The fault in our traits

& the borrow checker

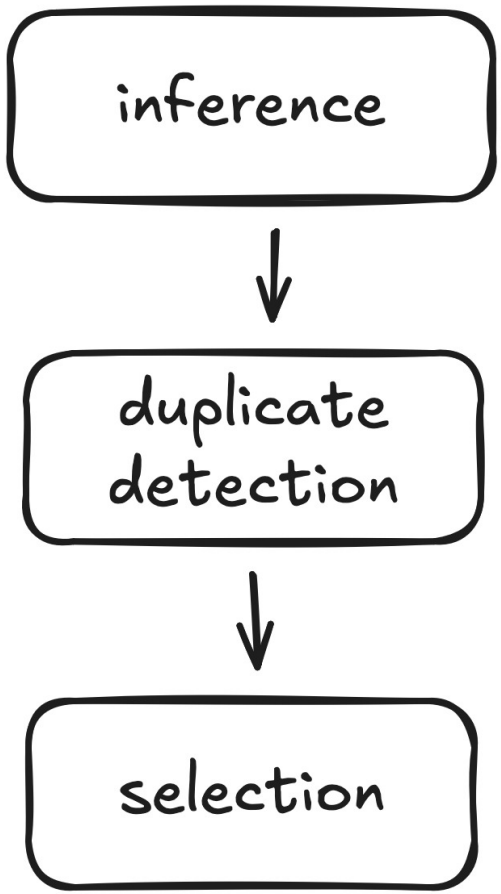
Aman Sharma

Aftershoot Inc.

January 15, 2026

Culling is selecting of ?good? images. The definition of good/bad is judged by multiple factors, but for the sake of this presentation we take every image, then run ML models on them get scores, and run algorithm on them to get selection/rejections.

A general overview of culling pipeline would look something this, different pipeline have difference in the steps in between, but the overall overview will look like this



Now how would you encode this into a trait based system? Let's start with an image:

```rust
#[derive(Clone, Derivative, serde::Serialize)]
#[derivative(Debug)]
pub struct PartialImage<ICnstr: PartialImageConstructor> {
    pub(crate) global_id: Uuid,
    pub(crate) local_id: i64,
    pub(crate) path: PathBuf,
    #[derivative(Debug = "ignore")]
    #[serde(skip_serializing)]
    pub(crate) embedding: Array1<f32>,

    pub(crate) timestamp: Option<f64>,
    pub(crate) image_width: Option<i32>,
    pub(crate) image_height: Option<i32>,

    pub(crate) face_batch: PartialFaceBatch<ICnstr>,
    pub(crate) properties: ICnstr::PartialImageProperties,
    pub(crate) scores: ICnstr::PartialImageScores,
}       Aman Sharma, 2 years ago
```

Now how would you share this structure across pipeline

```rust
pub(crate) trait PartialImageConstructor: PartialFaceConstructor
where
    for<'de> Self::PartialImageScores: serde::Deserialize<'de>,
    for<'de> Self::PartialImageProperties: serde::Deserialize<'de>,
{
    type ImageErr: error_stack::Context;
    type PartialImageScores: Send + Clone + Debug;
    type PartialImageProperties: Send + Clone + Debug;

    fn create_from_fresh(      Aman Sharma, 2 years ago
        core: Core,
        cradle: &Self::Cradle,
    ) -> error_stack::Result<PartialImage<Self>, Self::ImageErr>;

}
```

Now you must do it again with the faces

```rust
#[derive(Clone, Derivative, serde::Serialize)]
#[derivative(Debug)]
pub(crate) struct PartialFaceImportant<FCnst: PartialFaceConstructor> {
    pub(crate) id: Uuid,
    pub(crate) bounding_box: Rect,
    pub(crate) scores: FCnst::PartialFaceScores,
    #[derivative(Debug = "ignore")]
    #[serde(skip_serializing)]
    pub(crate) embedding: Array1<f32>,
    pub(crate) importance: Importance,
}
```

Aman Sharma, 2 years ago

Faces need to be shared among pipeline right?

```rust
pub(crate) trait PartialFaceConstructor
where
    Self: Sized,
    for<'de> Self::PartialFaceScores: serde::Deserialize<'de>,
{
    type PartialFaceScores: Send + Clone + Debug + serde::Serialize;
    type Cradle: CradleFuxxBlur + Sync;
    type FaceErr: error_stack::Context;

    fn create_from_fresh(
        fresh_face: FreshFace,
        parent_image: &Mat,
        cradle: &Self::Cradle,
    ) -> error_stack::Result<PartialFaceImportant<Self>, Self::FaceErr>;
}
```

now can we write any code about the real logic?

There are some scores that are dependent on duplicate detection, after which selection runs, to encode this into the type system, we must create another type for image.

```rust
#[derive(Derivative, serde::Serialize, Clone)]
#[derivative(Debug)]
pub struct CompleteImage<IFiller: PartialImageFiller> {      Aman Sharma, 2
    pub(crate) local_id: i64,
    pub(crate) global_id: Uuid,
    pub(crate) path: PathBuf,
    pub(crate) timestamp: Option<f64>,
    #[derivative(Debug = "ignore")]
    #[serde(skip_serializing)]
    pub(crate) embedding: Array1<f32>,
    pub(crate) image_width: Option<i32>,
    pub(crate) image_height: Option<i32>,
    pub(crate) buffer_height: i32,
    pub(crate) buffer_width: i32,
    pub(crate) face_batch: CompleteFaceBatch<IFiller>,
    pub(crate) properties: IFiller::CompleteImageProperties,
    pub(crate) scores: IFiller::CompleteImageScores,
}
```

lets add a trait so we can share this across pipeline

```rust
pub(crate) trait PartialImageFiller: PartialFaceFiller + PartialImageConstructor {      Aman S
    type CompleteImageScores: Debug + serde::Serialize + Clone;
    type CompleteImageProperties: Debug + serde::Serialize + Clone;
    type ImageFiller<'s>: Debug
    where
        Self: 's;
    type Settings: Debug;

    fn fill<'s>(
        partial_image: PartialImage<Self>,
        utils: Self::ImageFiller<'_>,
    ) -> CompleteImage<Self>;

    fn finalize(complete_image: CompleteImage<Self>, settings: &Self::Settings) -> FinalImage;
}
```

This in term made very brittle and hard to reason about, any change and addition to the feature or code required writing something horrendeous like this, or simply use a dozen of macros and work in together.

```rust
impl<ICnstr, F> Pilot for FaceDupNormalOptIn<ICnstr, F>
where
    ICnstr: PartialImageConstructor,
    <ICnstr as PartialFaceConstructor>::PartialFaceScores:
        FaceScoresFaceDupFuxx + FaceScoresFuxx + FaceScoresImportanceChangeFuxx,
    <ICnstr as PartialImageConstructor>::PartialImageScores:
        ImageScoresFuxxFaceDup + ImageScoresFuxxPerson,
    <ICnstr as PartialImageConstructor>::PartialImageProperties: ImagePropsFuxxFaceDup,
    F: Fn(&PartialImage<ICnstr>) -> ndarray::Array1<f32>,
{
    type Input<'i> = Vec<PartialImage<ICnstr>>;

    type Output<'o> = Vec<Vec<PartialImage<ICnstr>>>;
```

Even with the macros, composing features was difficult it is *also* because of the how quickly changes were made in the pipeline, and also basically the issue being of data drilling, as in data that is created on end of a program is required on the other end of program, also testing was impossible.

The core reason of creating any time of encapsulation is to have less code duplication, traits are good for code generation.

But they force you to think in term of little small pieces of data/methods, that can be accessed from within the traits. this severly restricts with what a computer can do, it primarily restricts things that look at two ?random? pieces of data, that might not have any explicit relationship in the domain model but an implicit relationship at runtime, this puts a bizzare restrictioin on a given software. that is how if you want to do things within an pipeline, everything works fine-ish, but accross even the same logic need to respresented again and again.

What's the solution??

We can flip the encapsulation boundary around the feature(s) and not the domain model itself.

instead of having traits that define methods and GATs, we directly do the thing that we want to do.

```rust
pub struct CullingGlobalState {
    pub scores: ImageScores,
    pub properties: ImageProperties,
    pub images: Vec<Image>,
}

pub struct ImageScores {
    pub blur: Vec<f32>
}

pub struct ImageProperties {
    pub is_selected: Vec<bool>
}

```

this kind of similar to how a ECS would work, this makes the code easy to reason about, and this kind of encapsulation should not bottleneck the application development.

Rust was created as a replacement of CPP. "Compile-time hiearchy of encapsulation that matches the domain model" - Casey Muratori