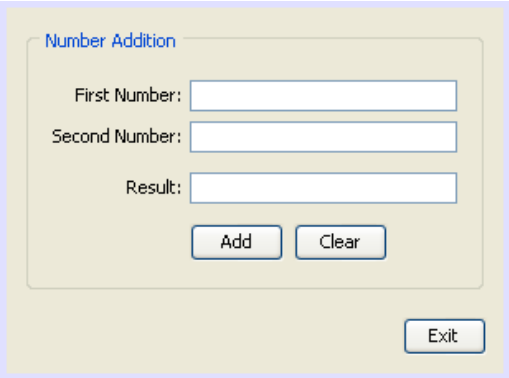
ÉCOLE NATIONALE D'INGÉNIEURS DE BREST

Langages Orientés Objet

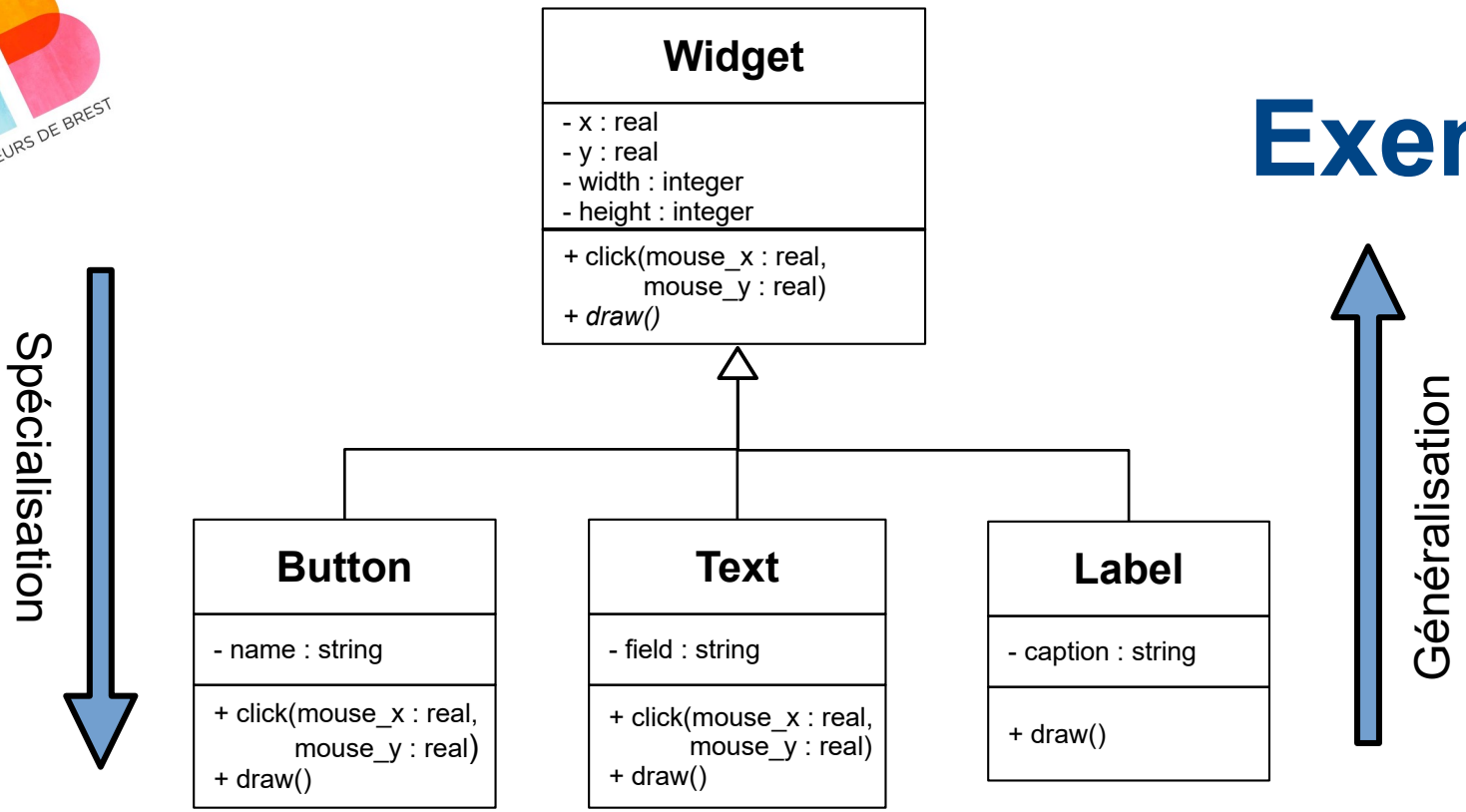
Héritage et
Polymorphisme dynamique

Héritage

- Un des concepts de la POO est l'héritage.
- L'héritage est une relation qui permet de créer des classes (dérivées, sous-classes) à partir d'autres classes (bases, superclasses) en incluant automatiquement les membres de ces dernières.
- Elle est aussi une relation de type « *c'est un* », toute classe dérivée est aussi de type classe de base.

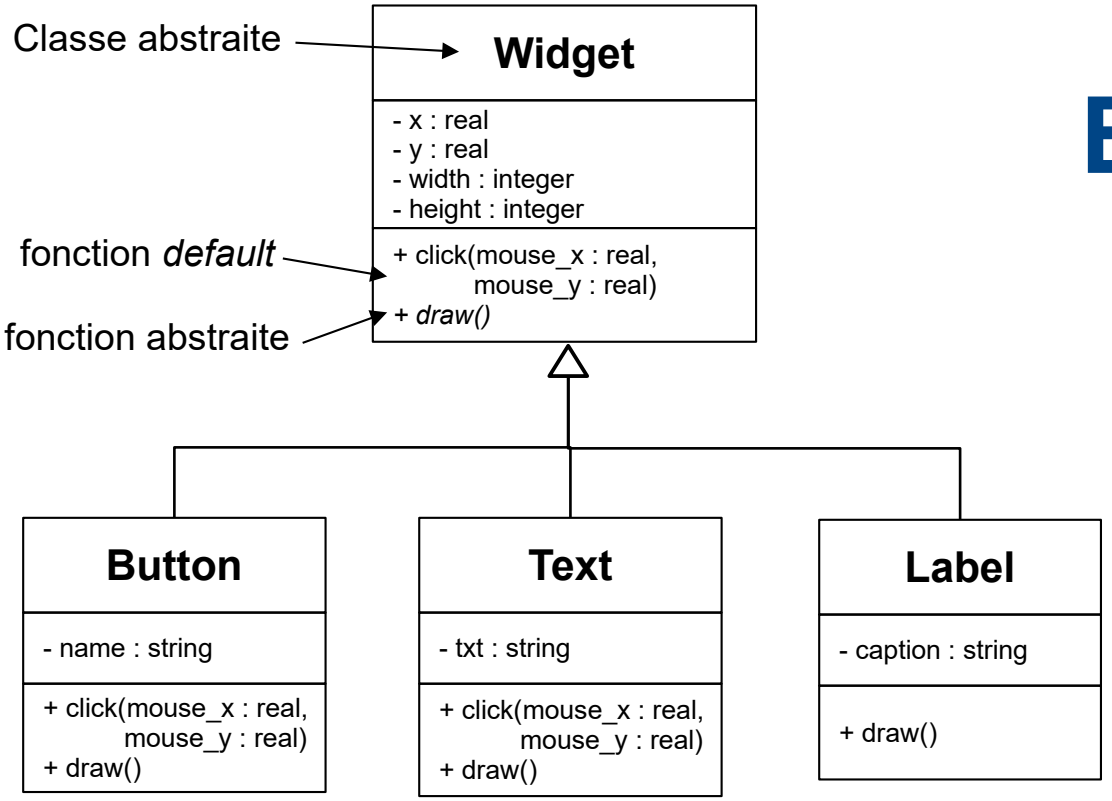


Exemple



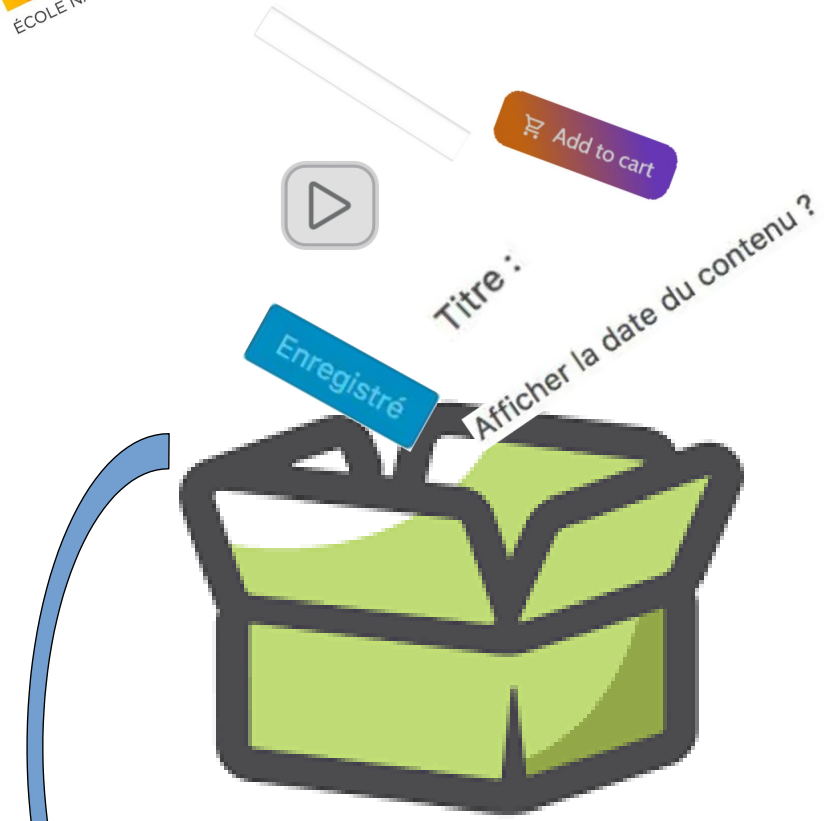
- L'héritage est une sorte de composition++, Button contient un Widget (avec tous ses membres) et est **aussi** un Widget
  - En programmation, cela veut dire que toute référence ou pointeur de type Widget peut être dirigé vers une instance de n'importe quel type de classe dérivées.
    - Un pointeur de type Widget peut pointer un objet de type Button

Exemple

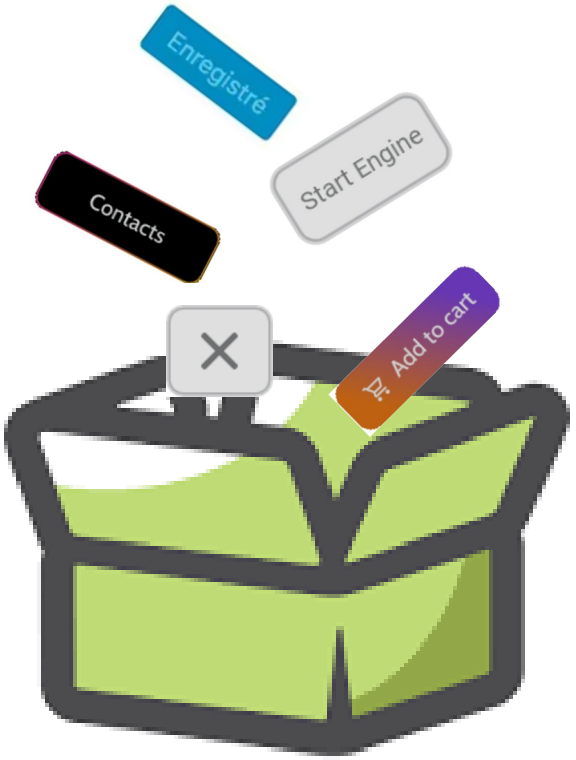


- Les classes dérivées héritent les fonctions de la classe de base et peuvent les redéfinir selon leurs besoins : **Polymorphisme** (plusieurs formes).
  - **draw()**, abstraite dans Widget : dépourvue d'implémentation.
    - Une classe contenant au moins une méthode abstraite est aussi abstraite classe abstraire = non instanciable.
- La classe de base peut fournir des fonctions *default*.
  - **click()**, par défaut elle ne fait rien : implémentation vide.
- Toute classe dérivée assure le comportement déclaré dans la classe de base.

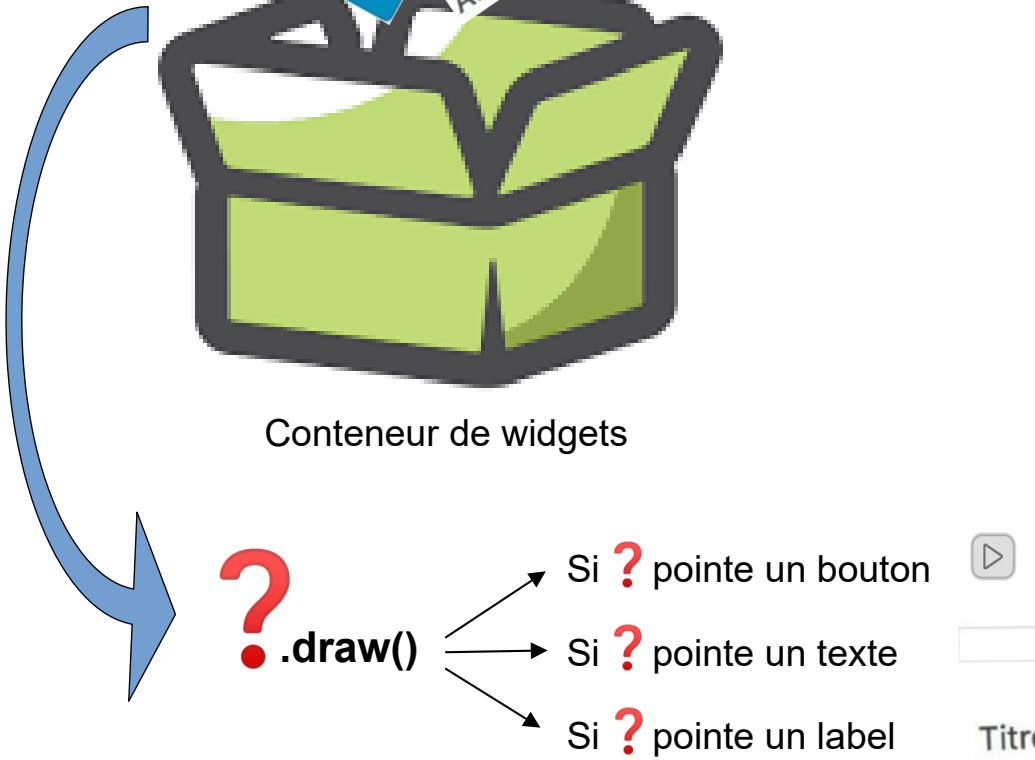
Polymorphisme



Conteneur de widgets



Conteneur de boutons



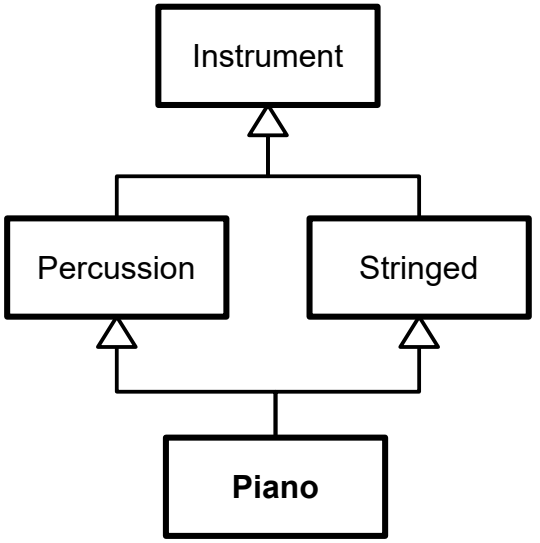
- Le type de ? est découvert au *runtime*.
- Le choix de la méthode est dynamique.

Critiques

- Initialement approche très prometteuse (extensibilité, réutilisation du code, flexibilité...)
- L'expérience a montré que ce n'est pas si rose...
  - l'utilisation de la visibilité protégée (**`protected`**) des données membres de la classe de base fragilise le maintient de l'invariant
    - privilégier la visibilité privée : https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Rh-protected
  - l'héritage exprime une relation **très forte** entre les classes
    - un changement dans la classe de base impacte toutes les classes dérivés (*fragile base class problem*) ;
  - la hiérarchie peut grandir rapidement, ce qui crée des solutions monolithiques dont le code est difficile à relire, comprendre et modifier ;
  - dans des grandes solutions l'apparition de l'héritage multiple (dériver de plusieurs classes de base) est inévitable.

Le problème de l'héritage à répétition

- Appelé aussi "Deadly Diamond of Death".
- Les membres de la classe **`Instrument`** sont présents en double exemplaire dans la classe **`Piano`**, une fois à cause de l'héritage venant de **`Percussion`**, une autre fois à cause de l'héritage en provenance de **`Stringed`**.
- En C++, il peut être résolu en faisant de l'héritage virtuel. Ceci implique de penser son code à l'avance, avoir une architecture définitive à la conception, ce qui en réalité est impossible et va contre tous les discours d'une facile extensibilité du code.
- Dans les langages plus modernes, l'héritage multiple n'existe plus.

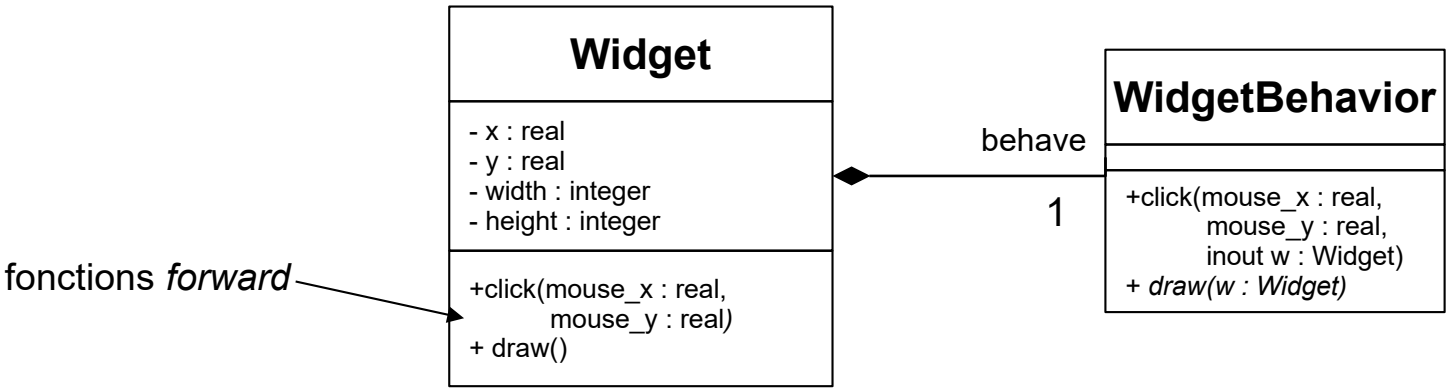


Obtenir des codes plus flexibles (1)

- **Préférer la composition** à l'héritage :
  - concevoir une classe en focalisant sur ce qu'elle fait plutôt que sur ce qu'elle est.
    - Son comportement polymorphe sera obtenu par composition avec un autre type qui déclare le comportement qu'on veut polymorphe.

Obtenir des codes plus flexibles (1)

- **Préférer la composition** à l'héritage :



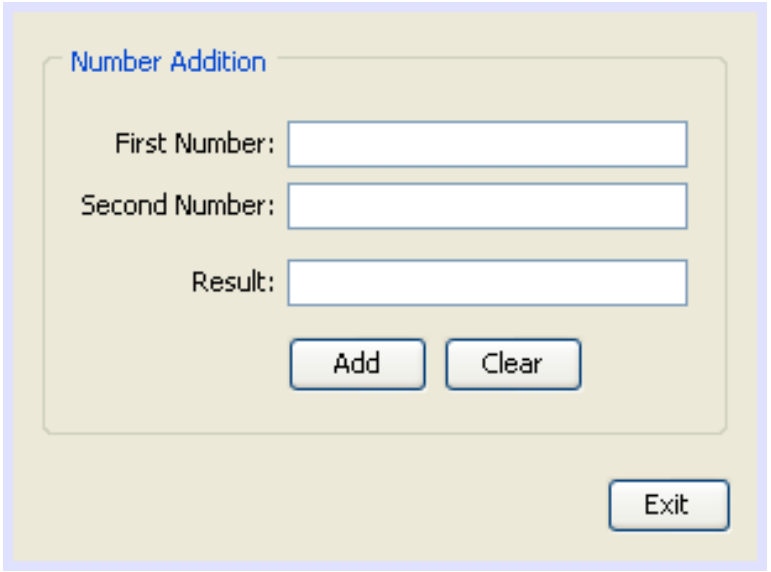
- Les fonctions click() et draw() de Widget, invoqueront celles de WidgetBehavior. Elles s'appellent des fonctions *forward*.
- Il suffit maintenant d'*hériter* de WidgetBehavior pour proposer autant de comportements polymorphes pour le clic et le dessin qu'on veut.
- ***Single responsability principle*** : une classe doit avoir une seule responsabilité
- → classes plus robustes

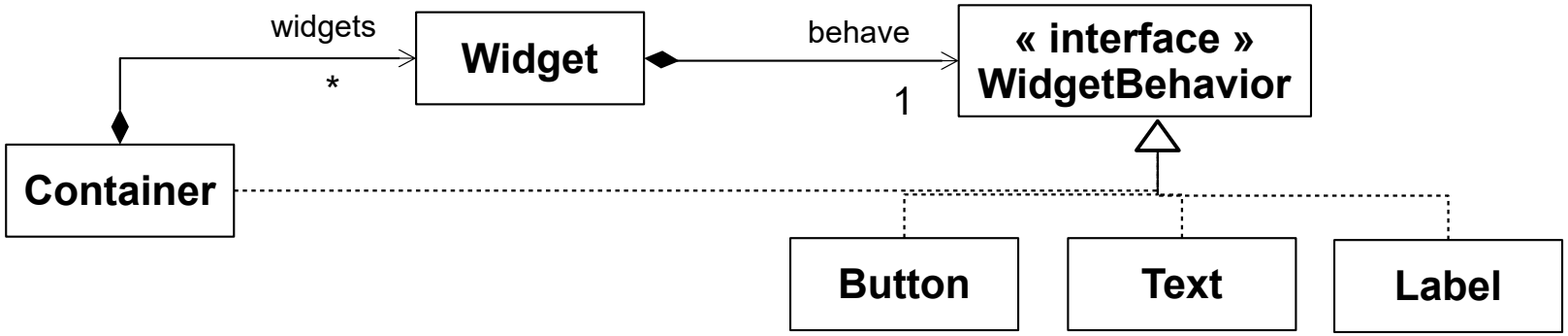
Obtenir des codes plus flexibles (2)

- Éviter les classes de base, utiliser plutôt des **interfaces** :
  https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-abstract
- Une interface :
  - est un type ;
  - ne possède pas d'état (données membres) ;
  - regroupe un ensemble de fonctions abstraites qui décrivent des comportements ;
  - peut avoir des fonctions *default*, qui ont une implémentation concrète ;

Obtenir des codes plus flexibles (2)

- Une interface :
  - ne peut pas être instanciée, elle peut seulement être *implémentée* (ou réalisée) par des classes ;
  - engage toute classe qui l'implémente à fournir une définition au moins pour toute fonction abstraite déclarée dans l'interface
    - la classe qui l'implémente hérite aussi du type de l'interface.
  - une classe peut implémenter plusieurs interfaces sans risquer les problèmes de l'héritage multiple.


Number Addition
First Number:
Second Number:
Result:
Add
Clear
Exit


widgets
*
Widget
behave
1
« interface »
WidgetBehavior
Container
Button
Text
Label

Interfaces dans les langages

- Certains langages ont des mots clefs pour déclarer une interface :
  - **interface** en Java et C#.
- D'autres non :
  - En C++, une interface est une structure sans attributs.
  - En Python, une interface est une classe sans attributs.
- En Rust, on utilise des **Trait**s.

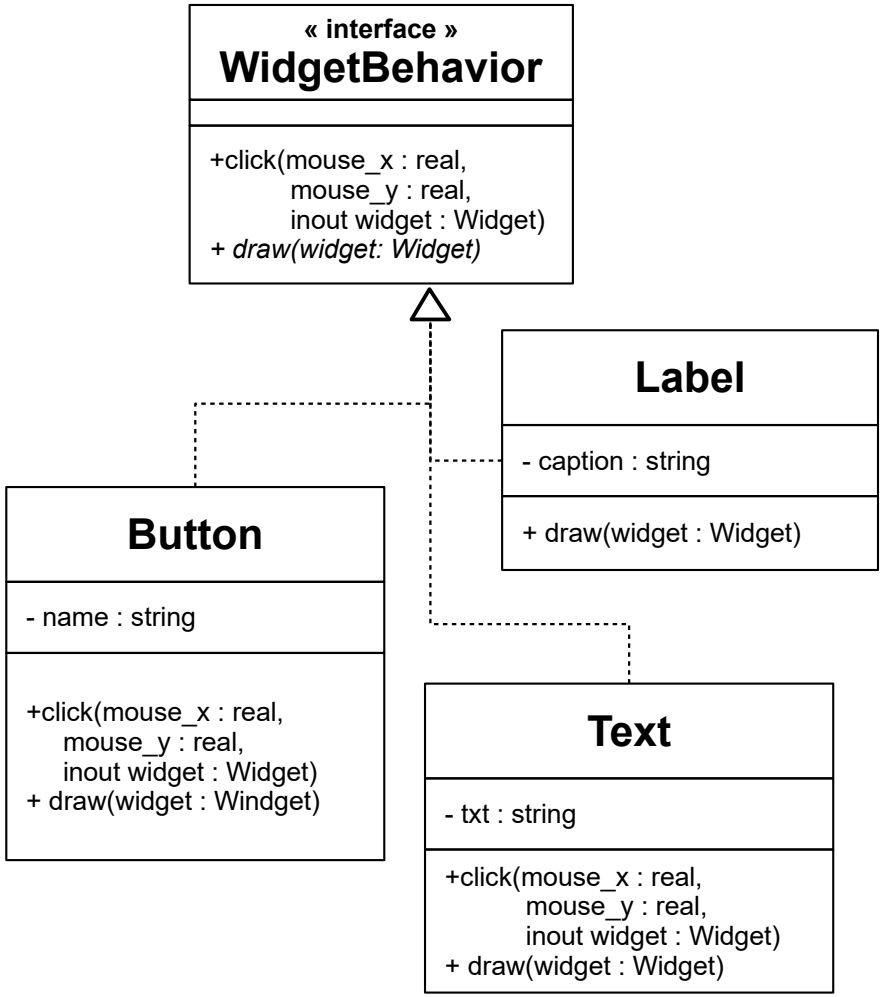
Trait

- Un Trait définit un ensemble de fonctionnalités que différents types (implémentant le Trait) possèdent et partagent.
- Un trait peut être utilisé pour obtenir du polymorphisme :
  - **statique**, la fonction adaptée au type spécifique de la donnée est déterminée dès la compilation (programmation générique) ;
  - **dynamique**, la fonction adaptée au type spécifique de la donnée n'est déterminée qu'au moment de l'exécution (*late binding*, programmation orientée objet).

Mise en place – Interface

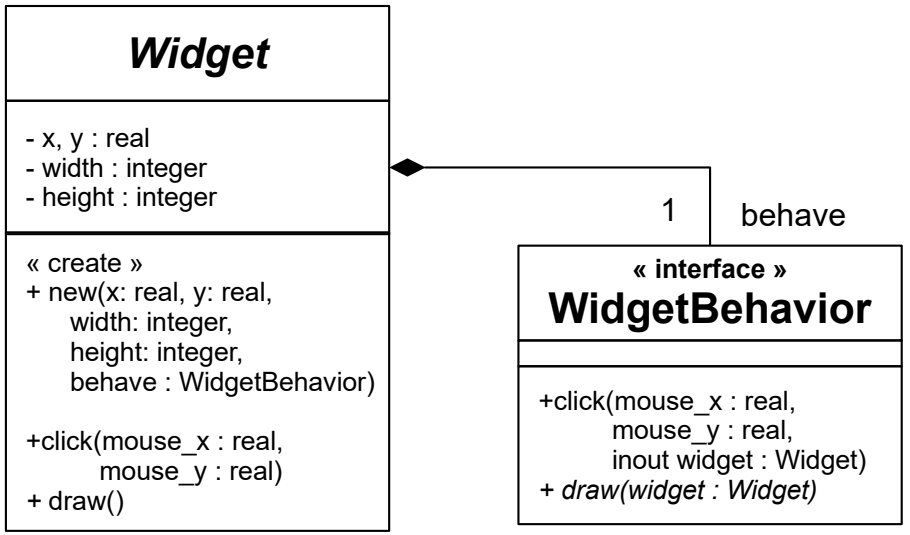
Trait

```
pub trait WidgetBehavior {
    fn click(
        &mut self,
        mouse_x: f32,
        mouse_y: f32,
        widget: &mut Widget,
    ) {
        println!("Ouch! Somebody clicked on me !");
    };
    fn draw(
        &self,
        widget: &Widget,
    );
}
```



Mise en place – Composite

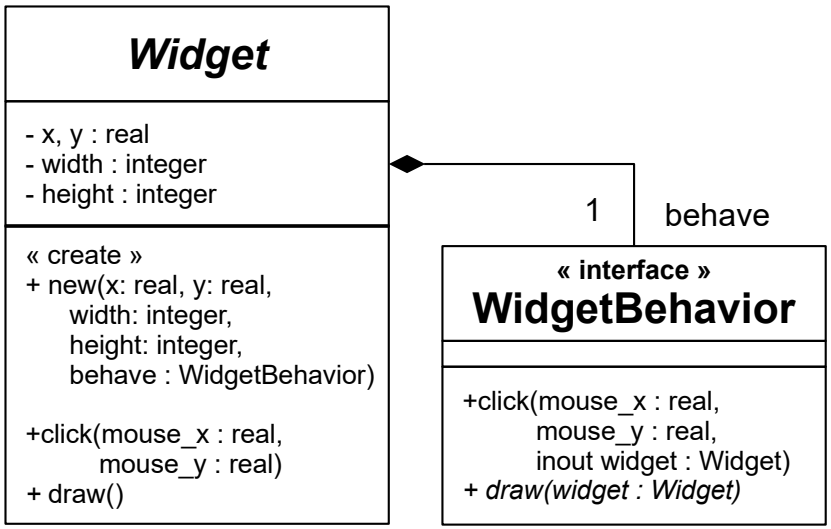
```
pub struct Widget {
    x: f32,
    y: f32,
    width: usize,
    height: usize,
    behave: Option<Box<dyn WidgetBehavior>>,
}

impl Widget {
    pub fn new(
        x: f32,
        y: f32,
        width: usize,
        height: usize,
        behave: Box<dyn WidgetBehavior>,
    ) -> Self {
        Self {x, y,
              width, height,
              Some(behave)
        }
    }

    //continues...
```



- **Box<dyn T>** : pointeur dynamique.
- Le type de la donnée pointée sera déterminé à l'exécution.

Mise en place – Composite

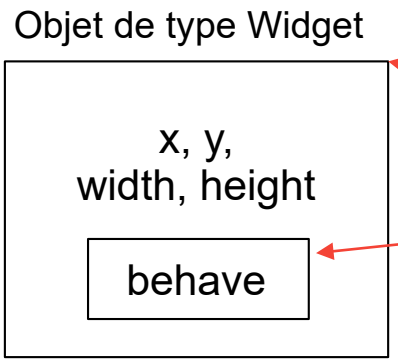
```
pub fn click(
      &mut self,
      mouse_x: f32,
      mouse_y: f32,
  ) {
      if mouse_x >= self.x
        && mouse_x < self.x
                   + self.width as f32
        && mouse_y >= self.y
        && mouse_y < self.y
                   + self.height as f32
    {
        if let Some(mut behave) = self.behave.take()
        {
            behave.click(mouse_x, mouse_y, self);
            self.behave = Some(behave);
        }
    }
  }

  pub fn draw(&self) {
    if let Some(behave) = &self.behave {
        behave.draw(&self);
    }
  }
}
```



- **click()** et **draw()** délèguent à *behave* la gestion du clic et du dessin : fonctions *forward.*

Pourquoi utiliser un Option ?

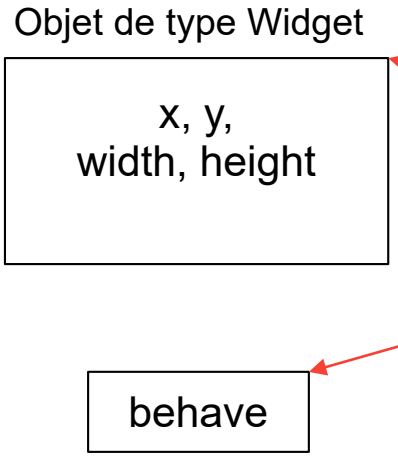
Problème




```
click(&self, widget: &mut Widget, mouse_w: f32, mouse_y: f32)
```

!!! Deux accès simultanés à la même donnée (*behave*), dont un en modification !!!
Impossible en Rust

Solution



On déplace temporairement le WidgetBehavior (à l'aide de la fonction take())

```
click(&self, widget: &mut Widget, mouse_w: f32, mouse_y: f32)
```

On n'accède plus simultanément à la même donnée. Tout va bien !

Implémenter l'interface

```
pub struct Button {
    pub name: String,
}

impl Button {
    pub fn new(name: String) -> Self {
        Self { name }
    }
}

impl WidgetBehavior for Button {
    fn click(
        &mut self,
        mouse_x: f32,
        mouse_y: f32,
        widget: &mut Widget,
    ) {
        println!("Button clicked !"); //...
    }
    fn draw(
        &self,
        widget: &Widget,
    ) {
        println!("Button drawn!");    //...
    }
}
```

Implémentation du trait WidgetBehavior pour le type Button

Les fonctions click() et draw(), déclarées dans WidgetBehavior, sont redéfinies ici. On parle d'***override***.

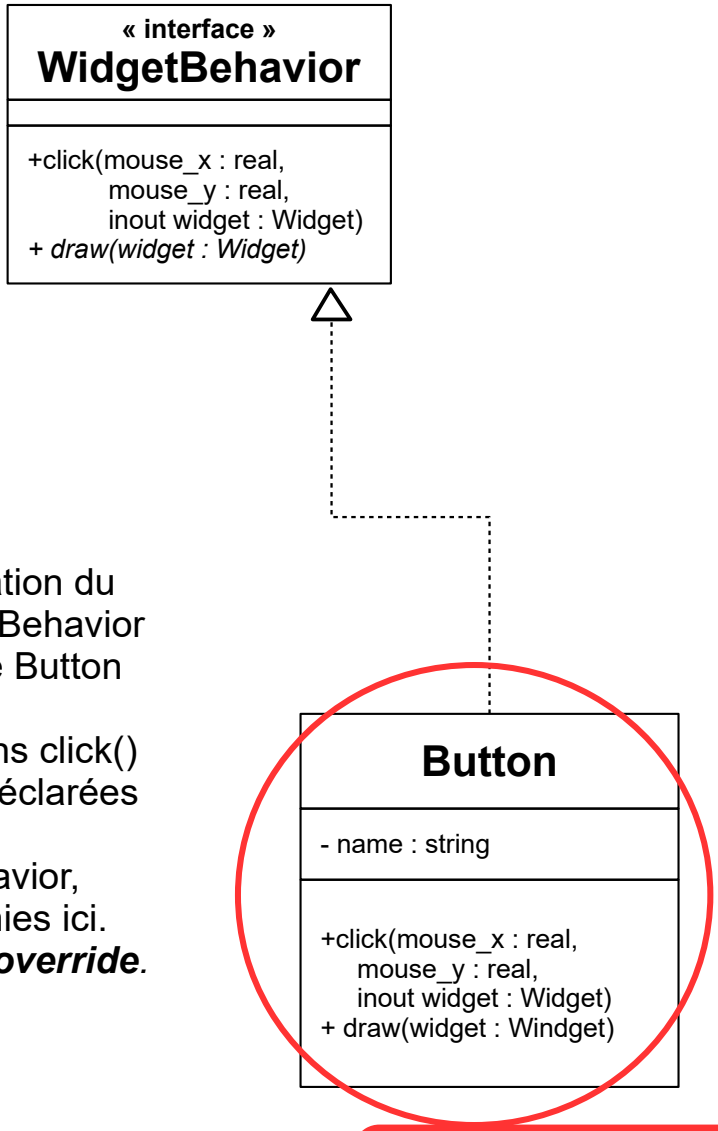
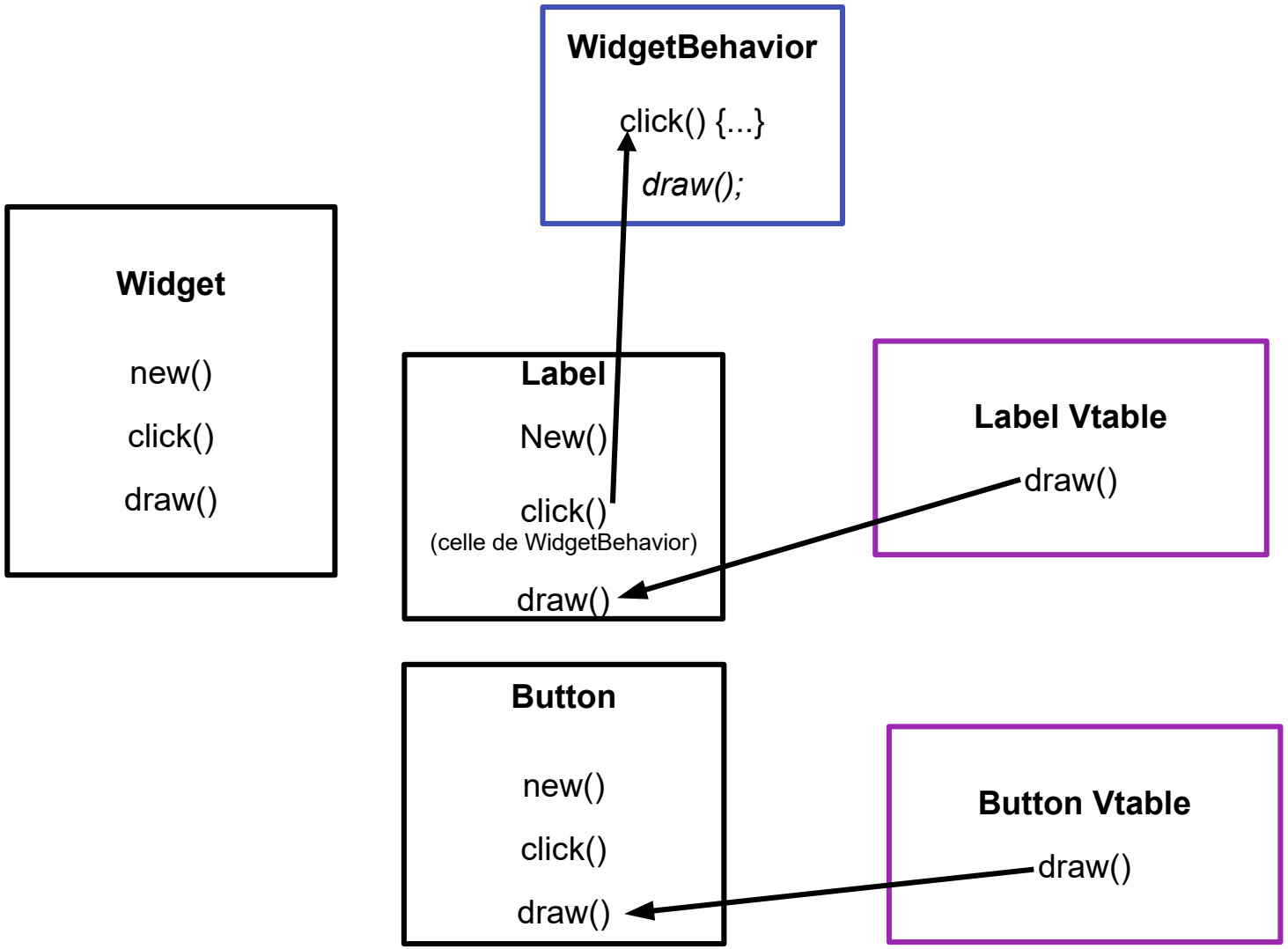


Table virtuelle

- La liaison dynamique est rendue possible grâce à la table virtuelle (*virtual table* ou *Vtable* ou *dispatch table*).
- Elle est implicitement créée pour toute classe contenant au moins une méthode abstraite.
- Elle est un tableau statique créé à la compilation contenant les pointeurs vers les fonctions abstraites.
- Ces pointeurs sont utilisés à l'exécution pour invoquer la fonction appropriée.

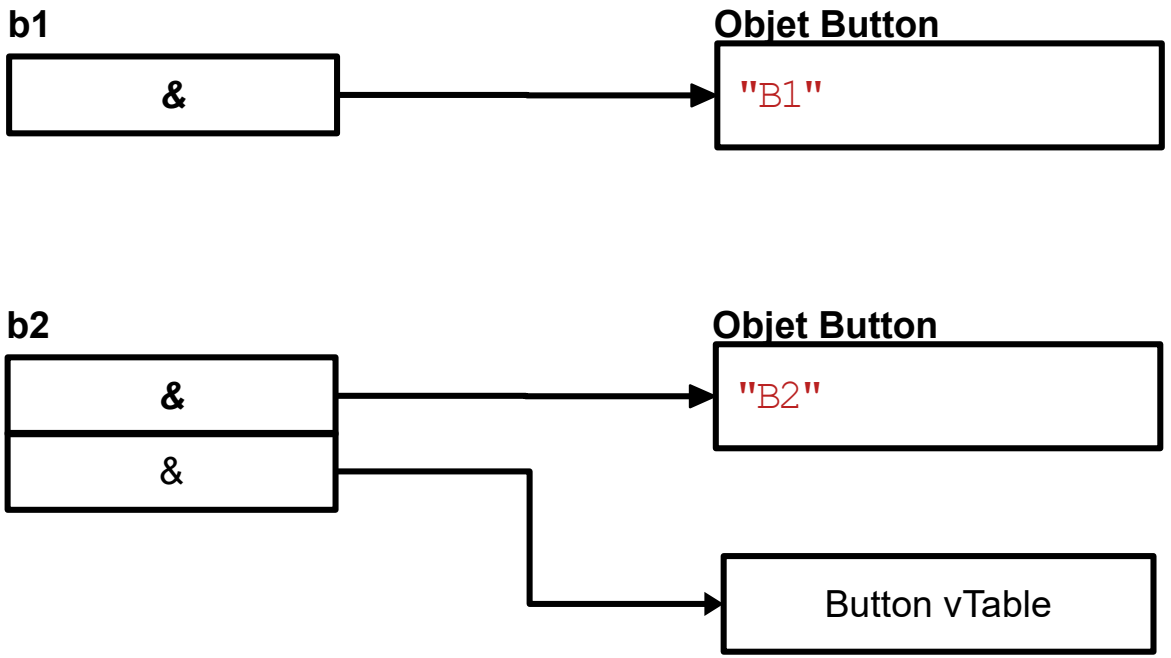
Table virtuelle – exemple (compil)



- Le compilateur crée une table virtuelle pour tout type qui implémente un trait.
- Selon le langage, le pointeur vers cette table est obtenu différemment.
- Dans la majorité des langages OO, il se trouve dans la donnée.
- En Rust, il se trouve dans le pointeur qui désigne la donnée.

En Rust...

```
let b1 = Box::new(Button::new("B1".to_owned()));

let b2 : Box<dyn WidgetBehavior> =
        Box::new(Button::new("B2".to_owned()));
```



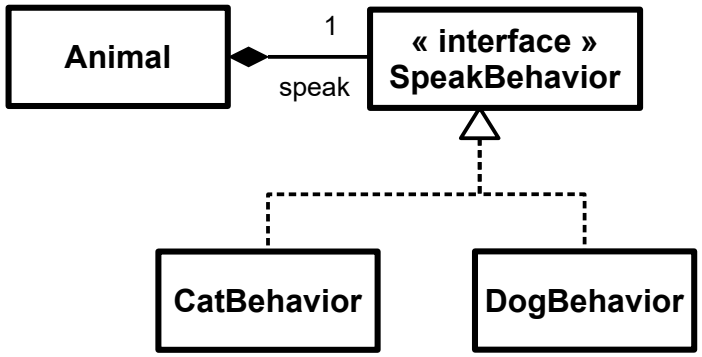
- Le pointeur vers la Vtable est ajouté implicitement au pointeur Box qui désignera une instance de Button ou Label ou Text ou Container.
- On parle, alors de *fat pointer*, parce qu'il conserve à la fois l'adresse de la donnée et l'adresse de sa Vtable.
- Le comportement dynamique n'est pas dans le type mais dans l'usage.

Trois règles à ne pas oublier

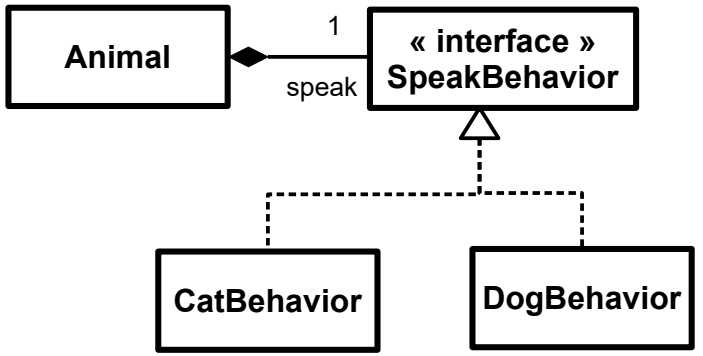
- L'excessive utilisation de l'héritage donne lieu à solutions monolithiques, difficiles à comprendre, maintenir, modifier, corriger, étendre.
  - **Privilégier la composition**
    - Doter une classe d'un comportement polymorphe à travers la composition avec interfaces déclarant le comportement voulu.
    - Ce comportement sera spécifié par les types qui implémenteront les interfaces.
  - Éviter les classe de base, **faire des interfaces**.
  - **Garder la profondeur de l'arbre d'héritage à 1**. Si cela n'est pas possible, toute classe intermédiaire doit être une interface.

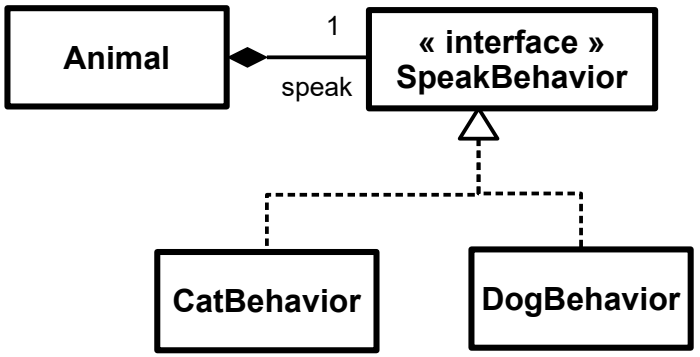
Exercice

- Une classe Animal dotée d'un comportement (le cri) polymorphe par composition avec une interface qui propose ce comportement (une fonction speak())
- Toute classe instanciable qui implémente l'interface SpeakBehavior fournit certainement une définition de la fonction speak()



```rust
pub trait SpeakBehavior {
    fn speak(&self);
}

pub struct Animal {
    name: String,
    speak: Box<dyn SpeakBehavior>,
}

impl Animal {
    pub fn new(
        name: String,
        speak: Box<dyn SpeakBehavior>,
    ) -> Self {
        Self { name, speak }
    }

    pub fn speak(&self) {
        print!("{} says: ", self.name);
        self.speak.speak();
    }
}
```






```rust
pub struct CatBehavior {}

impl SpeakBehavior for CatBehavior {
    fn speak(&self) {
        println!("MEEEEEOOOOOOWWWWWW!!!!!!",);
    }
}

pub struct DogBehavior {}

impl SpeakBehavior for DogBehavior {
    fn speak(&self) {
        println!("WOOOOOOUUUUUUFFFFFF!!!!!!",);
    }
}
```

```rust
fn main() {
    //Create a cat and a dog
    let cat = Animal::new("Hiccup".to_owned(),
                          Box::new(CatBehavior{}));
    let dog = Animal::new("Rex".to_owned(),
                          Box::new(DogBehavior{}));

    println!("\n~~~~~ Cat and Dog speak ~~~~~");
    cat.speak();
    dog.speak();
}
```

```
Hiccup says: MEEEEEOOOOOOWWWWWW!!!!!!
Rex says: WOOOOOOUUUUUUFFFFFF!!!!!!
```

Contexte

```
struct Buffer {
    data: Vec<u8>,
    pos: usize,
}

impl Buffer {
    fn new(data: Vec<u8>) -> Self {
        Self { data, pos: 0 }
    }

    fn next_byte(&mut self) -> Option<u8> {
        if self.pos < self.data.len() {
            let b = self.data[self.pos];
            self.pos += 1;
            Some(b)
        } else {
            None
        }
    }
}
```

- Supposons d'avoir un type Buffer qui fournit une fonction *next_byte()*. Cette fonction renvoie un byte à la fois dans le vecteur du buffer.
- Nous voulons une fonction *read_line()* qui permet de lire tout une ligne ('\n') dans un buffer.

Contexte

```rust
fn read_line(src: &mut Buffer) -> Option<String> {
    let mut bytes = Vec::new();
    while let Some(b) = src.next_byte() {
        bytes.push(b);
        if b == b'\n' { break; }
    }
    if bytes.is_empty() { None }
    else { Some(String::from_utf8_lossy(&bytes).to_string()) }
}

fn main() {
    println!("~~~~ reading buffer ~~~~");
    let mut buffer = Buffer::new(vec![65, 66, 67, 68, 10,
                                      120, 121, 122, 10]);
     if let Some(l) = read_line(&mut buffer) {
        println!("line : {:?}", l);
    }
}
```

```
> cargo run
~~~~ reading buffer ~~~~
line : "ABCD\n"
```

Problème

```
fn read_line_file(src: &mut std::fs::File)
    -> Option<String> {
    let mut bytes = Vec::new();
    use std::io::Read;
    let mut b = 0;
    while src.read_exact(
          std::slice::from_mut(&mut b)).is_ok() {
        bytes.push(b);
        if b == b'\n' {
            break;
        }
    }
    if bytes.is_empty() {
        None
    } else {
        Some(String::from_utf8_lossy(
                        &bytes).to_string())
    }
}
```

- Supposons maintenant de vouloir lire une ligne à partir d'autre chose qu'un Buffer, un file, par exemple. Il faudrait ré-écrire une nouvelle fonction *read_line()* qui cette fois reçoit un fichier en paramètre.
  - Duplication de code.

Solution : polymorphisme

- Il serait préférable d'avoir une seule fonction *read_line()* qui puisse accepter n'importe quel type qui soit capable de fournit un octet à la fois. Bref : qui fournisse une fonction comme *next_byte()*.
- Nous nous servons du polymorphisme.
  - Le trait **ByteReader** déclare la fonction next_byte().
  - La fonction *read_line()* accepte tout paramètre de type T où T est **ByteReader**.

```
trait ByteReader {
    fn next_byte(&mut self) -> Option<u8>;
}

fn read_line<T: ByteReader>(src: &mut T) -> Option<String> {
    let mut bytes = Vec::new();
    while let Some(b) = src.next_byte() {
        bytes.push(b);
        if b == b'\n' {
            break;
        }
    }
    if bytes.is_empty() {
        None
    } else {
        Some(String::from_utf8_lossy(&bytes).to_string())
    }
}
```

Solution : polymorphisme

- Il suffit que les types Buffer et File (et n'importe quel autre type nous voulons introduire) implémentent le trait **ByteReader**.

```
impl ByteReader for std::fs::File {
    fn next_byte(&mut self) -> Option<u8> {
        use std::io::Read;
        let mut b = 0;
        match self.read_exact(
                    std::slice::from_mut(&mut b)) {
            Ok(()) => Some(b),
            Err(_) => None,
        }
    }
}
```

```
struct Buffer {
    data: Vec<u8>,
    pos: usize,
}

impl Buffer {
    fn new(data: Vec<u8>) -> Self {
        Self { data, pos: 0 }
    }
}

impl ByteReader for Buffer {
    fn next_byte(&mut self) -> Option<u8> {
        if self.pos < self.data.len() {
            let b = self.data[self.pos];
            self.pos += 1;
            Some(b)
        } else {
            None
        }
    }
}
```

Solution : polymorphisme statique

- Partout où on aura besoin de lire une ligne, on pourra utiliser la fonction *read_line()* même si la source de bytes est différente.
  - Pas de duplication de code.
- Ce polymorphisme est **statique**. Le compilateur détermine le type passé à la fonction à la compilation est génère le code de la fonction *read_line()* déclinée explicitement pour ce type. Gain en performance.

```
fn main() {
    println!("~~~~ reading buffer ~~~~");
    let mut buffer = Buffer::new(vec![65, 66, 67, 68, 10, 121, 122, 10]);
    if let Some(l) = read_line(&mut buffer) {
        println!("line : {:?}", l);
    }

    println!("~~~~ reading file ~~~~");
    if let Ok(mut file) = std::fs::File::open("Cargo.toml") {
    if let Some(l) = read_line(&mut file) {
        println!("line : {:?}", l);
    }
}
```

ÉCOLE NATIONALE D'INGÉNIEURS DE BREST

Langages Orientés Objet

Héritage et
Polymorphisme dynamique