Implémentation du noyau réflexif ObjVlisp

Projet HAI931I - Systèmes Réflexifs & Models@Runtime

Auteur

Taha Benslimane

Encadrant

Christophe Dony

Année Universitaire 2025-2026

Faculté des Sciences
Université de Montpellier

Table des matières

Chapitre 1

Introduction

Ce rapport présente l'implémentation complète du micro-noyau réflexif ObjVlisp en Pharo Smalltalk. Le projet couvre la construction d'un système orienté objet minimal composé de deux classes fondamentales (`ObjObject` et `ObjClass`), l'implémentation des mécanismes de base (allocation, initialisation, envoi de messages, lookup), et l'extension du noyau avec des fonctionnalités avancées telles que des métaclasses personnalisées (`ObjAbstractClass` pour l'instanciation contrôlée, `ObjClassWithAccessors` pour la génération automatique de getters/setters), les variables de classe (Shared Variables), et un mécanisme d'appel super sécurisé basé sur l'introspection de la pile d'exécution. Chaque section présente le code implémenté accompagné d'une analyse détaillée démontrant la compréhension des concepts sous-jacents.

1.1 Contexte du projet

ObjVlisp est un modèle de langage orienté objet minimal et réflexif, conçu par Pierre Cointe et inspiré du noyau de Smalltalk-78. Ce modèle démontre qu'un système orienté objet complet peut être construit avec seulement deux classes fondamentales : `Object` (la racine de la hiérarchie d'héritage) et `Class` (la classe des classes).

1.2 Objectifs pédagogiques

L'implémentation de ce noyau permet de comprendre en profondeur :
— La structure interne des objets et des classes
— Les mécanismes d'allocation et d'initialisation
— Le lookup de méthodes et la résolution dynamique
— La réflexivité et les métaclasses
— Le bootstrap d'un système orienté objet

1.3 Choix d'implémentation

Nous avons choisi d'implémenter ObjVlisp en Pharo Smalltalk, en représentant les objets comme des tableaux (instances de la classe `Obj`, sous-classe de `Array`). Ce choix permet de se concentrer sur l'essence du modèle sans implémenter un compilateur complet.

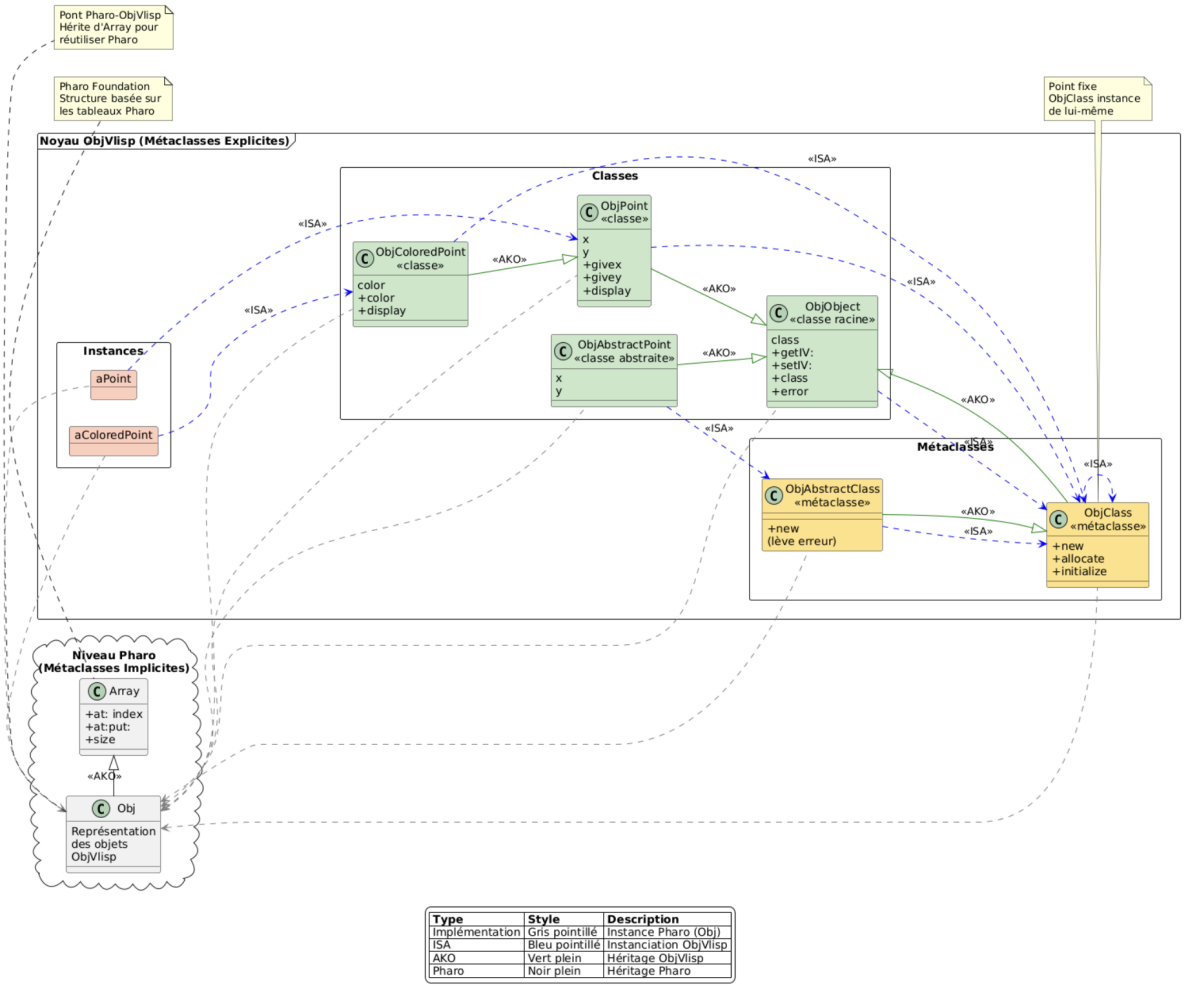

FIGURE 1.1 – Architecture ObjVLisp : Métaclasses explicites implanté en Pharo

Chapitre 2

Implémentation

2.1 Structure des objets et primitives de base

2.1.1 Représentation des objets

Choix de représentation

Dans notre implémentation, chaque objet ObjVlisp est représenté par un tableau (instance de `Obj`). Cette approche permet :

— Une manipulation simple des structures de données
— Un accès direct par index pour les variables d'instance
— Une réutilisation des fonctionnalités natives de Pharo

Structure d'une instance

Une instance ObjVlisp est représentée par un tableau de la forme :

```
#(ClassId valueIV1 valueIV2 ... valueIVn)
```

où :

— `ClassId` : Symbole identifiant la classe de l'instance (permet le lookup de méthodes)
— `valueIV1, valueIV2, ...` : Valeurs des variables d'instance dans l'ordre de déclaration

Exemple ObjPoint :

La classe `ObjPoint` déclare deux variables d'instance : `x` et `y`. Une instance représentant le point (10, 15) a donc la structure suivante :

Listing 2.1 – Structure d'une instance ObjPoint

```
1 #(#ObjPoint 10 15)
```

Structure d'une classe

Une classe ObjVlisp est elle-même un objet représenté par un tableau de la forme :

```
#(metaclass name superclass ivs keywords methodDict)
```

où :

— `metaclass` : Symbole identifiant la métaclasse (la classe de cette classe, ex : `#ObjClass`)
— `name` : Nom symbolique de la classe (ex : `#ObjPoint`)
— `superclass` : Symbole identifiant la classe parente dans la hiérarchie d'héritage (ex : `#ObjObject`)
— `ivs` : Collection ordonnée des variables d'instance complètes (ex : `#(class x y)`)
— `keywords` : Collection des mots-clés d'initialisation correspondant aux variables d'instance (ex : `#(x: y:)`)
— `methodDict` : Dictionnaire associant les sélecteurs de méthodes à leur implémentation (contient uniquement les méthodes définies localement)

Exemple ObjPoint :

La classe `ObjPoint` représentant des points 2D a la structure suivante :

Listing 2.2 – Structure de la classe ObjPoint

```
#(#ObjClass
  #ObjPoint
  #ObjObject
  #(class x y)
  #(x: y:)
  <IdentityDictionary: #givex #givey #x: #y: #display>)
```

2.1.2 Primitives d'accès

Calcul des offsets

Les offsets sont des constantes définissant la position de chaque métadonnée dans la structure d'une classe. Ces méthodes encapsulent l'accès direct par index, permettant une modification future de la structure sans casser le code existant.

Listing 2.3 – Primitives de structure de classe

```
Obj >> offsetForClass
    "Position de l'identifiant de métaclasse"
    ^ 1

Obj >> offsetForName
    "Position du nom de la classe"
    ^ 2

Obj >> offsetForSuperclass
    "Position de l'identifiant de superclasse"
    ^ 3

Obj >> offsetForIVs
    "Position de la liste des variables d'instance"
    ^ 4

Obj >> offsetForKeywords
    "Position de la liste des mots-clés d'initialisation"
    ^ 5

```

```
Obj >> offsetForMethodDict
    "Position du dictionnaire de méthodes"
    ^ 6
```

Accès à la structure d'objet

Chaque instance stocke l'identifiant de sa classe en première position. Ces primitives permettent d'y accéder afin qu'un objet puisse interroger sa classe pour résoudre les méthodes ou accéder à ses métadonnées.

Listing 2.4 – Primitive de structure d'objet

```
Obj >> objClass
    "Retourne la classe de l'objet"
    ^ Obj giveClassNamed: (self objClassId)

Obj >> objClassId
    "Retourne l'identifiant de classe de l'objet"
    ^ self at: self offsetForClass

Obj >> objClassId: anObjClassId
    "Définit l'identifiant de classe"
    self at: self offsetForClass put: anObjClassId
```

Accès à la structure de classe

Une classe, étant elle-même un objet, possède une structure riche contenant ses métadonnées : nom, superclasse, variables d'instance, mots-clés, et dictionnaire de méthodes. Les primitives suivantes encapsulent l'accès à chacun de ces éléments.

Listing 2.5 – Primitives de structure de classe

```
Obj >> objName
    "Retourne le nom de la classe (symbole)"
    ^ self at: self offsetForName

Obj >> objName: aName
    "Définit le nom de la classe"
    self at: self offsetForName put: aName

Obj >> objSuperclassId
    "Retourne l'identifiant de la superclasse"
    ^ self at: self offsetForSuperClass

Obj >> objSuperclassId: aSymbol
    "Définit la superclasse"
    self at: self offsetForSuperClass put: aSymbol

Obj >> objIVs
    "Retourne la liste des variables d'instance"
    ^ self at: self offsetForIVs

Obj >> objIVs: anOrderedCollection
    "Définit les variables d'instance"
```

```
    self at: self offsetForIVs put: anOrderedCollection

Obj >> objKeywords
    "Retourne les mots-clés d'initialisation"
    ^ self at: self offsetForKeywords

Obj >> objKeywords: anOrderedCollection
    "Définit les mots-clés"
    self at: self offsetForKeywords put: anOrderedCollection

Obj >> objMethodDict
    "Retourne le dictionnaire de méthodes"
    ^ self at: self offsetForMethodDict

Obj >> objMethodDict: aDictionary
    "Définit le dictionnaire de méthodes"
    self at: self offsetForMethodDict put: aDictionary
```

2.2 Accès aux variables d'instance

2.2.1 Calcul des offsets

Offset depuis la classe

Cette méthode cherche l'index d'une variable d'instance dans la liste complète (incluant les variables héritées). Le retour de 0 en cas d'absence permet de détecter les erreurs.

Listing 2.6 – Offset depuis la classe

```
Obj >> offsetFromClassOfInstanceVariable: aSymbol
    "Retourne l'offset de la variable d'instance dans la classe"
    | ivs |
    ivs := self objIVs.
    ^ ivs indexOf: aSymbol ifAbsent: [ 0 ]
```

Offset depuis l'instance

La structure étant définie au niveau classe, cette méthode délègue le calcul à la classe de l'instance. Elle ajoute une validation pour produire un message d'erreur explicite.

Listing 2.7 – Offset depuis une instance

```
Obj >> offsetFromObjectOfInstanceVariable: aSymbol
    "Retourne l'offset de la variable d'instance pour cette instance"
    | index |
    index := self objClass offsetFromClassOfInstanceVariable: aSymbol.
    (index = 0)
        ifTrue: [self error: 'Unexistant instance variable ', aSymbol,
                            ' for ', self asString]
        ifFalse: [^ index]
```

2.2.2 Accès aux valeurs

L'offset calculé permet l'accès direct à la valeur. Cette primitive offre une interface simple : fournir le nom de la variable, obtenir sa valeur, sans se soucier des détails d'implémentation.

Listing 2.8 – Lecture de variable d'instance

```
Obj >> valueOfInstanceVariable: aSymbol
    "Retourne la valeur de la variable d'instance"
    ^ self at: (self offsetFromObjectOfInstanceVariable: aSymbol)
```

2.3 Allocation et initialisation

La création d'un objet en ObjVlisp, comme dans tout système orienté objet, se décompose en deux phases distinctes : l'**allocation** (réservation de mémoire et structure de base) et l'**initialisation** (remplissage avec les valeurs spécifiques). Cette séparation permet une grande flexibilité : on peut allouer sans initialiser, initialiser différemment selon le contexte, ou même réinitialiser un objet existant.

2.3.1 Allocation d'instances

L'allocation est une responsabilité de la classe : elle connaît la structure de ses instances (nombre et ordre des variables d'instance) et peut donc créer un tableau de la bonne taille.

Primitive d'allocation

L'allocation crée un tableau de la taille exacte nécessaire (nombre de variables d'instance). Le premier slot est immédiatement rempli avec l'identifiant de classe, garantissant que tout objet "connaît" sa classe dès sa création. Les autres slots restent à `nil`, permettant une détection facile des variables non initialisées.

Listing 2.9 – Allocation d'une nouvelle instance

```
Obj >> allocateAnInstance
    "Alloue une nouvelle instance de la classe réceptrice"
    | size instance |
    size := self numberOfIVs.
    instance := Obj new: size.
    instance at: self offsetForClass put: self objName.
    2 to: size do: [:i | instance at: i put: nil].
    ^ instance
```

2.3.2 Initialisation d'instances

L'initialisation remplit les variables d'instance avec des valeurs fournies par l'utilisateur. ObjVlisp adopte le style Lisp avec des mots-clés (symboles terminés par `:`), permettant une initialisation déclarative et flexible : l'ordre des arguments n'a pas d'importance, et les valeurs omises restent à `nil`.

Support des mots-clés à la Lisp

Les mots-clés permettent d'associer explicitement chaque valeur à sa variable d'instance. Par exemple, `#(x: 10 y: 20)` indique clairement que 10 est la valeur de `x` et 20 celle de `y`. Les primitives suivantes extraient ces valeurs selon un schéma défini par la classe.

Listing 2.10 – Extraction de valeur par mot-clé

```
Obj >> keywordValue: aKey getFrom: anArray ifAbsent: aDefaultValue
    "Extrait la valeur associée au mot-clé dans le tableau"
    | i |
    i := anArray indexOf: aKey ifAbsent: nil.
    ^ i isNil
        ifTrue: [aDefaultValue]
        ifFalse: [anArray at: i + 1]
```

Listing 2.11 – Extraction selon un schéma

```
Obj >> returnValuesFrom: anInitargArray followingSchema:
   anArrayOfKeywords
    "Extrait les valeurs selon le schéma de mots-clés de la classe"
    "Exemple: self new returnValuesFrom: #(lulu 22 titi 35)
                       followingSchema: #(titi toto lulu titi)
              --> #(35 nil 22 35)"
    ^ anArrayOfKeywords collect:
        [:e |
            self keywordValue: e
                getFrom: anInitargArray
                ifAbsent: nil]
```

Primitive d'initialisation

Une fois les valeurs extraites selon le schéma, l'initialisation les place aux positions correctes dans l'instance. Cette méthode utilise le schéma défini par `objKeywords` pour garantir que chaque valeur termine à la bonne position, indépendamment de l'ordre fourni par l'utilisateur.

Listing 2.12 – Initialisation avec schéma

```
Obj >> initializeUsing: anAlternatedArray
    "Initialise l'objet selon le tableau de mots-clés/valeurs"
    | ivValues |
    ivValues := self
        returnValuesFrom: anAlternatedArray
        followingSchema: self objClass objKeywords.
    1 to: ivValues size do: [:i |
        | value |
        value := ivValues at: i.
        value ifNotNil: [self at: i + 1 put: value]
    ].
    ^ self
```

Observation : L'offset est `i + 1` car la position 1 contient l'identifiant de classe, et les variables d'instance commencent à la position 2.

2.4 Héritage statique des variables d'instance

L'héritage des variables d'instance en ObjVlisp est **statique** : au moment de la création d'une classe, les variables héritées sont fusionnées avec les variables locales pour former la liste complète des variables d'instance.

Principe de la fusion : Si `ObjObject` déclare `#(class)` et `ObjPoint` déclare localement `#(x y)`, alors la liste complète d'`ObjPoint` sera `#(class x y)`. L'ordre des variables héritées est préservé, les nouvelles variables sont ajoutées à la fin, et les duplications sont interdites.

Importance de l'ordre : Maintenir l'ordre des variables de la superclasse est crucial. Si `class` est à la position 1 dans `ObjObject`, elle doit rester à la position 1 dans toutes les sous-classes. Cela garantit que les offsets restent valides dans toute la hiérarchie : du code écrit pour `ObjObject` fonctionne sur n'importe quelle instance, même d'une sous-classe profonde.

Fusion des variables d'instance

La primitive `computeNewIVFrom:with:` réalise la fusion ordonnée : elle prend les variables héritées, ajoute les variables locales non-dupliquées, et retourne la liste complète dans le bon ordre.

Listing 2.13 – Calcul de l'héritage des variables

```
1 Obj >> computeNewIVFrom: superIVs with: localIVs
2     "Calcule les variables d'instance héritées et locales"
3     | ivs |
4   ^superIVOrdCol isNil
5     ifTrue: [localIVOrdCol]
6     ifFalse:
7       [ivs := superIVOrdCol asOrderedCollection copy.
8       localIVOrdCol do: [:e | (ivs includes: e) ifFalse: [ivs add: e]].
9       ivs]
```

2.5 Gestion des méthodes

2.5.1 Représentation des méthodes

Méthodes comme blocks

Dans notre implémentation, les méthodes sont représentées par des blocks Pharo encapsulés. Un block est une fermeture lexicale (closure) : une fonction anonyme capturant son environnement. Chaque méthode ObjVlisp est stockée dans le dictionnaire de méthodes de sa classe, associée à son sélecteur (nom symbolique comme `#givex` ou `#display`).

Listing 2.14 – Récupération du corps d'une méthode

```
Obj >> bodyOfMethod: aSelector
    "Retourne la méthode (block) associée au sélecteur <aSelector>.
    Retourne nil si le sélecteur n'est pas défini.
    Le récepteur est une objClass."
    ^ self objMethodDict at: aSelector ifAbsent: [nil]
```

Ajout de méthode

L'ajout d'une méthode transforme du code source (chaîne de caractères) en block exécutable via le compilateur Pharo. Le processus utilise la **currification** pour lier la superclasse au moment de la définition : le block externe capture `superClassOfClassDefiningTheMethod`, créant une fermeture que le block interne (la méthode réelle) peut utiliser pour les appels super pour éviter de passer explicitement la superclasse à chaque invocation.

Listing 2.15 – Ajout d'une méthode dans le dictionnaire

```
Obj >> addMethod: aSelector args: aString withBody: aStringBlock
    "Définit une méthode avec le sélecteur et le corps donnés"
    self objMethodDict
        at: aSelector
        put: (self methodWithHeader: aString andBody: aStringBlock).
    ^ self
```

Listing 2.16 – Compilation et currification

```
Obj >> methodWithHeader: col andBody: aString
    "Compile le code source en block et lie la superclasse (
       currification)"
    | block string resBlock |
    string := self stringOfBlockBodyFromHeader: col andBody: aString.

    block := OpalCompiler new
        source: string;
        evaluate.

    resBlock := block value:
        (Obj giveClassNamed: self objSuperclassId ifAbsent: [666]).

    ^ resBlock
```

Structure à deux niveaux : Le code compilé est `[:superClass | [:objself :args | corps]]`. Le block externe est évalué immédiatement avec la superclasse, produisant le block interne qui sera stocké et exécuté à chaque appel.

Listing 2.17 – Construction de la chaîne de code source

```
Obj >> stringOfBlockBodyFromHeader: col andBody: aString
    "Génère la chaîne de code source pour le block à deux niveaux"
    ^ '[:superClassOfClassDefiningTheMethod |
        [:objself ', (self blockArgsFrom: col), ' |
            ', aString, '
        ]
    ]'
```

2.5.2 Lookup de méthodes

Le lookup implémente la recherche de méthode dans la hiérarchie d'héritage : chercher d'abord dans la classe, puis remonter récursivement vers les superclasses jusqu'à trouver la méthode ou atteindre la racine (`ObjObject`). Ce mécanisme réalise l'héritage comportemental et le polymorphisme.

Listing 2.18 – Recherche de méthode dans la hiérarchie

```
Obj >> lookup: aSelector
    "Cherche la méthode dans la classe et ses superclasses"
    | method |

    method := self bodyOfMethod: aSelector.

    ^ method
        ifNotNil: ["méthode trouvée" method]
        ifNil: [
            (self objName == #ObjObject)
                ifTrue: ["racine atteinte, méthode introuvable" nil]
                ifFalse: [self objSuperclass lookup: aSelector]
        ]
```

2.6 Envoi de messages

2.6.1 Envoi simple

En ObjVlisp, `send:withArguments:` combine lookup et exécution. Le lookup démarre depuis la classe du récepteur et remonte la hiérarchie d'héritage. Si la méthode est trouvée, elle est exécutée. Sinon un message d'erreur est envoyé.

Listing 2.19 – Primitive d'envoi de message

```
Obj >> send: selector withArguments: arguments
    "Envoie un message à l'objet récepteur"
    ^ self basicSend: selector
           withArguments: arguments
           from: self objClass

Obj >> basicSend: selector withArguments: arguments from: aClass
    "Exécute la méthode trouvée à partir de aClass"
    | methodOrNil |
    methodOrNil := aClass lookup: selector.

    ^ methodOrNil
        ifNotNil: [methodOrNil valueWithArguments: (Array with: self),
            arguments]
        ifNil: [self sendError: selector withArgs: arguments]
```

2.6.2 Gestion des erreurs

Plutôt que lever une exception Pharo directement, ObjVlisp envoie un message `error` à l'objet l'objet récepteur du message non compris. Cela donne à l'objet une chance

d'intercepter l'erreur et de la gérer. La méthode `error` (définie dans `ObjObject`) peut être surchargée pour un traitement personnalisé.

Listing 2.20 – Message non compris

```
Obj >> sendError: selector withArgs: arguments
    "Envoie le message error à l'objet"
    ^ self send: #error
           withArguments: {(arguments copyWithFirst: selector)}
```

2.6.3 Super sends

Super permet d'invoquer une méthode de la superclasse, même si elle est surchargée dans la classe courante. Ce n'est pas un objet différent du récepteur (`self` reste le même), juste un point de départ différent pour le lookup. Le paramètre `aSuperclass` est lié lors de la définition de la méthode (via currification), garantissant que le lookup commence au bon endroit même si la méthode est héritée plus bas dans la hiérarchie.

Listing 2.21 – Appel super

```
Obj >> super: selector withArguments: arguments from: aSuperclass
    "Envoie un message en commençant le lookup depuis la superclasse"
    ^ self basicSend: selector
           withArguments: arguments
           from: aSuperclass
```

2.7 Bootstrap du système

2.7.1 Stratégie de bootstrap

Le bootstrap d'ObjVlisp suit une stratégie en trois phases :
— **Phase 1** : Création manuelle d'une version minimale de `ObjClass`
— **Phase 2** : Utilisation de cette classe pour créer `ObjObject`
— **Phase 3** : Recréation complète de `ObjClass` avec héritage

Principe du bootstrap : On crée juste assez de fonctionnalité pour que le système puisse se créer lui-même. C'est le paradoxe du bootstrap : créer les classes avec les classes.

2.7.2 Phase 1 : Création manuelle d'ObjClass

Structure minimale

La première étape consiste à créer "à la main" un tableau représentant `ObjClass`. Cette classe minimale connaît déjà sa structure (variables d'instance, mots-clés) mais prétend hériter d'`ObjObject` qui n'existe pas encore. Cette incohérence temporaire sera résolue en phase 3.

Listing 2.22 – Structure manuelle d'ObjClass

```
Obj class >> manualObjClassStructure
    | class |
    class := Obj new: 6.
    class objClassId: #ObjClass.
    class objName: #ObjClass.
    class objIVs: self classInstanceVariables.
    class objKeywords: #(#name: #superclass: #iv: #keywords: #methodDict
        :).
    class objSuperclassId: #ObjObject.
    class objMethodDict: (IdentityDictionary new: 3).
    ^ class
```

Méthode initialize manuelle

Cette version d'`initialize` ne peut pas utiliser super car `ObjObject` n'existe pas encore. Elle simule l'héritage manuellement avec `computeNewIVFrom: #(class)`, permettant de créer des classes avant que la hiérarchie complète soit en place.

Listing 2.23 – Initialize manuel pour la phase 1

```
Obj class >> defineManualInitializeMethodIn: class
    class
        addMethod: #initialize
        args: 'initArray'
        withBody:
            '| objsuperclass |
            objself initializeUsing: initArray. "Initialize as object"
            objsuperclass := Obj giveClassNamed: objself objSuperclassId
                                ifAbsent: [nil].
            objsuperclass isNil
                ifFalse:
                    [objself objIVs: (objself computeNewIVFrom:
                        objsuperclass objIVs with: objself objIVs)]
                ifTrue:
                    [objself objIVs: (objself computeNewIVFrom:
                        #(#class) with: objself objIVs)].
            objself objKeywords: (objself generateKeywords:
                (objself objIVs copyWithout: #class)).
            objself objMethodDict: (IdentityDictionary new: 3).
            Obj declareClass: objself.
            objself'
```

Assemblage final de la phase 1

Une fois la structure et `initialize` définis, on ajoute les méthodes essentielles de classe : `new` (création d'instances) et `allocate` (allocation mémoire). `ObjClass` est maintenant fonctionnelle, prête à créer d'autres classes.

Listing 2.24 – Création complète d'ObjClass manuel

```
Obj class >> manuallyCreateObjClass
    "self manuallyCreateObjClass"
    | class |
    class := self manualObjClassStructure.
    Obj declareClass: class.
    self defineManualInitializeMethodIn: class.
    self defineNewMethodIn: class.
    self defineAllocateMethodIn: class.
    ^ class

Obj class >> defineNewMethodIn: class
    class
        addMethod: #new
        args: 'initArray'
        withBody:
            '(objself send: #allocate withArguments: #())
                send: #initialize withArguments: {initArray}'.

Obj class >> defineAllocateMethodIn: class
    "Allocation via primitive allocateAnInstance"
    class
        addUnaryMethod: #allocate
        withBody: 'objself allocateAnInstance'
```

2.7.3 Phase 2 : Création d'ObjObject

Avec `ObjClass` fonctionnelle, nous créons maintenant `ObjObject` de manière normale en envoyant `new` à `ObjClass`. `ObjObject` est la racine comportementale : elle définit les méthodes de base (introspection, accès aux variables, erreurs) dont hériteront toutes les classes.

Listing 2.25 – Création normale d'ObjObject

```
Obj class >> createObjObject
    | objObject |
    objObject := self objObjectStructure.
    objObject addUnaryMethod: #class withBody: 'objself objClass'.
    objObject addUnaryMethod: #isClass withBody: 'false'.
    objObject addUnaryMethod: #isMetaclass withBody: 'false'.

    objObject
        addMethod: #error
        args: 'arrayOfArguments'
        withBody: 'Transcript show: ''error '', arrayOfArguments first.
                   ''error '', arrayOfArguments first'.
    objObject
        addMethod: #getIV
        args: 'iv'
        withBody: 'objself valueOfInstanceVariable: iv'.
    objObject
        addMethod: #setIV
        args: 'iv val'
        withBody:
```

```
            'objself at: (objself offsetFromObjectOfInstanceVariable: iv
                )
                    put: val'.
    objObject
        addMethod: #initialize
        args: 'initargs'
        withBody: 'objself initializeUsing: initargs'.
    ^ objObject
```

Listing 2.26 – Structure d'ObjObject

```
Obj class >> objObjectStructure
    ^ (Obj giveClassNamed: #ObjClass)
        send: #new
        withArguments: #(#(#name: #ObjObject #iv: #(#class)))
```

2.7.4 Phase 3 : Recréation complète d'ObjClass

Méthode initialize avec super

Maintenant qu'`ObjObject` existe, on peut recréer `ObjClass` proprement en utilisant **super**. Cette version d'`initialize` remplace la version manuelle de la phase 1 : elle hérite correctement le comportement d'`ObjObject` via super, puis ajoute la logique spécifique aux classes (calcul des variables héritées, génération des mots-clés). Le super call vers `ObjObject #initialize` remplace la condition `ifTrue:/ifFalse:`. Le système est maintenant cohérent et auto-suffisant.

Listing 2.27 – Initialize avec super

```
Obj class >> defineInitializeMethodIn: objClass
    objClass
        addMethod: #initialize
        args: 'initArray'
        withBody:
            'objself super: #initialize
                withArguments: {initArray}
                from: superClassOfClassDefiningTheMethod.
            objself objIVs: (objself
                computeNewIVFrom: (Obj giveClassNamed: objself
                    objSuperclassId) objIVs
                with: objself objIVs).
            objself computeAndSetKeywords.
            objself objMethodDict: IdentityDictionary new.
            objself objSharedVariables isNil ifTrue: [
                objself objSharedVariables: OrderedCollection new
            ].
            Obj declareClass: objself.
            objself'
```

Recréation et remplacement

La nouvelle version d'`ObjClass` est créée normalement via `send: #new`, utilisant l'ancienne version comme bootstrap. Elle déclare maintenant correctement hériter d'`ObjObject`

et possède toutes les variables d'instance nécessaires. Une fois créée, elle remplace l'ancienne dans le dictionnaire global, fermant la boucle du bootstrap.

Listing 2.28 – Recréation complète d'ObjClass

```
Obj class >> createObjClass
    "self bootstrap"
    | objClass |
    objClass := self objClassStructure.
    self defineAllocateMethodIn: objClass.
    self defineNewMethodIn: objClass.
    self defineInitializeMethodIn: objClass.

    "Tests de type pour métaclasses"
    objClass
        addUnaryMethod: #isMetaclass
        withBody: 'objself objIVs includes: #superclass'.

    "Une classe est un objet dont la classe est une métaclasse"
    objClass
        addUnaryMethod: #isClass
        withBody: 'objself objClass send: #isMetaclass withArguments:
           #()'.

    ^ objClass

Obj class >> objClassStructure
    "Création via le mécanisme normal, en spécifiant la superclasse"
    ^ (Obj giveClassNamed: #ObjClass) send: #new
        withArguments: #(#(#name: #ObjClass
                           #iv: #(#name #superclass #iv #keywords #
                              methodDict #sharedVariables)
                           #superclass: #ObjObject))
```

Listing 2.29 – Bootstrap de ObjVlisp

```
Obj class >> bootstrap
  "self bootstrap"

  self initialize.
  self manuallyCreateObjClass.
  self createObjObject.
  self createObjClass.
```

2.8 Création de classes utilisateur

Le bootstrap terminé, nous disposons d'un noyau fonctionnel capable de créer de nouvelles classes. Les classes utilisateur démontrent que le système fonctionne : elles sont créées via le mécanisme normal (`ObjClass » new`), héritent correctement du comportement, et peuvent être instanciées. `ObjPoint` et `ObjColoredPoint` illustrent l'héritage simple et la surcharge de méthodes.

2.8.1 Création d'ObjPoint

`ObjPoint` est la première classe créée entièrement via le système bootstrappé. Elle représente des points 2D avec deux variables d'instance (`x`, `y`) et définit des accesseurs (getters/setters) ainsi qu'une méthode d'affichage. Cette classe prouve que le mécanisme de création, l'héritage des variables (`class` héritée d'`ObjObject`), et l'ajout de méthodes fonctionnent correctement.

Listing 2.30 – Définition de la classe ObjPoint

```
Obj class >> createObjPoint
    | pointClass |

    pointClass := (self giveClassNamed: #ObjClass)
        send: #new
        withArguments: #((#name: #ObjPoint
                          #iv: #(#x #y)
                          #superclass: #ObjObject)).

    pointClass
        addUnaryMethod: #givex
        withBody: 'objself valueOfInstanceVariable: #x'.

    pointClass
        addMethod: #x:
        args: 'newX'
        withBody: 'objself send: #setIV withArguments: {#x. newX}.
           objself'.

    pointClass
        addUnaryMethod: #givey
        withBody: 'objself valueOfInstanceVariable: #y'.

    pointClass
        addMethod: #y:
        args: 'newY'
        withBody: 'objself send: #setIV withArguments: {#y. newY}.
           objself'.

    pointClass
        addUnaryMethod: #display
        withBody:
            'Transcript cr;
            show: ''aPoint with x = ''.
            Transcript show: (objself send: #givex withArguments: #())
               printString.
            Transcript show: '' and y = ''.
            Transcript show: (objself send: #givey withArguments: #())
               printString;
            cr.
            objself'.

    ^ pointClass
```

2.8.2 ObjColoredPoint : héritage d'une classe utilisateur

`ObjColoredPoint` hérite d'`ObjPoint`, démontrant que l'héritage fonctionne à tous les niveaux, pas seulement depuis `ObjObject`. Elle ajoute une variable locale (`color`) et surcharge `display` en utilisant **super** pour réutiliser l'affichage du point avant d'ajouter la couleur. Les variables `x` et `y` sont automatiquement héritées statiquement.

Listing 2.31 – Sous-classe de classe utilisateur

```
Obj class >> createObjPointColored
    | pointColoredClass |

    pointColoredClass := (self giveClassNamed: #ObjClass)
        send: #new
        withArguments: #((#name: #ObjColoredPoint
                          #iv: #(#color)
                          #superclass: #ObjPoint)).

    pointColoredClass
        addUnaryMethod: #color
        withBody: 'objself valueOfInstanceVariable: #color'.

    pointColoredClass
        addMethod: #color:
        args: 'newColor'
        withBody: 'objself send: #setIV withArguments: {#color. newColor
           }.
                   objself'.

    pointColoredClass
        addUnaryMethod: #display
        withBody:
            'objself super: #display
                withArguments: #()
                from: superClassOfClassDefiningTheMethod.
            Transcript cr;
                show: '' with Color = ''.
            Transcript show: (objself send: #color withArguments: #())
               printString;
                cr'.

    ^ pointColoredClass
```

Structure d'une instance :

```
"Un ObjColoredPoint rouge à (10, 20) :"
#(#ObjColoredPoint 10 20 #red)
     |              |  |   |
     classe         x  y   color
```

2.9 Extensions avancées

Le noyau ObjVlisp, bien que minimal, est extensible grâce au modèle de métaclasses explicites. Les extensions suivantes démontrent la puissance de la métaprogrammation :

en créant des métaclasses personnalisées, on peut modifier le comportement de création, d'initialisation, ou d'accès aux objets sans toucher au noyau. Chaque métaclasse définit un "protocole de classe" spécifique que ses instances (qui sont des classes) suivront automatiquement.

2.9.1 Métaclasse abstraite : ObjAbstractClass

Concept

Une classe abstraite ne peut pas être instanciée directement : elle sert de modèle pour ses sous-classes. En Pharo standard, ce concept est une convention (lever une erreur dans `new`), facilement contournable via `basicNew`. Avec `ObjAbstractClass`, l'interdiction est structurelle : la métaclasse surcharge `new` pour empêcher l'instanciation. Toute classe créée avec cette métaclasse hérite automatiquement de cette restriction, sans possibilité de contournement.

Implémentations

La méthode `new` vérifie si la classe est une instance d'`ObjAbstractClass` via `isAbstractMetaclass`. Si oui, elle lève une erreur ; sinon, elle délègue à la version normale via super. Les sous-classes d'une classe abstraite peuvent être concrètes (instanciables) si elles sont créées avec `ObjClass` au lieu d'`ObjAbstractClass`.

Listing 2.32 – Métaclasse abstraite

```
Obj class >> createObjAbstractClass
    | abstractClass |

    abstractClass := (self giveClassNamed: #ObjClass)
        send: #new
        withArguments: #((#name: #ObjAbstractClass
                          #iv: #()
                          #superclass: #ObjClass)).

    abstractClass
        addUnaryMethod: #isAbstractMetaclass
        withBody: 'true'.

    abstractClass
        addMethod: #new
        args: 'initArray'
        withBody:
            '| receiverClass |
            receiverClass := objself send: #class withArguments: #().
            (receiverClass send: #isAbstractMetaclass withArguments: #()
               )
                ifTrue: [objself send: #error
                            withArguments: {objself.
                                            #new.
                                            ''Cannot instantiate abstract
                                               class''}]
                ifFalse: [objself super: #new
                            withArguments: {initArray}
```

```
                                from: superClassOfClassDefiningTheMethod.]'.

    ^ abstractClass
```

Utilisation

Listing 2.33 – Usage de classe abstraite

```
"Créer une classe abstraite"
abstractPoint := objAbstractClass
    send: #new
    withArguments: #(#(#name: #ObjAbstractPoint
                        #iv: #(#x #y)
                        #superclass: #ObjObject)).

"Tentative d'instanciation : échoue"
abstractPoint send: #new withArguments: #(#(#x: 10 #y: 20))
    ">>> Erreur : Cannot instantiate abstract class"

"Créer une sous-classe concrète"
concretePoint := (Obj giveClassNamed: #ObjClass)
    send: #new
    withArguments: #(#(#name: #ConcretePoint
                        #superclass: #ObjAbstractPoint)).

"Instanciation réussie (classe concrète)"
concretePoint send: #new withArguments: #(#(#x: 10 #y: 20))
    ">>> #(#ConcretePoint 10 20)"
```

2.9.2 Génération automatique d'accesseurs

Concept

L'écriture manuelle d'accesseurs (getters/setters) pour chaque variable d'instance est répétitive et source d'erreurs. `ObjClassWithAccessors` automatise cette tâche : toute classe créée avec cette métaclasse génère automatiquement un getter et un setter pour chaque variable d'instance (sauf `class`).

Implémentation

La métaclasse surcharge `initialize` pour ajouter une étape de génération après l'initialisation normale. Pour chaque variable d'instance, elle crée deux méthodes : un getter (même nom que la variable) et un setter (nom suivi de :).

Listing 2.34 – Création de la métaclasse ObjClassWithAccessors

```
Obj class >> createObjClassWithAccessors
    | objClassWithAccessors |

    objClassWithAccessors := (self giveClassNamed: #ObjClass)
        send: #new
        withArguments: #((#name: #ObjClassWithAccessors
                          #iv: #()
                          #superclass: #ObjClass)).

    self defineInitializeMethodWithAccessorsIn: objClassWithAccessors.

    ^ objClassWithAccessors
```

Listing 2.35 – Surcharge de la méthode initialize

```
Obj class >> defineInitializeMethodWithAccessorsIn:
   objClassWithAccessors
    objClassWithAccessors
        addMethod: #initialize
        args: 'initArray'
        withBody:
        '| ivs |
        objself super: #initialize
            withArguments: {initArray}
            from: superClassOfClassDefiningTheMethod.

        ivs := objself objIVs.

        ivs do: [:iv |
            iv = #class ifFalse: [
                objself send: #generateAccessorsFor: withArguments: {iv}
            ]
        ].
        objself'.

    objClassWithAccessors
        addMethod: #generateAccessorsFor:
        args: 'ivName'
        withBody:
        '| getterName setterName |
        getterName := ivName.
        setterName := (ivName asString, '':'') asSymbol.
        objself
            addUnaryMethod: getterName
            withBody: ''objself send: #getIV withArguments: {#'', ivName
                asString, ''}''.
        objself
            addMethod: setterName
            args: ''newValue''
            withBody: ''objself send: #setIV withArguments: {#'', ivName
                asString, ''. newValue}. objself''.
        objself'
```

Fonctionnement :

1. Appel à super pour initialisation standard (variables héritées, mots-clés, etc.)
2. Récupération de la liste complète des variables d'instance
3. Génération des accesseurs pour chaque variable (sauf `class`)

Utilisation

Listing 2.36 – Classe avec accesseurs automatiques

```
"Créer une classe Person avec accesseurs auto"
personClass := objClassWithAccessors
    send: #new
    withArguments: #((#name: #Person
                      #iv: #(name age email)
                      #superclass: #ObjObject)).

"Les accesseurs existent automatiquement !"
aPerson := personClass send: #new
               withArguments: #((#name: #Alice #age: 30 #email: #
                  alice@example.com)).

"Getters générés"
aPerson send: #name withArguments: #().       ">>> #Alice"
aPerson send: #age withArguments: #().        ">>> 30"

"Setters générés"
aPerson send: #age: withArguments: #(31).
aPerson send: #email: withArguments: #(#alice@newdomain.com).
```

2.9.3 Super sécurisé : superFrom :

Problème

Le mécanisme original `super:withArguments:from:` souffre d'un défaut de sécurité : le programmeur peut modifier le sélecteur ou les arguments lors de l'appel super, créant des incohérences dangereuses.

Exemple de code dangereux :

Listing 2.37 – Super non sécurisé - modification possible

```
"Méthode setXY: définie sur ObjColoredPoint"
coloredPointClass
    addMethod: #setXY:
    args: 'newX newY'
    withBody:
        'objself super: #wrongMethod
            withArguments: #(wrongArgs)
            from: superClassOfClassDefiningTheMethod.
        objself'
```

Rien n'empêche le programmeur d'appeler `#wrongMethod` au lieu de `#setXY:` ou de passer des arguments incorrects. Cela viole le principe que super doit invoquer **la même**

méthode dans la superclasse, pas une méthode arbitraire. Les bugs résultants sont difficiles à diagnostiquer car le contrat implicite du super n'est pas respecté.

Solution

`superFrom:` capture automatiquement le sélecteur et les arguments depuis le contexte d'exécution, rendant impossible leur modification. Le programmeur fournit uniquement la superclasse ; le système extrait le reste par introspection de la pile d'appels.

Usage sécurisé :

Listing 2.38 – Super sécurisé - pas de modification possible

```
"Méthode setXY: avec super sécurisé"
coloredPointClass
    addMethod: #setXY:
    args: 'newX newY'
    withBody:
        'objself superFrom: superClassOfClassDefiningTheMethod.
        objself'
```

Le sélecteur (`#setXY:`) et les arguments (`newX`, `newY`) sont capturés automatiquement du contexte, garantissant que c'est bien `setXY:` qui est appelée dans `ObjPoint` avec les bons arguments.

Listing 2.39 – Capture automatique du contexte

```
Obj >> superFrom: aSuperClass
    "Capture automatiquement sélecteur et arguments pour empêcher
       modification"
    | context selector argsArray numArgs sendContext |

    context := thisContext sender.

    "Trouver le contexte de closure (la méthode ObjVlisp)"
    [context notNil] whileTrue: [
        (context closure notNil and: [context closure numArgs > 1])
           ifTrue: [
            "Closure trouvée - extraire les arguments"
            numArgs := context closure numArgs - 1. "Soustraire objself"
            argsArray := Array new: numArgs.
            2 to: context closure numArgs do: [:i |
                argsArray at: i - 1 put: (context tempAt: i)
            ].

            "Trouver le sélecteur depuis l'appel send:withArguments:"
            sendContext := context sender.
            [sendContext notNil] whileTrue: [
                (sendContext method selector == #send:withArguments: or:
                    [
                 sendContext method selector == #basicSend:withArguments
                    :from:])
                ifTrue: [
                    "Le sélecteur est le premier argument"
                    selector := sendContext tempAt: 1.

                    ^ self super: selector
```

```
                            withArguments: argsArray
                            from: aSuperClass
                    ].
                    sendContext := sendContext sender
                ].
            ].
            context := context sender
        ].

        self error: 'Could not find selector in send context'
```

Mécanisme d'introspection :

1. **Remonter la pile** : Parcourir les contextes d'exécution
2. **Trouver la closure** : Identifier le block représentant la méthode (plus d'un argument)
3. **Extraire les arguments** : Lire les valeurs via `tempAt:` (position 2+)
4. **Trouver le sélecteur** : Remonter jusqu'au contexte `send:withArguments:`
5. **Appeler super** : Utiliser le mécanisme normal avec les valeurs capturées

2.9.4 Variables partagées (Shared Variables)

Problème

Les variables partagées permettent à toutes les instances d'une classe de partager des données communes au niveau classe. Par exemple, dans un système de stations de travail, toutes les stations partagent le même domaine réseau (`'inria.fr'`) et la même imprimante par défaut (`'hp-laser-42'`). Ces données ne sont pas propres à chaque instance mais communes à la classe.

Sans support explicite, ces données finissent soit dans des variables globales polluant le namespace, soit dans des variables d'instance de métaclasse faisant confusion entre métadonnées structurelles et données applicatives.

Solution : Slot dédié

La solution consiste à ajouter un septième slot à la structure de classe, dédié exclusivement aux variables partagées. Ce slot contient une `OrderedCollection` de paires `#(nom valeur)`, permettant un accès dynamique et une introspection facile.

Structure étendue :

Listing 2.40 – Extension de la structure de classe avec slot pour variables partagées

```
Obj class >> manualObjClassStructure
    | class |
    class := Obj new: 7.  "7 slots au lieu de 6"
    class objClassId: #ObjClass.
    class objName: #ObjClass.
    class objIVs: #(#class #name #superclass #iv #keywords #methodDict
                    #sharedVariables).
    class objKeywords: #(#name: #superclass: #iv: #keywords: #methodDict
        :
                         #sharedVariables:).
```

```
    class objSuperclassId: #ObjObject.
    class objMethodDict: (IdentityDictionary new: 3).
    class objSharedVariables: OrderedCollection new.  "Slot 7 - nouveau"
    ^ class

Obj class >> objClassStructure
    "Structure pour recréation propre d'ObjClass avec variables partagées"
    ^ (Obj giveClassNamed: #ObjClass) send: #new
        withArguments: #((#name: #ObjClass
                          #iv: #(#name #superclass #iv #keywords #methodDict
                              #sharedVariables)
                          #superclass: #ObjObject))
```

Initialisation avec variables partagées

L'initialisation doit maintenant gérer le mot-clé `#sharedVariables:` et initialiser le slot 7 avec les valeurs fournies ou une collection vide.

Listing 2.41 – Initialize manuel (Phase 1) avec variables partagées

```
Obj class >> defineManualInitializeMethodIn: class
    class
        addMethod: #initialize
        args: 'initArray'
        withBody:
            ...
            "NOUVEAU : Initialisation des variables partagées"
            objself objSharedVariables isNil ifTrue: [
                objself objSharedVariables: OrderedCollection new
            ].

            Obj declareClass: objself.
            objself'
```

Listing 2.42 – Initialize normal (Phase 3) avec variables partagées

```
Obj class >> defineInitializeMethodIn: objClass
    objClass
        addMethod: #initialize
        args: 'initArray'
        withBody:
            ...
            "NOUVEAU : Vérification et initialisation des variables partagées"
            objself objSharedVariables isNil ifTrue: [
                objself objSharedVariables: OrderedCollection new
            ].

            Obj declareClass: objself.
            objself'
```

Rôle de la vérification `isNil` : Permet à `initializeUsing:` de définir les variables partagées avant l'appel à la suite de l'initialisation, sans être écrasées. Si `initializeUsing:` a déjà créé la collection, on ne la réinitialise pas.

Extraction depuis le tableau d'initialisation

La méthode `initializeUsing:` doit extraire le mot-clé `#sharedVariables:` et initialiser le slot 7 avec les paires fournies. Cette version simplifiée détecte le slot des shared variables par sa position (6ème valeur extraite = 7ème slot) et convertit automatiquement le tableau en `OrderedCollection`.

Listing 2.43 – initializeUsing : modifié pour variables partagées

```
Obj >> initializeUsing: anAlternatedArray
    "Initialise l'objet selon le tableau de mots-clés/valeurs."
    | ivValues |
    ivValues := self
        returnValuesFrom: anAlternatedArray
        followingSchema: self objClass objKeywords.

    1 to: ivValues size do: [:i |
        | value |
        value := ivValues at: i.

        "NOUVEAU : Conversion automatique pour le slot 7 (
           sharedVariables)"
        (i = 6 and: [value notNil])
            ifTrue: [self at: i + 1 put: value asOrderedCollection]
            ifFalse: [self at: i + 1 put: value]
    ].
    ^ self
```

Méthodes d'accès aux variables partagées

Deux primitives permettent l'accès sécurisé aux variables partagées : lecture et écriture avec gestion d'erreur si la variable n'existe pas. Ces méthodes parcourent la collection de paires `#(nom valeur)` pour trouver la variable demandée.

Gestion d'erreur : Si la variable n'existe pas, un message `error` est envoyé avec le symbole `#sharedVariableNotFound` et le nom de la variable, permettant un débogage clair. L'utilisateur sait immédiatement quelle variable est manquante.

Listing 2.44 – Méthodes d'accès dans createObjClass

```
Obj class >> createObjClass

    ...

    "NOUVEAU : Méthodes pour accéder aux variables partagées"
    objClass
        addMethod: #sharedVariableValue:
        args: 'varName'
        withBody:
            '| pair |
            pair := objself objSharedVariables
                        detect: [:p | p first = varName]
                        ifNone: [objself send: #error
                                    withArguments: {#
                                       sharedVariableNotFound. varName
                                       }].
```

```
            pair at: 2'.

    objClass
        addMethod: #sharedVariableValue:put:
        args: 'varName newValue'
        withBody:
            '| pair |
            pair := objself objSharedVariables
                        detect: [:p | p first = varName]
                        ifNone: [objself send: #error
                                    withArguments: {#
                                        sharedVariableNotFound. varName
                                        }].
            pair at: 2 put: newValue.
            newValue'.

    ^ objClass
```

Exemple d'utilisation :

Listing 2.45 – Accès aux variables partagées

```
"Lecture"
workstationClass send: #sharedVariableValue: withArguments: {#domain}.
    ">>> 'inria.fr'"

"Écriture"
workstationClass send: #sharedVariableValue:put:
    withArguments: {#domain. 'mit.edu'}.
    ">>> 'mit.edu'"

"Lecture de variable inexistante - déclenche erreur"
workstationClass send: #sharedVariableValue: withArguments: {#
   nonExistent}.
    ">>> Error: sharedVariableNotFound nonExistent"
```

Primitives de structure

Les accesseurs pour le slot 7 suivent le même pattern que les autres slots : un offset constant et des méthodes getter/setter.

Listing 2.46 – Offset et accesseurs pour variables partagées

```
"Offset constant"
Obj >> offsetForSharedVariables
    "Position du slot des variables partagées dans une classe"
    ^ 7

"Accesseur en lecture"
Obj >> objSharedVariables
    "Retourne la collection des variables partagées (OrderedCollection
       de paires)"
    ^ self at: self offsetForSharedVariables

"Accesseur en écriture"
Obj >> objSharedVariables: aCollection
```

```
    "Définit la collection des variables partagées"
    self at: self offsetForSharedVariables put: aCollection
```

Format de la collection : Les variables partagées sont stockées comme une `OrderedCollection` de paires `Array` : `#(#nomVariable valeur)`. Par exemple : `#((#domain 'inria.fr') (#defaultPrinter 'hp-laser'))`.

Utilisation : Classe Workstation

Exemple concret d'une classe utilisant des variables partagées pour stocker le domaine réseau et l'imprimante partagés par toutes les stations.

Listing 2.47 – Création d'une classe avec variables partagées

```
Obj class >> createObjWorkstation
    | workstationClass |

    "Créer la classe avec des variables partagées"
    workstationClass := (self giveClassNamed: #ObjClass)
        send: #new
        withArguments: #((#name: #ObjWorkstation
                          #iv: #(#ipAddress #computerName)
                          #superclass: #ObjObject
                          #sharedVariables: #((domain 'inria.fr')
                                              (defaultPrinter 'hp-laser-42'
                                                  )))).

    "Méthode pour accéder au domaine partagé"
    workstationClass
        addUnaryMethod: #getDomain
        withBody: 'objself objClass send: #sharedVariableValue:
                       withArguments: {#domain}'.

    "Méthode pour accéder à l'imprimante partagée"
    workstationClass
        addUnaryMethod: #getPrinter
        withBody: 'objself objClass send: #sharedVariableValue:
                       withArguments: {#defaultPrinter}'.

    "Méthode pour modifier le domaine (au niveau classe)"
    workstationClass
        addMethod: #setDomain:
        args: 'newDomain'
        withBody:
            'objself objClass send: #sharedVariableValue:put:
                withArguments: {#domain. newDomain}'.

    ^ workstationClass
```

Chapitre 3

Tests et Validations

La validation d'ObjVlisp repose sur une suite de tests systématiques couvrant tous les aspects du système : bootstrap, création de classes, héritage, envoi de messages, et extensions avancées.

3.1 Organisation de la suite de tests

Les tests sont regroupés dans une classe `ObjUserClassesTest` qui hérite de `TestCase`. La méthode `setUp` reconstruit l'intégralité du système ObjVlisp avant chaque test, garantissant l'indépendance et la reproductibilité des tests.

Listing 3.1 – Structure de la classe de test

```
TestCase subclass: #ObjUserClassesTest
    instanceVariableNames: 'aPoint pointColoredClass pointClass
                            aPointColored abstractPointClass
                            objClassWithAccessors
                            pointColoredClassWithAccessors
                            aPointColoredWithAccessors
                            workstationClass'
    ...

ObjUserClassesTest >> setUp
    Obj bootstrap.

    "Création des classes utilisateur"
    self assemblePointClass.
    self assemblePointColoredClass.
    self assemblePointInstance.
    ...

    "Classes avancées (abstraites, accesseurs, variables partagées)"
    self assembleAbstractPointClass.
    self assembleClassWithAccessors.
    self assembleWorkStationClass.

ObjUserClassesTest >> assembleWorkStationClass
    "Crée la classe Workstation avec variables partagées pour les tests"
    workstationClass := (Obj giveClassNamed: #ObjClass)
        send: #new
```

```
        withArguments: #((#name: #ObjWorkstation
                          #iv: #(ipAddress computerName)
                          #superclass: #ObjObject
                          #sharedVariables: #((domain 'inria.fr')
                                              (defaultPrinter 'hp-laser')))
                             ).
```

3.2 Exemples de tests représentatifs

3.2.1 Test de super sécurisé avec capture automatique

Ce test valide que `superFrom:` capture automatiquement le sélecteur et les arguments du contexte d'exécution, éliminant le risque d'erreur manuelle lors des appels super. Le test vérifie que la méthode surchargée dans la sous-classe peut appeler la version de la superclasse sans répéter explicitement le sélecteur ni les arguments.

Listing 3.2 – Validation de la capture automatique du contexte

```
testSuperFrom
    "Vérifie que superFrom: capture automatiquement sélecteur et
       arguments"

    "Définir une méthode dans ObjPoint"
    pointClass := Obj giveClassNamed: #ObjPoint.
    pointClass
        addMethod: #setXY:
        args: 'newX newY'
        withBody:
            'objself send: #setIV withArguments: {#x. newX}.
            objself send: #setIV withArguments: {#y. newY}.
            objself'.

    "Surcharger dans ObjColoredPoint en utilisant superFrom:"
    pointColoredClass := Obj giveClassNamed: #ObjColoredPoint.
    pointColoredClass
        addMethod: #setXY:
        args: 'newX newY'
        withBody:
            'objself superFrom: superClassOfClassDefiningTheMethod.
            objself'.

    "Appeler la méthode - doit automatiquement appeler super avec #setXY
       :"
    aPointColored send: #setXY: withArguments: #(50 75).

    "Vérifier que l'appel super a fonctionné - x et y mis à jour"
    self assert: (aPointColored send: #getIV withArguments: {#x}) equals
       : 50.
    self assert: (aPointColored send: #getIV withArguments: {#y}) equals
       : 75.
```

3.2.2 Test de classe abstraite avec interdiction d'instanciation

Ce test vérifie que les classes créées avec `ObjAbstractClass` ne peuvent pas être instanciées directement.

Listing 3.3 – Validation de l'interdiction d'instanciation

```
testInstanceObjAbstractPointClass
    "Vérifie qu'on ne peut pas instancier une classe abstraite"
    | abstractPoint |

    "Créer une classe abstraite"
    abstractPoint := Obj createObjAbstractClass.

    "Tenter d'instancier doit lever une erreur"
    self should: [
        abstractPoint send: #new withArguments: #(#(#x: 24 #y: 6))
    ] raise: Error.
```

3.2.3 Test de partage des variables de classe

Ce test valide que les variables partagées sont effectivement partagées entre toutes les instances d'une classe et que les modifications au niveau classe sont immédiatement visibles par toutes les instances existantes. C'est le comportement fondamental qui distingue les variables partagées des variables d'instance.

Listing 3.4 – Validation du partage entre instances

```
testSharedVariablesSharedAcrossInstances
    "Vérifie que toutes les instances partagent la même variable"
    | ws1 ws2 |

    "Ajouter méthode pour accéder à la variable partagée depuis les
       instances"
    workstationClass
        addUnaryMethod: #getDomain
        withBody: 'objself objClass send: #sharedVariableValue:
                       withArguments: {#domain}'.

    "Créer deux instances avec des IPs différentes"
    ws1 := workstationClass
        send: #new
        withArguments: #((#ipAddress: '192.168.1.10'
                          #computerName: 'ws1')).

    ws2 := workstationClass
        send: #new
        withArguments: #((#ipAddress: '192.168.1.11'
                          #computerName: 'ws2')).

    "Les deux instances voient la même valeur initiale"
    self assert: (ws1 send: #getDomain withArguments: #())
         equals: 'inria.fr'.
    self assert: (ws2 send: #getDomain withArguments: #())
         equals: 'inria.fr'.
```

```
27
28      "Modifier la variable partagée au niveau classe"
29      workstationClass
30          send: #sharedVariableValue:put:
31          withArguments: {#domain. 'stanford.edu'}.
32
33      "Les deux instances voient immédiatement la nouvelle valeur"
34      self assert: (ws1 send: #getDomain withArguments: #())
35           equals: 'stanford.edu'.
36      self assert: (ws2 send: #getDomain withArguments: #())
37           equals: 'stanford.edu'.
```

3.3 Couverture des tests

La suite de tests complète couvre systématiquement :

— **Bootstrap** : Création manuelle, recréation propre, point fixe
— **Classes utilisateur** : Création, héritage, polymorphisme
— **Envoi de messages** : Simple, super, gestion d'erreurs
— **Variables partagées** : Initialisation, accès, partage, modification
— **Classes abstraites** : Interdiction d'instanciation, héritage
— **Accesseurs automatiques** : Génération, fonctionnement
— **Super sécurisé** : Capture automatique, protection contre modifications

L'exécution complète de la suite valide qu'ObjVlisp implémente fidèlement le modèle à objets minimal tout en offrant les extensions nécessaires à un usage pratique.

Chapitre 4

Conclusion

4.1 Synthèse du travail réalisé

Ce projet a permis l'implémentation complète d'un noyau objet minimal et réflexif inspiré d'ObjVlisp, démontrant qu'un système orienté objet cohérent et auto-suffisant peut être construit avec seulement deux classes fondamentales : `Object` et `Class`. L'implémentation, réalisée en moins de 30 méthodes comme prévu par le modèle original, valide la puissance et l'élégance de cette architecture minimaliste.

4.1.1 Noyau fonctionnel et auto-suffisant

L'implémentation reproduit fidèlement le modèle ObjVlisp à travers ses trois phases de bootstrap :

- **Phase 1 - Création manuelle** : Construction d'`ObjClass` manuellement sans dépendances, établissant le point d'ancrage du système
- **Phase 2 - Première classe utilisateur** : Création d'`ObjObject` via le mécanisme normal, prouvant que le bootstrap a réussi
- **Phase 3 - Point fixe** : Recréation propre d'`ObjClass` utilisant l'héritage et super, fermant la boucle réflexive

4.1.2 Extensions avancées

Au-delà du noyau minimal, plusieurs extensions ont été implémentées pour démontrer l'extensibilité du modèle :

Classes abstraites (`ObjAbstractClass`) : Interdiction structurelle de l'instanciation via surcharge de `new` dans la métaclasse.

Génération automatique d'accesseurs (`ObjClassWithAccessors`) : Métaprogrammation illustrant comment du code peut générer du code. La surcharge d'`initialize` permet de créer dynamiquement un getter et un setter pour chaque variable d'instance, éliminant le code répétitif.

Variables partagées : Extension de la structure de classe avec un septième slot dédié aux variables de classe. Cette solution propre évite la pollution du namespace par des variables globales tout en maintenant une séparation claire entre métadonnées structurelles et données applicatives.

Super sécurisé (`superFrom:`) : Utilisation de l'introspection de contexte pour capturer automatiquement le sélecteur et les arguments, éliminant le risque d'erreur manuelle lors des appels super.

4.2 Leçons tirées de l'implémentation

4.2.1 Tout est objet

Dans ObjVlisp, il n'y a qu'un seul mécanisme pour tout faire : l'envoi de message. Soit pour lire une variable, créer un objet, ou définir une classe, nous envoyons un message.

La dualité des classes

Une classe joue deux rôles simultanés :

1. **Modèle pour ses instances** : Elle définit la structure (variables) et le comportement (méthodes) de ses instances
2. **Objet à part entière** : Elle peut recevoir des messages comme n'importe quel objet

4.2.2 Bootstrap : le paradoxe de l'œuf et la poule

Le bootstrap d'ObjVlisp résout un problème très courant dans les systèmes autodescriptifs : comment créer la première classe alors qu'il faut déjà une classe pour en créer d'autres ?

Le bootstrap se fait en trois étapes, chacune construisant sur la précédente :

— **Phase 1 - Construction manuelle** : On crée `ObjClass` "à la main", en construisant directement un tableau avec les bonnes valeurs. Cette version prétend hériter d'`ObjObject` qui n'existe pas encore. Mais elle fonctionne suffisamment pour démarrer.

```
"Construction manuelle"
objClass := Obj new: 7.   "Tableau de 7 cases"
objClass at: 1 put: #ObjClass.   "Identifiant"
objClass at: 2 put: #ObjClass.   "Nom"
objClass at: 3 put: #ObjObject. "Superclasse (qui n'existe pas
    encore!)"
         ...
```

— **Phase 2 - Première vraie classe** : Maintenant qu'`ObjClass` existe, on peut l'utiliser normalement pour créer `ObjObject`. Cette création prouve que le système fonctionne.

```
"Création normale cette fois"
objObject := objClass send: #new
    withArguments: #((#name: #ObjObject #iv: #(class) ...))
```

— **Phase 3 - Reconstruction propre** : Maintenant qu'`ObjObject` existe vraiment, on peut recréer `ObjClass` proprement. Cette fois, l'héritage d'`ObjObject` est réel et les appels super fonctionnent.

```
"Recréation propre - tout est cohérent maintenant"
newObjClass := objClass send: #new
    withArguments: #((#name: #ObjClass
                      #superclass: #ObjObject ...))
```

4.2.3 Métaclasses explicites vs implicites

ObjVlisp utilise des métaclasses **explicites** : le programmeur décide quelle métaclasse utiliser pour chaque classe. Pharo utilise des métaclasses **implicites** : le système crée automatiquement une métaclasse pour chaque classe. Les deux approches ont des avantages et inconvénients.

Métaclasses explicites (ObjVlisp)

Dans ObjVlisp, créer une classe nécessite de spécifier sa métaclasse :

```
"Classe normale : instance de Class"
normalClass := Class send: #new
    withArguments: #((#name: #Point ...))

"Classe abstraite : instance d'AbstractMetaclass"
abstractClass := AbstractMetaclass send: #new
    withArguments: #((#name: #AbstractPoint ...))
```

Avantages :

1. **Flexibilité totale** : On peut mélanger différentes métaclasses dans la même hiérarchie

```
"Workstation : classe normale"
Workstation instanceOf: Class
Workstation inheritsFrom: Object

"Node : classe abstraite qui hérite de Workstation"
Node instanceOf: AbstractMetaclass
Node inheritsFrom: Workstation

"SpecialWorkstation : classe normale qui hérite de Node"
SpecialWorkstation instanceOf: Class
SpecialWorkstation inheritsFrom: Node
```

   On a deux hiérarchies indépendantes :
   — **Instanciation** : Quelle métaclasse contrôle le comportement de la classe ?
   — **Héritage** : De quelle classe hérite-t-on le comportement des instances ?
2. **Réutilisation de métaclasses** : Une métaclasse peut servir pour plusieurs classes sans lien d'héritage entre elles
3. **Modification du comportement de classe** : Créer une nouvelle métaclasse permet de changer comment les classes se comportent, sans toucher au noyau du système

Inconvénients :

1. **Complexité de développement** : Le programmeur doit penser à deux hiérarchies simultanément
2. **Choix explicite** : Chaque nouvelle classe nécessite de choisir sa métaclasse
3. **Risque d'incohérence** : Rien n'empêche de créer des situations étranges (bien que valides)

Métaclasses implicites (Pharo)

Dans Pharo, chaque classe a automatiquement sa propre métaclasse :

```
"Créer Point crée automatiquement Point class"
Point >>> instance de Point class
Point class >>> hérite de Object class

"Créer ColoredPoint crée automatiquement ColoredPoint class"
ColoredPoint >>> instance de ColoredPoint class
ColoredPoint class >>> hérite de Point class
```

La hiérarchie des métaclasses est **isomorphe** à celle des classes : elle suit exactement la même structure.

Avantages :

1. **Simplicité** : Le programmeur n'a pas à penser aux métaclasses, elles sont automatiques
2. **Cohérence garantie** : La structure est prévisible et stable
3. **Une seule hiérarchie** : On pense uniquement à l'héritage des classes

Inconvénients :

1. **Moins de flexibilité** : On ne peut pas réutiliser une métaclasse pour plusieurs classes non-liées
2. **Héritage de métaclasse forcé** : Si `Vehicle` a une métaclasse spéciale, `Car` (sous-classe de `Vehicle`) hérite de cette métaclasse, même si on ne le souhaite pas

Exemple illustrant la différence

Scénario : On veut une classe abstraite `Vehicle` et une sous-classe concrète `Car`.
Avec ObjVlisp (explicite) :

```
"Vehicle : abstraite (ne peut pas être instanciée)"
Vehicle instanceOf: AbstractMetaclass
Vehicle inheritsFrom: Object

"Car : concrète (peut être instanciée)"
Car instanceOf: Class  "Métaclasse différente !"
Car inheritsFrom: Vehicle

"Résultat :"
Vehicle send: #new >>> ERROR (abstraite)
Car send: #new >>> OK (concrète)
```

Avec Pharo (implicite) :

```
"Vehicle : abstraite"
Vehicle instanceOf: Vehicle class
Vehicle class >>> surcharge new pour lever une erreur

"Car : hérite de Vehicle"
Car instanceOf: Car class
Car class inheritsFrom: Vehicle class  "Hérite new qui lève une erreur !
    "

"Résultat :"
Vehicle new >>> ERROR "abstraite, comme voulu"
Car new >>> ERROR "hérite l'erreur, pas voulu !"
```

Pour que `Car` soit concrète en Pharo, il faut surcharger `new` dans `Car class` pour annuler l'erreur de `Vehicle class`.

4.3 Perspectives et extensions possibles

L'implémentation actuelle d'ObjVlisp est volontairement minimaliste. Plusieurs extensions permettraient d'améliorer notre système. Nous présentons ici deux extensions majeures identifiées lors de l'implémentation.

4.3.1 Accès direct aux variables d'instance depuis les méthodes

Limitation actuelle

Dans notre implémentation, l'accès aux variables d'instance depuis les méthodes passe obligatoirement par des primitives explicites comme `getIV` ou `valueOfInstanceVariable:`. Pour calculer la distance entre deux points, chaque accès à `x` ou `y` nécessite un envoi de message complet avec lookup de méthode.

Extension désirée

L'objectif serait d'autoriser une syntaxe naturelle pour accéder aux variables d'instance directement depuis le corps des méthodes, similaire à ce qui existe dans la plupart des langages orientés objet. Au lieu d'écrire explicitement `objself send: #getIV withArguments: {#x}`, on pourrait simplement écrire `objself x`. De même, pour modifier une variable, on écrirait `objself x: newValue` au lieu de `objself send: #setIV withArguments: {#x. newValue}`.

Défis techniques

Transformation syntaxique lors de la compilation : Comment transformer automatiquement `objself x` en `objself send: #x withArguments: #()` lors de la compilation de la méthode ? Le compilateur doit être capable de reconnaître quelles expressions sont des accès à des variables d'instance (à transformer) et lesquelles sont des envois de messages ordinaires (à préserver tels quels).

Encapsulation et visibilité : Avec des accesseurs explicites, le programmeur peut choisir de ne pas définir de setter pour une variable qu'il veut en lecture seule. Avec

génération automatique, toutes les variables deviennent accessibles en lecture et écriture. Comment préserver le contrôle de l'encapsulation ? Faut-il un mécanisme pour marquer certaines variables comme privées ou en lecture seule ?

4.3.2 Héritage multiple

Limitation actuelle

ObjVlisp supporte uniquement l'héritage simple : une classe ne peut hériter que d'une seule superclasse. Cette limitation se manifeste dans la structure de classe où le slot `superclass` contient un unique identifiant de classe parente. Toute classe a exactement un parent (sauf `Object` qui n'en a aucun).

Cette restriction empêche de modéliser naturellement certaines situations où une classe devrait combiner les caractéristiques de plusieurs parents. Par exemple, un `DrawablePoint` devrait être à la fois un `Point` (avec coordonnées et calcul de distance) et un objet `Drawable` (avec couleur et capacité de dessin). Avec l'héritage simple, il faut choisir duquel hériter et dupliquer manuellement les fonctionnalités de l'autre.

Extension désirée

L'héritage multiple permettrait à une classe d'hériter de plusieurs superclasses simultanément. Le slot `superclass` deviendrait `superclasses` (une collection), et une classe pourrait spécifier plusieurs parents lors de sa création. La classe hériterait automatiquement de toutes les variables d'instance et méthodes de tous ses parents.

Défis techniques

Fusion des variables d'instance : Comment combiner les variables de plusieurs parents ? Si `Point` définit `#(class x y)` et `Drawable` définit `#(class color lineWidth)`, quelle sera la structure de `DrawablePoint` ? L'ordre importe pour la stabilité des offsets. Que faire si deux parents définissent une variable avec le même nom mais des sémantiques différentes ?

Ordre de résolution des méthodes : C'est le problème central de l'héritage multiple. Lors du lookup d'une méthode, dans quel ordre parcourir les superclasses ? Si `DrawablePoint` hérite de `Point` et `Drawable`, et qu'on cherche une méthode, commence-t-on par `Point` ou `Drawable` ? Et si les deux redéfinissent la même méthode, laquelle choisit-on ?

Conflits de méthodes : Si deux parents définissent la même méthode avec des comportements différents, laquelle hérite-t-on ? Doit-on forcer le programmeur à redéfinir explicitement la méthode pour résoudre l'ambiguïté ? Ou utilise-t-on l'ordre de résolution pour choisir automatiquement ?