Objective design meets Rust

Bartłomiej Kuras

Who am I

Bartłomiej Kuras

- 3 years in *Nokia* as C++ developer doing BTS stuff
- half an year in *DeLaval* as C++ developer doing near hardware stuff
- half an year in *Imagination* as C++ developer doing GPU simulator
- 2 years in *CreditSuisse* as C++ developer estimating credit risk
- since October in *Anixe* as Rust developer doing prizing calculation

Table of contents

Objective Programming Common definition

$$\text{OOP} = \text{Classes} + \text{Inheritance}$$

+ Polymorphism
+ Casting
+ Hermetization
+ Mixins
+ …

doesn't like it

Objective Programming Gang of four

Object-oriented programs are made up of objects. An object packages both data and the procedures that operate on that data. The procedures are typically called methods or operations.

- structures with fields as data
- impl blocks for structures as methods
- variables which are structures instances

We are already there.

Objective Programming Gang of four

```
struct Vec2d {
    x: f32,
    y: f32,
}

impl Vec2d {
  fn translate(&mut self, other: Vec2d) {
    self.x += other.x;
    self.y += other.y;
  }

  // ...
}
```

Objective Programming Wikipedia

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.

Objective Programming Wikipedia

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.

Features

- encapsulation
- composition, inheritance, and delegation
- polymorphism

SOLID GRASP

Objective Programming My definition

Programming paradigm focusing on hiding data behind behaviour.

Core features:

- types/classes/representations
- interfaces/concepts/contracts
- objects/instances
- polymorphism (possibly compile-time!)
- hermetization

Building blocks

Building blocks

Building blocks Types

```
struct Foo {
    f1: String,
    pub f2: bool,
    pub(crate) f3: FooBar,
}

impl Foo {
  fn method(&self) {
  }
}
```

```
enum Bar {
    B1,
    B2(Foo),
    B3{ f: Foo, u: u32 }
}

impl Bar {
  fn method(&self) {
  }
}
```

Buildng blocks Traits

```
trait Filo {
    fn push(&mut self, item: u32);
    fn pop(&mut self) -> Option<u32>;
}

impl Filo for Vec<u32> {
    fn push(&mut self, item: u32) {
        self.push(item)
    }

    fn pop(&mut self) -> Option<u32> {
        self.pop()
    }
}
```

Building blocks impl Trait aka CT Polymorphism

```
fn foo(x: &mut impl Filo) {
    x.push(5)
}

fn bar(x: &mut impl Filo) {
    println!("{:?}", x.pop())
}

fn main() {
    let mut v = vec![];
    foo(&mut v);
    bar(&mut v);
}
```

Building blocks dyn Trait aka RT Polymorphism

```
fn foo(x: &mut dyn Filo) {
    x.push(5)
}

fn bar(x: &mut dyn Filo) {
    println!("{:?}", x.pop())
}

fn main() {
    let mut v: Box<dyn Filo> = Box::new(vec![]);
    foo(&mut *v);
    bar(&mut *v);
}
```

Building blocks impl Trait as result

```
fn new_filo() -> impl Filo {
    vec![2, 3]
}

fn main() {
    let mut filo = new_filo();
    // println!("{}", filo.is_empty());
    println!("{:?}", filo.pop());
    println!("{:?}", filo.pop());
    println!("{:?}", filo.pop());
}
```

OOP in Rust

Guideline for being OOPish in Rust:

- define your interfaces as Traits
- define your classes as Structs implementing your Traits
- return ‘impl Trait‘/boxed ‘dyn Trait‘ wherever it makes sense
- take args as ‘impl Trait‘/‘dyn Trait‘ wherever it makes sense

Inheritance

Inheritance

Inheritance Implementing interfaces

```
class Filo {
public:
    virtual void push(int x) = 0;
    virtual int pop() = 0;
};

class VecFilo: public Filo {
public:
    virtual void push(int x) override { data.push_back(x); }
    virtual void pop() override {
        int last = data.back();
        data.pop_back();
        return last;
    }

private:
    std::vector<int> data;
};
```

Inheritance Implementing interfaces

```
trait Filo {
    fn push(&mut self, item: u32);
    fn pop(&mut self) -> Option<u32>;
}

impl Filo for Vec<u32> {
    fn push(&mut self, item: u32) {
        self.push(item)
    }

    fn pop(&mut self) -> Option<u32> {
        self.pop()
    }
}
```

Inheritance Extending functionality

```
class VecFiloWithPop2: public VecFilo {
public:
    std::tuple<int, int> pop2() {
        std::tuple { pop(), pop() }
    }
};
```

```
trait Pop2 {
    fn pop2(&mut self) -> (Option<u32>, Option<u32>);
}

impl Pop2 for Vec<u32> {
    fn pop2(&mut self) -> (Option<u32>, Option<u32>) {
        (self.pop(), self.pop())
    }
}
```

Inheritance Extending functionality

```
trait Pop2 {
    fn pop2(&mut self) -> (Option<u32>, Option<u32>);
}

impl<T: Filo> Pop2 for T {
    fn pop2(&mut self) -> (Option<u32>, Option<u32>) {
        (self.pop(), self.pop())
    }
}
```

Problem: there may be only one existing implementation for every type (direct or indirect).
Solution: specializations incomming! (RFC 1210)

Inheritance Reusing implementation

```
class FunnyVec: public std::vector<int> {
private:
    int funny_field;
};
```

Sometimes bad designs will fail faster in Rust

Catherine West @ Rustconf, QotW 252

Inheritance Reusing implementation

Not so bad scenario - vector and funny_field are independent

```
class ItemAccessor: public std::vector<int> {
public:
    int get_item() {
        return (*this)[index];
    }

private:
    size_t index;
};
```

Inheritance Reusing implementation

```cpp
class ItemAccessor {
public:
    int get_item() {
        return data[index];
    }

private:
    std::vector<int> & data;
    size_t index;
};
```

```rust
struct ItemAccessor<'a> {
    data: &'a Vec<u32>,
    index: usize,
}

impl<'a> ItemAccessor<'a> {
    fn get_item(&self) -> u32 {
        self.data[self.index]
    }
}
```

Inheritance Reusing implementation

```
class AccumulatedVec: public std::vector<int> {
public:
    void clear() { ... }
    void insert() { ... }
    std::vector<int>::iterator erase(
        std::vector<int> iterator pos) { ... }
    ...
private:
    int sum;
};
```

And then in 2011 someone adds ‘emplace‘...

Inheritance Reusing implementation

Using inheritance to reuse part of previous implementation is in most cases bad by design. This approach reuses not only existing implementation details, but also everything what will came up in future and cannot be predicted.

Prefer encapsulation over inheritance.

Inheritance Deref antipatern

```
struct ItemAccessor {
    data: Vec<u32>,
    index: usize,
}

impl Deref for ItemAccessor {
    type Target = Vec<u32>;

    fn deref(&self) -> &Vec<u32> {
        &self.data
    }
}
```

Problems:

- semantics - deref means "dereferencing to", not "is a"
- not working with traits

Polymorphism

Polymorphism

Polymorphism Box vs static dispatch

```
fn foo(x: &mut dyn Filo) {
    x.push(5)
}

fn bar(x: &mut dyn Filo) {
    println!("{:?}", x.pop())
}

fn main() {
    let mut v: Box<dyn Filo> = Box::new(vec![]);
    foo(&mut *v);
    bar(&mut *v);
}
```

Polymorphism Box vs static dispatch

```
fn foo(x: &mut impl Filo) {
    x.push(5)
}

fn bar(x: &mut impl Filo) {
    println!("{:?}", x.pop())
}

fn main() {
    let mut v: Box<dyn Filo> = Box::new(vec![]);
    foo(&mut *v);
    bar(&mut *v);
}
```

error[E0277]: the size for values of type ‘dyn Filo‘ cannot be known at compilation time

Polymorphism Box vs static dispatch

```
fn foo(x: &mut impl Filo) {
    x.push(5)
}

fn bar(x: &mut impl Filo) {
    println!("{:?}", x.pop())
}

fn main() {
    let mut v: Box<dyn Filo> = Box::new(vec![]);
    foo(&mut v);
    bar(&mut v);
}
```

error[E0277]: the trait bound ‘std::boxed::Box⟨dyn Filo⟩: Filo‘ is not satisfied

Polymorphism Box vs static dispatch

```
use std::ops::DerefMut;
impl Filo for Box<dyn Filo> {
    fn push(&mut self, item: u32) {
        self.deref_mut().push(item)
    }

    fn pop(&mut self) -> Option<u32> {
        self.deref_mut().pop()
    }
}
```

Pros:

- transparent

Cons:

- boilerplate
- heap allocation

Polymorphism Dispatch via enum

```
enum DynamicFilo {
    Vec(Vec<u32>),
    List(LinkedList<u32>),
}

impl Filo for DynamicFilo {
    fn push(&mut self, item: u32) {
        match self {
            DynamicFilo::Vec(v) =>
                Filo::push(v, item),
            DynamicFilo::List(l)  =>
                Filo::push(l, item),
        }
    }

    fn pop(&mut self) -> Option<u32> { /* ... */ }
}
```

Pros:

- everything on stack

Cons:

- more boilerplate
- difficult to be maintained
- not extensible

Plymorphism Dispatch via enum

https://crates.io/crates/enum_dispatch

```
#[enum_dispatch]
enum DynamicFilo {
    Vec(Vec<32>),
    List(LinkedList<u32>),
}

#[enum_dispatch(DynamicFilo)]
trait Filo {
    fn push(&mut self, item: u32);
    fn pop(&mut self) -> Option<u32>;
}
```

Pros:

- everything on stack
- reduced boilerplate

Cons:

- still need to maintain
- not extensible

Thank you

Discussion Time