level of hierarchy parsing. 'LongIdEntityDefinition' and 'UUIDEntityDefinition' are open for inheritance, and we need to handle not hardcoded subclasses. Additionally, casts significantly impact readability.

So, applying DOP in the context of Java 11 is possible **but** comes with the challenge of using an unlaconic implementation of type dispatch. This catalyzes the trend

toward efficiency, expressed in reducing code by simplification of language semantics. To leave these issues in the past, Java 17 introduced three features that simplify development in the DOP style: Sealed Contracts, Records and Pattern Matching in `instanceof`.

Let's start the refactoring with applying sealed contracts. This feature adds functionality for protecting hierarchies against uncontrolled inheritance. Now we can explicitly specify all permitted subclasses. Then let's add the sealed modifier to EntityDefinition and to the interfaces that extend it. And that is extremely important in case of explicit dispatch which we will see later. After that, we'll use the permits clause to register all subclasses. And since we're using Java 17, we should finish the refactoring with replacing classes with records where possible. Additionally, we'll rewrite constructors to use its compact versions. Updated entity definition is shown in Fig. 8.

```
sealed interface EntityDefinition<T>
    permits
        LongIdEntityDefinition,
        UUIDEntityDefinition {
    T id();
    SecuredEntityQualifier getQualifier();
}
sealed interface UUIDEntityDefinition extends
    EntityDefinition<UUID>
    permits CustomerDefinition { }
sealed interface LongIdEntityDefinition extends
    EntityDefinition<Long>
    permits
        ProductDefinition,
        ProductPropertiesDefinition,
        PurchaseDefinition { }
record ProductPropertiesDefinition(Long id)
    implements LongIdEntityDefinition {
    public ProductPropertiesDefinition {
        if (id <= 0)
            throw new IllegalArgumentException("id
> 0");
    }
    @Override
    public SecuredEntityQualifier getQualifier() {
        return PRODUCT_PROPERTIES;
    }
}
...
```

Fig. 8. Domain entity definition hierarchy in Java 17.

After all these changes, our hierarchy is finally sealed. Fig. 9 is a continuation of Fig. 5 that shows that all interfaces are green means that the whole structure is protected against uncontrolled inheritance.

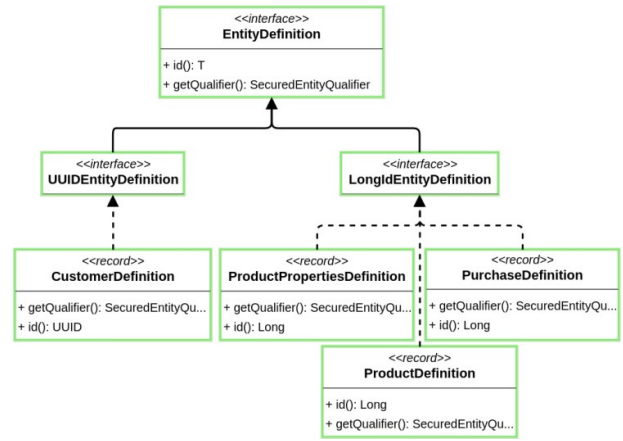


Fig. 9. Domain entity definition class diagram in Java 17.

The last feature in Java 17 related to DOP is pattern matching for `instanceof`. This is a shortcut for an `instanceof` check with a cast to checked type that improves readability. With this feature in mind we can perform a refactoring applying the pattern shown on Fig. 10.

```
// Java 11
if (entityDef instanceof LongIdEntityDefinition) {
    ...
} else if (entityDef instanceof
UUIDEntityDefinition) {
    var uuidDef = (UUIDEntityDefinition) entityDef;
    if (uuidDef instanceof CustomerDefinition) {
        result = customerRepo.findById(. . .);
    } else throw new DomainException("Not
expected");
} else {
    throw new DomainException("Not expected");
}

// Java 17
if (entityDef instanceof LongIdEntityDefinition
longDef) {
    ...
} else if (
    entityDef instanceof UUIDEntityDefinition
uuidDef
) {
    if (uuidDef instanceof CustomerDefinition def)
    {
        result = customerRepo.findById(def.id());
    } else {
        throw new DomainException("Absurdly
impossible");
    }
} else {
    throw new DomainException("Absurdly
impossible");
}
```

Fig. 10. Comparison of dispatch in Java 11 and Java 17.

But “exception” issue from Java 11-like approach is still not resolved. Moreover, in the case of sealed entity-definition hierarchy exception rising is pointless. All possible `instanceof` checks are already hardcoded. Yet `throw` instructions are required for compilation. And summing up for Java 17: it's possible to create inheritance hierarchies protected from unwanted inheritance, but the dispatch mechanism is still not as clear as needed.

If we build some kind of feature interdependencies graph, we can express our desires (see Fig. 5). Records and sealed types already work well together. Yet we'd like Pattern Matching for 'instanceof' and sealed types to integrate better (see Fig.11).

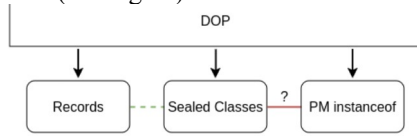


Fig. 11. Feature interdependencies graph for Java 17.

An important clarification is that these features (Sealed Contracts, Records and Pattern Matching in 'instanceof') add functionality to create algebraic data types (ADT). But ADT feature is convenient to use only with proper pattern matching support. And Java 21 adds this kind of support. It resolves the pointless exceptions issue and adds new functionality that simplifies the development in DOP style. In Java 17, Pattern Matching (PM) is still in its early stages. But in Java 21, the updated PM looks like this:

```
// Java 17
String match(Type type) {
    if (type instanceof TypeA a) {
        return "Its A: " + a;
    } else if (type instanceof TypeB b) {
        return "Its B: " + b;
    }
    throw new IllegalArgumentException("Unknown type");
}

// Java 21
String match(Type type) {
    return switch (type) {
        case TypeA a -> "Its A: " + a;
        case TypeB b -> "Its B: " + b;
    }
}
```

Dispatch through 'if' and 'instanceof' has been replaced with the 'switch' operator. Also this syntax allows us to handle subclasses with 'case' clauses. And this approach has some benefits against Java 17-like. Firstly, such a code is more readable than the 'instanceof-based' code because the 'switch' syntax is shorter. Secondly, with sealed types, we don't need to raise any exceptions or use a default branch. The compiler checks whether we have covered all possible subclasses of the sealed hierarchy at compile-time. And that is the strong point of using ADT hierarchies with switch-case pattern matching. If new classes are added to the sealed hierarchy and pattern matching is performed with 'switch' then it will be nearly impossible to forget to add new functionality to dispatch block of code. Thirdly, we have syntax for declaring a variable of the same type as the matcher, similar to 'instanceof'. Additionally, we have an ability to extract record attributes on the fly using a deconstruction pattern. One more advantage of this construction is the guard patterns. You can specify an additional matching condition directly in the 'case' expression with the 'when' keyword.

So, we've now covered two DOP-related features introduced in Java 21. Next, let's refactor our code with these features in mind. This is how fetching an entity would look. 'instanceof' has been replaced by a 'switch' with inline deconstruction:

```
Optional<? extends SecuredEntity>
findEntityByDefinition(
    EntityDefinition<?> entityDefinition
) {
    return switch (entityDef) {
        case LongIdEntityDefinition longDef -> {
            yield switch (longDef) {
                case ProductDefinition(Long id) ->
                    productRepo.findById(id);
                case . . .
            };
        }
        case CustomerDefinition(UUID id) ->
            customerRepo.findById(id);
    };
}
```

Then let's update our interdependencies diagram (Fig. 12).

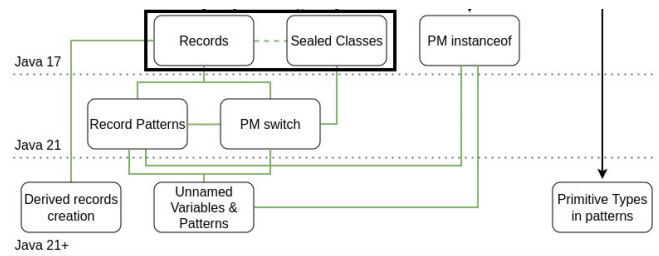


Fig. 12. Feature interdependencies graph for Java 21.

The law of systematicity is found in various layers and manifests itself both within each level and in their interaction with each other. We can see that all of these features align each other well. Records and sealed classes provide a solid foundation upon pattern matching is built. PM, in turn, allows us to use 'switch' for handling sealed hierarchies. And inline deconstruction adds further convenience. Yet, understanding the entire DOP feature set involves analyzing each layer, including features released after Java 21 or those still in development. Let's discuss these JEPs.

And we'll start with Unnamed Patterns. This JEP allows replacing names of unused variables in deconstruction to an underscore. However, this functionality is available only on Java version 22 and higher. There is the code of entity tree processor for Java 21, where unused variables still need to be named:

```
private Stream<T> processEntityTreeToStream(
    SecuredEntity entity,
    Function<SecuredEntity, T> mapper
) {
    var result = Stream.of(mapper.apply(entity));
    switch (entity) {
        case Purchase(
            var ignored,
            var products,
            var customerRef
        ) -> result = Stream.concat(. . .);
    }
}
```

```

    case Product(
        Long id,
        var ignored
    ) -> result = Stream.concat(. . .);
    case Customer ignored -> { . . . };
    case ProductProperties ignored ->
{ . . . };
}
return result;
}

```

In comparison with Java 21, where unused variables still need to be named, in Java 22 unused variables have been replaced with underscores, and the last two cases have been collapsed into one:

```

private Stream<T> processEntityTreeToStream(
    SecuredEntity entity,
    Function<SecuredEntity, T> mapper
) {
    var result = Stream.of(mapper.apply(entity));
    switch (entity) {
        case Purchase(
            var _,
            var products,
            var customerRef
        ) -> result = Stream.concat(. . .);
        case Product(
            Long id,
            var _
        ) -> result = Stream.concat(. . .);
        case Customer _, ProductProperties _ -> { .
. . };
    }
    return result;
}

```

It's essential to note that writing default branches is **not** recommended in such cases. With the default branch, adding a new subclass to a sealed contract, will **not** lead us to a compilation error. Consequently, there's a significant risk of forgetting to add logic to handle this new subclass.

Now, let's briefly cover the two least features: Derived Record Creation and Primitive Types in Patterns, instanceof, and switch. These features are now on their way to be released in LTS versions, so we'll keep our discussion brief. Derived Record Creation allows us to create a new instance of a record based on an existing one without modifying the original instance of course. The syntax is very straightforward: after the record typed variable, we can use the 'with' keyword, followed by the operations on its attributes.

Now according to Fig. 12 let's explore the last feature in our list - Primitive Types in Patterns, instanceof, and switch. The motivation behind this feature is the following:

- Enable uniform data exploration by allowing type patterns for all types, whether primitive or reference.
- Align type patterns with instanceof, and align instanceof with safe casting.
- Allow pattern matching to use primitive type patterns in both nested and top-level contexts.
- Provide easy-to-use constructs that eliminate the risk of losing information due to unsafe casts.
- Following the enhancements to switch in Java 5 (enum switch) and Java 7 (string switch), allow switch to process values of any primitive type.

This feature will significantly improve the ease of use of DOP in case of working with primitive types. Java.

V. COMMON CASES OF APPLYING DOP IN PRACTICE

So, we've already checked all Java features related to DOP and explored how they provide a solid foundation for designing modern applications. A key strength of DOP lies in its ability to handle state manipulation with fixed data structures. This makes it an excellent fit for scenarios where the flow of information and state transitions are paramount. For example:

- **State Machines:** DOP's focus on algebraic data types and pattern matching simplifies the representation and processing of finite states and their transitions.
- **CQRS (Command Query Responsibility Segregation):** DOP naturally supports the separation of concerns by enabling immutable command objects and concise query handling logic through pattern matching [13].
- **Stream Processing:** The structured nature of DOP aligns with the immutable and declarative approach often used in processing data streams.

By leveraging DOP in these areas, you can create robust, readable, and maintainable systems while fully utilizing the latest Java language features.

VI. CO-EVOLUTION STATEMENT

Java evolves similarly to a natural language, continuously adapting and growing. The DOP paradigm is also evolving and is built upon three key features: Records, Sealed Classes, and Pattern Matching. Together, these features make code more concise, safer, and less complex. Table 3 shows the main characteristics of the language change processes and their practical manifestation in terms of the development and evolution of the Java language.

TABLE III. MAIN CHARACTERISTICS FOR THE JAVA CHANGES

Characteristics	Development	Evolution
Nature of Change	Planned additions, such as JDK enhancements in line with JEPs (Java Enhancement Proposals), like introducing sealed classes or records to address specific developer needs	Evolution can be seen in how developers naturally push for patterns or practices that eventually become formalized. For example, <i>Optional</i> became widely used to avoid <i>null</i> long before the introduction of features like <i>Pattern Matching</i> for null checks.
Directionality	The Project Amber roadmap explicitly introduced features to simplify everyday programming tasks.	Evolution occurs as best practices emerge, like the rise of dependency injection frameworks that influenced modern Java design
Timeframe	Feature releases now follow a biannual schedule	The language evolves through trends, such as the growing use of reactive programming
Participation	Changes proposed and implemented by Oracle and the Java Community Process (JCP)	Developers influence the language through community-driven libraries and frameworks
Stability	Seeks stability and coherence, even if it introduces changes	Inherently unstable and adaptive, accommodating external influences

It is obvious that both processes are observed in Java - evolution sets the long-term background, and development accelerates the solution of current problems.

VII. CONCLUSIONS

Java and DOP are changing according to the principles of evolution. We systematically observe a series of conditioned transformations of the language, going in a certain direction, which is set by the DOP paradigm. What is characteristic, these changes are irreversible and obey the rules of economy. Economy is manifested in two aspects. Firstly, the syntax of the language itself changes. Secondly, a more effective toolkit for solving applied problems appears. Due to the irreversibility of these changes, their regularity and positivity, we are talking about the evolution of Java.

REFERENCES

- [1] J. Gosling and H. McGilton, "The Java language Environment: A white paper," Sun Microsystems, 1996. URL: https://www.stroustrup.com/1995_Java_whitepaper.pdf
- [2] R. D. Elbourn and W. H. Ware, "The Evolution of Concepts and Languages of Computing," in Proc. of the IRE, vol. 50, iss.5, May 1962, pp. 1059 – 1066. DOI: 10.1109/JRPROC.1962.288006
- [3] G. Neubig and M. Allamanis, "Modeling Natural Language, Program and their Intesection," in Proc. of the 2018 Conf. of the North American Chapter of the Association for Computational Linguistics: Tutorial Abstracts, pp. 1–3, New Orleans, Louisiana. Association for Computational Linguistics. DOI: 10.18653/v1/N18-6001
- [4] B. Goetz and N. Parlog, "AMA About the Java Language," Philly ETE 2021, URL: <https://www.youtube.com/watch?v=9si7gK94gLo>
- [5] J. Nölle, S. Hartmann, and P. Tinitis, "Language evolution research in the year 2020: A survey of new directions," *Language Dynamics and Change*, 10(1), 3-26. URL: <https://doi.org/10.1163/22105832-bja100054>
- [6] N. Parlog, "Java's evolution into 2022: The stateof four big initiatives," *Java Magazine*, Feb 19, 2022. URL: <https://blogs.oracle.com/javamagazine/post/java-jdk-18-evolution-valhalla-panama-loom-amber>
- [7] B. Goetz, "Java 22 ... and beyond," in JAX, May, 2024. URL: <https://www.youtube.com/watch?v=YBGVK5JuSJB>
- [8] F. :Desyatnikov, "Dynamic Model of the Java Ecosystem Hype Cycle," in Proc. of the 2024 IEEE EICon. DOI: 10.1109/EIConRus.2018.8317332 IEEE 2024 EICon. URL
- [9] N. Parlog, NIPAFX, "All Java 23 Features – Inside Java. Newscast #70", Jun 6, 2024. URL: <https://nipafx.dev/inside-java-newscast-70/>
- [10] B. Goetz, "Data Oriented Programming in Java," Jun 20, 2022. URL: <https://www.infoq.com/articles/data-oriented-programming-java/>
- [11] N. Parlog, "Data Oriented Programming in Java – Version 1.1," May 23, 2024. URL: <https://www.inside.java/2024/05/23/dop-v1-1-introduction/>
- [12] Project Amber, URL: <https://openjdk.org/projects/amber/>
- [13] S. Martinelli, "CQRS und DOP mit modernem Java", *Java Aktuell*, Vol. 4, 2024, pp.70-74.