Covariant Conversions (CoCo): A Design Pattern for Type-Safe Modular Software Evolution in Object-Oriented Systems

Jan Bessai
Technische Universität Dortmund, Germany

George T. Heineman
Worcester Polytechnic Institute, MA, USA

Boris Düdder
University of Copenhagen, Denmark

Abstract

Software evolution is an essential challenge for all software engineers, typically addressed solely using code versioning systems and language-specific code analysis tools. Most versioning systems view the evolution of a system as a directed acyclic graph of steps, with independent branches that could be merged. What these systems fail to provide is the ability to ensure stable APIs or that each subsequent evolution represents a cohesive extension yielding a valid system. Modular software evolution ensures that APIs remain stable, which is achieved by ensuring that only additional methods, fields, and data types are added, while treating existing modules through blackbox interfaces. Even with these restrictions, it must be possible to add new variations, fields, and methods without extensive duplication of prior module code. In contrast to most literature, our focus is on ensuring modular software evolution using mainstream object-oriented programming languages, instead of resorting to novel language extensions. We present a novel CoCo design pattern that supports type-safe covariantly overridden convert methods to transform earlier data type instances into their newest evolutionary representation to access operations that had been added later. CoCo supports both binary methods and producer methods. We validate and contrast our approach using a well-known compiler construction case study that other researchers have also investigated for modular evolution. Our resulting implementation relies on less boilerplate code, is completely type-safe, and allows clients to use normal object-oriented calling conventions. We also compare CoCo with existing approaches to the Expression Problem. We conclude by discussing how CoCo could change the direction of currently proposed Java language extensions to support closed-world assumptions about data types, as borrowed from functional programming.

2012 ACM Subject Classification Software and its engineering → Software evolution; Software and its engineering → Design patterns; Software and its engineering → Abstraction, modeling and modularity

Keywords and phrases Expression problem, software evolution, type safety, producer method, binary method

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.4

Supplementary Material *Software (ECOOP 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.2.4`
Software: `http://doi.org/10.5281/zenodo.4756838` [2]

Acknowledgements We would like to thank the reviewers of earlier versions of this paper for their carefully thought out, detailed reviews, as well as the many constructive remarks. They helped to improve our presentation, the artifacts, and the pattern drastically. Special thanks go to the reviewer who suggested to mitigate "parameterization boilerplate" with type members.

© Jan Bessai, George T. Heineman, and Boris Düdder;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

This paper presents a novel solution to the well-known Expression Problem (EP) [29], a research problem common to the fields of programming language design, multi-dimensional product line design, and software engineering. EP offers a concise representation of the challenge in implementing a system that evolves over time. The goal is to enable additive modular software evolution for data types and their methods.

The research community has identified a number of mandatory qualities that any approach must satisfy [27, 31]:

- It must be possible to add new variations, attributes, and methods to data types without changing existing software modules.
- Evolved modules must not duplicate the code of prior modules, so potential errors can be found and fixed locally.
- Hierarchical grouping of data types is an important feature of object-oriented programming that must remain intact. When a method implementation is the same for a subset of the hierarchy, it must not be necessary to repeat its definition.
- Modules must be able to evolve concurrently in branches to reflect the fact that independent features can be developed independently, possibly by different developers. It must be possible to merge these branches later so that work does not need to be duplicated.

Large systems must be able to deal with evolving dependencies without the need to rewrite existing code. This justifies the initial requirement of additive evolution, where data types and methods are considered incrementally without breaking APIs by refactoring, for example, by removing or renaming their components. Even as the software system evolves, developers must still be able to rely on compile-time checks to ensure completeness of evolutionary steps; in particular, static type-safety shall guarantee that methods are declared for all data type variants of their domain. Missing implementations shall be reported in a human-readable way at compile-time.

Solutions that apply to existing programming languages are preferable because they may provide immediate benefit to existing systems. Language extensions and new programming languages typically require years to manifest themselves in practice. They also often require rewrites of entire projects, which risk the well-known second-system effect [15]. This is also true of solutions which require code that diverges from the idiomatic use of its programming language, for example, by creating an embedded domain-specific language. There must be no restriction on the form of methods which can be added. There must be no exclusion of binary methods (methods that take data types of the evolving domain as input) or producer methods (methods which produce instances of data types in the evolving domain).

As presented in Section 2, the CoCo design pattern:

- Enables method signatures (the API) of domain logic to be stable, even when the types used in them evolve. This is facilitated by a conversion method that grants access to evolved APIs of types. It is covariantly overridden (hence the name Covariant Conversions) during the evolution process and integrated into a pattern of factories and delayed instantiation that allows implementing it safely.
- Operates fully within the constraints described above, as validated in several case studies in Section 3.
- Supports hierarchical grouping with method-deduplication [33].
- Relies only on inheritance, interfaces with default methods, and parametric polymorphism (without type bounds), which are common features of mainstream object-oriented languages [33], such as Java, Scala, and C#.
- Allows programmers to write idiomatic code in object-oriented languages to construct objects and invoke operations with method calls [30].

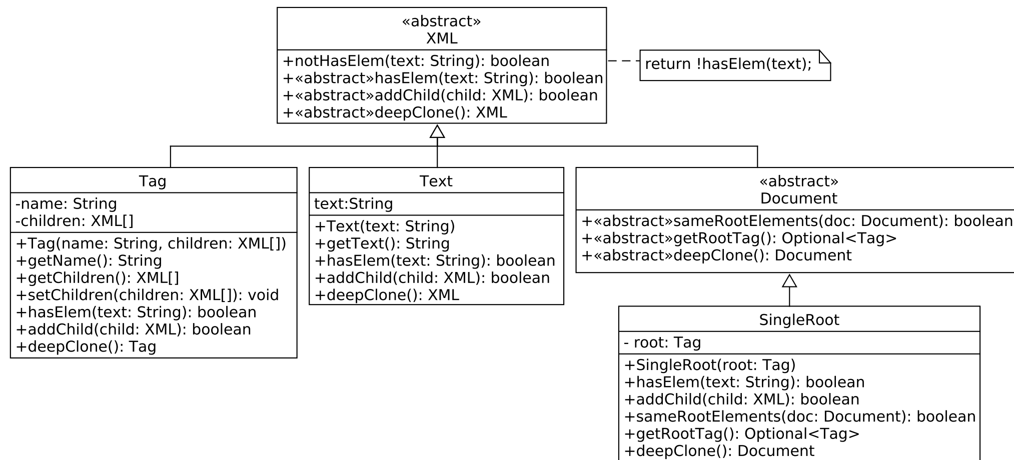

Figure 1 Example modeling XML components with multiple hierarchical classes, binary methods addChild, setChildren, sameRootElements and producer methods deepClone, getChildren, and getRootTag.

- Enables mergeable evolutionary branches [31], where developer teams can independently work on extensions that can be merged without changing or recompiling any existing code. This capability is a notable extension to EP [29] and guarantees future reusability within the code base.

Section 4 discusses how the CoCo design pattern takes a unique position in the known design space of possible solutions, where related approaches do not provide a solution to all extended constraints of EP as described above. Section 5 concludes with some remarks on pathways to broader adoption of CoCo.

2 Design Pattern

Let us consider a basic class hierarchy shown in Figure 1 to see why the CoCo design pattern is useful and how it would be applied. Assume that we want to design some classes to model XML data. We restrict ourselves to a subset of the possibilities of XML, which is interesting for demonstrating the various EP aspects discussed earlier. Figure 2 illustrates the intended use of the classes in Figure 1 with a simple XML document and its representation by a newly constructed object tree. An abstract base class **XML** is extended by classes **Tag** and **Text** as well as another abstract subdomain base class **Document**, which has a single subclass **SingleRoot**. The **XML** base class defines operations that all XML elements support. Method hasElem searches for a desired text in an **XML** element and its children. Calling hasElem on the example document from Figure 2 would return `true` for arguments `"CoCo"`, and `"relatedPattern"`, but `false` for `"Visitor"` or `"related"`. As a convenience, the default-implemented method notHasElem confirms that the desired text is not present.

Method deepClone recursively copies an entire **XML** element into a new object structure. The addChild method tries to add a given element to become a child node of an **XML** element. This might not always work and so addChild returns a boolean to indicate success. The simplest element, represented by class **Text**, represents plain text in a document. Therefore its addChild implementation does nothing and returns `false`. Objects of class **Tag** have a name and an array of children. Their addChild method appends the given element to that array and returns `true`. In contrast to **Text**, where operations work locally, the

```
<designPattern>
  <name>CoCo</name>
  <relatedPattern>Factory</relatedPattern>
</designPattern>
```

```
  new SingleRoot(
    new Tag(
      "designPattern",
        new XML[] {
          new Tag("name", new XML[] { new Text("CoCo") }),
          new Tag("relatedPattern", new XML[] { new Text("Factory") })
        }
     )
  );
```

Figure 2 Example XML document (top) and Java code for its construction (bottom) based on the classes shown in Figure 1.

hasElem and deepClone methods of **Tag** recursively call the appropriate methods of the XML elements known via the children array. Class **SingleRoot** is a special case of abstract class **Document**, where the topmost element is a single object of class **Tag**. Operations from **XML** are implemented performing recursive calls to the child root. Method getRootTag of **SingleRoot** returns the single root tag injected into an **Optional** wrapper class. The intended semantics of the parent method signature in **Document** is that getRootTag can be a partial method returning an empty **Optional** value for some possible implementations. Finally, sameRootElements in **SingleRoot** checks if the name of the current root tag matches the name of the root tag from the other document. This check uses getRootTag on the argument, checking if the partial result is present before comparing with the getName result.

Note that classes **XML**, **Tag**, and **Document** occur as types of parameters in addChild, setChildren, and sameRootElements. These methods are *binary methods* [4] because they involve the object on which they are called (`this`) and their parameter is also an object of a type present in the inheritance hierarchy. Methods deepClone, getChildren, and getRootTag are *producer methods* because their result is an instance produced from a type present in the inheritance hierarchy. Constructors are producer methods, and sometimes (in **Tag** and **SingleRoot**) binary methods. The return type of deepClone is *covariantly overridden* (i.e., safely replaced by a compatible subclass) in **Tag** and **Document** to enable recursive implementations that can pass cloned elements to other binary methods.

Extending the class hierarchy at any point with a new class is easy and can be done without recompiling or modifying existing code. This is, after all, the main modularity benefit of class-oriented programming. However, inserting a new method is problematic because it has to be inserted in a class, which needs to be recompiled and will cause a recursive recompilation of all sub-classes. Even worse, if the method is abstract or requires different implementations in some sub-classes, multiple classes have to change. In scenarios where the hierarchy is part of a library developed and distributed by a third-party, this is problematic and might even be impossible when the distribution is under a closed-source license.

A flawed but informative attempt to fix the issue is shown in Figure 3. Here, new classes (prefixed with E) are naively inserted to contain a new binary method validate and a new producer method asTag. Method validate is intended to use the current element as a schema to validate the given XML tag, while asTag checks if the current XML element is a tag and returns it as such, if possible. The new classes mirror the old ones, extending each of them together with their new parent. First of all, this is impossible in languages without multi-inheritance (e.g., Java). Additionally, the interplay between the new binary method validate and the old producer methods is fundamentally broken: method validate requires an

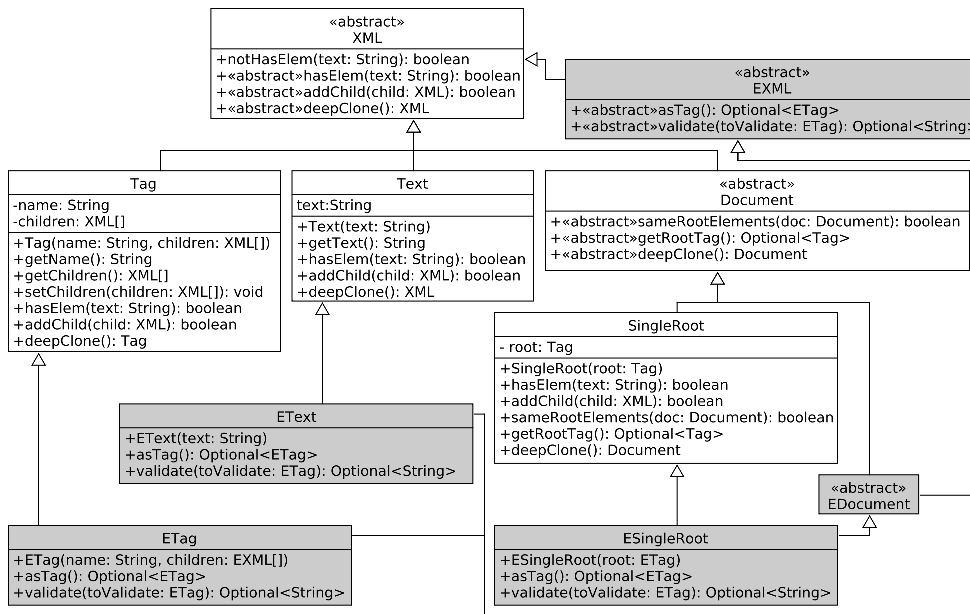

Figure 3 Flawed attempt to extend class hierarchy with a new binary method validate and a new producer method asTag provided in the extended interfaces **EXML**, **EDocument**, **ETag**, **EText**, and **ESingleRoot** (shaded background).

object of type **ETag**, while the producer methods provide objects of the old types (without an E-prefix) that were present before the extension. Wang and Oliveira [30] propose to solve the problem of multi-inheritance by turning the classes into a hierarchy of interfaces, which are implemented by some final classes that provide the code for getters, setters, and constructors. Their solution covariantly overrides all producer methods (including getters) whenever an evolution needs to add a method to the class hierarchy. In Figure 3, for example, **EXML** would become an interface with an abstract override of method deepClone(): XML to deepClone(): EXML. This trivial solution (at first glance) to the expression problem results in more problems upon closer inspection. One problem is that mutable attributes no longer work. Setters, as a special case of binary methods, cannot override their parameter type to evolved versions because method parameters are contravariant, that is, they need to be less specialized or remain the same with each inheritance step. In the example, the setChildren setter in class **Tag** would require an array of **EXML** which is more specialized than **XML**. Wang and Oliveira [30] propose to fix this by adding generic parameters with type bounds that abstract over any domain type reference. Their example remains incomplete because it does not show an extension with binary and producer methods *after these type bounds have been introduced*. In practice, the type bounds break inheritance because they covariantly modify the contravariant type of parameters of binary methods. They also break producer methods (such as deepClone) that need to construct instances of the evolved types, but invoke earlier constructors in their implementations. This forces producer code to be duplicated and modified with each evolution, violating the requirement not to duplicate code.

The CoCo design pattern eliminates the aforementioned issues by allowing references to the most generic (earliest) class type everywhere, providing abstract factory methods for constructors and conversion methods to convert earlier instances to later versions. Figure 4 shows the base class hierarchy implemented using the CoCo design pattern. Java and

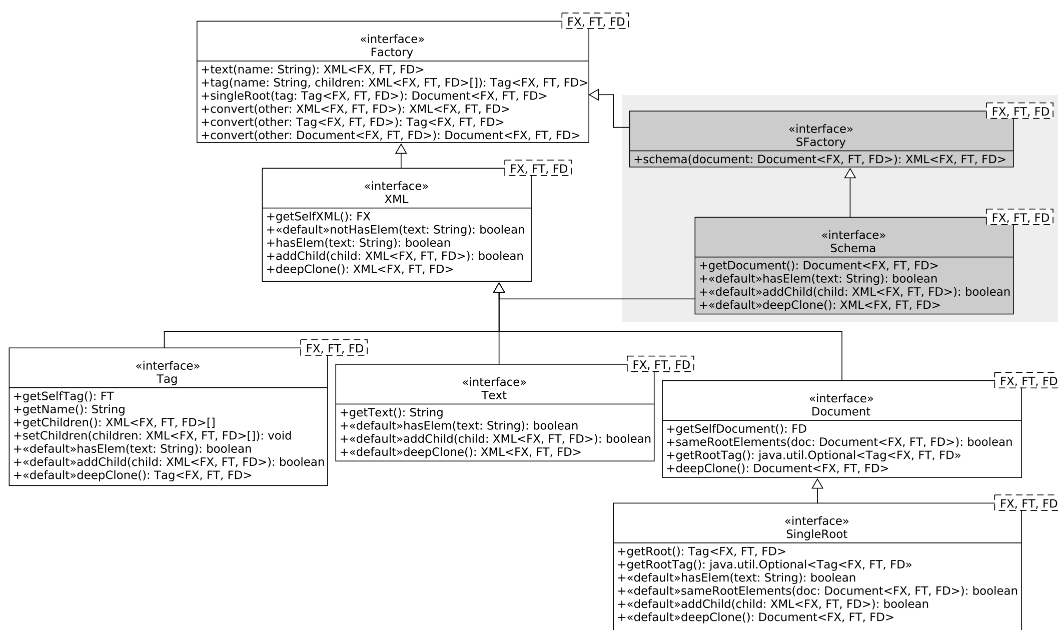

Figure 4 Hierarchy of Figure 1 implemented using the CoCo design pattern. Extension with a new data type **Schema** and factory shown in the tinted box. Methods are abstract in interfaces unless they have default implementations which are stereotyped by `«default»`.

Scala code for Figure 4 and the following figures throughout this section is available in the accompanying artifacts. As with the solution by Wang and Oliveira [30], all classes become interfaces to avoid issues with multi-inheritance. Methods are placed, as before, with implementations provided as default implementations (a feature available in Java, C#, and other mainstream OO languages). An additional **Factory** interface is introduced with abstract methods corresponding to each of the constructors of the naive object-oriented solution from Figure 1. Note that all interfaces are parameterized for domain types mentioned in the signature of methods. This parameterization allows delaying specifying which types will be finally used. A convert method is added to the factory for each parameterized type. Once an evolution is added, these convert methods will be covariantly overridden to refine their results to the latest evolutions of the converted classes. We choose to use inheritance from the **Factory** interface to make conversions and factory methods available in all parts of the type hierarchy. Conversions can be implemented in the next step, which will use the additional getSelf methods in each of the convertible interfaces. These methods are only necessary if a type is convertible (i.e., mentioned in a signature) and so the interfaces **XML**, **Document**, and **Tag** require new getSelf methods, while **Text** and **SingleRoot** can inherit them. Figure 4 also shows that the basic object-oriented feature to add new data types is not affected. The tinted box in the upper right contains a new data type **Schema**, which can be added independently of any code that was present before. Instances are instantiated by a newly-added extended Factory, **SFactory**. These additions appear in a new compilation unit without changing existing code. Unlike the Visitor pattern, CoCo does not compromise the advantage of object-oriented programming, namely, being able to freely add data types.

Figure 5 shows how the interfaces from Figure 4 are implemented by classes to allow instantiation. The implementation is fairly trivial, adding a finalized component (prefixed with F) for every component of the original class diagram. All getSelf methods are implemented by returning the current object (`this`). The convert methods of the final factories simply

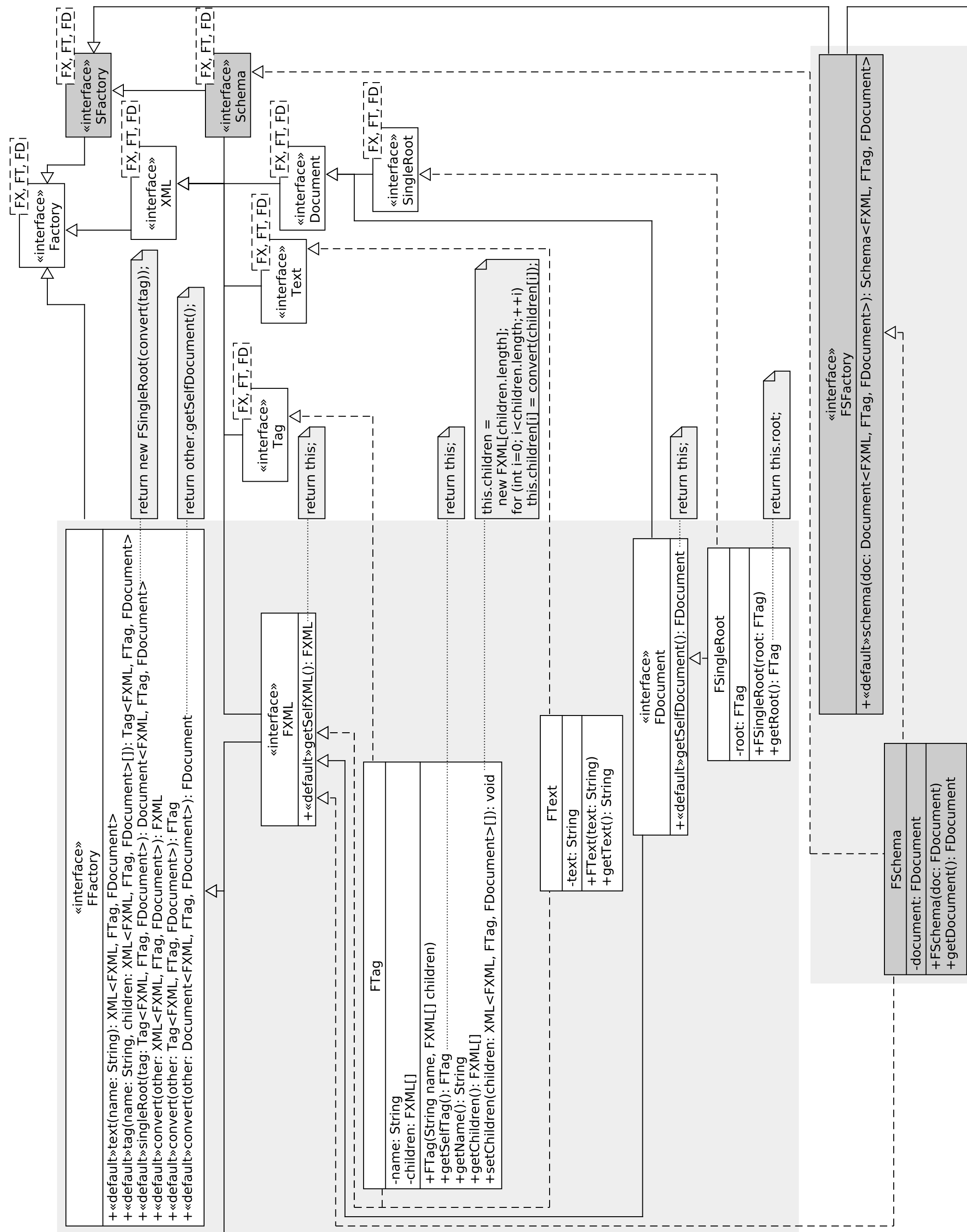

Figure 5 Final layers to instantiate the interfaces from Figure 4. The tinted upper box contains the final layer of the initial diagram, while the lower tinted box contains the separate finalized layer for the extension with **Schema**. Data types of the extension are filled in gray color. Comments in the center column show that only trivial getter, setter, constructor, and conversion code is added, which does not contain business logic or unsafe casts.

dispatch to the getSelf method of their argument. None of these methods need any casts, because in the final layer, generics parameters for the return type of getSelf are instantiated to the finalized types which inherit from the domain types at the most recent level. Delaying the implementation of convert and instantiation of generics until this moment enables convert methods (and thereby the pattern) to ensure stability of the domain logic methods, while their implementations can convert to the latest known evolution. Instantiation in factory methods is forwarded to constructor calls. Final classes (**FTag**, **FText**, **FSingleRoot**, **FSchema**) add constructors, getters, and setters with a field for each attribute reachable via the get-set-protocol prescribed by their interface. Note how the newly created layer contains no additional domain logic. In principle, a compiler extension or code generator could automatically produce it, and modern languages, such as Scala, allow to implement this layer with very few lines of code. Figure 5 already includes a (separate) final layer for the addition of the data type **Schema**.

Figure 6 shows a modular evolution that adds the binary method, validate, and the producer method, asTag, which previously failed in the naive approach. The method is added into a new extended domain interface **EXML**. Its parameter is typed by the *earliest* version of the domain interface **Tag** and does not need to evolve any further. This is possible because the implementation of validate has access to convert inherited from the extended factory interface **EFactory**, which allows to safely transform the parameter into an **ETag** instance. The *convert* methods are *covariantly* overridden with a refined result type (e.g., **EXML** instead of **XML**), which gives the CoCo design pattern its name.

Method validate is placed in the **EXML** interface and returns an optional error message if validation fails. If an element is not suitable to validate the given tag, it can return an error message. This behavior serves as a default implementation in **EXML** and is inherited in **ETag**, **EText**, **EDocument**, and **ESingleRoot**. Sharing default behavior for the general case in base classes is crucial for real world usability, which is discussed at length in [33]. Alternatively, we could have opened up **Schema** for extension by introducing it with a generic parameter, a getSelfSchema method, and a convert in **SFactory**, or we could do the same for the new type **ESchema**. In both cases validate would be available in **ESchema**, which in the first case could be obtained from any **Schema** and in the second case would serve as its own subdomain base class[1].

Implementing validate in **ESchema** is possible by recursively constructing new subschemata for the children of the document represented by the current schema. This is possible with the producer methods of the classes (for accessing children) and the factory methods from **EFactory**. Factory-produced instances are usable at the current level, because they can again be converted. The new producer method toTag is there to check if elements of the document are tags to recursively validate. It is defined in **EXML** and only overridden by the class **ETag**. Though not required in the example, extended interfaces could also choose to override an existing method introduced by an earlier extension, or require new getters and setters from their implementing final classes. Developers can remedy bad design choices (e.g., because new methods and data types can implement operations more efficiently) or add more fields to the domain data types. Since all extensions are provided in interfaces, even mainstream object-oriented languages such as Java and C# allow merging multiple domain evolutions by using multi-inheritance. In the example, this would result in **E**-prefixed classes, each extending multiple prior versions. Extended domain data types can then supply any

[1] The interested reader may find such domain extensions in the accompanying code for the TAPL case study discussed later

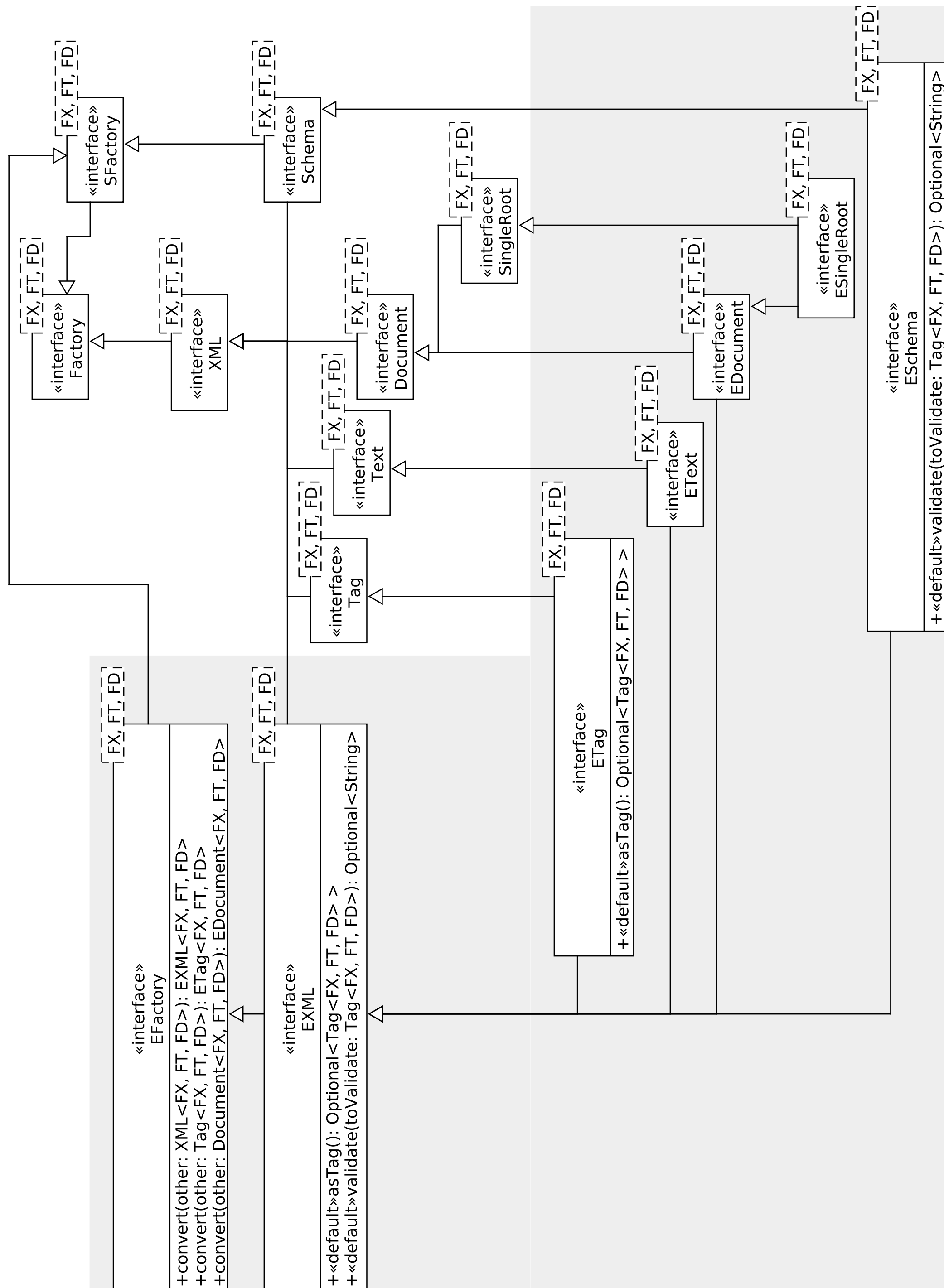

Figure 6 Extension with a new binary method toValidate and a producer method asTag. New components are placed in the tinted box. With convert, binary methods can refer to the old (less general) domain types instead of the new **E**-prefixed types, while avoiding contravariance issues from overriding parameters.

missing implementations for pairs of types and methods, where the type is present in one branch, while the method is required in interfaces from a different branch. The next section presents a case study on the traditional Expression Problem domain, illustrating why such a merge may be useful.

The class hierarchy for implementations of the extended layer from Figure 6 is shown in Figure 7. This time the extended interfaces need to be instantiated. Since extended interfaces are derived from the earlier versions, their final instantiations in **FEFactory**, **FEXML**, **FEDocument**, **FETag**, **FEText**, **FESingleRoot**, and **FESchema** remain compatible with any code that is written to work with their respective earlier versions (such as **Factory<FX,FT,FD>** or **XML<FX,FT,FD>** as long as the parameters **FX**, **FT**, **FD** remain abstract by the client. In any case, whether fixing a final version or abstractly working with its interfaces, clients can directly call methods without using visitors or object algebras. Clients also have access to convert methods via factories to ensure that objects created from producer methods provide the latest API required by the client. The final implementations are similar to those in Figure 5, but they do not contain non-trivial or domain-specific replicated code because all they do is add trivial getter, setter, constructor, and conversion methods.

```
public class Client {
 static class ClientM0<FX,FT,FD> {
  private final Factory<FX,FT,FD> factory;
  final Document<FX,FT,FD> demoDoc;
  public ClientM0(Factory<FX,FT,FD> factory) {
   this.factory = factory;
   this.demoDoc =
    factory.singleRoot(
     factory.tag("designPattern",
      factory.tag("name", factory.text("CoCo")),
      factory.tag("relatedPattern", factory.text("Factory"))));
  }
  public void run() {
   System.out.println("Has␣CoCo:␣" + demoDoc.hasElem("CoCo"));
   System.out.println("Has␣relatedPattern:␣" + demoDoc.hasElem("relatedPattern"));
   System.out.println("Has␣Visitor:␣" + demoDoc.hasElem("Visitor"));
   System.out.println("Has␣related:␣" + demoDoc.hasElem("related"));
  }
 }
 static class ClientM2<FX,FT,FD> {
  private final EFactory<FX,FT,FD> factory;
  private final ClientM0<FX,FT,FD> collaborator;
  final XML<FX,FT,FD> schema;
  public ClientM2(EFactory<FX,FT,FD> factory, ClientM0<FX,FT,FD> collaborator) {
   this.factory = factory;
   this.collaborator = collaborator;
   this.schema =
    factory.schema(factory.singleRoot(
     factory.tag("designPattern",
      factory.tag("name"),
      factory.tag("relatedPattern"))));
  }
  public void run() {
   collaborator.run();
   Optional<Tag<FX,FT,FD>> root = collaborator.demoDoc.getRootTag();
   if (root.isEmpty()) { return; }
   Optional<String> isValid = factory.convert(schema).validate(root.get());
   System.out.println("Errors:␣" + isValid.toString());
  }
 }
 public static void main(String[] args) {
  EFactory<FEXML, FETag, FEDocument> factory = new FEFactory() {};
  ClientM0<FEXML, FETag, FEDocument> client = new ClientM0<>(factory);
  ClientM2<FEXML, FETag, FEDocument> evolved = new ClientM2<>(factory, client);
  evolved.run();
 }
}
```

Listing 1 Stand alone example for evolving client code using Factories

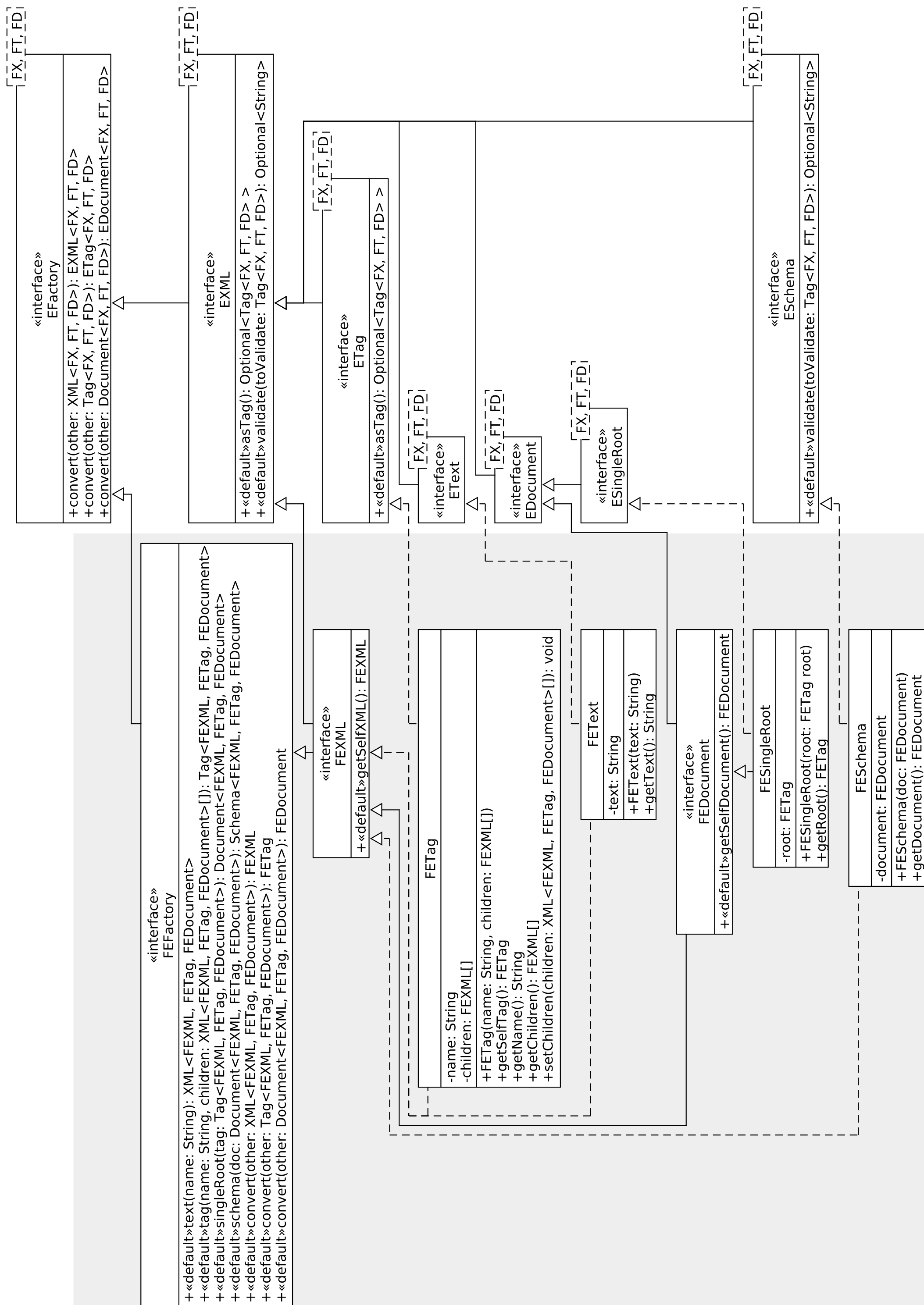

Figure 7 Final layer to instantiate the interfaces from Figure 6. New components are placed in the tinted box on the left. The finalized classes are similar to those from Figure 5, but implement the extended interfaces and instantiate generics to the newest finalized versions.

Listing 1 contains evolving client code using the pattern: the two client classes are **ClientM0** and **ClientM2** (while wrapped using a single class **Client** for this paper, this is not essential). The clients both have factories at their intended level, received as arguments via their constructor (which is just dependency injection, popular in object-oriented programming [8]). Finalized types in the clients are kept as generic parameters. The first client, **ClientM0**, operates at the first level and initializes the document from Listing 2 in its constructor. Its `run` method performs the previously described calls to `hasElem`. The second client, **ClientM2**, is passed a reference to the first, and then constructs a schema for the example XML document. Its `run` method interacts with the first client by calling its `run` method and also by using its document, and validating its root tag (if present). Only in the very last stage of the program, does the `main` method instantiate the generic parameters with the finalized types of the latest required level, and an **FEFactory** is passed to construct all required objects. This final construction step could also be automated by a dependency injection framework such as Guice [11]. The client code is idiomatic Java and the pattern is visible only in the passing of generic parameters and the occasional call to `convert` (here used to convert **Schema** to **ESchema** to gain access to its `validate` method). More advanced languages, such as Scala, can turn `convert` into an implicit conversion, automatically inserting it whenever the compiler expects a more advanced type (we will see this in the next section). In Scala we can also bundle together all generic parameters into one and use type members of path dependent types to access them[2]. This mitigates accumulation of generic parameters (sometimes called "parameterization boilerplate"). Readers with advanced Scala skills will find a demonstration of parameter bundling with path-dependent types in the artifacts for the XML example.

As we have seen in the previous example, CoCo is a design pattern rather than a framework-based approach. It is, therefore, appropriate to conclude this section using the traditional classification of design patterns established in [9].

Pattern Name and Classification

Covariant Conversions (CoCo), Creational and Behavioral Class Pattern.

Intent

CoCo structures data type classes to be extended in the future with new classes, new operations, and new fields without modifying earlier code. Type-safe `convert` methods transform earlier data type instances into their newest evolutionary representation to access operations added later.

Motivation

Traditional inheritance-based object-oriented programming languages do not allow methods to be added to a class hierarchy without modifying previously written code. This is known as the Expression Problem [29] and imposes particular difficulties when producer or binary methods are involved.

Applicability

Use the CoCo design pattern when

- you intend to deploy your classes in compiled form and still allow future evolutions to add new operations
- you want to merge two or more independent evolutions
- you want to introduce new hierarchy levels to an existing subtype structure
- you want to override an existing operation of an existing data type

[2] This is an idea suggested by a reviewer of this paper.

Structure

There are two families of interfaces in the pattern. A *domain interface* undergoes evolutions over time, as new data types and operations are added, and even new subdomains are identified. A *factory interface* provides the API for creating objects from the growing family of data types in the domain and a parameterized convert method that is covariantly overridden to ensure access for data types to all operations defined for the current evolution stage. Each evolution defines an *extension-factory* interface to instantiate objects of that evolution and specify the signature of the conversion method, and *extension-domain* interfaces that specify operations available for the data types in that evolution, as well as their hierarchy. Each of the inheritance relationships exists to share a signature or provide a default implementation.

Collaborations

The actual instantiation of a data type object is deferred to the finalized classes. Objects can ensure API compatibility with convert methods that invoke getSelf to return an instance of the current object at its latest evolution stage. The client code simply invokes operations on the returned objects using regular object-oriented method invocations.

Consequences

This approach minimizes code duplication by ensuring the designer can place the implementation of an operation in either an extension-domain interface or the relevant logic interfaces. An important implication of this pattern is that the compiler can statically detect missing operations in the finalized classes (i.e., a logic-interface is missing a method definition). Also, because there never is a need to dynamically cast objects, there can be no run-time exception during convert. If an evolution only adds new data types (tinted box in Figure 4), there is no need to introduce data type extension interfaces for earlier data types. Each evolution can support a hierarchy to structure the data types as needed. Domain data types need to be exposed via type variable abstractions if they are ever to be used in method signatures.

Implementation All objects are accessed through a hierarchy of domain interfaces, which has a factory interface as its base, and is refined via inheritance in subsequent evolutions, as shown in Figures 4 and 6. Instantiation occurs in a thin layer of finalized interfaces and classes. The top-level domain and factory interfaces have type parameters for every domain type that is mentioned in a signature and needs conversion. Arguments for these parameters ultimately refer to some finalized interface. Subdomain extensions are useful when parts of your subdomain have methods meant only for those subdomains. Not just a matter of having uniform access to the subdomain type hierarchy, methods could be implemented in the intermediate stages of the hierarchy to be shared throughout. Producer and binary methods may refer to the earliest points of definition of domain types in their signatures and operate using factory conversion and construction methods. This avoids any variance issues during inheritance. Client code invoking a producer operation can call convert to ensure the object conforms to the latest evolution stage. It can keep the type parameters for finalized classes abstract and only rely on non-finalized domain factories and interfaces to remain compatible with future updates to the code.

Related Patterns

CoCo uses abstract factories [9] for uniform access to object construction; as discussed in [27]; this allows for future extension by allowing instances of newer evolutions to be supplied to existing code. Accessing APIs through interfaces provided by factories is compatible with the principles of inversion of control, also known as dependency injection. Covariant overrides and finalized classes are inspired by Wang and Oliveira [30].

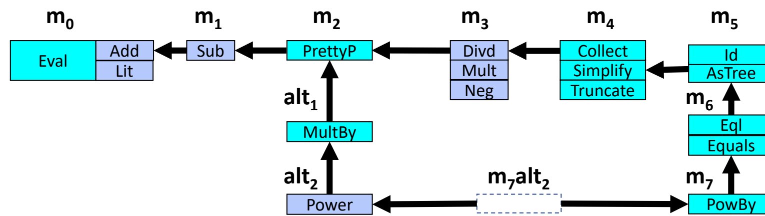

Figure 8 Extension Graph history for mathematical expressions domain.

```
package exp
trait Exp[T] extends Factory[T] { def getSelf:T }
trait Factory[T] {
 implicit def convert(other:Exp[T]) : Exp[T]
}
//////////////////////////////////////////////////////////
package exp.m0
trait Exp[T] extends exp.Exp[T] with Factory[T] { def eval : Double }
trait Factory[T] extends exp.Factory[T] {
 def lit(value : Double) : exp.Exp[T]
 def add(left : exp.Exp[T], right : exp.Exp[T]) : exp.Exp[T]
 implicit override def convert(e : exp.Exp[T]) : Exp[T]
}
trait Lit[T] extends Exp[T] {
 def value : Double
 def eval : Double = value
}
trait Add[T] extends Exp[T] {
 def left : exp.Exp[T]
 def right : exp.Exp[T]
 def eval : Double = left.eval + right.eval
}

object finalized {
 trait Exp extends exp.m0.Exp[Exp] with Factory { def getSelf : Exp = this }
 trait Factory extends exp.m0.Factory[Exp] {
  override def lit(value : Double) : Exp = new Lit(value)
  override def add(left : exp.Exp[Exp], right : exp.Exp[Exp]) : Exp =
      new Add(left, right)
  override implicit def convert(e : exp.Exp[Exp]) : Exp = e.getSelf
 }

 class Lit(val value: Double) extends Exp with exp.m0.Lit[Exp]
 class Add(val left : exp.Exp[Exp],
           val right : exp.Exp[Exp]) extends Exp with exp.m0.Add[Exp]
}
```

Listing 2 Scala implementation of initial version of hierarchy

```
package exp.m1
trait Factory[T] extends exp.m0.Factory[T] {
 def sub(left : exp.Exp[T], right : exp.Exp[T]) : exp.Exp[T]
}
trait Sub[T] extends exp.m0.Exp[T] with Factory[T] {
 def left : exp.Exp[T]
 def right : exp.Exp[T]
 def eval : Double = left.eval - right.eval
}
object finalized {
 import exp.m0.finalized.Exp
 trait Factory extends exp.m1.Factory[Exp] with exp.m0.finalized.Factory {
   override def sub(left : exp.Exp[Exp], right : exp.Exp[Exp]) : Exp =
      new Sub(left, right)
 }
 class Sub(val left : exp.Exp[Exp],
           val right : exp.Exp[Exp]) extends exp.m1.Sub[Exp] with Exp with Factory
}
```

Listing 3 Modular extension adding `Sub` data type with minimal extensions required to factories

3 Case Studies

We have applied the CoCo design pattern to two standard case studies (evolving mathematical expressions and an example from a course on compiler construction) to demonstrate its effectiveness, with full implementations provided as artifacts with the paper.

The evolution history for the mathematical expression domain is captured in Figure 8 using an *extension graph* [27]. CoCo was the only pattern for which we achieved no violation of any of the constraints imposed by the Expression Problem, as we discuss in Section 4. This rich example offers a rigorous benchmark to validate any proposed EP solution. From an initial system, *m0*, with **Lit** and **Add**, evolution *m1* adds the **Sub** data type, while *m2* adds the prettyp operation that creates a string representation of an expression. An independent branch, *alt1*, diverges and adds a new producer operation, multBy, and data type, **Power** is added in *alt2*. The main branch continues development, each new evolution introducing new data types and operations as specified, such as division, multiplication, negation, literal collecting and expression simplification. Truncate is an example of an operation with a side effect. The subsequent three evolutions – *m5*, *m6*, and *m7* – introduce two binary operations equals and eql for equality checks (with equals using equality on trees computed by astree and eql dispatching to its argument instead), and a producer operation powBy for exponentiation.

```
package exp.m2
trait Exp[T] extends exp.m0.Exp[T] with Factory[T] {
 def prettyp : String
}
trait Factory[T] extends exp.m1.Factory[T] {
 implicit override def convert(e : exp.Exp[T]): exp.m2.Exp[T]
}
trait Lit[T] extends exp.m0.Lit[T] with Exp[T] {
 def prettyp : String = value.toString
}
trait Add[T] extends exp.m0.Add[T] with Exp[T] {
 def prettyp:String = String.format("(%s+%s)", left.prettyp, right.prettyp)
 // The compiler implicitly rewrites this to:
 // String.format("(%s+%s)", convert(left).prettyp, convert(right).prettyp)
}
trait Sub[T] extends exp.m1.Sub[T] with Exp[T] {
 def prettyp:String = String.format("(%s-%s)", left.prettyp, right.prettyp)
}

object finalized {
 trait Exp extends exp.m2.Exp[Exp] with Factory {
  def getSelf : Exp = this
 }
 trait Factory extends exp.m2.Factory[Exp] {
  override def lit(value : Double) : Exp = new Lit(value)
  override def add(left : exp.Exp[Exp],
                   right : exp.Exp[Exp]) : Exp = new Add(left, right)
  override def sub(left : exp.Exp[Exp],
                   right : exp.Exp[Exp]) : Exp = new Sub(left, right)

  implicit override def convert(e : exp.Exp[Exp]) : Exp = e.getSelf
 }

 class Lit(val value : Double) extends Exp with exp.m2.Lit[Exp]
 class Add(val left : exp.Exp[Exp],
           val right : exp.Exp[Exp]) extends exp.m2.Add[Exp] with Exp
 class Sub(val left : exp.Exp[Exp],
           val right : exp.Exp[Exp]) extends exp.m2.Sub[Exp] with Exp
}
```

Listing 4 Modular extension that adds `prettyp` operation, requiring extensions for existing data types to contain domain logic, and extended factories to contain implicit conversion methods

A final combined branch, *m7alt2*, merges together two independent branches, *alt2* and *m7*, leading to optimizations where powBy from the main branch is reimplemented to return newly constructed elements of type **Power** from *alt2* (and similarly for multBy and **Mult**).

This case study is fully implemented in Java, Scala, and C# 8.0 with small variations, as provided in the accompanying artifacts. The C# implementation using .NET Core 3.1 illustrating applicability to languages not based on the Java virtual machine. Aside from the language-specific syntax of C#, it completely conforms to the solution we described above.

The Scala implementation of CoCo reveals that only a small amount of boilerplate code is required. Within the space limitations of this paper, we can actually show the full code for extensions up to *m2*. Listing 2 contains the initial `exp.Exp[T]` domain interface and factory meant for a future family of data types in the domain of mathematical expressions. In Scala, interfaces with default methods are represented by traits. The `exp.Exp[T]` trait contains the signature of the `getSelf` method that returns an instance of the parameterized type, `T`. The `exp.Factory[T]` trait specifies the signature of the `convert` method that converts an `other` instance into the most recent realization of the `Exp[T]` type in the domain.

The initial definition of the system, *exp.m0*, provides the `exp.m0.Exp` trait that extends the domain with a new `eval` method computing the numerical value of a data type instance. Two new data types are defined – `Lit` and `Add` – which implement `eval` in the context of values that might (in a future evolution) be further specialized. The `finalized` object acts as a namespace to group the final trait implementations. In accordance with the pattern, they are comprised of a concrete `finalized.Factory` that offers methods to instantiate and convert the known data types, as well as trivial implementations for the domain interfaces. No domain logic of the `eval` method leaks into this finalized area.

```
package exp.m4
trait Exp[T] extends exp.m2.Exp[T] with Factory[T] {
  def simplify : exp.Exp[T]
  def truncate(level:Int) : Unit
  def collect : List[Double]
}
trait Factory[T] extends exp.m3.Factory[T] {
  implicit override def convert(toConvert:exp.Exp[T]) : exp.m4.Exp[T]
}

trait BinaryExp[T] extends Exp[T] {
  var _left : exp.Exp[T]
  var _right : exp.Exp[T]
  def left : Exp[T] = _left
  def right : Exp[T] = _right

  def truncate(level:Int) : Unit = {
    if (level > 1) {
      left.truncate(level-1)
      right.truncate(level-1)
    } else {
      _left = lit(left.eval)
      _right = lit(right.eval)
    }
  }
}

trait Add[T] extends exp.m2.Add[T] with BinaryExp[T] {
 def simplify : exp.Exp[T] = {
    if (left.eval + right.eval == 0) {
      this.lit(0)
    } else if (left.eval == 0) {
      right.simplify
    } else if (right.eval == 0) {
      left.simplify
    } else {
      this.add(left.simplify, right.simplify)
    }
  }
  def collect : List[Double] = left.collect ++ right.collect
}
```

Listing 5 Partial listing from *exp.m4* containing `BinaryExp` generic implementation

Listing 3 encapsulates the first modular evolution, *exp.m1*, adding the `Sub` data type to the system. With minimal new code (analogous to the tinted part in Figures 4 and 5), the new `Factory` classes extend existing factories to provide methods to instantiate `Sub` objects.

Listing 4 encapsulates the second modular evolution, *exp.m2*, that adds a new `prettyp` operation to the system. The existing `Exp` and `Factory` traits are refined from the prior evolution without any code duplication. The new operation must be applicable to all existing data types, so new refined types are created for `Lit`, `Add`, and `Sub`. The finalized `Factory` again instantiates and converts these refined data types. In the `prettyp` implementation for types `exp.m2.Add` and `exp.m2.Sub` we see a feature of Scala at work, which declares `convert` as an implicit method inserted by the compiler when necessary. This reduces the boilerplate code but is not strictly necessary; other languages would manually invoke `convert`.

```
package exp.m7alt2
import exp.m5.{Node, Tree}
trait Exp[T] extends exp.m7.Exp[T] with exp.alt1.Exp[T] with Factory[T] {
  override def powby(other : exp.Exp[T]) : exp.Exp[T] = power(this, other)
  override def multby(other : exp.Exp[T]) : exp.Exp[T] = mult(this, other)

  def isPower(base : exp.Exp[T], exponent : exp.Exp[T]) : Boolean = false
}
trait Power[T] extends exp.alt2.Power[T] with Factory[T] with Exp[T]
  with exp.m5.BinaryExp[T] {
  def base : exp.Exp[T] = _left
  def exponent : exp.Exp[T] = _right
  def simplify : exp.Exp[T] = {
    if (exponent.eval == 0) { lit(1) }
    else if (exponent.eval == 1) { base.simplify }
    else if (base.eval == 0) { lit(0) }
    else if (base.eval == 1) { lit(1) }
    else { power(base.simplify, exponent.simplify) }
  }
  def collect: List[Double] = base.collect ++ exponent.collect
  def id: Int = 80440
  def eql(that : exp.Exp[T]) : Boolean = that.isPower(base, exponent)
  override def isPower(base : exp.Exp[T], exponent : exp.Exp[T]) : Boolean =
    base.eql(this.base) && exponent.eql(this.exponent)
}
trait Factory[T] extends exp.alt2.Factory[T] with exp.m7.Factory[T] {
  implicit override def convert(toConvert: exp.Exp[T]) : exp.m7alt2.Exp[T]
}

object finalized {
  trait Exp extends exp.m7alt2.Exp[Exp] with Factory {
    def getSelf: Exp = this
  }

  trait Factory extends exp.m7alt2.Factory[Exp] {
    override def lit(value: Double): Exp = new Lit(value)
    override def add(left: exp.Exp[Exp], right: exp.Exp[Exp]): Exp =
      new Add(left, right)
    /* ... similar methods omitted ...*/
    override def power(base: exp.Exp[Exp], exponent: exp.Exp[Exp]): Exp =
      new Power(base, exponent)

    implicit override def convert(e: exp.Exp[Exp]): Exp = e.getSelf
  }

  class Lit(val value: Double) extends exp.m7.Lit[Exp] with Exp
  class Add(var _left: exp.Exp[Exp], var _right: exp.Exp[Exp])
    extends exp.m7.Add[Exp] with Exp

  /* ... similar class definitions omitted ... */

  class Power(var _left: exp.Exp[Exp], var _right: exp.Exp[Exp])
    extends exp.m7alt2.Power[Exp] with Exp
}
```

Listing 6 Merging branches *exp.m7* and *exp.alt2*

In evolution *exp.m4*, the `truncate` operation can be generically implemented for any expression with two recursively-defined child attributes. This is accomplished with an intermediate trait, `exp.m4.BinaryExp` shown in Listing 5, that is inherited by data types, such as `Add`, and can also be further extended by subsequent evolutions.

Listing 6 for *exp.m7alt2* shows how to merge different evolved branches. Here, `multby` and `powby` are overridden at the `Exp` level of the hierarchy to always instantiate appropriate domain data types, which become available after the merge. Traits can inherit from their predecessors in the two branches using the Scala's `with` keyword. A refinement of the trait for the exponentiation data type `exp.alt2.Power` is required to supply method implementations for the operations added in the main branch after divergence. For the new main branch data types, this would have also been possible but is not necessary in the example because the only new method in the alternative branch is `multby`, which is specified in `exp.m7alt2.Exp`. The finalized classes work exactly as expected, which is why some of their code is omitted in the listing (but available in the accompanying code repository).

Binary methods [4] are challenging because they involve the object on which they are called (i.e., `this`) and their parameter is also an object of a type present in the inheritance hierarchy. Listing 6 combines two independent branches, bringing together for the first time the `Power` data type (for exponentiation) and the `eql` operation that checks whether two mathematical expressions are equal. Our solution (coded in the `Power` trait) conforms to the strong binary method equality proposed by Zenger and Odersky [31] which dispatches on the arguments.

Listing 7 shows client code for a unit test of `exp.m2.Add`. The test is structured in a reusable version `TestTemplate` that keeps the final class parameter `T` abstract and works with the factory from `exp.m2`, as well as a concrete executable version `ActualTest` which refines the abstract class to use the implementations from `exp.m2.finalized`. This way tests are able to fully reuse the abstract part and instantiate it to use evolved finalized interfaces instead.

```
package exp.m2
import org.scalatest.FunSuite

trait TestTemplate[T] extends Factory[T] with exp.m1.TestTemplate[FT] {
  val suite : FunSuite
  import suite._

  override def test() : Unit = {
    super.test()

    val expr1 = this.add(this.lit(1.0), this.lit(2.0))
    assert("(1.0+2.0)" === expr1.prettyp)

    val expr2 = this.lit(2.0)
    assert("2.0" === expr2.prettyp)

    assert("(1.0-2.0)" === this.sub(this.lit(1.0), this.lit(2.0)).prettyp)
    assert( "((1.0-2.0)+(5.0+6.0))" === this.add(this.sub(this.lit(1.0),
    this.lit(2.0)), this.add(this.lit(5.0), this.lit(6.0))).prettyp)
   }
}

class M2Test extends FunSuite { self =>

  object ActualTest extends TestTemplate[exp.m2.finalized.Exp] with
                             finalized.Factory {
    val suite: FunSuite = self
  }

  test("M2") { ActualTest.test() }
}
```

Listing 7 Test for `exp.m2.Add` in abstract and concrete version

Table 1 Observations for the TAPL case study.

	EVF	**Castor**	**CoCo**
Duplication free domain code	no	no	yes
Fully modular	no	no	yes
Feasible w/o code generator	no	no	yes
Statically typesafe	no	yes	yes
Boilerplate free client code	no	yes	yes
Code Structuring Principle	functions	functions	classes
Human written LOC	763	768	825 (+ 862[a])
Generated LOC	1892	N/A[b]	0

[a]boilerplate for finalized class layer

[b]code generated by compiler internal macros

In our second case study, we implemented, in Java, parts of the Types and Programming Languages (TAPL) textbook by Pierce [22]. The example was also chosen to show the features of the Extended Visitor Framework (EVF) [33] and Castor [34]. Our solution implements typed and untyped compiler modules for natural numbers, Booleans, floats and strings, let-bindings, function application, and lambda-calculus. Compared to EVF and Castor, there are immediate differences in the way code and files are structured: CoCo encourages object oriented-design, placing all functionality of one domain data type evolution into one compilation unit (i.e., class or interface), while the other frameworks are inherently functional with one function definition for all domain data types corresponding to one compilation unit. This switch of perspective enabled us to find multiple *modularity violations* in the solutions provided for the other frameworks. These can be traced back to the original OCaml implementation available with the textbook. A prominent example is the pretty print function, which converts abstract syntax trees of the compiler into human-readable strings. In the CoCo solution, it is a method `print()` which returns a `String` and belongs to the generic interface `Element` for syntax tree nodes. In EVF (and similarly in Castor), pretty-printing is implemented as an object algebra returning instances of the interface `IPrint`, which provides a functional closure over a context to keep track of variable names for binders. This violates modularity because most compiler modules are not concerned with – and do not even supply – syntax for variables. In CoCo, we were able to completely avoid this issue by attaching the necessary name information to the tree upon traversal – the object-oriented view lets us cleanly encapsulate state where it is needed instead of passing it around in a map. Another modularity violation in EVF and Castor is that domain code for lambda terms is *duplicated* to add type annotations to binders. In CoCo, we simply added new fields for the evolutions that require types. Tests in EVF seem to require substantial boilerplate for algebra initialization, similar to the layer of finalized classes in CoCo, but with the crucial difference that finalized classes are not replicated by library clients. In both frameworks, EVF and Castor, users have to interact with and understand generated code, which we found mentally challenging when trying to re-engineer the case study. For EVF, this was especially problematic because of its calling convention through algebraic interfaces and a lack of type-safety where classes accept visitors of previous evolutions. In Table 1 we summarize our findings, where lines of code are counted by cloc [6] on the parts implemented in all three case studies.

4 Related Work

The CoCo Design pattern is most directly related to the approach by Wang and Oliveira [30], which also uses interfaces with covariant overrides to provide multiple inheritance as well as future refinement of data type references. In contrast, there are no issues with binary or producer methods in CoCo, and we are able to support side effects with formal setter methods. In CoCo, one only covariantly overrides the `convert` method, but in [30], one has to covariantly override every single reference to domain data types.

Harrison *et al.* describe an approach that generically abstracts over the final implementation type to improve type-safety in interface-based programming and client-side APIs [12]. From the client perspective, the APIs are similar enough to project the positive results from their case study to clients using CoCo.

From the earliest investigations into the Expression Problem, the Visitor design pattern [9] was essential, whether described by Krishnamurthi *et al.* [17] or Wadler's original email [29]. It only seems natural to turn to Visitor to support newly defined operations in subsequent evolutions.

A formal type-theoretic analysis, conducted by Oliveira [5], reveals that Visitors are related to Church encodings in functional programming languages, showing that advanced type system features, such as F-bounded polymorphism (also used in [27]), are required for type-safe Visitors that remain extensible and satisfy all constraints of EP.

Further investigations on different styles of Visitor encodings [33] reveal how to externalize Visitors and combine them with ideas from Object algebras [32]. Solutions of this form have the drawback of breaking the traditional way to design and invoke object-oriented APIs, and lead to non-standard, idiosyncratic code as discussed earlier in Section 3. External Visitors [33] provide type safety not present in normal visitors but cannot be further extended without substantial code duplication of domain logic.

EP often occurs in frameworks for designing DSLs. MontiCore [13], as well as the Revisitor implementation pattern [18], rely on Visitors. They both hide runtime typechecks behind a layer of code generated from domain-specific languages that extend Java with the explicit purpose of building DSL frameworks. Chapter 2 of [25] provides an overview of modular DSL Frameworks, which could also have been used to implement the compiler case study in Section 3.

Verna [28] provides a detailed account of the practical issues that arise from implementing the Visitor design pattern. In consensus with our observations, the problems with Visitor include non-idiomatic calling conventions and lack of extensibility, without losing type-safety or forcing code duplication. CoCo eliminates all these issues by avoiding Visitors, relying on the idiomatic placement of methods in domain data type interfaces, and providing type-safe conversion methods.

The CASTOR framework [34], which is a follow-up to EVF [33], requires self-type annotations, path-dependent types, and traits. This combination is (to our knowledge) only available in Scala. Despite its heavy requirements, CASTOR does not provide a complete EP solution, because (as the authors acknowledge) nested pattern-matching is not checked for exhaustiveness. In CoCo, no similar problem occurs because dispatching on children is always safe, and default implementations are properly placed in domain data type interfaces. Zhang and Oliveira [34] observe that avoiding exhaustiveness issues is possible by either adding new language features to Scala or duplicating default logic for new data types, in violation of requirements for EP. The more pressing issue, however, is forcing programmers to rewrite existing systems in Scala to realize the benefits from CASTOR. Additionally, the

Scala sub-dialect used by Castor relies on advanced language features, such as macros, which are inaccessible to novice programmers and introduce difficult to understand compiler error messages.

The CoCo design pattern is immediately applicable to numerous mainstream programming languages, such as Java, C#, and C++. However, it cannot be used in Rust or a multi-paradigm language such as Go because both programming languages have discarded class-based inheritance hierarchies in favor of constructs akin to type classes from functional programming languages, such as Haskell. This switch was motivated because of prominent solutions to the EP in functional programming languages, including tagless final [16] and trees that grow [19].

Language extensions are routinely proposed, such as extensible pattern matching with extractors [26], but this introduces compatibility issues with existing code; it additionally requires code generators for substantial boilerplate. Many proposed language extensions deal with the problem of self-types, which was studied in the context of family polymorphism [7]. In essence, the idea is to *existentially quantify* over the domain type and bound the existential quantification by the domain type at the current evolution level. Evidence for this quantification is then associated with each instantiated object and a way to return the current object as an instance of the existentially quantified type is given. This is in sharp contrast to CoCo, where we *universally qualify* over the domain type and avoid any type bounds. Instead, all conversion is centralized in a convert-method and delayed until the last possible point in a finalized getSelf-method, which is no longer generic and thereby does not have to deal with type-bounds. Listing 8 shows a short snippet of Scala, which allows both encodings, to illustrate the essential difference.

```
trait Exp[T] {
  def getSelf: T                    // No bound on T
}
trait FExp extends Exp[FExp] {      // Bound ensured in finalized instance
  def getSelf: FExp = this
}
trait ExpFamily {
  type Self <: ExpFamily            // Bound to ensure compatibility
  def getSelf: Self                 // Returned as a compatible type
}
```

Listing 8 Scala code illustrating CoCo vs existentially encoded family polymorphism

Saito *et al.* [24] show how to extend a minimal Java core calculus with the features necessary for family polymorphism. The idea can also be rephrased with path types [14], which is why the more powerful dependent object types [1] of Scala are so suitable to illustrate it. Still, practical integration into programming languages poses serious challenges, which are beyond the scope of this work, but are addressed with solution proposals in [23, 35]. The latter proposal [35] might be interesting for future work to reconcile CoCo with languages such as Rust because it combines family polymorphism with type classes.

The program languages research community has extensively studied EP since its initial formulation by Wadler in 1998 [29]. Wadler proposed an experimental language, GJ, based on Visitor using a language mechanism to allow a type variable to be indexed by any inner class defined in that variable's bound (similar to the bound on the inner type presented in Listing 8). This requires projections out of generic types [21] which encounters soundness issues and was not added to Java, and even partially dropped from Scala 3.0. The Extensible Visitor [17] requires a runtime check in a Java solution. The Interpreter design pattern [9] has also been suggested to solve EP, but its Factory classes would have to be modified whenever new data types are added. Object Algebras [20] similarly use interfaces to define the evolving interface of the system while factory objects provide concrete implementations. However,

Table 2 Language features necessary for the CoCo design pattern, EP approaches, and related work.

Approach	Data type/ operation extension	Producer methods	Binary methods	Merging independent evolutions	Hierarchical ordering of subdomains
CoCo	interfaces with default methods; covariant overriding of `convert` method return type; parametric polymorphism for `getSelf`	multi-inheritance from interfaces	`convert` method	multi-inheritance from interfaces	inheritance
Trivially [30]	inheritance and interfaces with covariant overriding of return types	not available without violates EP	not available without violates EP	multi-trait-inheritance in trait-based languages	not discussed
Extensible Visitor [17]	inheritance and dynamic cast (violates EP); parametric polymorphism; single-class inheritance	inheritance	method overriding	multi-inheritance from interfaces	not discussed
Interpreter [3]	inheritance and dynamic cast (violates EP)	duplicate methods (violates EP)	dynamic cast (violates EP)	multi-inheritance from interfaces	not discussed
Torgersen [27]	inheritance and dynamic cast (violates EP); final methods; parametric F-bounded polymorphism	not discussed	not discussed	multi-class inheritance	not discussed
EVF [33]	inheritance; parametric polymorphism; interfaces with default methods; multi-inheritance from interfaces; annotation-based macros; lack of type-safety for earlier visitors (violates EP)	parametric polymorphism	parametric polymorphism	multi-inheritance from interfaces	inheritance
Castor [34]	path-dependent types; self-type annotations; multi-trait-inheritance in trait-based languages; partial pattern matching on types (violates EP)	not discussed	not discussed	multi-trait-inheritance in trait-based languages	inheritance

supporting producer methods is only possible when the factory object algebras have access to "the latest" object algebra in the evolution history, which can be accomplished by modifying a special "combined" object algebra that composes together all known factories. This modifies existing code and, in addition, working with object algebras involves considerable boilerplate code for clients, to the point that researchers recommend using code generators [32]. Table 2 summarizes the language features required by various EP solutions.

5 Conclusion and Future Work

The CoCo design pattern combines a number of programming idioms commonly used in object-oriented design (abstract factories, access to and sharing of implementations through interfaces, dependency inversion) with the novel addition of *covariantly* overridden *conversion* methods. This allows the modular future extensibility of class hierarchies with new data types, methods, and fields without code duplication.

We have illustrated how this solves the Expression Problem within the constraints of mainstream object oriented languages, improves modularity, and reduces the amount of boilerplate when compared to other EP approaches. It satisfies the constraints for a "full and

final" solution as summarized by Torgersen [27]. While feasible without tool support, a path for widescale adoption of the pattern should consider compiler assistance for generating the boilerplate code required for the finalized class layer. Scala's implicit conversions are among useful compiler features to make CoCo more straightforward. However, relying on compiler extensions would require fixing a particular language and semantics which we *intentionally avoided* here, to leave the pattern applicable to a broad range of languages. In this line of future work, a precise formal definition and proofs about it become meaningful and should be provided. A further question for future work will be if the additional structure exposed by the pattern can be exploited in code analysis tools to provide better insights into the evolution and code quality of projects. One of the main contributions of the CoCo design pattern is to illustrate that the current trend toward integrating functional programming and specifically pattern matching into object oriented languages (e.g., JEP 394 [10]) is not necessarily the only way forward.

CoCo avoids unsafe instance-of pattern matching and the alternative closed-world assumption (i.e., data types can no longer be extended) to make it safe, without adding new features to the type system **and** remaining compatible with the object-oriented paradigm of programming. We also hope that CoCo solves some of the prevailing issues around the overuse of the visitor pattern [28].

References

1 Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 249–272. Springer, 2016. doi:10.1007/978-3-319-30936-1_14.

2 Jan Bessai, George Heineman, and Boris Düdder. JanBessai/ecoop2021artifacts: State published with paper, 2021. doi:10.5281/zenodo.4756838.

3 Kim B. Bruce. Some challenging typing issues in object-oriented languages. *Electron. Notes Theor. Comput. Sci.*, 82(7):1–29, 2003. doi:10.1016/S1571-0661(04)80799-0.

4 William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In Frances E. Allen, editor, *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, pages 125–135. ACM Press, 1990. doi:10.1145/96709.96721.

5 Bruno C. d. S. Oliveira. Modular visitor components. In Sophia Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, volume 5653 of *Lecture Notes in Computer Science*, pages 269–293. Springer, 2009. doi:10.1007/978-3-642-03013-0_13.

6 Al Danial. Cloc code analysis tool, September 2020. URL: https://github.com/AlDanial/cloc.

7 Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer, 2001. doi:10.1007/3-540-45337-7_17.

8 M. Fowler. Inversion of Control Containers and the Dependency Injection pattern, 2004. URL: http://martinfowler.com/articles/injection.html.

9 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

10 Brian Goetz and Gavin Bierman. JEP 394: Pattern matching for instanceof. Technical report, Open JDK, Oracle Corporation, 2021. URL: http://openjdk.java.net/jeps/394.

11 Guice Framework for Java, 2021. URL: https://github.com/google/guice.

12 William Harrison, David Lievens, and Fabio Simeoni. Safer typing of complex API usage through Java generics. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, page 67–75, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1596655.1596666.

13 Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. Compositional language engineering using generated, extensible, static type-safe visitors. In Andrzej Wasowski and Henrik Lönn, editors, *Modelling Foundations and Applications - 12th European Conference, ECMFA@STAF 2016, Vienna, Austria, July 6-7, 2016, Proceedings*, volume 9764 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2016. doi:10.1007/978-3-319-42061-5_5.

14 Atsushi Igarashi and Mirko Viroli. Variant path types for scalable extensibility. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 113–132. ACM, 2007. doi:10.1145/1297027.1297037.

15 Frederick P. Brooks Jr. *The mythical man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., 1995.

16 Oleg Kiselyov. Typed tagless final interpreters. In Jeremy Gibbons, editor, *Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, volume 7470 of *Lecture Notes in Computer Science*, pages 130–174. Springer, 2010. doi:10.1007/978-3-642-32202-0_3.

17 Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In Eric Jul, editor, *ECOOP'98 - Object-Oriented Programming, 12th European Conference, Brussels, Belgium, July 20-24, 1998, Proceedings*, volume 1445 of *Lecture Notes in Computer Science*, pages 91–113. Springer, 1998. doi:10.1007/BFb0054088.

18 Manuel Leduc, Thomas Degueule, Benoît Combemale, Tijs van der Storm, and Olivier Barais. Revisiting visitors for modular extension of executable DSMLs. In *20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17-22, 2017*, pages 112–122. IEEE Computer Society, 2017. doi:10.1109/MODELS.2017.23.

19 Shayan Najd and Simon Peyton Jones. Trees that grow. *J. Univers. Comput. Sci.*, 23(1):42–62, 2017. URL: http://www.jucs.org/jucs_23_1/trees_that_grow.

20 Bruno C. d. S. Oliveira and William R. Cook. Extensibility for the masses. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, pages 2–27, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

21 Lionel Parreaux. What is type projection in Scala, and why is it unsound?, 2019. Blog Entry. URL: https://lptk.github.io/programming/2019/09/13/type-projection.html.

22 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002. URL: https://www.cis.upenn.edu/~bcpierce/tapl/.

23 Sukyoung Ryu. ThisType for object-oriented languages: From theory to practice. *ACM Trans. Program. Lang. Syst.*, 38(3):8:1–8:66, 2016. doi:10.1145/2888392.

24 Chieri Saito, Atsushi Igarashi, and Mirko Viroli. Lightweight family polymorphism. *J. Funct. Program.*, 18(3):285–331, 2008. doi:10.1017/S0956796807006405.

25 Stefan Sobernig. *Variable Domain-specific Software Languages with DjDSL - Design and Implementation*. Springer, 2020. doi:10.1007/978-3-030-42152-6.

26 Nicolas Stucki, Paolo G. Giarrusso, and Martin Odersky. Truly abstract interfaces for algebraic data types: the extractor typing problem. In Sebastian Erdweg and Bruno C. d. S. Oliveira, editors, *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala, SCALA@ICFP 2018, St. Louis, MO, USA, September 28, 2018*, pages 56–60. ACM, 2018. doi:10.1145/3241653.3241658.

27 Mads Torgersen. The Expression Problem Revisited. In Martin Odersky, editor, *Proceedings of the 18th European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 123–143. Springer International Publishing, 2004. `doi:10.1007/978-3-540-24851-4_6`.

28 Didier Verna. Revisiting the visitor: the "just do it" pattern. *J. Univers. Comput. Sci.*, 16(2):246–270, 2010. `doi:10.3217/jucs-016-02-0246`.

29 Philip Wadler. The expression problem, 1998. E-Mail to the Java Genericity Mailing List. URL: `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`.

30 Yanlin Wang and Bruno C. d. S. Oliveira. The expression problem, trivially! In Lidia Fuentes, Don S. Batory, and Krzysztof Czarnecki, editors, *Proceedings of the 15th International Conference on Modularity, MODULARITY 2016, Málaga, Spain, March 14 - 18, 2016*, pages 37–41. ACM, 2016. `doi:10.1145/2889443.2889448`.

31 Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem, 2004. URL: `http://infoscience.epfl.ch/record/52625`.

32 Haoyuan Zhang, Zewei Chu, Bruno C. d. S. Oliveira, and Tijs van der Storm. Scrap your boilerplate with object algebras. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 127–146. ACM, 2015. `doi:10.1145/2814270.2814279`.

33 Weixin Zhang and Bruno C. d. S. Oliveira. EVF: an extensible and expressive visitor framework for programming language reuse. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 29:1–29:32. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.ECOOP.2017.29`.

34 Weixin Zhang and Bruno C. d. S. Oliveira. CASTOR: Programming with extensible generative visitors. *Sci. Comput. Program.*, 193:102449, 2020. `doi:10.1016/j.scico.2020.102449`.

35 Yizhou Zhang and Andrew C. Myers. Familia: unifying interfaces, type classes, and family polymorphism. *Proc. ACM Program. Lang.*, 1(OOPSLA):70:1–70:31, 2017. `doi:10.1145/3133894`.