CD5130

Metaprogramming in Java, C# and C++

Patrik Lindegrén
Joakim Östlund

24th May 2006
Abstract

We take a look at generics for three programming languages: C++, Java and C#. We will look at what generic and metaprogramming is, and in what way these languages support these aspects of modern programming. We will also compare the languages with respect to these aspects, and see what weaknesses and strengths each offers.
1 Introduction

As computer programs grow in size and complexity, it becomes increasingly hard to keep the entire programming team up to date on what functionality currently exists in the software; at the same time, this becomes increasingly important. This problem can be solved by naming conventions for datasets and functions, but it is never the less a meta-structuring of the program; all objects still exist on a global level, and it might for instance be hard for a given programmer to know what values in a dataset can be safely altered directly, and what values must be altered through a function, and which such functions exist.

Thus was the paradigm of Object Oriented Programming (OOP) born. OOP strives to give the programmer a natural way of compartmentalizing data, as well as fundamental way of structuring code. It contains several key concepts, such as encapsulation and abstraction, which solves many of the problems with classical programming. It also gives rise to some problems of its own.

1.1 Problem statement

Let’s say that we are working on program. In this program, we frequently need to place objects on stacks. Due to the fact that we have functions that take a stack and perform different operations on the objects on that stack depending on their type, we need to make sure that each stack contains only one single type of object. How should we implement this, while adhering to the principles of OOP?

One approach would be to use a built-in generic data type, such as void * in C++. The problem with this is that we forfeit the compiler’s type checking features, and instead force the programmer(s) to manually double check each use of a stack for type correctness. And what happens if we accidentally push an incorrect object on a stack? The program might silently fail, causing data corruption down the line; it may crash at random times, and so on. And this type of problem can be hard to spot during a source code audit, causing it to go unfixed for long periods of time.

Another way might be to create a superclass using generic data types, and then implement a child class for each specific data type. This will give us compiler type checking, but will create an unneeded clutter of classes. Subclassing in this manner is also inflexible, and if the superclass is changed, a lot of work might be needed to update all the derivatives.

1.2 Generics

One solution to this problem is generics. Generics is the use of a data type within a class of function, which can take on the form of any other data type, while allowing the compile to enforce strong type checking.

How generics are implemented differs from language to language, and will be discussed later in this report (see section 5.2), but the usage is similar; upon creating an instance of a class, or calling a function, the programmer specifies the data type expected. If a function call is made, the compile can type check
the input parameters and the return value. If it’s an instance of a class, the specified type will remain known to the compiler during that class’s lifetime. This allows us to tell the compiler that this instance of our generic stack will be a stack of apples, and the compiler will only let us push apples onto that stack. Any attempt to push an orange onto the stack will result in a compiler error.

1.3 Metaprogramming

Metaprogramming is a technique that addresses situations where it is the code itself, rather than the type of data, that needs to be implemented in a generic fashion, but be specific in each instance. An example of this that will be thoroughly looked at later in this report is a vector class (see section 2.3.4).

Vectors are a multi-dimensional mathematical entity that is used in a lot of calculations, specifically in computer games. Because they are commonly used in high-performance applications, we want them to be fast. But there is also needs for vectors of different dimension; commonly 2D and 3D vectors are used. While one theoretically could create a generic class, and specify the dimensions of it every time an operation was performed on a vector, this would be somewhat counter-intuitive, and would create code bloat. Also, one would need to use loops to perform calculations, such as adding two vectors, which adds overhead. And, just like in the original problem statement, creating one class for each dimensionality is undesirable because it adds more points of maintenance to the code.

Through the use of metaprogramming, the programmer can create a generic class, to which we pass a parameter upon instantiation, so the compile can generate specific code for a vector of that particular dimension, giving us code for each dimension of vector we would ever need, specialized for that dimension.

2 Generics in C++

2.1 Overview

Generics in C++ is achieved by using templates [8] [9] [10]. There are two kinds of templates, function templates and class templates.

2.2 Implementation

A template starts with the keyword template followed by the template arguments, which is a comma-separated list of generic data types, and the function or class definition:

```cpp
template < Arguments > Function() {}
template < Arguments > class {};
```

The template arguments are identifiers, which can be:

- a data type.
- a class.
• `typename`, which is a variable data type that can be any standard data type.
• `class`, which is a variable data type that can be any standard data type or class.

A template function and class can be defined like this:

```cpp
template < typename T, class C, int I >
void Function()
{
}

template < typename T, class C, int I >
class Class
{
}
```

In functions and classes the template arguments can be used exactly as if they where classes or built-in data types.

When the templates are used the compiler generates a specific function for every data type that uses the template, this is called template instantiation. If this function is used with three different data types, the compiler will create three different functions. The compiler only generates definitions for templates that require instantiation. For example when instantiating a class, only the member functions that are used will be generated.

The templates help the programmer to create functions that works in the same way with different data types, so that the functions don’t need to be copies, which would be very annoying to handle, for example when something needs to be changed. The source also becomes much smaller.

2.2.1 Function templates

The following template function works for any built-in data type, and it returns the greatest value of the two parameters. This template function works exactly as the max function in the `standard template library`.

```cpp
template < typename T >
T Max( T A, T B )
{
    if( A > B )
        return A;
    else
        return B;
}
```

This function template could be used like this:

```cpp
int iMax = Max( 2, 3 );
float fMax = Max( 2.0f, 3.0f );
```
In the above example we had implicit instantiation of the templates; the compiler used the types of the arguments and function return type to choose a suitable instantiation. To explicitly specify which instantiation to use:

```c
float fMax = Max<float>( 2, 3 );
```

The Max template function could be created with macros:

```c
#define Max( A, B ) ( ( A ) > ( B ) ? ( A ) : ( B ) )
```

When using templates the compiler performs type checking. Macros are not type-safe as templates. With templates it’s possible to create this Max function so that it only works for specific data types or specializations that work in a different way depending on the data type.

### 2.2.2 Class templates

Class templates are often used as containers. Examples of this are a list or vector class, like list and vector in the *standard template library* [1]. The template classes make it possible to have vectors of integers, strings or other classes. A template vector class definition could look something like this:

```c
template < class C >
class vector
{
private:
    C *Data;
    int iSize;
    ...
public:
    vector( const int iSize );
    ~vector();
    C &operator [] ( const int i );
    Resize( const int iSize );
    ...
};
```

The class contains an array of the class C, which is initiated by the constructor and deleted by the destructor. The class also contains functionality to resize itself. The template class would be used like this:

```c
vector< int > V( 3 );
V[0] = 3;
V[1] = 2;
V[2] = V[0] + V[1];
V.Resize( 5 );
```

### 2.2.3 Template specialization

It is possible to specialize a template function for a specific data type. If the Max function (defined in section 2.2.1) was used to compare strings (char*) like this:

```c
7
char *szMax = Max( "AAAA", "BBBB" );

the result would be the pointer with the highest value. To achieve a function
that "sorts" strings, we could use template specialisation like this:

template <>
char* Max( char* A, char* B )
{
    if( strcmp( A, B ) > 0 )
        return A;
    else
        return B;
}

It is possible to use template specialization for classes as well. A specialization
of the vector class (defined in section 2.2.2) could look like this:

template <>
class vector < float >
{
};

It is possible to specialize a part of the template arguments, if there is more
than one, this is called partial template specialization[10]. This works for both
function templates and class templates.

2.3 Advanced uses - template metaprogramming

Templates in C++ can be used to express more than generic programming.
With templates it’s possible to make the compiler pre-evaluate a part of the
program in compile-time, this is called template metaprogramming. You can
see it as "programming at compile-time". The compiler can generate everything
from compile-time constants to advanced data structures. Template metapro-
gramming can create highly optimized programs doing advanced computations
at compile-time instead of run-time, leading to faster programs. By moving
computations to compile-time, the program compile-time increases.

2.3.1 Example - max function

template < int A, int B >
struct Max
{
    enum { RET = ( A > B ) ? A : B };  
}

This Max function does the same thing as the previous Max example (defined in
section 2.2.1), but in compile-time. Note that this one only works with integers.
The enum forces the compiler to evaluate the expression and assign the resulting
value to RET. The RET value could also have been a static const int. The function
could be used like this:

int iMax = Max< 2, 3 >::RET;
This Max template must be called with constant parameters, known at compile-time. The following would not work:

```cpp
int iA = 2, iB = 3;
int iMax = Max< iA, iB >::RET;
```

The variables iA and iB must be constants.

### 2.3.2 Example - factorial

The following example shows how template specialization can be used to calculate a factorial in compile-time.

```cpp
template < int N >
struct Factorial
{
    enum { RET = N * Factorial< N - 1 >::RET };
};
template <>
struct Factorial< 0 >
{
    enum { RET = 1 };
};
```

The Factorial template is used like this:

```cpp
int iFactorial = Factorial< 5 >::RET;
```

The instance Factorial< 5 > uses the instance Factorial< 4 > to calculate the result, which uses Factorial< 3 > and so on. The result is assigned to the variable iFactorial, in this case it is 120.

### 2.3.3 Example - loop unrolling

The recursive method used in the factorial example (see section 2.3.2) can be used not only to calculate constants but also to generate code. A common for-loop can do some unnecessary operations:

```cpp
for( int i=3; i >= 0; i-- )
{
    cout << i << endl;
}
```

Every loop i is decreased and compared with 0. In situations where the number of iterations are known, and speed is essential, templates can be used to remove this overhead, by *loop-unrolling*:

```cpp
template < int i >
struct LOOP
{
    static inline void EXEC()
    {
        cout << i << endl;
        LOOP< i - 1 >::EXEC();
    }
};
```
The LOOP structure can be used like this:

```cpp
LOOP<3>::EXEC();
```

This will generate the following code:

```cpp
cout << 3 << endl;
cout << 2 << endl;
cout << 1 << endl;
cout << 0 << endl;
```

The final code has no `i` variable at all, and does no unnecessary computations at run-time. With loop-unrolling it is possible to create fast code, but when there is a lot of code inside the loop and many iterations, this will create larger executables for a very little gain. Loop-unrolling should be used only when it's necessary.

### 2.3.4 Example - fast vector math

A very common structure is the one used for vector math [12]. Often there are one structure for every one of the most used dimensions, like 2D, 3D and 4D. The structure is used to manage the vector data and a number of operations, the basic are addition, subtraction, multiplication, division, dot product. A very simple version of the structure for a 3D vector that supports different data types could look something like this:

```cpp
template <typename DataType>
struct CVector3D
{
    typedef CVector3D<DataType> Type;
    DataType m_adVal[3];

    inline CVector3D( const DataType x, const DataType y, const DataType z )
    {
        m_adVal[0] = x;
        m_adVal[1] = y;
        m_adVal[2] = z;
    }

    inline Type &operator = ( const Type &v )
    {
        m_adVal[0] = v[0];
```

10
m_adVal[1] = v[1];
m_adVal[2] = v[2];
return *this;
}
inline Type operator + ( const Type &v ) const
{
    return Type( m_adVal[0] + v[0], m_adVal[1] + v[1],
                 m_adVal[2] + v[2] );
}
inline DataType operator [] ( const int i ) const
{
    return m_adVal[i];
};

The structure only implements addition in this case, this is to save space, the other operations would look almost identical. If A, B, C and D were instances of CVector3D, they could be used like this:

A = B + C + D;

This kind of vector structure is ok in most of the cases, but not when the application needs to be as efficient as possible and the vectors are used a lot. For every addition, the structure creates a temporary vector, where the result of the addition is saved. This vector is used as input to the next addition. C + D creates a temporary variable, let’s call it T1. T1 is added to B and the temporary variable T2 is created. A is then assigned to T2. Every temporary variable will be saved in the memory or in a processor register. Since the vectors are 3D there will be 3 floats saved for every temporary variable. The more operations on the vectors and the higher the dimension of the vector the less is the chance that the temporary variables will be kept in registries. To make this more effective we want to keep the compiler from using temporary variables, and reading and writing from memory. This can be achieved by solving the problem one dimension at the time:

A.X = B.X + C.X + D.X;
A.Y = B.Y + C.Y + D.Y;
A.Z = B.Z + C.Z + D.Z;

Templates can be used to create a structure CVector that can generate this code from the original A = B + C + D. This might look simple, but it’s not. This structure will not only be used for different data types, it will also work for any dimension. The CBase structure contains the data of the vector and functionality for creating the fast code, in this listing without any operations for assign or addition. This base structure is inherited by the CVector:

template < int ta_iDimension, class ta_DataType >
struct CBase
{
    typedef CBase< ta_iDimension, ta_DataType > Type;

    ta_DataType m_aVal[ta_iDimension];
inline ta_DataType& operator[]( const int i )
{ return ( ( ta_DataType* )this)[i]; }
inline const ta_DataType operator[]( const int i ) const
{ return ( ( ta_DataType* )this)[i]; }
inline const ta_DataType Evaluate ( const int i ) const
{ return ( ( ta_DataType* )this)[i]; }
};

template < int ta_Dimension, class ta_DataType >
struct CVector : public CBase< ta_Dimension, ta_DataType >
{
...
};

The CBase structure has operators to access the vectors components and an Evaluate function. This function will be used when the expressions are evaluated, as we will see later on. A class CVectorArgument that can handle an operand in the expression and a class CVectorExpression2 that can hold a part of the expression, are created:

template < class ta_A >
class CVectorArgument
{
    const ta_A &Argv;
public:
    inline CVectorArgument( const ta_A& A ) : Argv( A ) {}
    template < typename ta_DataType >
    inline const ta_DataType Evaluate( const int i ) const
    {
        return (ta_DataType)Argv.Evaluate< ta_DataType >( i );
    }
};
template< class ta_A, class ta_B, class ta_Evaluator >
class CVectorExpression2
{
    const CVectorArgument< ta_A > Argument1;
    const CVectorArgument< ta_B > Argument2;
public:
    inline CVectorExpression2( const ta_A& A1, const ta_B& A2 ) :
    Arg1( A1 ), Arg2( A2 ) {}
    template < typename ta_DataType >
    inline const ta_DataType Evaluate ( const int i ) const
    {
        return ta_Evaluator::Evaluate< ta_DataType >
            ( i, Arg1, Arg2 );
    }
};

The CVectorArgument class will hold a reference to an operand, a CVector, expression or possibly another class. This reference is set by the constructor. The
Evaluate function looks exactly the same as the one in CBase, but it returns the result of the argument’s Evaluate function, which should be a CVector.

The class CVectorExpression2 holds a binary expression, containing references to the two operands. The arguments are two CVectorArgument, which evaluates vectors, or another CVectorExpression2. The constructor sets the references to the CArgument:s. When Evaluate is called for the expression, it uses the template argument ta_Evaluator, which is a structure that operates on the arguments. The ta_Evaluator could for example be a CAdd structure:

```cpp
struct CAdd
{
    template < typename ta_DataType, class ta_A, class ta_B >
    inline static const ta_DataType Evaluate( const int i,
        const ta_A& A, const ta_B& B )
    {
        return A.Evaluate< ta_DataType >(i) +
            B.Evaluate< ta_DataType >(i);
    }
};
```

The CAdd structure only contains an Evaluate function which returns the sum of the arguments Evaluate functions. The expression A + B could be expressed like this:

```
CVectorExpression2< A, B, CAdd >
```

A more advanced expression like A + B * C could look like this:

```
CVectorExpression2< A, CVectorExpression2< B, C, CMul >, CAdd >
```

To create a expression like this, an operator + is added to the CBase structure:

```cpp
template < class ta_A, class ta_B >
inline const CVectorExpression2< const ta_A, const ta_B, Add >
operator + ( const ta_A& A, const ta_B& B )
{
    return CVectorExpression2< const ta_A, const ta_B, Add >( A, B );
}
```

To get the result from that expression, a operator = is added to the CBase structure. The operator = needs to loop over the vector components. To unroll that loop a CAssignment structure is created:

```cpp
template < class ta_VecArg >
struct CAssignment
{
    template < int I >
    struct recurse
    {
        enum { COUNTER = I + 1 }; 
        static inline void Assign( Type &V, const ta_VecArg& A )
        {
            // Code...
        }
    }
};
```
V[I] = (ta_DataType)A.Evaluate<ta_DataType>(I);
    recurse<COUNTER>::Assign(V,A);
};

};
template<>
struct recurse<ta_iDimension>
{
    static inline void Assign(Type &V, const ta_VecArg &A){}
};
static inline void Assign(Type &V, const ta_VecArg &A)
{
    recurse<0>::Assign(V,A);
}

};
template<class ta_Type>
inline const Type &operator=(const ta_Type &A)
{
    CAssign<ta_Type>::Assign(*this,A);
    return *this;
}

The = operator can assign a CVector or a CVectorExpression2 to the vector. With the CVector3D you could call a function with an expression like this:

Function(A + B);

This doesn’t work with the CVector, because the A + B creates a CVectorExpression2 not a CVector. With the CVector3D structure the expression was evaluated for every part, creating a temporary vector, with the CVector structure, the user needs to tell the compiler when to do this, by creating a CVector:

Function(CVector(A + B));

This assumes that the CVector will have a constructor that can take an expression. The constructor is easy to write, since it only needs to use the existing = operator.

To be able to add a number to a vector we would like to write:

A = B + 0.5f;

This could easily be achieved by creating a specialization of the CVectorArgument:

template<>
class CVectorArgument<float>
{
    const float &Argv;

    public:
        inline CVectorArgument(const ta_A &A) : Argv(A) {};

    template<typename ta_DataType>
    inline const ta_DataType Evaluate(const int i) const
    {
        return (ta_DataType)Argv;
    }
A specialization for integer, double or other types could be added easily.

This method to solve the problem a dimension at the time, works good for operations where only one of the components are used for every operation. Cross product is an example of a operation that uses all the components of the vector to calculate the resulting vector. An implementation of cross product would need to use a temporary variable, consider this example:

\[
A = A \times B;
\]

Since the cross product operator uses all components to calculate all the resulting components, \(A[0]\) will change when the first dimension is calculated. When \(A[1]\) is to be calculated, it uses \(A[0]\) which has changed, therefore the result will be wrong.

CVector will be able to evaluate expressions with different data types. This implementation uses the data type of the result vector, and uses it to typecast every operand in the expression. For example, if the result is a vector of doubles \(VD\), and the operands are a vector of integers \(VI\) and a integer number:

\[
VD = VI + 5;
\]

The code generated will be:

\[
VD[0] = (double)VI[0] + (double)5;
\]

This is not very efficient, and the problem could not be solved without any form of explicit typecasting. A worse problem could be if the result is vector of integers and the operands are floats:

\[
VI = 0.1f \times 4000.0f;
\]

The code generated will be:

\[
VI[0] = (int)0.1f \times (int)4000.0f;
\]

The result will be 0 * 4000, which is 0. To avoid this with this implementation, the user should see to it that the data types don’t mix, or remove the typecasting from the vector class, which will give warnings when data types are mixed.

### 2.4 Benefits and drawbacks with templates

**Benefits:**

- Can create truly generic code.
- Can be used when macros are not enough.
- Can add extra type-safety where void pointers were used before.

**Drawbacks:**

- Portability: Not all compilers can handle templates, other does it badly.
• Bad error messages, hard to develop and debug.
• Hard to read and understand, the code can become big, look very strange and non-intuitive.
• Hard to maintain.
• Compile-time increases.
• Templates can’t be compiled to binary, so the code must be public. When something using a template changes, it must recompile everything.

2.5 Applications
There are many libraries using templates to achieve generic and fast code. The Standard Template Library (STL) [1] is included in to the C++ Standard Library. It describes containers, iterators, and algorithms.

Boost [5] is a collection of libraries that take heavily use of templates. It contains structures, algorithms, math components and more. It also makes it easier with generic programming and template metaprogramming. The libraries work well together with the STL and aim to become a part of the future C++ Standard. Boost was started by members of the C++ Standards Committee Library Working Group. Now there are thousands of programmers from the C++ community participating in the development. Before anything is added to the libraries it is subjected to thorough peer review to be able to guarantee quality.

Blitz++ [7] is a library that was developed to handle complex scientific computing problems without any performance loss. C++ is slower than languages like Fortran, but some of these differences can be eliminated with the use of templates, to create more efficient code by moving computations from run-time to compile-time. The library contains a vector class that works in the same way as the vector class described in section 2.3.4.

2.6 Future development
C++0X [13] is the new version of the C++ standard that is currently been worked on. The aim is to get it ready before 2010, therefore the name C++0X. The standard will contain a number of new features, perhaps some of them already exist in the Boost library.

3 Generics in Java
3.1 Overview
Java’s generics are focused on giving programmers a very flexible and type safe method of creating generic functions and classes. Towards this end, Java employs some special structures, like wildcard generics and bounded wildcard generics, to allow more flexibility in data types, but still retain type checking.
3.2 Implementation

Java does not create code for each type-specific instance of a generic, like C++ does (REF HÄR). Instead, all generic data types will be replaced by the built-in generic type "Object", where possible, and byte-code will only be generated once for each function and class. Thus, it is not possible to retain static values in the generic object, nor is it possible to use the data type specifier as a type cast. The following code will demonstrate a practice that is possible in C++, but will generate compile warnings in Java, and is generally discouraged, due to this fact:

```java
<T> void AddObjectToList(SomeClass<T> List, LegacyClass Obj) {
    List.AddToList( // Takes an object of type T
        (T) Obj.GetObject()); // GetObject returns an Object, typecast this to type T
}
```

This code snippet will generate an "unchecked type" warning, since the code will be converted into this:

```java
void AddObjectToList(SomeClass<Object> List, LegacyClass Obj) {
    List.AddToList(
        (Object) Obj.GetObject());
}
```

And the run-time will be unable to type check the variables for you.

General

Generic interfaces and functions are declared with a comma-separated list of generic data types to be used, e.g.

```java
public interface Stack<T> {
    void Push(T Item);
    T Pop();
}
```

Each type specifier in the list corresponds to one data type; do not use two type specifiers if you expect the use to always use the same type for both. One example of when we would like to use more than one type specifier is in a "pair" class, which will use one data type for the key identifier, and one for the key value:

```java
public interface Pair<K, V> {
    K m_Key;
    V m_Value;
    ...
}
```

And this might be used in several ways:

```java
Pair<String, int> pSI;
Pair<float, MyClass> pFMC;
```
3.3 Wildcards

In Java, the data type Object is often used as a generic type. A pre-generics implementation of a stack would probably use Object as parameter type and return type in the push/pop functions. Now, with generics, we have a better method that allows us to type check our generics structures. However, consider the following legacy function, written before generics and implementing an Object stack:

```java
void PerformActionOnStack(Stack TheStack) {
    ...}
```

This worked well before, since all stacks were of the same kind. Now, however, we want our function to accept our new generic-stacks. A first approach might be to try something like this:

```java
void PerformActionOnStack(Stack<Object> TheStack) {
    ...}
```

Thus reverting back to the old "use Object for generics" behaviour. The problem is that Java does not allow you to typecast e.g. a Stack of Apple to a Stack of Object. The solution is to use a wildcard:

```java
void PerformActionOnStack(Stack<?> TheStack) {
    ...}
```

This tells Java that this function accepts a "Stack of Unknown", which will allow any type of stack to be passed to the function. Something to note is that while you are using a wild carded generic you will not be able to use the functions that take variables of the generic data type(s) as parameters, but you will be able to use those that return values of those types.

In the example above, we would be able to use the Pop() function of the stack, since we are able to typecast "Unknown" to "Object", but we cannot use the Push() function, since Java will not allow us to typecast anything to "Unknown."

3.4 Bounded wildcards

Assume an application that makes use of a generic class List. This application keeps track of Students, Teachers and Administrative personal at a school. Students, Teachers and Administrative personal are kept in separate Lists, but they all share the common superclass Person. All Persons have some common traits that we might be interested in printing, so we create a function to do this:

```java
void PrintInfo(List<Person> TheList) {
    ...}
```

The problem with this is that we only accept Lists of Persons, passing a List of Students would cause a compile error. We could solve this by using a wild
card List, but we want as much type checking as possible, and simply using a List of Unknown would allow us to accidentally pass a List of Classrooms to the function. The solution is to use bounded wildcards, giving us the following function:

```java
void PrintInfo(List<? extends Person> TheList) {
    ...
}
```

Thus we tell the compiler that what this function accepts is a List of "Unknown subtype of Person." Note that this could include the class Person itself, as well as any class that implements it. The same restrictions applies to bounded wildcards as to unbounded ones; we cannot typecast anything into an Unknown subtype of Person, not even a Person, but typecasting from it can be done to any valid type.

### 3.5 Metaprogramming in Java

Java does not contain any native constructs for performing metaprogramming. However, as metaprogramming techniques are becoming more commonly used and refined, Java programmers have become ever more aware of this limitation, and some has even gone so far as to call the lack of metaprogramming "the downfall of Java"[14].

But all is not lost: metaprogramming can be accomplished through the use of a pre-processor, independent of the actual Java language. Some of these are actually implemented as importable java-packages, such as RECODER[17], where you create a program and compile it with your regular Java compiler. The output of the program is the new code that you can compile to get the program you were writing.

Other, more common approaches are regular pre-processors that parse a source file and create a pure java source file that can be compiled. Some of these, like MetaJ[16], consist of a mixture of plain Java code that will be copied verbatim into the output file, as well as special metaprogramming declarations that will control how the output code is created. Others, like TyRuBa[15], are an entirely different meta-language, aimed at producing code for specific purpose (in TyRuBas case, first order logic).

Due to the large number of metaprogramming pre-processors and packages that are available for Java, it's hard to present any one as an example of how to do it. For Java programmers, the task rather becomes to figure out what functionality they need, and chose a tool accordingly.

### 4 Generics in C#

#### 4.1 Overview

C# wishes to provide the same generics behaviour that is present in C++, but with added flexibility and control. A big focus has been made on making all
generic data types type safe.

4.2 Implementation

C#'s implementation of generics is a hybrid of C++'s and Java's implementation methods. Code is only generated once for each generic object during compile-time, but the compiler includes a lot of meta-data in regards to the generic properties of the objects. Instantiation then occurs during run-time, and the first time a new data type specifier is used, a new instance of the generic object is created. This allows compile-time type checking, as well as run-time type checking and validation.

4.3 General

Generics in C# has the same syntax as generics in Java, with the declaration containing a comma-separated list of the generic data types to be used:

```csharp
public interface Stack<T> {
    void Push(T Item);
    T Pop();
}
```

The same discussion as for Java holds true in C#: each type specifier should correspond to one data type, and not one particular use. More than one type specifier should only be used when the user might need to use several distinct types:

```csharp
public interface Pair<K, V> {
    K m_Key;
    V m_Value;
    ...
}
```

The usage of generic types is also the same as in Java:

```csharp
Pair<String, int> pSI;
Pair<float, MyClass> pFMc;
```

4.4 Constraints

Constraints are used to control what data types can be used with your generic class/function. Unlike wildcards in Java, whose primary application is allowing generic structures to be passed as arguments in a function call, C#'s restraints tell the compiler what data types can actually be used to instantiate your class, or call your function.

Let’s recall our school software application from the Java chapter. Instead of using Lists of Persons and their derivatives, we construct a container class that can handle one or more Persons, or any subclass of Person. Such a class would be meaningless if instantiated with String as value type parameter, since the class assumes that the value type will have certain basic properties. To solve this, we use a constraint when declaring the class:

```csharp
public interface Container<T> where T : Person
```
public class PeopleHandler<T> where T : Person {
    ...
}

This tells the compiler that we require that the value type parameter be a derivative of Person, or Person itself. Anything else will result in a compiler error. C# encourages the use of constraints on value type parameters when constructing classes and interfaces, and provides a set of standard interfaces, such as IComparable if you require the ability to compare values of the given value type, that should be used to ensure that all value types used follow a standardized layout and functionality scheme.

4.5 Metaprogramming in C#

Just as is the case with Java, C# does not contain any native construct for metaprogramming. Instead, we need to construct some sort of pre-processor to generate the necessary code. Unfortunately, as of today, few such exists for C#.

5 A comparison of generics in C++, Java and C#

5.1 Overview

C++, Java and C# are syntactically very similar, which is not very surprising considering that the two latter languages drew somewhat on C++ when they were designed. When it comes to generics though, the syntax varies some. C++ uses its complex template metaprogramming engine to do generics, while Java and C# have special constructs aimed at creating generics. Despite all the similarities between the three languages, they each take their own approach to determining what features are needed from generics, and how they should be implemented.

5.2 Implementation

C++ implements generics through binary instantiation. This means that each time a function of class method is used with a specific data type, if that data type has not previously been used; the compiler creates specific code for that class or function using that data type. The benefit of this behaviour is improved performance; using template functions and classes does not incur any extra overhead compared to non-template versions. Drawbacks are generally limited to increased binary size and slower compile times.

Java chose another approach to this, to make sure that the new generics worked without having to modify the Java VM. Instead of creating specific bit code, the generic java source code is transformed into regular non-generics code using the built-in generic data types (of course, this happens after type checking has taken place). The upside to this is an optimized binary size as well as a smaller memory footprint. It does however add a little overhead, since everything will be implicitly type cast to and from a generic data type. Another issue that crops
up is that all typing information is lost at run-time, preventing run-time type casts, as well as the usage of some language constructs such as `InstanceOf`.

C# uses a method that combines those of C++ and Java. While the binary itself does only contain one copy of any given code, some meta-information is saved about data types used in conjunction with the generic objects. This meta-information is used at run-time to instantiate classes and functions as they are being used, unless they already have been instantiated for that data type. This method will give smaller binaries than the C++ way, and an initially smaller memory footprint. However, as the objects are instantiated, the memory footprint will eventually reach the same size. Using the run-time for instantiation makes sure that all type information is saved, and can be used by language constructs, but adds overhead as the code needs to be created "on the fly".

5.3 Specifics

C++ does not have any native way of constraining generics. Once a class or function is generic, you can instantiate it using any class or data type (with one exception, if the `typename` keyword is used, you can only use basic data types). There is only one way to limit the use to a specific class, that’s with template specialization. This isn’t a valid solution, since it forces the user to duplicate code. Furthermore the specializations can’t handle instantiation with subclasses. For that to work there would have to be a specialization for every subclass. All this is against the whole generic concept.

Since the C++ templates can be instantiated with every data type or class, the errors occur when the compiler tries to instantiate the template with a type that doesn’t implement the methods or operations needed in the template. This stops the user of the template from doing a mistake, in most cases. But when the type used implements the method, the compilation may work fine, but the end result isn’t what was expected. An example of this is seen in section 2.2.3 where the `max` function works with pointers, but doesn’t give the result that one would want. Further, when passing a generic object as an argument in a function call, the called function must be created as a template function, and will be instantiated accordingly.

Java does not have any constructs to constrain the instantiation of generic objects either, but it does have wildcards that can be used in function parameters, and bounded wildcards that can be used to specify that a certain function or method can only be called with instances of a generic class that uses a subset of data types.

C# has the same limitation as C++ when it comes to using generic objects as function arguments; you need to have the function as a generic as well, and instantiate it accordingly. C# does however have the ability to constraining what data types can be used in instantiation of classes and functions, which allows you to control what data you will receive.
6 A comparison of metaprogramming in C++, Java and C#

Of the three languages discussed here, C++ is the only one that has native metaprogramming facilities. Both Java and C# require external pre-processors or complex libraries that are capable of generating code, which then needs to be compiled separately. The fact is that few programmers know of metaprogramming techniques, make use of them or use them correctly. Nevertheless, metaprogramming can be used to accomplish a lot of good things, and it is a real weakness in a language to exclude such features at a native level.

7 Conclusion

Generics can be the right tool when you need a more general solution, like for a container class. Other problems can be solved without generics. With generics you often create less code, that does more, but often it’s more complicated to write and harder to understand and maintain.

Metaprogramming is a good way to make the compiler generate code or move calculations to compile-time. With templates in C++ it’s possible to write code that’s faster, and does less computations run-time. The problem is that this often takes longer time to write. In the end the developer needs to make a decision, is the faster run-time worth the extra development-time and compile-time.

The languages builds on different philosophies with different implementations. Know a couple of languages, there strength and weaknesses, and pick the one best suited for the problem at hand.
References


http://en.wikipedia.org/wiki/Template_metaprogramming

http://www.artima.com/intv/generics.html


[5] The Boost Library
http://www.boost.org/libs/libraries.htm

http://en.wikipedia.org/wiki/Boost_library

[7] The Blitz++ Library
http://www.oonumerics.org/blitz/docs/

[8] MSDN on Templates

[9] Templates

[10] Templates Tutorial
http://www.is.pku.edu.cn/~qzy/cpp/vc-stl/templates.htm

http://www.flipcode.org/cgi-bin/farticles.cgi?show=63820

[12] Faster Vector Math Using Templates
http://www.flipcode.com/articles/article_fastervectormath.shtml

[13] C++0X
http://www.artima.com/cppsource/cpp0x.html

[14] Meta-Programming, Java’s Downfall
http://ozone.wordpress.com/2006/02/20/metaprogramming-javas-downfall/

http://tyruba.sourceforge.net/

[16] MetaJ Java Meta-Programming System
http://www.emn.fr/x-info/sudholt/research/metaj/

[17] RECODER Java Meta-Programming System
http://recoder.sourceforge.net/

[18] Introduction to Generics (C# Programming Guide)