Simple Unit
2.0-a0

Samuel Andrés

11 octobre 2025

Table des matières

1 Versionnement et recommandations **1**

2 Spécification abstraite **2**
2.1 Périmètre 2
2.2 Convertisseurs 3
2.3 Types d'unités 4
2.4 Facteurs 4
2.5 Méthodes de construction 4
2.6 Transformeurs 6

3 Spécification d'implémentation **6**
3.1 Construction des convertisseurs 7
3.2 Convertisseurs vers les unités fondamentales 7
3.2.1 Depuis une unité fondamentale 7
3.2.2 Depuis une unité transformée 8
3.2.3 Depuis une unité dérivée 8

4 Validation **8**

5 Versionnement **10**

1 Versionnement et recommandations

Le numéro de version de ce document est composé d'un numéro de version majeure, d'un numéro de version mineure et d'un numéro de révision. Ces numéros sont agencés comme suit, concernant les versions stables :

<version majeure>.<version mineure>-r<révision>

Et de la manière suivante pour les versions en cours de préparation :

<version majeure>.<version mineure>-a<révision>

Les versions en cours de préparation sont publiées à titre indicatif et ne doivent pas être considérées comme *a priori* représentatives de la version définitive. Les numéros de version majeure et mineure qui les composent sont considérés comme n'étant pas encore atteints dans leur version définitive. Ils ne peuvent donc servir de référence à des implémentations qui reprendraient leur numérotation. Enfin, il n'est pas garanti qu'une version stable utilisera

les mêmes numéros de version majeure et mineure qu'une version en cours de préparation.

Une sémantique particulière est attachée aux composantes d'un numéro de version stable.

Un changement de numéro de version majeure intervient lors d'une modification du contrat cassant la rétrocompatibilité avec les versions précédentes.

Un changement de numéro de version mineure intervient lors d'une modification du contrat qui ne casse pas la rétrocompatibilité.

Un changement de numéro de version de révison intervient à tout autre modification de ce document sans effet sur le contrat *ou* relevant de corrections d'erreurs flagrantes.

Les changements de version majeure, mineure et de révision interviennent chacune par incrémentation d'une valeur entière. Le numéro de révision est réinitialisé à 0 lors d'un changement de version mineure ou majeure. Le numéro de version mineure est réinitialisé à 0 lors d'un changement de version majeure.

On *recommande* que les implémentations se référant à cette spécification suivent elles-mêmes un versionnement à trois valeurs (majeure, mineure et révision) les versions majeure et mineure se référant respectivement aux versions majeure et mineure de la présente spécification et la version de révision étant réservée aux correctifs internes de l'implémentation ou aux modifications qui ne relèvent pas de critères impératifs de la présente spécification (par exemple, si une implémentation simplement *suggérée* par la présente spécification vient à disparaître d'une implémentation, quoique cela crée ainsi une rupture de rétrocompatibilité). On *recommande* aussi dans ce cas que l'implémentation indique de manière explicite que son numéro de version se réfère au numéro de version de la présente spécification. Dans le cas contraire, toute implémentation se référant à la présente spécification sans s'aligner sur ses numéros de version majeure et de version mineure *doit* indiquer explicitement le numéro de version complet de la spécification.

2 Spécification abstraite

2.1 Périmètre

La présente spécification abstraite s'inspire de la JSR 363 de la communauté Java. Elle en restreint pour une part le périmètre aux conversions affines entre unités et adopte une représentation conceptuelle un peu différente, quoique reposant sur le même principe de représentation des unités en un graphe d'objets permettant de calculer dynamiquement la formule d'un convertisseur d'une unité à l'autre.

D'autre part, cette spécification ajoute aux concepts inspirés de la JSR la notion de *transformation* et la distingue des opérations de *conversion* entre unités.

Une opération de *conversion* est un calcul purement *mathématique* entre deux unités de même dimension physique.

Une opération de *transformation* représente une formule physique. Les unités impliquées en entrée et en sortie peuvent être identiques, par exemple, pour transformer une pression d'un constituant donné à une température de référence en une pression du même constituan à une température donnée). Les unités

impliquées en entrée et en sortie peuvent aussi être différentes et de dimensions différentes (transformer une tension en courant à résistance donnée).

Ces opérations de *transformation* se donnent pour premier objectif d'attirer l'attention des utilisateurs sur le fait que des opérations parfois traitées comme des conversions d'unités relèvent en fait d'une nature différente, non pas purement mathématique, mais physique. Le traitement des formules physiques comme de simples conversions mathématiques d'une unité à l'autre provoque des erreurs de conception, de manipulation et de compréhension des unités de mesure, masquant en particulier des changements de dimension.

Cette spécification est destinée à guider l'implémentation de bibliothèques de construction et de conversion d'unités et non pas àfournir des collections d'unités courantes déjà construites. Il ne s'agit pas non plus d'une bibliothèque d'interprétation de chaines de caractères comme unités ni d'écriture d'unités sous forme de chaînes de caractères.

En définissant les unes par rapport aux autres les unités qui lui sont nécessaires, la présente spécification vise à permettre à l'utilisateur d'une implémentation de s'affranchir de la question rébarbative de l'implémentation des convertisseurs d'unité à unité.

En adoptant un périmètre limité, la présente spécification s'assigne également l'objectif de s'abstraire des spécificités des langages de programmation de manière à ce que la représentation conceptuelle qu'elle propose puisse être aisément implémentée en plusieurs d'entre eux en y consacrant un minimum d'effort. Toutefois, la spécification est explicitement orientée objet.

La présente spécification n'inclut pas les contrôles d'homogénéité de conversion d'unités ni de cohérence des dimension ni des grandeurs. La cohérence des conversions est laissée à l'utilisateur.

On se restreint à la représentation des conversions et des unités communes de manière à n'implémenter que des cas relativement simples de conversions affines ou linéaires.

La conceptualisation des bases, des systèmes d'unités et des considérations relatives aux mesures n'entrent pas dans le champ de cette spécification.

Sauf mention contraire relative aux éléments optionnels ou recommandés, le respect de la spécification abstraite est impératif.

Dans les diagrammes de cette section, les méthodes en italique sont optionnelles. Leur implémentation est *recommandée* pour qui souhaite s'aligner sur la spécification d'implémentation qui les utilise.

2.2 Convertisseurs

Un convertisseur représente une conversion affine ou linéaire d'une unité à une autre. La méthode *convert()* prend en paramètre une valeur exprimée dans l'unité source et retourne son expression dans l'unité cible. L'unité source est celle à partir de laquelle s'effectue l'appel de la méthode de création de convertisseur, l'unité cible étant passée en paramètre à cette méthode.

Les méthodes *scale()* et *offset()* retournent respectivement la pente et l'ordonnée à l'origine de la formule de conversion affine et la méthode *inverse()* retourne le convertisseur inverse de l'unité cible vers l'unité source du convertisseur.

On *recommande* qu'un convertisseur soit identique, au sens de l'instance, au convertisseur inverse de son convertisseur inverse.

La méthode *linear()* est optionnelle. Elle construit un convertisseur qui ne garde que la partie linéaire du convertisseur sur lequel la méthode est appelée. Lorsque cette méthode est implémentée, on *recommande* que si le convertisseur sur lequel elle est appelé est déjà linéaire, alors la méthode renvoie ce convertisseur lui-même au sens de l'instance.

La méthode *linearPow()* est optionnelle. Elle construit un convertisseur qui ne garde que la partie linéaire du convertisseur sur lequel la méthode est appelé, mais en l'élevant à la puissance indiquée en paramètre. Lorsque cette méthode est implémentée, on *recommande* que si le convertisseur sur lequel elle est appelée est déjà linéaire *et* que la puissance indiquée en paramètre est égale à 1, alors la méthode renvoie ce convertisseur lui-même au sens de l'instance.

La méthode *concatenateTo()* produit un convertisseur dont la méthode *convert()* est équivalente à l'évaluation en premier lieu de la méthode *convert()* du convertisseur indiqué en paramètre et en second lieu de la méthode *convert()* du convertisseur d'appel sur la valeur qui en résulte.

UnitConverter
+ scale() : decimal + offset() : decimal + inverse() : UnitConverter *+ linear() : UnitConverter* *+ linearPow(pow : decimal) : UnitConverter* + convert(value : decimal) : decimal + concatenateTo(converter : UnitConverter) : UnitConverter

Figure 1 – Contrat du convertisseur d'unités.

On *recommande* que les implémentations fournissent des convertisseurs immuables et indiquent le cas échéant cette caractéristique de manière explicite.

2.3 Types d'unités

Le type abstrait *Unit* représente une unité de manière générale. Les unités concrètes sont de trois types : les unités fondamentales (*FundamentalUnit*) ne sont définies à partir d'aucune autre unité. Les unités transformées (*TransformedUnit*) sont définies par une unité de référence et un convertisseur qui représente la formule linéaire ou affine qui permet de passer de l'unité transformée à son unité de référence. Les unités dérivées (*DerivedUnit*) sont définies par une collection d'unités chacune élevée à une puissance rationnelle, l'association d'une unité et d'une puissance rationnelle constitutant un facteur (*Factor*) de l'unité dérivée. Il n'est pas requis que les facteurs soient eux-mêmes des unités.

Les unités transformées et dérivées sont ainsi définies à partir d'autres unités et donc, directement ou indirectement, de manière ultime à partir d'un jeu d'unités qui sont toutes fondamentales.

Toute unité est capable de fournir d'une part avec la méthode *getConverterTo()*, un convertisseur vers une unité cible (sous réserve, dont l'appréciation est laissée à l'utilisateur, de la cohérence entre l'unité de départ et l'unité cible du point de vue des unités fondamentales qui la composent) et d'autre part avec

la méthode *toBase()*, un convertisseur vers le jeu d'unités fondamentales qui la composent.

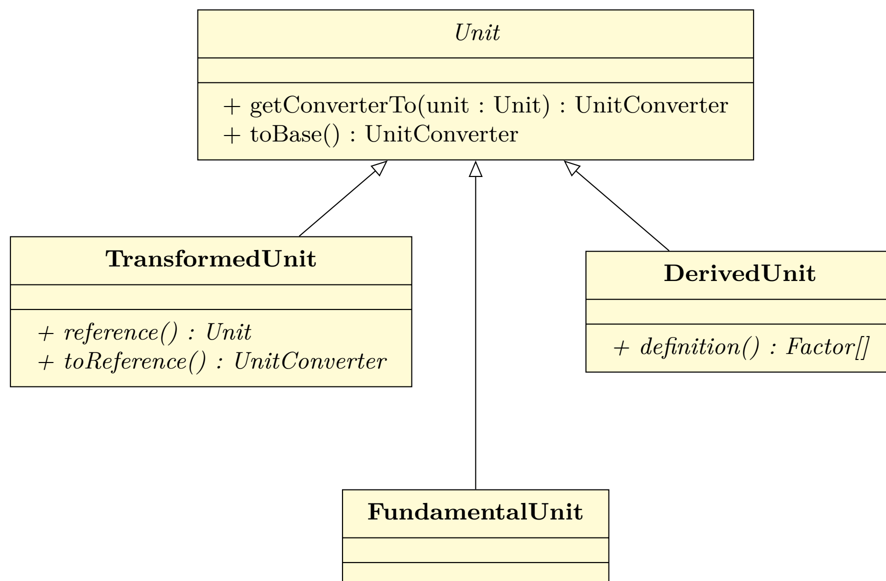

Figure 2 – Contrats des différents types d'unités.

Les unités sont des instances immuables (impératif).

2.4 Facteurs

Un facteur représente l'association d'une unité à un exposant rationnel. La combinaison des facteurs permet de construire les unités dérivées.

On *suggère* que le concept d'unité spécialise celui de facteur de manière à considérer par polymorphisme qu'une unité est un facteur d'elle-même élevée à la puissance 1. Cette représentation permet d'éviter la construction de facteurs d'unités à la puissance 1 dans la définition de nombreuses unités dérivées courantes.

2.5 Méthodes de construction

On *suggère* l'implémentation de méthodes de construction d'unités filles à partir d'une unité parente à la manière des méthodes spécifiées par la JSR 363, parmi lesquelles les méthodes utilitaires suivantes :

- La méthode *shift()* construit une unité transformée dont l'unité de référence est l'unité d'appel de la méthode et le convertisseur à l'unité de référence est une translation dont la valeur est indiquée en paramètre.
- La méthode *scaleMultiply()* construit une unité transformée dont l'unité de référence est l'unité d'appel de la méthode et le convertisseur à l'unité de référence est un agrandissement dont la valeur est indiquée en paramètre.
- La méthode *scaleDivide()* construit une unité transformée correspondant à un appel à *scaleMultiply()* avec un argument donc la valeur est inversée.

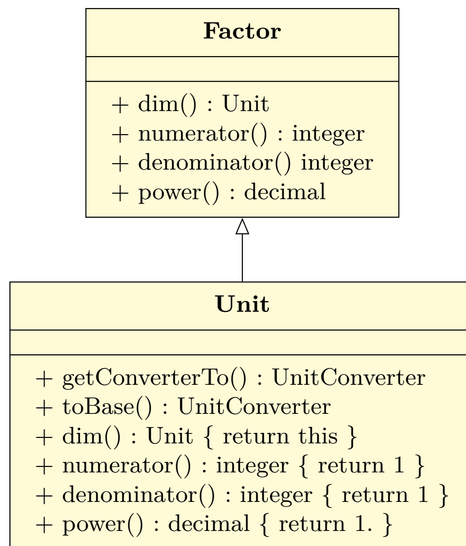

FIGURE 3 – Contrats du facteur et spécialisation suggérée par l'unité.

— La méthode *factor()* construit un facteur dont l'unité est l'unité d'appel et la puissance est un rationnel tel que le numérateur corresponde au premier paramètre de la méthode et le dénominateur au second.

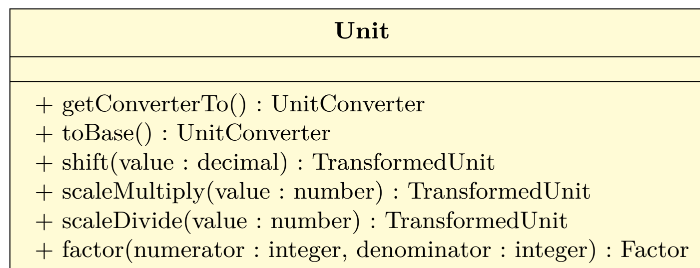

FIGURE 4 – Quelques méthodes de construction suggérées pour l'unité.

2.6 Transformeurs

Un transformeur (*UnitTransformer*) représente l'expression d'une valeur dans une unité cible à partir d'une valeur exprimée dans une unité source dans un contexte physique déterminé. Cette situation physique est caractérisée par une formule (*UnitTransformFormula*).

La formule elle-même implémente le contrat d'un transformeur, se retournant lui-même comme formule. Elle définit en plus, les méthodes *source()* et *target()* qui exposent respectivement les unités source et cible dans lesquelles les valeurs d'entrée et de sortie de la formule doivent être interprétées pour garantir un résultat correct.

Une formule dispose également d'une méthode *transformer()* permettant de construire un transformeur d'unités dérivé de la formule à partir d'une unité

source différente et vers une unité cible différente.

Enfin, il est possible de concaténer les formules au moyen de la méthode *concatenateTo()*, laquelle produit une formule appliquant en premier lieu les opérations correspondant à la formule passée en paramètre, suivies des opérations correspondant à la formule d'appel. L'unité source de la formule en résultant est l'unité source de la formule passée en paramètre et son unité cible est l'unité cible de la formule d'appel. Il importe que veiller à l'homogénéïté de l'unité cible de la formule passée en paramètre avec l'unité source de la formule d'appel afin que la concaténation puisse le cas échéant gérer les opérations de conversions en interne de l'une à l'autre.

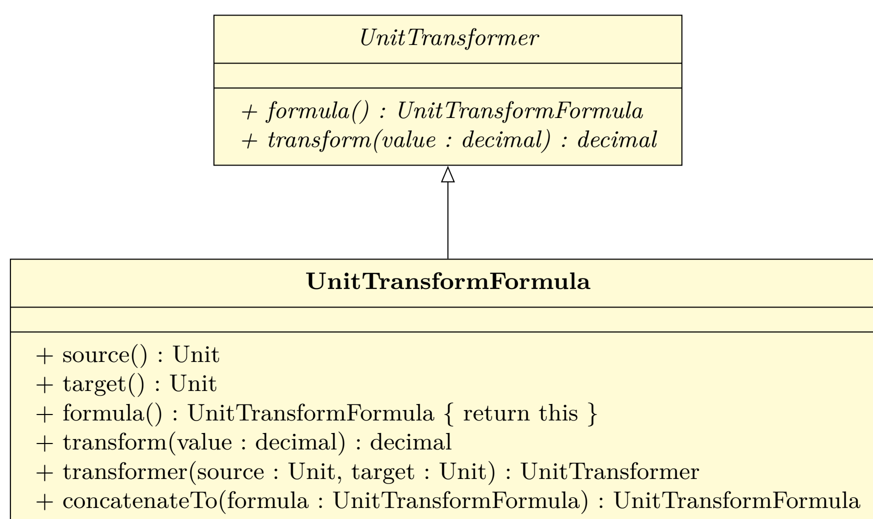

Figure 5 – Contrat du transformeur d'unités et de sa formule de référence.

3 Spécification d'implémentation

La spécification d'implémentation est indicative. La spécification dans son ensemble peut être respectée par une implémentation sans qu'elle s'aligne pour autant sur les choix d'implémentation exposés par cette section.

3.1 Construction des convertisseurs

Les convertisseurs sont construits par concaténation de la conversion de l'unité de départ vers les unités fondamentales qui la composent et de la conversion inverse de l'unité cible vers ces mêmes unités fondamentales (Algorithme 1).

Algorithm 1 Implémentation de Unit.getConverterTo()

```
procedure Unit.getConverterTo(target)          ▷ convertisseur de this à target
    return target.toBase().inverse().concatenateTo(this.toBase())
end procedure
```

Pour cela, chaque type d'unité doit être capable de définir la conversion par laquelle exprimer une valeur dans les unités fondamentales qui la composent en implémentant la méthode *toBase()* de manière spécifique chacune à sa nature.

3.2 Convertisseurs vers les unités fondamentales

3.2.1 Depuis une unité fondamentale

La conversion d'une unité fondamentale vers le jeu d'unités fondamentales qui la constitue est l'identité. On *recommande* d'implémenter cette identité comme un convertisseur singleton.

Algorithm 2 Implémentation de FundamentalUnit.toBase()

```
procedure FundamentalUnit.toBase
    return identity()        ▷ un UnitConverter correspondant à l'identité.
end procedure
```

3.2.2 Depuis une unité transformée

La conversion d'une unité transformée vers le jeu d'unités fondamentales qui la composent consite à concaténer la conversion vers son unité de référence à la conversion vers le jeu d'unités fondamentales qui composent cette dernière.

Algorithm 3 Implémentation de TransformedUnit.toBase()

```
procedure TransformedUnit.toBase
    return this.reference().toBase().concatenateTo(this.toReference())
end procedure
```

3.2.3 Depuis une unité dérivée

Enfin, la conversion d'une unité dérivée vers le jeu d'unités fondamentales qui la composent consiste à parcourir les facteurs de définition de l'unité, d'en extraire la conversion vers le jeu d'unités fondamentales de l'unité du facteur, de n'en garder que la part linéaire et de l'élever à la puissance du facteur. Chaque facteur fournissant ainsi un convertisseur vers son jeu d'unités fondamentales, tous sont concaténés de manière à produire un convertisseur global.

L'algorithme ne garde que la partie linéaire des convertisseurs résultants des unités référencées par chaque facteur car on considère que les translations doivent disparaître ainsi qu'on s'y attend, par exemple en convertissant des degrés Celcius par mètre en Kelvin par mètre.

4 Validation

La conformité aux critères de validation est impérative au respect de la spécification. On ne donne cependant pas de critère relatif à la précision des conversions laissée à une appréciation raisonnable.

Algorithm 4 Implémentation de DerivedUnit.toBase()

```
procedure DerivedUnit.toBase
    result : UnitConverter ← identity()
    for factor ∈ this.definition() do
        toFactorBase ← factor.dim().toBase().linearPow(factor.power())
        result ← toFactorBase.concatenateTo(result)
    end for
    return result
end procedure
```

Algorithm 5 Test d'unités transformées

```
procedure TransformedTest
    m : Unit ← FundamentalUnit()
    km : Unit ← m.scaleMultiply(1000)
    cm : Unit ← m.scaleDivide(100)
    cmToKm : UnitConverter ← cm.getConverterTo(km)
Require : cmToKm.convert(3) = 0.00003
Require : cmToKm.inverse().convert(0.00003) = 3
end procedure
```

Algorithm 6 Test d'unités dérivées

```
procedure DerivedTest
    m : Unit ← FundamentalUnit()
    km : Unit ← m.scaleMultiply(1000)
    km2 : Unit ← DerivedUnit(km.factor(2))
    cm : Unit ← m.scaleDivide(100)
    cm : Unit ← DerivedUnit(cm.factor(2))
    km2ToCm2 : UnitConverter ← km2.getConverterTo(cm2)
Require : km2ToCm2.convert(3.) = 30000000000
Require : km2ToCm2.inverse().convert(30000000000.) = 3
end procedure
```

5 Versionnement

Versionnement		
Version	Date	modification
2.0-a0	11 octobre 2025	fonctionnalité d'API : transformeurs d'unités, modifications du versionnement, modification du nom de la concaténation des convertisseurs
1.0-r2	15 novembre 2024	correctif : fixation de la date et ajout du tableau de versionnement
1.0-r1	20 juillet 2024	correctif : structuration de la section sur les convertisseurs
1.0-r0	28 avril 2024	version initiale
0.1-r0	18 novembre 2022	version de travail

Algorithm 7 Test d'unités dérivées (dimensions combinées)

```
procedure CombinedDerivedTest
    m : Unit ← FundamentalUnit()
    kg : Unit ← FundamentalUnit()
    g : Unit ← kg.scaleDivide(1000)
    ton : Unit ← kg.scaleMultiply(1000)
    gPerM2 : Unit ← DerivedUnit(g, m.factor(-2))
    km : Unit ← m.scaleMultiply(1000)
    tonPerKm2 : Unit ← DerivedUnit(ton, km.factor(-2))
    cm : Unit ← m.scaleDivide(100)
    tonPerCm2 : Unit ← DerivedUnit(ton, cm.factor(-2))
    toTonPerKm2 : UnitConverter ← gPerM2.getConverterTo(tonPerKm2)
    toTonPerCm2 : UnitConverter ← gPerM2.getConverterTo(tonPerCm2)
Require : toTonPerKm2.convert(1) = 1
Require : toTonPerKm2.inverse().convert(3) = 3
Require : toTonPerCm2.convert(1) = 1E-10
Require : toTonPerCm2.convert(3) = 3E-10
Require : toTonPerCm2.offset() = 0
Require : toTonPerCm2.scale() = 1E-10
Require : toTonPerCm2.inverse().offset() = 0
Require : toTonPerCm2.inverse().convert(3E-10) = 3
end procedure
```

Algorithm 8 Test d'unités dérivées (transformations affines)

```
procedure AffineDerivedTest
    k : Unit ← FundamentalUnit()
    c : Unit ← k.shift(273.15)
    m : Unit ← FundamentalUnit()
    cPerM : Unit ← DerivedUnit(c, m.factor(-1))
    kPerM : Unit ← DerivedUnit(k, m.factor(-1))
    kToC : UnitConverter ← k.getConverterTo(c)
    kPerMToCPerM : UnitConverter ← kPerM.getConverterTo(cPerM)
Require : gPerM2ToTonPerKm2.convert(1) = 1
Require : gPerM2ToTonPerKm2.inverse().convert(3) = 3
Require : kPerMToCPerM.convert(3) = 3
Require : kPerMToCPerM.inverse().convert(3) = 3
end procedure
```

Algorithm 9 Test des transformations d'unités

```
procedure UnitTransformerTest
    copper : UnitTransformFormula ← UnitTransformFormula(source=Volume.CM3, target=Mass.G, kernel=x : x * 8.94)
Require : copper.transform(1) = 8.94
    copperM3ToG : UnitTransformer = copper.transformer(source=Volume.M3, target=Mass.G)
Require : copperM3ToG.transform(1) = 8.94e6
    copperCm3ToKg = copper.transformer(source=Volume.CM3, target=Mass.KG)
Require : copperCm3ToKg.transform(1) = 8.94e-3
    copperM3ToKg = copper.transformer(source=Volume.M3, target=Mass.KG)
Require : copperM3ToKg.transform(1) = 8.94e3
end procedure
```