**Subject: Support for generics without type erasure (aka templates)**

New Lomboc annotation: @Template
Targets: generic class, generic method

Effect: when applied to the target, allows the target implementation to use the generic type parameters as if they were not subjects to the type erasure

General implementation design:

When first code use of the target is detected (e.g. invocation of method that is annotated with @Template, inheritance of the class, annotated with @Template), triggers the generation of new, non-generic target implementation, with use of the specific types instead of generic type parameters.
The subsequent uses of the target with the same set of type parameters reuses the generated implementation.

Note:

This document is just a draft with some narrow examples to demonstrate the idea; not all the edge cases are covered.

**Template method example:**

User code:

```
public class A
{
  @Template
  public static <T> Class<T> Foo(T ignored)
  {
    return T.getClass();
  }
}

public class U
{
   public static void main(String[] args)
   {
     Class<Integer> integerType = A.Foo(new Integer());
   }
}
```

Generated Code:

- original generic method removed from A
- A gets generated method equivalent to generic, but with specific type generated
- Usage of the method is replaced to invoke generated method

```
public class A
{
  public static Class<Integer> Foo_Template_Integer(Integer ignored)
  {
    return Integer.getClass();
  }
}

public class Main
{
   public static void main(String[] args)
   {
     Class<Integer> integerType = A.Foo_Template_Integer(new Integer());
   }
}
```

### Template class example:

User code:

```
@Template
public class A<T>
{
  protected int field;

  public A(int value)
  {
    field = value;
  }

  public Class<T> Foo()
  {
    return T.getClass();
  }

  public T Bar()
  {
    final int arg = 1;
    return new T(arg);
  }
}

public class DerivedFromA extends A<Integer>
{
  public DerivedFromA(int value)
  {
    super(value);
  }
}

public class U
{
  public static void main(String[] args)
  {
    A bare = new A(1);
    A<Integer> inline =  new A<>(2);
    DerivedFromA derived =  new DerivedFromA(3);
  }
}
```

Generated code:

- Template instantiations of the class are the nested classes of original class, inherited from the original class, with generic type parameters replaced with specific classes
- Generic class enhanced to support that inheritance
- Generic class enhanced to support type erasure cases like:

```
A<Integer> intA = new A<>();
A<?> base = intA;
Class<?> type = base.Foo(); // will return Integer.class
```

- Usages of A<Integer> class updated to use generated classes and methods

```
public class A<T>
{
  public class TEMPLATE_TYPE_PARAMS
  {
    public final Class<T> T_PARAM;

    public TEMPLATE_TYPE_PARAMS(Class<T> T_PARAM)
    {
      this.T_PARAM = T_PARAM;
    }
  }

  public final TEMPLATE_TYPE_PARAMS TYPE_PARAMS;

  protected int field;

  protected A(Class<T> T_PARAM, int value)
  {
     TYPE_PARAMS = new TEMPLATE_TYPE_PARAMS(T_PARAM);
     field = value;
  }

  public A(int value)
  {
     this(Object.class, value);
  }

  public Class<T> Foo()
  {
     return TYPE_PARAMS.T_PARAM;
  }

  @SneakyThrows(ReflectiveOperationException.class)
  public T Bar()
  {
     final int arg = 1;

     return TYPE_PARAMS.T_PARAM.getConstructor(int.class).newInstance(arg);
  }

  public static class Template_Integer extends A<Integer>
  {
    public A_Template_Integer(int value)
    {
      super(Integer.class, value);
    }

    public Class<Integer> Foo()
    {
      return Integer.class;
    }

    public Integer Bar()
    {
      final int arg = 1;
```

```
      return new Integer(1);
    }
  }
}

public class DerivedFromA extends A.Template_Integer
{
  public DerivedFromA(int value)
  {
    super(Integer.class, value);
  }
}

public class Main
{
  public static void main(String[] args)
  {
    A bare =  new A(1);
    A<Integer> inline =  new A.Template_Integer(2);
    DerivedFromA derived =  new DerivedFromA(3);
  }
}
```