



«UN PEU TOUT ÇA»

L'AUTRE RAISON POUR LAQUELLE QU'HÉRITAGE PUBLIC ET VALEURS NE FONT PAS
BON MÉNAGE

25 MAI 2021, LUC HERMITTE



Sommaire

- 1 Avant-propos
- 2 La raison technique
- 3 La raison conceptuelle
- 4 Conclusion
- 5 Références



Plan partiel

① Avant-propos

② La raison technique

③ La raison conceptuelle

④ Conclusion

⑤ Références



Avant-propos

Le sujet



FredTingaud 03/05/2021

question bête @Imghs , ça ne te dirait pas de faire une vidéo de ~30 minutes sur le sujet pour le Meetup, par hasard ?



4



Imghs 03/05/2021

Le lequel de sujet? LSP? Sémantiques? PpC?



FredTingaud 03/05/2021

Un peu tout ça



L'autre raison pour laquelle héritage public et valeurs ne font pas bon ménage.



L'autre raison pour laquelle héritage public et valeurs ne font pas bon ménage.

➤ Comment ça «autre» ?



L'autre raison pour laquelle héritage public et valeurs ne font pas bon ménage.

- Comment ça «autre» ?
- Depuis quand ils ne font pas bon ménage ?



Plan partiel

① Avant-propos

② La raison technique

Rappels sur les sémantiques

Le slicing

③ La raison conceptuelle

④ Conclusion

⑤ Références



Rappel sur les sémantiques

Pour ceux qui sont passés à côté

Sur les forums francophones on ne manquera pas de vous rappeler que :

➡ hiérarchie polymorphe \Rightarrow entité \Rightarrow interdiction de copier



Rappel sur les sémantiques

Aux origines de notre schisme terminologique on retrouve

Objects can be characterised with respect to identity, state and behaviour. However, the relative significance of each of these properties varies between objects, as the following stereotypical object categories illustrate :

- Entity** *Entities express system information, typically of a persistent nature. **Identity** is important in **distinguishing** entity objects from one another.*
- Value** *For value-based objects **interpreted content** is the dominant characteristic, followed by behaviour in terms of this state. In contrast to entities, values are transient and **do not have significant enduring identity**.*

...

[HENNEY, 'Objects of Value']



Rappel sur les sémantiques

Pendant ce temps chez nos voisins

Ailleurs l'approche est plus technique, on qualifie d'abord la propriété de l'objectif

value semantics on manipule des valeurs

reference semantics on manipule via pointeurs et références



Rappel sur les sémantiques I

Value ou Regular ?

*'Entity types normally do not override `Object.equals()`, because there is no point to comparing their content, since it is not overly relevant anyway. In contrast, value types do override `Object.equals()`, because equality means "equal content". For a value type there is a significant difference between **identity** and **equality**. In other words, only classes that represent value types have a need to override `Object.equals()`.*

For this reason, the subsequent discussion of implementing `equals()` is only relevant for value types.'

[LANGER et KREFT, 'Secrets of equals 1 : Not all implementations of `equals()` are equal



Rappel sur les sémantiques II

Value ou Regular ?

`std::semiregular` specifies that an object of a type can be copied, moved, swapped, and default constructed

`std::regular` specifies that a type is regular, that is, it is both semiregular and equality comparable

[C++20], [STEPANOV et MCJONES, *Elements of Programming*]



2. Understanding Value Semantics

Value-Semantic Properties

A *value-semantic* type T defines the following:

- Default construction: T a b
- Copy

“Regular Type”

- Swap (if well-formed): T $a(\alpha)$, $b(\beta)$; `swap(a, b);`
`assert($\beta == a$);`
`assert($\alpha == b$);`

188

[LAKOS, *CppCon 2015* : "Value Semantics : It ain't about the syntax !, Part I"]

Rappel sur les sémantiques

Applications – cf. [JOLY, CPPP-19 : "Élégance, style épuré et classe"]

L'intérêt dans tout ça

- on a des **heuristiques simples** pour identifier des catégories de types
- auxquelles on va associer des **recettes de cuisines** toutes faites
 - copiables ou pas
 - cf. règles FCOC $\rightarrow 3 \rightarrow 2 \rightarrow 0/5$
 - ou mieux défautier/supprimer/définir [MERTZ, *The rule of zero revisited : the rule of all or nothing*])
- et ainsi éviter
 - des bugs – j'y viens bientôt
 - mais aussi une vraie problématique conceptuelle
 - perte de temps à développer des choses dont on n'a jamais besoin – déjà perdu 3 mois à vouloir suivre la Forme Canonique Orthodoxe de Coplien sur une entité



Rappel sur les sémantiques

Rapidement : mon modèle simplifié

	Aggrégat	Valeur	Entité	Capsule RAIL	Hybrides
Invariant	✗	✓	✓	✓	✓
Identité	✗/✓	✗	✓	✗	?
Copiable	attributs	✓	✗	✗	✓
Déplaçable	attributs	✓ / copié	✗	✓	?
Comparable	attributs	✓ (Regular)	✗ (morceaux)	✗	?
Typiquement manipulé		valeur	réf./ptr.	valeur	valeur/réf.
Ex. sous-catégorie		objets math.	hiér. polym.		exceptions



Le slicing

L'erreur la plus connue, et typique du C++

```
struct Base {
    int i;
    friend std::ostream & operator<<(std::ostream & os, Base const& v)
    { return os << "i: " << std::setw(3) << v.i; }
};

struct Child : Base {
    int j;
    friend std::ostream & operator<<(std::ostream & os, Child const& v)
    { return os << static_cast<Base const&>(v) << " - j: " << std::setw(3) << v.j; }
};

int main() {
    Child c1{1, 2};
    Child c2{10, 42};

    std::cout << "c1 : " << c1 << "\n"; // c1 : i: 1 - j: 2
    std::cout << "c2 : " << c2 << "\n"; // c2 : i: 10 - j: 42

    // tronçonnage classique
    Base s = c1;
    std::cout << "s : " << s << "\n"; // s : i: 1

    // quand on commence à s'emmeler les pinceaux
    Base & b = c2;
    b = c1;
    std::cout << "c2 : " << c2 << "\n"; // c2 : i: 1 - j: 42
}
```



En vrai, cela nous rattrape généralement à notre insu sur

```
void scenario_type__mode_feignasse_on(Base b) {  
    // Toute spécificité de l'enfant est perdue!  
}
```



Le slicing

Et n'oublions pas la vtbl !

```
#include <iostream>
struct Base {
    int i;
    Base(int i_) : i{i_} {}
    virtual int f(int x) const noexcept { return x + i; }
};

struct Child : Base {
    int j;
    Child(int i_, int j_) : Base{i_}, j{j_} {}
    int f(int x) const noexcept override { return j * x + i; }
};

int let_s_slice_vtbl(Base b) {
    return b.f(42);
}

int main()
{
    Child c{1, 2};
    std::cout << let_s_slice_vtbl(c) << "\n"; // 43 au lieu de 85... normal? pas normal?
}
```



Le slicing

Et n'oublions pas la vtbl !

```
#include <iostream>
struct Base {
    int i;
    Base(int i_) : i{i_} {}
    virtual int f(int x) const noexcept { return x + i; }
};

struct Child : Base {
    int j;
    Child(int i_, int j_) : Base{i_}, j{j_} {}
    int f(int x) const noexcept override { return j * x + i; }
};

int let_s_slice_vtbl(Base b) {
    return b.f(42);
}

int main()
{
    Child c{1, 2};
    std::cout << let_s_slice_vtbl(c) << "\n"; // 43 au lieu de 85... normal? pas normal?
}
```

C'est vraiment ce que voulait faire notre prédécesseur ?



Le slicing

La conclusion technique évidente

'C.67 : A polymorphic class should suppress copying'

[C++CG]

```
$> clang-tidy slicing-virtual.cpp --check=* --std=c++17
slicing-virtual.cpp:21:36: warning: slicing object from type 'Child' to 'Base' discards 4 bytes of state [cppcoreguidelines-slicing]
    std::cout << let_s_slice_vtble(c) << "\n"; // 43 au lieu de 85... normal? pas normal?
                                ^
slicing-virtual.cpp:21:36: warning: slicing object from type 'Child' to 'Base' discards override 'f' [cppcoreguidelines-slicing]
```



① Avant-propos

② La raison technique

③ La raison conceptuelle

Erreurs de programmation

La Programmation par Contrat

Principe de Substitution de Liskov
(LSP)

Égalite et LSP Sont sur un bateau

④ Conclusion

⑤ Références



La raison conceptuelle

Posons quelques bases

Avant d'entrer dans le vif du sujet, faisons un petit détour par la programmation par contrat.



C'est quoi une erreur de programmation ?

➤ Erreur dans des algorithmes/calculs ;

- p.ex. `sin()` qui renvoie des valeurs supérieures à 1, mélange entre des pieds et des mètres, ...

➤ Erreur dans des suppositions

- p.ex. pointeurs non nuls, indice hors bornes. . .



Que faire en cas d'erreur de programmation ?

➡ Rien, pas même chercher à la détecter



Que faire en cas d'erreur de programmation ?

- Rien, pas même chercher à la détecter
 - Programme qui crashe plus loin, sans contexte ;
 - ou programme qui donne des résultats aberrants, qui seront détectés, un jour, ou peut-être jamais.



Que faire en cas d'erreur de programmation ?

- Rien, pas même chercher à la détecter
 - Programme qui crashe plus loin, sans contexte ;
 - ou programme qui donne des résultats aberrants, qui seront détectés, un jour, ou peut-être jamais.
- La détecter pour lancer une exception



Que faire en cas d'erreur de programmation ?

- Rien, pas même chercher à la détecter
 - Programme qui crashe plus loin, sans contexte ;
 - ou programme qui donne des résultats aberrants, qui seront détectés, un jour, ou peut-être jamais.
- La détecter pour lancer une exception
 - On est prévenus que quelque chose ne va pas,
 - on ne plante pas (ni en prod, ni en dev & tests), mais ...
 - perte du contexte pour investigation par l'équipe de dev



Que faire en cas d'erreur de programmation ?

- Rien, pas même chercher à la détecter
 - Programme qui crashe plus loin, sans contexte ;
 - ou programme qui donne des résultats aberrants, qui seront détectés, un jour, ou peut-être jamais.
- La détecter pour lancer une exception
 - On est prévenus que quelque chose ne va pas,
 - on ne plante pas (ni en prod, ni en dev & tests), mais ...
 - perte du contexte pour investigation par l'équipe de dev
- La détecter pour claquer une assertion

Que faire en cas d'erreur de programmation ?

- Rien, pas même chercher à la détecter
 - Programme qui crashe plus loin, sans contexte ;
 - ou programme qui donne des résultats aberrants, qui seront détectés, un jour, ou peut-être jamais.
- La détecter pour lancer une exception
 - On est prévenus que quelque chose ne va pas,
 - on ne plante pas (ni en prod, ni en dev & tests), mais ...
 - perte du contexte pour investigation par l'équipe de dev
- La détecter pour claquer une assertion
 - On est prévenus, en phase de développement et de tests, que quelque chose ne va pas, et on dispose d'un contexte exact du soucis au moment où il est détecté ;
 - on fait comme si tout allait bien en phase de prod (sauf si on décide de doubler par une exception si le projet exige de la *programmation défensive*.)



Que faire en cas d'erreur de programmation ?

- Rien, pas même chercher à la détecter
 - Programme qui crashe plus loin, sans contexte ;
 - ou programme qui donne des résultats aberrants, qui seront détectés, un jour, ou peut-être jamais.
- La détecter pour lancer une exception
 - On est prévenus que quelque chose ne va pas,
 - on ne plante pas (ni en prod, ni en dev & tests), mais ...
 - perte du contexte pour investigation par l'équipe de dev
- La détecter pour claquer une assertion
 - On est prévenus, en phase de développement et de tests, que quelque chose ne va pas, et on dispose d'un contexte exact du soucis au moment où il est détecté ;
 - on fait comme si tout allait bien en phase de prod (sauf si on décide de doubler par une exception si le projet exige de la *programmation défensive*.)
- Attendre le C++23



Quels outils pour s'en protéger ?

➤ Invariants statiques

- p.ex. constructeur vs initialisation différée, référence vs pointeur, `signed` vs `unsigned`, ...

➤ Typage renforcé

- p.ex. cf. `Boost.unit`, ou les *User-Defined literals* du C++11.

➤ Assertions statiques

- p.ex. taille tableau statique == nombre d'énumérés, type reçu supporte au moins 42000 valeurs (prog. générique), ...

➤ Outils d'analyse statique du code source (Frama-C, Polyspace, autre ?)

➤ TU, TV et assertions

➤ Formaliser les contrats (PpC)



Quels outils pour s'en protéger ?

➤ Invariants statiques

- p.ex. constructeur vs initialisation différée, référence vs pointeur, `signed` vs `unsigned`, ...

➤ Typage renforcé

- p.ex. cf. `Boost.unit`, ou les *User-Defined literals* du C++11.

➤ Assertions statiques

- p.ex. taille tableau statique == nombre d'énumérés, type reçu supporte au moins 42000 valeurs (prog. générique), ...

➤ Outils d'analyse statique du code source (Frama-C, Polyspace, autre ?)

➤ TU, TV et assertions

➤ Formaliser les contrats (PpC)



La Programmation par Contrat

Ils sont partout !

➤ Qu'on le veuille ou non



La Programmation par Contrat

Ils sont partout !

- Qu'on le veuille ou non
- Toute fonction a un contrat



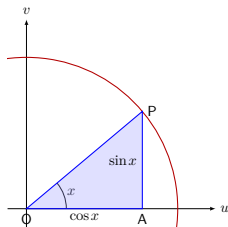
La Programmation par Contrat

Illustration

Exemple

`double sin(double x)`

- x est en nombre flottant (double précision) exprimant un angle en radian
- le résultat est l'ordonnée du point P sur le cercle unitaire dont l'angle \widehat{AOP} vaut x
- incidemment compris entre -1 et 1 .



- *Pré-condition* : conditions que doit remplir l'appelant d'une fonction pour que cette dernière ait une chance de bien se dérouler. En cas de non respect du contrat par l'appelant, l'appelé ne garantit rien et tout peut arriver.
- *Post-condition* : conditions vérifiées par l'appelé, et la valeur retournée, après appel d'une fonction.
- *Invariant* : ensemble de propriétés qu'une classe doit respecter avant et après chaque appel de fonction de l'interface.



- La PpC, c'est avant tout des garanties si tout va bien et c'est tout.
- On respecte \Rightarrow on aura un comportement prévisible et valide. Mais si on ne respecte pas le contrat, tout peut arriver.
C'est (quasi) le pays des *Undefined behaviours*.



- À rapprocher des domaines de définition
- Si respect des pré-conditions avant appel, alors l'appel doit réussir et produire les résultats attendus dans les post-conditions.
- Le responsable est l'appelant.
- Les assertions sont nos amies – mode *fail fast*.



- Garanties sur résultats d'une fonction si pré-conditions remplies, et aucune erreur de *runtime*.
- Si la fonction sait qu'elle ne peut pas remplir ses post-conditions, alors elle doit échouer.
- Il ne s'agit pas de détecter les erreurs de programmation, mais de contexte.
- Le responsable est l'appelé.
- Relève plus du test unitaire que de l'assertion.

Programmation par Contrat

Post-conditions

```
std::vector<int> sort(std::vector<int> s)  
[[post: i<j: res[i] <= res[j]]]
```



Programmation par Contrat

Post-conditions

```
std::vector<int> sort(std::vector<int> s)
```

```
[[post: i<j: res[i] <= res[j]]]
```

```
sort({2, 1, 3, 2}) -> {1, 2, 3} ??
```

Programmation par Contrat

Post-conditions

```
std::vector<int> sort(std::vector<int> s)
```

```
[[post: i<j: res[i] <= res[j]]]
```

```
sort({2, 1, 3, 2}) -> {1, 2, 3} ??
```

```
sort({1, 3, 2, 2}) -> {0, 1, 2, 3} ??
```

Programmation par Contrat

Post-conditions

```
std::vector<int> sort(std::vector<int> s)  
[[post: i<j: res[i] <= res[j]]]
```

$\text{sort}(\{2, 1, 3, 2\}) \rightarrow \{1, 2, 3\} ??$

$\text{sort}(\{1, 3, 2, 2\}) \rightarrow \{0, 1, 2, 3\} ??$

$\text{sort}(\{1, 2, 3, 2\}) \rightarrow \{1, 2, 3, 3\} ??$



```
std::vector<int> sort(std::vector<int> s)
```

```
[[post: i<j: res[i] <= res[j]]]
```

```
sort({2, 1, 3, 2}) -> {1, 2, 3} ??
```

```
sort({1, 3, 2, 2}) -> {0, 1, 2, 3} ??
```

```
sort({1, 2, 3, 2}) -> {1, 2, 3, 3} ??
```

La post-condition,

- ce n'est pas seulement trié, mais exactement les mêmes éléments,
- ce qui coûte aussi cher ici.

S'applique à des zones durant lesquelles une propriété restera vraie.

- Invariant de boucle (voire, variant de boucle)
- Référence (pointeur jamais nul)
- Variable (devrait toujours être «*Est utilisable, et est dans un état cohérent et pertinent*», positionné après construction)
- Invariant de classe (propriété toujours observable depuis du code extérieur aux instances de la classe).

Programmation par Contrat

Acteurs et Responsabilités

```
double metier() { // écrit par l'Intégrateur  
    const double i = interrogeES(); // écrit par le responsable UI  
    return sqrt(i); // écrit par le Mathématicien  
}
```

`sqrt` échoue (assertion, résultat abérrant, NaN, ...)

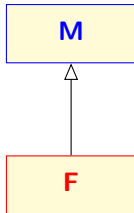
- Si `i` est positif \Rightarrow Mathématicien
- Si `i` est négatif \Rightarrow pas le Mathématicien mais
 - Si `interrogeES()` a pour post-cond positif \Rightarrow Responsable UI
 - Sinon \Rightarrow Intégrateur



PpC \rightarrow LSP

Héritage et préconditions

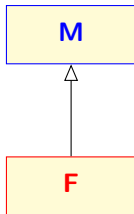
```
void g(M & o) {  
  for (auto x : E_M)  
    o.f(x);  
}
```



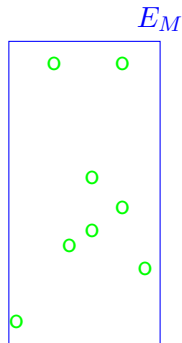
PpC \rightarrow LSP

Héritage et préconditions

```
void g(M & o) {  
    for (auto x : E_M)  
        o.f(x);  
}
```



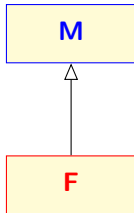
```
void M::f(E x)  
[[pre: x ∈ E_M]];  
...  
g(M{});
```



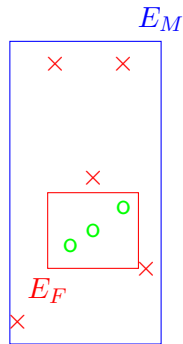
PpC \rightarrow LSP

Héritage et préconditions

```
void g(M & o) {  
    for (auto x : E_M)  
        o.f(x);  
}
```



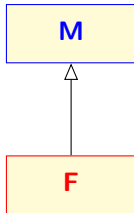
```
void F::f(E x) override  
[[pre: x ∈ E_F]];  
...  
g(F{});
```



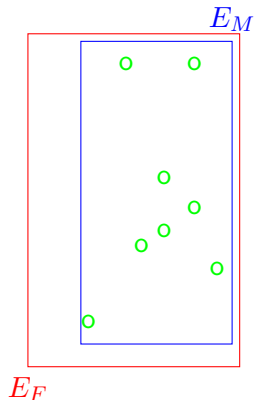
PpC \rightarrow LSP

Héritage et préconditions

```
void g(M & o) {  
  for (auto x : E_M)  
    o.f(x);  
}
```



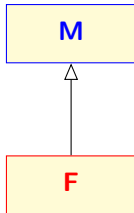
```
void F::f(E x) override  
[[pre: x ∈ E_F]];  
...  
g(F{});
```



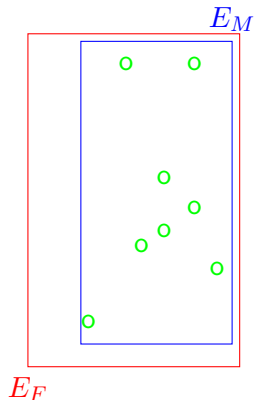
PpC \rightarrow LSP

Héritage et préconditions

```
void g(M & o) {  
    for (auto x : E_M)  
        o.f(x);  
}
```



```
void F::f(E x) override  
[[pre: x ∈ E_F]];  
...  
g(F{});
```

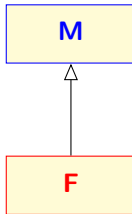


➞ On ne peut qu'affaiblir

PpC \rightarrow LSP

Héritage et postconditions

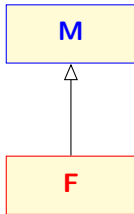
```
void h(M & o) {  
    assert(o.f() in  $E'_M$ );  
    ...  
}
```



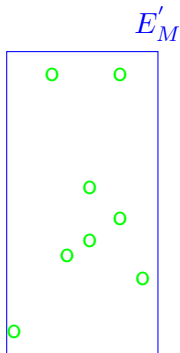
PpC \rightarrow LSP

Héritage et postconditions

```
void h(M & o) {  
    assert(o.f() in  $E'_M$ );  
    ...  
}
```



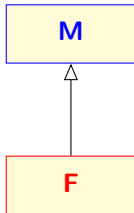
```
E M::f()  
[[post r: r  $\in E'_M$ ]];  
...  
h(M{});
```



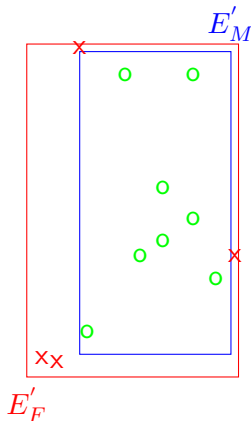
PpC \rightarrow LSP

Héritage et postconditions

```
void h(M & o) {  
    assert(o.f() in  $E'_M$ );  
    ...  
}
```



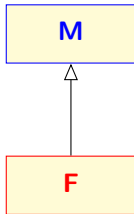
E F::f() override
[[post r: $r \in E'_F$]];
...
h(F{});



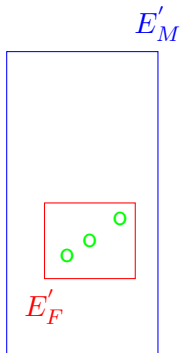
PpC \rightarrow LSP

Héritage et postconditions

```
void h(M & o) {  
    assert(o.f() in  $E'_M$ );  
    ...  
}
```



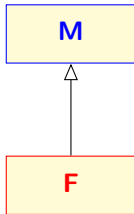
E F::f() override
[[post r: $r \in E'_F$]];
...
h(F{ });



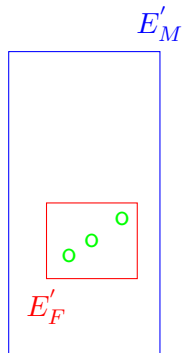
PpC \rightarrow LSP

Héritage et postconditions

```
void h(M & o) {  
    assert(o.f() in  $E'_M$ );  
    ...  
}
```



E F::f() override
[[post r: $r \in E'_F$]];
...
h(F{});



➡ On ne peut que renforcer

PpC → LSP

Évolution dans une hiérarchie de classe

- pré-conditions : elles ne peuvent être qu'affaiblies par les classes dérivées.
- post-condition : elles ne peuvent être que renforcées par les classes dérivées.
- une classe dérivée ne peut qu'ajouter des invariants.



Partout où on attend un objet de type A, on peut passer un objet de type B, si et seulement si B dérive publiquement de A

- EST-SUBSTITUABLE-A : Règle fondamentale pour la construction des hiérarchies de classes (remplace le EST-UN)
- Une `ListeTrie` N'EST PAS SUBSTITUABLE A une `Liste`.
Un `Point3d` n'est pas un `Point2d`.
Un `PointColoré` n'est pas un `Point` [2]

Quelques contrevenants aux LSP

Carrés et Rectangles

```
struct Rectangle {  
    double largeur() const;  
    double longueur() const;  
    double aire() const {return largeur()*longueur();}  
  
    void double_largeur()  
        ;  
private: ...  
};
```

```
auto f(Rectangle &o) {  
    if (o.aire() >= 24) throw...  
    ...  
    z = ...;  
    o.double_largeur();  
  
    return z / sqrt(42 - o.aire());  
}
```



Quelques contrevenants aux LSP

Carrés et Rectangles

```
struct Rectangle {  
    double largeur() const;  
    double longueur() const;  
    double aire() const {return largeur()*longueur();}  
  
    void double_largeur()  
    [[post: 2 * __old__.aire() == aire()]];  
private: ...  
};
```

```
auto f(Rectangle &o) {  
    if (o.aire() >= 24) throw...  
    ...  
    z = ...;  
    o.double_largeur();  
    assert(o.aire() < 42);  
    return z / sqrt(42 - o.aire());  
}
```



Quelques contrevenants aux LSP

Carrés et Rectangles

```
struct Rectangle {  
    double largeur() const;  
    double longueur() const;  
    double aire() const {return largeur()*longueur();}  
    virtual  
    void double_largeur()  
    [[post: 2 * __old__.aire() == aire()]];  
private: ...  
};  
  
struct Carre : Rectangle [[inv: largeur() == longueur()]]  
{  
    void double_largeur() override  
    [[post: 2 * __old__.aire() == aire()]]; // toujours!  
};
```

```
auto f(Rectangle &o) {  
    if (o.aire() >= 24) throw...  
    ...  
    z = ...;  
    o.double_largeur();  
    assert(o.aire() < 42);  
    return z / sqrt(42 - o.aire());  
}
```



Quelques contrevenants aux LSP

Carrés et Rectangles

```
struct Rectangle {
    double largeur() const;
    double longueur() const;
    double aire() const {return largeur()*longueur();}
    virtual
    void double_largeur()
        [[post: 2 * __old__.aire() == aire()]];
private: ...
};

struct Carre : Rectangle [[inv: largeur() == longueur()]]
{
    void double_largeur() override
        [[post: 2 * __old__.aire() == aire()]]; // toujours!
};
```

```
auto f(Rectangle &o) {
    if (o.aire() >= 24) throw...
    ...
    z = ...;
    o.double_largeur();
    assert(o.aire() < 42);
    return z / sqrt(42 - o.aire());
}
```

➤ Contrats incompatibles !



Quelques contrevenants aux LSP

Carrés et Rectangles

```
struct Rectangle {  
    double largeur() const;  
    double longueur() const;  
    double aire() const {return largeur()*longueur();}  
    virtual  
    void double_largeur()  
    [[post: 2 * __old__.aire() == aire()]];  
private: ...  
};  
  
struct Carre : Rectangle [[inv: largeur() == longueur()]]  
{  
    void double_largeur() override  
    [[post: 2 * __old__.aire() == aire()]]; // toujours!  
};
```

```
auto f(Rectangle &o) {  
    if (o.aire() >= 24) throw...  
    ...  
    z = ...;  
    o.double_largeur();  
    assert(o.aire() < 42);  
    return z / sqrt(42 - o.aire());  
}
```

➤ Contrats incompatibles !

➤ Quel contrat respecter

- la post-condition de `double_largeur` ?
- ou l'invariant des carrés ?

➤ Pas de solution !

- Un carré modifiable n'est pas un rectangle modifiable !!



Quelques contrevenants aux LSP

Liste et liste triée

```
struct List {  
    size_t size() const;  
    int const & back() const; [[pre: size() > 0]]  
    void push_back(int i)  
        [[post: __old__.size()+1 == size() && back() == i]];  
};
```

```
List& func(List & l) {  
    ...  
    l.push_back(42);  
    assert(l.back() == 42 && "Parce que!");  
    return l;  
}  
  
func(List{1, 4, 2}) // -> {1, 4, 2, 42}
```



Quelques contrevenants aux LSP

Liste et liste triée

```
struct List {
    size_t size() const;
    int const & back() const; [[pre: size() > 0]]
    void push_back(int i)
        [[post: __old__.size()+1 == size() && back() == i]];
};

struct SortedList : List
[[inv: is_sorted(*this)]]
{};
```

```
List& func(List & l) {
    ...
    l.push_back(42);
    assert(l.back() == 42 && "Parce que!");
    return l;
}

func(List{1, 4, 2}) // -> {1, 4, 2, 42}
```

Quelques contrevenants aux LSP

Liste et liste triée

```
struct List {  
    size_t size() const;  
    int const & back() const; [[pre: size() > 0]]  
    void push_back(int i)  
        [[post: __old__.size()+1 == size() && back() == i]];  
};  
  
struct SortedList : List  
[[inv: is_sorted(*this)]]  
{};
```

```
List& func(List & l) {  
    ...  
    l.push_back(42);  
    assert(l.back() == 42 && "Parce que!");  
    return l;  
}  
  
func(List{1, 4, 2}) // -> {1, 4, 2, 42}  
  
func(SortedList{1, 4, 2}) // -> {1, 2, 4, 42} OK
```

Quelques contrevenants aux LSP

Liste et liste triée

```
struct List {  
    size_t size() const;  
    int const & back() const; [[pre: size() > 0]]  
    void push_back(int i)  
        [[post: __old__.size()+1 == size() && back() == i]];  
};
```

```
struct SortedList : List  
[[inv: is_sorted(*this)]]  
{};
```

```
List& func(List & l) {
```

```
    ...  
    l.push_back(42);  
    assert(l.back() == 42 && "Parce que!");  
    return l;  
}
```

```
func(List{1, 4, 2}) // -> {1, 4, 2, 42}
```

```
func(SortedList{1, 4, 2}) // -> {1, 2, 4, 42} OK
```

```
func(SortedList{100, 4, 2}) // -> {2, 4, 100, 42} oups
```

```
func(SortedList{100, 4, 2}) // -> {2, 4, 42, 100} reoups
```

Égalite et LSP Sont sur un bateau

Propriétés de l'égalité

Relation d'équivalence

réflexive $x = x$

symétrique $x = y \Leftrightarrow y = x$

transitive $x = y$ et $y = z \Rightarrow x = z$



Égalite et LSP Sont sur un bateau

Points et PointColorés

Supposons

```
struct Point { int x; int y; };  
struct ColoredPoint : Point { Color c; };
```

Plus une fonction qui permet de comparer des Point



Égalité et LSP Sont sur un bateau

Points et PointColorés

Supposons

```
struct Point { int x; int y; };  
struct ColoredPoint : Point { Color c; };
```

Plus une fonction qui permet de comparer des Point

Substituabilité syntaxique oblige, on peut passer les uns et les autres à une fonction qui attend des Point const&.



Égalité et LSP Sont sur un bateau

Points et PointColorés

Supposons

```
struct Point { int x; int y; };  
struct ColoredPoint : Point { Color c; };
```

Plus une fonction qui permet de comparer des Point

Substituabilité syntaxique oblige, on peut passer les uns et les autres à une fonction qui attend des Point const&.

et donc...



Égalite et LSP Sont sur un bateau

Points et PointColorés

Supposons

```
struct Point { int x; int y; };  
struct ColoredPoint : Point { Color c; };
```

Plus une fonction qui permet de comparer des Point

Substituabilité syntaxique oblige, on peut passer les uns et les autres à une fonction qui attend des Point const&.

et donc... on peut les comparer !



Égalite et LSP Sont sur un bateau

Points et PointColorés

Supposons

```
struct Point { int x; int y; };  
struct ColoredPoint : Point { Color c; };
```

Plus une fonction qui permet de comparer des Point
Substituabilité syntaxique oblige, on peut passer les uns et les autres à une fonction
qui attend des Point const&.

et donc... on peut les comparer !

Refuser la comparaison, c'est refuser le LSP, et donc ouvrir la porte à des erreurs
inattendues.



Égalite et LSP Sont sur un bateau

Points et PointColorés – disclaimer

- Les problématiques présentées font l'objet de nombreux articles dans le monde Java [BLOCH, *Effective Java*], [LANGER et KREFT]...
- Beaucoup tournent autour de `Myclass.equals(Object)`,
- et peu explorent une symétrie complète.



Égalité et LSP Sont sur un bateau

Points et PointColorés – disclaimer

- Les problématiques présentées font l'objet de nombreux articles dans le monde Java [BLOCH, *Effective Java*], [LANGER et KREFT]...
- Beaucoup tournent autour de `Myclass.equals(Object)`,
- et peu explorent une symétrie complète.
- Certains vont même jusqu'à dire : *The Liskov Substution Principle does not hold in Java* [alblue]



Égalité et LSP Sont sur un bateau

Points et PointColorés – disclaimer

- Les problématiques présentées font l'objet de nombreux articles dans le monde Java [BLOCH, *Effective Java*], [LANGER et KREFT]...
- Beaucoup tournent autour de `Myclass.equals(Object)`,
- et peu explorent une symétrie complète.
- Certains vont même jusqu'à dire : *The Liskov Substution Principle does not hold in Java* [alblue]

Soyons fous, supposons un multi-method (polymorphisme sur plus d'un paramètre) en standard, ou juste un *double-dispatch* dynamique.

- C'est à dire : $P == PC$, ou $P == P$, ou $PC == PC$ dispatchera à la bonne comparaison quelque soit le type statique de l'objet considéré



Égalite et LSP Sont sur un bateau

Points et PointColorés

➤ Donc LSP oblige, on veut pouvoir écrire comparer les objets dynamiques

```
assert(Point{1, 2} == PointColoré{1, 2, rouge});
```



Égalite et LSP Sont sur un bateau

Points et PointColorés

- Donc LSP oblige, on veut pouvoir écrire comparer les objets dynamiques

```
assert(Point{1, 2} == PointColore{1, 2, rouge});
```

- donc on a aussi

```
assert(Point{1, 2} == PointColore{1, 2, vert});
```

Égalité et LSP Sont sur un bateau

Points et PointColorés

- Donc LSP oblige, on veut pouvoir écrire comparer les objets dynamiques

```
assert(Point{1, 2} == PointColore{1, 2, rouge});
```

- donc on a aussi

```
assert(Point{1, 2} == PointColore{1, 2, vert});
```

- Mais...



Égalité et LSP Sont sur un bateau

Points et PointColorés

- Donc LSP oblige, on veut pouvoir écrire comparer les objets dynamiques

```
assert(Point{1, 2} == PointColore{1, 2, rouge});
```

- donc on a aussi

```
assert(Point{1, 2} == PointColore{1, 2, vert});
```

- Mais... la transitivité dit alors que

```
assert(PointColore{1, 2, rouge} == PointColore{1, 2, vert});
```



'Value type superclasses . The problematic cases are non-final classes that represent value types. They must implement equals() to reflect the value semantics and they are superclasses, because every non-final class by definition is a potential superclass.'

If the designer of such a non-final class decides in favor of implementing equals() using instanceof, then no subclass can ever add fields and override equals() without violating the transitivity requirement of the equals() contract.

If the designer decides in favor of implementing equals() using getClass(), then no subclass object will ever be comparable to a superclass object and trivial extensions may not make a lot of sense.'

[LANGER et KREFT, 'Secrets of equals 1 : Not all implementations of equals() are equal']



Égalité et LSP Sont sur un bateau

Points et PointColorés – sinon

On peut donc

- laisser tomber la transitivité
- laisser tomber les comparaisons dans une hiérarchie (LSP)
- ou introduire une valeur par défaut pour les champs manquants
 - couleur 0... (sens ?)
 - z nul (et pourquoi pas 7 droites perpendiculaires entre elles ?)

Et respecte-t'on vraiment un *esprit* du LSP ?



Égalité et LSP Sont sur un bateau

Points et PointColorés – sinon

On peut donc

- laisser tomber la transitivité
- laisser tomber les comparaisons dans une hiérarchie (LSP)
- ou introduire une valeur par défaut pour les champs manquants
 - couleur 0... (sens ?)
 - z nul (et pourquoi pas 7 droites perpendiculaires entre elles ?)

Et respecte-t'on vraiment un *esprit* du LSP ?

- ou ne pas permettre de substituabilité syntaxique pour importer du code
 - composition (Réponse de [BLOCH, *Effective Java*])
 - héritage privé

Et employer des fonctions dédiées pour comparer des objets de types différents
[LAKOS, *CppCon 2015 : "Value Semantics : It ain't about the syntax!", Part I"*]



Égalite et LSP Sont sur un bateau

'It doesn't really makes sense to talk about things that may or may not have the same type.'

[LAKOS, CppCon 2015 : "Value Semantics : It ain't about the syntax!", Part I", part 1 4



Plan partiel

① Avant-propos

② La raison technique

③ La raison conceptuelle

④ Conclusion

Perspectives

Questions ?

⑤ Références



Je n'ai pas traité

- Des outils liés à la PpC (assertion, types opaques, tests unitaires, analyse statiques de code)
- De l'approche semi-antagoniste à la PpC offensive : la programmation excessivement défensive.
- Quid de l'enseignement
 - Quelle est la validité pédagogique d'exemples simples (de factorisation de données...) qui ont un désign incorrect
- substituabilité syntaxique
- polymorphicvalue
- semi-slicing $C \rightarrow P \text{ const\&} \rightarrow C \text{ const\&}$



Questions ?







Plan partiel

- ① Avant-propos
- ② La raison technique

- ③ La raison conceptuelle
- ④ Conclusion
- ⑤ **Références**







Références I

-  Julien BLANC. *Programmation par contrat, application en C++*.
http://julien-blanc.developpez.com/articles/cpp/Programmation_par_contrat_cplusplus/. Déc. 2009.
-  Joshua BLOCH. *Effective Java*. Mai 2008.
-  Bjarne STROUSTRUP et Herb SUTTER, éd. *C++ Core Guidelines*.
<https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md>. C++CG, 2015.
-  Gabriel DOS REIS et al.
<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0542r1.html>.
SupportforcontractbasedprogramminginC++. Juin 2017.







Références II

-  Philippe DUNSKI et Luc HERMITTE. *Coder Efficacement - Bonnes pratiques et erreurs à éviter (en C++)*. www.d-booker.fr/programmation-et-langage/157-coder-efficacement.html. Fév. 2014.
-  Kevlin HENNEY. 'Objects of Value'. In : *Programmer's workshop* (nov. 2003).
-  Loïc JOLY. *CPPP-19 : "Élégance, style épuré et classe"*. https://raw.githubusercontent.com/cppp-france/CPPP-19/master/elegance_style_epure_et_classe-Loic_Joly/elegance_style_epure_et_classe-Loic_Joly.pdf. 2019.
-  Andrzej KRZEMIENSKI. *Préconditions en C++ - partie 1*. <http://akrzemi1.developpez.com/tutoriels/c++/preconditions/partie-1/>. Traduction. Jan. 2013.






Références III

-  Andrzej KRZEMIENSKI. *Préconditions en C++ - partie 2*. <http://akrzemi1.developpez.com/tutoriels/c++/preconditions/partie-2/>. Traduction. Fév. 2013.
-  Andrzej KRZEMIENSKI. *Préconditions en C++ - partie 3*. <http://akrzemi1.developpez.com/tutoriels/c++/preconditions/partie-3/>. Traduction. Mar. 2013.
-  Andrzej KRZEMIENSKI. *Préconditions en C++ - partie 4*. <http://akrzemi1.developpez.com/tutoriels/c++/preconditions/partie-4/>. Traduction. Avr. 2013.
-  John LAKOS. *CppCon 2015 : "Value Semantics : It ain't about the syntax!", Part I*. <https://www.youtube.com/watch?v=W3xI1HJUy7Q>. Sept. 2015.



Références IV

-  John LAKOS. *Defensive Programming Done Right - part 1*. https://www.youtube.com/watch?v=1QhtXRMp3Hg&feature=youtube_gdata. Sept. 2014.
-  John LAKOS. *Defensive Programming Done Right - part 2*. https://www.youtube.com/watch?v=tz2khnjnUx8&feature=youtube_gdata. Sept. 2014.
-  Angelika LANGER et Klaus KREFT. 'Secrets of equals 1 : Not all implementations of equals() are equal'. In : *Java Solutions* (avr. 2002).
-  Angelika LANGER et Klaus KREFT. 'Secrets of equals 2 : How to implement a correct slice comparison in Java'. In : *Java Solutions* (avr. 2002).



Références V



Arne MERTZ. *The rule of zero revisited : the rule of all or nothing*.
<http://arne-mertz.de/2015/02/the-rule-of-zero-revisited-the-rule-of-all-or-nothing/>. Fév. 2015.



Bertrand MEYER. *Conception et programmation orientées objet*.
<http://www.editions-eyrolles.com/Livre/9782212122701/conception-et-programmation-orientees-objet>. Jan. 2008.








Gregory PAKOSZ. *Assertions Or Exceptions ?* <https://pempek.net/articles/2013/11/16/assertions-or-exceptions/>. Nov. 2013.



Gregory PAKOSZ. *Cross Platform C++ Assertion Library*.
<https://pempek.net/articles/2013/11/17/cross-platform-cpp-assertion-library/>. Nov. 2013.



Références VI

-  John REGEHR. *Use of assertions*.
<https://blog.regehr.org/archives/1091>. Fév. 2014.
-  Alexander STEPANOV et Paul MCJONES. *Elements of Programming*. Juin 2019.
ISBN : 978-0-578-22214-1. URL : <http://elementsofprogramming.com>.
-  Herb SUTTER. *When and How to Use Exception*.
<http://www.drdobbs.com/when-and-how-to-use-exceptions/184401836>.
Jan. 2004.
-  Matthew WILSON. *Contract Programming 101*.
<http://www.artima.com/cppsource/deepspace3.html>. 2006.
-  Matthew WILSON. *Imperfect C++*. 2004.

