Pour aller plus loin : Conception préliminaire

Denis Conan

CSC4102

Télécom SudParis Janvier 2024

Table des matières

Pour aller plus loin : Conception préliminaire
Denis Conan, , Télécom SudParis, CSC4102
Janvier 2024 **1**

1 Aspects dynamiques — Diagramme de communications 3
1.1 Participant, lien d'interaction, message conditionné, messages en séquence 4
1.2 Messages emboîtés, création d'objet 5
1.3 Itération de messages, suppression d'objet 6
1.4 Choix entre séquence et communications 7

2 Compléments autour du sous-typage 8
2.1 Subsomption et liaison dynamique 9
2.2 Substitution par sous-typage, syntaxe 10
2.3 Substitution par sous-typage, sémantique 11
2.4 Substitution par sous-typage : UML et JAVA 12
2.5 Opération/méthode Polymorphique 13

Références 14

1 Aspects dynamiques — Diagramme de communications

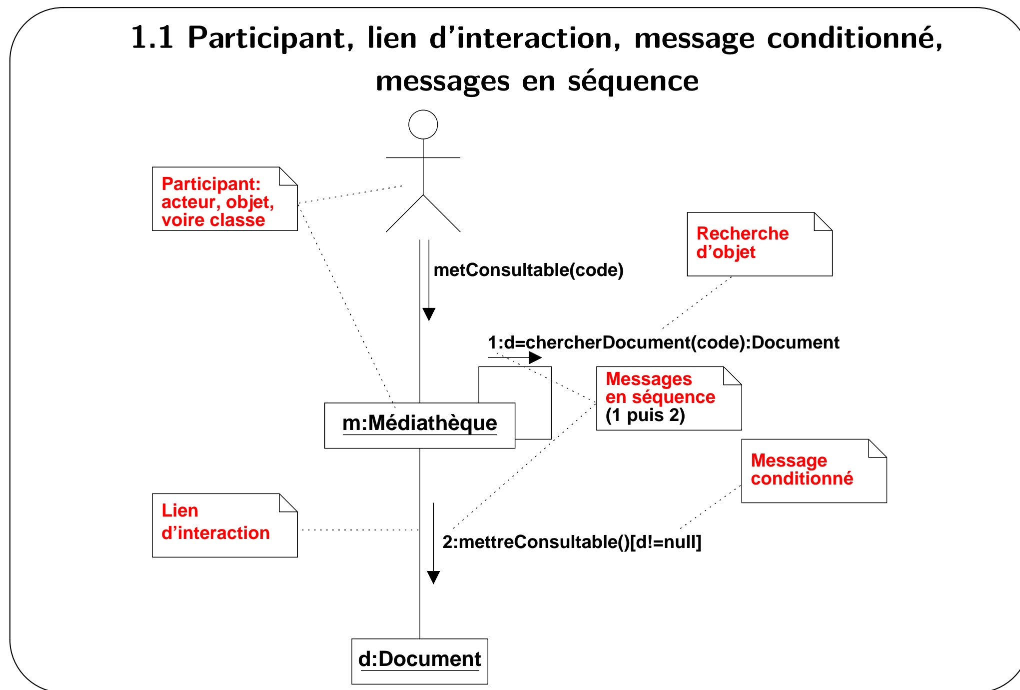
2

1.1 Participant, lien d'interaction, message conditionné, messages en séquence 3
1.2 Messages emboîtés, création d'objet 4
1.3 Itération de messages, suppression d'objet 5
1.4 Choix entre séquence et communications 6

Le diagramme de communications (comme le diagramme de séquence) est un diagramme d'interactions qui représente une vue dynamique du système. Le diagramme de communications représente les interactions entre objets en mettant moins en évidence l'aspect temporel mais en faisant ressortir les relations entre objets. Ce modèle montre les différents messages qui se propagent d'un objet à l'autre : il attire l'attention du concepteur sur le fait qu'un message transite sur une association, qui doit donc exister entre la classe de l'objet appelant et la classe de l'objet appelé. Comme dans les diagrammes de séquence, un objet doit avoir une méthode appropriée pour traiter chaque événement qu'il reçoit.

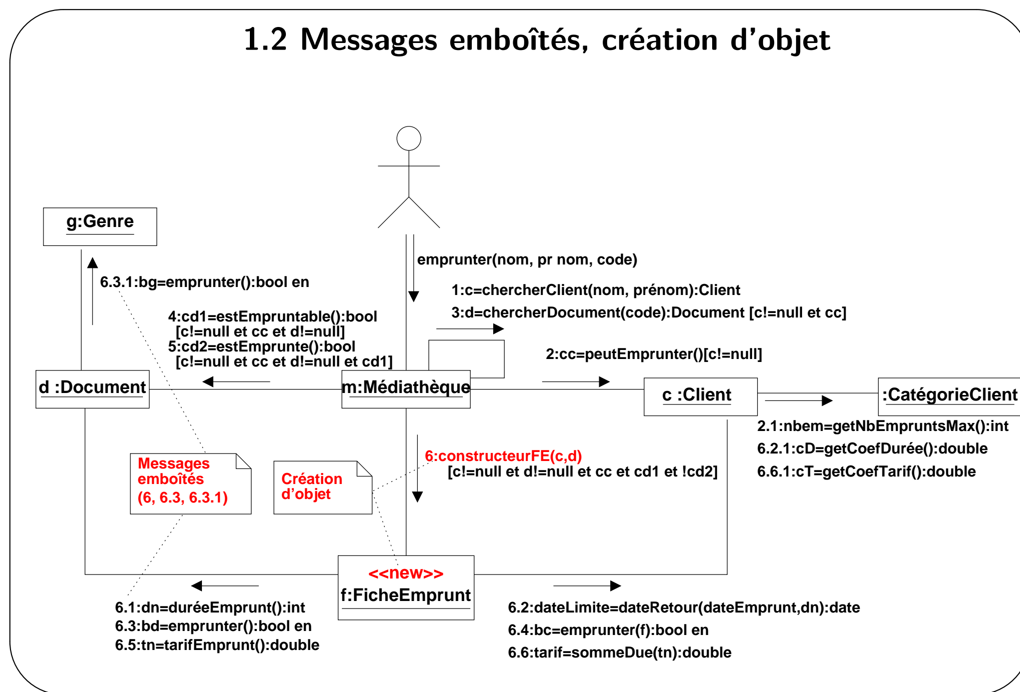
Le diagramme de communications est pratiquement équivalent au diagramme de séquence : on peut construire l'un à partir de l'autre. Le diagramme de séquence possède une approche plus temporelle et le diagramme de communication une approche plus spatiale. La dernière diapositive de cette section donne une comparaison et des critères de choix.

3



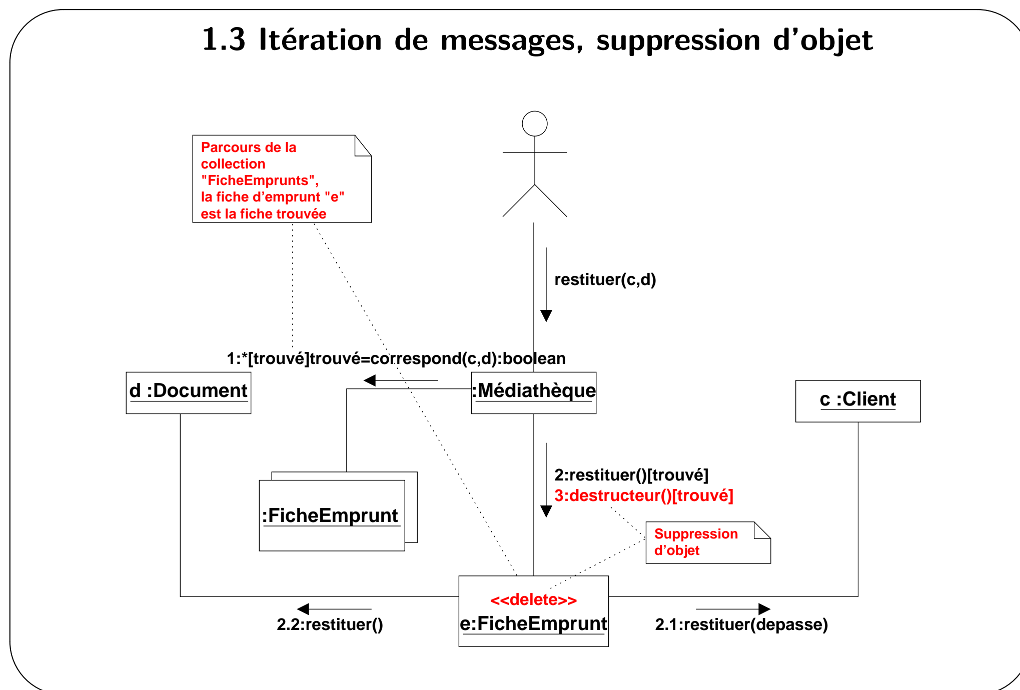
Dans cette diapositive, nous présentons, à travers l'exemple du cas d'utilisation simple « mettre consultable », la terminologie et la syntaxe du diagramme de communications. Dans le diagramme de communications, l'aspect temporel n'est pas complètement caché car chaque message est numéroté. Un diagramme de communications ne fait pas de distinction entre les différents types de messages.

4



Il y a emboîtement des messages pour marquer la relation de cause à effet (un objet ayant reçu le message 2 déclenche en réaction le message 2.1). La séquence de messages du diagramme de cette diapositive est donc : message 1 en réaction au message venant de l'acteur ; puis message 2, qui provoque le message 2.1 ; et lorsque les traitements des messages 2.1 et 2 sont terminés, envoi du message 3, etc.

5



La modélisation dans un diagramme de communications de ce qui correspond au fragment « loop » du diagramme de séquence utilise les lettres et l'étoile. Dans cette exemple, le message « 1 :*[trouvé]trouvé=correspondre(c,d) ;boolean » exprime le parcours de la collection des fiches d'emprunt avec la condition d'arrêt « trouvé » et l'appel de l'opération correspondre sur chaque instance de la collection. L'étoile au début de l'expression du message « 1 » indique l'itération. Par convention, l'indice est souvent nommé par une lettre (par exemple « i »). Cette lettre est ensuite utilisée pour nommer l'itération en cours. Ainsi, la réaction à l'appel de l'opération correspondre sur chaque instance de la collection pourrait provoquer les messages emboîtés « 1.i.1 » et« 1.i.2 ».

1.4 Choix entre séquence et communications

\# 6

- Les plus du diagramme de séquence :
  - Montrer l'ordre des interactions (le premier objectif du diagramme)
  - Montrer les messages asynchrones (différents types de messages)
  - Montrer les parties optionnelles, les itérations (concept de fragment)
- Les plus du diagramme de communications :
  - Montrer les liens entre participants (le premier objectif du diagramme)
    - Lien visuel direct avec le graphe des classes du diagramme de classes
  - Montrer les participants (l'un des objectifs du diagramme)
  - Un peu plus facile à construire et à adapter (numérotation plutôt que séquencement)
- Les deux sont équivalents pour :
  - Montrer la signature des messages / opérations

Non montré : « parallélisme » avec fragment « `par` » *Versus* numérotation avec des lettres

Cette diapositive compare les diagrammes de séquence et de communications. Ils sont très proches, mais dans des cas particuliers et selon les interlocuteurs, le choix est important. Dans le module, pour ne pas multiplier les notations, nous utilisons uniquement le diagramme de séquence.

2 Compléments autour du sous-typage

Barbara Liskov, ACM A.M. Turing Award, 2008

7

- Principe de substitution de Liskov et Wing
  [Cardelli and Wegner, 1985, Liskov, 1987, Liskov and Wing, 1994]
  - *What is wanted here is something like the following substitution property: if for each object o_1 of type S there is an object o_2 of type T such that, for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T.*
- C'est une bonne propriété recherchée lors de la conception

Les diapositives qui suivent développent le concept de substitution.
Aux curieux en culture générale sur le génie logiciel, nous proposons le visionage de la vidéo suivante :

- Barbara Liskov, "*Programming the Turing Machine*", Princeton University's Centennial Celebration of Alan Turing, Août 2012
  `http://www.youtube.com/watch?v=ibRar7sWulM`
  - Dans cet exposé d'environ 1h, B. Liskov retrace les étapes ayant amené son équipe à introduire le polymorphisme et la généricité dans les langages de programmation. B. Liskov présente quelques concepts de développement génie logiciel dans leur contexte historique et explique pourquoi chaque concept a pour objectif d'aider les développeurs. Citons par exemple la programmation structurée (sans instruction `go to`), le développement par raffinement, la modularité, l'encapsulation, la généricité et le polymorphisme. La fin de la présentation et la séance des questions/réponses discute de la question difficile du meilleur premier langage d'apprentissage de la programmation, notamment de Java, Python, et ML. Voici à titre indicatif les références des publications étudiées dans cet exposé (nous pouvons fournir une copie de chacune à la demande) :
    - E.W. Dijkstra, "*Go to Statement Considered Harmful*", Communications of the ACM, 11(3), pages 147–148, March 1968.
    - N. Wirth, "*Program Development by Stepwise Refinement*", Communications of the ACM, 14(4), pages 221–227, April 1971.
    - D.L. Parnas, "*Information Distribution Aspects of Design Methodology*", In Proceedings of the IFIP Congress, 1971.
    - B. Liskov, "*A Design Methodology for Reliable Software Systems*", In Proceedings of the Fall Joint Computer Conference, pages 191–199, Anaheim, California, December 1972.
    - O.-J. Dahl and C.A.R. Hoare, "*Hierarchical Program Structures*", In *Structured programming*, chapter 3, pages 175–220. Academic Press Ltd., 1972.
    - W. Wulf and M. Shaw. "*Global Variable Considered Harmful*", ACM SIGPLAN Notices, 8(2), pages 28–34, February 1973.
    - B. Liskov and S. Zilles, "*Programming with Abstract Data Types*, In Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages, pages 50–59, Santa Monica, California, USA, April 1974.
    - B. Liskov. "*Data Abstraction and Hierarchy*", ACM SIGPLAN Notices, 23(5), pages 17–34, January 1988.

  Nous ajoutons à la liste de ces références l'article suivant : L. Cardelli and P. Wegner, "*On Understanding Types, Data Abstraction, and Polymorphism*", ACM Computing Surveys, 17(4), pages 471–522, December 1985.

8

2.1 Subsomption et liaison dynamique

- Rappel : s est du type $T \equiv s \in T$; S est un sous-type de $T \equiv S \subseteq T$
- Subsomption [Abadi and Cardelli, 1996]
  - Soient s et t de type S et T respectivement
  - On peut écrire « $t := s$ »
    - $s \in S \wedge S \subseteq T \implies s \in T$
  - On parlera aussi de transtypage vers le haut ou *upcast*
  - $\implies$ Factorisation du comportement
- Liaison dynamique [OMG, 2013, Bloch, 2008]
  - Après « $t := s$ », T est appelé le type formel de t et S le type actuel de t
    C'est le type actuel qui est utilisé
    - Si S redéfinit l'opération *op* préalablement définie dans T
      après « $t := s$ », « $t.op()$ » utilise *op* redéfinie dans S
- Mais *op* redéfinie dans S est-elle utilisable à la place de *op* définie dans T ?
  On dit « substituer à *op* définie dans T sa redéfinition dans S »

Extraits librement adaptés de [Abadi and Cardelli, 1996, Simons, 2002a, Simons, 2002b]

« *The notion of subtyping derives ultimately from subsets in set theory. In exactly the same way that: "t is of type T" can be interpreted as* $s \in T$*. So, the subtyping relationship "S is a subtype of T" can be interpreted consistently as the subset relationship* $S \subseteq T$*.*

Consider the following code fragments, in the scope of those definitions:

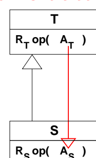
```
T t;
S s;
s = new S();
t = s;
op(s); // op is declared as void op(T)
```

Here an instance of class S is assigned to a variable holding instances of class T. Similarly, an instance of class S is passed to an operation m that expects instances of class T. Both code fragments would be illegal in a language like Pascal, because the types S and T do not match. In Pascal, a strongly-typed language, a variable can only receive a value of exactly the same type, a property known as monomorphism (literally, the same form). Furthermore, types are checked on a name equivalence, rather than structural equivalence basis. This means that, even if a programmer declared S to be synonym for T, the Pascal type system would still treat the two as non-equivalent, because of their different names.

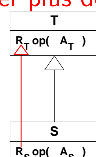
In object-oriented languages these code fragments are made legal by the characteristic property of subtype relation called subsomption: If s is an instance of type S and S is a subtype of T, then s is also an instance of type T. »

2.2 Substitution par sous-typage, syntaxe

- Soit S un sous-type de T avec une opération *op* redéfinie dans S
- Soient l'opération *op* : $A_T \rightarrow R_T$ à un seul argument de type A_T et une valeur de retour de type R_T, et l'opération *op* : $A_S \rightarrow R_S$ redéfinie dans S
- Conditions pour que op_S : $A_S \rightarrow R_S$ soit utilisable à la place de op_T : $A_T \rightarrow R_T$
  1. $A_S \supseteq A_T$: op_S doit accepter au moins autant de valeurs que op_T en accepte
     - $S \subseteq T$ et $A_S \supseteq A_T$
       direction opposée
       contravariance des arguments



  2. $R_S \subseteq R_T$: op_S ne doit pas retourner plus de valeurs que op_T en livre
     - $S \subseteq T$ et $R_S \subseteq R_T$
       même direction
       covariance de la valeur de retour



9

[Abadi and Cardelli, 1996, Simons, 2002b]
En JAVA, les méthodes redéfinies doivent avoir les mêmes types (invariance)

Extraits librement adaptés de [Simons, 2002b]

« *For a substitute object $s \in S$ to behave exactly like the original object $t \in T$, then every method of T must have a corresponding method in S that behaves like the original method of T. If we are only interested in syntactic compatibility (that is, between interfaces), this reduces to checking the type signatures of related pairs of methods. In many object-oriented languages, the correspondence between the methods of T and S is at least partly assured by defining S as an extension of T —in this case, S may inherit the majority of T's methods unchanged. However, we must cater for the general case, in which S substitutes different methods in place of those in T (known as method redefinition, or overriding).*

Consider a method op of T, which we shall call $op_T : A_T \rightarrow R_T$, to indicate that it is a function accepting an argument of some type A_X and yielding a result of some other type R_X. This is to be replaced by a substitute method op of S, which we shall call $op_S : A_S \rightarrow R_S$, with the intention that S should still behave like T. Under what conditions can op_S be safely substituted in place of op_T? From the syntactic point of view, there are two obligations:

- *op_S must be able to handle at least as many argument values as op_T could accept; we express this as a constraint on the domains (argument types): $A_S \supseteq A_T$; and*
- *op_S must deliver a result that contains no more values than the result of op_T expected; we express this as a constraint on the codomains (result types): $R_S \subseteq R_T$.*

A helpful way to think about these obligations is to consider how a program might fail if they were broken. What if op_S accepted fewer argument values than op_T? In this case, there might be some valid arguments supplied to op_T in the original working program that were not recognised by op_S after the substitution, causing a run-time exception. What if op_S delivered more result values than op_T? In this case, the call-site expecting the result of op_T might receive a value that was outside the specified range when op_S was invoked instead.

"If the domain (argument type) A_S is larger than the domain A_T, and the codomain (result type) R_S is smaller than the codomain R_T, then the function type $A_S \rightarrow R_S$ is a subtype of the function type $A_T \rightarrow R_T$." Note how there is an assymmetry in this rule: in the subtype function, the codomain is also a subtype, but the domain is a supertype. For this reason, we sometimes say that the domains are contravariant (they are ordered in the opposite direction) and the codomains are covariant (they are ordered in the same direction) with respect to the subtyping relationship between the functions.

Very few languages obey both the covariant and contravariant parts of the function subtyping rule (Trellis is one example). Languages such as JAVA and C++ are less flexible than these rules allow, in that they require replacement methods to have exactly the same types. Partly, this is due to interactions with other rules for resolving name overloading. »

2.3 Substitution par sous-typage, sémantique

10

- Sémantique définie par :
  - Les préconditions et les postconditions des opérations
  - Les invariants des instances
    - Les propriétés d'un objet qui sont toujours vérifiées pendant la vie de l'objet
- Soient les opérations $op_T : A_T \rightarrow R_T$ de T et $op_S : A_S \rightarrow R_S$ du sous-type S
- Précondition de op_S = précondition de op_T affaiblie
  c.-à-d., précondition de $op_S \implies$ précondition de op_T
- Postcondition de op_S = postcondition de op_T renforcée
  c.-à-d., postcondition de $op_S \impliedby$ postcondition de op_T
- Invariant de S = invariant de T renforcée
  c.-à-d., invariant de $S \impliedby$ invariant de T

Extraits librement adaptés de [Simons, 2003a]

« *Type compatibility is not just a matter of observing the conventions on type signatures. An object could offer all the expected operations, but still execute in a completely perverse way. It is equally important to know whether a component behaves in the way expected by the program in which it is used. For this, an approach is required which can model the semantics of object types and capture precisely how they execute. A means of incorporating semantics, that is the meaning of operations, is by specifying preconditions, postconditions together with invariants characterising the unchanging properties of object types. These preconditions, postconditions and invariants form in the following the axioms of the type.*

We already know the following. Consider defining a set S by comprehension in relation to a set T, such that S contains all those elements in T which satisfy the extra axiom $p(x)$:

- $S = \{\forall x \in T | p(x)\}$

It is clear that, if all elements of T pass the test $p(x)$, then $S = T$. However, if some elements of T fail the test, then $S \subset T$. Therefore, we can assert that $S \subseteq T$, and this also means that S is a subtype of T.

If we refine an axiom in a subtype, the subtyping condition is this: the refined axiom must logically entail the original axiom. We can then analyse the behaviour in terms of invariants, preconditions and postconditions. It is useful to think in terms of strengthening assertions:

- *Strengthening an invariant is identical to strengthening the axioms, since the invariant applies constantly to the instances of type as a whole;*
- *Strengthening a method postcondition corresponds either to strengthening the axioms on the result-type of the method, or strengthening the axioms on the instance of the type itself; or possibly to both of these;*
- *Strengthening a method precondition corresponds either to strengthening the axioms on the argument-types of the method, or strengthening the axioms on the instance of the type itself; or possibly to both of these.*

A method is a valid replacement for another if it obeys the function (syntactic) subtyping rule. In particular, the subtype method's arguments must be of the same, or more general (but not more restricted) types; and its result must be of the same, or a more restricted (but not more general) type. Translating this into assertions, the method's preconditions may possibly be weakened (but never strengthened) and the method's postconditions may possibly be strengthened (but never weakened). »

11

2.4 Substitution par sous-typage : UML et JAVA

- Syntaxe
  - UML [OMG, 2013]
    - *Different type-conformance systems adopt different schemes for how the types of parameters and results may vary when an operation is redefined in a specialization. When the type may not vary, it is called invariance. When the parameter type may be specialized in a specialized type, it is called covariance. When the parameter type may be generalized in a specialized type, it is called contravariance. In UML, such rules for type conformance are intentionally not specified. Redefined parameters shall have compatible multiplicity, and the same direction, ordering and uniqueness as the redefined parameters.*
  - JAVA [Simons, 2002b]
    - Les méthodes redéfinies doivent avoir les mêmes types
      Autrement dit, invariance des arguments et de la valeur de retour

12

- Sémantique
  - UML [OMG, 2013]
    - *The specification of a user-defined constraint is often expressed as a text string in some language, whose syntax and interpretation is as defined by that language. In some situations, a formal language (such as OCL) or a programming language (such as JAVA) may be appropriate, in other situations natural language may be used.*
  - JAVA [Simons, 2003b]
    - Pas de construction pour spécifier les préconditions, postconditions, et invariants[a]
    - Donc, des tests unitaires devront être préparés pour les vérifier

a. Un langage comme Eiffel propose de telles constructions (`https://www.eiffel.com/`)

Extraits librement adaptés de [Simons, 2003b]

« *Syntactically sound subtyping is exhibited by JAVA, although JAVA is less flexible in its redefinition rule. In JAVA, it is still possible to redefine methods to execute in arbitrary ways, resulting in unpredictable behaviour in substitutable components.*

The fact that these faults do not give rise to system crashes more often than they do is explained mostly by the fact that programmers strive to write code in a consistent way, adopting style guidelines over and above what the type systems are capable of checking. »

2.5 Opération/méthode Polymorphique

- Polymorphisme : « qui offre des apparences, des formes diverses »
- Opérations polymorphiques : opérations de même nom offrant des comportements différents
- Soient r, o et a respectivement de type R, O et A
  Variations autour de l'instruction « `r o.op(a);` »
  - Polymorphisme *ad hoc* = surcharge
    - Plusieurs opérations *op* dans le type O[a]
  - Polymorphisme d'inclusion ou de sous-type = redéfinition
    - Type formel de o est O, mais type actuel ? *op* est-elle une redéfinition ?
  - Polymorphisme paramétrique : Cf. séance 5, polymorphisme paramétrique avec les types paramétrés (*generics* en JAVA)

13

a. Ne pas oublier la subsomption (transtypage vers le haut [*upcast*]) qui intervient aussi

Références

[Abadi and Cardelli, 1996] Abadi, M. and Cardelli, L. (1996). *A Theory of Objects*. Springer-Verlag.

[Bloch, 2008] Bloch, J. (2008). *Effective Java, 2nd Edition*. Addison-Wesley.

[Cardelli and Wegner, 1985] Cardelli, L. and Wegner, P. (1985). On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4) :471–523.

[Liskov, 1987] Liskov, B. (1987). Keynote Address — Data Abstraction and Hierarchy. In *Addendum to the Proceedings of the ACM conference on Object-oriented Programming Systems, Languages and Applications*, pages 17–34, Orlando, Florida, USA.

[Liskov and Wing, 1994] Liskov, B. and Wing, J. (1994). A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6) :1811–1841.

[OMG, 2013] OMG (2013). OMG Unified Modeling Language, Version 2.5. Specification number formal/2013-09-05.

[Simons, 2002a] Simons, A. (2002a). The Theory of Classification, Part 1 : Perspectives on Type Compatibility. *Journal of Object Technology*, 1(1) :55–61.

[Simons, 2002b] Simons, A. (2002b). The Theory of Classification, Part 4 : Object Types and Subtyping. *Journal of Object Technology*, 1(5) :27–35.

[Simons, 2003a] Simons, A. (2003a). The Theory of Classification, Part 5 : Axioms, Assertions and Subtyping. *Journal of Object Technology*, 2(1) :13–21.

[Simons, 2003b] Simons, A. (2003b). The Theory of Classification, Part 6 : The Subtyping Inquisition. *Journal of Object Technology*, 2(2) :17–26.