Funky Pools – Active Containers for Refactoring Legacy Code

Norman Birkett – normotr@gmail.com – 201-982-1519

Introduction

The **Funky Pool** is a smart or active container class. It's smart in the sense that the container knows something about the objects in it, and it's active in the sense that those objects calculate values inside the container, with the container managing interactions between them. The objects inside the pool are called **Funks – Funktional Units, Named & Krossreferenced**.

Funky Pools were built to solve several problems. Among them was the need to refactor a legacy codebase marked by huge, flat classes and other smells detailed in the next column.

Funky Pools are working well. They've been in production since October, 2016, and are now being used by about twenty-five companies who use our application. I've coded around 350 families of funks. The design is accomplishing its purposes. Development work on Funky Pools is very active in the run-up to CppCon 2018, with further optimization and unit testing the main focus at the moment.

There are three parts to any application of Funky Pools:

1. The pool code itself, which implements the container
2. The client code outside the pool, which inserts input funks into the pool and asks the pool for the values of calculable funks
3. The client code inside the pool—the funks themselves

I'm not at liberty (yet) to disclose the pool implementation. One of my goals for this poster is to find out if there is any interest in seeing that code. What I can show you is some outside-the-pool code and funk code, built around an example invented for this poster.

Building code around Funky Pools has been surprising and interesting.

- You can think about Funky Pools as extending containerization to structs.
- Funky Pools implement a more flexible kind of polymorphism than what we're used to.
- Using Funky Pools feels like a new programming paradigm.
- Ease of testability is a big bonus of using Funky Pools.

I only have room here for a brief conceptual sketch of Funky Pools. I hope what's here spurs your imagination! And I really look forward to hearing your questions and feedback.

Potential

- It may be that the greatest potential of Funky Pools is as a temporary refactoring waystation for code that needs a lot of cleanup.
- Or they may turn out to be a suitable long-term home for certain kinds of intrinsically complex functionality.
- They appear to be extensible.
    - To parallel computation of Funks?
    - To streams of inputs and outputs?

Of course, any code that can be written in pool form can be translated into normal C++ structs or classes (albeit at the cost of some artificiality). And there is a performance cost to using pools. So it may be that the greatest value of Pools is ultimately as a temporary phase for refactoring a tangled mass of badly organized code, with the goal of refactoring the code out of the Pool at the end. Even this would be of considerable value for some of us.

Please Take My Survey

It only has six questions, all optional! If you'd like me to get in touch, you'll be given an opportunity to enter contact information.

https://docs.google.com/forms/d/e/1FAIpQLSf1R_ovTvdV1KSPO_t2fDIzU3xyqepOVMF9T6uB9aOmBbjmqA/viewform?usp=sf_link

Real-World Context

I maintain the cashflow generation code in the calculations library of a Treasury and Risk Management (TRM) application. The library is native C++, but it is used in a .NET environment.

We faced some peculiar problems. The most pressing was that our recursive calculations of cashflows were causing stack overruns from time to time, and Windows doesn't handle this situation gracefully. Not much memory is allocated to the stack in the .NET world. So in effect we needed to relocate these recursive operations from the stack to heap, and convert them to non-recursive, iterative operations that would mimic a stack in heap storage. I'm not sure how many other teams will face that problem.

But the other problem that motivated Funky Pools is a lot more common. Our calculations library originated around twenty years ago as C-flavored C++ code, and its maintainers until recent years were more focused on the financial-engineering side of things than on software engineering. The result is "legacy code" in serious need of cleanup and improvement.

It hasn't helped that we have greatly expanded the functional range of our cashflow generation capability over the past several years. As a result, the codebase has greatly outgrown its own structure, and it's now becoming less efficient for us to add new features and fix certain bugs.

More specifically, this code is marked by such smells as these:

- Large, flat classes without much internal decomposition into lower-level abstractions
- Limited, and often misguided, use of inheritance and polymorphism
- Large, flat functions—many of which run into thousands of lines
- The "bucket brigade" anti-pattern, in which an input is handed through a chain of classes before it reaches the place where it's actually needed

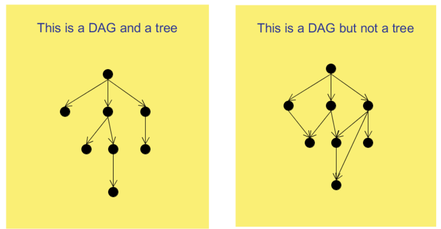
The class within which cashflow generation occurs is a monstrosity with about 110 member variables (many of which are complex aggregates), 430 public functions, and 90 private or protected functions. One of its members is the vector of cashflow objects, each of which has about 70 member variables and 200 public functions.

No tool—including Funky Pools—can solve all the problems in code like this. But I hoped that the Funky Pool could play a part in facilitating progress, and so far it is fulfilling that hope.

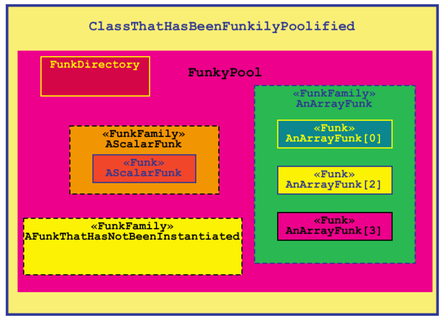
A Central Concept

When we want to break down a giant class into its constituents, how are they related to each other, and how can we model those webs of relationships? Funky Pools model them as **DAGs – Directed Acyclic Graphs**. DAGs are similar to trees, but less restrictive. Some DAGs are in fact trees, but some are not.

How does a Funky Pool use the DAG model? Each Funk that is requested is the root node of a DAG, and that request will trigger the calculation of all the nodes of that DAG—also Funks—that haven't yet been calculated.



Overview of Poolification



When a class is using a Funky Pool, the key components are:

The **FunkDirectory**, which guides the Pool in its operations. It contains a list of **FunkNames**. Each **FunkName** defines a **FunkFamily**, which is a set of one or more possible **Funks**.

For each **FunkName**, the **FunkDirectory** contains the following information:

- The type of its value (or computational result). Any **Funk** in this family will produce the same type of output. The type is represented as a **FunkValueType**.
- Whether members of this family are **Input** or **Calculable Funks**.
- For **Calculable Funks**, the name of the **FunkFactory** that makes the **Funks**.

Both **Funks** and **FunkFactories** inherit from classes defined as part of the definition of Funky Pools. All of their ability to interface with the Pool comes from this inheritance. The most important element of that interface is that each contains a vector of **FunkInput** objects. The Pool uses that to figure out what other **Funks** each **Funk** or **FunkFactory** depends on. In other words, that's what it uses to construct the DAG for the requested **Funk**.

In practice, the majority of **Calculable Funks** are elements of **arrays** or **matrices**, with names like "MyName[n]". These arrays and matrices turn out to be one of the central features of Funky Pools. They are permitted to be sparse, and the elements are entirely independent of one another and can have diverse implementations.

Which leads us to Funky Pools' distinct form of **polymorphism**. A **FunkFactory** can create various kinds of **Funks** for the same **FunkName**, and those different kinds of **Funks** don't have to take the same set of inputs.

Once a **Funky Pool** exists in a codebase, it facilitates an incremental and systematic process of transformation by four pool-specific **refactorings**:

- **Flood**: extend the pool's edge to take in more code. An old **Input Funk** becomes a **Calculable FunkFamily**, and the code that calculated the input value migrates into one or more new **Funks** in that **FunkFamily**.
- **Dissolve** one **Calculable FunkFamily** into several **Calculable FunkFamilies**.
- **Crystallize** several Calculable **FunkFamilies** into one, typically for performance reasons.
- **Drain**: remove some code from the pool. The reverse of **Flood**.

There appear to be other common and useful **refactorings** for **Calculable Funks**. One that can be useful for performance tuning is **Scalarize**, where you convert, for example, an array of **Funks** of type LONG into a scalar **Funk** of type POINTER TO VECTOR OF LONGS.

Example: A Tank

For an example to model on this poster, I settled on an industrial tank simulation. (Perhaps this is because I live in New Jersey.)

The example is too simple to need Funky Pools in real life, but it had to be short and easy. As it is, the code is just an outline, so you'll have to use your imagination. A lot.

Anyway, our tank is a circular cylinder, and the problem has some contrived aspects:

- We have an hourly measure of the depth of the liquid in the tank, which is always available at hour 0 and may be missing for up to one hour at a time later on.
- For every hour, we have the net inflow to or outflow from the tank (as volume).

```
class Tank {
    double m_radius;
    vector<double> m_hourlyDepth; // -1.0 if not available
    vector<double> m_hourlyNetFlow;

public:
    Tank(const double radius, const vector<double>& depths, const
            vector<double>& netflows) …

    double GetDepth(const size_t hr) {
        if (m_hourlyDepth[hr] >= 0.0)
            return m_hourlyDepth[hr];
        else {
            double volume = ConvertDepthToVolume(m_hourlyDepth[hr-1]);
            volume += m_hourlyNetFlow[hr];
            double depth = ConvertVolumeToDepth(volume);
            return depth;
        }
    }
};
```

The Funky Pool Version

Let's start with the FunkDirectory (the pool's guide to the Funks):

Arrays need size limitation.

```
class TankFunkDirectory : public FunkDirectory {
public:
    TankFunkDirectory(void) {
        Define_Input_Scalar("Radius", FunkValueType::DOUBLE);
        Define_Input_Scalar("NumHours", FunkValueType::LONG);
        Define_Input_Array("InputHourlyDepth[]", FunkValueType::DOUBLE,
                                                 "NumHours");
        Define_Input_Array("HourlyNetFlow[]", FunkValueType::DOUBLE,
                                                 "NumHours");
        Define_Calculable_Array("HourlyDepth[]", FunkValueType::DOUBLE,
                                HourlyDepth_Calc_Factory(), "NumHours");
    }
};
```

Now the Tank class itself ("outside the pool" code):

The key methods are CheckIn and FetchValue.

```
class Tank {
    FunkyPool pool;
public:
    Tank(const double radius, const vector<double>& depths,
            const vector<double>& netflows) : pool(TankFunkDirectory()) {
        pool.CheckIn("Radius", FunkValue(radius));
        pool.CheckIn("NumHours", FunkValue((long)depths.size()));
        for (size_t idx = 0; idx < depths.size(); ++idx) {
            pool.CheckIn("InputHourlyDepth[N]", idx, FunkValue(depths[idx]));
            pool.CheckIn("HourlyNetFlow[N]", idx, FunkValue(netflows[idx]));
        }
    }

    double GetDepth(const int hr) {
        return pool.FetchValue<double>("HourlyDepth[N]", hr);
    }
};
```

Next, we have a Funk with its FunkFactory (a piece of "inside the pool" code):

Pool-facing behaviors are inherited.

```
class HourlyDepth_Calc : public ArrayElemFunk {
public:
    HourlyDepth_Calc(const FunkName* funkName, const size_t idx)
                                        : ArrayElemFunk(funkName, idx) {
        inputs.emplace_back("InputHourlyDepth[N]", idx - 1); // [0]
        inputs.emplace_back("HourlyNetFlow[N]", idx); // [1]
        inputs.emplace_back("Radius"); // [2]
    }
protected:
    virtual void CalculateAndStoreValue(void) override {
        double depthPrevHour = FunkInputGet<double>(inputs[0]);
        double netFlowThisHour = FunkInputGet<double>(inputs[1]);
        double radius = FunkInputGet<double>(inputs[2]);

        double volumePrevHour{ CalculateVolume(depthPrevHour, radius) };
        double volumeThisHour{ volumePrevHour + netFlowThisHour };

        double depthThisHour{ CalculateDepth(volumeThisHour, radius) };

        *myValue = depthThisHour;
    }
};
```

Inputs are declared in ctors, and/or the Factory can insert them in the Funk (not shown).

```
class HourlyDepth_Calc_Factory : public ArrayElemFunkFactory {
public:
    HourlyDepth_Calc_Factory(void) {
        inputs.emplace_back("InputHourlyDepth[N]"); // [0]
    }
    virtual Funk* MakeFunk(const FunkName* funkName, const size_t idx) override {
        double inputDepth = FunkInputGet<double>(inputs[0]);

        if (inputDepth >= 0.0)
            return new ArrayElemPointer(funkName, idx,
                                        "InputHourlyDepth[N]", idx);

        else
            return new HourlyDepth_Calc(funkName, idx);
    }
};
```

Funks can be pointers to other Funks.

Testability is Built In

A **Funky Puddle** is a Funky Pool that allows any Funk to be checked in as an input—for testing.

```
TEST_GROUP(TankUnitTests) {
    FunkyPuddle* puddle;

    void setup() {
        puddle = new FunkyPuddle(TankFunkDirectory());
    }
    void teardown() {
        delete puddle;
    }
};

TEST(TankUnitTests, HourlyDepthCalcWorksFromPriorDepthPlusNewFlows) {
    double radius{ 20.0 };
    double thisHourDepthNotAvailable{ -1.0 };
    double prevHourDepth{ 12.0 };
    double netFlowInLastHour{ -5000.0 };
    double expected{ 8.02 };
    double tolerance{ 0.003 };

    puddle->CheckIn("Radius", FunkValue(radius));
    puddle->CheckIn("InputHourlyDepth[0]", FunkValue(prevHourDepth));
    puddle->CheckIn("InputHourlyDepth[1]", FunkValue(thisHourDepthNotAvailable));
    puddle->CheckIn("HourlyNetFlow[1]", FunkValue(netFlowInLastHour));

    double actual = puddle->FetchValue<double>("HourlyDepth[1]");
    DOUBLES_EQUAL(expected, actual, tolerance);
}
```

Just CheckIn your target Funk's inputs and Fetch its Value – instant unit test. Funky!