Component Programming in The D Programming Language

by Walter Bright

Reusable Software

- an axiom
- we all buy into it
- we all try to write reusable software

The Reality

- 35 years of programming, and very little reusable code
- copy/pasta doesn't count
- I've missed the boat somewhere
- Other programmers often feel the same way

Something Went Wrong

- It's not for lack of trying
- It's not because I'm a better programmer now than I was, prefering to write it better now
- I need to look deeper

A Troubling Look

- My abstractions are leaky
  - and the dependencies leak out into the rest of the code
- The components are too specific
  - they work for type T, but not for type U
- Need to reset back to first principles

What is a Component?

- more than just reusable software
  - there are lots of libraries of reusable code, but these aren't really components
- a component follows a predefined interface
  - so components can be swapped, added, and composed, even though they are developed independently
- most libraries roll their own interfaces
  - require scaffolding to connect to other libraries

What would a general component interface be?

- read input
- process that input
- write output

Even if a program doesn't fit that model, it is usually composed of subsystems that do

In Pseudo-Code

source => algorithm => sink

or, composing:

source => algorithm1 => algorithm2 => sink

Where Have We Seen That Before?

The Unix “files and filters” command line model

Files and Filters

- Incredibly successful and powerful
- Found its way into C as the “file interface”
  - files are both sources and sinks
  - algorithms are the 'filters'
  - pipes connect them
  - there are even pseudo-file systems to take advantage

http://linux.die.net/man/5/proc

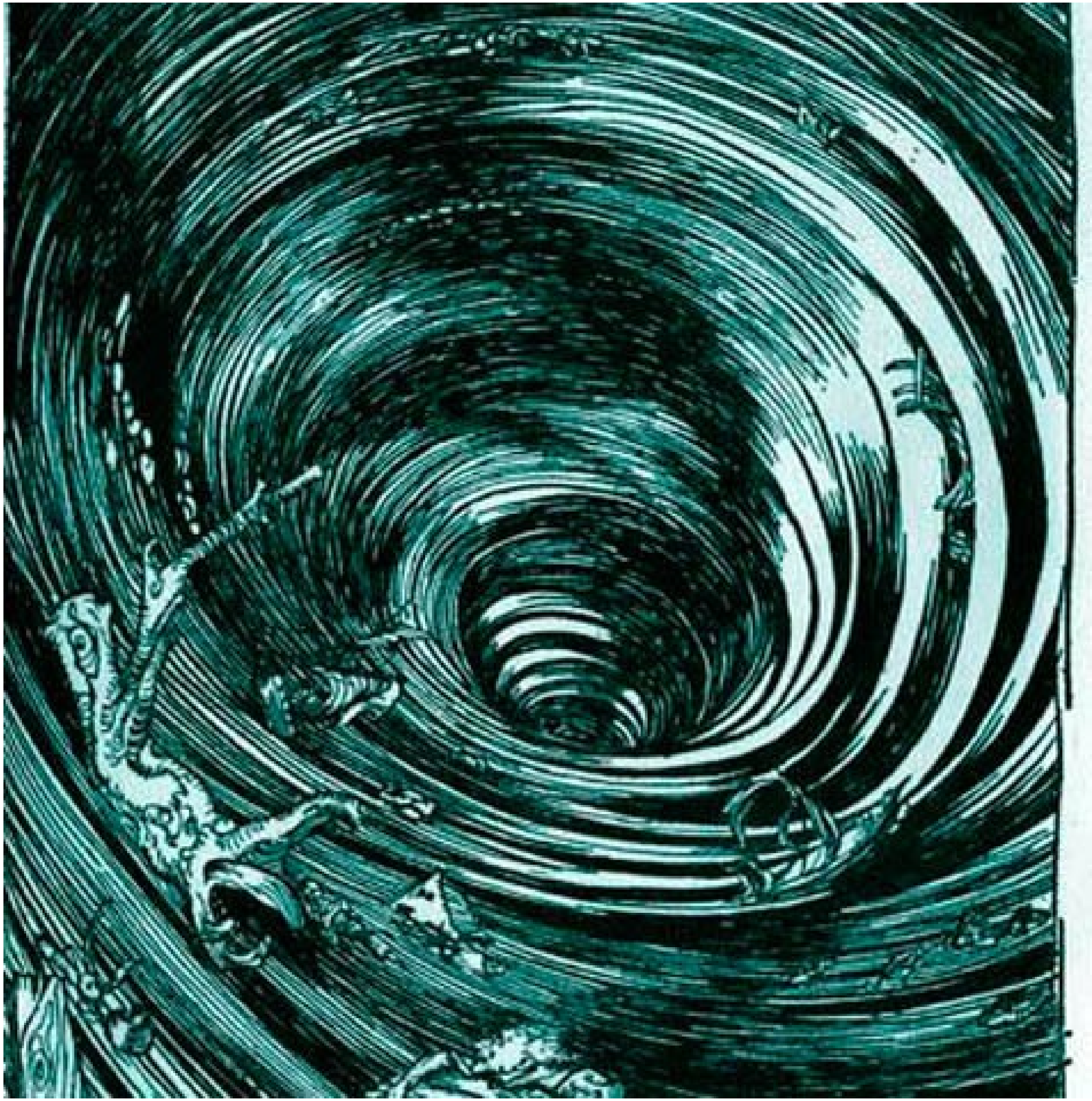
Not Perfect

- Evolved, rather than was designed
  - http://en.wikipedia.org/wiki/Ioctl
- Data is viewed only as a stream of bytes
  - awkward for algorithms that need random access, for example
- But it shows what a component is and that components deliver

Looking Back At My Code

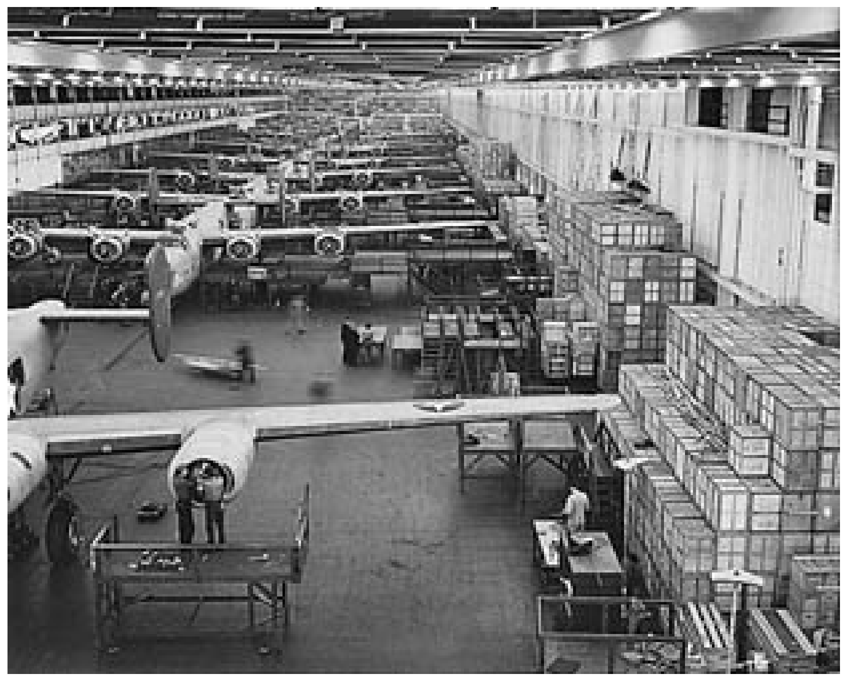
```
void main(string[] args)
{
    string pattern = args[1];
    while (!feof(stdin))
    {
        string line = getLine(stdin);
        if (match(pattern, line))
            writeLine(stdout, line);
    }
}
```

Doesn't look like source => algorithm => sink, it looks like:



Arthur Rackham

Rather Than An Assembly Line



National Archives

I Prefer This

```
void main(string[] args)
{
    string pattern = args[1];
    stdin => byLines => match(pattern) => stdout;
}
```

The Next Design

- C++ OOP
  - did not result in better component programming
- C++ iostreams
  - started looking like source => algorithm => sink
    - by overloading >> operator
  - but never went beyond reading/writing files
- Many successful C++ libraries, but they were not components as discussed here

And Then Came Alexander Stepanov

- Revolutionized C++ with iterators and algorithms, the STL (Standard Template Library)
  - more than just files
  - algorithms
  - common interface
  - compiled to highly efficient code

Not Quite There

```
for (i = L.begin(); i != L.end(); ++i)
    ... do something with *i ...
```

still looks like loops. Then came std::for_each(), std::transform(), but still won't compose because iterators need to be in pairs.

Back To The Drawing Board

- sources
  - streams, containers, generators
- algorithms
  - filter, map, reduce, sort
- sinks
  - streams, containers

Source	Algorithm	Sink
file	sort	file
tree	filter	tree
array	map	array
socket	reduce	socket
list	max	list
iota	search	
random numbers	odd	
	word count	

Summing Up Requirements

- snap together, i.e. composability
- strong encapsulation support
- generate industrial quality efficient code
- natural syntax looking like:
  source=>algorithm=>sink
- work with types not known in advance

Requirements for an InputRange

- is there data available?
  - `bool empty;`
- read the current input datum
  - `E front;`
- advance to the next datum
  - `void popFront();`

InputRange is *not* a Type!

- it's a *Concept*
- all it needs to have are the primitives
  - `empty`
  - `front`
  - `popFront`

InputRange reads chars from stdin

```
private import core.stdc.stdio;
struct StdinByChar {
    @property bool empty() {
        if (hasChar)
            return false;
        auto c = fgetc(stdin);
        if (c == EOF)
            return true;
        ch = cast(char)c;
        hasChar = true;
        return false;
    }
    @property char front() {  return ch;  }
    void popFront() { hasChar = false;  }

  private:
    char ch;
    bool hasChar;
}
```

Read From stdin, Write to stdout

```
for (auto r = StdinByChar();
     !r.empty;
     r.popFront())
{
    auto c = r.front;
    fputc(c, stdout);
}
```

With a Little Language Magic

```
foreach (c; StdinByChar())
    fputc(c, stdout);
```

Look, Ma, no types!

ForwardRange

adds a property:

```
@property R save;
```

(where R is the ForwardRange type)

Returns a copy of the position, not the data.

Original and copy can traverse the range independently.

Singly linked list is the canonical example. Merge sort uses forward ranges.

Bidirectional Range

adds a property and a method:

```
@property E back;
void popBack();
```

Analogous to front and popFront, but they work their way backwards from the end.

Doubly linked list is a typical example, but also UTF-8 and UTF-16 are bidirectional encodings.

Random Access Range

adds:

```
E opIndex(size_t I);
```

to index the data with [ ], and adds one of 2 options:

1. a BidirectionalRange that offers the length property or is infinite:

```
@property size_t length;
```

2. a ForwardRange that is infinite

(`empty` always yields false for an infinite range)

Sinks (OutputRanges)

has a method called:

```
void put(E e);
```

Element e of type E gets “put” into the range.

OutputRange writes to stdout

```
struct StdoutByChar {
    void put(char c) {
        if (fputc(c, stdout) == EOF)
            throw new Exception("stdout error");
    }
}
```

Recall our earlier:

```
foreach (c; StdinByChar())
    fputc(c, stdout);
```

Using an OutputRange, this becomes:

```
StdoutByChar r;
foreach (c; StdinByChar())
    r.put(c);
```

and it even handles errors correctly ! It copies from its input to its output. We could call it 'copy', and...

```
void copy(ref StdinByChar source,
          ref StdoutByChar sink) {
    foreach (c; source)
        sink.put(c);
}
```

Lovely as long as you've got types StdinByChar and StdoutByChar, and it's back to copy/pasta for any other types.

Using a Template

```
void copy(Source, Sink)(ref Source source,
                         ref Sink sink) {
    foreach (c; source)
        sink.put(c);
}
```

Solves the generic problem, but it will accept any input types, leading to disaster

Adding Constraints

```
Sink copy(Source, Sink)(ref Source source,
                        ref Sink sink)
    if (isInputRange!Source &&
        isOutputRange!(Sink, ElementType!Source))
{
    foreach (c; source)
        sink.put(c);
    return sink;
}
```

and there's our first algorithm! (The return is there to make it composable.)

Current Status

```
StdinByChar source;
StdoutByChar sink;

copy(source, sink);
```

not there yet!

Add UFCS

```
func(a,b,c)
```

can be written as:

```
a.func(b,c)
```

so now the component copy looks like:

```
StdinByChar source;
StdoutByChar sink;

source.copy(sink);
```

and we're there!

Filters

```
int[] arr = [1,2,3,4,5];
auto r = arr.filter!(a => a < 3);
writeln(r);
```

which will print:

```
[1, 2]
```

Maps

```
int[] arr = [1,2,3,4,5];
auto r = arr.map!(a => a * a);
writeln(r);
```

which will print:

```
[1, 4, 9, 16, 25]
```

Reducers

```
int[] arr = [1,2,3,4,5];
auto r = arr.reduce!((a,b) => a + b);
writeln(r);
```

which will print:

```
15
```

Putting It Together

```
import std.stdio;
import std.array;
import std.algorithm;

void main() {
    stdin.byLine(KeepTerminator.yes)
    map!(a => a.idup).
    array.
    sort.
    copy(
       stdout.lockingTextWriter());
}
```

Language Features Needed

- Exception handling for errors
- Generic functions
- Template constraints
- UFCS (Uniform Function Call Syntax)
- Ranges are concepts, not types
- Inlining, customization, optimization
- Specialization
- Type Deduction
- Tuples

Conclusion

- components are a way of reusable code
- components are a combination of convention and language support
- many advanced features of D come together to support components
- builds on earlier successes of files & filters, streams, and the STL