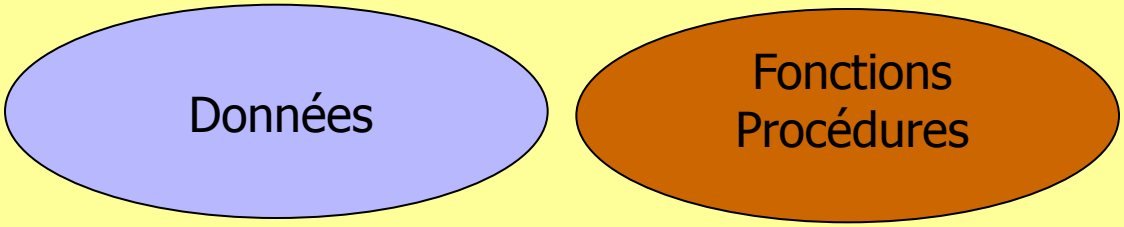
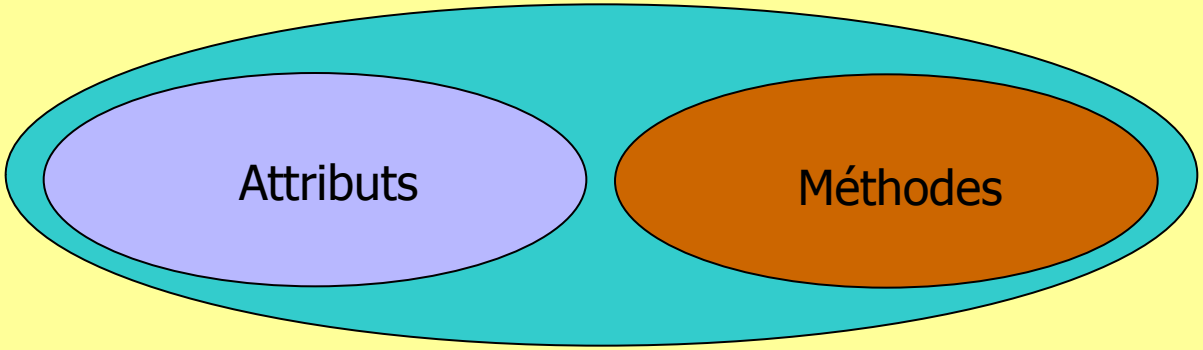
Spécifications axiomatiques des types abstraits

Les trois approches

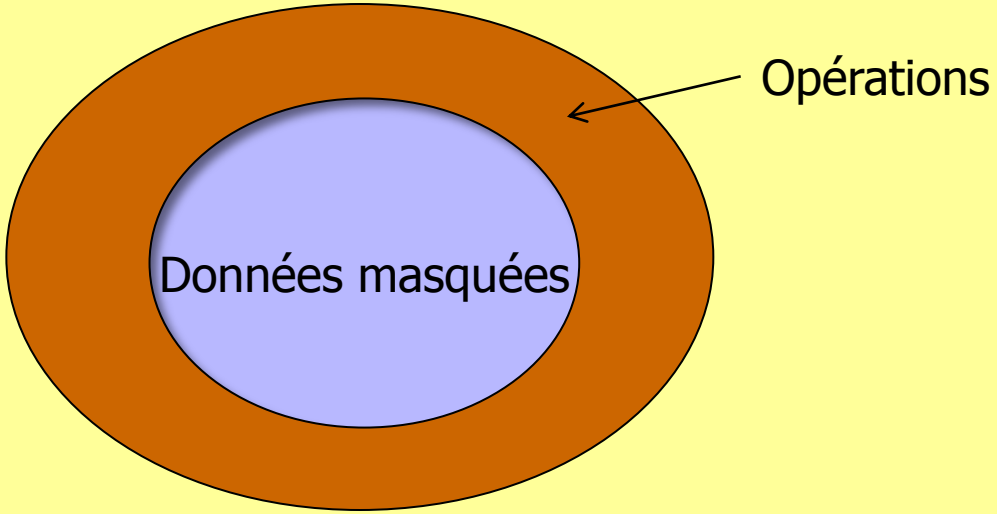
Approche procédurale



Approche Objet



Approche Type abstrait



iinformations

traitements

Les données masquées

- Avec les types abstraits on ne manipule un objet qu'à travers les opérations de base ou d'extension.
- On ne parle donc jamais de l'implémentation de l'objet en mémoire vive.
- Le programmeur a donc le choix de l'implémentation pour les structures de données à condition de respecter la définition des opérations

Les opérations d'un type abstrait

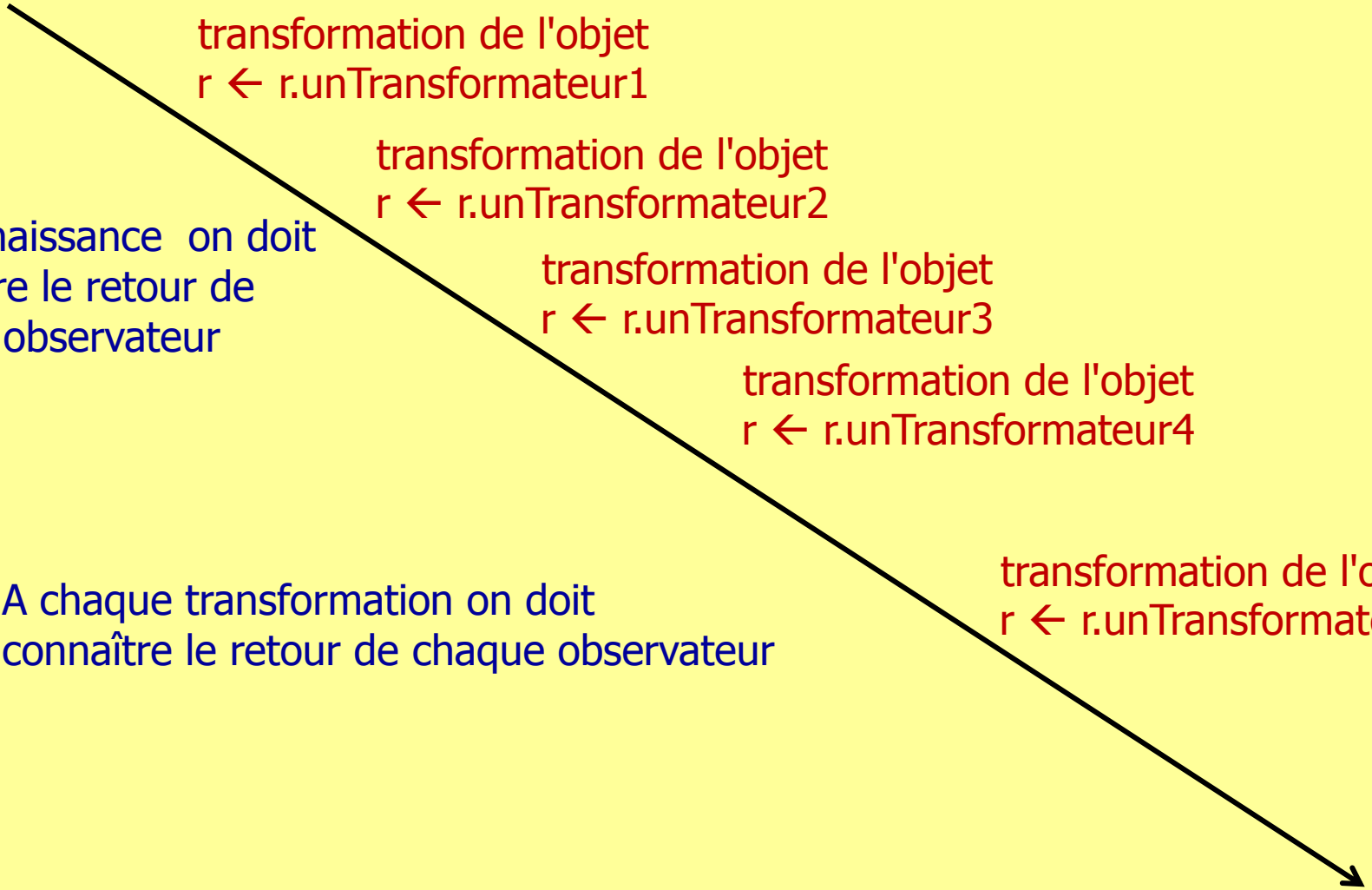
- Une opération de base est une opération dont l'exécution dépend des données masquées
- Une opération d'extension est une opération dont l'exécution ne dépend que de l'exécution d'opérations de base et/ou d'opérations d'extension.

Les opération de base

- La spécification d'une opération de base est définie à l'aide d'axiomes et de préconditions.
- Un axiome est une expression booléenne admise comme vraie sans démonstration.

La vie d'un objet

naissance de l'objet
r ← monConstructeur



transformation de l'objet
r ← r.unTransformateur1

transformation de l'objet
r ← r.unTransformateur2

transformation de l'objet
r ← r.unTransformateur3

transformation de l'objet
r ← r.unTransformateur4

transformation de l'objet
r ← r.unTransformateur5

Dès la naissance on doit connaître le retour de chaque observateur

A chaque transformation on doit connaître le retour de chaque observateur

Restriction aux opérations de base

- Une opération d'extension se définit à l'aide d'opérations de base et d'opérations d'extension.
- Par récurrence on peut donc montrer que la définition d'une opération d'extension ne dépend que de la définition d'opérations de base.
- Un type abstrait est donc défini sans ambiguïté si les opérations de base sont définies sans ambiguïtés.

Complétude suffisante

- Le graphique de vie d'un objet montre qu'un type abstrait est entièrement défini si à tout instant de la vie d'un objet les axiomes nous indiquent sans ambiguïté le retour de tout observateur
- Nous allons donc associer à chaque cas un axiome qui définit le retour de l'opération.

Complétude suffisante

- Pour chaque constructeur cons et pour chaque obervateur obse, un axiome doit définir le retour de l'appel cons().obse().
- Pour chaque objet o, chaque transformateur trans et pour chaque observateur obse, un axiome doit définir le retour de l'appel o.trans().obse().
- Pour chaque objet o, chaque couple de transformateur (trans1,trans2), un axiome doit définir le retour de l'appel o.trans1().trans2().

Exemple trivial : Pas de transformateurs

TYPE ABSTRAIT Terme

Opérations de base

Constructeur Terme : creerTermeOperande(REEL operande) : Terme

Constructeur Terme : creerTermeOperateur(CHAINE operateur) : Terme

Observateur Terme : estOperateur() : BOOLEEN

Observateur Terme : recOperateur() : CHAINE

Observateur Terme : recOperande() : REEL

Pré-conditions

definie(t.recOperateur()) <==> t.estOperateur()

definie(t.recOperande()) <==> Non t.estOperateur()

Axiomes

creerTermeOperateur(operateur).estOperateur()

non creerTermeOperande(operande).estOperateur()

creerTermeOperande(operande).recOperande() = operande

creerTermeOperateur(operateur).recOperateur() = operateur

Exemple moins simple: Vecteur

TYPE ABSTRAIT Vecteur

Opérations de base

Constructeur Vecteur : creerVecteur(ENTIER bi, ENTIER bs) : Vecteur

Transformateur Vecteur : affVal(ENTIER i,Element e) : Vecteur

Observateur Vecteur : recVal(ENTIER i) : ENTIER

Observateur Vecteur : estInitialise(ENTIER i) : BOOLEEN

Observateur Vecteur : borneInf() : ENTIER

Observateur Vecteur : borneSup() : ENTIER

Axiomes

- creerVecteur(bi, bs).borneInf() = bi
- creerVecteur(bi, bs).borneSup() = bs
- i <= bs ET i >= bi => NON creerVecteur(bi, bs).estInitialise(i)
- i <= v.borneSup() ET i >= v.borneInf() => v.affVal(i,e).recVal(i) = e
- i <= v.borneSup() ET i >= v.borneInf() => v.affVal(i,e).estInitialise(i)

Préconditions

définie(v.estInitialise(i)) $\Leftrightarrow$ i <= v.borneSup() ET i >= v.borneInf()

définie(v.recVal(i)) $\Leftrightarrow$ v.estInitialise(i)

définie(v.affVal(i,e)) $\Leftrightarrow$ i <= v.borneSup() ET i >= v.borneInf()

Exemple compliqué: Pile

TYPE ABSTRAIT Pile

Opérations de base

Constructeur Pile : pileVide() : Pile

Transformateur Pile : empiler(Element e) : Pile

Transformateur Pile : depiler() : Pile

Observateur Pile : sommet() : Element

Observateur Pile : estVide() : Entier

Axiomes

- pileVide().estVide()
- p.empiler(e).sommet() = e
- non p.empiler(e).estVide()
- p.empiler(e).depiler() = av(p)
- p1 = copie(p) ⇒ p.depiler().empiler(p1).sommet()

Préconditions

- définie(p.sommet()) ⇔ non p.estVide()
- définie(p.depiler()) ⇔ non p.estVide()

cette solution n'est pas exhaustive