



**UNIVERSITÉ DE
MONTPELLIER**

HAI914I

**Module : HAI931I - M2-S1 Informatique GL : MétaProgrammation et
Réflexivité.**

Projet Pharo sur la meta - programmation

Réalisé par :

M^{lle} MEKHNACHE Toudherth – M2 GL

Promotion 2023/2024

Table des matières

Introduction	1
1 Introduction	2
2 Développement Technique Principal	3
2.1 Building a minimal reflective class-based kernel	3
2.1.1 Structure et Primitives	3
2.1.2 Structure d'une Classe dans ObjVLisp	3
2.1.3 Trouver la classe d'un objet :	5
2.1.4 Gestion des Offsets des Variables d'Instance	5
2.1.5 Allocation d'Instance et Initialisation des Objets dans ObjVLisp . .	6
2.1.6 Gestion des Mots-clés dans ObjVLisp	7
2.1.7 Initialisation des Objets	7
2.1.8 Héritage Statique des Variables d'Instance	7
2.1.9 Gestion des Méthodes et Envoi de Messages dans ObjVLisp	8
2.1.10 Send message et super dans ObjVLisp	8
2.1.11 Recherche de Méthodes (Method Lookup) dans ObjVLisp	9
2.1.12 Comparaison des Deux Versions de basicSend :withArguments :from :	9
2.2 Bootstrapping du Système dans ObjVLisp	10
2.2.1 Création manuelle d'ObjClass	11
2.2.2 Création de ObjObject	11
2.2.3 Création de ObjClass	12
2.2.4 Premières Classes Utilisateur	12
2.2.5 Première MetaClasse Utilisateur	13
2.2.6 Nouvelles Fonctionnalités à Implémenter	13
3 Conclusion	16
Conclusion	16

Chapitre 1

Introduction

Dans le domaine de la programmation orientée objet, la flexibilité et la puissance des systèmes de classes sont des aspects cruciaux pour le développement de logiciels robustes et évolutifs. Ce projet vise à explorer ces facettes en construisant un noyau minimal réflexif basé sur les classes, inspiré par le modèle ObjVLisp. ObjVLisp, conçu initialement par P. Cointe, s'inspire du noyau de Smalltalk-78 et se distingue par ses métaclasses explicites et sa composition de seulement deux classes fondamentales : `Object` et `Class`. Ce noyau offre une plateforme idéale pour étudier les relations fondamentales entre les classes et les objets, ainsi que pour examiner des concepts avancés tels que l'héritage, l'allocation et l'initialisation d'objets, et la résolution de messages.

Chapitre 2

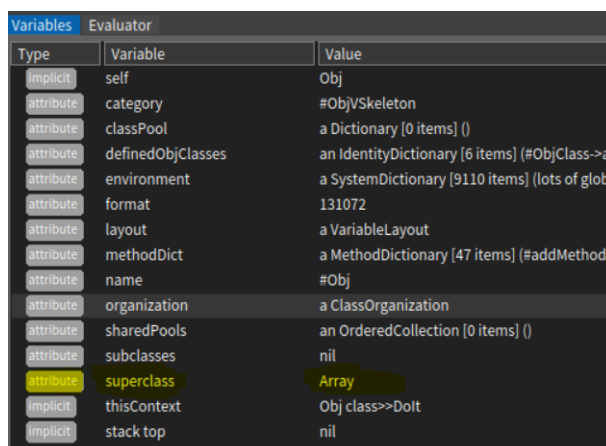
Développement Technique Principal

2.1 Building a minimal reflective class-based kernel

2.1.1 Structure et Primitives

Dans cette section, nous abordons la représentation des objinstances dans ObjVLisp. Nous avons choisi de représenter ces objinstances sous forme de tableaux, en utilisant la classe Obj, une sous-classe de Array.

Nous observons que la classe Obj hérite bien de Array en analysant la définition de la classe Obj. La figure 2.1 suivante a été utilisée pour confirmer cette relation d'héritage.



The screenshot shows a 'Variables' pane with a table of variables and their values. The 'superclass' variable is highlighted in yellow, showing its value as 'Array'.

Type	Variable	Value
implicit	self	Obj
attribute	category	#ObjVSkeleton
attribute	classPool	a Dictionary [0 items] ()
attribute	definedObjClasses	an IdentityDictionary [6 items] (#ObjClass->ar
attribute	environment	a SystemDictionary [9110 items] (lots of globa
attribute	format	131072
attribute	layout	a VariableLayout
attribute	methodDict	a MethodDictionary [47 items] (#addMethod:a
attribute	name	#Obj
attribute	organization	a ClassOrganization
attribute	sharedPools	an OrderedCollection [0 items] ()
attribute	subclasses	nil
attribute	superclass	Array
implicit	thisContext	Obj class>>Dolt
implicit	stack top	nil

FIGURE 2.1 – La classe Obj hérite de la superclass Array

Cette déclaration indique que la classe Obj est effectivement une sous-classe de Array. Ce choix de représentation des objinstances en tant que tableaux est conforme avec la structure interne de Pharo et facilite la manipulation et l'accès aux attributs des objets dans le cadre de notre implémentation d'ObjVLisp.

2.1.2 Structure d'une Classe dans ObjVLisp

Dans ObjVLisp, la structure d'une classe est définie de manière à encapsuler tous les aspects nécessaires pour décrire et gérer un objet.

L'objectif ici est d'implémenter un ensemble de primitives en ObjVLisp pour la classe ObjPoint. Ces primitives permettront de gérer et d'accéder aux différentes composantes de

la classe, telles que l'identifiant de la classe, le nom, la superclasse, les variables d'instance, les mots-clés d'initialisation, et le dictionnaire de méthodes. La figure 2.2 représente une présentation de la structure d'une classe.

```

#(
  #ObjClass      offsetForClass (1)
  #ObjPoint      offsetForName (2)
  #ObjObject     offsetForSuperclass (3)
  #(class x y)   offsetForIVs (4)
  #(:x :y)       offsetForKeywords (5)
  nil            offsetForMethodDict (6)
)

```

FIGURE 2.2 – Présentation de la structure d'une classe

1. **Gestion de l'Identifiant de la Classe (Class ID) :** Cette partie explique l'utilisation des primitives `objClassId` et `objClassId : aSymbol` dans `ObjVLisp`. Elles sont cruciales pour identifier la classe d'une instance et permettent de récupérer ou modifier cet identifiant.

```

|| objClassId
|   ^ self at: self offsetForClass .
|
|| objClassId: anObjClassId
|   ^ self at: self offsetForClass put: anObjClassId.

```

2. **Gestion du Nom de la Classe :** Les primitives `objName` et `objName : aSymbol` facilitent la lecture et la définition du nom de la classe. Elles sont essentielles pour l'identification et la documentation des classes dans `ObjVLisp`.

```

|| objName
|   ^ self at: self offsetForName .
|
|| objName: aName
|   ^ self at: self offsetForName put: aName.

```

3. **Gestion de la Superclasse :** Cette section présente les primitives pour gérer l'identifiant de la superclasse des objets, un aspect clé pour comprendre et manipuler la hiérarchie des classes dans `ObjVLisp`.

```

|| objSuperclassId
|   ^ self at: self offsetForSuperclass .
|
|| objSuperclassId: anObjClassId
|   ^ self at: self offsetForSuperclass put: anObjClassId .

```

4. **Gestion des Variables d'Instance :** Les primitives associées aux variables d'instance permettent de récupérer et de modifier la liste des variables d'une classe, un élément crucial pour la définition des propriétés des objets.

```

|| objIVs
|   ^ self at: self offsetForIVs.
|
|| objIVs: anOrderedCollection
|   ^ self at: self offsetForIVs put: anOrderedCollection .

```

5. Gestion de la Liste des Mots-clés d'Initialisation :

Ces primitives sont utilisées pour gérer les mots-clés d'initialisation des classes, facilitant ainsi la configuration des instances lors de leur création.

```

|| objKeywords
|   ^ self at: self offsetForKeywords.
|
|| objKeywords: anOrderedCollection
|   ^ self at: self offsetForKeywords put: anOrderedCollection .

```

6. Gestion du Dictionnaire de Méthodes :

Ces primitives sont essentielles pour accéder et manipuler le dictionnaire de méthodes de la classe, permettant de contrôler les comportements et les opérations disponibles pour les instances.

```

|| objMethodDict
|   ^ self at: self offsetForMethodDict .
|
|| objMethodDict: aDictionary
|   ^ self at: self offsetForMethodDict put: aDictionary.

```

2.1.3 Trouver la classe d'un objet :

Dans cette partie on doit implémenter la primitive `objClass` dans `ObjVLisp`, qui est essentielle pour identifier la classe d'un objet donné.

La primitive `objClass` a été implémentée pour retourner l'objet de classe représentant la classe de l'instance sur laquelle elle est appelée. Elle utilise la structure interne de l'objet pour accéder à son identifiant de classe et, en utilisant la primitive existante `giveClassNamed : aSymbol` au niveau de la classe `Obj`, elle renvoie l'objet de classe correspondant.

L'implémentation de cette méthode est définie comme suit :

```

|| objClass
|   ^ Obj giveClassNamed: self objClassId.

```

2.1.4 Gestion des Offsets des Variables d'Instance

Méthode `offsetFromClassOfInstanceVariable : aSymbol` Cette méthode est conçue pour trouver l'index (ou l'offset) d'une variable d'instance nommée `aSymbol` dans une classe donnée. Si `aSymbol` n'est pas présent dans la liste des variables d'instance de la classe, la méthode retourne 0.

```

|| offsetFromClassOfInstanceVariable: aSymbol
|   listIV index |
|   listIV := self objIVs.
|   index := listIV indexOf: aSymbol.
|   ^ index ifNotNil: [ index ] ifNil: [ 0 ].

```

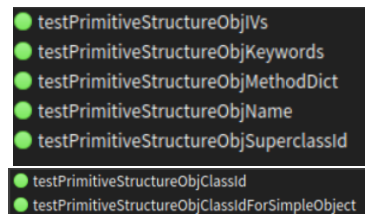
Cette méthode récupère d'abord la liste des variables d'instance de la classe, puis recherche l'index de `aSymbol` dans cette liste. Si `aSymbol` est trouvé, son index est renvoyé ; sinon, la méthode renvoie 0.

Méthode `offsetFromObjectOfInstanceVariable : aSymbol` Cette méthode renvoie l'offset de la variable d'instance `aSymbol` dans un objet. Si la variable n'est pas définie dans l'objet, une erreur est levée.

```
offsetFromObjectOfInstanceVariable: aSymbol
| offset |
offset := self objClass offsetFromClassOfInstanceVariable:
aSymbol.
offset = 0 ifTrue:
[ ~ self error: 'Variable d'instance non définie: ',
aSymbol asString ].
~ offset
```

Ici, la méthode détermine d'abord l'offset de `aSymbol` en utilisant `offsetFromClassOfInstanceVariable` : sur la classe de l'objet. Si l'offset est 0 (ce qui signifie que la variable n'est pas définie), une erreur est levée.

Ces méthodes sont cruciales pour une gestion précise des variables d'instance dans ObjVLisp. Elles permettent un accès et une manipulation efficaces des propriétés des objets, ce qui est essentiel pour de nombreuses opérations orientées objet. Après l'implémentation de ces méthodes on observe que les tests de 'step1-tests-structure of objects' et 'step2-tests-structure of classes' ont été exécutés avec succès.



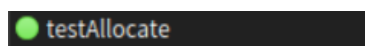
```
testPrimitiveStructureObjIvs
testPrimitiveStructureObjKeywords
testPrimitiveStructureObjMethodDict
testPrimitiveStructureObjName
testPrimitiveStructureObjSuperclassId
testPrimitiveStructureObjClassId
testPrimitiveStructureObjClassIdForSimpleObject
```

2.1.5 Allocation d'Instance et Initialisation des Objets dans ObjVLisp

La primitive '`allocateAnInstance`' a été implémentée dans le protocole '`instance allocation`'. Cette méthode, lorsqu'elle est appelée sur un `objClass`, crée une nouvelle instance de cette classe avec toutes les variables d'instance initialisées à nil et l'`objClassId` correctement défini.

Dans le cas de la classe `ObjPoint`, qui a deux variables d'instance, l'appel de `ObjPoint allocateAnInstance` renvoie `##ObjPoint nil nil`, indiquant une instance nouvellement créée avec des variables d'instance non initialisées.

La validité de `allocateAnInstance` a été confirmée par des tests unitaires. Le test `testAllocate` dans la classe `ObjTest` vérifie que la nouvelle instance a le bon identifiant de classe et que toutes ses variables d'instance sont nil.



```
testAllocate
```

2.1.6 Gestion des Mots-clés dans ObjVLisp

Pour exécuter avec succès le test `testKeywords` dans `ObjVLisp`, il est essentiel d'avoir préalablement implémenté les méthodes `generateKeywords` et `keywordValue`. Ces méthodes fournissent la fonctionnalité de base nécessaire pour manipuler les mots-clés et leurs valeurs associées, qui sont cruciaux pour le test `testKeywords`.

Implémentation de `generateKeywords`

La méthode `generateKeywords` transforme une collection de symboles en mots-clés. Chaque élément du tableau passé en argument est converti en un mot-clé en lui ajoutant `'.'` et en le transformant en symbole.

```
|| generateKeywords: anArray  
|| ^anArray collect: [:e | (e , '.') asSymbol]
```

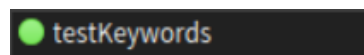
Cette méthode est utilisée pour créer un tableau de mots-clés à partir d'un tableau de symboles, facilitant ainsi la spécification des valeurs des variables d'instance lors de l'initialisation des objets.

Implémentation de `keywordValue`

La méthode `keywordValue` permet de récupérer la valeur associée à un mot-clé spécifique dans un tableau. Si le mot-clé n'est pas trouvé, une valeur par défaut est retournée.

```
|| keywordValue: aSymbol getFrom: anArray ifAbsent: aDefaultValue  
|| | i |  
|| i := anArray indexOf: aSymbol ifAbsent: nil.  
|| ^i isNil  
|| ifTrue: [aDefaultValue]  
|| ifFalse: [anArray at: i + 1]
```

Cette méthode est essentielle pour extraire des valeurs spécifiques à partir d'un tableau de mots-clés et de valeurs, jouant un rôle clé dans l'initialisation des objets avec des paramètres spécifiques.



2.1.7 Initialisation des Objets

La méthode `initializeUsing : anAlternatedArray` a été implémentée pour initialiser une instance d'`ObjVLisp` (`ObjObject`) selon les directives fournies par `anAlternatedArray`. Cette approche permet d'initialiser dynamiquement les instances avec des valeurs spécifiques.

La méthode prend un tableau alterné `anAlternatedArray`, qui contient des paires de mots-clés et de valeurs, et initialise l'instance en fonction de ces paires.

Dans cette implémentation, `returnValuesFrom : anAlternatedArray followingSchema : self objClass objKeywords.` : est utilisée pour extraire les valeurs de `anAlternatedArray` suivant le schéma défini par les mots-clés de la classe de l'objet (`objKeywords`). Ensuite, chaque valeur extraite est assignée à la variable d'instance correspondante de l'objet.

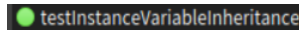
2.1.8 Héritage Statique des Variables d'Instance

La méthode `'computeNewIVFrom :'` a été conçue pour gérer l'héritage des variables d'instance dans les classes en `ObjVLisp`. Son but est de créer une collection ordonnée des

variables d'instance qui combine celles de la superclasse (`superIVOrdCol`) et celles définies localement (`localIVOrdCol`), tout en évitant les doublons et en préservant l'ordre.

Dans cette implémentation, si `superIVOrdCol` est nil, la méthode retourne simplement `localIVOrdCol`. Sinon, elle crée une copie de `superIVOrdCol`, puis itère sur `localIVOrdCol` pour ajouter chaque élément qui n'est pas déjà présent dans la collection résultante.

L'implémentation de `computeNewIVFrom :with :` représente un élément clé de la gestion des variables d'instance dans `ObjVLisp`, facilitant un héritage structuré et ordonné des propriétés des classes. Cette méthode contribue à maintenir l'intégrité et la structure des classes dans la hiérarchie d'héritage. Le test de cette implémentation a été validé.



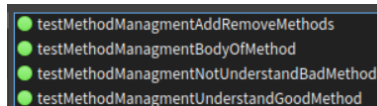
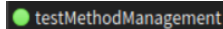
2.1.9 Gestion des Méthodes et Envoi de Messages dans `ObjVLisp`

Les primitives **`addMethod`**, **`removeMethod`**, **`doesUnderstand`** ont été implémentées pour gérer la définition, la suppression et la vérification des méthodes dans `ObjVLisp`, nous avons implémenté la méthode `bodyOfMethod` pour que les tests fonctionnent correctement.

`bodyOfMethod : aSelector` Cette méthode retourne le corps de la méthode associée à `aSelector` dans la classe réceptrice.

```
|| bodyOfMethod: aSelector
||   ^ self objMethodDict at: aSelector ifAbsent: [ nil ]
```

Les assertions dans le test 'testMethodManagement' ont validé la capacité de la classe à comprendre les sélecteurs de méthodes, à ajouter et à supprimer des méthodes, et à récupérer le corps des méthodes définies.



2.1.10 Send message et super dans `ObjVLisp`

Cette section décrit l'implémentation des primitives clés pour l'envoi de messages dans `ObjVLisp`, y compris la gestion des messages normaux et des messages super.

`send : selector withArguments : arguments` Cette méthode envoie un message identifié par `selector` au récepteur (`self`), avec un tableau d'arguments `arguments`. Elle s'appuie sur `basicSend :selector withArguments :arguments from :` pour rechercher la méthode dans la classe du récepteur.

```
|| send: selector withArguments: arguments
||   ^ self basicSend: selector withArguments: arguments from: self
||     objClass.
```

`super : selector withArguments : arguments from : aSuperclass` Permet d'invoquer une méthode surclassée de la superclasse.


```

super: selector withArguments: arguments from: aSuperclass
  ^ self basicSend: selector withArguments: arguments from:
    aSuperclass.

```

L'implémentation de ces primitives illustre la flexibilité de l'envoi de messages en ObjVLisp, permettant à la fois des envois de messages normaux et des appels à des méthodes surclassées, tout en gérant correctement les erreurs.

la méthode de test `testMethodLookup` suivra pour vérifier que la logique d'envoi de messages super fonctionne comme prévu.

 `testMethodLookup`

2.1.11 Recherche de Méthodes (Method Lookup) dans ObjVLisp

La méthode `lookup: selector` est implémentée pour effectuer la recherche d'une méthode dans la classe réceptrice (`self`), qui est une `objClass`. Cette recherche est essentielle pour résoudre les appels de méthode dans ObjVLisp.


La méthode cherche d'abord la méthode correspondant au sélecteur `selector` dans le dictionnaire de méthodes de la classe réceptrice.


```

lookup: selector
  | method superclassid |
  method := self objMethodDict at: selector ifAbsent: [nil].
  method ifNil:
    [ superclassid := self objSuperclassId.
      superclassid
        ifNil: [ ^ nil ]
        ifNotNil: [ ^ self objSuperclass lookup: selector ] ]
  ifNotNil: [ ^ method. ]

```

Cependant, si elle n'est pas trouvée dans la classe réceptrice, la recherche se poursuit récursivement dans la superclasse. Ce processus se répète jusqu'à ce que la méthode soit trouvée ou que la hiérarchie de classes soit entièrement parcourue. Dans le cas où la méthode reste introuvable dans toute la chaîne d'héritage, la méthode retourne `nil`, indiquant l'échec de la recherche. L'importance de cette implémentation réside dans sa capacité à maintenir l'intégrité du modèle d'envoi de messages d'ObjVLisp, tout en gérant efficacement la hiérarchie d'héritage des classes. Pour valider cette fonctionnalité, des tests spécifiques tels que `testNilWhenErrorInLookup` et `testRaisesErrorSendWhenErrorInLookup` sont exécutés, assurant ainsi que la méthode se comporte comme prévu dans divers scénarios.

 `testNilWhenErrorInLookup`

 `testErrorRaisedSendWhenErrorInLookup`

2.1.12 Comparaison des Deux Versions de `basicSend :withArguments :from :`

2.1.12.1 Première Implémentation de `basicSend :withArguments :from :`

Dans cette version, la méthode commence par rechercher une méthode correspondant au sélecteur fourni (`selector`). Si une telle méthode est trouvée (c'est-à-dire, `methodOrNil` n'est pas `nil`), elle est alors exécutée avec les arguments spécifiés. En revanche, si aucune

méthode correspondante n'est trouvée (`methodOrNil` est `nil`), une erreur Pharo standard est immédiatement signalée, indiquant généralement un message non compris.

2.1.12.2 Deuxième Implémentation de `basicSend :withArguments :from :`

Dans cette version modifiée, bien que la structure de base reste inchangée, un changement significatif est introduit dans la gestion des erreurs. Au lieu de signaler directement une erreur lorsque `methodOrNil` est `nil`, la méthode fait appel à `sendError :withArgs :`. Cette approche innovante confère à l'objet récepteur la responsabilité de gérer l'erreur. Ainsi, plutôt que de générer immédiatement une erreur, cette responsabilité est déléguée, offrant ainsi une plus grande flexibilité.

Cette modification est en harmonie avec les principes de la programmation orientée objet (OOP). En utilisant `sendError`, le sélecteur et les arguments sont encapsulés dans un tableau qui est ensuite envoyé à la méthode `error` de l'objet récepteur. Ce mécanisme permet à l'objet récepteur de décider comment gérer l'erreur, lui donnant ainsi l'opportunité de réagir de manière appropriée selon son contexte et sa logique interne.

L'adoption de `sendError` pour la gestion des messages non compris représente une avancée significative par rapport à l'approche traditionnelle. Elle offre une flexibilité accrue, se conforme davantage aux principes de l'OOP, et améliore la maintenabilité ainsi que l'évolutivité de notre système.

Le teste `testSendErrorRaisesErrorSendWhenErrorInLookup` a été exécuté avec succès dans mon projet. Ce test a validé le bon fonctionnement de la gestion des erreurs comme le montre la figure 2.4.

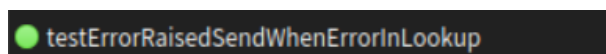


FIGURE 2.3 – methode de test `testSendErrorRaisesErrorSendWhenErrorInLookup`

2.2 Bootstrapping du Système dans ObjVlisp

Le processus de "bootstrapping" implique la création des classes fondamentales `ObjObject` et `ObjClass` à partir d'elles-mêmes. Ce processus se déroule en trois étapes principales :

- Création Manuelle de `ObjClass` : D'abord, la classe `ObjClass` est créée manuellement pour établir la structure de base nécessaire.
- Création de `ObjObject` avec `ObjClass` : Ensuite, `ObjClass` est utilisée pour créer la classe `ObjObject` de manière standard, en s'appuyant sur la structure déjà établie de `ObjClass`.
- Reconstruction Complète de `ObjClass` : Finalement, `ObjClass` est recrée complètement pour assurer sa pleine fonctionnalité.

Pour garantir que chaque phase de ce processus se déroule correctement et efficacement, une série de tests détaillés a été mise en place dans la classe `ObjTestBootstrap`. L'objectif de ces tests est de s'assurer que les classes `ObjClass` et `ObjObject` fonctionnent comme prévu, en validant la cohérence et la performance des implémentations réalisées.

2.2.1 Création manuelle d'ObjClass

Dans le cadre du protocole 'bootstrap objClass manual' au niveau de la classe, nous avons procédé à la lecture de l'implémentation de la primitive `manualObjClassStructure`. Cette primitive est fondamentale car elle renvoie un `objObject` qui représente la classe `ObjClass`, établissant ainsi la structure de base essentielle pour le bootstrapping de notre système.

Nous avons également exécuté avec succès la méthode de test `testManuallyCreateObjClassStructure`. Cette étape de validation a été cruciale pour s'assurer que la structure de `ObjClass` créée manuellement répondait à nos attentes et spécifications. Le test a confirmé que l'objet renvoyé par `manualObjClassStructure` se comporte correctement et possède les caractéristiques attendues de la classe `ObjClass`.

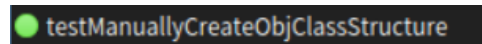


FIGURE 2.4 – methode de test `testManuallyCreateObjClassStructure`

Dans le système `ObjVlisp`, `defineAllocateMethodIn` : ajoute la méthode `allocate` à une classe, permettant la création de nouvelles instances, tandis que `defineNewMethodIn` : implémente la méthode `new`, qui combine la création et l'initialisation d'instances avec des paramètres spécifiques. Ces deux méthodes sont essentielles pour la gestion efficace des objets dans le système.

Le test illustré dans la figure 2.5 a validé avec succès la fonctionnalité de l'objet renvoyé par `testManuallyCreateObjClassAllocate`. Ce test a confirmé que les méthodes `allocate` et `new` implémentées fonctionnent correctement, en vérifiant que les instances de classe sont non seulement créées, mais également initialisées conformément aux exigences définies.

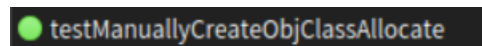


FIGURE 2.5 – methode de test `testManuallyCreateObjClassAllocate`

2.2.2 Création de ObjObject

La création de `ObjObject`, la première classe concrète du système `ObjVlisp`, représente une étape fondamentale dans l'établissement du noyau du système. Ce processus se déroule en plusieurs étapes essentielles :

- **Structuration de ObjObject** : La primitive `objObjectStructure` est utilisée pour créer la classe `ObjObject`. Elle invoque la méthode `new` sur `ObjClass`, spécifiant `#ObjObject` comme le nom de la classe et `class` comme son unique variable d'instance. `ObjObject` est distinct en ce qu'il n'a pas de superclasse, étant la racine du graphe d'héritage.
- **Définition des Méthodes Comportementales** : À travers la méthode `createObjObject`, des méthodes fondamentales sont ajoutées à `ObjObject`. Cela inclut `#class`, qui retourne la classe de l'instance, `#isClass` et `isMetaclass`, qui retournent toutes deux `false`, ainsi que des méthodes pour la gestion d'erreurs, la manipulation des variables d'instance, et l'initialisation des objets.
- **Tests de Validation** : Pour assurer la correcte implémentation et fonctionnalité de `ObjObject`, des méthodes de test telles que `testCreateObjObjectStructure`,

testCreateObjObjectMessage, et testCreateObjObjectInstanceMessage sont exécutées. Ces tests vérifient que ObjObject est correctement formé et que ses méthodes fonctionnent comme attendu, comme le montre les figures 2.6 et 2.7.

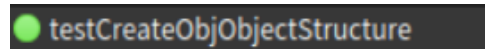


FIGURE 2.6 – methode de test testCreateObjObjectStructure

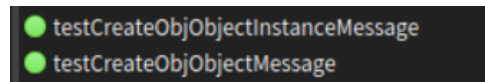


FIGURE 2.7 – methode de test testCreateObjObjectMessage et testCreateObjObjectInstanceMes

2.2.3 Création de ObjClass

La création de la classe ObjClass dans ObjVLisp implique plusieurs étapes clés pour établir sa structure et son comportement. Voici un résumé des étapes principales :

- La primitive objClassStructure responsable de la création de la structure de base de ObjClass. Elle est implémentée dans Obj class et invoque la méthode new sur elle-même pour créer la classe ObjClass.
- La methode createObjClass utilise objClassStructure pour obtenir une instance de ObjClass. Elle configure ensuite ObjClass en ajoutant diverses méthodes. En plus de ça la classe possède des methodes telles que #isClass et isMetaclass sont ajoutées à ObjClass pour définir son comportement.
- Méthodes comme defineAllocateMethodIn :, defineNewMethodIn :, et defineInitializeMethodIn : sont implémentées pour ajouter des fonctionnalités supplémentaires à ObjClass.

Le testCreateObjClassMessage sont exécutés pour s'assurer que ObjClass est correctement créée et que ses méthodes fonctionnent comme prévu.

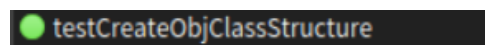


FIGURE 2.8 – La methode de testCreateObjClassStructure

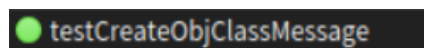


FIGURE 2.9 – La methode de testCreateObjClassMessage

2.2.4 Premières Classes Utilisateur

2.2.4.1 ObjPoint :

La classe ObjPoint est créée en envoyant le message new à ObjClass avec des arguments spécifiant le nom de la classe (ObjPoint), les variables d'instance ((x y)), et la superclasse (ObjObject).

Des messages définis dans ObjObject sont envoyés à cette instance pour obtenir et définir les valeurs des variables d'instance (getIV, setIV).

Des méthodes telles que `x`, `x :`, et `display` sont ajoutées à la classe `ObjPoint`. Par exemple, `addUnaryMethod : givex` pour obtenir la valeur de `x`, et `addUnaryMethod : display` pour afficher les informations de l'objet.

Le diagramme uml présent dans la figure 2.10 représente la classe `ObjPoint` comment elle est représenté, avec une instance `aPoint`.

2.2.4.2 ObjColoredPoint :

Cette partie décrit la création de la classe `ObjColoredPoint`, qui étend les fonctionnalités de base de `ObjObject` en y ajoutant des caractéristiques de couleur, et comment interagir avec ses instances en utilisant des méthodes spécifiques. La classe est créée, des instances sont générées et manipulées, et de nouvelles méthodes sont définies et testées pour enrichir la classe avec des fonctionnalités supplémentaires liées à la couleur. le diagramme de classe uml de la figure 2.10 définit comment cette classe est présentée ainsi qu'une instance de cette classe.

2.2.5 Première MetaClasse Utilisateur

La section "A First User Metaclass : `ObjAbstract`" de projet en concerne la création d'une métclasse nommée `ObjAbstract`. Cette métclasse est conçue pour définir des classes abstraites, c'est-à-dire des classes qui ne peuvent pas créer d'instances.

`ObjAbstract` est une métclasse qui définit des classes abstraites, qui doit être configurée pour lever une erreur lorsqu'on essaie de créer une instance avec le message `new`.

L'utilisation de `ObjAbstractClass` pour définir `ObjAbstractPoint` comme classe abstraite illustre comment les développeurs peuvent utiliser cette métaclasse pour créer des structures de classe spécifiques dans le système `ObjVLisp`.

Cette approche montre une compréhension approfondie de la programmation orientée objet, en particulier dans la création de hiérarchies de classes et la définition de comportements de classe spéciaux, comme présenté dans la figure 2.10.

2.2.6 Nouvelles Fonctionnalités à Implémenter

cette partie du projet démontre une volonté d'étendre les capacités d'`ObjVLisp` en incorporant des fonctionnalités avancées qui automatisent certaines tâches de développement et offrent une gestion plus sophistiquée des données au niveau de la classe. Cela reflète une évolution vers un système plus robuste et efficace, capable de gérer des scénarios de programmation complexes.

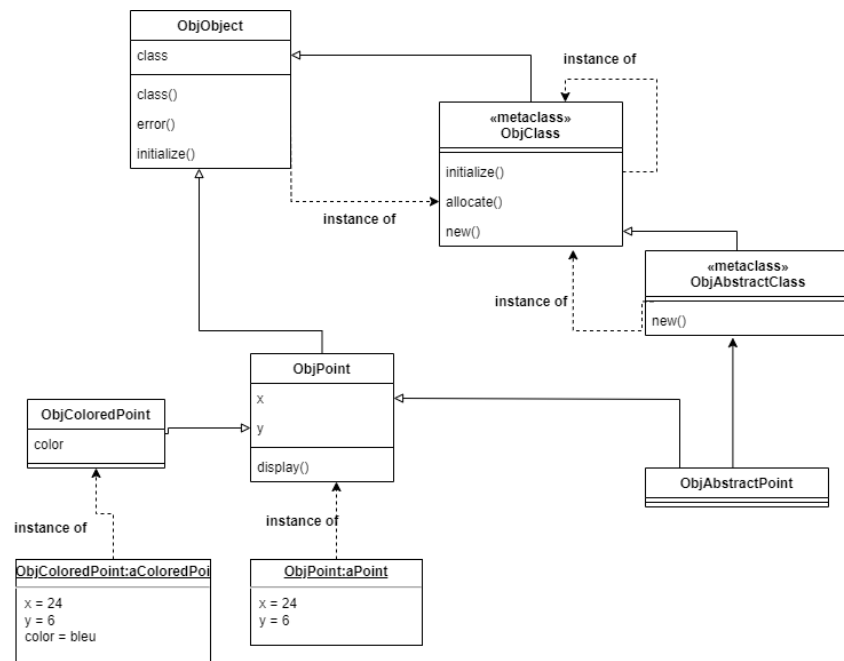


FIGURE 2.10 – Diagramme de classe represente le bootstaping

La figure 2.11 suivant represente que tout les testes de ce projet on été valider avec succes.

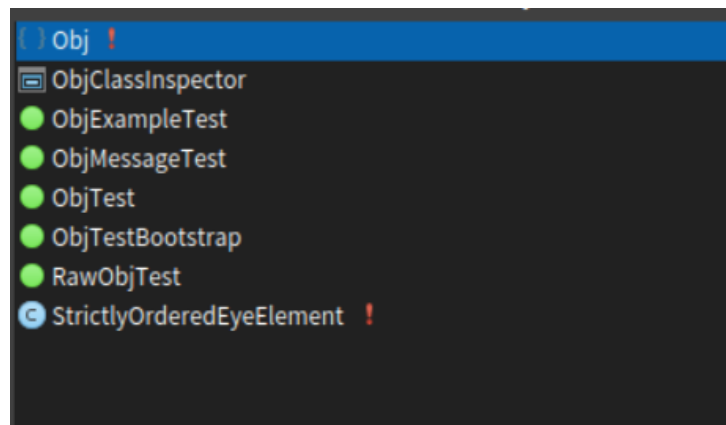


FIGURE 2.11 – Tous les tests ont marché

Chapitre 3

Conclusion

La réalisation de ce projet de construction d'un noyau minimal réflexif basé sur les classes dans le modèle ObjVLisp a été une aventure enrichissante qui a permis d'approfondir la compréhension des systèmes de programmation orientée objet. En naviguant à travers les complexités de l'implémentation, de la gestion des classes et des objets, et de la résolution dynamique des méthodes, ce projet a démontré l'élégance et l'efficacité du modèle ObjVLisp. Les défis rencontrés, notamment dans la représentation des objets, l'héritage des variables d'instance, et la gestion des envois de messages super, ont renforcé la compréhension des principes fondamentaux de la programmation orientée objet. En outre, l'utilisation de Pharo a non seulement facilité l'implémentation mais a également apporté une dimension pratique à l'apprentissage théorique. Ce projet a donc non seulement atteint son objectif de construire un noyau réflexif, mais a également ouvert la voie à de nouvelles explorations dans le domaine de la programmation orientée objet.