Nate Bachmeier

Week 3: Algorithms and Data Structures

TIM-8101: Principals of Computer Science

Northcentral University

March 24th, 2019

# What are template classes?

Mad-Libs is a famous word game which involves two parties. The first will come up with a short story and then insert place holders for some of the nouns, verbs and adjectives. The second person without knowing the story will provide values for these placeholders. Finally, the story is read aloud, and everyone has a good laugh.

Templates are very similar to Mad-Libs, as the programmer can write the story as a sequence of place holders and steps needed to declare a function or class. The compiler will then create an instance of the template and fill in the place holders with literal values. These literal values could be class types, constants, or similar operations.

For instance, a class could be written to hold a list of integers and expose public methods add, remove, and get. These functions would then perform the necessary actions to manipulate some private integer array structure.

When the program also needs to hold a list of strings, then the developer would want to reuse as much of the list of integers code as possible. The code could be manually copied however this introduces challenges as bug fixes must go into multiple places. Instead they could create one template and ask the compiler to emit the different versions at build time.

Another implementation could have been written as one class which holds a list of objects. Since everything in object-oriented programming is an object, the list of objects can hold either the string or the integer.

However, this introduces new challenges since the implementation is essentially disabling the type system and its associated compile time checks. This has the potential to result in runtime errors such as invalid cast exceptions, as the list can also contain unexpected data types. In a large system tracking down the inputs that lead to these errors can be tedious and very time consuming. There is also performance overhead as the assembly level must box and unbox the arguments at runtime (Hristov, 2013).

# Where are they used? Include a real-world example.

## Data Structures and Algorithms

Templates are often associated with data structures and algorithms use cases. It is common for their implementation to be mechanical and to work irrespective of the data type they operate on (Sedgewick & Wayne, 2011).

Consider a stack or queue data structures which is pushing and popping items onto a list. It does not matter if those items are strings, sockets, or tacos—the data structure will behave the same. Then look at a sorting algorithm such as quick and merge sort. They can order any data type that implements a comparability interface.

## Metaprogramming

Some implementations of templating allow for metaprogramming during the compilation phase. This allows the generation to perform customized actions based on annotations or constant calculations (Liberty, 1997).

For instance, the developer might create an interface with different methods representing different schematized log messages. Using an interface creates a clear separation between the component and the logging system. A factory class could then be used to emit the concrete log client based on the interface definition. This enforces consistency across all log clients as one central location is emitting each concrete logger implementation.

Having the ability to inject constants into the template can also solve certain classes of issues. For example, the template might behave differently if the size of a word is 32 or 64bits (Pikus, 2015). Another use case is to generating lookup tables during the build time.

Object Relational Models (ORM) are another consumer of the metadata driven model. First, a process connects to a data store and fetches the schema information. Then it uses it to template serialization definitions for each of the tables. The output of these templates is then fed into another round of templates to provide create, read, update, and delete (CRUD) operations. This results in strongly typed generic code that can be statically verified.

# What are some benefits and drawbacks of using template classes?

## Increased Size

Each instantiation of a template results in the generation of more code which needs to live within the binary. This will increase the size requirements of the of the program. For some embedded systems with limited resources, this can introduce challenges.

Languages like C# have partially mitigated the code bloat scenario by exposing generics instead of templates. Generics are a runtime constructs that exploit the type system directly instead of relying on code generation (Microsoft, 2015). This provides many of the core use cases but is more limited than C++ templates which run at the compilation level. An example can be seen with the inability to inject constant values into a generic type definition.

## Recursive Templating

When a template is used to generate a class, that class is part of the type system and treated the same as any other class. This property exposes the ability for templates to operate on templates and then pass them to more templates. That enables general algorithms to expand into very specific concrete implementations (Liberty, 1997) (Pikus, 2015).

The simplicity of that model makes it easy for developers to abuse the technology and write unmaintainable code. There are still use cases that require a concrete structure instead of a mash up of the typed data structures. Consider the scenario of a dictionary that is keyed on a tuple and valued with list of async function pointers. The return signature might be written as Dictionary<Tuple<T,U,V>,IList<Func<Task<X,Y,Z>>>. This can introduce readability concerns for the caller as they must decipher the meaning. Perhaps T and U are both strings, but which one is the name or address?

Alternatively, a container class and small object model could be returned. Those classes could then be documented, and the intent made clear. Compare the previous example with Dictionary<Customer, ActionList>, which is easier to understand. Inside of the ActionList could be templated properties that are well named.

# What are some alternatives to template classes, and where/when would you use them?

Templating technologies are not limited to the compilation phase and can occur during other stages of build or runtime.

## C-Style Macros

During the preprocessing stage, macros can be applied to perform literal manipulations on the code itself.

For instance, an application might defined a macro called ISOK(expression,message); and liberally use it throughout the system. The preprocessor could expand this snippet into a try-catch block that automatically logs a message on error. This makes the code cleaner and easier to maintain as the redundancies are written by the compiler not the developer.

However, improperly written macros can result in cryptic error messages as they are literal text expansions. This can lead to the developer scratching their head at lines such as MIN(MIN(x,y),z); and an error that the line is missing a parenthesis.

## Moq Framework

One of the challenges with writing integration tests is that they often become coupled with live services. This can make them expensive to run or error prone as certain failure cases are difficult to reproduce. An innovative solution to this problem can be seen in the Moq framework for .net.

Developers first create an instance of the factory class Mock<T> where T is the interface to be generated. The mock exposes a setup method which can be passed expression trees and callback functions. The framework generates a new assembly in memory and constructs the templated instance (moq, 2019).

By having this technology execute during the runtime layer, it mitigates the lack of support for templates in the C# language itself, cleanly hides the generated code, and provides a mechanism to weave additional runtime state objects into the mocked instance.

# What research has been conducted using template classes?

## LISP

Clojure is a language that is derived from LISP and has taken a minimalistic stand point toward built-in language constructs (Hickey, 2019). As a “LISt Processing” language, everything is specified as a list, with the first item specifying the macro name followed by its arguments. The macros are expanded at runtime and processed until they result in a single data value.

This makes it very trivial to extend the language with new keywords and operators. In fact, nearly all of operators within Clojure are runtime expanded macros (Hickey, 2019). When new operators are introduced they can be highly customized to the domain specific problem.

Consider a build script that is written in Clojure and exposes macros for source checkout, code compilation, publishing to binaries to a network share, and executing unit tests. A clear separation then exists between the build owner and the component owner; with neither required to understand the internals of their counterpart. The central definitions of these macros and are less costly to maintain than had they been copied into each repository.

## Nemerle

Nemerle is a managed language for the Common Language Runtime and has taken templating to another level. This is accomplished by allowing developers to extend the compilation pipeline and had made custom syntax trees first-class citizens.

In the language tutorial the authors describe how to write a custom for-loop syntax tree and then use it anywhere else in the program (Russian Software Developer Network, 2012). Having the ability to extend the language gives the developer unlimited flexibility in the way they template. It also provides a mechanism for adding domain specific operators which can greatly increase the simplicity and readability of application code.

The tutorial also describes the <[expression]> operator which is executed during compilation. The expression can directly access the parser state to manipulate the class structures, add additional interface inheritance, or provide additional methods. This results in a fluidity to software authoring, as the developer focuses on intent instead of boiler plate logic.

# What is some future research you can envision using template classes?

## Better Serialization

Web service programming typically involves passing either JSON or XML based object representation. The serialization and deserialization are often performed by generic runtime parsers which can be slow to operate. In contrast, templating could be used to create very specific parsers that are highly optimized for the individual object.

These optimized implementations could save 100s of milliseconds per round trip between the client and the service. Multiply that by the number of calls received by an Internet scale service such as Facebook or Amazon-- and it quickly saves the CPU years of parallel compute time.

Binary serializers such Google’s protocol buffers have partially solved this issue (Shuai & Xiaojun, 2017). The challenge with binary serializers is that both the client and service need additional dependencies. Having additional dependencies restricts the potential audience and increases barrier to entry.

## Other Use Cases

Templating technologies are an essential tool for authoring maintainable code, and they are used in most modern applications. It could easily be envisioned that every future research project will use them in one way or another. Imagine having to rewrite the linked list from scratch for each assignment. That would not only be tedious, but also and error prone without templates.

# References

Hickey, R. (2019). *Learn Clojure.* Retrieved from Clojure: https://clojure.org/guides/learn/syntax

Hristov, K. (2013, February 17). *Difference between Generics in C#, Java and C++.* Retrieved from Developing the Future: http://www.developingthefuture.net/difference-between-generics-in-c-java-and-c/

Liberty, J. (1997). *Teach Yourself C++ in 21 Days.* Sams Publishing.

Microsoft. (2015, July 19). *Generics (C# Programming Guide).* Retrieved from Microsoft Documentation: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/

moq. (2019, March 23). *moq*. Retrieved from Github: https://github.com/moq/moq

Pikus, F. (2015, October 20). *CppCon 2015: C++ Metaprogramming: Journey from simple to insanity and back.* Retrieved from YouTube: https://youtu.be/CZi6QqZSbFg

Russian Software Developer Network. (2012, August 31). *https://github.com/rsdn/nemerle/wiki/Macros-tutorial.* Retrieved from Github: https://github.com/rsdn/nemerle/wiki/Macros-tutorial

Sedgewick, R., & Wayne, K. (2011). *Algorithms Fourth Edition.* Pearson Education.

Shuai, L., & Xiaojun, H. (2017). Dubbo's serialization protocol extension and optimization of its RPC protocol thrift. *IEEE International Conference on Software Engineering and Service Science (ICSESS) Software Engineering and Service Science (ICSESS)*, 721-726.