Interaction-Based Programming with Classages

Yu David Liu Scott F. Smith

Department of Computer Science
The Johns Hopkins University
Baltimore, MD 21218, USA

{yliu, scott}@cs.jhu.edu

ABSTRACT

This paper presents *Classages*, a novel interaction-centric object-oriented language. Classes and objects in *Classages* are fully encapsulated, with explicit interfaces for all interactions they might be involved in. The design of *Classages* touches upon a wide range of language design topics, including encapsulation, object relationship representation, and object confinement. An encoding of Java's OO model in *Classages* is provided, showing how standard paradigms are supported. A prototype *Classages* compiler is described.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features - classes and objects

General Terms

Languages, Design

Keywords

Classages, Interaction, Interface, Encapsulation, Relationship, Confinement

1. INTRODUCTION

In this paper we define *Classages*, an object-oriented programming language with more refined notions of interaction between classes and objects. *Classages* provides explicit support for the three interactions most commonly identified during the software design phase: *static composition* between two classes, *peer-to-peer communication* between objects, and *whole-part communication* between objects. These object interactions have a strong relationship with UML associations : peer-peer communication models the common form of UML's association relationship, and whole-part communication models a special form of UML association relationship, composition (UML's term for aggregation where the part does not outlive the whole). Other aspects of (UML) object associations, including lifespan, multiplicity, navigatibility and statefulness, are also addressed by *Classages*.

Our goal is not to explicitly create a UML-based programming language, but to create a language that does the best job of expressing object interaction relationships; it just so happens that UML does a better job of expressing object interactions than is commonly found in OO language syntax, and so our language shares several features of UML. Some of the new features on the other hand do not have a direct UML analogue. Having a language that more directly expresses object interactions, such as those found in an initial UML design, will bring more explicit intuition about the architecture into the program, which will help make the programming phase a smooth continuation of the software design process.

To achieve full encapsulation, *Classages* interactions invariably occur on explicitly defined interfaces of classes and objects: there are no backdoor channels of interaction such as nonlocal variables. Classes in *Classages* are equipped with *interaction interfaces* to support the three sorts of interactions defined above, and those interfaces are the complete definition of how a class interacts with the outside world.

The standard object model [1] based on inheritance and messaging also is fundamentally centered on interaction, and we are not claiming there is anything wrong with that model, only that it can be refined via a more precise vocabulary of interactions and relationships. Since the dawn of OO languages, a class has been defined as a collection of fields and operations. Once this notion is fixed, the forms of interaction that can be expressed are narrowed: objects can interact by invoking each others' operations, and classes are combined together by a means such as inheritance or mixing. This is a less interaction-centric view than we pursue—in the standard model, issues such as visibility control, object ownership [10, 11], environmental acquisition of data [12], and explicit encapsulation policies [28, 26] are topics that must be added on later. We show that by taking an interaction-based view from the beginning, these ideas naturally fall out in the course of language design.

Contributions of *Classages* include

- Building in a notion of strong object encapsulation, by implicitly requiring all interactions a class or its runtime instance might be involved in to be explicitly declared on interfaces.

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.

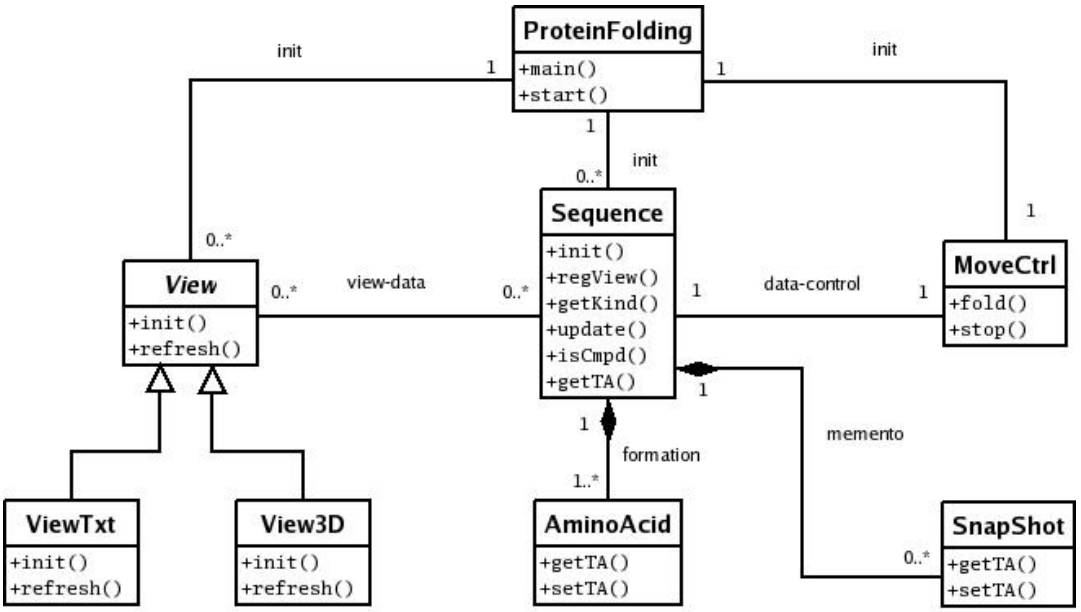

Figure 1: The UML Class Diagram for a Protein Folding Application (method signatures omitted)

- First-class support for class-class and object-object relationships via explicit, named interfaces.
- First-class support for whole-part interactions where the part object is confined and owned by the whole.
- A new flexible class composition model where overriding can happen either statically or dynamically, message dispatch can happen statically or dynamically, and novel forms of feature composition are possible.

The *Classages* compiler, together with the language manual and examples, can be downloaded at

`http://www.cs.jhu.edu/~yliu/Classages`

2. A HIGH-LEVEL ACCOUNT

In this section, we give a high-level account of important concepts in *Classages*. From this point on, we refer to classes in our language as *classages*, and their corresponding instances as *objectages*. The terms *class* and *object* are still used to refer to the standard OO concepts.

2.1 A Running Example

We illustrate our ideas with an example protein folding simulation application [1]. The application goal is to simulate the structure-forming process of proteins when the sequence of amino acids is given. The simplified design is illustrated by a UML class diagram in Fig. 1. The design follows the classic model-view-controller pattern. The model part, a protein sequence, is modeled by the `Sequence` class, the instance of which is composed of multiple `AminoAcid` instances. The 3D structure of the sequence is recorded by the `AminoAcid` instances, each of which records its *torsion angles* with respect to the previous amino acid in the sequence; the `getTA` and `setTA` methods are used for access to these values. The core algorithm is modeled by the controller class `MoveCtrl`, whose major method `fold` repeatedly selects at random three consecutive amino acids on the sequence and *move*-s them in a way such that the resulting 3D structure has a lower energy. Every successful move of the sequence is recorded, modeled in the diagram by a `SnapShot` class, which follows the **Memento** design pattern [16]. After thousands of successful moves, the final structure of the protein is obtained by analyzing the statistical distribution of the snapshots. For human interaction, the folding process is typically illustrated by 3D graphs (`View3D`) and/or text output (`ViewTxt`), following the **Observer** pattern. For now, readers can ignore the particular meanings of the methods in the diagram.

How such a UML diagram can be implemented in Java is known. We now give a brief overview of how this design is implemented in *Classages*, and along the way will introduce the most important language concepts.

[1]This oversimplified illustration is inspired by LINUS [29], a program developed by two biophysicists at Johns Hopkins. The original program was written in Python with OO features.

2.2 Classages Statics and Dynamics

Fig. 2 shows how the design of Fig. 1 is implemented in *Classages*; (a) shows the static code blocks, and (b) shows one runtime snapshot of the application. As with classes and objects, each classage can have multiple objectage instances at runtime; for example, there are two instances of `ViewTxt` in the Figure, indicating that the application currently has two different windows showing a textual display.

2.2.1 The Static View

The static view closely corresponds with the UML class diagram in Fig. 1. Every object-object association of the general form in the diagram results in two interaction interfaces called *connectors* being defined, one on each classage.

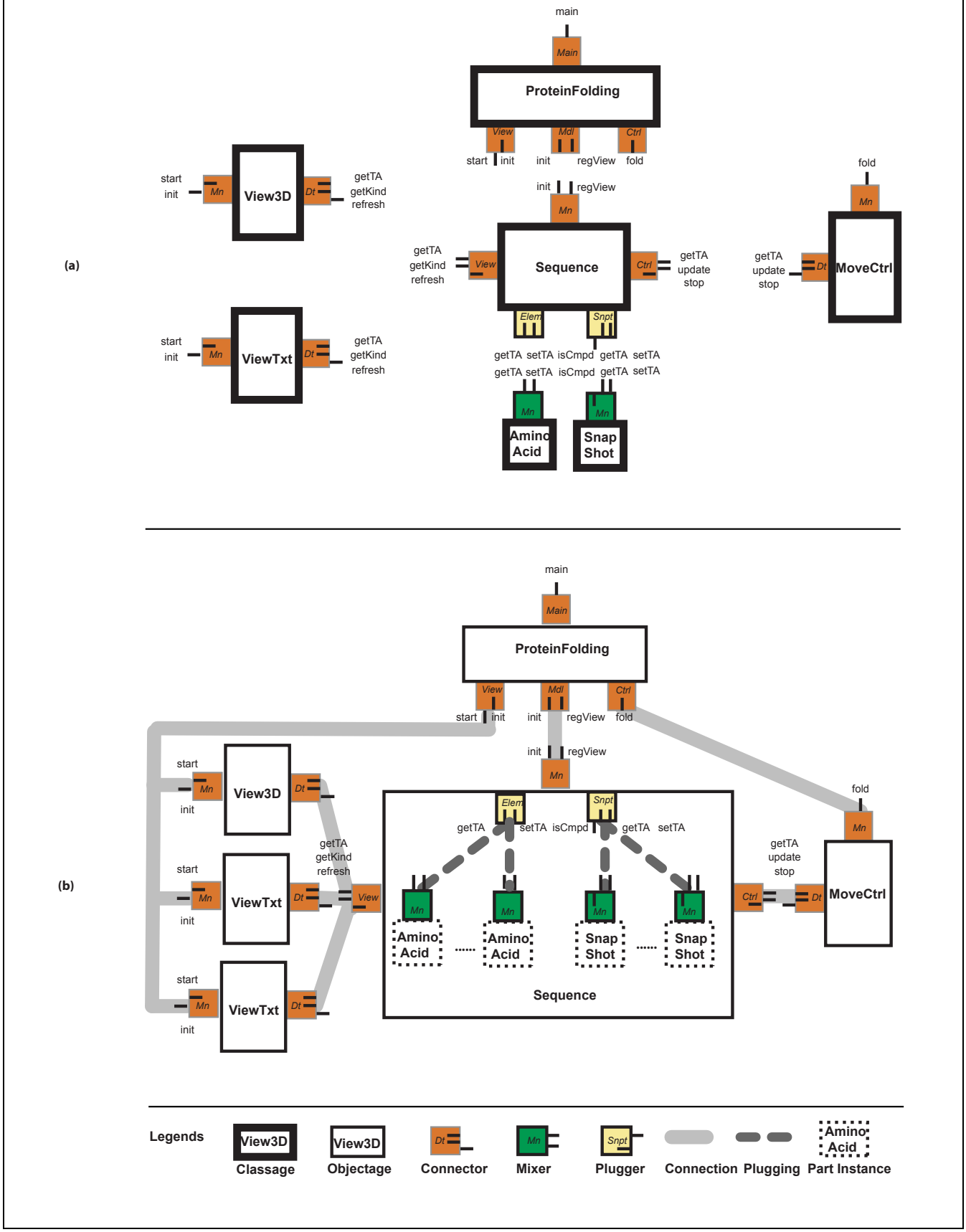

Figure 2: A *Classages* Design of the Application in Fig. 1 (a) The Static View (b) The Dynamic View

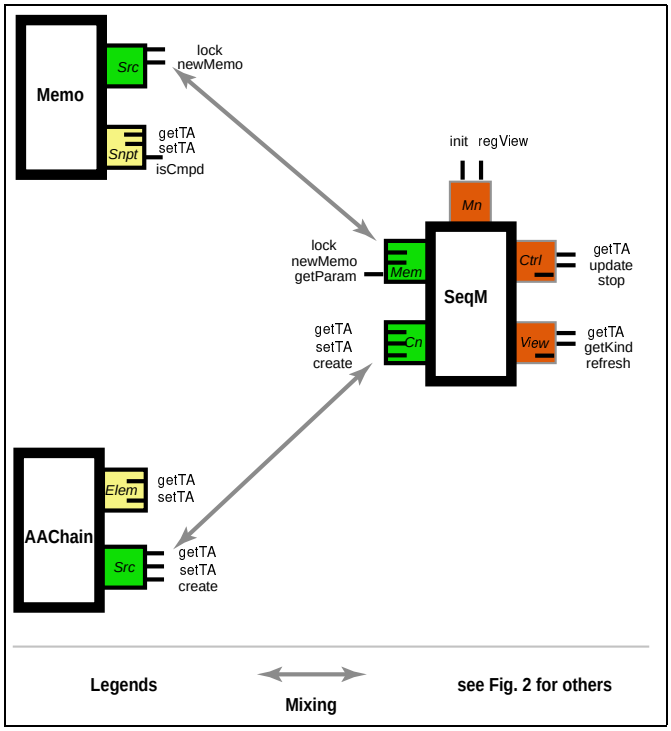

Figure 3: A Mixing Process to produce classage `Sequence`

```
classage Sequence {
    connector Mn {
        export void init() {...}
        export void regView(ViewI o) {...}
        export void unregView(ViewI o) {...}
    }
    plugger Elem {
        import double getTA()
        import void setTA(double t)
    }
    plugger Snpt {
        import double getTA(int i)
        import void setTA(double[] a)
        export boolean isCmpd() {...}
    }
    connector View {
        import void refresh()
        export double getTA(int i) = ::getTA
        export int getKind() {...}
        state Time lastUpdated
    }
    connector Ctrl {
        import void stop()
        export void update(double[] a) {
            forall (c::View) { c->refresh(); }
            Snpt p = plugin Snapshot with Snpt >> Mn;
            p..setTA(a);
            if (::allLocked()) { stop(); }
        }
        export double getTA(int i) = ::getTA
    }
    boolean allLocked() {...}
    double getTA(int i) {...}
    ...other local fields and methods ...
}
```

Figure 4: A *Classages* Code Snippet for `Sequence`

And, every composition relationship in the UML class diagram results in two interaction interfaces, a *plugger* on the whole-classage, and a *mixer* on the part-classage (the reason for the different sort of interface on the two sides originates in the asymmetry of whole-part relationships, and will be detailed below). By comparing the `Sequence` class in the UML diagram and the one in Fig. 2, notice that the public visible methods on `Sequence` in the UML diagram are split across three different connectors and two pluggers. This is not just good for program understanding, but is also important for improved object encapsulation: the `Sequence` classage can interact with different parties via different interfaces, always exposing the least and demanding the most. This aspect of our language is related to Encapsulation Policies [28, 26].

At compile time, static compositions can occur, as shown in Fig. 3. This diagram details how the ultimate `Sequence` classage can have its responsibilities distributed over several smaller classages, and these can be pieced together via a *mixing* process similar to mixin composition [9, 15] to produce `Sequence`. Here the three smaller classages are: the `AAChain` classage taking charge of amino acid chain organization, the `Memo` classage taking charge of storing snapshots and the final statistics over them, and the `SeqM` classage taking charge of forming the MVC pattern. Two mixings have occurred: the first involving `Memo` and `SeqM`, and the second involving `AAChain` and the result of the first mixing. The result of mixing is also a classage with mixers, connectors and pluggers. Mixing happens between a pair of mixers; this is different from mixins, where there is no explicit interface of mixing, and as a result more name conflicts are likely. Fig. 3 can be defined in *Classages* by the following code snippet:

```
classage MemoSeq  = SeqM + Memo with Mem >> Src
classage Sequence = MemoSeq + AAChain with Cn >> Src
```

Mixers are bi-directional interfaces composed of both import and export declarations. The import declarations are "expectations" of methods by the other party participating in the mixing interaction, and the export declarations are "contributions" of the methods to the other party. Mixing is a method matching process where "contributions" meet "expectations".

Note that connectors and pluggers are *late-binding interfaces*: they are only satisfied at runtime. At mixing time, the `Sequence` classage takes on any pluggers and connectors defined by its components, as can be seen by comparing Figures 3 and 2.

2.2.2 The Dynamic View

The two fundamentally dynamic interactions in *Classages* are connection and plugging. They are how runtime peer-to-peer communication and whole-part communication can be represented in *Classages*, respectively.

Connection is a form of long-lived and potentially stateful interaction between objectages, and whose legal actions, such as what methods can be invoked, are well-defined via interfaces. In Fig. 2 (b), there is a live connection between the objectages `ProteinFolding` and `View3D`. At a high level, a connection represents a long-lived relationship between peers, and often embodies a UML association. A connection happens on a pair of connectors, another form of bi-

directional interface with import and export declarations. To establish a connection between the View connector of the Proteinfolding objectage and the Mn connector of the View3D objectage, the ProteinFolding objectage at runtime can evaluate the following expression, where v is reference to objectage View3D:

```
c = connect v with View >> Mn;
```

This is again a process of "contributions" meeting "expectations", only it happens at runtime. The result of the connect expression, c, is called a *connection handle*, the first-class incarnation of the peer-to-peer communication relationship. During the connection, the ProteinFolding objectage can call init to initialize the 3D view of protein structures, via the *Classages* syntax c->init().

Plugging is another form of long-lived interaction, focusing on whole-part and not peer-to-peer interactions. In Fig. 2 (b), the objectage Sequence is holding multiple pluggings of Snapshot instances, indicating multiple snapshots of the protein structures have been taken and recorded. Unlike connection interaction which happens between two objectages, a plugging interaction can be figuratively thought of as *a "part" being plugged into the object memory space of the objectage representing the "whole"*. The illustration here indeed is precise from a memory management point of view: when the memory space for the whole is released, the part is also released, and from an access control point of view, outsiders of Sequence cannot have direct access to Snapshot instances. Pluggings give the programmer a strong tool for confining objects, of the general flavor of other work including [10, 3, 11, 31, 8]. It also can be looked on as a method for dynamically extending an already-created object. A plugging represents a more tightly-coupled relationship than a connection: it is analogous to a UML composition association, which constrains the part to not outlive the whole. Another implicit constraint in *Classages* is the part only comes into being as part of the whole—there is no such thing as an isolated part instance. Plugging happens on the *plugger* interface of the whole and the *mixer* interface of the part: the whole is a dynamic object, and at the time of plugging the part is but a code template to be instantiated. A plugging interaction between the the Snpt plugger of the Sequence objectage and the Mn mixer of the Snapshot classage is established at runtime by the following expression within Sequence:

```
p = plugin Snapshot with Snpt >> Mn;
```

The result of the plugin expression, p, is called a *plugging handle*, the first-class incarnation of a whole-part communication relationship. After plugging, the Sequence objectage can call getTA to query the torsion angles of the just-created snapshot, expressed by p..getTA() in *Classages*. Similarly, any Snapshot instance can check whether the data should be stored in compressed form by the code isCmpd()—the "whole" need not be explicitly mentioned, it is implicit that the Sequence being communicated is the one the Snapshot sits in.

A plugee can itself have plugees, giving explicit language support for hierarchical containment relationships. Both connections and pluggings can be terminated explicitly, via disconnect c and unplug p expressions respectively, where c is a connection handle and p is a plugging handle.

2.3 Interactions and Interaction Interfaces

As we have seen, three kinds of interactions play crucial roles in *Classages*: mixing, connection and plugging. The first-class representation of these relationships in *Classages* are related to research in first-class object relationships [25, 18, 13, 6, 17]. At a high level, interaction interfaces indicate a willingness to communicate, a role to play, and a relationship to participate in. In *Classages*, *all* interaction is via these interfaces; there is *no* syntax for a message send from one objectage to another independent of an established interface. This may seem like a radical departure from the norm, but we believe closing one door opens up many others.

2.4 A Classages Code Snippet

Fig. 4 gives the *Classages* code for the Sequence classage illustrated in Fig. 2, here written as a single code piece and not via a composition as in Fig. 3. Its structure mirrors Fig. 2, with each interface declared directly. We show code for the update method in connector Ctrl. Since the Ctrl connector is going to be connected to the MoveCtrl objectage at runtime, the update method will be invoked by MoveCtrl every time it finds a new "folding" that has a lower energy. In this case, all the GUI views are updated (via the forall expression), and a new snapshot is stored (via the plugin expression). By analyzing existing data, the Sequence objectage can make a decision as to whether the current "folding" is precise enough (via local method allLocked), and if the case, inform MoveCtrl to terminate (by invoking stop). Other details of this snippet will be explained below.

3. FROM PRINCIPLES TO CLASSAGES

In this section, we show how the *Classages* design emerged from basic principles of language design. We believe the seven principles presented below are not sufficiently supported by mainstream OO languages, and altogether they help answer the important question of "why a new language?". Along the way, we expand on various aspects of the language that were left out of the previous example.

3.1 On Separation of Statics and Dynamics

PRINCIPLE 1. *Static and dynamic behaviors of an entity should be specified independently.*

In OO languages, a class has two roles to play: it both serves as a piece of code for reuse, and also as the "mold" via which objects are generated. Although a class's reuse behavior is orthogonal to its runtime behavior, existing OO models tend to treat a class's interface for reuse as closely related to its runtime interface. In mixin systems, the methods of a mixin compound that can take messages at runtime *happen to be* the methods that each participating mixin exports for static composition. Java is more advanced in this regard: its protected visibility modifier provides a way to distinguish between the two interfaces, because protected methods can only be used for static reuse, and not for runtime method invocation[2]. Overriding unfortunately muddies the waters here for Java. Overriding—a code-reuse operation—is tethered to dynamic dispatch, a runtime operation. The result of this is that a programmer's intent to reuse can conflict

[2]The modifier protected in Java also signifies package access, a property that is orthogonal to the discussion here.

with their intent for runtime communication, and lead to problems such as the fragile base class problem, detailed in Sec. 6.1.

3.1.1 The Classages Solution

In *Classages*, how a classage is to be reused is completely orthogonal to how its runtime instance is to communicate with other objectages. The reuse (*i.e.* code composition) behaviors are all specified on *mixers*, while the runtime behaviors are all specified on *pluggers* and *connectors*. Thus there is a completely natural separation of concerns.

With this separation, overriding only happens on mixers, and message dispatch only on connectors. Flexibility is gained by this separation, as will be discussed in Sec. 6.1 below.

3.2 On Separation of Internals and Externals

PRINCIPLE 2. *At runtime, communication between an object and its component parts is fundamentally of a different flavor than communication with its peers.*

How this principle is implemented in *Classages* is clear: pluggings are used for the former, and connections for the latter. What we want to justify is why this separation is fundamentally important. In fact, the essence of this *Classages* guiding principle has been the center of philosophy for centuries: self and others. For a real-world analogy,

- Connectors show how an objectage lives with its external world, *i.e.*, how it communicates with others. In the human world, it shows the "social" aspect of an individual.
- Pluggers show how an objectage lives with its internal world, *i.e.*, how it is formed and how it evolves. In the human world, it shows the "introspective" aspect of an individual.

This separation has some interesting ramifications on language design:

First, on *protection of internal representations*. Historically, protecting internal representations has been a major dimension of programming language design, and is realized in both ADTs and objects. In the standard object model, internal objects can be placed in private fields, but this provides only partial protection because nothing prevents private fields from being passed to other objects or returned as values. In *Classages* we achieve a stronger protection of the "part" from being accessed from outside of the "whole" via the *Classages* type system, introduced in Sec. 5.

Second, for *garbage collection*. Since whole-part relationships are explicitly supported and we have a guarantee the part will not escape the whole, the part instance can always be safely garbage collected when the whole is garbage collected.

Third, for *serialization*. By differentiating pluggers and connectors, the programmers' intent of how tightly coupled two objects are related is expressed: the whole-part relationship is more tightly coupled than peer-to-peer relationships. A tight serialization policy could be defined that serializes only parts of wholes, not objectages interacting only by connectors. A broader policy could serialize all plugged and connected objectages, but still not serialize objectages for which only a reference was held in a field. In Java, all objects the current object has references to are also serialized, provided they implement `Serializable`. This policy is not based on the coupling of objects but on the sorts of objects, and so cannot be as discriminating as a policy that can use how objects are coupled.

3.3 On Interaction Bi-directionality

PRINCIPLE 3. *Interaction is fundamentally bi-directional.*

The root *inter-* of the very word *interaction* already suggests a reciprocal relationship.

It is important for OO programming because, even in existing languages, bi-directional interaction is also pervasive, though sometimes in a less explicit way:

- For class-class interaction via inheritance, a naive perspective would be that the superclass provides and the subclass consumes, but overriding reverses this: the subclass provides and the superclass consumes. Another example is the abstract modifier of Java.
- For object-object interaction, a naive perspective would be the message sender asks and the receiver answers, but callbacks, where the sender sends this as a parameter, exist because there is a need for the receiver to ask and the sender to answer. Other examples include the Observer design pattern [16], event systems, Web interactions, human-computer interaction applications (GUI, robotics), *etc.*

In the realm of relationships, bi-directional interactions suggest relationships with two-way navigatibility. The argument that bi-directional interaction can be modeled by two uni-directional interactions is not satisfactory to us, since the language cannot signify how the two uni-directional relationships are related, or how they depend on each other. It is about as intuitive as buying two cellphones, a "hear" cellphone from one company and a "talk" phone from another, and then using the two phones together every time you want to have a phone conversation.

3.3.1 The Classages Solution

In *Classages*, all interaction interfaces are by default defined as bi-directional, with both exports to indicate what it *provides* to the other party engaged in the interaction, and imports to indicate what it *expects* from the other to provide. Predictably, interactions (be it mixing, connecting, or plugging) happen when two interacting parties match the interfaces with each other and get expectations (imports) satisfied by provisions (exports) of the other. Being "satisfied" is a type property enforced by the static typechecker. The two interaction interfaces do not need to be strict reflections of each other: for instance in Fig. 3, it is fine for `Mem` mixer to have extra exports, such as `getParam`. In fact, for each matching import-export we only require the export method is of a subtype of the import method.

With bi-directional interfaces, dependencies between classes and objects are made explicit, thus reducing coupling between them and promoting understanding of classes in isolation. In the context of GUI design in Fig. 2, how a view objectage is controlled by other objectages, and how it can take commands from users and to control other objectages, are both clearly specified.

Support \ Relationship Lifespan	Permanent	Long-lived	Ephemeral
Real World	conceptual relationship	concrete relationship	concrete relationship
General OO	class relationship	object relationship	object relationship
UML	generalization	association	association
Java/C++	inheritance	object reference in fields	message passing
Classages	mixing	connection and plugging	ephemeral invocation

Figure 5: The Spectrum of Relationship Lifespan

3.4 On Interactions with Least Privilege

PRINCIPLE 4. *Entities should declare the most precise interface, i.e. the interface with the least privilege, for each interaction they are involved with.*

COROLLARY 1. *Each class should be able to have multiple interfaces for its static interactions. Each object should be able to have multiple interfaces for its dynamic interactions.*

The Principle of Least Privilege—a cornerstone of security research—also impacts encapsulation in OO languages: a class should only expose its components absolutely necessary for static interactions (such as inheritance), and an object should expose only the components absolutely necessary for dynamic interactions.

Controlled by its various visibility modifiers, a Java class can be viewed as only providing one monolithic interface for its subclass, and a monolithic interface for its runtime clients. These monolithic interfaces will not be least, because everything used by anyone needs to be put on the monolithic interface.

3.4.1 The Classages Solution

In *Classages*, programmers can define multiple mixers in a classage to adapt to different static composition needs, and multiple connectors or pluggers in an objectage to support associations with different peers or different parts. Fig. 2 gives one example: the `Sequence` classage has three connectors, `View`, `Ctrl` and `Mn`, and two pluggers, `Elem` and `Snpt`. By exporting `init` in connector `Mn` but not others, *Classages* avoids accidental use of `init` by `MoveCtrl` and `View3D`.

Be clear that Java's `interface` mechanism does not serve this purpose, as pointed out in *e.g.* [28]: any variable declared with an `interface` type can always be downcast to the object type implementing the `interface` and then it will have access to methods not belonging to the `interface`. This is because Java `interface` types are designed for subtype polymorphism, not for encapsulation.

Note that we are defining an encapsulation mechanism, and not a security mechanism, since connections are never refused and so access to data can always be gained by providing the proper interface. For example, in Fig. 2, if an objectage `BadGuy` intends to communicate with the `Sequence` objectage via connector `Mn`, it is not possible to have access to method `getTA` of `Sequence`, and thus some accidental bugs may be avoided. On the other hand, if `BadGuy` had an interface for connecting with `Ctrl`, the `getTA` method would then be accessible, since the name `Ctrl` is public.

3.5 On Relationship Lifespan

PRINCIPLE 5. *A relationship's lifespan sits somewhere on the spectrum from permanent to long-lived to ephemeral.*

The design process of *Classages* is also a study of relationships. One important aspect of relationships is its lifespan. We summarize support, both from the real world and from programming languages, in Fig. 5.

In the real world, a *conceptual relationship* is a law, or a theorem: an invariant and unchanging relationship between concepts. For example, the concept of carbon monoxide is invariably a molecule consisting of one carbon and one oxygen atom. The concrete molecule—such as a particular carbon-oxygen molecule is formed dynamically by a reaction, will be stable for some time, and will then decompose by another reaction. Other relationships will be very short, for example carbon monoxide from automobile exhaust that quickly oxidizes to form ozone. In this case the carbon-oxygen relationship was *ephemeral*.

Here we study how OO languages support relationships of different lifespans. In mainstream OO languages, conceptual relationships are classes permanently related by inheritance, and object message passing is an ephemeral relationship, starting when a method is invoked and ending when the return value is sent back. Two message sends to the same object are independent ephemeral events. Using message passing to model object-object relationships is arguably weaker than what is implied by a UML association, which is an explicit, potentially long-lived, object-object relationship. In the standard OO model, it is the *fields* that models long-term relationships, where one object can store a reference to another object in a field. This is especially true for the case of a UML aggregation or composition relationship [17]. The standard model is less precisely defining a relationship, however: the communication is via messages which is a semi-public channel since many other objects – not just the objects being held in the fields – could also be sending messages to the same object. We clarify this perhaps subtle distinction with an example.

Suppose Fig. 1 was implemented in Java, and a model checking tool is interested in ensuring a very simple temporal constraint: every time the `Sequence` object refreshes the `View3D` object, the latter should call back `getTA`, to get the latest torsion angle data. An **Observer** design pattern will lead to a `View3D` object being implemented this way:

```
class View3D {
    private Sequence source;
    ...
    public refresh() {
        int x = source.getTA();
        ...
    }
}
```

Since there is no static way to ensure the `Sequence` object invoking `refresh` is indeed the *value* stored in `source`, the simple callback constraint we want to check will be very

difficult to verify statically. The fundamental reason behind this is Java does not have a way to indicate the longstanding relationship between two objects, and thus how the `getTA` invocation here is always part of such a relationship between objects.

3.5.1 The Classages Solution

Mixing reflects the conceptual relationship between classages, since classages already mixed together cannot be severed at runtime, it represents a relationship with a permanent lifespan.

Both connection and plugging represent long-lived relationships. A connection is explicitly established via the connect expression, and it is alive until the connection is explicitly terminated via the disconnect expression. Similarly, plugging has its lifespan between the plugin expression being evaluated and the unplug being evaluated. Since long-term relationships are more explicitly supported than in the standard object model, there is deeper knowledge about object relationships. We can show this by revisiting the above **Observer** example, in *Classages*:

```
classage View3D {
    ...
    connector Dt {
        import int getTA()
        export refresh() {
            int x = getTA();
            ...
        }
    }
}
```

A connection between a `Sequence` objectage and a `View3D` objectage is established on the `Dt` connector on the `View3D` side. When `Sequence` invokes `refresh`, the `refresh()` of the connected `View3D` objectage is invoked, which leads to invocation of the `getTA` method, *always* of the original `Sequence`. This callback relationship is statically fixed, because there is an explicit object-object interaction defined between the `Sequence` and `View3D` objectages. A model checker will then easily be able to verify the callback constraint.

Classages' support for ephemeral relationships is to model them as brief versions of long-lived relationships: *Classages* reuses connectors for ephemeral relationships. To initiate an ephemeral interaction with method `m` defined in the connector `C` of objectage `o`, one can use syntactic sugar `o.m() at C`. This is de-sugared as a connection established with the `C` connector of `o`, method `m` being invoked, and then the connection terminated. The sugared syntax is intentionally similar to Java/C++ method invocation, because they both represent ephemeral relationships.

3.6 On Relationship Multiplicity

PRINCIPLE 6. *A dynamic relationship between two sorts can be a one-to-one, one-to-many, or many-to-many relationship.*

This obvious fact of relationships does not need further justification; it is realized in UML class diagrams via multiplicity declarations.

In the example, a `Sequence` objectage might have multiple `View3D` view objectages for display, and a `View3D` objectage might at the same time display the shape of multiple `Sequence` objectages.

3.6.1 The Classages Solution

Pluggers and connectors in *Classages* are by default *generative*, *i.e.* multiple pluggings or connections might be available to the same plugger or connector at the same time. For instance in Fig. 2 (b), the `View` connector of the `Sequence` objectage is connected to three views, because the same data change might require multiple views to be notified. Likewise, the `Elem` plugger is plugged in with multiple `AminoAcid` instances and the `Snpt` plugger with multiple `Snapshot` instances, showing the one-to-many multiplicity behind many whole-part relationships.

One important issue in this context is how an objectage can then distinguish different interactions on the same interface. Recall that each connection is given first-class status via a connection handle, and similarly a plugging gets a plugging handle. According to *Classages* syntax, access to methods defined in connectors is via a handle to a live connection. For instance, for a `ProteinFolding` objectage to invoke the `init` method defined by the `View3D` objectage, the expression used is `c->init()`, where `c` is the connection handle representing the connection with the `View3D` objectage.

When a connector (or a plugger) is known to interact with *only* one objectage at runtime, it is inconvenient for programmers to keep track of the connection handle (or the plug handle) explicitly. For convenience, *Classages* also allows programmers to specify a connector or a plugger as a singleton, and then refer to an import or export method in it without using a handle, via syntax `I::m`, where `I` is the connector or plugger name and `m` is the method name.

3.7 On Relationship Statefulness

PRINCIPLE 7. *Relationships are frequently stateful in their own right.*

Long-lived relationships are rarely stateless. For instance in a data-view relationship such as that between `Sequence` and `View3D` objects, programmers might be interested in when each view was last updated. A standard way for Java/C++ programmers is to declare an object field either in `Sequence` or in `View3D` to store the information, but since the relationship between `Sequence` and `View3D` is many-to-many an array has to be used; maintaining an array in this case will require additional overhead. The fundamental problem here is that the last updated information is fundamentally part of the state of the data-view relationship itself.

3.7.1 The Classages Solution

Classages allows connections to hold their own state. It is achieved by allowing connectors to declare *connection-specific fields*. For example, see the `lastUpdated state` declarations in connector `View` of `Sequence` shown in Fig. 4. Every time a connection is established on the connector, fresh connector state is allocated. The state is private to the connection and cannot be directly accessed by other connections.

A related issue is whether pluggings should also hold their plugging-specific states. The answer is yes, but due to the asymmetry between plugger and plugee, each "part" has a unique "whole" and so the private state can always rest as a field of the part. This state is also not exposed, because parts are kept internal to wholes.

program	::=	$\overline{atomic \mid compound \mid rename}$
atomic	::=	classage *cn* $\{\overline{interface \mid local}\}$
compound	::=	classage *cn* = *cn* + *cn* with $\overline{tn \texttt{ >> } tn}$
	|	classage *cn* = *cn* + *cn* with $\overline{tn \texttt{ >> } tn}$ as *tn*
rename	::=	classage *cn* = *cn* rename *tn* >> *tn*
interface	::=	*mixer* | *plugger* | *connector*
local	::=	*field* | *method*
mixer	::=	mixer *tn* $\{\overline{im \mid em}\}$
plugger	::=	plugger *tn* $\{\overline{im \mid em}\}$
	|	singleton plugger *tn* $\{\overline{im \mid em}\}$
connector	::=	connector *tn* $\{\overline{im \mid em \mid connf}\}$
	|	singleton connector *tn* $\{\overline{im \mid em \mid connf}\}$
field	::=	$\tau\ fn = e$
method	::=	$\tau\ mn(fml_1, \ldots, fml_n)\ \{e\}$
im	::=	import *methodSig*
em	::=	export *method*
connf	::=	state *field*
methodSig	::=	$\tau\ ln(formal_1, \ldots, formal_n)$
formal	::=	τ *varn*

cn	classage name
tn	interaction interface name
mn	method name
fn	field name
varn	variable name
e	expression, see Fig. 7
τ	declared type, see Sec. 5

Figure 6: *Classages* Top-level Abstract Syntax

e	::=	() | *varn* | *cst*	
		create *cn* $(e_1, \ldots, e_n)$	instantiation
		plugin *cn* with *tn* >> *tn*	plugin
		unplug *e*	unplug
		connect *e* with *tn* >> *tn*	connect
		disconnect *e*	disconnect
		$e = e \mid (\tau)e$	assignment/cast
		forall$(varn : tn)\{e\}$	iteration
		$fn \mid fn = e$	see Sec. 4.2 for below
		$::fn \mid ::fn = e$	
		$mn(e_1, \ldots, e_n)$	
		$::mn(e_1, \ldots, e_n)$	
		$n::mn(e_1, \ldots, e_n)$	
		$e..mn(e_1, \ldots, e_n)$	
		$e\texttt{->}mn(e_1, \ldots, e_n)$	
		$e.mn(e_1, \ldots, e_n)$ at *tn*	
		$e; e$	
cst			constant

Figure 7: *Classages* Expressions

4. CLASSAGES SYNTAX

The core syntax for *Classages* top-level structures is defined in Fig. 6 and the core syntax for expressions is in Fig. 7. This "core" is not all of the syntax accepted by the *Classages* compiler: the current implementation also accepts Java expressions as it is based on an extensible Java compiler framework [22]. This choice is made intentionally as core *Classages* itself does not define useful features such as arrays, exceptions and rich primitive types. As *Classages* has its own object model, we might in the future choose to disable some Java features in the implementation, such as defining Java classes within a *Classages* program. For brevity, Fig. 6 also omits classage constructors. This more predictable construct is explained in the *Classages* homepage (weblink in Sec. 1).

4.1 Top-Level Structures

A program in *Classages* is composed of classages, with one of them being the bootstrapping classage with a special connector `Main`. The `ProteinFolding` classage in Fig. 2 has such a `Main`.

A classage has a unique name, and can be either defined from scratch, with constructors, local fields, local methods, and three kinds of interaction interfaces inside, or defined as the mixing of two classages. The resulting classage of the second case is called a *compound*. In addition to the basic grammar for mixing demonstrated in Sec. 2, as is used for combining the merged participating mixers to produce a new mixer, with a name specified by the as clause. The underlying process of producing the merged mixer is very similar to mixin composition. Renaming of interaction interfaces at the top level is also supported.

Within interaction interfaces, an import declaration specifies the type information for the method to be imported; an export declaration provides the method, with the standard syntax associating the method body directly with the export declaration. As an example, the `update` method has its body defined in the `Ctrl` connector of Fig. 4. For convenience, some sugared syntax is also provided. Also in Fig. 4,

```
export double getTA(int i) = ::getTA
```

declares the exported method `getTA` has its method body bound to the local method `getTA` of the classage. It is equivalent to

```
export double getTA(int i) { return ::getTA(i); }
```

.

Likewise,

$$\texttt{export}\ \tau\ \texttt{m}(\tau_1\ \texttt{x}_1, \ldots, \tau_k\ \texttt{x}_k)\ = \texttt{n::m}'$$

declares the exported method `m` has its method body bound to a method `m'` declared in interaction interface named `n`, where `n` might be a mixer, a singleton plugger, or a singleton connector. Syntactically, it is equivalent to

$$\texttt{export}\ \tau\ \texttt{m}(\tau_1\ \texttt{x}_1, \ldots, \tau_k\ \texttt{x}_k)\ \{\texttt{return n::m}'(\texttt{x}_1, \ldots, \texttt{x}_n);\}$$

When no ambiguity arises, programmers may use expression export τ m$(\tau_1$ x$_1, \ldots, \tau_k$ x$_k)$ directly, meaning the m being exported has its body defined in either of the aforementioned cases, and there could only be exactly one definition matching the method signature.

4.2 Expressions

Referring to local features always starts with symbol `::`, including local field read $::fn$, write $::fn = e$, and local method invocation $::mn(e_1, \ldots, e_n)$. In contrast, an expression can refer to interface features (import, export and connection-specific fields) directly by their names if the expression is also in the scope of the interface. This case involves connection-specific field read fn, write $fn = e$ and import/export method invocation $mn(e_1, \ldots, e_n)$. When the expression is not in the scope of the interface where the import/export method is defined, there are a few different forms of method invocation, depending on the nature of the interface:

- To invoke a method *mn* defined in a mixer, a singleton plugger or a singleton connector, $tn::mn(e_1, \ldots, e_n)$ is used, where *tn* is the name of the interface.
- To invoke a method *mn* defined in a generative plugger, $e..mn(e_1, \ldots, e_n)$ is used, where e will eventually be evaluated to a plugging handle.
- To invoke a method *mn* defined in a generative connector, $e->mn(e_1, \ldots, e_n)$ is used, where e will eventually be evaluated to a connection handle.
- To initiate an ephemeral interaction (Sec. 3.5), expression $e.mn(e_1, \ldots, e_n)$ `at` *tn* is used. For this form of invocation, *tn* must be a connector on objectage e that does not contain `import` and `state` declarations. This coincides with the fact that ephemeral interactions should always be stateless with no control dependencies.

Objectage instantiation is achieved by an expression `create`, identical to Java's `new`. The return value of this expression is an objectage reference. We have explained other expressions in previous sections.

5. CLASSAGE TYPING

Classages is a strongly typed language with explicit type declarations. There are three main type sorts: *objectage types*, *connection types* and *plugging types*. They type the three important first-class values respectively: objectage references, connection handles, and plugging handles. A programmer declares a plugging type by using a plugger name, and the underlying plugging type being used by the type system is the signature information for all invocable methods during the plugging interaction. Predictably, the methods of concern include all those declared either as an `import` or an `export` on the plugging interface where the plugging happens. Similarly, a programmer declares a connection type by using a connector name, and the underlying connection type being used by the type system is the signature information for all invocable methods during the connection interaction, plus type information for connection-specific fields. We now focus on a few important aspects of *Classages*' type system.

5.1 Objectage Types and Polymorphism

As with other OO languages, *Classages* has a distinction between classage types and objectage types. Unlike classages, an objectage's runtime behavior, in view of its peers, is solely dependent on its connectors. Thus, a programmer declares a type for an objectage reference by using a classage name, and the underlying objectage type used in the type system is the type information of all its connectors, each of which includes the connector's name and the method signatures of `import`s and `export`s declared inside. For convenience, we call the type information for each connector a *connector signature*. Thus, informally, an objectage type is a set of connector signatures.

This representation follows Java in the use of a class name to define an object type, but differs in other dimensions. Object subtyping in Java is intensionally defined by `extends` or `implements`, whereas *Classages* has a structural subtyping system at heart, and classage names themselves are not part of subtyping. The reason we take a different approach is to achieve maximum polymorphism: ontologically, we believe the essence of an objectage is how many ways it can communicate with peers, and what the ways are. Hence, as long as objectage $\mathcal{O}_1$ can support at least as many and compatible means of communications as $\mathcal{O}_2$, $\mathcal{O}_1$ should be allowed to appear anywhere $\mathcal{O}_2$ appears.

This intuition leads to the following design for objectage subtyping: $\mathcal{O}_1$ is a subtype of $\mathcal{O}_2$ iff $\mathcal{O}_1$ contains at least as many connector signatures as $\mathcal{O}_2$, and for any pair of connector signatures of the same name, the one contained by $\mathcal{O}_1$ (say $\mathcal{C}_1$) must be *more refined* than that of $\mathcal{O}_2$ (say $\mathcal{C}_2$). By "refined", we mean structural subtyping in support of both width subtyping and depth subtyping on connector signatures: (1) Width-wise, $\mathcal{C}_1$ may have more `export`s (covariant) and fewer `import`s (contravariant); (2) Depth-wise, for each pair of methods `export`ed from both $\mathcal{C}_1$ and $\mathcal{C}_2$, the one in $\mathcal{C}_1$ must have a signature being the subtype of that in $\mathcal{C}_2$ (covariant); likewise, for each pair of methods `import`ed from both $\mathcal{C}_1$ and $\mathcal{C}_2$, the one in $\mathcal{C}_2$ must have a signature being a subtype of that in $\mathcal{C}_1$ (contravariant).

5.2 Typing Interactions

Whether a *Classages* interaction (connection, plugging, mixing) can be soundly established (even at runtime) is always checkable at compile time, by typechecking the interface matching.

Intuitively, interactions between two parties are sound only when each party provides what the other party expects. In type terms, for any method declared as an `import` in one interface, the other interface must declare a method of the same name as an `export`, which at the same time has a signature being the subtype of that of the former. This can be thought of as a "method specialization" happening at interaction time.

5.3 Typing Parameter Passing

Classages has three fundamentally different interactions, and to achieve appropriate encapsulation, different policies for parameter passing are needed for the different interactions. A table summarizing all the cases is presented in Fig. 8.

First, a connection handle should not be passed across connections. For instance in Fig. 2 if the `Sequence` objectage were allowed to send connection handles to `MoveCtrl`, it might send over the one representing its connection with a `View3D` objectage. Thus, even though a `MoveCtrl` objectage does not have any connector to indicate its intention of communicating with an `View3D` objectage, a *de facto* interaction channel would be established. This would violate the principle that interactions must occur on explicit interfaces only. Note that this policy does not prevent an objectage reference from being passed over connections, and so the `Sequence` objectage can freely send an objectage reference of `View3D` to `MoveCtrl`: by analogy with in the real world, this is the standard "Mr. `MoveCtrl`, may I introduce Ms. `View3D`?" process, and is crucial for objectages to learn of each other. Once objectages are introduced, they need to connect with each other to have any meaningful interaction. By disallowing the passing of connection handles, we are disallowing masquerading: "Mr. `MoveCtrl`, can you pretend to be me and communicate with Ms. `View3D`?"

Second, a plugging handle should not be passed during a connection. Passing plugging handles to other objectages is the same to expose an objectage's internal representation,

Interaction \ Parameters	Primitive Data	Objectage References	Plugging Handles	Connection Handles
Mixing	OK	OK	OK	OK
Connecting	OK	OK	No	No
Plugging	OK	OK	downward OK	OK

Figure 8: *Classages* Parameter Passing

and violates the nature of encapsulation, access control, and the essence of ADT's. For instance, if a `Sequence` objectage were allowed to pass the plugging handle representing its interaction with an `AminoAcid` part, the receiver of this value, say `MoveCtrl`, would be able to tamper with the internal representation of `Sequence`. Worse, since `MoveCtrl` now would hold a reference, the lifetime of the `AminoAcid` instance would not just depend on `Sequence` alone. This in principle would violate the essence of UML's composition relationship, the aim of our support for plugging in the first place.

Third, a plugging handle should not be passed from a plugee to its plugger during a plugging interaction. Internal representations could be hierarchical: in Fig. 2(b), each `Sequence` objectage contains multiple `AminoAcid` parts as its internal representation, and an `AminoAcid` part might itself be composed of multiple `Atom` parts; *Classages* supports such hierarchical plugging. For each `AminoAcid` instance, the internal representation of how multiple `Atom` instances are structured together is an internal matter, and should not be tampered with by the `Sequence` objectage.

5.4 On Type Soundness

We have yet to formally prove the soundness of *Classages*, but it is structurally similar to our *Assemblages* module calculus [19] which was proved sound, and we are confident a sound type system can be designed.

In the presence of `disconnect` and `unplug` expressions, a connection handle or plugging handle may become *stale* because its connection has been disconnected/unplugged. Runtime checks are thus needed to make sure the handles are not stale.

Some runtime checks need to be performed for the use of singleton connectors and pluggers. At runtime they should be associated with at most one interaction at any given time. When expression $n::mn(e_1, \ldots, e_n)$ is evaluated, at least one connection/plugging should be alive on interaction interface n.

6. TECHNICAL ISSUES

In this section, we elucidate two technical aspects of *Classages*: overriding and message dispatch; and interaction interfaces for compound classages.

6.1 Overriding and Message Dispatch

Overriding and message dispatch are closely related issues in Java-like languages. The key innovation of *Classages* in this regard is it decouples the two issues, so that programmers can separate the concern of how the code is composed (overriding) from how an objectage should behave at runtime (message dispatch), and thus achieve maximum flexibility. We now use a very simple example – a class `C` with a method `m` – to answer how a classage can be defined such that (1) `m` is overridable; (2) a message targeted to `m` is statically dispatched; (3) a message targeted to `m` is dynamically dispatched.

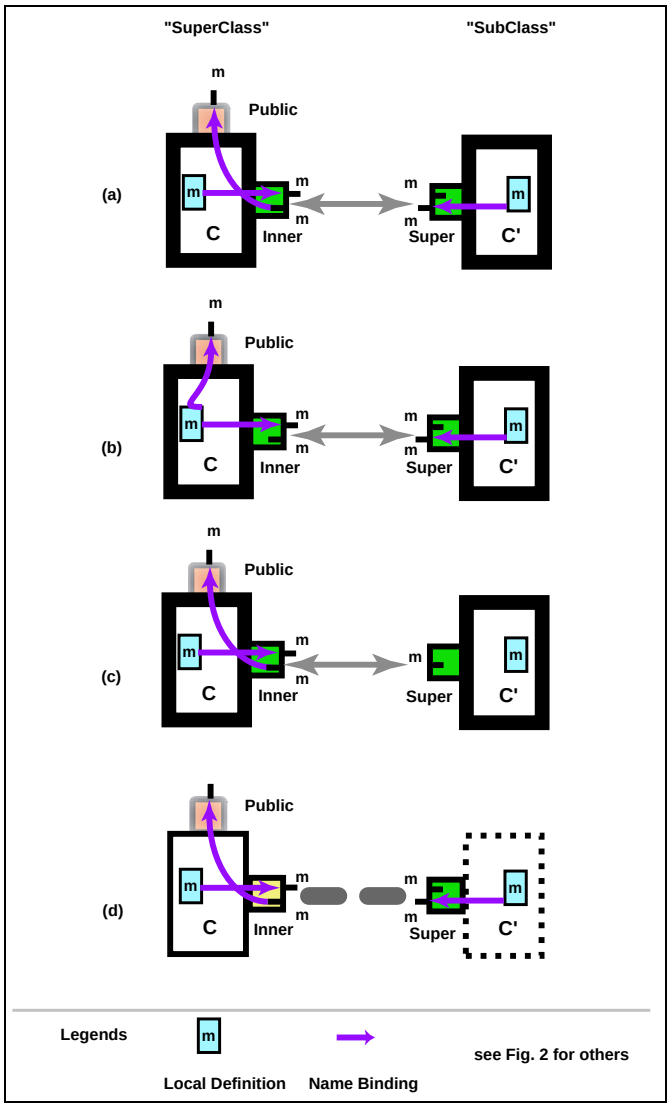

Figure 9: Overriding and Message Dispatch (a) static overriding and dynamic dispatch (b) static overriding and static dispatch (c) static overriding and dynamic dispatch, with no fragile base class problem (d) dynamic overriding and dynamic dispatch

The various cases are shown in Fig. 9. Note that in a static composition-based language, there is no distinction between a superclass and a subclass. We here only use it in the informal sense, analogous to a similar situation like in

Java. Also note that the "subclass" here is just the extension code, and needs to be composed with the "superclass" to get what is called a subclass in Java.

6.1.1 Overriding

In *Classages*, the question of whether a method is overridable is solely a concern for mixers, *i.e.*, interfaces for code composition. Instead of introducing an "overridable" modifier for mixer methods, an overridable method in a mixer is defined by *declaring the same method both as an* export *and an* import. The export is the default, but overridable, implementation and the import indicates the interest to take in an overriding method which will take precedence over the default exported. An example is shown in Fig. 9 (a), where method `m` is defined as an overridable method in mixer `Inner` of classage `C`. Anywhere in the code of `C` that `m` of `Inner` is referred to, the body of `m` is (1) the `export m` of `Inner` if mixer `Inner` is not matched up via mixing; otherwise (2) the export provided by the other classage involved in the mixing interaction—in Fig. 9 (a), this means the `export m` provided by mixer `Super` of classage `C'`.

6.1.2 Dynamic & Static Dispatch

On the other hand, dispatching messages from peer objectages is solely a concern for connectors, *i.e.*, runtime interfaces. Examples of dynamic dispatch and static dispatch for method `m` are shown in Fig. 9 (a) and (b) respectively. In (a), the `export m` in the connector `Public` is defined to be the `m` of `Inner` (referred to by expression `Inner::m`), *i.e.* the overridable method, while in (b) the `export m` in the connector is defined to be the local `m` (referred to by expression `::m`).

6.1.3 The Fragile Base Class Problem

We have just mentioned two cases where *other* objectages might invoke `m`. Note that even inside the body of classage `C`, the "self-calls" follow the same pattern: programmers can still use the same pair of expressions – either `Inner::m`() or `::m`() to distinguish whether the dynamically bound method or the statically bound one should be called. Interestingly, this obvious way of programming helps programmers avoid the Fragile Base Class Problem [30], the problem that self-calls defined in the superclass might be accidentally overridden by a subclass. By equipping a language both with dynamic dispatch and static dispatch, the "superclass" author can have the choice of whether overriding should happen.

This however is just part of the solution *Classages* provides. Indeed, any language that supports static dispatch can achieve similar effect explained earlier. A flip side of the Fragile Base Class problem is whether the "subclass" author can have a choice over whether overriding should happen. In *Classages*, even when the base class uses dynamic dispatch, the "subclass" can still use discretion to avoid accidental overriding, as is illustrated in Fig. 9 (c). Suppose when classage `C` is written, its author is unable to anticipate the overriding behaviors of its future "subclasses", and therefore defines the "self-calls" by using dynamic dispatch, so as not to lose expressiveness and generality. In *Classages*, the "subclass" author has a way of refusing overriding without making sacrifices over the runtime interface, by simply *not* export*-ing it on the* `Super` *interface.* Note that in C++, this is an impossible task.

6.1.4 Super and Inner

Readers might wonder how a Java-style `super` call can be made. In both Fig. 9 (a) and (b), notice that classage `C'` on its `Super` mixer also declares method `m` as both import and export, *i.e.*, overridable. The effect is one of *mutual overriding*, where `C` and `C'` can use each other's `m` method. Hence when `C'` is interested in calling method `m` defined in `C`, it can simply use `Super::m`.

Not only that, the essence of Beta-style [20] message dispatch is also supported. Fig. 9 (b) shows the case where message dispatch is always directed to `C`, and inside the body of the locally defined `m`, programmers can write `Inner::m` to call the one defined by `C'`.[3]

6.1.5 Dynamic Overriding

So far we have discussed static method overriding, the canonical form of overriding as found in Java and C++. *Classages* does not stop there: it also elegantly supports a form of *dynamic overriding, i.e.* object methods being dynamically changed at runtime. In Fig. 9 (d), note that *plugger* `Inner` of `C` here declares `m` as overridable, and `m` in connector `Public` is now defined as the method provided by this plugger. Similarly to the static overriding case, language semantics ensures anywhere in the code of `C` where `m` of `Inner` is referred to, the body of `m` is (1) the `export m` of `Inner` if plugger `Inner` is not plugged in with a plugee; and otherwise is (2) the export provided by the plugee. The difference between Fig. 9 (a) and (d) is that plugging in Fig. 9(d) is a dynamic interaction. From the perspective of peer objectages, the instance of `C` has an ability to dynamically change the body of its public methods.

6.2 Interfaces for Compound Classages

When explaining Fig. 3, we claimed the result of the two mixing processes is a compound classage no different from the `Sequence` classage in Fig. 2. In *Classages*, the difference between compound classages and atomic classages are only syntactical; semantically, a compound classage can always be "flattened out" and is equivalent to an atomic classage with mixers, pluggers and connectors. In fact, this is also how the compiler currently implements it.

With mixins, Traits [27] and many other existing code composition systems, such a "flattening-out" process would be fairly standard, if it were not for pluggers and connectors. Pluggers and connectors are interfaces for runtime interactions and mixing interactions do not directly happen on them, but since they are associated with classages, mixing must consider how a compound classage should carry over the pluggers and connectors of the classages participating in mixing.

6.2.1 Intuition

We now consider a compound `C` from the mixing of `A` and `B`. The following discussion only focuses on the case for connectors. The case for pluggers is nearly identical.

Each declared connector signifies one means for its runtime instance to communicate with the outside world. Intuitively, objectages instantiated from the compound classage of `A` and `B` should be able to communicate with the outside

[3]There is a weakness in modeling the use of **inner** inside a Beta-class at the bottom of the hierarchy from this view, because the meaning of the corresponding expression `Inner::m` could be wrong. But this case also needs special care in Beta.

world both by A's means and B's means, *i.e.* the compound should have connectors both defined by A and by B.

One special case is A and B might declare connectors of the *same name*. Only two possibilities exist: accidental name clash or intentional name correspondence. For the first case programmers can just rename connectors so that name clash does not happen at mixing time. The second case is interesting because on a high level, a connector name can be viewed as the "keyword" of the "communication policies", and the overlapping of connector names signifies A and B intend to be involved in interactions of the same nature. An example is when mixing happens between two classages both with a Public connector indicating interactions with the "general public". Such a name correspondence is usually not an accidental clash. The mixing process should allow the merging of such connectors, indicating their common interest in communicating with the "general public". Since a connector is composed of imports, exports and connection-specific fields, we need to separate our discussion into three parts to make clear what we mean by merging connectors.

First, declaring an import on a connector shows an expectation the classage has for the outside at runtime. Suppose now A has a connector Cn with import m defined, and yet in B's connector Cn, m is not an import. If the resulting compound C were to include m as an import, it would raise B's expectation for the outside world. One can imagine a connect o with Cn >> Cn' typechecked in B would fail in C, if objectage o in the expression happened not to provide m. On the other hand, if C were not to include m as an import, it would lower A's expectation for the outside world. But remember when A is written, it has already assumed the expectations (imports) will be satisfied when connection is established. Thus it might contain code such as h->m() (h is a connection handle on connector Cn), which would fail to typecheck if C did not import m. This shows that A and B must always have the same import declaration in connector Cn to be mergeable in a type-safe way.

Second, declaring an export on a connector simply shows a classage's willingness to contribute to the outside world. Thus, as long as A and B do not compete to contribute the same method, the merged connector should have all export method from A and from B. By "compete" we mean the two methods are not distinguishable, which would confuse the party receiving the "contributions".

Third, declaring a connection-specific field simply shows an objectage holds some private data to the connection to be established on the holding connector. If A and B each hold some connection-specific field, they should both show up in the merged connector. In the case of name clash, they should be α-renamed, since such a name clash is purely accidental from the perspective of A and B.

6.2.2 The Classages Solution

In summary, there are two conditions a mixing process must satisfy: (1) If both A and B define connectors by name Cn, the two connectors in concern must have identical import declarations. (2) If both A and B define a connector by name Cn, the two connectors in concern cannot export two methods which are not distinguishable from overloading sense.

The following describes how connectors of the two mixing-participating parties are carried over to the compound:

- If A or B (but not both) defines a connector by name Cn, the connector Cn also belongs to C.
- If both A and B defines a connector by name Cn and the conditions above are satisfied, C should also contain a connector Cn, with import declarations to be either of the two parties (they are the same anyway), with export merged, and connection-specific fields merged with free α-conversion.

How mixers are carried from the mixing-participating parties to the compound is standard: they are either matched and consumed (via classage C = A + B with $\overline{tn >> tn}$) or matched and re-exported as a new mixer of the compound (via classage C = A + B with $\overline{tn >> tn \text{ as } tn}$), or unmatched and carried over to the compound.

7. IMPLEMENTATION

A prototype *Classages* compiler has been implemented using the Polyglot compiler framework [22]. The extensibility of Polyglot allows programmers to reuse Java's existing features. The current implementation provides its own classage/objectage model and relies on Polyglot's base implementation of Java to provide useful language features such as arrays, exceptions and rich primitive types.

The implementation covers all language features introduced in the syntax (Sec. 4), including translation to target Java code and the type system. All *Classages* code examples used in this paper can be successfully compiled. The current status of implementation is that it lacks explicit library support, and advanced features such as reflection are not included.

The compilation process consists of several standard passes including parsing, type-building, disambiguation, typechecking and Java code generation, performed by a series of visitor traversals on the AST. An important issue is *when* compound classages are "flattened-out" into atomic classages. Since mixing is a purely static interaction, the compiler could in theory perform flattening right after parsing. This approach however would lead to duplicate compilation: when a classage A is also used to form a compound B, a flattening of B early on would lead to the code inside A being typechecked twice. Our compiler hence does not flatten compound classages until the last pass, when the target code is produced. Compared with the specification of Sec. 6.2, the implementation is a bit more complicated due to the need for $\alpha-$conversion of local fields and methods during flattening. The classage $cn = cn + cn$ with $\overline{n >> n}$ syntax involves consumption of mixers; the current implementation handles it by giving the consumed mixers some internal names not overlapping with programmers' name space. The classage $cn = cn + cn$ with $\overline{n >> n \text{ as } n}$ syntax involves interface renaming.

7.1 The Translation

Each classage is translated into a top-level Java class, and its interaction interfaces are translated into Java inner classes, the use of which eases up implementation issues such as scoping. The Java top-level class contains factory methods for the creation of each inner class.

All top level classes contain the fields cstore recording live connections, and pstore recording live pluggings. Bookkeeping operations on these fields are defined in a common Java interface SYS_CLASSAGE that all top-level classes implement. All inner classes contain one field named other: recording the other party communicating on the interaction

interface. Recall both connections and pluggings happen between a *pair* of interaction interfaces. Bookkeeping operations for this field are defined in a common Java interface `SYS_I` all inner classes implement. Inside each inner class, every export method is mapped to a public method, while every import method is implemented as a redirection to the method on `other` by the same name.

A `connect o with n1 >> n2` expression is translated to Java code instantiating the two inner classes representing `n1` and `n2` respectively, and updating the `other` fields of the two and the live connection stores. Without considering the subtyping intricacies discussed below in Sec. 7.2, the translation is:

```
x = this.factorymethod_n1();
y = o.factorymethod_n2();
x.other = y;
y.other = x;
this.cstore.add(y);
o.cstore.add(x);
```

A plugin expression can be similarly translated. Other expressions are relatively straightforward.

7.2 Implementing Structural Subtyping in Java

One of the most challenging implementation issues is how *Classages*' structural subtyping system is to be implemented in Java, which uses nominal and not structural subtyping. Consider the following example, where `B` is a subtype of `A`, by structure:

```
classage A {
    connector Q {export void f(int x){...}}
}
classage B {
    connector Q {
        export void f(int x) {...}
        export void g(double x){...}
    }
    connector Z {...}
}
classage C {
    connector P {...}
    int m(A x) {
        P c = connect x with P >> Q;
        c->f(3);
    }
}
```

The tricky issue is the polymorphism supported by structural subtyping: local invocation `::m(b)` in `C` should typecheck, where `b` is an objectage of `B`. To make the same type-checkable *Classages* expression get through the Java typechecker, a naive implementation might just use casting. But Java's casting only allows casts up and down inheritance hierarchies, and casting one class to an unrelated class will fail. Some tricks can be attempted to build more nominal subtype relationships by introducing additional interfaces and implements relationships, but these approaches give incomplete results, especially considering Java does not support depth subtyping with its interface mechanism.

7.2.1 Our Solution

We first state the guiding rules of our solution:

R1. Any objectage variable declaration is translated to a variable declaration with top type `SYS_CLASSAGE` in the target Java program. This includes variables in the form of method formal parameters, fields and local variables.

R2. If a classage has an interaction interface named `A`, its implementing top-level Java class must implement a top-level one-method Java interface containing method void `factorymethod_A`().

R3. If an interaction interface contains an export method named `m`, its implementing Java inner class must implement a top-level one-method Java interface containing the signature of method `m`, where, as per **R1** above, all formal parameters of `m` with objectage types are replaced with the type `SYS_CLASSAGE`.

R1 is applied to ensure expressions involving the use of objectage subtyping should not be translated to target code rejected by Java's type checker. In the earlier example, if the formal parameter `x` of `m` is translated with a top type, one can imagine that invocation `::m(b)` can always be translated as a Java-typecheckable expression, even when `b` is an objectage of type `B`.

Now consider the translation of `connect x with P >> Q`. In Sec. 7.1 we presented an incomplete translation, containing the Java statement `y = o.factorymethod_n2()`. Notice that now `o` according to **R1** has a top type, so the invocation does not typecheck in Java. One might be tempted to use downcasting, and fixing the statement in the form such as `y = ((A)o).factorymethod_n2()`. This approach does get around Java's static type checker, but at runtime, it will likely throw an exception due to invalid downcasting: although `o` is declared to have top type in the method of `m`, it can still be instantiated with an object with type, say `B`. As long as `B` is not nominally a subtype of `A`, the downcast will fail at runtime.

R2 is introduced to avoid runtime exceptions at downcasts. Using this rule, the same statement can be translated as `y = ((I)o).factorymethod_n2()`, where `I` is the one-method Java interface containing method `factorymethod_n2`. Since the name `n2` is known locally, such a translation also does not depend on global information. Such a downcast always succeeds, because to ensure structural subtyping, any `o` must contain interaction interface `n2`, and hence must have implemented the same interface according to **R2**.

R3 is introduced in a similar vein as **R2**, only in this case on a different level: *Classages* objectage subtyping happens on two levels, with one level ensuring the subtype has at least as many as connectors as the supertype, and the other ensuring within the same connector, the methods also conform to the subtyping constraint. **R3** is useful for the method level.

7.3 On Efficiency

Since the solution here involves significant downcasting, the compiler will suffer from inefficiencies, but it will allow us to build a functional prototype with passable performance, without the need to get involved in JVM modifications.

Independent of the choice of target languages, the inclusion of some expressive language features in *Classages* inherently leads to some decisions being made dynamically. *Classages* structural subtyping will lead to dynamic lookup for method mn in expression e->$mn(e_1, \ldots, e_n)$, where a connection e is invoked. When n is a singleton plugger name,

```
abstract class A {
    abstract public void a();
    protected void c() {...}
    private void l() {...}
}
interface Itr {
    public void b();
}
class B extends A implements Itr {
    final public void a() {...}
    public void b() {...};
    final protected void c() {...}
}
```

Figure 10: A Typical Java Snippet

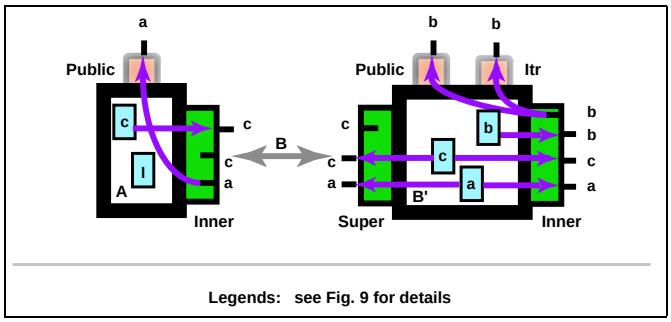

Figure 11: A *Classages* Illustration of Fig. 10

expression $n::mn(e_1, \ldots, e_n)$ also needs to dynamically decide whether the default export mn or the overridden one should be invoked, depending on whether the plugger is currently plugged in with a classage.

8. ENCODING JAVA IN CLASSAGES

In this section we show how standard notions of class, object, inheritance, *etc* can be encoded in *Classages*. As an example, a Java snippet and its corresponding *Classages* implementation is shown in Fig. 10 and Fig. 11 respectively. (For simplicity, we have left out the technicality that class `Object` is the default root class of all Java classes.)

Classages can encode Java class, interface, methods with various modifiers (public, protected, private, final and abstract), private fields, overriding, dynamic dispatch, super invocation and subtyping. Non-private fields can always be modeled with a pair of getter/setter methods.

8.1 Intuition

Intuitively, a simple Java class without interface declarations can be thought of as defining three interactions with the outside world: (1) how it interacts with its superclass, (2) how it interacts with its subclass, and (3) how its instance communicates with peers at runtime. To support these three interactions, the encoding classage will use for (1) a mixer `Super`, for (2) a mixer `Inner`, and for(3) a connector `Public`. In Java, a protected method can be invoked inside the body of subclasses and cannot be invoked at runtime by peers; such a method in a *Classages* context will be exported to the `Inner` mixer, but not to the `Public` connector. Comparatively, a public method should be exported on both, and a private method should exported on neither. Modifier abstract means a method must be imported from a subclass, hence in *Classages* an import should be declared in `Inner` for the method. The essence of final is to avoid overriding, and this can be achieved by not including the method as an import on `Inner`. To support overriding, the actual encoding is more complicated, in a way we explained in Sec. 6.1; see that section for the intuitions.

As explained in Sec. 3.4, Java's interface is not intended to strictly enforce encapsulation, but is meant to support subtype polymorphism. In that sense, as long as the subtyping relations of Java can be modeled in *Classages*, we do not need to encode this type-level language construct at all. The subtyping relation can be easily preserved in the encoding: *Classages* has a structural subtyping system, and objectage subtyping is only concerned with connectors. It can be easily seen in the encoding connector `Public` exports all public methods of a Java class, and it must contain all methods declared in a Java interface implemented by that Java class.

Java-style inheritance is encoded in *Classages* by mixing. In Java, a subclass `B` is defined based on the pre-knowledge of what superclass `A` it inherits from. In a composition-style language like ours, this pre-knowledge is not necessary, because the extension itself, *i.e.* what `B` has defined in its own body, could be a unit of reuse as well. Thus, encoding a Java-style subclass involves two steps: first define the extension using a classage, say $\texttt{B}'$, and then define `B` as mixing of `A` and $\texttt{B}'$. This is exactly what is done in Fig. 11.

8.2 Encoding

We now formally define the encoding of Java classes into classages. The encoding is defined on a linear inheritance hierarchy only, but the encoding has an unambiguous mapping onto a hierarchy tree. We start off by defining a few functions:

$$
\begin{array}{rcl}
\iota(\mathcal{C}, N) & ::= & \text{imports in interface } N \text{ of classage } \mathcal{C} \\
\epsilon(\mathcal{C}, N) & ::= & \text{exports in interface } N \text{ of classage } \mathcal{C} \\
\texttt{public}(C) & ::= & \text{public methods in Java class } C \\
\texttt{protected}(C) & ::= & \text{protected methods in Java class } C \\
\texttt{final}(C) & ::= & \text{final methods in Java class } C \\
\texttt{abstract}(C) & ::= & \text{abstract methods in Java class } C \\
\texttt{override}(C) & ::= & \text{overriding, } i.e. \text{ not newly defined, methods in } C
\end{array}
$$

Given a linear Java inheritance hierarchy C_1, C_2, $\ldots$, C_n, where C_1 is the root class and C_i is a subclass of C_{i-1} for $2 \leq i \leq n$, the *Classages* encoding of C_i is a classage with a connector Public, and two mixers Inner and Super. For the root class C_1, its encoding $\mathcal{C}_1$ has the following structure:

$$
\begin{array}{rcl}
\iota(\mathcal{C}_1, \mathsf{Public}) & = & \emptyset \\
\epsilon(\mathcal{C}_1, \mathsf{Public}) & = & \texttt{public}(C_1) \\
\iota(\mathcal{C}_1, \mathsf{Super}) & = & \emptyset \\
\epsilon(\mathcal{C}_1, \mathsf{Super}) & = & \emptyset \\
\iota(\mathcal{C}_1, \mathsf{Inner}) & = & (\texttt{public}(C_1) \cup \texttt{protected}(C_1)) \cap \overline{\texttt{final}(C_1)} \\
\epsilon(\mathcal{C}_1, \mathsf{Inner}) & = & (\texttt{public}(C_1) \cup \texttt{protected}(C_1)) \cap \overline{\texttt{abstract}(C_1)}
\end{array}
$$

This definition has largely been explained by the informal explanations earlier. Note that when both import and export of an interaction interface are empty, the interface does not exist: in our context, the root class does not have a Super mixer.

For any class C_i on the hierarchy where $2 \leq i \leq n$, we first compute a classage Δ_i, and then obtain $\mathcal{C}_i$ (the encoding classage for C_i) by mixing $\mathcal{C}_{i-1}$ and Δ_i, with the `Inner`

mixer of the former matching the `Super` mixer of the latter. Δ_i is defined as follows:

$$
\begin{aligned}
\iota(\Delta_i, \mathsf{Public}) &= \emptyset \\
\epsilon(\Delta_i, \mathsf{Public}) &= \texttt{public}(C_i) \cap \overline{\texttt{override}(C_i)} \\
\iota(\Delta_i, \mathsf{Super}) &= \epsilon(\Delta_{i-1}, \mathsf{Inner}) \\
\epsilon(\Delta_i, \mathsf{Super}) &= \iota(\Delta_{i-1}, \mathsf{Inner}) \\
\iota(\Delta_i, \mathsf{Inner}) &= (\texttt{public}(C_i) \cup \texttt{protected}(C_i) \cup \epsilon(\Delta_i, \mathsf{Super})) \cap \overline{\texttt{final}(C_i)} \\
\epsilon(\Delta_i, \mathsf{Inner}) &= (\texttt{public}(C_1) \cup \texttt{protected}(C_1) \cup \iota(\Delta_i, \mathsf{Super})) \cap \overline{\texttt{abstract}(C_1)}
\end{aligned}
$$

This definition is inductive because Java's inheritance does not just allow a direct subclass to use or override a method defined by the superclass, it also allows use and overriding by any *indirect* subclass on the inheritance hierarchy. In a *Classages* context, this means a superclass method, no matter being used/overridden by the direct subclass or not, must be forwarded from the `Super` mixer to `Inner` mixer, since an indirect subclass might still use/override it. The presentation here is simply a mechanical encoding to demonstrate expressiveness; in reality, when one programs in *Classages*, one programs in a *Classages* way: Java's coding pattern is also optimized for inheritance. Programming in a language featuring code composition, programmers should not think of constructing deep hierarchies in the first place. For instance, by coding a labeled color point, a typical Java programmer will create an inheritance hierarchy: `Point`, `ColorPoint`, `LabeledColorPoint`, where as a *Classages* implementation would naturally be the composition of three classages: `Point`, `Color` and `Label`.

Java's method invocation is encoded by ephemeral invocations on default connector Public. Other expressions, such as new and field access can be encoded via the direct analogues in classages.

9. PROGRAMMING IN CLASSAGES

This section explores a few practical issues of *Classages* programming. The goal here is to illustrate how the language performs in not-so-obvious or seemingly-not-supported programming scenarios. It also aims to give readers a better understanding of the language expressiveness and potential limitations.

9.1 The Need for Self Reference

Self reference – such as Java's expression this – is a central issue for OO languages. this has two distinct purposes in OO languages: 1) for sending messages to the object itself; 2) to pass this to other objects to allow them to call back.

Classages does not currently include this in the language syntax. This decision was made because the two aforementioned factors bolstering the inclusion of this in an OO language are already supported in *Classages*: see Sec. 6.1 for support of self-messaging and Sec. 3.5 for callbacks. That said, there are still a few programming situations which are easy in Java that become harder in *Classages*: for instance, an objectage cannot register itself with another objectage, but instead has to rely on a third party holding its objectage reference to register it. If more engineering practices point to such difficulties, we may decide to add this to the language explicitly. But even if that were to happen, this in *Classages* will still play a more minor role in *Classages* than in mainstream OO languages.

9.2 Programming Collection Classes

We now explain how a Java programmer's need to define and use collection classes/interfaces such as `List` and `Set` is met in *Classages*.

We first identify a few problems with Java's solution. A whole-part relationship between `Sequence` and `AminoAcid` in Fig. 1 will typically be implemented in Java by having a `Sequence` object holding a collection object, say `LinkedList`, which subsequently holds a number of `AminoAcid` objects. To refer to each `AminoAcid` object, some `Iterator` object also needs to be created. Such a solution demonstrates an impedance mismatch between design and implementation, in the sense that a simple 1:n binary relationship between `Sequence` and `AminoAcid` demonstrated by UML suddenly becomes complicated interactions among four classes: `Sequence`, `AminoAcid`, `LinkedList` and `Iterator`. Such a discrepancy not only introduces problems in understanding, but also leads to subtle issues of alias protection [2].

Classages directly supports connectors and pluggers with multiplicity. Thus, a 1:n relationship between `Sequence` and `AminoAcid` in UML is faithfully implemented by only two classages. Functionalities represented by Java's `List` are directly supported: Java `List`'s `add` can be analogously thought of as the plugin/connect expressions, Java's `remove` on lists as the unplug/disconnect expressions, and iterations as the forall expression.

This solution covers the cases where programmers need relationship multiplicity but do not care what specific data structure is used to achieve it. However, there are situations where programmers need to pick a specific data structure, for instance, choosing a `HashMap` over a `LinkedList` to achieve efficiency. *Classages* also support the definition and use of classages analogous to Java's `HashMap`: the analogous classage `HashMappage` would have some connector(s) to define how the hash map is maintained, including element addition, deletion and lookup. A second generative connector of `HashMappage` can support iteration over the map; there is no need to create a distinct `Iterator` object. A client objectage can communicate with a `HashMappage` objectage by establishing connections.

9.3 Self Mixing/Plugging/Connecting

Classages can be mixed with themselves, as long as they have matchable mixers. For instance, given a classage `A` with two mixers `In` and `Out`, where the import and export of these two mixers complement each other, it is possible to create a compound `AA` as follows:

```
classage AA = A + A with In >> Out
```

The resulting classage `AA` will still have two mixers, the `Out` mixer carried over from the first `A`, and the `In` mixer carried over from the second `A`.

An analogous scenario works for plugging as well: an objectage can plug in a classage of the same type to its runtime, as long as they have a matched plugger/mixer pair. For instance, given a classage `B` with one plugger `Part` and one mixer `Whole`, where the import and export of these two interfaces complement each other, it is fine to evaluate the following expression inside an objectage of `B`:

```
plugin B with Part >> Whole
```

This particular coding pattern is useful to implement the **Composite** design pattern, where the whole can itself be a part of a whole of the same type.

Two objectages generated from the same classage can be obviously connected together, analogously with how two Java objects of the same class can communicate.

9.4 Reusable and Composable Connectors and Pluggers

In a realistic language where library support is the norm, it is desirable to predefine a few commonly used connectors and pluggers as reusable building blocks in the library, and users can compose their own connectors and pluggers out of them. Although first-class connectors and pluggers are not supported directly, they can easily be encoded in *Classages*. We now explain the case for connectors; the case for pluggers is identical.

For reusability, consider the following scenario Java programmers often encounter: an object `MyButton` should implement a `MouseListener` interface to communicate with the event source, inside which callback methods such as `mousePressed`, `mouseClicked` are defined. In *Classages*, `MouseListener` is naturally coded as a connector of the `MyButton` objectage. The question is then how a *Classages* library can define a reusable `MouseListener` connector so that the `MyButton` programmer does not have to define default behaviors of `MouseListener` from scratch. A reusable `MouseListener` can be encoded by defining a classage, say `MouseListenerage`, with one connector `MouseListener`, and one mixer, say, `MouseListenerMix`, where the mixer defines default behavior which can be overridden by mixing with other classages and the connector is defined as in Sec. 6.1. Classage `MyButton` can thus be formed as the compound of `MouseListenerage` and a classage implementing the button functionalities. Such an encoding of reusable connectors also facilitates connector composibility.

10. RELATED WORK

That interactions always happen on interfaces is the norm in module and component system research. In this regard, *Classages* is a way of cross-over thinking from module systems and components to the level of objects. *Classages* is especially close to our module calculus *Assemblages* [19]. Despite the structural similarity between the two, there is a fundamental semantical difference due to the difference between modules and objects. For instance, in *Classages* plugging is designed to achieve better object encapsulation and lifetime management; mixing is closely compared with inheritance; connecting is used to model association relationships. All these issues, together with technical issues such as overriding and object polymorphism, are not concerns of a module system. This comment also applies to comparisons with other module and component systems, which we do not list one by one. A few systems that particularly affect our design of interaction interfaces, bi-directionality for instance, include Units [14], CORBA [23], Cells [24] and ArchJava [4]. Interested readers can refer to [19] for more detailed discussions. In software engineering community, a few architectural description languages [5, 21] also lay emphasis on interactions between different software parts, but their focus is on modeling high-level software architectural constraints, not designing fine-grained language constructs such as classes and objects directly used by programmers.

On the object level, mixin systems [9, 15] address the issue of class composition as a replacement of inheritance. Classage mixing is in a similar vein, but *Classages* separates the interfaces for static behaviors from interfaces for dynamic behaviors, and each mixin only has one interface playing dual roles. In this regard, *Traits* [27] are also designed with a separation of concern in mind: *traits* are only used for code composition, and are not instantiatable. Instantiation can only happen to a second language construct, *classes*, to glue *traits* together. *Classages* can use one language construct to model what both *traits* and *classes* are modeling, and still preserve separation of concerns, by declaring different kinds of interfaces on a single classage.

Encapsulation has always been a goal of OO languages. This can be traced back to the design of object itself and ADT. Recently, *Encapsulation Policies* (EP) [28, 26] have been defined which allow a class to define multiple EPs, specifying how it can be reused or its instances can be communicated at runtime. *Classages* can also allow a class to define multiple policies for its encapsulation, by declaring multiple mixers, pluggers and connectors. If the *Classages* principles in Sec. 3 are used to evaluate EP, it has a focus on the principle of least privilege (PRINCIPLE 4), and a separation between statics and dynamics (PRINCIPLE 1). EP does not single out the case that one object might encapsulate another object as its part. All objects are peers. Since the project is less focused on the interaction aspect of OO languages, encapsulation policies are not bi-directional.

Ownership types [10, 8] and alias protection [3, 11] uses type annotations for protecting an object's internal representation, with the ultimate goal that an object cannot be accessed from outside its owner. In *Classages*, plugging interaction defines an owner-ownee relationship, and the type system's restriction over parameter passing (see Sec. 5) determines first-class plugging handles cannot be passed outside its owner, which is the same as the inaccessibility from outside the owner. By distinguishing plugging handles and connection handles, *Classages* also separates the "internal references" within an objectage, and the "external references" between objectages, and thus can also achieve the interesting property of "external uniqueness is unique enough" [11]. As this paper is not focused strongly on the topic of ownership, we do not model more advanced notions of ownership such as borrow expression in [11], and lent in [3]. The lifetime constraint between a whole and a part in a plugging interaction makes *Classages* loosely related to region-based memory management [31], where the part can be viewed as holding a region with a shorter lifetime than that of the whole.

JACQUES [12] is a model proposed to support so-called environmental acquisition. It aims to model UML's aggregation associations, loose whole-part relationships where the part might outlive the whole, and JACQUES focuses on how the part can import fields and methods from the whole, the environmental acquisition. *Classages* can model two distinct forms of aggregation, plugging or connecting. For UML composition associations, plugging is an obvious choice, but plugging restricts external objectages from possessing a reference to the part so that a lifetime constraint can be guaranteed. If this external access were needed, a connector can be used instead of a plugger to model the aggregation, and in general our connectors can be used to model all of the whole-part relationships of JACQUES. JACQUES's acquires declaration can be modeled by declaring imports in the part connector, and JACQUES's contains declaration can be analogously thought of as the connector on the whole. JACQUES

demands that each class declare what classes are allowed to be a class's whole (using contained). This intensional approach is not taken in *Classages*: any two objectages with matched connectors can form the relationship.

UML allows designers to declare *association classes* to represent the relationship between two classes. At programming language level, Rumbaugh [25] first pointed out the importance of supporting first-class relationships, but mainstream OO languages' support for relationships is limited. Recently, as a reverse engineering effort, a tool [17] was proposed to recover binary relationships from Java. A few projects, including complex associations [18], first-class dependencies [13], and most recently, RelJ [6] have more explicit support for relationships than ours: relationships in these projects are not just first-class values as in *Classages*, they can also be independently defined by programmers as a top-level construct. Comparatively, our approach is more lightweight, but still covers the rich aspects of relationships as these projects cover.

Classages has syntax for directly support concepts embodied in several of the design patterns. Declaring explicit interfaces for classages is related to **Facade**. The design of interfaces with both imports and exports is a key component of **Observer**, allowing maximum de-coupling of interacting parties. By declaring imports on mixing interfaces, methods can be defined by other classages, which corresponds with **Template Method**. Plugging interaction is directly related to protecting internal representations. In Fig. 2 plugging has been used to facilitate **Memento**, by protecting the internal state. Programs using the **Composite**, **Strategy**, **State** patterns gain more direct support in *Classages*.

For code reuse, *Classages* uses composition in place of inheritance. Although we have shown inheritance is encodable in *Classages* from a language perspective, it is indeed debatable whether composition can totally replace inheritance from the software engineering perspective. This perhaps can only be answered by the mass of programmers, but some experiments, *e.g.* refactoring Smalltalk's collection hierarchy using Traits [7], indeed have shown promising signs for composition. Another question that needs practical *Classages* programming experience to decide is how much all the interfaces on classages get in the way compared to how much they help.

11. CONCLUSION

Classages models three fundamental interactions within the object world: static composition, peer-to-peer communication and whole-part interaction. Despite the fundamentally different nature of the three, they are unified under the same rationale: each interaction involves two parties; each party defines explicit interfaces on what it contributes to the interaction and what it expects from the interaction, and a successful interaction is established if the interfaces from both parties match each other. Despite the wide range of topics *Classages* touches upon, this unified view can both keep the formal language core small for reasoning, and help programmers learn the language quickly.

In the future, we are interested in exploring some language features the core language may be able to positively impact, in particular concurrency (see Sec. 3.7), serialization, and garbage collection (see Sec. 3.2). The current implementation does not place efficiency as the primary goal; improvement in this regard will become an immediate task.

12. REFERENCES

[1] Abadi, M., and Cardelli, L. *A Theory of Objects*. Springer, 1996.

[2] Aldrich, J., and Chambers, C. Ownership domains: Separating aliasing policy from mechanism. In *Proceedings of the 18th ECOOP* (2004), pp. 1–25.

[3] Aldrich, J., Kostadinov, V., and Chambers, C. Alias annotations for program understanding. In *OOPSLA '02*, pp. 311–330.

[4] Aldrich, J., Sazawal, V., Chambers, C., and Notkin, D. Language support for connector abstractions. In *Proceedings of the 17th ECOOP* (2003), pp. 74–102.

[5] Allen, R., and Garlan, D. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology 6*, 3 (1997), 213–249.

[6] Bierman, G., and Wren, A. First-class relationships in an object-oriented language. In *FOOL '05: Proceedings of the 12th International Workshop on Foundations of Object-oriented Languages*.

[7] Black, A. P., Schärli, N., and Ducasse, S. Applying traits to the smalltalk collection classes. In *OOPSLA '03*, pp. 47–64.

[8] Boyapati, C., Liskov, B., and Shrira, L. Ownership types for object encapsulation. In *POPL'03*, pp. 213–223.

[9] Bracha, G., and Cook, W. Mixin-based inheritance. In *Proceedings of OOPSLA/ECOOP'90* (1990), pp. 303–311.

[10] Clarke, D. *Object Ownership and Containment*. PhD thesis, University of New South Wales, July 2001.

[11] Clarke, D., and Wrigstad, T. External uniqueness is unique enough. In *Proceedings of the 17th ECOOP* (2003), pp. 176–200.

[12] Cobbe, R., and Felleisen, M. Environmental acquisition revisited. In *POPL'05*, pp. 14–25.

[13] Ducasse, S., Blay-Fornarino, M., and Pinna, A.-M. A reflective model for first class dependencies. In *OOPSLA'95*, pp. 265–280.

[14] Flatt, M., and Felleisen, M. Units: Cool modules for HOT languages. In *PLDI'98*, pp. 236–248.

[15] Flatt, M., Krishnamurthi, S., and Felleisen, M. Classes and mixins. In *POPL'98*, pp. 171–183.

[16] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[17] Guéhéneuc, Y.-G., and Albin-Amiot, H. Recovering binary class relationships: Putting icing on the uml cake. In *OOPSLA'04*, pp. 301–314.

[18] Kristensen, B. Complex Associations: Abstractions in Object-Oriented Modeling. In *OOPSLA'94*, pp. 272–283.

[19] Liu, Y. D., and Smith, S. F. Modules with interfaces for dynamic linking and communication. In *Proceedings of the 18th ECOOP* (2004), pp. 414–439.

[20] Madsen, O. L., Møller-Pedersen, B., and Nygaard, K. *Object-oriented programming in the BETA programming language*. ACM Press/Addison-Wesley Publishing Co., 1993.

[21] Medvidovic, N., Rosenblum, D. S., and Taylor, R. N. A language and environment for architecture-based software development and evolution. In *Proceedings of the 1999 International Conference on Software Engineering*, pp. 44–53.

[22] Nystrom, N., Clarkson, M. R., and Myers, A. C. Polyglot: An extensible compiler framework for java. In *Proceedings of the 12th International Conference on Compiler Construction* (2003), pp. 138–152.

[23] Object Management Group. *The Object Management Group*. URL: http://www.omg.org.

[24] Rinat, R., and Smith, S. F. Modular internet programming with cells. In *Proceedings of the 16th ECOOP* (2002), pp. 257–280.

[25] Rumbaugh, J. Relations as semantic constructs in an object-oriented language. In *OOPSLA'87*, pp. 466–481.

[26] Schärli, N., Black, A. P., and Ducasse, S. Object-oriented encapsulation for dynamically typed languages. In *OOPSLA'04*, pp. 130–149.

[27] Schärli, N., Ducasse, S., Nierstrasz, O., and Black, A. P. Traits: Composable units of behaviour. In *Proceedings of the 17th ECOOP* (2003), pp. 248–274.

[28] Schärli, N., Ducasse, S., Nierstrasz, O., and Wuyts, R. Composable encapsulation policies. In *Proceedings of the 18th ECOOP* (2004), pp. 26–50.

[29] Srinivasan, R., and Rose, G. Linus: A hierarchic procedure to predict the fold of a protein. *PROTEINS: Structure, Function, and Genetics 22* (1995), 81–99.

[30] Steyaert, P., Lucas, C., Mens, K., and D'Hondt, T. Reuse contracts: managing the evolution of reusable assets. In *OOPSLA'96*, pp. 268–285.

[31] Tofte, M., and Talpin, J.-P. Region-based memory management. *Information and Computation 132*, 2 (1997), 109–176.