Can Data-oriented-design be Improved?

OLLIVIER ROBERGE

Cppcon
The C++ Conference

2023

October 01 - 06

Can data-oriented-design be improved?

Ollivier Roberge

Collège Jean-de-Brébeuf

A brief historic

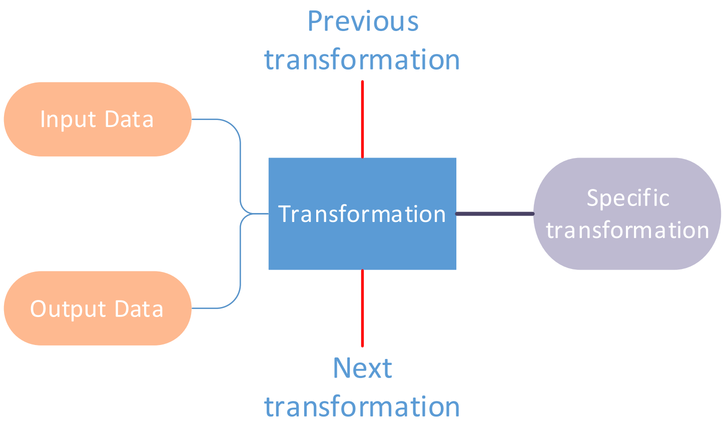
- 1960s
  - FORTRAN
  - Lisp
- 1980s
  - OOP
  - Structured programming
  - Modules
- 2000s
  - Template metaprogramming
  - Concurrency
- 2020s
  - ???

What is DoD about?

- DoD (“Data oriented design”)
- Not about cache lines, nor struct layout (at its core)
- From wikipedia: ”As a design paradigm, data-oriented-design focuses on optimal transformations of data and focuses on modelling programs as transforms.”

Minimalist definition of DoD

$$Data_{Output} = F(Data_{input})$$

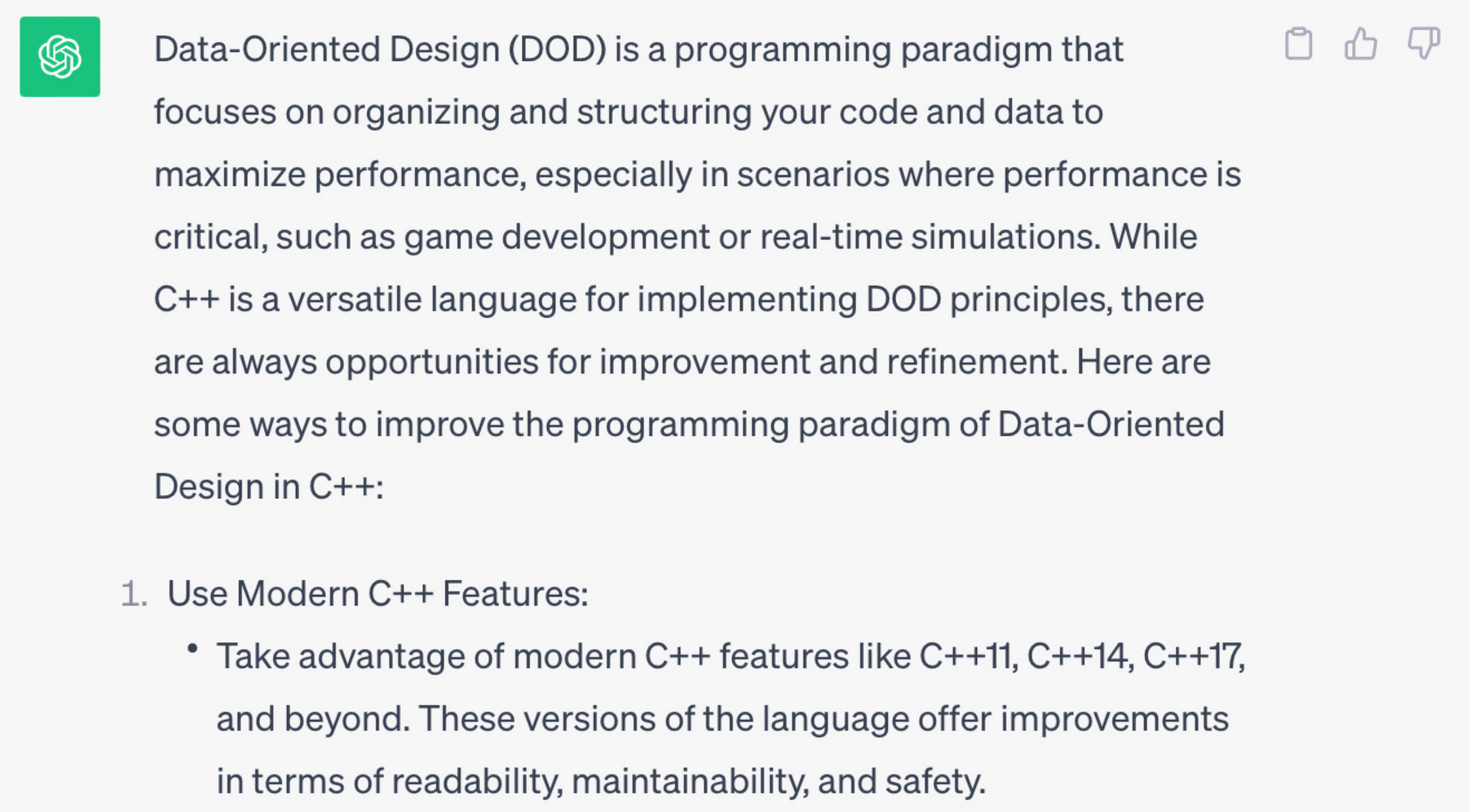


How DoD is used in actual code

- Platform specific code
- Procedural/imperative code
- Problem/dataset specific code
- Hand-optimize cache lines and struct layout

How can we improve it?

- We could ask ChatGPT…



Data-Oriented Design (DOD) is a programming paradigm that focuses on organizing and structuring your code and data to maximize performance, especially in scenarios where performance is critical, such as game development or real-time simulations. While C++ is a versatile language for implementing DOD principles, there are always opportunities for improvement and refinement. Here are some ways to improve the programming paradigm of Data-Oriented Design in C++:

1. Use Modern C++ Features:
   - Take advantage of modern C++ features like C++11, C++14, C++17, and beyond. These versions of the language offer improvements in terms of readability, maintainability, and safety.

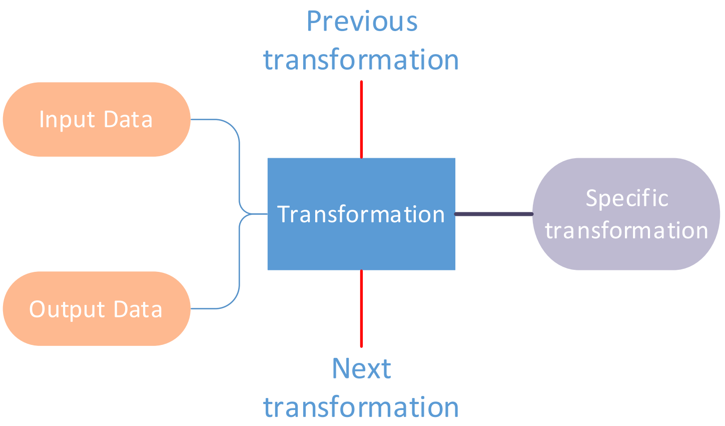
That’s cool,
but it won’t get us very far…

How can we improve it? (second try)

- At its core DoD is just:

$$Data_{Output} = F(Data_{input})$$

- … With a heavy focus on the data.
- What if we looked at the opposite philosophy?



The opposite philosophy

Data oriented code

- Platform specific code
- Procedural/imperative code
- Problem/dataset specific code
- Hand-optimize cache lines and struct layout

Opposite philosophy

- Cross platform code
- Declarative code
- Generic code
- Delegates cache lines and struct layout to the compiler

The opposite philosophy

Data oriented code

- Platform specific code
- Procedural/imperative code
- Problem/dataset specific code
- Hand-optimize cache lines and struct layout

Opposite philosophy

- Cross platform code
- Declarative code
- Generic code
- Delegates cache lines and struct layout to the compiler

The “opposite philosophy” sounds a lot like functional programming.

What is functional programming about?

- It’s about chaining functions to make programs:

$$H\left(G\left(F\left(Data_{input}\right)\right)\right)$$

What is functional programming about?

- It’s about chaining functions to make programs:

$$H\left(G\left(F(Data_{input})\right)\right)$$

- If you squeeze them together then we could have:

What is functional programming about?

- It’s about chaining functions to make programs:

$$H\left(G\left(F(Data_{input})\right)\right)$$

- If you squeeze them together the we could have:

- And your program needs to do something:

$$Data_{Output} = F(Data_{input})$$

Data oriented design

=

Functional programming

On the ”equality”…

- DoD and functional programming may have strong differences
- … But they also have strong similarities
- Much more than to OOP
    - See Java programming

Still on the “equality”

- Extremes are to avoid
- All in on DoD is besides the point
- All in on functional may not be desirable
  - See Ben Deane’s ”Applicative: The Forgotten Functional Pattern” from Wednesday
  - Would be no different than rewriting our programs in Haskell or some Lisp flavor

What is is available (in C++) for FP

- In runtime code
  - Lambdas
  - STL algorithms
  - Ranges
  - Functions pointers
- In compile-time code
  - Types are data (ex: metaprogramming)
- We can apply ideas from functional programming in C++

Introducing “transformation-oriented design”

- The concept defined here will be referred as “transformation-oriented design”
  - TOD for brevity's sake
- We could, in a nutshell, define it as compiler level functional programming.
- Focus must be kept on transformations (“functions”)

How can we improve it? (third try)

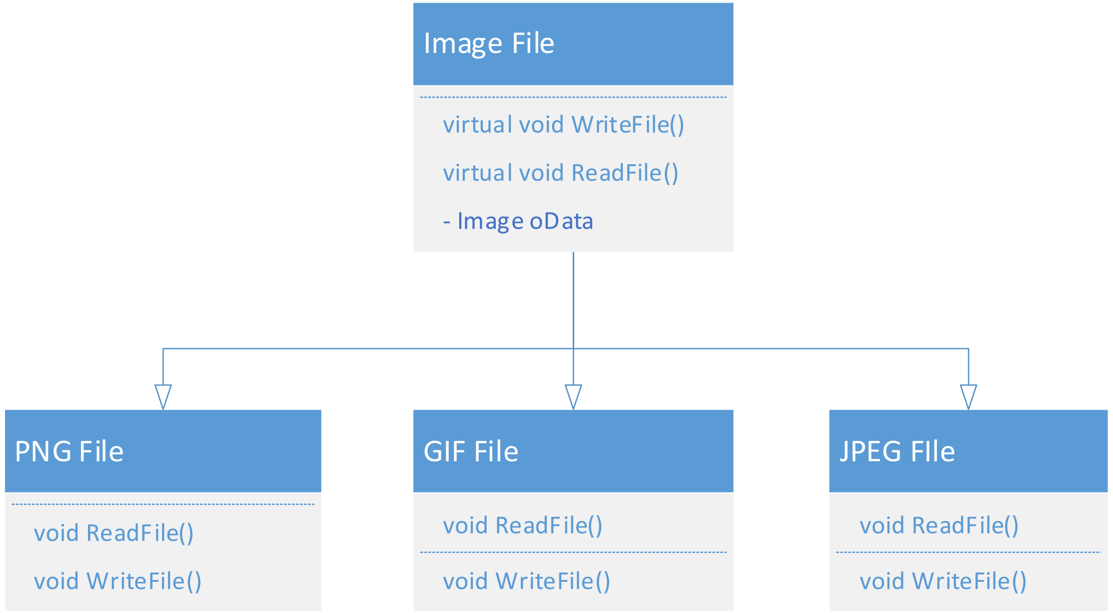
- What if we looked at the opposite of our concept?
- It worked one time, why not two?
- OOP may not be dead yet
  - Contrarily to what Stoyan Nikolov’s “OOP Is Dead, Long Live Data-oriented Design” may make you want to believe
- We can go grab some stuff from it before it dies

A case study of image loading

- Image loading is used in many industries
  - Video games
  - Cinema
  - Earth remote sensing
  - Medical imagery
- Image concepts are varied enough

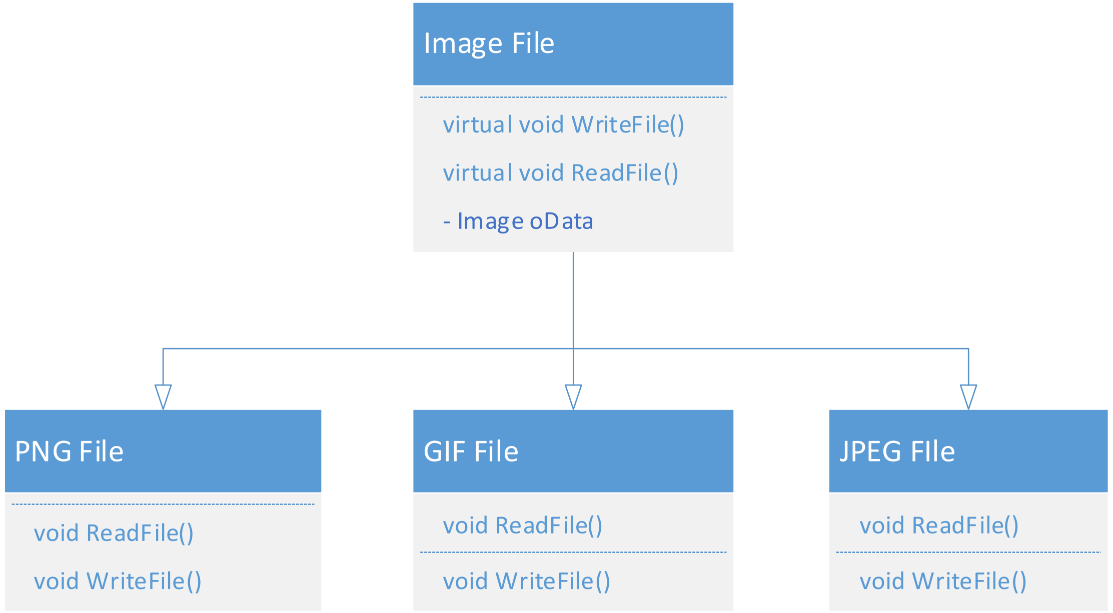
Traditional OOP approach

- Base Image class
  - A virtual loading method
  - A virtual signature method to identify the file type
  - Holds data
- Derived file implementation class
  - Specializes loading and identification methods
  - May hold file specific data
- Let’s adapt it to TOD



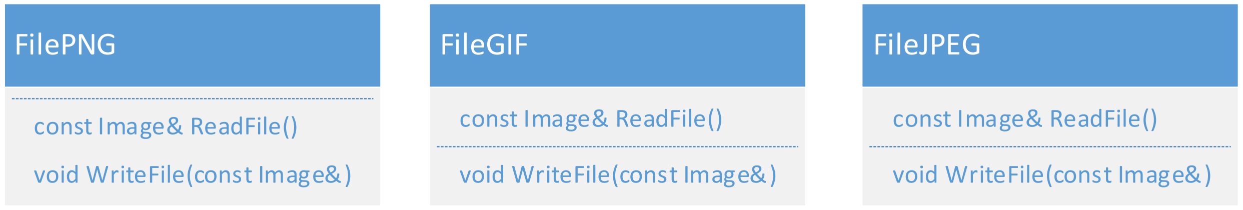
Traditional OOP downsides

- Virtual calls and vtables are mandatory
- Not very flexible
- Locked in the original design
  - Both in data and functions



How it would look in TOD

- Objects only contains functions
- Not inherently linked with data
- No base class
  - A type list replaces it



FilePNG

const Image& ReadFile()

void WriteFile(const Image&)

FileGIF

const Image& ReadFile()

void WriteFile(const Image&)

FileJPEG

const Image& ReadFile()

void WriteFile(const Image&)

```
using ImageFormatDriver_t = typelist<FilePNG, FileGIF, FileJPEG>;
```

Transform-oriented design vs OOP

Concept	OOP	TOD
Unifying principle	Base class	Type list of transform packages
Derived classes	Formal with language	Specific classes
Data associate to class	Yes	No in most cases
Virtual methods	Inherent to the design	None
Signature method	Strict	Free

A transform package definition

- Packages are classes that contain:
  - Static functions/transformations
  - Using definitions
  - Enums
  - Static data
- With the following properties:
  - They are manipulable at compile time only
  - They aren’t instantiable nor destructible
- You don’t need to use that definition

TOD design impacts

- Abstract from specific transformations
- “Minimalist” documentation
- Code modularity

Library

- Contains transformation definitions
- …And other utility functions

Package implementation files

- Contains the objects implementing specific transformations.

User-side code

- Definition of the specific used transformations
- Definition of the execution path

Synthesis

- At this point you should understand the TOD philosophy:
   1. “Transform packages”
   2. Compile time functional programming
   3. Use modern C++ features as ChatGPT suggested

- Now you should be able to write code with it.

Let’s write some code…

- Objectives for the code:
  - Replace hardcoded “switches”
  - Automate said switching process
  - Calling functions only if they exist
    - Ideally without reflection

Example 1 (part 1)

```
#include <iostream>
#include <concepts>
namespace Local { // May be found in metaprogramming
  template<size_t Index, typename Type, typename TList>
  constexpr bool TypeInListImpl() {
    if constexpr (std::is_same_v<Type, TypeAt_t<Index, TList>>) return true;
    else if constexpr (Index == 0) return false;
    else return Local::TypeInListImpl<Index - 1, Type, TList>();
 }
}
template<typename Type, typename TList>
constexpr bool TypeInList() {
   return Local::TypeInListImpl<Length_v<TList> -1, Type, TList>();
}
```

Example 2 (part 2)

```
struct Def {
   static bool Signature() {
      std::cout << "Def" << std::endl; return false;
   }
   static void Mess() { std::cout << "...default" << std::endl; }
};
struct A {
   static bool Signature() { std::cout << “A" << std::endl; return true; }
};
struct B {
   static bool Signature() { std::cout << “B" << std::endl; return false; }
   static void Mess(int i) { std::cout << "...b" << std::endl; }
};
struct C {
   static bool Signature() { std::cout << “C" << std::endl; return false; }
   static void Mess() { std::cout << "...c" << std::endl; }
};
```

Example 1 (part 3)

```
using TransformObjectList = TypeList<Def, A, B>;
#define TRANSFORM_CALL(Function) \
  template<typename Type, typename TList, typename TReturn, typename... Args> \
      TReturn TransformCall##Function(Args&& ...args) { \
    using Default = TypeAt_t<0, TList>; \
    if constexpr (!TypeInList<Type, TList>()) return Default::Function(args...); \
    if constexpr (requires() { Type::Function; }) return Type::Function(args...); \
    else return Default::Function(args...); \
}
```

Example 1 (part 4)

```
using TransformObjectList = TypeList<Def, A, B>;

TRANSFORM_CALL(Mess);

int main() {
   TransformCallMess<A, TransformObjectList, void>();  // Write
« …default". No fonction "Mess" in A class.
   TransformCallMess<B, TransformObjectList, void>(1); // Write « ….b".
   TransformCallMess<C, TransformObjectList, void>();  // Write
"default". C class is not in typeList
}
```

Example 1 (part 5)

- Switches are mappings
  - From a C++ variable
  - To a type

Example 2

```
using ImageFormat = TypeList<BMP, JPEG, PNG, TIFF>;

namespace Local {
  template<size_t Index, typename TList> size_t SignatureImpl() {
    if (TypeAt_t<Index, TList>) return Index;
    if constexpr (Index == 0) return -1;
    else return Local::SignatureImpl< Index - 1, TList>();
  }
}

template<typename Type, typename TList> constexpr size_t Sinature() {
  return Local::SignatureImpl<Length_v<TList> -1, Type, TList>();
}
```

On future developments

- We could do with a bit of simplification
- This may achieved through proposals such as
  - tag_invoke
  - Metaclasses
  - Static reflection
- Data as types
- Or it may be just a matter of creating std::type_sequence and it's friends

Takeaways

- You aren’t really the one writing the code
- Optimization is done at all levels
- Find similarities in between stark differences.

Conclusion

- Did we improve DOD?
  - Maybe…
- They address different “levels” of problem solving
- TOD is compatible with other paradigms
  - The inner workings can be in any paradigm
- You could use it in your code starting from today
  - It doesn’t need to be all about templates

A brief historic (again)

- 1960s
  - FORTRAN
  - Lisp
- 1980s
  - OOP
  - Structured programming
  - Modules
- 2000s
  - Template metaprogramming
  - Concurrency
- 2020s
  - ???

Thanks for your attention

- Please give feedback on sched
- You can contact me by:
  - Finding me in the hallways of CppCon
  - E-mailing ollivier.roberge@gmail.com
  - Messaging me on Discord (@au_lit)
  - Messaging me on Instagram (@ollivier.roberge)