Chunk's Combinatory Algebra Definition

Nicolás Scarcella, Mariana Matos, Leonardo Cesario, Axel Bergh, Alfredo Sanzo

November 8, 2012

1 Abstract

Almost every Object Oriented language define a different way to specify objects' behavior. Tough many of these variants seems to converge to widely spread concepts, like Simple Inherited Classes or Prototypes, new constructions keep arising in an attempt to solve their characteristic limitations. However, since most of these constructions are developed to be integrated in an already existent language, few of them present a radical solution and are more likely to just try to patch the existing tools embraced by the original model.

We may consider the implementation of Traits in the Pharo Smalltalk platform an example of this situation. Pharo extends the rigid Simple Inheritance Class based model of Smalltalk with constructions called "Traits", which can be used to implement methods in the same way Classes do, but can't define attributes nor be instantiated. Traits are used by Classes to acquire their method definitions and, since a Class can use many Traits, their present a good flexible way to avoid code repetition in those situations where single inheritance proves too rigid. Problem is the more code gets moved to Traits in order to make it reusable the less relevant the Class becomes; and since Traits can not be used without a Class, soon more and more Classes loose their original role to become Trait's atrophied limbs. Even if Classes still play an important role defining glue code to combine the Traits, in some point is inevitable to get the feeling that, if Traits where autonomous entities, there would be no need for Classes at all.

On the other hand, some languages do present more independent entities, like Scala's Mixins, but these constructions combination mechanisms tend to be less flexible.

This document presents the idea of Chunks, not as a complement to another existing mechanism, but as an alternative to the Class and Prototype paradigms. Even if the reference implementation provided is coded in Smalltalk, it should be easy to port to any other dynamically typed platform.

The main objectives of this tool are:

- Reduce the number of constructions required to have a fully flexible type definition.
- Provide a unified solution capable of both linearization and flattening like algebraic combinators.
- Establish a way to reify type combinations without involving other concepts.
- Replace the need of Glue Code with an extensible combinatory algebra.

2 ChunkDefinition

Chunks are entities that provide an interface to instantiate and define CObjects' behavior. Chunks are in many aspects similar to Classes, with the exception that they do not attend to any inheritance mechanism. Instead, Chunks can be combined and extended by applying a set of combinatory operations defined in their own basic interface.

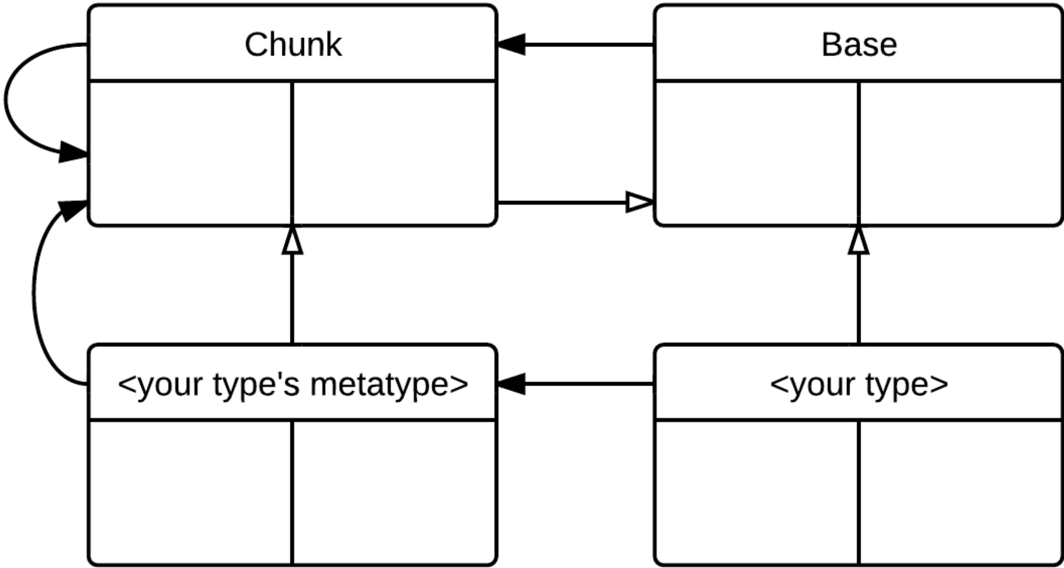
To maintain the abstraction level Chunks are also CObjects and are instance of their own Chunks. Thus any messages they respond to, including the combinatory algebra, can be extended without recurring to external tools or parsers.

Since Chunks can only be defined in terms of another Chunk, the Chunk **Base** is provided by the system as a prototype to any other. **Base** already provides the behavior every CObject must respond to. Also, **Base** responds to the basic behavior for a Chunk, such as CObjects instantiation and the combinatory algebra.

```
newInstance := Base new
```

Chunk provides the basic chunk-side methods (class methods).

Base provides the basic methods every CObject should understand.



The metatype defines your CObjects' chunk-side methods.

You define types parting from Base to provide methods to your CObjects

A ——► B : B is A's Chunk.

A ——▷ B : A is defined from B.

3 Combinatory Algebra Definition

Chunk's combinatory algebra allows to create new Chunks from existing ones. New Chunks implement their methods by applying a Flattening process, so the method lookup of a CObject is always resolved by it's Chunk.

The defined operations are as follow.

3.1 Method Addition

```
{Chunk} + #methodSelector = {method definition}
```

This operation creates a new Chunk that provides to it's instances the same methods the receiving does, plus the defined method. This operation has no side effect on the receiving Chunk.

Adding a method that is already defined in the receiving Chunk results in a Chunk that overrides the original implementation with the new one.

The following pieces of code produce the same result

```
Base
  + #m = {method definition 1}
  + #m = {method definition 2}
```

```
Base
  + #m = {method definition 2}
```

A method definition can be expressed in one of the following ways:

3.1.1 A Block defining method's body

```
Base + #eat: = [ :someFood | self energy: self energy + someFood providedEnergy. ^ self energy]
```

Resulting method will be:

```
eat: someFood
    self energy: self energy + someFood providedEnergy.
    ^ self energy
```

Notice that the block won't ever be evaluated. It's content will be copied to method body.

Block's argument count must be the same as the method's.

3.1.2 A reference to another Chunk's method definition

```
Target >> #targetMethodSelector
```

```
Squid + #swim: = Fish >> #swim:
```

Both target and definer selectors argument count must match, but there is no restriction about what Chunks can be targeted. This kind of indirect definition creates a link between the involved Chunks so that, whenever the targeted method is modified in the target Chunk, so does the defined method in the receiving Chunk.

3.1.3 A concatenation of other method definitions

```
{definition 1} , {definition 2} , . . . , {definition N}
```

```
Base + #tryToFly = [self validateFlying] , Bird >> #fly ,[^ true]
```

Concatenated definitions must require the same number of arguments the selector does or not require arguments at all. Resulting method will evaluate the concatenated implementations sequentially. A **.** will be inserted between any definition that is not finalized with a **.** already. Any ˆ operator starting a sentence will be ignored, except the ones located on the last definition concatenated.

The following pieces of code produce the same result

```
Base + #m:n: =
   [ :a :b| a ifTrue:[self w. ^10]. self x] ,
   [^ self y] ,
   [ :c :d| d value > 5 ifTrue:[^12]. ^13]
```

```
Base + #m:n: = [ :a :b |
    a ifTrue:[self w. ^ 10].
    self x.
    self y.
    b value > 5 ifTrue:[^12].
    ^13
]
```

3.1.4 A composition of other methods definitions

```
{definition 1} | {definition 2} | ... | {definition N}
```

```
Base + #finalPrice = Item >> #price | [ :price | ^ price + self taxes] | Promotion >> #applySale:
```

Composed definitions must require exactly one argument or not require arguments at all, except the first one, which must require the same number of arguments the selector does. Resulting method will evaluate the first method definition and pass the result as argument of the next one, returning the folded result on the last definition. Any ˆ operator within the composed definitions will be substituted by a call to the **return:** message of the **thisContext** metavariable, so the composition does not get interrupted.

The following pieces of code produce the same result

```
Base + #m:n: =
   [ :a :b| a ifTrue:[^self w]. self x] |
   [ :y| ^ self process: y] |
   [ :d| d work ifTrue:[^12]. ^13]
```

```
Base + #m:n: = [ :a :b || t1 t2 t3|
    t1:=[ :t4 :t5 |
        t4 ifTrue:[thisContext return: self w].
        self x
    ] value: a value: b.
    t2:=[ :t |
        thisContext return: (self process: t)
    ] value: t1.
    t3:=[ :t|
        t work ifTrue:[thisContext return: 12].
        thisContext return: 13
    ] value: t2.
   ^ t3 ]
```

3.1.5 Other Method Addition Characteristics

When adding a method with any of the above definitions, any message sent to the **self** metavariable will be silently added to the resulting Chunk as a **Required Method** (read ahead) if not already defined.

The following pieces of code produce the same result

```
Base
   + #fly: = [  :aDistance  |  self   energy: self
energy  -  ( self  energyRequiredToFly:  aDistance )]
```

```
Base
   ? #energy
   ? #energy:
   ? #energyRequiredToFly:
   + #fly: = [  :aDistance  |  self   energy: self
energy  -  ( self  energyRequiredToFly:  aDistance )]
```

3.2 Method Subtraction

```
{Chunk} – #methodSelector
```

```
Person - #run
```

This operation creates a new Chunk that provides to it's instances the same methods the receiving does, except for the one associated to the given selector. This operation has no side effect on the receiving Chunk. Removing a selector that the Chunk does not define provides a Chunk equal to the receiver.

3.3 Method Requirement

```
{Chunk} ? #methodSelector
```

```
Weapon ? #damage
```

This operation creates a new Chunk that provides to it's instances the same methods the receiving does, plus the defined method. This operation has no side effect on the receiving Chunk. If the Chunk does not already provide an implementation for the given selector, the method will raise an exception when invoked.

The following pieces of code produce the same result

```
A ? #m
```

```
A + #m = [ RequiredMethodException  raise ]
```

Otherwise, in case the receiving Chunk already provides an implementation for the given selector, this operation provides a Chunk equal to the receiver.

The following pieces of code produce the same result

```
A
   + #m = [^5]
   ? #m
```

```
A
   ? #m
   + #m = [^5]
```

```
A
   + #m = [^5]
```

3.4 FieldAddition

```
{Chunk} * #Name
```

```
Client
* #id
* #credit = 100
```

This operation creates a new Chunk that provides to it's instances the same methods the receiving does, plus the accessors for the defined Field. This operation has no side effect on the receiving Chunk. The name of a Field must be a unary symbol. Fields allow CObject composition and constitute the only way available to represent their inner state. When a Field is added, a getter named after it is defined along with a setter with the same name finalized by :. If the Chunk already provides implementations for the Field's getter or setter, the previous definitions are ignored.

The following pieces of code produce the same result

```
A
  + #m = [^5]
  + #m: = B >> #n:
  * #m
```

```
A * #m
```

Adding a method named as an already added Field ignores both the getter and setter implementations.

The following pieces of code produce the same result

```
A
  * #m
  * #n
  + #m = [^5]
  + #n: = B >> #o:
```

```
A
  + #m = [^5]
  + #n: = B >> #o:
```

Requiring a method named as any of the accessors of a Field previously defined has no effect.

The following pieces of code produce the same result

```
A
  * #m
  ? #m
```

```
A
  * #m
```

It is also possible to define an initial value for the Field. This value will be set automatically when the instance is created.

```
{Chunk} * #Name = {Initial Value}
```

If the initial value is not a niladic or monadic block, it will become the initial value of the Field.

```
Person * #account = 0
```

The instances created by following pieces of code are equal.

The following pieces of code produce the same result

```
joe := Person * #account new
joe account: 100
```

```
joe := Person * #account = 100 new
```

If the initial value is a niladic or monadic block, the initial value of the Field will be the result of evaluating that block. In case of being a monadic block, the argument it receives will be the newly created instance.

```
Person * #job = [ Unemployed new]
```

Notice that this is different than the following code:

```
Person * #job = Unemployed new
```

Instances of the Chunk generated by the second piece of code will be **identical**, while the instances of the first piece will be different CObjects.

If no initial value is specified for the Field, instances will default it's value to **nil**.

The following pieces of code produce the same result

```
Person * #job
```

```
Person * #job = nil
```

3.5 Chunk Merge

```
{Chunk 1} + {Chunk 2}
```

```
Attacker + Defender
```

This operation creates a new Chunk that provides to it's instances the same methods the receiving does, plus all the methods provided by the given argument Chunk. This operation has no side effect on any of the involved Chunks. This operation is proven to be associative and commutative. When both Chunks provides **different** implementations for the same message it produces a conflict.

If the conflict is between a required method and a non-required one, the conflict solves automatically by using the non-required implementation.

The following pieces of code produce the same result

```
A + #m = [^5]
  + (B ? #m)
```

```
A + B + #m = [^5]
```

If the conflict is between non-required methods the Chunk result of the merge operation will implement that method as an exception raise.

The following pieces of code produce the same result

```
A + #m = [^5]
  + (B + #m = [^7])
```

```
A + B + #m = [ UnsolvedConflictException raise]
```

Notice that, in order to properly solve a conflict, further operations are required such as adding a new definition for the conflicted method.

3.6 Chunk Overlap

```
{Chunk 1} +> {Chunk 2}
```

```
Human +> Warrior
```

This operation creates a new Chunk that provides to it's instances the same methods the receiving does, plus all the methods provided by the given argument Chunk. This operation has no side effect on any of the involved Chunks. When both Chunks provide **different** implementations for the same message it produces a conflict.

If the conflict is between a required method and a non-required one, the conflict solves automatically by using the non-required implementation.

The following pieces of code produce the same result

```
A + #m = [^5]
  +> (B? #m)
```

```
A +> B + #m = [^5]
```

```
A + B + #m = [^5]
```

If the conflict is between non-required methods the Chunk result of the overlap operation will automatically prefer the argument Chunk's implementation.

The following pieces of code produce the same result

```
A + #m = [^5]
  +> (B + #m = [^7])
```

```
A + B + #m = [^7]
```

Overriden method definitions may be invoked by sending the message to the result of the **super** message. This message gets automatically implemented during merge.

3.6.1 Reverse Chunk Overlap

As a syntactic sugar, a Reverse Chunk Overlap operation may be provided.

```
{Chunk 1} <+ {Chunk 2}
```

This operation should be essentially the same as Chunk Overlap, but switching the roles of the receiver and received Chunks

The following pieces of code produce the same result

```
A <+ B
```

```
B +> A
```

3.7 Chunk Registration

```
{Container Chunk} register: {Chunk} as: #Name
```

```
Units register: A as: #Warrior
```

This operation creates a new Chunk that provides to it's instances the same methods the argument Chunk does and generates the proper access methods in the container Chunk to access it. Chunk Registration is the only way provided by the algebra to generate a static access to a Chunk. The side effects on the container Chunk behaves in the following way:

If the provided name is not already in use by other method it becomes a reference to the generated Chunk.

If the provided name is already a method selector in either the Chunk or CObject side, it will be overridden. In the case where the overridden method was a previously registered Chunk, any reference to the old Chunk will be updated, including any methods defined as references to other methods.

Any Chunk registered in the Image will be statically accessible from everywhere.

Notice that, as the Chunks are registered "inside" other Chunks, the parent Chunk reference is required to gain static access to them. This implies that, in order to be accessed statically, the sequence of names of every ancestor is required in order to access a Chunk. This sequence can be think of as a **Full ChunkName**.

```
Imageregister: A as: #Tlon.

tlonChunk := Tlon.
alsoTlonChunk := Image Tlon.
tlonChunk == alsoTlonChunk.

Tlon register: B as: #Uqbar.
Tlon Uqbar register: C as: #Orbis.
Tlon Uqbar Orbis register: D as: #Tertius.

aTertius := Tlon Uqbar Orbis Tertius new.
```

Also notice that this name structure allows the definition of Chunks with the same name on different parents.

```
Tlon Uqbar register: Base as: #A.
Tlon Uqbar Orbis register: Base as: #A.
```

Once a Chunk is registered, a copy of it is stored in a Field on on the parent Chunk. Also, the registered Chunk generated copy will include Fields pointing to any other Chunks defined by the parent unless it already defines a method with the same name.

In order to avoid using full names on code, Chunk importation can be simulated by defining a method that returns the desired Chunk.

The following pieces of code produce the same result

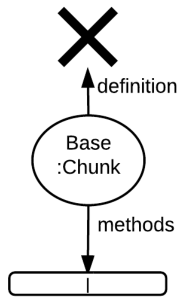
```
Base
  + #buildTertius = [^ Tlon Uqbar Orbis
Tertius new]
```

```
Base
  * #Tertius = Tlon Uqbar Orbis Tertius
  + #buildTertius = [^ Tertius new]
```

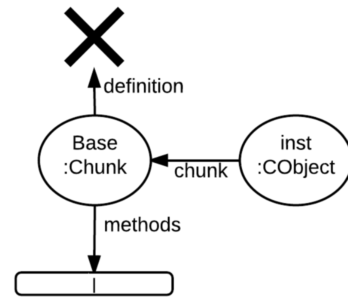
Notice that this imports will be treated as any other methods, so it can be manipulated with the combinatory algebra. Also, this pattern facilitates the use of dependency injection by providing a single access point to the Chunk. These import-like methods are particularly nice in languages where the message send is defaulted to the receiver and can be omitted.

The following graphic illustrates how registration phases look.

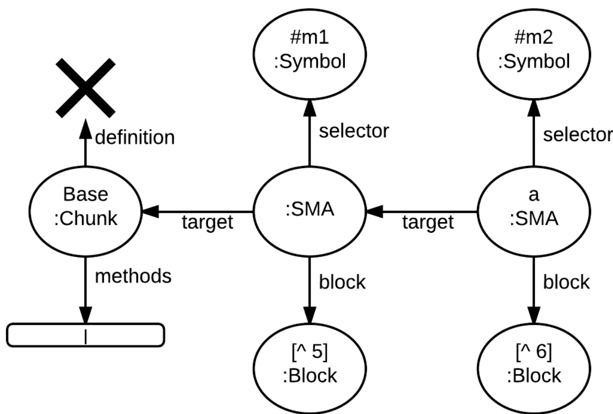
1) <<Initial State>>



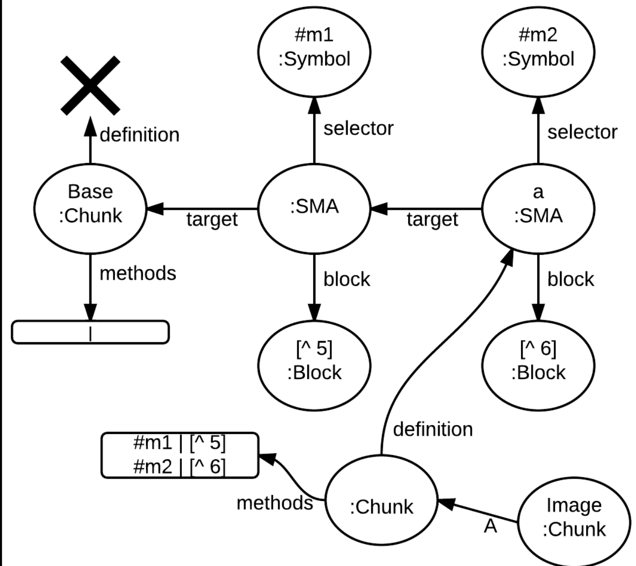
2) inst := Base new



3) a := Base
+ #m1 = [^ 5]
+ #m2 = [^ 6]



4) Image register: a As: #A



4 Chunk swapping

The basic interface of a CObject includes the accessors for it's Chunk. This makes possible to swap a CObject's Chunk dynamically. As the Chunk defines CObject's Fields, changing it in runtime may imply a change in it's inner state, in which case the following rules would apply:

Any Fielddefined in both the actual and target Chunk will keep it's actual value after the transition, regardless it's initialization value in any of the Chunks.

```
cobj := (A * #m) new.
cobj m: 5.
cobj chunk: (B * #m=7).
cobj m == 5.
```

Fields defined in the target Chunk only will be set to their initialization values.

```
cobj := A new.
cobj chunk: (B * #m = 5).
cobj m == 5.
```

Notice that, after a Chunk swap, references stored in Fields defined only in the original Chunkwill be lost.

5 Comparison with Traits

In this section we present an example of how Chunks can be used to model a complex domain, along with an equivalent implementation using Traits and a comparative analysis. We also present a possible UML extension to document Chunks and some design tips.

5.1 Domain description

Let's consider a system that allows to take bets for different types of raffles. The earnings for the winning gambles is determined in two basic ways: there can be a pot of cash based on the money collected from all bets or have a pre-established prize for the winners. The games to develop initially will be Prode, Quiniela, Loto and Always Pays. It should be easy to add new games with this characteristics and require the least programming effort as possible.

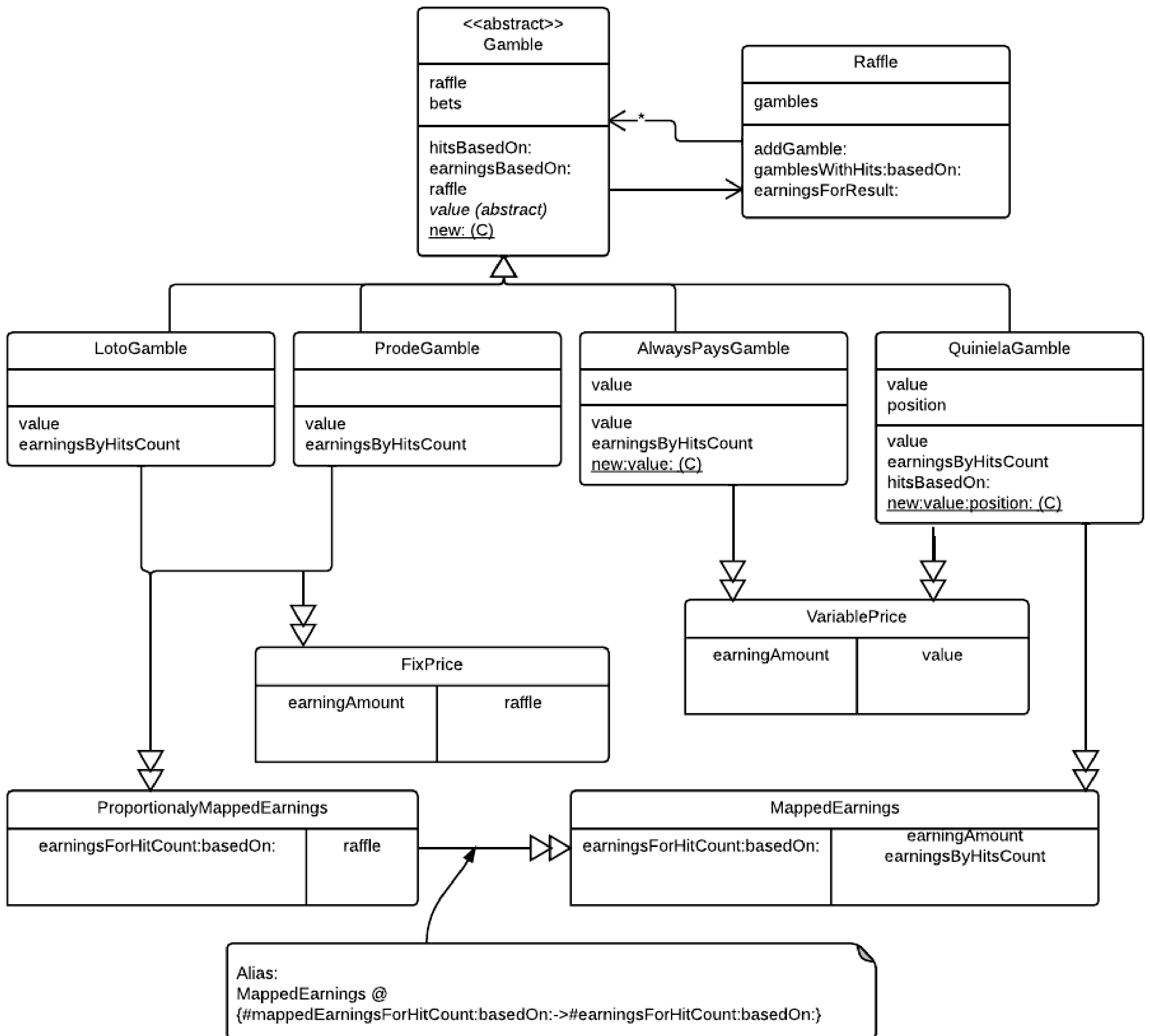
Prode This game consists on choosing a result for each one of the 13 football matches of the week, being the possible options: local team wins, visiting team wins or tied game. A gamble for this game has a fix value of $5 and it's based on a cash pot. The payment method for this game depends on the hits count in a way that the first prize will be for those who guess the 13 games correctly, sharing 50% of the pot. The second prize will be for those who get 12 hits, sharing 35% of the pot. Finally, the third prize is for those who got 11 hits, sharing 11% of the pot unless it's lower than the gambled money, in that case they would recover the spent money.

Quiniela This game, unlike Prode, has a pre-established payment and it's based on the selection of 20 random four digit numbers. The gamblers bet an amount of money of their choice on a single number, that could be from 1 to 4 digits, and the position from 1 to 20 in which that number should be sorted in. The earnings for the player depend on the amount of digits from the bet: 1 digit matches get 7 times the value gambled. 2 digit matches get 70 times the value gambles. 3 digit matches get 500 times the value gambled. 4 digit matches get 3500 times the value gambles.

Loto This game consists on the random selection of 6 numbers between 00 and 38. The prize for each gamble is $2. A gamble is considered winner if there are 4 to 6 hits in any order. The payment is based on a pot that will be split between the winners in the following manner: 6 hits: 40% of the pot 5 hits: 17% of the pot 4 hits: 5% of the pot

Always Pays This game is based on choosing 15 numbers between 00 y 99. The gamble value is determined by the player and all raffles give an earning to all gamblers. The earnings for a gamble is calculated with the following formula: $(hitscount/10)^3 * gambledvalue$

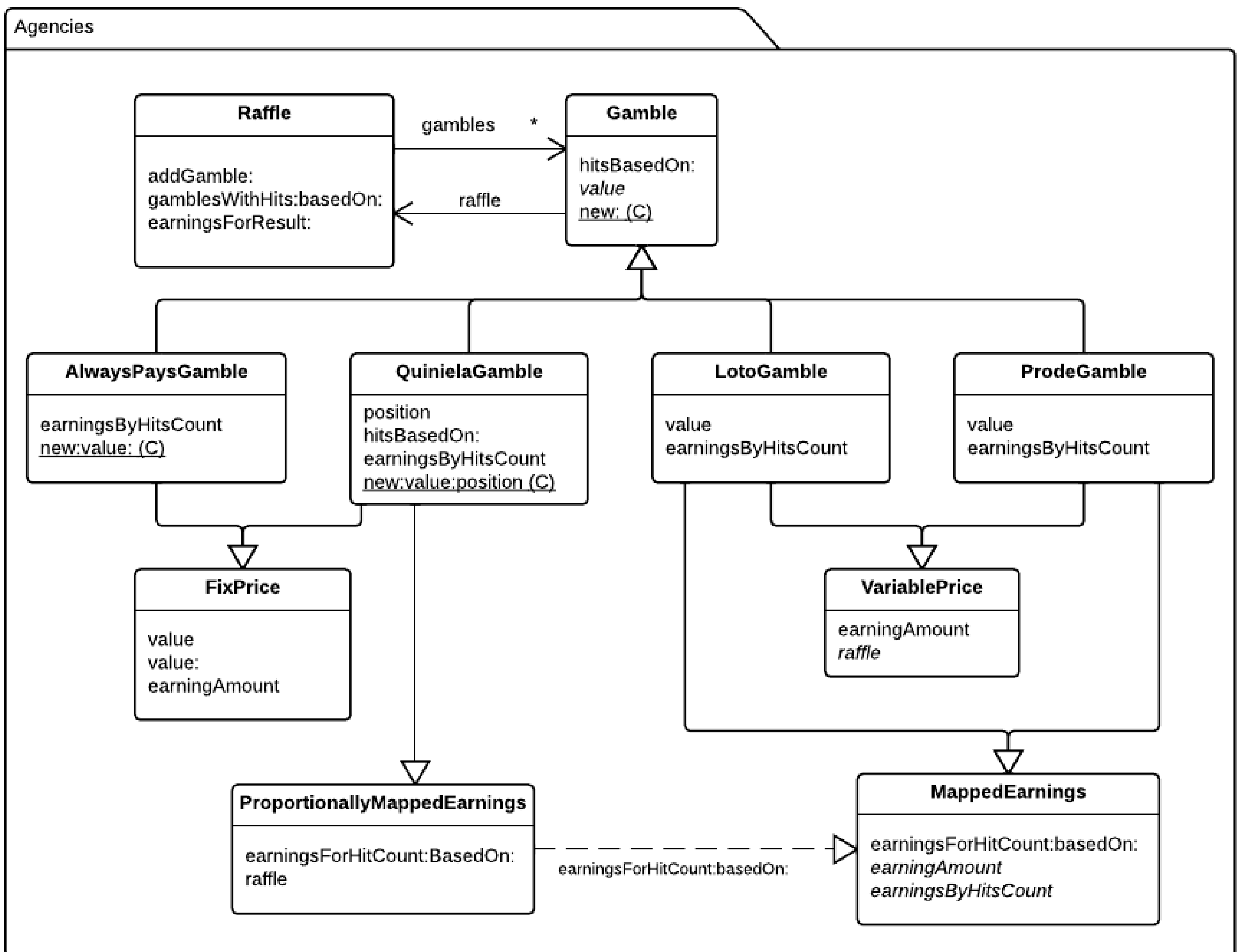
The following is the class diagram of a possible resolution made using both inheritance mechanisms and Traits usage for code sharing. The source code in which is based can be found on the appendix at the end of the document.



In this implementation we can observe that it is possible to prevent repetition between the different types of gambles by using the FixPrice, VariablePrice, ProportionallyMappedEarnings and MappedEarnings Traits, along with the Gamble superclass that includes all of the generic behavior between the different gambles. Also, one of the Traits uses another one in its definition to calculate the earnings for a given hits count.

The approach used to design the solution using Traits is not standard since these classes don't include only glue code, there is more complex logic that is not present in any Trait. If all the logic was extracted as new Traits, the amount of elements needed to model the problem would grow, making it more complex.

The next diagram shows a solution to the same problem using Chunks.



A → B: An instance of A references an instance of B

A - - -▷ B: A includes methods from B

A ——▷ B: A extends B

It's possible to see that the same abstractions that existed in the prior solution are also found in this alternative. The proposed solution is able to address the different needs that arose using the Traits approach without having logic repetition. Further more, since Pharo has an implementation of stateless Traits, the 'value' variable and its accessors had to be defined both in AlwaysPaysGamble and QuinielaGamble even thought it was related to the VariablePrice abstraction, which can be solved without problems in Chunks using the 'value' field.

In the case of the Quiniela game, the previous version redefines the method **#hitsBasedOn:** and reuses the inherited method by sending that message to super. In the Chunks version it's possible to add a method in QuinielaGamble that internally invokes the definition found in Gamble, so it's possible to reuse code in a similar way that is offered by inheritance.

Regarding the ProportionallyMappedEarnings, its definition includes only the **#earningsForHitCount:basedOn:** method from the MappedEarnings Chunk, which is why the arrow has a dashed line to denote the low coupling between both abstractions.

Finally, all of these Chunks are registered in the Agencies Chunk, so they can use each other as easily as having static access with the advantage of being able to define other Raffle Chunk for example outside of Agencies scope without any inconveniences. This currently can't be done with Classes and Traits in Pharo, since the language doesn't support namespaces.

A proposed graphic notation for Chunks diagram includes a vertical layout for more compact diagrams, italics for required methods and underlined for Chunk side methods, open arrow tips for relationships on an instance level and closed arrow tips for relationships on Chunk level. In the case of Chunk level relationships, a solid line is used for extention that can be annotated for exclusion (if nothing is annotated the full behavior is included in the extendee) and a dashed line

means only a subset of methods are explicitly included in the definition, which should always be annotated.

6 Appendix: Example Source Code

6.1 Traits Code

```
Gamble>> value
    self subclassResponsibility

Gamble>> bets
    ^ bets

Gamble>> hitsBasedOn: aResult
    | hits |
    hits := OrderedCollection new.
    self bets reverse withIndexDo: [ :bet :i |
        bet = (aResult reverse at: i) ifTrue:[ hits add: bet]
    ].
    ^ hits size

Gamble>> bets: colOfBets
    bets := colOfBets

Gamble>> earningsBasedOn: aResult
    ^ self earningsForHitCount: (self hitsBasedOn: aResult) basedOn: aResult

Gamble>> raffle: aRaffle
    raffle := aRaffle

Gamble>> raffle
    ^ raffle

Gamble class>> new: aRaffle
    ^ self new raffle: aRaffle.

MappedEarnings>> earningsForHitCount: hitCount basedOn: aResult
    ^ (self earningsByHitsCount at: hitCount ifAbsent:[0])
            * self earningAmount

MappedEarnings>> earningsByHitsCount
    ^ self requirement

MappedEarnings>> earningAmount
     ^ self requirement

ProportionallyMappedEarnings>> earningsForHitCount: hitCount basedOn: aResult
    ^ (self mappedEarningsForHitCount: hitCount basedOn: aResult)
    / (self raffle gamblesWithHits: hitCount basedOn: aResult)

ProportionallyMappedEarnings>> raffle
     ^ self requirement

Raffle>> addGamble: aGamble
    gambles add: (aGamble raffle: self)

Raffle>> gambles: colOfGambles
    gambles := colOfGambles

Raffle>> earningsForResult: aResult
    ^ self gambles inject: Dictionary new into:[ :a :gamble |
        a at: gamble put: (gamble earningsBasedOn: aResult in: self)
    ]
```

```
Raffle>> initialize
    gambles := OrderedCollection new.

Raffle>> gambles
    ^ gambles

Raffle>> gamblesWithHits: hitCount basedOn: aResult
    ^ (self gambles select:[ :gamble |( gamble hitsBasedOn: aResult) = hitCount]) size

FixedPrice>> earningAmount
    ^ self raffle gambles sum: #value

FixedPrice>> raffle
     ^ self requirement

VariablePrice>> earningAmount
    ^ self value

VariablePrice>> value
     ^ self requirement

LotoGamble>> value
    ^ 2

LotoGamble>> earningsByHitsCount
    ^ Dictionary keys: #(6 5 4) values: #(0.4 0.17 0.05)

QuinielaGamble>> hitsBasedOn: aResult
    | hits |
    hits := super hitsBasedOn: (aResult at: self position).
    ^ hits = self bets size ifTrue:[ hits] ifFalse:[0]

QuinielaGamble>> earningsByHitsCount
    ^ Dictionary keys: #(1 2 3 4) values: #(7 70 500 3500)

QuinielaGamble>> value: aValue
    value := aValue

QuinielaGamble>> value
    ^ value

QuinielaGamble>> position
    ^ position

QuinielaGamble>> position: aPosition
    position := aPosition

QuinielaGamble class>>new: aRaffle value: aValue position: aPosition
    ^ (self new: aRaffle) value: aValue; position: aPosition.

ProdeGamble>> value
    ^ 5

ProdeGamble>> earningsByHitsCount
    ^ Dictionary keys: #(13 12 11) values: #(0.5 0.35 0.11)

AlwaysPaysGamble>> value
    ^ value

AlwaysPaysGamble>> value: aValue
    value := aValue
```

```
AlwaysPaysGamble>> earningsForHitCount:hitCount basedOn:aResult
    ^ (hitCount / 10 raisedTo: 3) * self earningAmount

AlwaysPaysGamble class>> new: aRaffle value: aValue
    ^ (self new: aRaffle) value: aValue.
```

6.2 Chunks Code

```
Image register: Image Base as: #Agencies.

Image Agencies register:
    Image Base
        * #gambles = [ OrderedCollection new]
        + #addGamble: = [ :aGamble | self gambles add: (aGamble raffle: self)]
        + #gamblesWithHits:basedOn: = [ :hitCount :aResult|
            ^ (self gambles select:[ :gamble | (gamble hitsBasedOn: aResult) = hitCount]) size
        ]
        + #earningsForResult: = [ :aResult |
            ^ self gambles inject: Dictionary new into:[ :a :gamble |
                a at: gamble put: (gamble earningsBasedOn: aResult in: self)
            ]
        ]
as: #Raffle.


Image Agencies register:
    Image Base
        * #raffle
        * #bets
        + #hitsBasedOn: = [ :aResult || hits|
            hits := OrderedCollection new.
            self bets reverse withIndexDo: [ :bet :i |
                bet = (aResult reverse at: i) ifTrue:[ hits add: bet]
            ].
            ^ hits size
        ]
        + #earningsBasedOn: = [ :aResult || hits|
            ^ self earningsForHitCount: (self hitsBasedOn: aResult) basedOn: aResult
        ]
        ? #value
as: #Gamble.

Image Agencies Gamble chunk: (Image Agencies Gamble chunk
    + #new: = [ :bets | ^self new bets: bets]
) buildChunk.


Image Agencies register:
    Image Base
        + #earningAmount = [^ self raffle gambles sum: #value]
as: #VariablePrice.

Image Agencies register:
    Image Base
        + #earningAmount = [^ self value]
        * #value
as: #FixedPrice.


Image Agencies register:
    Image Base
```

```
        ? #earningsByHitsCount
        ? #earningAmount
        + #earningsForHitCount:basedOn: = [ :hitCount :aResult |
            ^ (self earningsByHitsCount at: hitCount ifAbsent:[0])
            * self earningAmount
        ]
as: #MappedEarnings.


Image Agencies register:
    Image Agencies MappedEarnings
        ? #raffle
        + #earningsForHitCount:basedOn: = [ :hitCount :aResult |
            ^ (self
                eval: Image Agencies MappedEarnings >> #earningsForHitCount:basedOn:
                with: {hitCount. aResult}
            )
            / (self raffle gamblesWithHits: hitCount basedOn: aResult)
        ]
as: #ProportionalMappedEarnings.


Image Agencies register:
    Image Agencies Gamble +
    Image Agencies VariablePrice +
    Image Agencies ProportionalMappedEarnings
        + #value = [^ 2]
        + #earningsByHitsCount = [^ Dictionary keys: #(6 5 4) values: #(0.4 0.17 0.05)]
as: #LotoGamble.


Image Agencies register:
    Image Agencies Gamble +
    Image Agencies VariablePrice +
    Image Agencies ProportionalMappedEarnings
        + #value = [^ 5]
        + #earningsByHitsCount = [^ Dictionary keys: #(13 12 11) values:#(0.5 0.35 0.11)]
as: #ProdeGamble.


Image Agencies register:
    Image Agencies Gamble +
    Image Agencies FixedPrice +
    Image Agencies MappedEarnings
        * #position
        + #earningsByHitsCount = [^ Dictionary keys: #(1 2 3 4) values: #(7 70 500 3500)]
        + #hitsBasedOn: = [ :aResult || hits|
            hits := self
               eval: Image Agencies Gamble >> #hitsBasedOn:
               with: {aResult at: self position}.
            ^ hits = self bets size ifTrue:[hits] ifFalse:[0]
        ]
as: #QuinielaGamble.

Image Agencies QuinielaGamble chunk: (Image Agencies QuinielaGamble chunk
    + #new:position:value: = [ :bets :position :value |
        ^(self new: bets) position: position; value: value
    ]
) buildChunk.


Image Agencies register:
    Image Agencies Gamble + Image Agencies FixedPrice
```

```
        + #earningsForHitCount:basedOn: = [ :hitCount :aResult |
            ^ ( hitCount / 10 raisedTo: 3)
            * self earningAmount
        ]
as: #AlwaysPaysGamble.

Image Agencies AlwaysPaysGamble chunk: ( Image Agencies AlwaysPaysGamble chunk
    + #new:value: = [ :bets :value | ^(self new: bets) value: value]
) buildChunk.
```

7 References

- Schärli, Ducasse, Nierstrasz, Black. 2003. **Traits: Composable Units of Behaviour**
- Ducasse, Nierstrasz, Schärli, Dcs. 2006. **Traits, A Mechanism for Fine-grained Reuse**
- Bergel, Ducasse, Nierstrasz, Dcs, Wuyts. 2006. **Stateful Traits & their Formalization**
- Bracha, Cook. 1990. **Mixin-Based Inheritance**
- Lieberman. 1986. **Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems**
- Haahr. 1996. **A Monotonic Superclass Linearization for Dylan**
- Bracha. 2007. **Article: Lethal Injection** - http://gbracha.blogspot.com.ar/2007/12/some-months-ago-i-wrote-couple-of-posts.html