Persistent storage framework

Developing a flexible framework for storing data

Report by:

John Juul Jensen <jjj@ghostdev.com>

CPH STL Report 2003-4

Department of Computer Science
University of Copenhagen
Fall 2003

Contents

Contents 2

1 Introduction 4

2 Storing data 5
- 2.1 Automating the store process . . . 6
- 2.2 Making generated classes usable . . . 11
- 2.3 Constructing a serializer interface . . . 13

3 The actual framework 17
- 3.1 POD types . . . 17
- 3.2 STL containers . . . 18
- 3.3 Compound types . . . 20
- 3.4 Legacy types . . . 25
- 3.5 The serializers . . . 26

4 Test 28

References 29

A PSF Source listings 30
- A.1 Compound class . . . 30
- A.2 Serializer class . . . 31
- A.3 Serialize functions . . . 31
- A.4 DeSerializer class . . . 35
- A.5 DeSerialize functions . . . 36
- A.6 XML (de)serializer . . . 41
- A.7 Binary (de)serializer . . . 49
- A.8 Pretty Print serializer . . . 54
- A.9 Utilities . . . 57

B Test Source listings 59
- B.1 Test Program . . . 59
- B.2 Results . . . 66
- B.3 Data . . . 70

Abstract

Storage of data in computer programs can hardly be considered a difficult task, but it is often tedious and error prone. It is therefore desirable to automate this task if possible. Automatic storage of data (*serialization* here after) can easily be incorporated into a class hierarchy during construction, but it is much harder to achieve if legacy code is involved. So while being able to create serializable classes from scratch is essential, it must also be possible to augment legacy code, so that it supports serialization just as easily as specialized classes.

There are many other problems when discussing serialization. For example, how should the data be stored, the choices are many; we could store it as binary data, raw text or XML, just to name a few. It would be nice if the ability of classes, or other data types, to be serialized were independent of the storage mode. A different problem is that of safety and by that I mean 'type safety' in the normal C++ sense, but also dealing with corrupt/mangled data files. We need an implementation that is robust.

1 Introduction

This project is about developing a serialization framework which deals with all the problems mentioned above. The three main principles should be flexibility, robustness and, ease of use. The implementation presented is based on two basic components, serializable data types and serializers.

All basic C++ types (*POD*[1] types) plus the most commonly used STL containers will be serializable. Tools for creating serializable classes and for augmenting legacy classes will also be included. Furthermore, it will feature three serializers, two for random access streams (binary files and XML files) and one for one-way streams, in the form of a pretty printer. Other serializers that this framework could support includes SQL DB serializers, zip file serializers, or serializing to a set of graphic controls so the user can change the data run-time. The XML serializer will be type safe in the sense that all serialized data will be tagged with a unique type identifier, which can be checked when deserializing. Finally, some effort will be spent on minimizing the size of serialized data, especially the binary data format.

Although the (de)serializers may be important for working with persistent data types, their implementation is not considered a high priority in this project. The (de)serializers presented here will be fully functional, so testing can be performed and as a proof of concept.

The reader of this report is expected to have a good understanding of C++, templates in particular, and generic programming in general.

[1] Plain Old Data

2 Storing data

This section deals with the process of automating data storage, to aid the following discussion, a small example of how one would normally go about storing some data is presented below:

Listing 1: Standard way of storing data

```
// A class that can be stored and retrieved from a stream
class CTest1
{
public:
    void Save( std::ostream& i_ostream);
    void Load( std::istream& i_istream);

protected:
    int m_1;
    int m_2;
};

void CTest1::Save( std::ostream& i_ostream)
{
    // Store all the members in a stream
    i_ostream << m_1 << std::endl << m_2;
}

void CTest1::Load( std::istream& i_istream)
{
    // Retrieve the members from a stream
    i_istream >> m_1;
    i_istream >> m_2;
}

int main()
{
    {
        // Create a stream and store class
        std::fstream fileout( "class.txt", std::ios::out);
        CTest1 test;
        test.Save( fileout);
    }
    {
        // Create a stream and retrieve class
        std::fstream filein( "class.txt", std::ios::in);
        CTest1 test;
        test.Load( filein);
    }
}
```

Although details may differ, the principle in the code presented above should be familiar to most people. As it should be, because this is basically the only way to store and retrieve data. So if this is the way to do it, then what can be changed and why should it be changed? The answer to these questions is, nothing and no reason. What can be done though is to make it easier, easier to code and easier to maintain. Some of the problems with the code above is outlined here:

1. Adding an extra member to the class. In order to maintain a working class, the `Save` and `Load` functions must be extended to handle the new member, but what if the programmer responsible for making the change forgets to extend the `Save` and `Load` functions.

2. Changing file format. This will require changing all classes that save data in that format, which could mean a lot of work and also the risk of introducing new bugs in an otherwise working program.

Finding a way to avoid these issues is the path to making things easier for ourselves.

The first problem is somewhat more difficult than the second. How can one stop people from forgetting things? Well you cannot and even if you could, it would be nice if we did

not have to change the Save function just because we added a new member. In section 2.1 it will be revealed how we can achieve these not so simple goals and why C++ needs a new keyword.

The second problem is easily solved. Instead of writing directly to a file, a network socket or whatever, an abstract interface will be used when saving or loading data. The use of an abstract interface, whose implementation translates and routes the data to the correct place, frees the programmer from having to deal with file formats, except when implementing the interface. Easy, problem solved, or is it. What kind of interface is general enough to handle all kinds of data? Section 2.3 deals with the problem of constructing such an interface.

2.1 Automating the store process

As stated earlier you cannot stop people from forgetting things[2], so how can this be made a non-issue. Perhaps taking a step back and trying to see things in a more abstract setting will make things clearer.

The basic process of storing data is as follows: `Store some type`
`Store another type`
`Store a class` → `Store each member of the class`
`Store yet another type`

For now let us assume that storing 'some type', 'another type', and 'yet another type' is not a problem. Then all that is left is the class. The goal is to automate that class, so that all of its members are stored[3], regardless of how many there are, what types they have, and how often they are changed. Some programming languages like JavaScript lets you enumerate all members of a class at run-time, if that could be done in C++, code like the following would be possible:

Listing 2: Run-time iteration of members

```
class CTest2
{
public:
    void Save( std::ostream& i_ostream);

protected:
    int m_1;
    int m_2;
};

void CTest2::Save( std::ostream& i_ostream)
{
    // Iterate members at run-time
    for ( int i = 0; i < number_of_members; ++i)
    {
        i_ostream << member[i];
    }
}
```

Of course this is not really desirable, because that would mean run-time evaluation of types and that is not really in the spirit of C++. Other languages like C# requires that all

[2]Although some Chinese herbs are said to have that effect, but making sure that every programmer you will ever meet is properly medicated, is an impossible task and possibly illegal.

[3]Without loss of generality, it can be assumed that all members of a class will need to be stored.

types have a 'ToString' member, which could probably be used to the same effect.

Member enumeration is quite desirable and since the types of all class members are known at compile-time, it seems reasonable that member enumeration should be possible. Sadly this is not the case. What C++ needs in order to achieve this, is a `for_each_member` keyword, that would call a given function for each member using normal overloading rules. An example of a class that uses the proposed keyword is given below:

Listing 3: Usage of for_each_member keyword

```
template<typename T>
void Store( const T& i_param, std::ostream& i_ostream)
{
    ...
}

void Store( int i_n, std::ostream& i_ostream)
{
    ...
}

class CTest3
{
public:
    void Save( std::ostream& i_ostream);

protected:
    int m_int;
    float m_float;
    std::string m_string;
};

void CTest3::Save( std::ostream& i_ostream)
{
    // Evaluates to
    // Store( m_int, i_ostream); // Non-template Store
    // Store( m_float, i_ostream); // Template Store
    // Store( m_string, i_ostream); // Template Store
    for_each_member Store( member, i_ostream);
}
```

The exact syntax and semantics of the proposed keyword is not important here, but the benefits of such a keyword should be clear to anyone. So now the big question is how can the behavior of this keyword be emulated, using normal C++. Only two things are needed — convert the run-time loop to a compile-time loop and find a way to reference members by an integer index.
The first part is simple, run-time loops can be converted to a compile-time recursion using standard template specialization:

Listing 4: Compile-time iteration

```
class CTest4
{
public:
    void DoLoop()
    {
        // Execute the loop once for each member
        Loop<number_of_members>();
    }

    template<int i>
    void Loop()
    {
        // Do something with member i
        ...
        Loop<i-1>();
    }

    template<>
    void Loop<0>()
    {
        // Do something with member 0
        ...
    }
```

```
};
```

That takes care of the recursion, but before tackling the member enumeration, some facts, that may appear somewhat disconcerting at first, must be faced. Since enumeration of class members is impossible in C++ as it is today, compromises must be made on certain aspects of class generation. Later on though, it will be clear that these compromises may in fact be for the best, since they can enforce good coding practices in a very strict manner.

The compromises in question stems from utilizing one of the hottest new concepts in C++, namely type lists. Type lists were introduced by Andrei Alexandrescu in his book 'Modern C++ Design'[1]. Basically, as the name suggest, it allows us to work with and manipulate lists of types, much like the lists know from Lisp or ML. Since type lists are based on template recursion, all type list operations are performed at compile-time, which suggests that they fit our purpose well.

Type lists alone would not do much good though, but class generation using type lists, a technique also developed by Andrei, turns out to be just what the doctor ordered. The actual mechanics behind the class generation is not important, so they will not be discussed further, instead an example of how class generation is used is presented below:

Listing 5: Generating a class

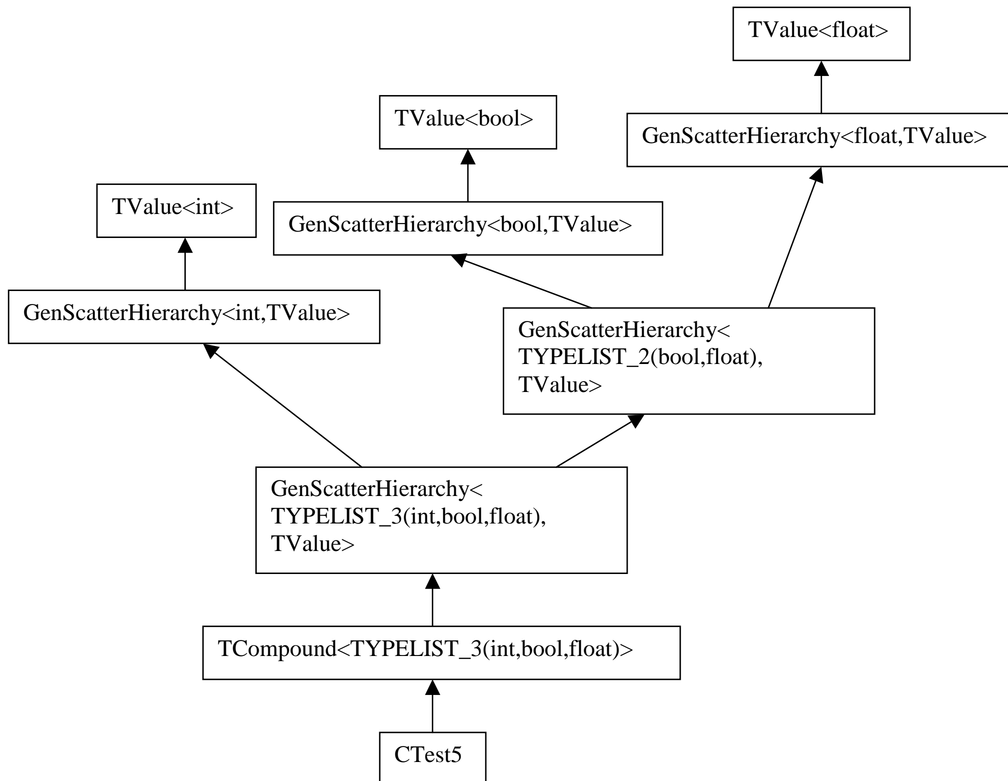
```
template<typename t_type>
class TValue
{
public:
    t_type m_value;
};

template<class t_TypeList>
class TCompound : public GenScatterHierarchy< t_TypeList, TValue>
{
public:
    enum { number_of_members = Loki::TL::Length<t_TypeList>::value};
};

typedef TYPELIST_3(int,bool,float) Types;
class CTest5 : public TCompund<Types>
{
};
```

The important thing to notice, in the above example, is the `GenScatterHierarchy` class that `TCompund` inherit from. `GenScatterHierarchy` is part of the generic programming library 'Loki'[1], which is authored and maintained by Andrei as open source software. 'Loki' provides a number of interesting generic programming tools, but for our purposes the type list implementation and the `GenScatterHierarchy` class will suffice. `GenScatterHierarchy` generates a class hierarchy, using its two template arguments. The first argument must be a type list and the second a template class. `GenScatterHierarchy` inherit from the template class argument instantiated with the head[4] of the type list and also inherit from itself with the tail of the type list and the template class as arguments. An inheritance diagram for the code in listing 5 can be seen in figure 1. As can be seen `CTest5` inherits from `TValue` three times, once for each member of the type list and each of the `TValue` classes has been instantiated with a type from the type list. The

[4]A type list is treated as a tail recursive list just like in ML, where the first element is the head and the rest of the list is the tail.

GenScatterHierarchy class has no members and does not affect the memory footprint[5] of the generated class, or the runtime overhead of accessing the class.

Figure 1: Inheritance hierarchy for class CTest5 in listing 5



A class, with one member for each item in the type list, has now been generated, but all the members have the name m_value. This makes it hard to access the right member, since it involves casting the generated class to one of its base classes in order to access the member of that base class. If the type list argument contains the same type more that once, which is likely, then the cast can lead to an ambiguity that cannot be solved, meaning that the members of that particular type are inaccessible. Luckily, 'Loki' provides a mechanism, in the form of a template function called Field, that allows access to the base classes by positional index. The base class resulting from type i in the type list, can be accessed by providing i as template argument to this function, along with a reference to the generated class. Before putting all the things in this section together, let us first create a function that stores a CTest5 class, so that the Field function can be seen in action.

Listing 6: A Store function for CTest5, continued from listing 5

[5] This is not entirely true, since multiple inheritance can cause some memory overhead due to multiple virtual function tables being generated.

```
void Store( const CTest5& i_param, std::ostream& i_ostream)
{
    // The Field function with template argument 0 returns a reference to TValue<int>
    i_ostream << Field<0>(i_param).m_value;
    // The Field function with template argument 1 returns a reference to TValue<bool>
    i_ostream << Field<1>(i_param).m_value;
    // The Field function with template argument 2 returns a reference to TValue<float>
    i_ostream << Field<2>(i_param).m_value;
}
```

Compile-time recursions makes it possible to loop once for each member of a class and members of type list generated classes can be accessed by positional index, combining these techniques is therefore the next logical step. The `Store` function from listing 6 can be rewritten as a template function that takes a `TCompund` object instead of a specific class and that is exactly what happens in listing 7

Listing 7: A function for storing compounds

```
template<class t_TypeList>
class TStoreCompoundHelper
{
public:
    typedef TCompound<t_TypeList> Compound;

    TStoreCompoundHelper( const Compound& i_compound) :
        m_compound( i_compound)
    {
    }

    template<int t_i>
    void StoreInternal( std::ostream& i_ostream)
    {
        i_ostream << Loki::Field<Compound::number_of_members-t_i-1>(m_compound).m_value
        StoreInternal<t_i-1>( i_ostream);
    }

    template<>
    void StoreInternal<0>( std::ostream& i_ostream)
    {
        i_ostream << Loki::Field<Compound::number_of_members-1>(m_compound).m_value
    }

private:
    const Compound& m_compound;
};

template<class t_TypeList>
void Store( const TCompound<t_TypeList>& i_param, std::ostream& i_ostream)
{
    TStoreCompoundHelper<t_TypeList> helper( i_param);
    helper.StoreInternal<TCompound<t_TypeList>::number_of_members-1>( i_ostream);
}
```

The code in listing 7 probably requires some explanation. The `Store` function makes use of a helper class that takes care of the compile-time recursion and also performs the actual storing of data. It is also worth noting that, since the iteration is performed in reverse, counting down to zero, the index must be inverted before passing it to the `Field` function. If the index was not inverted, the members of the compound class would be stored in reverse order, which is not a big deal, but it is nice to know that the first member is also stored first.

At this point, most of the obstacles have been overcome and working toward a more complete solution is now possible. So from here on, the code examples given will resemble the finished implementation more than those given in this section. The reason for this is that the following sections discuss actual real-world implementation and usage issues.

2.2 Making generated classes usable

Listing 5 demonstrated how a class, that can be stored automatically, could be generated. However, the members of the generated class will most certainly need to be accessed from other parts of the program. So some way of making the class easy to work with is needed, without requiring too much work when implementing the class, since that would void the purpose of auto generation. It is common practice in object oriented programming to make members of classes private and provide public 'get'/'set' functions to access these members. Therefore, it seems reasonable that the auto generated classes should uphold this practice. A little macro magic will be a big help here. The next listing is basically listing 5, with a few additions to make life easier.

Listing 8: A better compound and some macros

```
template<typename t_type>
class TValue
{
public:
    t_type m_value;
};

template<class t_TypeList>
class TCompound : public GenScatterHierarchy< t_TypeList, TValue>
{
public:
    enum { number_of_members = Loki::TL::Length<t_TypeList>::value};

    template<unsigned int t_i>
    Loki::TL::TypeAt<t_TypeList,t_i>::Result& Member()
    {
        return Loki::Field<t_i>(*this).m_value;
    }

    template<unsigned int t_i>
    const Loki::TL::TypeAt<t_TypeList,t_i>::Result& Member() const
    {
        return Loki::Field<t_i>(*this).m_value;
    }
};

#define ACCESSORS_1(T1)\
const Loki::TL::TypeAt<TypeList,0>::Result& T1() const { return Member<0>();}\
Loki::TL::TypeAt<TypeList,0>::Result& T1() { return Member<0>();}\
const Loki::TL::TypeAt<TypeList,0>::Result& Get##T1() const { return Member<0>();}\
void Set##T1( const Loki::TL::TypeAt<TypeList,0>::Result& i_in) { Member<0>() = i_in;}

#define ACCESSORS_2(T1,T2) ACCESSORS_1(T1)\
const Loki::TL::TypeAt<TypeList,1>::Result& T2() const { return Member<1>();}\
Loki::TL::TypeAt<TypeList,1>::Result& T2() { return Member<1>();}\
const Loki::TL::TypeAt<TypeList,1>::Result& Get##T2() const { return Member<1>();}\
void Set##T2( const Loki::TL::TypeAt<TypeList,1>::Result& i_in) { Member<1>() = i_in;}

#define ACCESSORS_3(T1,T2,T3) ACCESSORS_2(T1,T2)\
const Loki::TL::TypeAt<TypeList,2>::Result& T3() const { return Member<2>();}\
Loki::TL::TypeAt<TypeList,2>::Result& T3() { return Member<2>();}\
const Loki::TL::TypeAt<TypeList,2>::Result& Get##T3() const { return Member<2>();}\
void Set##T3( const Loki::TL::TypeAt<TypeList,2>::Result& i_in) { Member<2>() = i_in;}

typedef TYPELIST_3(int,bool,float) Types;
class CTest5 : public TCompund<Types>
{
public:
    ACCESSORS_3( IntMember, BoolMember, FloatMember);
};
```

`TCompund` got a template function, `Member`, that should help to deconvolute the process of accessing the members of the base classes. Furthermore, a set of macros that creates accessor functions was added. To the casual user the class `CTest5` now looks like the class in listing 9.

Listing 9: CTest5 with the functions that the ACCESSORS macro produces

```
class CTest5
{
```

```
public:
    const int& IntMember() const;
    int& IntMember();
    const int& GetIntMember() const;
    void SetIntMember( const int& i_in);

    const bool& BoolMember() const;
    bool& BoolMember();
    const bool& GetBoolMember() const;
    void SetBoolMember( const bool& i_in);

    const float& FloatMember() const;
    float& FloatMember();
    const float& GetFloatMember() const;
    void SetFloatMember( const float& i_in);
};
```

This brings up an important point, the code in listing 8 does not exactly convey any sense of what `CTest5` is supposed to do. That is to be expected of example code, but the problem lies deeper than that. `CTest5` lacks the normal structure that one expects to see, when inspecting the functionality of a class and it is therefore hard to deduce how the class should be used. The solution is to document the code properly, but also add, what I call, phantom members. Phantom members are comments that resemble actual members, their only purpose is to show the functions added by the macros in listing 8. `CTest5` documented with phantom members would look something like:

Listing 10: CTest5 documented with phantom members

```
class CTest5
{
public:
    // Members generated by the ACCESSORS macro below
    // const int& GetIntMember() const;
    // void SetIntMember( const int& i_in);

    // const bool& GetBoolMember() const;
    // void SetBoolMember( const bool& i_in);

    // const float& GetFloatMember() const;
    // void SetFloatMember( const float& i_in);

    ACCESSORS_3( IntMember, BoolMember, FloatMember);
};
```

The framework for generating classes, that have been constructed in this and the previous section, produces classes that are on par with hand crafted classes, in terms of usability, and better in the sense that all members can be enumerated. In the next section an interface for storing data, in a flexible manner, will be developed.

2.3 Constructing a serializer interface

Previously in this section all the examples have dealt with storing data in a pretty hard-coded manner, the data was passed to a stream and that was that. From now on, the goal is nothing less than total flexibility. It should be possible to store data in any form, no matter how complex the data might be. To emphasize this point, the term 'Storing' will be dropped in favor of the term 'Serialization'. 'Storage' and 'Storing' will now be used whenever a specific data format is referred to.

To achieve total flexibility things need to be kept separated. The serialization process should be non-intrusive, meaning that the data types will not participate actively in the process. The serializer interface should be independent of the way data is stored and lastly the interface must be able to handle all types of data.

By now it is clear that an abstract serializer interface is needed, but how can one, that handles all data types, be developed? Examining what kinds of data types that are possible, is a crucial step in the right direction. The first thing that comes to mind is the built-in types, the POD types. At first glance one might think that these types can be handled by some kind of template function, but since the requirement is an abstract interface this is not possible. So the interface must contain a function for each POD type, given that there are thirteen built-in types this seems manageable. The built-in types are:

- char
- unsigned char
- short
- unsigned short
- int
- unsigned int
- long
- unsigned long
- float
- double
- long double
- wchar_t
- bool

Some of these types may or may not be the same on a given platform, for example, double and long double are often the same type.

Next come the user-defined types, which consists of enums, classes, structs and pointer types. Enums are hard to deal with, since their value[6] may vary from compiler to compiler, unless explicitly specified. Enums will therefore not be serializable in this implementation, but may be included when a fitting solution has been found.

Pointers can readily be serialized, provided that they point to a single object. If a pointer points to an array of objects, there is no way of knowing how many objects should be serialized and consequently data can be lost. Furthermore, if more than one pointer points to the same object, the result is a graph-like object structure, which could result in data being serialized multiple times and in the worst-case scenario result in an infinite loop. It is by no means impossible to serialize pointers properly, but for the sake of simplicity they have been excluded from this implementation.

Classes and structs are the same in this context and there is no point in differentiating between them, so from now on only classes will be dealt with. Classes have an inherent tree-like structure, unless they contain pointers, but pointers have just excluded so that is not an issue. This tree structure comes from the fact that classes are *compound* types, types that contain other types and so on ad infinitum. The serializer interface needs some way of dealing with compound types, but since template functions cannot appear in the interface and since having a function for each class type is unrealistic, some way of treating all classes in the same manner is needed.

Although just excluded, pointers could also point to arrays of objects, that does not mean that we cannot deal with arrays. Arrays are fundamental in programming and the basis for a lot of important data types such as C-strings, std::string and std::vector. Although C-strings are commonly used, they are considered unsafe, old fashioned, and most importantly related to pointers, so C-strings are not supported in this implementation. On the other hand, std::string and std::vector are safe and sensible ways of dealing with arrays of objects. One might think that arrays can be treated as compound objects, but there are some fundamental differences, that must be taken into consideration. Arrays will not be dealt with directly, instead objects that handles arrays, such as std::string, will be considered, these kinds of objects will be referred to as *ranges* from here on. The first big difference between compounds and ranges is that compounds can contain a lot of different types, while ranges only contain elements of one type. Some serializers might be able to take advantage of this fact, so it is important to make this distinction. The other big difference is that the order, in which the elements in a range are serialized, is very important. Consider serializing a string, it would be expected that the characters in the string were deserialized in the same order as they were serialized, otherwise the meaning of the string may be lost. Compound objects, are not sensitive to this issue as long as all elements in a compound object are serialized and deserialized, the order is of no importance.

By now it should be clear what compounds and ranges are and what their differences are, but a good way to represent them in the interface has yet to be found. Consider what happens when a compound is serialized, a scope is created and everything serialized within

[6] Enums can be seen as constant integral values.

that scope is considered to be a member of that compound. This idea works equally well for ranges and translates very well into code, where scoping is a natural concept.

The interface is based on the principle that all data types can be seen as either POD types, compound types, or range types. POD types can readily be serialized, while compounds and ranges must be 'traversed' recursively to be serialized. The 'traversal' idea works because compounds and ranges can be unfolded to a tree-like structure where every leaf is a POD type.

Before constructing the serializer interface, there is a few things that must be considered. How can two serialized compounds be distinguished from each other? They cannot, not without having some kind of identifier. So what kind of identifier will suffice, it must be something that is constant between program executions and from compiler to compiler. This means that only a hard-coded constant will do the trick, but what kind of constant? There is a number of choices, integers, UIDs, or strings. As it turns out strings are an excellent choice, not only is the chance of an identifier clash sufficiently small, unlike integers[7], but they also allow for creating data files, that are readable by humans. So, when serializing a compound a string identifier must be provided, but is that all that is needed? What about type safety, it could actually be critical for a program to be able to verify that deserialized data was of the correct type. How can the interface be supplied with type information, the POD types are not a problem since their types are already known, but the compounds and the ranges are more abstract and their type cannot be deduced from the context. Supplying the type information as a string is an option, but it turns out that there is a better alternative. Besides the things already mentioned, 'Loki' provides a small class, called TypeInfo, which acts as a wrapper for the built-in class type_info. TypeInfo allows passing type information around as a class, at run-time. TypeInfo supports comparisons and less-than operations, which means that a given implementation of the serializer interface could map these TypeInfo classes to type strings, or whatever that implementation needs to identify types. This solution is not perfect; a given implementation would need to be supplied with some appropriate mapping of types, but it frees the interface from the responsibility, which means more flexibility in dealing with types.

The responsibilities of the serializer interface have been determined and so the final interface is presented below:

Listing 11: The final serializer interface

```
class ISerializer
{
public:
    virtual ~ISerializer() = 0 {};
    virtual void Serialize( const char& i_in) = 0;
    virtual void Serialize( const unsigned char& i_in) = 0;
    virtual void Serialize( const short& i_in) = 0;
    virtual void Serialize( const unsigned short& i_in) = 0;
    virtual void Serialize( const int& i_in) = 0;
    virtual void Serialize( const unsigned int& i_in) = 0;
    virtual void Serialize( const long& i_in) = 0;
    virtual void Serialize( const unsigned long& i_in) = 0;
    virtual void Serialize( const float& i_in) = 0;
    virtual void Serialize( const double& i_in) = 0;
    virtual void Serialize( const long double& i_in) = 0;
```

[7] Not in the sense that there is not enough numbers to choose from, but rather that the chance of two programmers assigning the same number to different compounds, is greater than them choosing the same name.

```
    virtual void Serialize( const wchar_t& i_in) = 0;
    virtual void Serialize( const bool& i_in) = 0;

    // Creates a scope within the serializer
    virtual std::auto_ptr<ISerializer> SerializeCompound( const std::wstring& i_wstrName, const Loki::TypeInfo& i_type) = 0;

    // Serializing a range means that all items serialized in this scope must be of the same type
    virtual std::auto_ptr<ISerializer> SerializeRange( const std::wstring& i_wstrName, const Loki::TypeInfo& i_type) = 0;
};
```

Having rigidly defined the responsibilities of `ISerializer`, the definition holds no big surprises, except that `SerializeCompound` and `SerializeRange` returns auto pointers. The auto pointers however, are a direct consequence of the scoping idea discussed earlier, in that they both signal and enforce that once the auto pointer goes out of scope all members of that compound or range, that is currently being serialized, are assumed to have been serialized. The interface provides no form of error checking, since all errors are assumed to be reported via exceptions and so all operations that does not throw, are by definition successful. Furthermore, the returned auto pointers are always valid.

Having seen the serializer interface, there is not much point in embarking on a lengthy discussion on the deserializer, since the two are complementary and all arguments made above also holds for the deserializer. So without further ado, here is the deserializer interface:

Listing 12: The final deserializer interface

```
class IDeSerializer
{
public:
    virtual ~IDeSerializer() = 0 {};
    virtual bool DeSerialize( char& i_in) = 0;
    virtual bool DeSerialize( unsigned char& i_in) = 0;
    virtual bool DeSerialize( short& i_in) = 0;
    virtual bool DeSerialize( unsigned short& i_in) = 0;
    virtual bool DeSerialize( int& i_in) = 0;
    virtual bool DeSerialize( unsigned int& i_in) = 0;
    virtual bool DeSerialize( long& i_in) = 0;
    virtual bool DeSerialize( unsigned long& i_in) = 0;
    virtual bool DeSerialize( float& i_in) = 0;
    virtual bool DeSerialize( double& i_in) = 0;
    virtual bool DeSerialize( long double& i_in) = 0;
    virtual bool DeSerialize( wchar_t& i_in) = 0;
    virtual bool DeSerialize( bool& i_in) = 0;

    // creates a scope within the serializer
    virtual std::auto_ptr<IDeSerializer> DeSerializeCompound( const std::wstring& i_wstrName, const Loki::TypeInfo& i_type) = 0;

    // DeSerializing a range means that all item serialized in this scope must be of the same type
    virtual std::auto_ptr<IDeSerializer> DeSerializeRange( const std::wstring& i_wstrName, const Loki::TypeInfo& i_type) = 0;
};
```

`IDeSerializer` bears a striking resemblance to `ISerializer`, the only difference being that the POD functions returns bool instead of void. The boolean return value does not act as an error code, but rather as an *end of range* marker, which indicates that the range contains no more items. Stricter range checking can be performed by the deserializer itself, if the corresponding serializer, at the time of serialization, has provided the size of the range. The compound and range functions can also return an *end of range* marker in the form of a null auto pointer. Again actual errors will be reported via the exception mechanism.

3 The actual framework

The basis for the framework presented in this report was developed in section 2. To take advantage of these tools, this section presents a set of functions that ease the process of (de)serializing data. Each function serializes or deserializes a specific type. These functions, along with the `TCompound` class, the `ISerializer`, and the `IDeSerializer` interface makes up the Persistent Storage Framework, or PSF. In the actual implementation all classes and functions lives in the PSF name space, but just like in section 2 all functions are assumed to live in the global name space. The source code for the Persistent Storage Framework is listed in appendix A.

3.1 POD types

The POD type `Serialize` and `DeSerialize` functions comes in two flavors, one that creates a named compound scoping before (de)serializing and one that (de)serializes in the current scope. In this respect POD types are different from compounds and ranges, since the latter types must have a scope of their own, otherwise proper (de)serialization cannot be guaranteed. POD types however, can coexist in the same scope in a meaningful manner, a great example of this is strings. Strings are ranges of POD types, usually char or wchar_t, which coexist in the same scope, but strings themselves cannot coexist, since it would be difficult to tell where one string ended and another string began. The prototypes for the POD `Serialize` functions are as follows:

Listing 13: Prototype POD `Serialize` functions

```
void PSF::Serialize( const type& i_in, ISerializer& i_serializer)
{
    i_serializer.Serialize( i_in);
}

void PSF::Serialize( const type& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName)
{
    Serialize( i_in, *i_serializer.SerializeCompound( i_wstrName, Loki::TypeInfo(typeid(type))));
}
```

Likewise the prototypes for the POD `DeSerialize` functions are:

Listing 14: Prototype POD `DeSerialize` functions

```
bool PSF::DeSerialize( type& o_out, IDeSerializer& i_deserializer)
{
    return i_deserializer.DeSerialize( o_out);
}

bool PSF::DeSerialize( type& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName)
{
    std::auto_ptr<IDeSerializer> pDeSerializer( i_deserializer.DeSerializeCompound( i_wstrName, Loki::TypeInfo(typeid(type))));
    if ( !pDeSerializer.get()) return false;

    return DeSerialize( o_out, *pDeSerializer);
}
```

The `DeSerialize` functions returns a boolean value, just like their relatives in the `IDeSerializer` interface.

These prototypes are prime candidates for a template implementation, but sadly my compiler[8] does not support partial ordering of template functions and they have therefore

[8] Microsoft Visual C++ .NET 2002

been explicitly defined as non-template functions, although a set of template functions have also been defined for compilers with better template support.

3.2 STL containers

All STL containers can be seen as ranges, that means that unlike POD types, they need to be (de)serialized in a separate named scope. However, it is not always possible for the programmer to supply a name for the scope. Consider for example a list of strings, it may be possible to supply a name for the list scope, but all strings in the list will be automatically serialized by the framework and it is therefore not possible to supply a name for the sub scopes. One might argue that the `Serialize` function should take two name parameters, one for the list scope and one for the sub scopes, but in the case of a list of lists of strings three name parameters would be needed. A better alternative is therefore to supply a default name, for the cases where no name can be supplied.

The process of serializing a STL container is therefore to, first create a named scope and them serialize all items in the container in that scope. Since all STL containers support the iterator mechanism, it seems reasonable to take advantage of this. So before creating `Serialize` functions for all STL containers, a function that serialize an iterator range will be defined:

Listing 15: Function that serializes an iterator range

```
template<class t_iterType>
void SerializeRange( t_iterType i_iterBegin, t_iterType i_iterEnd, ISerializer& i_serializer,
    const std::wstring& i_wstrName, const Loki::TypeInfo& i_typeInfo)
{
    std::auto_ptr<ISerializer> pSerializer( i_serializer.SerializeRange( i_wstrName, i_typeInfo));
    for ( ; i_iterBegin != i_iterEnd; ++i_iterBegin)
    {
        Serialize( *i_iterBegin, *pSerializer);
    }
}
```

This functions starts by creating a scope, with the type of the container that is currently being serialized, and then continues to serialize all items in that scope.

Having defined the `SerializeRange` function, it is time to define `Serialize` functions for some commonly used STL containers:

Listing 16: `Serialize` functions for STL containers, `basic_string`, `list`, and `vector`

```
template<typename t_type>
void Serialize( const std::basic_string<t_type>& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName = L"String")
{
    SerializeRange( i_in.begin(), i_in.end(), i_serializer, i_wstrName, Loki::TypeInfo(typeid(std::basic_string<void>)));
}

template<typename t_type>
void Serialize( const std::list<t_type>& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName = L"List")
{
    SerializeRange( i_in.begin(), i_in.end(), i_serializer, i_wstrName, Loki::TypeInfo(typeid(std::list<void>)));
}

template<typename t_type>
void Serialize( const std::vector<t_type>& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName = L"Vector")
{
    SerializeRange( i_in.begin(), i_in.end(), i_serializer, i_wstrName, Loki::TypeInfo(typeid(std::vector<void>)));
}
```

The STL container `Serialize` functions are much as could be expected, they are templated on the contained type of the container and they have a default name argument if none was

supplied. The only remarkable thing is that the type info supplied to `SerializeRange` is not the actual type of the container being serialized, but rather the type of the container templated on type void. The reason for this is that the mapping of types mentioned in section 2.3, would be needlessly complicated if a mapping was needed for all possible instantiations of a given container, instead of just one mapping that includes all instantiations of a given container.

The `Serialize` functions given in listing 16 can not be generalized to all possible STL containers, some containers, like the associative container `map`, does not contain the actual elements it is templated on, but rather a `pair` of those elements. So in order to support associative containers and the `pair` construct in general, a little more work is needed:

Listing 17: `Serialize` functions for `pair` and `map`

```
template<typename t_typeFirst,typename t_typeSecond>
void Serialize( const std::pair<t_typeFirst,t_typeSecond>& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName = L"Pair")
{
    std::auto_ptr<ISerializer> pSerializer( i_serializer.SerializeCompound( i_wstrName, Loki::TypeInfo(typeid(std::pair<void,void>))));
    Serialize( i_in.first, *pSerializer, L"First");
    Serialize( i_in.second, *pSerializer, L"Second");
}

template<typename t_key,typename t_type>
void Serialize( const std::map<t_key,t_type>& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName = L"Map")
{
    SerializeRange( i_in.begin(), i_in.end(), i_serializer, i_wstrName, Loki::TypeInfo(typeid(std::map<void,void>)));
}
```

The purpose of the functions in the above listing is obvious. The first function serializes a `pair`, by first creating a scope and then serializing it members in that scope. The second function serializes a `map`, just like `basic_string`, `list`, and `vector` was serialized. In both cases the type info supplied to `SerializeRange`, is the class templated on type void. This concludes the work on the STL `Serialize` functions and it is now time to take a look at their counterparts, the `DeSerialize` functions.

The STL `DeSerialize` functions, follow much the same pattern as the `Serialize` functions, in that their common functionality can be factorized out into a `DeSerializeRange` function:

Listing 18: Function that deserializes a range

```
template<class t_iteratorType>
bool DeSerializeRange( t_iteratorType o_iter, IDeSerializer& i_deserializer, const std::wstring& i_wstrName, const Loki::TypeInfo& i_typeInfo)
{
    std::auto_ptr<IDeSerializer> pDeSerializer( i_deserializer.DeSerializeRange( i_wstrName, i_typeInfo));
    if ( !pDeSerializer.get()) return false;

    t_iteratorType::container_type::value_type val;
    while ( DeSerialize( val, *pDeSerializer))
    {
        *o_iter = val;
        o_iter++;
    }
    return true;
}
```

The `DeSerializeRange` functions returns a boolean value, as all `DeSerialize` functions should. Like its counterpart `SerializeRange` it works on iterators, rather than on the container itself. The reason for this is that by working on iterators, the `SerializeRange` and `DeSerializeRange` functions can used to (de)serialize ranges that does not conform to the STL container guidelines, as long as they support the iterator mechanism. If a given non-STL container does not support iterators it is relatively easy to provide a wrapper class

that can act as an iterator.

The *while* loop in `DeSerializeRange` deserves some attention, since it is here the actual work goes on. The loop continues as long as the `DeSerialize` function does not return false, indicating that all items from the range have been deserialized. Inside the loop, the output iterator is assigned the newly deserialized value and incremented. The only constraint that `DeSerializeRange` poses on the data types being deserialized is that they must have a default constructor.

All the STL container `DeSerialize` functions are defined in terms of `DeSerializeRange`:

Listing 19: Functions for deserializing common STL containers

```
template<typename t_type>
bool DeSerialize( std::basic_string<t_type>& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName = L"String")
{
    std::back_insert_iterator<std::basic_string<t_type> > iter( o_out);
    return DeSerializeRange( iter, i_deserializer, i_wstrName, Loki::TypeInfo(typeid(std::basic_string<void>)));
}

template<typename t_type>
bool DeSerialize( std::list<t_type>& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName = L"List")
{
    std::back_insert_iterator<std::list<t_type> > iter( o_out);
    return DeSerializeRange( iter, i_deserializer, i_wstrName, Loki::TypeInfo(typeid(std::list<void>)));
}

template<typename t_type>
bool DeSerialize( std::vector<t_type>& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName = L"Vector")
{
    std::back_insert_iterator<std::vector<t_type> > iter( o_out);
    return DeSerializeRange( iter, i_deserializer, i_wstrName, Loki::TypeInfo(typeid(std::vector<void>)));
}

template<typename T>
T& UnConst( const T& i_in)
{
    return const_cast<T&>( i_in);
}

template<typename t_typeFirst,typename t_typeSecond>
bool DeSerialize( std::pair<t_typeFirst,t_typeSecond>& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName = L"Pair")
{
    std::auto_ptr<IDeSerializer> pDeSerializer( i_deserializer.DeSerializeCompound( i_wstrName, Loki::TypeInfo(typeid(std::pair<void,void>))));
    if ( !pDeSerializer.get()) return false;

    // Should be compile-time const removal
    return DeSerialize( UnConst(o_out.first), *pDeSerializer, L"First") &&
        DeSerialize( o_out.second, *pDeSerializer, L"Second");
}

template<typename t_key,typename t_type>
bool DeSerialize( std::map<t_key,t_type>& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName = L"Map")
{
    std::insert_iterator<std::map<t_key,t_type> > iter( o_out, o_out.end());
    return DeSerializeRange( iter, i_deserializer, i_wstrName, Loki::TypeInfo(typeid(std::map<void,void>)));
}
```

The only thing in listing 19 worth noting, is the removal of const modifiers in the `pair` `DeSerialize` function. This is necessary due to the fact, that the elements of a `map` are defined as `pair<const Key, Type>`, so in order for the framework to function properly, the const modifier must be removed from first member of the `pair`.

3.3 Compound types

Before defining `Serialize` and `DeSerialize` functions for compound types, a few additions to the `TCompound` class defined in listing 8 is needed. The first addition deals with the problem of naming the members. Every member of a compound class is serialized in its own scope and to create a scope we need a name. The names for the members are shared be-

tween all instances of a given compound, so in order to reduce the memory footprint of the compound, the names should be defined as statics. Below is a new version of TCompound supporting names for members and a class CTest6 defining these names:

Listing 20: Compound class with support for names

```
template<typename t_type>
class TValue
{
public:
    t_type m_value;
};

template<class t_TypeList, class t_Unique>
class TCompound : public GenScatterHierarchy< t_TypeList, TValue>
{
    public:
    enum { number_of_members = Loki::TL::Length<t_TypeList>::value};
    static const wchar_t* ms_namelist[];

    template<unsigned int t_i>
    Loki::TL::TypeAt<t_TypeList,t_i>::Result& Member()
    {
        return Loki::Field<t_i>(*this).m_value;
    }

    template<unsigned int t_i>
    const Loki::TL::TypeAt<t_TypeList,t_i>::Result& Member() const
    {
        return Loki::Field<t_i>(*this).m_value;
    }
};

typedef TYPELIST_3(int,bool,float) Types;
class CTest6 : public TCompund<Types,CTest6>
{
public:
    ACCESSORS_3( IntMember, BoolMember, FloatMember);
};
const wchar_t* CTest6::ms_namelist[] = { L"IntMember", L"BoolMember", L"FloatMember", 0};
```

The only new additions to TCompound, are the template parameter t_Unique and the static const array of strings ms_namelist. The content of ms_namelist is defined below class CTest6 and is by definition an array of strings terminated by a null pointer. A string in the array denotes the name of the member that shares that strings positional index, so the second string 'BoolMember' is the name of the second member of the class namely the bool. The names have been chosen so that they match the names of the accessor functions, this is not a requirement, but is generally recommended to maintain consistency. The extra template parameter is needed because a static member of a template, is shared between all instances of that template sharing the same template parameters. So if a class, unrelated to CTest6, was to inherit from TCompound with the same template parameters as CTest6, they would also share the same instance of ms_namelist, which is undesirable. So in order to differentiate between TCompound templates a second template parameter, which guarantees that this particular instance of TCompound is unique, is needed. The argument passed as the second template parameter could in theory be anything, as long as it guarantees uniqueness, but later on this template parameter will serve a second purpose, that requires it to be the class inheriting from TCompound.

The second issue that must be addressed before defining Serialize and DeSerialize functions for TCompound is that of inheritance. The compounds themselves are very much defined in terms of inheritance, so it seems obvious that compounds should be able to inherit from other compounds and still be serializable. Unfortunately this is a complicated matter, so to keep things simple this framework only supports single inheritance. TCompound itself does not require big changes to support inheritance:

Listing 21: Final compound class

```
template<typename t_type>
class TValue
{
public:
    t_type m_value;
};

template<class t_TypeList,class t_Unique,class t_Base = Loki::EmptyType>
class TCompound : public Loki::GenScatterHierarchy< t_TypeList, TValue>, public t_Base
{
public:
    enum { number_of_members = Loki::TL::Length<t_TypeList>::value};
    static const wchar_t* ms_namelist[];
    typedef t_TypeList TypeList;
    typedef t_Unique Unique;
    typedef t_Base Base;
    typedef Loki::GenScatterHierarchy< t_TypeList, TValue> ScatterBase;

    template<unsigned int t_i>
    Loki::TL::TypeAt<TypeList,t_i>::Result& Member()
    {
        return Loki::Field<t_i>(static_cast<ScatterBase&>(*this)).m_value;
    }

    template<unsigned int t_i>
    const Loki::TL::TypeAt<TypeList,t_i>::Result& Member() const
    {
        return Loki::Field<t_i>(static_cast<const ScatterBase&>(*this)).m_value;
    }
};
```

As can be seen, the changes are few, an extra template parameter denotes the desired base class and a few typedefs to keep things readable. The base class template parameter defaults to `Loki::EmptyClass` which is, as the name suggests, an empty class, should no base class be desired. Otherwise the only change is the addition of the static_cast inside the `Member` functions. This is needed because when a `TCompound` inherits from another `TCompound`, the former `TCompound` effectively inherits from `GenScatterHierarchy` twice resulting in ambiguities when using the `Field` functions to access members. The static_cast help disambiguate the access, thus solving the problem.

`TCompound`, as it appears in listing 21, is in its final form and the `Serialize` function is therefore presented below:

Listing 22: `Serialize` function for `TCompound`

```
template<class t_TypeList,class t_Unique,class t_Base>
void Serialize( const TCompound<t_TypeList,t_Unique,t_Base>& i_compound,
            ISerializer& i_serializer,
            const std::wstring& i_wstrName = L"Compound")
{
    std::auto_ptr<ISerializer> pSerializer( i_serializer.SerializeCompound( i_wstrName, typeid( t_Unique)));
    TCompoundSerializer<t_TypeList,t_Unique,t_Base> compoundSerializer( i_compound);
    compoundSerializer.Serialize( *pSerializer);
}
```

The `Serialize` function for compounds resembles the STL `Serialize` functions, in that it creates a scope with the desired name and the type of the class that inherited from it. The serialization of the members themselves are delegated to the helper class `TCompoundSerializer`:

Listing 23: Helper class for serializing compounds

```
template<class t_TypeList,class t_Unique,class t_Base>
class TCompoundSerializer
{
public:
    typedef TCompound<t_TypeList,t_Unique,t_Base> Compound;

    TCompoundSerializer( const Compound& i_compound) :
        m_compound( i_compound)
    {
    }
```

```
    void Serialize( ISerializer& i_serializer) const
    {
        TInner<Compound> inner( m_compound);
        inner.Serialize( i_serializer);
    }

protected:

    template<class t_InnerCompund>
    class TInner
    {
    public:
        TInner( const t_InnerCompund& i_innerCompound) :
            m_innerCompound( i_innerCompound)
        {
        }

        void Serialize( ISerializer& i_serializer) const
        {
            TInner<t_InnerCompund::Base> inner( m_innerCompound);
            inner.Serialize( i_serializer);
            SerializeInternal<0>( i_serializer);
        }
    private:
        enum { number_of_loops = t_InnerCompund::number_of_members-1};

        template<int t_i>
        void SerializeInternal( ISerializer& i_serializer) const
        {
            PSF_ASSERT( t_InnerCompund::ms_namelist[t_i]);
            const wchar_t* pwszName = t_InnerCompund::ms_namelist[t_i];
            ::Serialize( m_innerCompound.Member<t_i>(), i_serializer, pwszName);
            SerializeInternal<t_i+1>( i_serializer);
        }

        template<>
        void SerializeInternal<number_of_loops>( ISerializer& i_serializer) const
        {
            PSF_ASSERT( t_InnerCompund::ms_namelist[number_of_loops]);
            const wchar_t* pwszName = t_InnerCompund::ms_namelist[number_of_loops];
            ::Serialize( m_innerCompound.Member<number_of_loops>(), i_serializer, pwszName);
        }
        const t_InnerCompund& m_innerCompound;
    };

    template<>
    class TInner<Loki::EmptyType>
    {
    public:
        TInner<Loki::EmptyType>( const t_InnerCompund& i_innerCompound) {}
        void Serialize( ISerializer& i_serializer) const {}
    };

    const Compound& m_compound;
};
```

`TCompoundSerializer` is quite a mouthful, but basically its purpose is to encapsulate two compile-time recursions. First it recurses through the compounds base classes and then for each compound, it iterates through its members, serializing them along the way. The recursions themselves are handled by the nested class `TInner`, which recurses until the class being serialized is of type `Loki::EmptyType`, at which point the recursion terminates, because `TInner` is specialized on `Loki::EmptyType`. A compounds base class is serialized before the compound itself, but they are both serialized within the same scope. The base class recursion is initiated by the `Serialize` function in `TInner`, while the member iteration is performed by `SerializeInternal` in `TInner`. `SerializeInternal` calls the scope version of the general `Serialize` function, for the member currently being serialized, with the name supplied by `ms_namelist`. As a small safety precaution `SerializeInternal` asserts that the strings in `ms_namelist` are not null, since that would be the case at some point, if the name list did not contain an entry for each member.

The `DeSerialize` function for compounds, hardly differs from its `Serialize` counter part, except that as with all `DeSerialize` functions this one also returns a boolean value, otherwise it is an exact replica of the code in listing 22 and 23.

Listing 24: `DeSerialize` function for `TCompound`, with helper class

```
template<class t_TypeList,class t_Unique,class t_Base>
class TCompoundDeSerializer
{
public:
    typedef TCompound<t_TypeList,t_Unique,t_Base> Compound;

    TCompoundDeSerializer( Compound& i_compound) :
        m_compound( i_compound)
    {
    }

    bool DeSerialize( IDeSerializer& i_deserializer)
    {
        TInner<Compound> inner( m_compound);
        return inner.DeSerialize( i_deserializer);
    }

private:
    template<class t_InnerCompund>
    class TInner
    {
    public:
        TInner( t_InnerCompund& i_innerCompound) :
            m_innerCompound( i_innerCompound)
        {
        }

        bool DeSerialize( IDeSerializer& i_deserializer)
        {
            TInner<t_InnerCompund::Base> inner( m_innerCompound);
            bool bResult = inner.DeSerialize( i_deserializer);
            return bResult && DeSerializeInternal<0>( i_deserializer);
        }
    private:

        enum { number_of_loops = t_InnerCompund::number_of_members-1};

        template<int t_i>
        bool DeSerializeInternal( IDeSerializer& i_deserializer)
        {
            PSF_ASSERT( t_InnerCompund::ms_namelist[t_i]);
            const wchar_t* pwszName = t_InnerCompund::ms_namelist[t_i];
            bool bResult = PSF::DeSerialize( m_innerCompound.Member<t_i>(), i_deserializer, pwszName);
            return bResult && DeSerializeInternal<t_i+1>( i_deserializer);
        }

        template<>
        bool DeSerializeInternal<number_of_loops>( IDeSerializer& i_deserializer)
        {
            PSF_ASSERT( t_InnerCompund::ms_namelist[number_of_loops]);
            const wchar_t* pwszName = t_InnerCompund::ms_namelist[number_of_loops];
            return PSF::DeSerialize( m_innerCompound.Member<number_of_loops>(), i_deserializer, pwszName);
        }

        t_InnerCompund& m_innerCompound;
    };

    template<>
    class TInner<Loki::EmptyType>
    {
    public:
        TInner<Loki::EmptyType>( const t_InnerCompund& i_innerCompound)
        {
        }

        bool DeSerialize( IDeSerializer& i_deserializer) const
        {
            return true;
        }
    };

    Compound& m_compound;
};

template<class t_TypeList,class t_Unique,class t_Base>
bool DeSerialize( TCompound<t_TypeList,t_Unique,t_Base>& o_compound,
                IDeSerializer& i_deserializer,
                const std::wstring& i_wstrName = L"Compound")
{
    std::auto_ptr<IDeSerializer> pDeSerializer( i_deserializer.DeSerializeCompound( i_wstrName, typeid(t_Unique)));
```

```
    if ( !pDeSerializer.get()) return false;
    TCompoundDeSerializer<t_TypeList,t_Unique,t_Base> compoundDeSerializer( o_compound);
    return compoundDeSerializer.DeSerialize( *pDeSerializer);
}
```

3.4 Legacy types

Serialization of legacy compounds, classes not generated with the help of `TCompound`, has actually already been covered in section 3.2, where `pair` was serialized as a legacy compound would have been. For the sake of completeness, this section contains a small walk through of how to (de)serialize legacy types.

There two ways to serialize legacy types, the choice depending on how accessible the members of the type are. This assumes that the type is a compound, legacy ranges have been discussed in section 3.2. If all members of a class are freely accessible, either as public members or through accessor functions, the best choice is to let external `Serialize` and `DeSerialize` functions handle everything and thus acting in an non-intrusive manner. This is the case in the following example, where `Serialize` and `DeSerialize` functions are created for a small class, representing a point in three-dimensional space:

Listing 25: Non-intrusive `Serialize` and `DeSerialize` functions

```
class CPoint3
{
public:
    float x;
    float y;
    float z;
}

void Serialize( const CPoint3& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName = L"Point3")
{
    std::auto_ptr<ISerializer> pSerializer( i_serializer.SerializeCompound( i_wstrName, Loki::TypeInfo(typeid(CPoint3))));
    Serialize( x, *pSerializer);
    Serialize( y, *pSerializer);
    Serialize( z, *pSerializer);
}

bool DeSerialize( CPoint3& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName = L"Point3")
{
    std::auto_ptr<IDeSerializer> pDeSerializer( i_deserializer.DeSerializeCompound( i_wstrName, Loki::TypeInfo(typeid(CPoint3))));
    if ( !pDeSerializer.get()) return false;

    return DeSerialize( x, *pDeSerializer) &&
        DeSerialize( y, *pDeSerializer) &&
        DeSerialize( z, *pDeSerializer);
}
```

The above example is actually quite interesting because, it uses the non-scoping version of the `Serialize` and `DeSerialize` functions. This is possible because all members of `CPoint3` are POD types and can therefore live in the same scope, as explained in section 3.1. The danger lies in the fact that things serialized in a compound scope, without being in their own named scope, are not guaranteed to be deserialized in the same order as they were serialized. This can be remedied by serializing `CPoint3` as a range instead of a compound.

The second example deals with a rather restricted class, that does not allow direct access to its members. In this situation it is necessary to retrofit the class with extra helper functions.

Listing 26: Intrusive `Serialize` and `DeSerialize` functions

```
class CRestricted
{
public:
    void Serialize( ISerializer& i_serializer)
    {
        Serialize( m_member1, i_serializer, L"Member1");
        Serialize( m_member2, i_serializer, L"Member2");
    }

    bool DeSerialize( IDeSerializer& i_deserializer)
    {
        return DeSerialize( m_member1, i_deserializer, L"Member1") &&
            DeSerialize( m_member2, i_deserializer, L"Member2");
    }
private:
    int m_member1;
    std::string m_member2;
}

void Serialize( const CRestricted& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName = L"Restricted")
{
    std::auto_ptr<ISerializer> pSerializer( i_serializer.SerializeCompound( i_wstrName, Loki::TypeInfo(typeid(CRestricted))));
    i_in.Serialize( *pSerializer);
}

bool DeSerialize( CRestricted& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName = L"Restricted")
{
    std::auto_ptr<IDeSerializer> pDeSerializer( i_deserializer.DeSerializeCompound( i_wstrName, Loki::TypeInfo(typeid(CRestricted))));
    if ( !pDeSerializer.get()) return false;

    return o_out.DeSerialize( *pDeSerializer);
}
```

3.5 The serializers

The implementation of the (de)serializers developed for this framework, will not be discussed in the report. Instead there will be a brief summary of the XML (de)serializer, the binary (de)serializer and the pretty print serializer.

XML (de)serializer

The XML (de)serializer, is implemented as two separate classes, one that serializes data to an XML file and one that deserializes data from an XML file. Both classes requires a type map that maps Loki::TypeInfo to strings. The serializer uses the type map to add type information to the XML file, the deserializer uses it to perform a rudimentary form of type checking. The XML (de)serializer also supports range sizes, that means that the size of a range is written in the XML file and used for checking that all data is present when deserializing. Lastly the deserializer also requires an `IChecker` interface that determines the desired checking policy. Current checking policies include, none or null, assert, and exceptions. The possible errors are:

Missing type Indicates that no type info was present in the data stream.

Type mismatch Indicates that the type of data in the stream does not match the type supplied by the program.

Unknown type Indicates that the program failed to supply a type identifier.

Semantic error Indicates that the format of data in the stream does not match the expected type. For example, the stream contains the string 'abc', but the program expects an integer.

Range size error Indicates that an error occurred while deserializing a range. The expected size of the range did not match the number of items deserialized from that range.

The names supplied when creating scopes are used as tag names for the XML node that represent the scope. The XML parser used is Xerces XML Parser.

Binary (de)serializer

The binary (de)serializer is much simpler than its XML counter part. It does not support types, nor does it checking policies. The file format used is a chunk based format, where each scope is denoted by a name and a length. The name of a chunk is actually an integer hash value of the name supplied when creating the scope. The binary (de)serializer can operate on all random-access binary streams. Despite its simplicity the binary (de)serializer is actually quite robust. The binary format is designed to use as little space as possible and the overhead of a scope is only 8 bytes.

Pretty Print serializer

The pretty print serializer, serializes the data as raw text. It works on all text stream. Scoping is denoted by indention. The serializer does not support types, but could easily be extended to do so.

4 Test

Testing of the Persistent Storage Framework is performed by the program in appendix B.1, the results of the test can be inspected in appendix B.2, and the data files not generated by the program itself is listed in appendix B.3.

The program tests the framework by serializing various data types to the serializers, deserializing the same data again and finally outputs it in the log file, through the pretty print serializer. The data types that are serialized include a number of POD types, some STL containers, a simple and a complex compound type, and a compound that inherits from another compound. The data is both deserialized in the correct order and in reverse, to test the frameworks resilience to mangled data files.

The programs also test various error conditions. Only the XML deserializer supports error checking, so no error checking is performed for the binary deserializer. The XML deserializer is tested with all the input files in appendix B.3, which test for missing types, mismatching types, semantic errors, and range size errors. The XML deserializer is also tested with a valid XML file, but with an incomplete type map, thus testing for unknown types.

The result of the test can be seen in appendix B.2 and 'Log.txt' in particular. All test results are as expected and the framework appears to function as intended.

References

[1] Andrei Alexandrsecu.
Modern C++ Design.
Addison-Wesley, 2001.

A PSF Source listings

A.1 Compound class

Listing 27: "Compound.hpp"

```
#ifndef COMPOUND_HPP
#define COMPOUND_HPP

#include "Loki/Typelist.h"
#include "Loki/HierarchyGenerators.h"

namespace PSF
{
    template<class t_type>
    class TValue
    {
    public:
        t_type m_value;
    };

    template<class t_TypeList, class t_Unique, class t_Base = Loki::EmptyType>
    class TCompound : public Loki::GenScatterHierarchy< t_TypeList, TValue>, public t_Base
    {
    public:
        enum { number_of_members = Loki::TL::Length<t_TypeList>::value};
        static const wchar_t* ms_namelist[];
        typedef t_TypeList TypeList;
        typedef t_Unique Unique;
        typedef t_Base Base;
        typedef Loki::GenScatterHierarchy< t_TypeList, TValue> ScatterBase;

        template<unsigned int t_i>
        Loki::TL::TypeAt<TypeList, t_i>::Result& Member()
        {
            return Loki::Field<t_i>(static_cast<ScatterBase&>(*this)).m_value;
        }

        template<unsigned int t_i>
        const Loki::TL::TypeAt<TypeList, t_i>::Result& Member() const
        {
            return Loki::Field<t_i>(static_cast<const ScatterBase&>(*this)).m_value;
        }
    };
};

#define ACCESSORS_1(T1)\
const Loki::TL::TypeAt<TypeList,0>::Result& T1() const { return Member<0>();}\
Loki::TL::TypeAt<TypeList,0>::Result& T1() { return Member<0>();}\
const Loki::TL::TypeAt<TypeList,0>::Result& Get##T1() const { return Member<0>();}\
void Set##T1( const Loki::TL::TypeAt<TypeList,0>::Result& i_in) { Member<0>() = i_in;}

#define ACCESSORS_2(T1,T2) ACCESSORS_1(T1)\
const Loki::TL::TypeAt<TypeList,1>::Result& T2() const { return Member<1>();}\
Loki::TL::TypeAt<TypeList,1>::Result& T2() { return Member<1>();}\
const Loki::TL::TypeAt<TypeList,1>::Result& Get##T2() const { return Member<1>();}\
void Set##T2( const Loki::TL::TypeAt<TypeList,1>::Result& i_in) { Member<1>() = i_in;}

#define ACCESSORS_3(T1,T2,T3) ACCESSORS_2(T1,T2)\
const Loki::TL::TypeAt<TypeList,2>::Result& T3() const { return Member<2>();}\
Loki::TL::TypeAt<TypeList,2>::Result& T3() { return Member<2>();}\
const Loki::TL::TypeAt<TypeList,2>::Result& Get##T3() const { return Member<2>();}\
void Set##T3( const Loki::TL::TypeAt<TypeList,2>::Result& i_in) { Member<2>() = i_in;}

#define ACCESSORS_4(T1,T2,T3,T4) ACCESSORS_3(T1,T2,T3)\
const Loki::TL::TypeAt<TypeList,3>::Result& T4() const { return Member<3>();}\
Loki::TL::TypeAt<TypeList,3>::Result& T4() { return Member<3>();}\
const Loki::TL::TypeAt<TypeList,3>::Result& Get##T4() const { return Member<3>();}\
void Set##T4( const Loki::TL::TypeAt<TypeList,3>::Result& i_in) { Member<3>() = i_in;}

#define ACCESSORS_5(T1,T2,T3,T4,T5) ACCESSORS_4(T1,T2,T3,T4)\
const Loki::TL::TypeAt<TypeList,4>::Result& T5() const { return Member<4>();}\
Loki::TL::TypeAt<TypeList,4>::Result& T5() { return Member<4>();}\
const Loki::TL::TypeAt<TypeList,4>::Result& Get##T5() const { return Member<4>();}\
void Set##T5( const Loki::TL::TypeAt<TypeList,4>::Result& i_in) { Member<4>() = i_in;}

#define ACCESSORS_6(T1,T2,T3,T4,T5,T6) ACCESSORS_5(T1,T2,T3,T4,T5)\
const Loki::TL::TypeAt<TypeList,5>::Result& T6() const { return Member<5>();}\
Loki::TL::TypeAt<TypeList,5>::Result& T6() { return Member<5>();}\
const Loki::TL::TypeAt<TypeList,5>::Result& Get##T6() const { return Member<5>();}\
void Set##T6( const Loki::TL::TypeAt<TypeList,5>::Result& i_in) { Member<5>() = i_in;}

#define ACCESSORS_7(T1,T2,T3,T4,T5,T6,T7) ACCESSORS_6(T1,T2,T3,T4,T5,T6)\
const Loki::TL::TypeAt<TypeList,6>::Result& T7() const { return Member<6>();}\
Loki::TL::TypeAt<TypeList,6>::Result& T7() { return Member<6>();}\
```

```
const Loki::TL::TypeAt<TypeList,6>::Result& Get##T7() const { return Member<6>();}\
void Set##T7( const Loki::TL::TypeAt<TypeList,6>::Result& i_in) { Member<6>() = i_in;}

#define ACCESSORS_8(T1,T2,T3,T4,T5,T6,T7,T8) ACCESSORS_7(T1,T2,T3,T4,T5,T6,T7)\
const Loki::TL::TypeAt<TypeList,7>::Result& T8() const { return Member<7>();}\
Loki::TL::TypeAt<TypeList,7>::Result& T8() { return Member<7>();}\
const Loki::TL::TypeAt<TypeList,7>::Result& Get##T8() const { return Member<7>();}\
void Set##T8( const Loki::TL::TypeAt<TypeList,7>::Result& i_in) { Member<7>() = i_in;}

#define ACCESSORS_9(T1,T2,T3,T4,T5,T6,T7,T8,T9) ACCESSORS_8(T1,T2,T3,T4,T5,T6,T7,T8)\
const Loki::TL::TypeAt<TypeList,8>::Result& T9() const { return Member<8>();}\
Loki::TL::TypeAt<TypeList,8>::Result& T9() { return Member<8>();}\
const Loki::TL::TypeAt<TypeList,8>::Result& Get##T9() const { return Member<8>();}\
void Set##T9( const Loki::TL::TypeAt<TypeList,8>::Result& i_in) { Member<8>() = i_in;}

#define ACCESSORS_10(T1,T2,T3,T4,T5,T6,T7,T8,T9,T10) ACCESSORS_9(T1,T2,T3,T4,T5,T6,T7,T8,T9)\
const Loki::TL::TypeAt<TypeList,9>::Result& T10() const { return Member<9>();}\
Loki::TL::TypeAt<TypeList,9>::Result& T10() { return Member<9>();}\
const Loki::TL::TypeAt<TypeList,9>::Result& Get##T10() const { return Member<9>();}\
void Set##T10( const Loki::TL::TypeAt<TypeList,9>::Result& i_in) { Member<9>() = i_in;}

#define ACCESSORS_11(T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11) ACCESSORS_10(T1,T2,T3,T4,T5,T6,T7,T8,T9,T10)\
const Loki::TL::TypeAt<TypeList,10>::Result& T11() const { return Member<10>();}\
Loki::TL::TypeAt<TypeList,10>::Result& T11() { return Member<10>();}\
const Loki::TL::TypeAt<TypeList,10>::Result& Get##T11() const { return Member<10>();}\
void Set##T11( const Loki::TL::TypeAt<TypeList,10>::Result& i_in) { Member<10>() = i_in;}

#endif
```

A.2 Serializer class

Listing 28: "Serializer.hpp"

```
#ifndef SERIALIZER_HPP
#define SERIALIZER_HPP

#include <string>
#include <memory>
#include "Loki/TypeInfo.h"

namespace PSF
{
    class ISerializer
    {
    public:
        virtual ~ISerializer() = 0 {};

        virtual void Serialize( const char& i_in) = 0;
        virtual void Serialize( const unsigned char& i_in) = 0;
        virtual void Serialize( const short& i_in) = 0;
        virtual void Serialize( const unsigned short& i_in) = 0;
        virtual void Serialize( const int& i_in) = 0;
        virtual void Serialize( const unsigned int& i_in) = 0;
        virtual void Serialize( const long& i_in) = 0;
        virtual void Serialize( const unsigned long& i_in) = 0;
        virtual void Serialize( const float& i_in) = 0;
        virtual void Serialize( const double& i_in) = 0;
        // same as double
        // virtual void Serialize( const long double& i_in) = 0;
        virtual void Serialize( const bool& i_in) = 0;
        virtual void Serialize( const wchar_t& i_in) = 0;

        // creates a scope within the serializer
        virtual std::auto_ptr<ISerializer> SerializeCompound( const std::wstring& i_wstrName, const Loki::TypeInfo& i_type) = 0;

        // Serializing a range means that all item serialized in this scope must be of the same type
        virtual std::auto_ptr<ISerializer> SerializeRange( const std::wstring& i_wstrName, const Loki::TypeInfo& i_type) = 0;
    };
};
#endif
```

A.3 Serialize functions

Listing 29: "Serialize.hpp"

```
#ifndef PSF_SERIALIZE_HPP
#define PSF_SERIALIZE_HPP
```

```
#include <list>
#include <vector>
#include <map>
#include <set>

#include "Serializer.hpp"
#include "Assert.hpp"

#include "Compound.hpp"

#include "CompilerConfig.hpp"

namespace PSF
{
    // Serialize PSF compunds
    template<class t_TypeList,class t_Unique,class t_Base>
    class TCompoundSerializer
    {
    public:
        typedef TCompound<t_TypeList,t_Unique,t_Base> Compound;

        TCompoundSerializer( const Compound& i_compound) :
            m_compound( i_compound)
        {
        }

        void Serialize( ISerializer& i_serializer) const
        {
            TInner<Compound> inner( m_compound);
            inner.Serialize( i_serializer);
        }

    private:
        template<class t_InnerCompund>
        class TInner
        {
        public:
            TInner( const t_InnerCompund& i_innerCompound) :
                m_innerCompound( i_innerCompound)
            {
            }

            void Serialize( ISerializer& i_serializer) const
            {
                TInner<t_InnerCompund::Base> inner( m_innerCompound);
                inner.Serialize( i_serializer);
                SerializeInternal<0>( i_serializer);
            }
        private:

            enum { number_of_loops = t_InnerCompund::number_of_members-1};

            template<int t_i>
            void SerializeInternal( ISerializer& i_serializer) const
            {
                PSF_ASSERT( t_InnerCompund::ms_namelist[t_i]);
                const wchar_t* pwszName = t_InnerCompund::ms_namelist[t_i];
                PSF::Serialize( m_innerCompound.Member<t_i>(), i_serializer, pwszName);
                SerializeInternal<t_i+1>( i_serializer);
            }

            template<>
            void SerializeInternal<number_of_loops>( ISerializer& i_serializer) const
            {
                PSF_ASSERT( t_InnerCompund::ms_namelist[number_of_loops]);
                const wchar_t* pwszName = t_InnerCompund::ms_namelist[number_of_loops];
                PSF::Serialize( m_innerCompound.Member<number_of_loops>(), i_serializer, pwszName);
            }

            const t_InnerCompund& m_innerCompound;
        };

        template<>
        class TInner<Loki::EmptyType>
        {
        public:
            TInner<Loki::EmptyType>( const t_InnerCompund& i_innerCompound)
            {
            }

            void Serialize( ISerializer& i_serializer) const
            {
            }
        };

        const Compound& m_compound;
    };
```

```
    template<class t_TypeList,class t_Unique,class t_Base>
    void Serialize( const TCompound<t_TypeList,t_Unique,t_Base>& i_compound,
                    ISerializer& i_serializer,
                    const std::wstring& i_wstrName = L"Compound")
    {
        std::auto_ptr<ISerializer> pSerializer( i_serializer.SerializeCompound( i_wstrName, typeid( t_Unique)));
        TCompoundSerializer<t_TypeList,t_Unique,t_Base> compoundSerializer( i_compound);
        compoundSerializer.Serialize( *pSerializer);
    }
};

namespace PSF
{
    template<class t_iterType>
    void SerializeRange( t_iterType i_iterBegin, t_iterType i_iterEnd,
                         ISerializer& i_serializer,
                         const std::wstring& i_wstrName,
                         const Loki::TypeInfo& i_typeInfo)
    {
        std::auto_ptr<ISerializer> pSerializer( i_serializer.SerializeRange( i_wstrName, i_typeInfo));
        for ( ; i_iterBegin != i_iterEnd; ++i_iterBegin)
        {
            Serialize( *i_iterBegin, *pSerializer);
        }
    }

    // Serializers for common containers
    template<typename t_type>
    void Serialize( const std::basic_string<t_type>& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName = L"String")
    {
        SerializeRange( i_in.begin(), i_in.end(), i_serializer, i_wstrName, Loki::TypeInfo(typeid(std::basic_string<void>)));
    }

    template<typename t_type>
    void Serialize( const std::list<t_type>& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName = L"List")
    {
        SerializeRange( i_in.begin(), i_in.end(), i_serializer, i_wstrName, Loki::TypeInfo(typeid(std::list<void>)));
    }

    template<typename t_type>
    void Serialize( const std::vector<t_type>& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName = L"Vector")
    {
        SerializeRange( i_in.begin(), i_in.end(), i_serializer, i_wstrName, Loki::TypeInfo(typeid(std::vector<void>)));
    }

    template<typename t_type>
    void Serialize( const std::set<t_type>& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName = L"Set")
    {
        SerializeRange( i_in.begin(), i_in.end(), i_serializer, i_wstrName, Loki::TypeInfo(typeid(std::set<void>)));
    }

    template<typename t_typeFirst,class t_typeSecond>
    void Serialize( const std::pair<t_typeFirst,t_typeSecond>& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName = L"Pair")
    {
        std::auto_ptr<ISerializer> pSerializer( i_serializer.SerializeCompound( i_wstrName, Loki::TypeInfo(typeid(std::pair<void,void>))));
        Serialize( i_in.first, *pSerializer, L"First");
        Serialize( i_in.second, *pSerializer, L"Second");
    }

    template<typename t_key,typename t_type>
    void Serialize( const std::map<t_key,t_type>& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName = L"Map")
    {
        SerializeRange( i_in.begin(), i_in.end(), i_serializer, i_wstrName, Loki::TypeInfo(typeid(std::map<void,void>)));
    }

    // POD Serializers
    #ifdef PSF_PARTIAL_ORDERING_OF_TEMPLATES
    // Template serializers
    template<typename t_type>
    void PSF::Serialize( t_type& i_in, ISerializer& i_serializer)
    {
        i_serializer.Serialize( i_in);
    }

    template<typename t_type>
    void PSF::Serialize( t_type& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName)
    {
        Serialize( i_in, *i_serializer.SerializeCompound( i_wstrName, Loki::TypeInfo(typeid(t_type))));
    }
    #else
    // Explicit serializers (for compilers that does not support partial ordering of templates)
    void Serialize( const char& i_in, ISerializer& i_serializer);
    void Serialize( const unsigned char& i_in, ISerializer& i_serializer);
    void Serialize( const short& i_in, ISerializer& i_serializer);
    void Serialize( const unsigned short& i_in, ISerializer& i_serializer);
    void Serialize( const int& i_in, ISerializer& i_serializer);
    void Serialize( const unsigned int& i_in, ISerializer& i_serializer);
```

```
    void Serialize( const long& i_in, ISerializer& i_serializer);
    void Serialize( const unsigned long& i_in, ISerializer& i_serializer);
    void Serialize( const float& i_in, ISerializer& i_serializer);
    void Serialize( const double& i_in, ISerializer& i_serializer);
    void Serialize( const bool& i_in, ISerializer& i_serializer);
    void Serialize( const wchar_t& i_in, ISerializer& i_serializer);

    void Serialize( const char& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName);
    void Serialize( const unsigned char& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName);
    void Serialize( const short& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName);
    void Serialize( const unsigned short& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName);
    void Serialize( const int& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName);
    void Serialize( const unsigned int& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName);
    void Serialize( const long& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName);
    void Serialize( const unsigned long& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName);
    void Serialize( const float& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName);
    void Serialize( const double& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName);
    void Serialize( const bool& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName);
    void Serialize( const wchar_t& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName);
    #endif
};

#endif
```

Listing 30: "Serialize.cpp"

```
#include "Serialize.hpp"

#ifndef PSF_PARTIAL_ORDERING_OF_TEMPLATES

// POD Serializers
void PSF::Serialize( const char& i_in, ISerializer& i_serializer)
{
    i_serializer.Serialize( i_in);
}

void PSF::Serialize( const unsigned char& i_in, ISerializer& i_serializer)
{
    i_serializer.Serialize( i_in);
}

void PSF::Serialize( const short& i_in, ISerializer& i_serializer)
{
    i_serializer.Serialize( i_in);
}

void PSF::Serialize( const unsigned short& i_in, ISerializer& i_serializer)
{
    i_serializer.Serialize( i_in);
}

void PSF::Serialize( const int& i_in, ISerializer& i_serializer)
{
    i_serializer.Serialize( i_in);
}

void PSF::Serialize( const unsigned int& i_in, ISerializer& i_serializer)
{
    i_serializer.Serialize( i_in);
}

void PSF::Serialize( const long& i_in, ISerializer& i_serializer)
{
    i_serializer.Serialize( i_in);
}

void PSF::Serialize( const unsigned long& i_in, ISerializer& i_serializer)
{
    i_serializer.Serialize( i_in);
}

void PSF::Serialize( const float& i_in, ISerializer& i_serializer)
{
    i_serializer.Serialize( i_in);
}

void PSF::Serialize( const double& i_in, ISerializer& i_serializer)
{
    i_serializer.Serialize( i_in);
}

void PSF::Serialize( const bool& i_in, ISerializer& i_serializer)
{
    i_serializer.Serialize( i_in);
}

void PSF::Serialize( const wchar_t& i_in, ISerializer& i_serializer)
```

```
{
    i_serializer.Serialize( i_in );
}


void PSF::Serialize( const char& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName)
{
    Serialize( i_in, *i_serializer.SerializeCompound( i_wstrName, Loki::TypeInfo(typeid(char))));
}

void PSF::Serialize( const unsigned char& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName)
{
    Serialize( i_in, *i_serializer.SerializeCompound( i_wstrName, Loki::TypeInfo(typeid(unsigned char))));
}

void PSF::Serialize( const short& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName)
{
    Serialize( i_in, *i_serializer.SerializeCompound( i_wstrName, Loki::TypeInfo(typeid(short))));
}

void PSF::Serialize( const unsigned short& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName)
{
    Serialize( i_in, *i_serializer.SerializeCompound( i_wstrName, Loki::TypeInfo(typeid(unsigned short))));
}

void PSF::Serialize( const int& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName)
{
    Serialize( i_in, *i_serializer.SerializeCompound( i_wstrName, Loki::TypeInfo(typeid(int))));
}

void PSF::Serialize( const unsigned int& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName)
{
    Serialize( i_in, *i_serializer.SerializeCompound( i_wstrName, Loki::TypeInfo(typeid(unsigned int))));
}

void PSF::Serialize( const long& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName)
{
    Serialize( i_in, *i_serializer.SerializeCompound( i_wstrName, Loki::TypeInfo(typeid(long))));
}

void PSF::Serialize( const unsigned long& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName)
{
    Serialize( i_in, *i_serializer.SerializeCompound( i_wstrName, Loki::TypeInfo(typeid(unsigned long))));
}

void PSF::Serialize( const float& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName)
{
    Serialize( i_in, *i_serializer.SerializeCompound( i_wstrName, Loki::TypeInfo(typeid(float))));
}

void PSF::Serialize( const double& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName)
{
    Serialize( i_in, *i_serializer.SerializeCompound( i_wstrName, Loki::TypeInfo(typeid(double))));
}

void PSF::Serialize( const bool& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName)
{
    Serialize( i_in, *i_serializer.SerializeCompound( i_wstrName, Loki::TypeInfo(typeid(bool))));
}

void PSF::Serialize( const wchar_t& i_in, ISerializer& i_serializer, const std::wstring& i_wstrName)
{
    Serialize( i_in, *i_serializer.SerializeCompound( i_wstrName, Loki::TypeInfo(typeid(wchar_t))));
}
#endif
```

A.4 DeSerializer class

Listing 31: "DeSerializer.hpp"

```
#ifndef DESERIALIZER_HPP
#define DESERIALIZER_HPP

#include <string>
#include <memory>
#include "Loki/TypeInfo.h"

namespace PSF
{
    class IDeSerializer
    {
```

```
    public:
        virtual ~IDeSerializer() = 0 {};

        virtual bool DeSerialize( char& o_out) = 0;
        virtual bool DeSerialize( unsigned char& o_out) = 0;
        virtual bool DeSerialize( short& o_out) = 0;
        virtual bool DeSerialize( unsigned short& o_out) = 0;
        virtual bool DeSerialize( int& o_out) = 0;
        virtual bool DeSerialize( unsigned int& o_out) = 0;
        virtual bool DeSerialize( long& o_out) = 0;
        virtual bool DeSerialize( unsigned long& o_out) = 0;
        virtual bool DeSerialize( float& o_out) = 0;
        virtual bool DeSerialize( double& o_out) = 0;
        // same as double
        // virtual bool DeSerialize( long double& o_out) = 0;
        virtual bool DeSerialize( bool& o_out) = 0;
        virtual bool DeSerialize( wchar_t& o_out) = 0;

        // creates a scope within the deserializer
        virtual std::auto_ptr<IDeSerializer> DeSerializeCompound( const std::wstring& i_wstrName, const Loki::TypeInfo& i_type) = 0;

        // DeSerializing a range means that all item deserialized in this scope must be of the same type
        virtual std::auto_ptr<IDeSerializer> DeSerializeRange( const std::wstring& i_wstrName, const Loki::TypeInfo& i_type) = 0;
    };
};

#endif
```

A.5 DeSerialize functions

Listing 32: "DeSerialize.hpp"

```
#ifndef DESERIALIZE_HPP
#define DESERIALIZE_HPP

#include <list>
#include <vector>
#include <map>
#include <set>

#include "DeSerializer.hpp"
#include "Assert.hpp"

#include "Compound.hpp"

#include "CompilerConfig.hpp"

namespace PSF
{
    // DeSerialize PSF compunds
    template<class t_TypeList,class t_Unique,class t_Base>
    class TCompoundDeSerializer
    {
    public:
        typedef TCompound<t_TypeList,t_Unique,t_Base> Compound;

        TCompoundDeSerializer( Compound& i_compound) :
            m_compound( i_compound)
        {
        }

        bool DeSerialize( IDeSerializer& i_deserializer)
        {
            TInner<Compound> inner( m_compound);
            return inner.DeSerialize( i_deserializer);
        }

    private:
        template<class t_InnerCompund>
        class TInner
        {
        public:
            TInner( t_InnerCompund& i_innerCompound) :
                m_innerCompound( i_innerCompound)
            {
            }

            bool DeSerialize( IDeSerializer& i_deserializer)
            {
                TInner<t_InnerCompund::Base> inner( m_innerCompound);
                bool bResult = inner.DeSerialize( i_deserializer);
                return bResult && DeSerializeInternal<0>( i_deserializer);
            }
```

```
        private:

            enum { number_of_loops = t_InnerCompund::number_of_members-1};

            template<int t_i>
            bool DeSerializeInternal( IDeSerializer& i_deserializer)
            {
                PSF_ASSERT( t_InnerCompund::ms_namelist[t_i]);
                const wchar_t* pwszName = t_InnerCompund::ms_namelist[t_i];
                bool bResult = PSF::DeSerialize( m_innerCompound.Member<t_i>(), i_deserializer, pwszName);
                return bResult && DeSerializeInternal<t_i+1>( i_deserializer);
            }

            template<>
            bool DeSerializeInternal<number_of_loops>( IDeSerializer& i_deserializer)
            {
                PSF_ASSERT( t_InnerCompund::ms_namelist[number_of_loops]);
                const wchar_t* pwszName = t_InnerCompund::ms_namelist[number_of_loops];
                return PSF::DeSerialize( m_innerCompound.Member<number_of_loops>(), i_deserializer, pwszName);
            }

            t_InnerCompund& m_innerCompound;
        };

        template<>
        class TInner<Loki::EmptyType>
        {
        public:
            TInner<Loki::EmptyType>( const t_InnerCompund& i_innerCompound)
            {
            }

            bool DeSerialize( IDeSerializer& i_deserializer) const
            {
                return true;
            }
        };

        Compound& m_compound;
    };

    template<class t_TypeList, class t_Unique, class t_Base>
    bool DeSerialize( TCompound<t_TypeList,t_Unique,t_Base>& o_compound,
                      IDeSerializer& i_deserializer,
                      const std::wstring& i_wstrName = L"Compound")
    {
        std::auto_ptr<IDeSerializer> pDeSerializer( i_deserializer.DeSerializeCompound( i_wstrName, typeid(t_Unique)));
        if ( !pDeSerializer.get()) return false;
        TCompoundDeSerializer<t_TypeList,t_Unique,t_Base> compoundDeSerializer( o_compound);
        return compoundDeSerializer.DeSerialize( *pDeSerializer);
    }
};

namespace PSF
{
    template<class t_iteratorType>
    bool DeSerializeRange( t_iteratorType o_iter,
                           IDeSerializer& i_deserializer,
                           const std::wstring& i_wstrName,
                           const Loki::TypeInfo& i_typeInfo)
    {
        std::auto_ptr<IDeSerializer> pDeSerializer( i_deserializer.DeSerializeRange( i_wstrName, i_typeInfo));
        if ( !pDeSerializer.get()) return false;

        t_iteratorType::container_type::value_type val;
        while ( DeSerialize( val, *pDeSerializer))
        {
            *o_iter = val;
            o_iter++;
        }
        return true;
    }


    // DeSerializers for common containers
    template<typename t_type>
    bool DeSerialize( std::basic_string<t_type>& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName = L"String")
    {
        std::back_insert_iterator<std::basic_string<t_type> > iter( o_out);
        return DeSerializeRange( iter, i_deserializer, i_wstrName, Loki::TypeInfo(typeid(std::basic_string<void>)));
    }

    template<typename t_type>
    bool DeSerialize( std::list<t_type>& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName = L"List")
    {
        std::back_insert_iterator<std::list<t_type> > iter( o_out);
        return DeSerializeRange( iter, i_deserializer, i_wstrName, Loki::TypeInfo(typeid(std::list<void>)));
    }
```

```
    template<typename t_type>
    bool DeSerialize( std::vector<t_type>& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName = L"Vector")
    {
        std::back_insert_iterator<std::vector<t_type> > iter( o_out);
        return DeSerializeRange( iter, i_deserializer, i_wstrName, Loki::TypeInfo(typeid(std::vector<void>)));
    }

    template<typename t_type>
    bool DeSerialize( std::set<t_type>& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName = L"Set")
    {
        std::insert_iterator<std::set<t_type> > iter( o_out, o_out.end());
        return DeSerializeRange( iter, i_deserializer, i_wstrName, Loki::TypeInfo(typeid(std::set<void>)));
    }

    template<typename T>
    T& UnConst( const T& i_in)
    {
        return const_cast<T&>( i_in);
    }

    template<typename t_typeFirst,typename t_typeSecond>
    bool DeSerialize( std::pair<t_typeFirst,t_typeSecond>& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName = L"Pair")
    {
        std::auto_ptr<IDeSerializer> pDeSerializer(
            i_deserializer.DeSerializeCompound( i_wstrName, Loki::TypeInfo(typeid(std::pair<void,void>))));
        if ( !pDeSerializer.get()) return false;

        // Should be compile-time const removal, but sadly that does not seem
        // to be possible i VC7
        return DeSerialize( UnConst(o_out.first), *pDeSerializer, L"First") &&
            DeSerialize( o_out.second, *pDeSerializer, L"Second");
    }

    template<typename t_key,typename t_type>
    bool DeSerialize( std::map<t_key,t_type>& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName = L"Map")
    {
        std::insert_iterator<std::map<t_key,t_type> > iter( o_out, o_out.end());
        return DeSerializeRange( iter, i_deserializer, i_wstrName, Loki::TypeInfo(typeid(std::map<void,void>)));
    }

    // POD DeSerializers
    #ifdef PSF_PARTIAL_ORDERING_OF_TEMPLATES
    // Template serializers
    template<typename t_type>
    bool PSF::DeSerialize( t_type& o_out, IDeSerializer& i_deserializer)
    {
        return i_deserializer.DeSerialize( o_out);
    }

    template<typename t_type>
    bool PSF::DeSerialize( t_type& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName)
    {
        std::auto_ptr<IDeSerializer> pDeSerializer( i_deserializer.DeSerializeCompound( i_wstrName, Loki::TypeInfo(typeid(t_type))));
        if ( !pDeSerializer.get()) return false;
        return DeSerialize( o_out, *pDeSerializer);
    }
    #else
    // Explicit deserializers (for compilers that does not support partial ordering of templates)
    bool DeSerialize( char& i_in, IDeSerializer& i_deserializer);
    bool DeSerialize( unsigned char& i_in, IDeSerializer& i_deserializer);
    bool DeSerialize( short& i_in, IDeSerializer& i_deserializer);
    bool DeSerialize( unsigned short& i_in, IDeSerializer& i_deserializer);
    bool DeSerialize( int& i_in, IDeSerializer& i_deserializer);
    bool DeSerialize( unsigned int& i_in, IDeSerializer& i_deserializer);
    bool DeSerialize( long& i_in, IDeSerializer& i_deserializer);
    bool DeSerialize( unsigned long& i_in, IDeSerializer& i_deserializer);
    bool DeSerialize( float& i_in, IDeSerializer& i_deserializer);
    bool DeSerialize( double& i_in, IDeSerializer& i_deserializer);
    bool DeSerialize( bool& i_in, IDeSerializer& i_deserializer);
    bool DeSerialize( wchar_t& i_in, IDeSerializer& i_deserializer);

    bool DeSerialize( char& i_in, IDeSerializer& i_deserializer, const std::wstring& i_wstrName);
    bool DeSerialize( unsigned char& i_in, IDeSerializer& i_deserializer, const std::wstring& i_wstrName);
    bool DeSerialize( short& i_in, IDeSerializer& i_deserializer, const std::wstring& i_wstrName);
    bool DeSerialize( unsigned short& i_in, IDeSerializer& i_deserializer, const std::wstring& i_wstrName);
    bool DeSerialize( int& i_in, IDeSerializer& i_deserializer, const std::wstring& i_wstrName);
    bool DeSerialize( unsigned int& i_in, IDeSerializer& i_deserializer, const std::wstring& i_wstrName);
    bool DeSerialize( long& i_in, IDeSerializer& i_deserializer, const std::wstring& i_wstrName);
    bool DeSerialize( unsigned long& i_in, IDeSerializer& i_deserializer, const std::wstring& i_wstrName);
    bool DeSerialize( float& i_in, IDeSerializer& i_deserializer, const std::wstring& i_wstrName);
    bool DeSerialize( double& i_in, IDeSerializer& i_deserializer, const std::wstring& i_wstrName);
    bool DeSerialize( bool& i_in, IDeSerializer& i_deserializer, const std::wstring& i_wstrName);
    bool DeSerialize( wchar_t& i_in, IDeSerializer& i_deserializer, const std::wstring& i_wstrName);
    #endif
}

#endif
```

Listing 33: "DeSerialize.cpp"

```
#include "DeSerialize.hpp"

#ifndef PSF_PARTIAL_ORDERING_OF_TEMPLATES

// POD DeSerializers
bool PSF::DeSerialize( char& o_out, IDeSerializer& i_deserializer)
{
    return i_deserializer.DeSerialize( o_out);
}

bool PSF::DeSerialize( unsigned char& o_out, IDeSerializer& i_deserializer)
{
    return i_deserializer.DeSerialize( o_out);
}

bool PSF::DeSerialize( short& o_out, IDeSerializer& i_deserializer)
{
    return i_deserializer.DeSerialize( o_out);
}

bool PSF::DeSerialize( unsigned short& o_out, IDeSerializer& i_deserializer)
{
    return i_deserializer.DeSerialize( o_out);
}

bool PSF::DeSerialize( int& o_out, IDeSerializer& i_deserializer)
{
    return i_deserializer.DeSerialize( o_out);
}

bool PSF::DeSerialize( unsigned int& o_out, IDeSerializer& i_deserializer)
{
    return i_deserializer.DeSerialize( o_out);
}

bool PSF::DeSerialize( long& o_out, IDeSerializer& i_deserializer)
{
    return i_deserializer.DeSerialize( o_out);
}

bool PSF::DeSerialize( unsigned long& o_out, IDeSerializer& i_deserializer)
{
    return i_deserializer.DeSerialize( o_out);
}

bool PSF::DeSerialize( float& o_out, IDeSerializer& i_deserializer)
{
    return i_deserializer.DeSerialize( o_out);
}

bool PSF::DeSerialize( double& o_out, IDeSerializer& i_deserializer)
{
    return i_deserializer.DeSerialize( o_out);
}

bool PSF::DeSerialize( bool& o_out, IDeSerializer& i_deserializer)
{
    return i_deserializer.DeSerialize( o_out);
}

bool PSF::DeSerialize( wchar_t& o_out, IDeSerializer& i_deserializer)
{
    return i_deserializer.DeSerialize( o_out);
}


bool PSF::DeSerialize( char& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName)
{
    std::auto_ptr<IDeSerializer> pDeSerializer( i_deserializer.DeSerializeCompound( i_wstrName, Loki::TypeInfo(typeid(char))));
    if ( !pDeSerializer.get()) return false;

    return DeSerialize( o_out, *pDeSerializer);
}

bool PSF::DeSerialize( unsigned char& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName)
{
    std::auto_ptr<IDeSerializer> pDeSerializer( i_deserializer.DeSerializeCompound( i_wstrName, Loki::TypeInfo(typeid(unsigned char))));
    if ( !pDeSerializer.get()) return false;

    return DeSerialize( o_out, *pDeSerializer);
}

bool PSF::DeSerialize( short& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName)
{
    std::auto_ptr<IDeSerializer> pDeSerializer( i_deserializer.DeSerializeCompound( i_wstrName, Loki::TypeInfo(typeid(short))));
    if ( !pDeSerializer.get()) return false;
```

```
    return DeSerialize( o_out, *pDeSerializer);
}

bool PSF::DeSerialize( unsigned short& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName)
{
    std::auto_ptr<IDeSerializer> pDeSerializer( i_deserializer.DeSerializeCompound( i_wstrName, Loki::TypeInfo(typeid(unsigned short))));
    if ( !pDeSerializer.get()) return false;

    return DeSerialize( o_out, *pDeSerializer);
}

bool PSF::DeSerialize( int& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName)
{
    std::auto_ptr<IDeSerializer> pDeSerializer( i_deserializer.DeSerializeCompound( i_wstrName, Loki::TypeInfo(typeid(int))));
    if ( !pDeSerializer.get()) return false;

    return DeSerialize( o_out, *pDeSerializer);
}

bool PSF::DeSerialize( unsigned int& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName)
{
    std::auto_ptr<IDeSerializer> pDeSerializer( i_deserializer.DeSerializeCompound( i_wstrName, Loki::TypeInfo(typeid(unsigned int))));
    if ( !pDeSerializer.get()) return false;

    return DeSerialize( o_out, *pDeSerializer);
}

bool PSF::DeSerialize( long& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName)
{
    std::auto_ptr<IDeSerializer> pDeSerializer( i_deserializer.DeSerializeCompound( i_wstrName, Loki::TypeInfo(typeid(long))));
    if ( !pDeSerializer.get()) return false;

    return DeSerialize( o_out, *pDeSerializer);
}

bool PSF::DeSerialize( unsigned long& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName)
{
    std::auto_ptr<IDeSerializer> pDeSerializer( i_deserializer.DeSerializeCompound( i_wstrName, Loki::TypeInfo(typeid(unsigned long))));
    if ( !pDeSerializer.get()) return false;

    return DeSerialize( o_out, *pDeSerializer);
}

bool PSF::DeSerialize( float& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName)
{
    std::auto_ptr<IDeSerializer> pDeSerializer( i_deserializer.DeSerializeCompound( i_wstrName, Loki::TypeInfo(typeid(float))));
    if ( !pDeSerializer.get()) return false;

    return DeSerialize( o_out, *pDeSerializer);
}

bool PSF::DeSerialize( double& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName)
{
    std::auto_ptr<IDeSerializer> pDeSerializer( i_deserializer.DeSerializeCompound( i_wstrName, Loki::TypeInfo(typeid(double))));
    if ( !pDeSerializer.get()) return false;

    return DeSerialize( o_out, *pDeSerializer);
}

bool PSF::DeSerialize( bool& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName)
{
    std::auto_ptr<IDeSerializer> pDeSerializer( i_deserializer.DeSerializeCompound( i_wstrName, Loki::TypeInfo(typeid(bool))));
    if ( !pDeSerializer.get()) return false;

    return DeSerialize( o_out, *pDeSerializer);
}

bool PSF::DeSerialize( wchar_t& o_out, IDeSerializer& i_deserializer, const std::wstring& i_wstrName)
{
    std::auto_ptr<IDeSerializer> pDeSerializer( i_deserializer.DeSerializeCompound( i_wstrName, Loki::TypeInfo(typeid(wchar_t))));
    if ( !pDeSerializer.get()) return false;

    return DeSerialize( o_out, *pDeSerializer);
}
#endif
```

A.6 XML (de)serializer

Listing 34: "XMLSerializer.hpp"

```
#ifndef XML_SERIALIZER_HPP
#define XML_SERIALIZER_HPP

#include "Serializer.hpp"
#include "TypeMap.hpp"

#include <xercesc/dom/DOM.hpp>

#include <sstream>

namespace PSF
{
    class CXMLSerializerBase : public ISerializer
    {
    public:
        CXMLSerializerBase( XERCES_CPP_NAMESPACE::DOMDocument* i_pdomDoc,
                            XERCES_CPP_NAMESPACE::DOMElement* i_pdomElementRoot,
                            const SerializerTypeMap& i_mapType,
                            bool i_bRange);

        virtual ~CXMLSerializerBase();
        virtual void Serialize( const int& i_in);
        virtual void Serialize( const unsigned int& i_in);
        virtual void Serialize( const bool& i_in);
        virtual void Serialize( const char& i_in);
        virtual void Serialize( const unsigned char& i_in);
        virtual void Serialize( const short& i_in);
        virtual void Serialize( const unsigned short& i_in);
        virtual void Serialize( const long& i_in);
        virtual void Serialize( const unsigned long& i_in);
        virtual void Serialize( const float& i_in);
        virtual void Serialize( const double& i_in);
        // same as double
        // virtual void Serialize( const long double& i_in);
        virtual void Serialize( const wchar_t& i_in);

        virtual std::auto_ptr<ISerializer> SerializeCompound( const std::wstring& i_wstrName,
                                                              const Loki::TypeInfo& i_type);
        virtual std::auto_ptr<ISerializer> SerializeRange( const std::wstring& i_wstrName,
                                                           const Loki::TypeInfo& i_type);

    protected:
        CXMLSerializerBase();
        XERCES_CPP_NAMESPACE::DOMDocument* m_pdomDoc;
        XERCES_CPP_NAMESPACE::DOMElement* m_pdomElementRoot;
        const SerializerTypeMap& m_mapType;
        size_t m_nItemsSerialized;
        bool m_bRange;
        std::wstringstream m_wstrstream;
    };

    class CXMLSerializerProxy
    {
    public:
        CXMLSerializerProxy( const std::wstring& i_wstrFilename,
                             const std::wstring& i_wstrRoot,
                             const SerializerTypeMap& i_mapType = SerializerTypeMap());

        virtual ~CXMLSerializerProxy();

        operator ISerializer&();

    protected:
        std::wstring m_wstrFilename;
        XERCES_CPP_NAMESPACE::DOMImplementation* m_pdomImpl;
        XERCES_CPP_NAMESPACE::DOMDocument* m_pdomDoc;
        std::auto_ptr<CXMLSerializerBase> m_pBase;
    };
};

#endif
```

Listing 35: "XMLSerializer.cpp"

```
#include "XMLSerializer.hpp"

#include <xercesc/util/PlatformUtils.hpp>
#include <xercesc/util/XMLString.hpp>
#include <xercesc/framework/LocalFileFormatTarget.hpp>
XERCES_CPP_NAMESPACE_USE
```

```cpp
#include <exception>
#include <sstream>
#include <iostream>

template<class t_type>
void SerializePOD( const t_type& i_in, std::wstringstream& i_wstrstream, wchar_t i_wcDelimiter = L' ')
{
    i_wstrstream << std::boolalpha << i_in;
    if ( i_wcDelimiter) i_wstrstream << i_wcDelimiter;
}

///////////////////////////////////////////////////////////////////////////////////////
// CXMLSerializerBase : Implmentation
///////////////////////////////////////////////////////////////////////////////////////

PSF::CXMLSerializerBase::CXMLSerializerBase( DOMDocument* i_pdomDoc,
                                             DOMElement* i_pdomElementRoot,
                                             const SerializerTypeMap& i_mapType,
                                             bool i_bRange) :
    m_pdomDoc( i_pdomDoc),
    m_pdomElementRoot( i_pdomElementRoot),
    m_mapType( i_mapType),
    m_bRange( i_bRange),
    m_nItemsSerialized( 0)
{
}

PSF::CXMLSerializerBase::~CXMLSerializerBase()
{
    XERCES_CPP_NAMESPACE::DOMText* pdomText = m_pdomDoc->createTextNode( m_wstrstream.str().c_str());
    m_pdomElementRoot->appendChild( pdomText);

    if ( m_bRange)
    {
        std::wstringstream wstrstream;
        wstrstream << m_nItemsSerialized;
        m_pdomElementRoot->setAttribute( L"size", wstrstream.str().c_str());
    }
}

void PSF::CXMLSerializerBase::Serialize( const char& i_in)
{
    SerializePOD( i_in, m_wstrstream, 0);
    m_nItemsSerialized++;
}

void PSF::CXMLSerializerBase::Serialize( const unsigned char& i_in)
{
    SerializePOD( i_in, m_wstrstream);
    m_nItemsSerialized++;
}

void PSF::CXMLSerializerBase::Serialize( const short& i_in)
{
    SerializePOD( i_in, m_wstrstream);
    m_nItemsSerialized++;
}

void PSF::CXMLSerializerBase::Serialize( const unsigned short& i_in)
{
    SerializePOD( i_in, m_wstrstream);
    m_nItemsSerialized++;
}

void PSF::CXMLSerializerBase::Serialize( const int& i_in)
{
    SerializePOD( i_in, m_wstrstream);
    m_nItemsSerialized++;
}

void PSF::CXMLSerializerBase::Serialize( const unsigned int& i_in)
{
    SerializePOD( i_in, m_wstrstream);
    m_nItemsSerialized++;
}

void PSF::CXMLSerializerBase::Serialize( const long& i_in)
{
    SerializePOD( i_in, m_wstrstream);
    m_nItemsSerialized++;
}

void PSF::CXMLSerializerBase::Serialize( const unsigned long& i_in)
{
    SerializePOD( i_in, m_wstrstream);
    m_nItemsSerialized++;
}
```

```
void PSF::CXMLSerializerBase::Serialize( const float& i_in )
{
    SerializePOD( i_in , m_wstrstream );
    m_nItemsSerialized++;
}

void PSF::CXMLSerializerBase::Serialize( const double& i_in )
{
    SerializePOD( i_in , m_wstrstream );
    m_nItemsSerialized++;
}

void PSF::CXMLSerializerBase::Serialize( const wchar_t& i_in )
{
    SerializePOD( i_in , m_wstrstream , 0);
    m_nItemsSerialized++;
}

void PSF::CXMLSerializerBase::Serialize( const bool& i_in )
{
    SerializePOD( i_in , m_wstrstream );
    m_nItemsSerialized++;
}

std::auto_ptr<PSF::ISerializer> PSF::CXMLSerializerBase::SerializeCompound( const std::wstring& i_wstrName,
                                                                             const Loki::TypeInfo& i_type)
{
    m_nItemsSerialized++;
    DOMElement* pdomElement = m_pdomDoc->createElement( i_wstrName.c_str());
    m_pdomElementRoot->appendChild( pdomElement);
    SerializerTypeMap::const_iterator iterType = m_mapType.find( i_type);
    if ( m_mapType.end() != iterType)
    {
        pdomElement->setAttribute( L"type", iterType->second.c_str());
    }

    return std::auto_ptr<ISerializer>( new CXMLSerializerBase( m_pdomDoc, pdomElement, m_mapType, false));
}

std::auto_ptr<PSF::ISerializer> PSF::CXMLSerializerBase::SerializeRange( const std::wstring& i_wstrName,
                                                                          const Loki::TypeInfo& i_type)
{
    m_nItemsSerialized++;
    DOMElement* pdomElement = m_pdomDoc->createElement( i_wstrName.c_str());
    m_pdomElementRoot->appendChild( pdomElement);
    SerializerTypeMap::const_iterator iterType = m_mapType.find( i_type);
    if ( m_mapType.end() != iterType)
    {
        pdomElement->setAttribute( L"type", iterType->second.c_str());
    }

    return std::auto_ptr<ISerializer>( new CXMLSerializerBase( m_pdomDoc, pdomElement, m_mapType, true));
}

////////////////////////////////////////////////////////////////////////////////////
// CXMLSerializerProxy : Implmentation
////////////////////////////////////////////////////////////////////////////////////

PSF::CXMLSerializerProxy::CXMLSerializerProxy( const std::wstring& i_wstrFilename,
                                               const std::wstring& i_wstrRoot,
                                               const SerializerTypeMap& i_mapType) :
    m_wstrFilename( i_wstrFilename),
    m_pdomImpl( 0),
    m_pdomDoc( 0)
{
    try
    {
        XMLPlatformUtils::Initialize();
        m_pdomImpl = (DOMImplementation*)DOMImplementationRegistry::getDOMImplementation( L"Core XML 1.0 LS");
        m_pdomDoc = m_pdomImpl->createDocument( 0, i_wstrRoot.c_str(), 0);
        XERCES_CPP_NAMESPACE::DOMElement* pdomElementRoot = m_pdomDoc->getDocumentElement();
        m_pBase = std::auto_ptr<CXMLSerializerBase>( new CXMLSerializerBase( m_pdomDoc, pdomElementRoot, i_mapType, false));
    }
    catch (const XMLException& toCatch)
    {
        char* pszMessage = XMLString::transcode( toCatch.getMessage());
        throw std::exception( pszMessage);
    }
}

PSF::CXMLSerializerProxy::~CXMLSerializerProxy()
{
    DOMWriter* pdomWriter = m_pdomImpl->createDOMWriter();

    if ( pdomWriter->canSetFeature(XMLUni::fgDOMWRTDiscardDefaultContent, true))
    {
        pdomWriter->setFeature(XMLUni::fgDOMWRTDiscardDefaultContent, true);
    }
```

```
    if ( pdomWriter->canSetFeature(XMLUni::fgDOMWRTFormatPrettyPrint, true))
    {
        pdomWriter->setFeature(XMLUni::fgDOMWRTFormatPrettyPrint, true);
    }

    {
        LocalFileFormatTarget formatTarget( m_wstrFilename.c_str());
        pdomWriter->writeNode( &formatTarget, *m_pdomDoc);
    }

    XMLPlatformUtils::Terminate();
}

PSF::CXMLSerializerProxy::operator PSF::ISerializer&()
{
    return *m_pBase;
}
```

Listing 36: "XMLDeSerializer.hpp"

```
#ifndef XML_DESERIALIZER_HPP
#define XML_DESERIALIZER_HPP

#include "DeSerializer.hpp"
#include "TypeMap.hpp"
#include "Checker.hpp"

#include <xercesc/dom/DOM.hpp>

#include <sstream>

namespace PSF
{
    class CXMLDeSerializerBase : public IDeSerializer
    {
    public:
        CXMLDeSerializerBase( XERCES_CPP_NAMESPACE::DOMElement* i_pdomElementRoot,
                              bool i_bSearch,
                              const SerializerTypeMap& i_mapType,
                              const IChecker& i_checker);

        virtual ~CXMLDeSerializerBase();

        virtual bool DeSerialize( char& o_out);
        virtual bool DeSerialize( unsigned char& o_out);
        virtual bool DeSerialize( short& o_out);
        virtual bool DeSerialize( unsigned short& o_out);
        virtual bool DeSerialize( int& o_out);
        virtual bool DeSerialize( unsigned int& o_out);
        virtual bool DeSerialize( long& o_out);
        virtual bool DeSerialize( unsigned long& o_out);
        virtual bool DeSerialize( float& o_out);
        virtual bool DeSerialize( double& o_out);
        // same as double
        // virtual bool DeSerialize( long double& o_out);
        virtual bool DeSerialize( wchar_t& o_out);
        virtual bool DeSerialize( bool& o_out);

        virtual std::auto_ptr<IDeSerializer> DeSerializeCompound( const std::wstring& i_wstrName,
                                                                  const Loki::TypeInfo& i_type);
        virtual std::auto_ptr<IDeSerializer> DeSerializeRange( const std::wstring& i_wstrName,
                                                               const Loki::TypeInfo& i_type);
    protected:
        // returns the element used for deserialization and updates the current element
        XERCES_CPP_NAMESPACE::DOMElement* UpdateCurrentElement( const std::wstring& i_wstrName);
        void CheckType( const std::wstring& i_wstrType, const Loki::TypeInfo& i_type);

        CXMLDeSerializerBase();
        XERCES_CPP_NAMESPACE::DOMElement* m_pdomElementRoot;
        XERCES_CPP_NAMESPACE::DOMElement* m_pdomElementCurrent;
        bool m_bSearch;
        const SerializerTypeMap& m_mapType;
        const IChecker& m_checker;
        size_t m_nItemsDeSerialized;
        size_t m_nRangeSize;

        std::wstringstream m_wstrstream;
    };

    class CXMLDeSerializerProxy
    {
    public:
        CXMLDeSerializerProxy( const std::wstring& i_wstrFilename,
                               const SerializerTypeMap& i_mapType,
                               const IChecker& i_checker);
```

```
        virtual ~CXMLDeSerializerProxy();

        operator IDeSerializer&();

    protected:
        XERCES_CPP_NAMESPACE::DOMImplementation* m_pdomImpl;
        XERCES_CPP_NAMESPACE::DOMBuilder* m_pdomBuilder;
        XERCES_CPP_NAMESPACE::DOMDocument* m_pdomDoc;
        std::auto_ptr<CXMLDeSerializerBase> m_pBase;
    };
};

#endif
```

Listing 37: "XMLDeSerializer.cpp"

```
#include "XMLDeSerializer.hpp"
//#include "XMLUtils.hpp"

#include <xercesc/util/PlatformUtils.hpp>
#include <xercesc/util/XMLString.hpp>
#include <xercesc/framework/LocalFileFormatTarget.hpp>
XERCES_CPP_NAMESPACE_USE

#include <exception>
#include <sstream>
#include <iostream>

template<class t_type>
bool DeSerializePOD( t_type& o_out, std::wstringstream& i_wstrstream, const PSF::IChecker& i_checker, size_t& io_nItemsDeSerialized)
{
    i_wstrstream >> std::boolalpha >> o_out;
    if ( !i_wstrstream.eof() && !i_wstrstream.good())
    {
        i_checker.SematicError();
    }
    if ( i_wstrstream.good()) io_nItemsDeSerialized++;
    return i_wstrstream.good();
}

XERCES_CPP_NAMESPACE::DOMElement* FindElement( const XERCES_CPP_NAMESPACE::DOMElement* i_pdomElementParent, const std::wstring& i_wstrName)
{
    if ( !i_pdomElementParent) return 0;
    XERCES_CPP_NAMESPACE::DOMNode* pdomNode;
    for ( pdomNode = i_pdomElementParent->getFirstChild(); pdomNode != 0; pdomNode = pdomNode->getNextSibling())
    {
        if ( XERCES_CPP_NAMESPACE::DOMNode::ELEMENT_NODE != pdomNode->getNodeType() || i_wstrName != pdomNode->getNodeName()) continue;

        return (XERCES_CPP_NAMESPACE::DOMElement*)pdomNode;
    }
    return 0;
}

XERCES_CPP_NAMESPACE::DOMElement* NextElement( const XERCES_CPP_NAMESPACE::DOMElement* i_pdomElement)
{
    if ( !i_pdomElement) return 0;
    XERCES_CPP_NAMESPACE::DOMNode* pdomNode = i_pdomElement->getNextSibling();
    while ( pdomNode && XERCES_CPP_NAMESPACE::DOMNode::ELEMENT_NODE != pdomNode->getNodeType())
    {
        pdomNode = pdomNode->getNextSibling();
    }

    return (XERCES_CPP_NAMESPACE::DOMElement*)pdomNode;
}

/////////////////////////////////////////////////////////////////////////////////////
// CXMLDeSerializerBase : Implmentation
/////////////////////////////////////////////////////////////////////////////////////

PSF::CXMLDeSerializerBase::CXMLDeSerializerBase( DOMElement* i_pdomElementRoot,
                                                 bool i_bSearch,
                                                 const SerializerTypeMap& i_mapType,
                                                 const IChecker& i_checker) :
    m_pdomElementRoot( i_pdomElementRoot),
    m_pdomElementCurrent( 0),
    m_bSearch( i_bSearch),
    m_mapType( i_mapType),
    m_checker( i_checker),
    m_nItemsDeSerialized( 0),
    m_nRangeSize( -1),
    m_wstrstream( std::ios_base::in)
{
    const wchar_t* pszSize = m_pdomElementRoot->getAttribute( L"size");
    if ( pszSize)
    {
        std::wstringstream wstrstream( pszSize);
        wstrstream >> m_nRangeSize;
```

```
    }

    DOMNode* pdomNode = m_pdomElementRoot->getFirstChild();
    while ( pdomNode && DOMNode::ELEMENT_NODE != pdomNode->getNodeType())
    {
        pdomNode = pdomNode->getNextSibling();
    }

    m_pdomElementCurrent = (DOMElement*)pdomNode;

    std::wstring wstrText;
    for ( pdomNode = m_pdomElementRoot->getFirstChild(); pdomNode != 0; pdomNode = pdomNode->getNextSibling())
    {
        if ( pdomNode->getNodeType() == XERCES_CPP_NAMESPACE::DOMNode::TEXT_NODE)
        {
            wstrText += pdomNode->getNodeValue();
        }
    }

    m_wstrstream.str( wstrText);
}


PSF::CXMLDeSerializerBase::~CXMLDeSerializerBase()
{
    if ( m_nRangeSize != -1)
    {
        if ( m_nItemsDeSerialized != m_nRangeSize)
        {
            m_checker.RangeSizeError();
            // emit range error
        }
    }
    // todo : assert member are null
}

XERCES_CPP_NAMESPACE::DOMElement*  PSF::CXMLDeSerializerBase::UpdateCurrentElement( const std::wstring& i_wstrName)
{
    while ( m_pdomElementCurrent && i_wstrName != m_pdomElementCurrent->getTagName())
    {
        m_pdomElementCurrent = NextElement( m_pdomElementCurrent);
    }

    if ( !m_pdomElementCurrent && m_bSearch)
    {
        m_pdomElementCurrent = FindElement( m_pdomElementRoot, i_wstrName);
    }

    DOMElement* pdomElement = m_pdomElementCurrent;
    m_pdomElementCurrent = NextElement( m_pdomElementCurrent);
    return pdomElement;
}

bool PSF::CXMLDeSerializerBase::DeSerialize( char& o_out)
{
    wchar_t wch;
    m_wstrstream >> std::noskipws;
    bool bResult = DeSerializePOD( wch, m_wstrstream, m_checker, m_nItemsDeSerialized);
    m_wstrstream >> std::skipws;
    if ( bResult) o_out = wch;
    return bResult;
}

bool PSF::CXMLDeSerializerBase::DeSerialize( unsigned char& o_out)
{
    short n;
    bool bResult = DeSerializePOD( n, m_wstrstream, m_checker, m_nItemsDeSerialized);
    if ( bResult) o_out = n;
    return bResult;
}

bool PSF::CXMLDeSerializerBase::DeSerialize( short& o_out)
{
    return DeSerializePOD( o_out, m_wstrstream, m_checker, m_nItemsDeSerialized);
}

bool PSF::CXMLDeSerializerBase::DeSerialize( unsigned short& o_out)
{
    return DeSerializePOD( o_out, m_wstrstream, m_checker, m_nItemsDeSerialized);
}

bool PSF::CXMLDeSerializerBase::DeSerialize( int& o_out)
{
    return DeSerializePOD( o_out, m_wstrstream, m_checker, m_nItemsDeSerialized);
}

bool PSF::CXMLDeSerializerBase::DeSerialize( unsigned int& o_out)
{
```

```
    return DeSerializePOD( o_out, m_wstrstream, m_checker, m_nItemsDeSerialized);
}

bool PSF::CXMLDeSerializerBase::DeSerialize( long& o_out)
{
    return DeSerializePOD( o_out, m_wstrstream, m_checker, m_nItemsDeSerialized);
}

bool PSF::CXMLDeSerializerBase::DeSerialize( unsigned long& o_out)
{
    return DeSerializePOD( o_out, m_wstrstream, m_checker, m_nItemsDeSerialized);
}

bool PSF::CXMLDeSerializerBase::DeSerialize( float& o_out)
{
    return DeSerializePOD( o_out, m_wstrstream, m_checker, m_nItemsDeSerialized);
}

bool PSF::CXMLDeSerializerBase::DeSerialize( double& o_out)
{
    return DeSerializePOD( o_out, m_wstrstream, m_checker, m_nItemsDeSerialized);
}

bool PSF::CXMLDeSerializerBase::DeSerialize( wchar_t& o_out)
{
    // streams skip whitespace per default, but for strings it is important
    m_wstrstream >> std::noskipws;
    bool bResult = DeSerializePOD( o_out, m_wstrstream, m_checker, m_nItemsDeSerialized);
    m_wstrstream >> std::skipws;
    return bResult;
}

bool PSF::CXMLDeSerializerBase::DeSerialize( bool& o_out)
{
    return DeSerializePOD( o_out, m_wstrstream, m_checker, m_nItemsDeSerialized);
}

std::auto_ptr<PSF::IDeSerializer> PSF::CXMLDeSerializerBase::DeSerializeCompound( const std::wstring& i_wstrName,
                                                                                  const Loki::TypeInfo& i_type)
{
    DOMElement* pdomElement = UpdateCurrentElement( i_wstrName);

    if ( !pdomElement) return std::auto_ptr<IDeSerializer>( 0);

    CheckType( pdomElement->getAttribute( L"type"), i_type);
    m_nItemsDeSerialized++;
    return std::auto_ptr<IDeSerializer>( new CXMLDeSerializerBase( pdomElement, true, m_mapType, m_checker));
}

std::auto_ptr<PSF::IDeSerializer> PSF::CXMLDeSerializerBase::DeSerializeRange( const std::wstring& i_wstrName,
                                                                               const Loki::TypeInfo& i_type)
{
    DOMElement* pdomElement = UpdateCurrentElement( i_wstrName);

    if ( !pdomElement) return std::auto_ptr<IDeSerializer>( 0);

    CheckType( pdomElement->getAttribute( L"type"), i_type);
    m_nItemsDeSerialized++;
    return std::auto_ptr<IDeSerializer>( new CXMLDeSerializerBase( pdomElement, false, m_mapType, m_checker));
}

void PSF::CXMLDeSerializerBase::CheckType( const std::wstring& i_wstrType, const Loki::TypeInfo& i_type)
{
    if ( i_wstrType.empty())
    {
        // No type info
        m_checker.TypeMissing();
        return;
    }

    SerializerTypeMap::const_iterator iterType = m_mapType.find( i_type);
    if ( m_mapType.end() == iterType)
    {
        // Unknown type
        m_checker.TypeUnknown();
        return;
    }

    if ( iterType->second != i_wstrType)
    {
        // Type mismatch
        m_checker.TypeMismatch();
    }
}

//////////////////////////////////////////////////////////////////////////////////
//
//////////////////////////////////////////////////////////////////////////////////
```

```
PSF::CXMLDeSerializerProxy::CXMLDeSerializerProxy( const std::wstring& i_wstrFilename,
                                              const SerializerTypeMap& i_mapType,
                                              const IChecker& i_checker) :
    m_pdomImpl( 0),
    m_pdomBuilder( 0),
    m_pdomDoc( 0),
    m_pBase( 0)
{
    try
    {
        XMLPlatformUtils::Initialize();
        m_pdomImpl = (DOMImplementation*)DOMImplementationRegistry::getDOMImplementation( L"Core XML 1.0 LS");
        m_pdomBuilder = m_pdomImpl->createDOMBuilder( DOMImplementationLS::MODE_SYNCHRONOUS, 0);

        m_pdomDoc = m_pdomBuilder->parseURI( i_wstrFilename.c_str());

        m_pBase = std::auto_ptr<CXMLDeSerializerBase>( new CXMLDeSerializerBase( m_pdomDoc->getDocumentElement(), true, i_mapType, i_checker));
    }
    catch (const DOMException& toCatch) {
        char* pszMessage = XMLString::transcode( toCatch.msg);
        throw std::exception( pszMessage);
    }
    catch (const XMLException& toCatch)
    {
        char* pszMessage = XMLString::transcode( toCatch.getMessage());
        throw std::exception( pszMessage);
    }
}

PSF::CXMLDeSerializerProxy::~CXMLDeSerializerProxy()
{
    m_pdomBuilder->release();
    XMLPlatformUtils::Terminate();
}

PSF::CXMLDeSerializerProxy::operator PSF::IDeSerializer&()
{
    return *m_pBase;
}
```

A.7 Binary (de)serializer

Listing 38: "BinarySerializer.hpp"

```
#ifndef PSF_BINARY_SERIALIZER_HPP
#define PSF_BINARY_SERIALIZER_HPP

#include "Serializer.hpp"

namespace PSF
{
    class CBinarySerializer : public ISerializer
    {
    public:
        CBinarySerializer( std::ostream& i_ostream );
        virtual ~CBinarySerializer();
        virtual void Serialize( const int& i_in );
        virtual void Serialize( const unsigned int& i_in );
        virtual void Serialize( const bool& i_in );
        virtual void Serialize( const char& i_in );
        virtual void Serialize( const unsigned char& i_in );
        virtual void Serialize( const short& i_in );
        virtual void Serialize( const unsigned short& i_in );
        virtual void Serialize( const long& i_in );
        virtual void Serialize( const unsigned long& i_in );
        virtual void Serialize( const float& i_in );
        virtual void Serialize( const double& i_in );
        // same as double
        // virtual void Serialize( const long double& i_in );
        virtual void Serialize( const wchar_t& i_in );

        virtual std::auto_ptr<ISerializer> SerializeCompound( const std::wstring& i_wstrName, const Loki::TypeInfo& i_type );
        virtual std::auto_ptr<ISerializer> SerializeRange( const std::wstring& i_wstrName, const Loki::TypeInfo& i_type );
    protected:
        std::ostream& m_ostream;
        size_t m_nStartPos;
    };
};

#endif
```

Listing 39: "BinarySerializer.cpp"

```
#include "BinarySerializer.hpp"
#include "BinaryUtils.hpp"

#include <iostream>


template<class t_type>
void SerializePOD( const t_type& i_in, std::ostream& i_ostream)
{
    std::streamsize nTypeSize = sizeof(t_type);
    //i_ostream.write( reinterpret_cast<const char*>(&nTypeSize), 4);
    i_ostream.write( reinterpret_cast<const char*>(&i_in), nTypeSize);
}

//////////////////////////////////////////////////////////////////////////////////
// CBinarySerializer : Implmentation
//////////////////////////////////////////////////////////////////////////////////

PSF::CBinarySerializer::CBinarySerializer( std::ostream& i_ostream) :
    m_ostream( i_ostream),
    m_nStartPos( i_ostream.tellp())
{
    unsigned int nTypeSize = 0;
    m_ostream.write( reinterpret_cast<const char*>(&nTypeSize), 4);
}

PSF::CBinarySerializer::~CBinarySerializer()
{
    size_t nEndPos = m_ostream.tellp();
    size_t nTypeSize = nEndPos - m_nStartPos - 4;
    m_ostream.seekp( m_nStartPos, std::ios_base::beg);
    m_ostream.write( reinterpret_cast<const char*>(&nTypeSize), 4);
    m_ostream.seekp( nEndPos, std::ios_base::beg);
}

void PSF::CBinarySerializer::Serialize( const int& i_in)
{
    SerializePOD( i_in, m_ostream);
}

void PSF::CBinarySerializer::Serialize( const unsigned int& i_in)
{
```

```
    SerializePOD( i_in , m_ostream );
}

void PSF::CBinarySerializer::Serialize( const bool& i_in )
{
    SerializePOD( i_in , m_ostream );
}

void PSF::CBinarySerializer::Serialize( const char& i_in )
{
    SerializePOD( i_in , m_ostream );
}

void PSF::CBinarySerializer::Serialize( const unsigned char& i_in )
{
    SerializePOD( i_in , m_ostream );
}

void PSF::CBinarySerializer::Serialize( const short& i_in )
{
    SerializePOD( i_in , m_ostream );
}

void PSF::CBinarySerializer::Serialize( const unsigned short& i_in )
{
    SerializePOD( i_in , m_ostream );
}

void PSF::CBinarySerializer::Serialize( const long& i_in )
{
    SerializePOD( i_in , m_ostream );
}

void PSF::CBinarySerializer::Serialize( const unsigned long& i_in )
{
    SerializePOD( i_in , m_ostream );
}

void PSF::CBinarySerializer::Serialize( const float& i_in )
{
    SerializePOD( i_in , m_ostream );
}

void PSF::CBinarySerializer::Serialize( const double& i_in )
{
    SerializePOD( i_in , m_ostream );
}

void PSF::CBinarySerializer::Serialize( const wchar_t& i_in )
{
    SerializePOD( i_in , m_ostream );
}

std::auto_ptr<PSF::ISerializer> PSF::CBinarySerializer::SerializeCompound( const std::wstring& i_wstrName,
                                                                           const Loki::TypeInfo& i_type )
{
    unsigned int nNameHash = HashString( i_wstrName );
    m_ostream.write( reinterpret_cast<const char*>(&nNameHash), 4);
    return std::auto_ptr<ISerializer>( new CBinarySerializer( m_ostream ));
}

std::auto_ptr<PSF::ISerializer> PSF::CBinarySerializer::SerializeRange( const std::wstring& i_wstrName,
                                                                        const Loki::TypeInfo& i_type )
{
    return SerializeCompound( i_wstrName, i_type );
}
```

Listing 40: "BinaryDeSerializer.hpp"

```
#ifndef PSF_BINARY_DESERIALIZER_HPP
#define PSF_BINARY_DESERIALIZER_HPP

#include "DeSerializer.hpp"

namespace PSF
{
    class CBinaryDeSerializer : public IDeSerializer
    {
    public:
        CBinaryDeSerializer( std::istream& i_istream , bool i_bSearch = true );
        virtual ~CBinaryDeSerializer();

        virtual bool DeSerialize( int& o_out );
        virtual bool DeSerialize( unsigned int& o_out );
        virtual bool DeSerialize( bool& o_out );
        virtual bool DeSerialize( char& o_out );
        virtual bool DeSerialize( unsigned char& o_out );
```

```
        virtual bool DeSerialize( short& o_out);
        virtual bool DeSerialize( unsigned short& o_out);
        virtual bool DeSerialize( long& o_out);
        virtual bool DeSerialize( unsigned long& o_out);
        virtual bool DeSerialize( float& o_out);
        virtual bool DeSerialize( double& o_out);
        // same as double
        // virtual bool DeSerialize( long double& o_out);
        virtual bool DeSerialize( wchar_t& o_out);

        virtual std::auto_ptr<IDeSerializer> DeSerializeCompound( const std::wstring& i_wstrName, const Loki::TypeInfo& i_type);
        virtual std::auto_ptr<IDeSerializer> DeSerializeRange( const std::wstring& i_wstrName, const Loki::TypeInfo& i_typeRange);
    protected:
        bool FindElement( const std::wstring& i_wstrName);

        std::istream& m_istream;
        size_t m_nStartPos;
        unsigned int m_nSize;
        bool m_bSearch;
    };
};
#endif
```

Listing 41: "BinaryDeSerializer.cpp"

```
#include "BinaryDeSerializer.hpp"
#include "BinaryUtils.hpp"

#include <iostream>

template<class t_type>
bool DeSerializePOD( t_type& o_out, std::istream& i_istream, std::streamsize i_nMaxPos)
{
    // Check if stream is valid
    if ( !i_istream) return false;
    // Check if the current position has exceeded the specified size for this chunk
    if ( i_istream.tellg() >= i_nMaxPos) return false;

    std::streamsize nTypeSize = sizeof( t_type);
    // i_istream.read( reinterpret_cast<char*>(&nTypeSize), 4);
    i_istream.read( reinterpret_cast<char*>(&o_out), nTypeSize);
    return i_istream.good();
}

////////////////////////////////////////////////////////////////////////////////
// CBinaryDeSerializer : Implmentation
////////////////////////////////////////////////////////////////////////////////

PSF::CBinaryDeSerializer::CBinaryDeSerializer( std::istream& i_istream, bool i_bSearch) :
    m_istream( i_istream),
    m_bSearch( i_bSearch)
{
    m_istream.read( reinterpret_cast<char*>(&m_nSize), 4);
    m_nStartPos = m_istream.tellg();
}

PSF::CBinaryDeSerializer::~CBinaryDeSerializer()
{
    // seek to end when done
    m_istream.seekg( m_nStartPos + m_nSize, std::ios_base::beg);
}

bool PSF::CBinaryDeSerializer::FindElement( const std::wstring& i_wstrName)
{
    // Check if stream is valid
    if ( !m_istream) return false;

    unsigned int nNameHash = HashString( i_wstrName);

    // Check if the current position has exceeded the specified size for this chunk
    if ( m_istream.tellg() < m_nStartPos + m_nSize)
    {
        // Check if the current position matches the requested
        unsigned int nReadHash;
        m_istream.read( reinterpret_cast<char*>(&nReadHash), 4);
        if ( nReadHash == nNameHash) return true;
    }

    // The position didn't match can we search?
    if ( !m_bSearch) return false;

    // seek to start of block when starting search
    m_istream.seekg( m_nStartPos, std::ios_base::beg);

    // Iterate through blocks
    while ( &m_istream && m_istream.tellg() < m_nStartPos + m_nSize)
    {
```

```
        unsigned int nReadHash;
        m_istream.read( reinterpret_cast<char*>(&nReadHash), 4);
        if ( nReadHash == nNameHash) return true;

        unsigned int nSkip;
        m_istream.read( reinterpret_cast<char*>(&nSkip), 4);
        m_istream.seekg( nSkip, std::ios_base::cur);
    }

    return false;
}


bool PSF::CBinaryDeSerializer::DeSerialize( int& o_out)
{
    return DeSerializePOD( o_out, m_istream, m_nStartPos + m_nSize);
}

bool PSF::CBinaryDeSerializer::DeSerialize( unsigned int& o_out)
{
    return DeSerializePOD( o_out, m_istream, m_nStartPos + m_nSize);
}

bool PSF::CBinaryDeSerializer::DeSerialize( bool& o_out)
{
    return DeSerializePOD( o_out, m_istream, m_nStartPos + m_nSize);
}

bool PSF::CBinaryDeSerializer::DeSerialize( char& o_out)
{
    return DeSerializePOD( o_out, m_istream, m_nStartPos + m_nSize);
}

bool PSF::CBinaryDeSerializer::DeSerialize( unsigned char& o_out)
{
    return DeSerializePOD( o_out, m_istream, m_nStartPos + m_nSize);
}

bool PSF::CBinaryDeSerializer::DeSerialize( short& o_out)
{
    return DeSerializePOD( o_out, m_istream, m_nStartPos + m_nSize);
}

bool PSF::CBinaryDeSerializer::DeSerialize( unsigned short& o_out)
{
    return DeSerializePOD( o_out, m_istream, m_nStartPos + m_nSize);
}

bool PSF::CBinaryDeSerializer::DeSerialize( long& o_out)
{
    return DeSerializePOD( o_out, m_istream, m_nStartPos + m_nSize);
}

bool PSF::CBinaryDeSerializer::DeSerialize( unsigned long& o_out)
{
    return DeSerializePOD( o_out, m_istream, m_nStartPos + m_nSize);
}

bool PSF::CBinaryDeSerializer::DeSerialize( float& o_out)
{
    return DeSerializePOD( o_out, m_istream, m_nStartPos + m_nSize);
}

bool PSF::CBinaryDeSerializer::DeSerialize( double& o_out)
{
    return DeSerializePOD( o_out, m_istream, m_nStartPos + m_nSize);
}

bool PSF::CBinaryDeSerializer::DeSerialize( wchar_t& o_out)
{
    return DeSerializePOD( o_out, m_istream, m_nStartPos + m_nSize);
}

std::auto_ptr<PSF::IDeSerializer> PSF::CBinaryDeSerializer::DeSerializeCompound( const std::wstring& i_wstrName,
                                                                                const Loki::TypeInfo& i_type)
{
    if ( !FindElement( i_wstrName)) return std::auto_ptr<IDeSerializer>( 0);
    return std::auto_ptr<IDeSerializer>( new CBinaryDeSerializer( m_istream, true));
}

std::auto_ptr<PSF::IDeSerializer> PSF::CBinaryDeSerializer::DeSerializeRange( const std::wstring& i_wstrName,
                                                                             const Loki::TypeInfo& i_type)
{
    if ( !FindElement( i_wstrName)) return std::auto_ptr<IDeSerializer>( 0);
    return std::auto_ptr<IDeSerializer>( new CBinaryDeSerializer( m_istream, false));
}
```

Listing 42: "BinaryUtils.hpp"

```
#ifndef PSF_BINARY_UTILS_HPP
#define PSF_BINARY_UTILS_HPP

namespace PSF
{
    unsigned int HashString( const std::wstring& i_wstr);
};

#endif
```

Listing 43: "BinaryUtils.cpp"

```
#include <string>
#include "BinaryUtils.hpp"

unsigned int PSF::HashString( const std::wstring& i_wstr)
{
    unsigned int nHash = 0;
    for ( std::wstring::const_iterator iter = i_wstr.begin(); i_wstr.end() != iter; ++iter)
    {
        nHash = (nHash << 1)^*iter;
    }
    return nHash;
}
```

A.8 Pretty Print serializer

Listing 44: "PrettyPrintSerializer.hpp"

```
#ifndef PRETTYPRINT_SERIALIZER_HPP
#define PRETTYPRINT_SERIALIZER_HPP

#include "Serializer.hpp"
#include <iostream>

namespace PSF
{
    template<class Elem>
    class TPrettyPrintSerializer : public ISerializer
    {
    public:
        TPrettyPrintSerializer( std::basic_ostream<Elem>& i_ostream, int i_nNesting = 0);
        virtual ~TPrettyPrintSerializer();
        virtual void Serialize( const int& i_in);
        virtual void Serialize( const unsigned int& i_in);
        virtual void Serialize( const bool& i_in);
        virtual void Serialize( const char& i_in);
        virtual void Serialize( const unsigned char& i_in);
        virtual void Serialize( const short& i_in);
        virtual void Serialize( const unsigned short& i_in);
        virtual void Serialize( const long& i_in);
        virtual void Serialize( const unsigned long& i_in);
        virtual void Serialize( const float& i_in);
        virtual void Serialize( const double& i_in);
        // same as double
        // virtual void Serialize( const long double& i_in);
        virtual void Serialize( const wchar_t& i_in);

        virtual std::auto_ptr<ISerializer> SerializeCompound( const std::wstring& i_wstrName, const Loki::TypeInfo& i_type);
        virtual std::auto_ptr<ISerializer> SerializeRange( const std::wstring& i_wstrName, const Loki::TypeInfo& i_type);
    protected:
        std::basic_ostream<Elem>& m_ostream;
        int m_nNesting;
    };
};

#include "PrettyPrintSerializer.inl"

#endif
```

Listing 45: "PrettyPrintSerializer.inl"

```
#include "PrettyPrintSerializer.hpp"

#include <exception>
#include <sstream>
#include <iostream>

/////////////////////////////////////////////////////////////////////////////////////
// TPrettyPrintSerializer : Implmentation
/////////////////////////////////////////////////////////////////////////////////////

template<class Elem>
PSF::TPrettyPrintSerializer<Elem>::TPrettyPrintSerializer( std::basic_ostream<Elem>& i_ostream, int i_nNesting) :
    m_ostream( i_ostream),
    m_nNesting( i_nNesting)
{
}

template<class Elem>
PSF::TPrettyPrintSerializer<Elem>::~TPrettyPrintSerializer()
{
}


template<class Elem>
void PSF::TPrettyPrintSerializer<Elem>::Serialize( const int& i_in)
{
    m_ostream << i_in << m_ostream.widen(' ');
}

template<class Elem>
void PSF::TPrettyPrintSerializer<Elem>::Serialize( const unsigned int& i_in)
{
    m_ostream << i_in << m_ostream.widen(' ');
}

template<class Elem>
void PSF::TPrettyPrintSerializer<Elem>::Serialize( const bool& i_in)
{
```

```
    m_ostream << std::boolalpha << i_in << m_ostream.widen(' ');
}

template<class Elem>
void PSF::TPrettyPrintSerializer<Elem>::Serialize( const char& i_in)
{
    m_ostream << i_in << m_ostream.widen(' ');
}

template<class Elem>
void PSF::TPrettyPrintSerializer<Elem>::Serialize( const unsigned char& i_in)
{
    m_ostream << i_in << m_ostream.widen(' ');
}

template<class Elem>
void PSF::TPrettyPrintSerializer<Elem>::Serialize( const short& i_in)
{
    m_ostream << i_in << m_ostream.widen(' ');
}

template<class Elem>
void PSF::TPrettyPrintSerializer<Elem>::Serialize( const unsigned short& i_in)
{
    m_ostream << i_in << m_ostream.widen(' ');
}

template<class Elem>
void PSF::TPrettyPrintSerializer<Elem>::Serialize( const long& i_in)
{
    m_ostream << i_in << m_ostream.widen(' ');
}

template<class Elem>
void PSF::TPrettyPrintSerializer<Elem>::Serialize( const unsigned long& i_in)
{
    m_ostream << i_in << m_ostream.widen(' ');
}

template<class Elem>
void PSF::TPrettyPrintSerializer<Elem>::Serialize( const float& i_in)
{
    m_ostream << i_in << m_ostream.widen(' ');
}

template<class Elem>
void PSF::TPrettyPrintSerializer<Elem>::Serialize( const double& i_in)
{
    m_ostream << i_in << m_ostream.widen(' ');
}

template<class Elem>
Elem Convert( const wchar_t& i_in)
{
    // should use the streams current locale to do this conversion
    return (Elem)i_in;
}

template<>
wchar_t Convert<wchar_t>( const wchar_t& i_in)
{
    return i_in;
}

template<class Elem>
void PSF::TPrettyPrintSerializer<Elem>::Serialize( const wchar_t& i_in)
{
    m_ostream << Convert<Elem>(i_in);
}

template<class Elem>
std::auto_ptr<PSF::ISerializer> PSF::TPrettyPrintSerializer<Elem>::SerializeCompound( const std::wstring& i_wstrName,
                                                                                      const Loki::TypeInfo& i_type)
{
    Elem space = m_ostream.widen(' ');
    std::basic_string<Elem> strNesting( m_nNesting*4, space);
    std::basic_string<Elem> strName;
    std::wstring::const_iterator iterName;
    std::wstringstream wstream;
    for ( iterName = i_wstrName.begin(); iterName != i_wstrName.end(); ++iterName)
    {
        char c = wstream.narrow( *iterName, '?');
        strName.push_back( m_ostream.widen( c));
    }
    m_ostream << std::endl << strNesting << strName << space << m_ostream.widen(':') << space;
    return std::auto_ptr<ISerializer>( new TPrettyPrintSerializer( m_ostream, m_nNesting+1));
}
```

```
template<class Elem>
std::auto_ptr<PSF::ISerializer> PSF::TPrettyPrintSerializer<Elem>::SerializeRange( const std::wstring& i_wstrName,
                                                                                   const Loki::TypeInfo& i_type)
{
    return SerializeCompound( i_wstrName, i_type);
}
```

A.9 Utilities

Listing 46: "assert.hpp"

```
#ifndef PSF_ASSERT_H
#define PSF_ASSERT_H

#ifdef NDEBUG
#define PSF_ASSERT( __EXP)
#else
#define PSF_ASSERT( __EXP) { if (!(__EXP)) _asm{ int 3}}
#endif

#endif
```

Listing 47: "CompilerConfig.hpp"

```
#ifndef PSF_COMPILER_CONFIG_HPP
#define PSF_COMPILER_CONFIG_HPP

#if (_MSC_VER > 1300)
#   define PSF_PARTIAL_ORDERING_OF_TEMPLATES
#endif

#endif
```

Listing 48: "Checker.hpp"

```
#ifndef CHECKER_HPP
#define CHECKER_HPP

#include <string>
#include <memory>
#include "Loki/TypeInfo.h"

namespace PSF
{
    class IChecker
    {
    public:
        virtual ~IChecker() {};

        // Type Error reporting functions

        // TypeMissing indicates that no type info was present in the data stream
        virtual void TypeMissing() const = 0;
        // TypeMismatch indicates that the type of data in the stream does not match
        // the type supplied by the program
        virtual void TypeMismatch() const = 0;
        // TypeUnknown indicates that the program failed to supply a type identifier
        virtual void TypeUnknown() const = 0;
        // SemanticError indicates that the format of data in the stream does not
        // match the expected type, fx. the stream contains the string 'abc', but
        // the program expects an integer
        virtual void SematicError() const = 0;

        // Range Error reporting functions

        // RangeSizeError indicates that an error occured while deserializing a range.
        // The expected size of the range did not match the number of items deserialized
        // from that range
        virtual void RangeSizeError() const  = 0;
    };

    class CCheckerNull : public IChecker
    {
    public:
        virtual ~CCheckerNull();

        // Type Error reporting functions
        virtual void TypeMissing() const;
        virtual void TypeMismatch() const;
        virtual void TypeUnknown() const;
        virtual void SematicError() const;

        // Range Error reporting functions
        virtual void RangeSizeError() const;
    };

    class CCheckerAssert : public IChecker
    {
    public:
        virtual ~CCheckerAssert();
```

```
        // Type Error reporting functions
        virtual void TypeMissing() const;
        virtual void TypeMismatch() const;
        virtual void TypeUnknown() const;
        virtual void SematicError() const;

        // Range Error reporting functions
        virtual void RangeSizeError() const;
    };

    class CCheckerException : public IChecker
    {
    public:
        virtual ~CCheckerException();

        // Type Error reporting functions
        virtual void TypeMissing() const;
        virtual void TypeMismatch() const;
        virtual void TypeUnknown() const;
        virtual void SematicError() const;

        // Range Error reporting functions
        virtual void RangeSizeError() const;
    };
};
#endif
```

Listing 49: "Checker.cpp"

```
#include "Loki/TypeInfo.h"
#include <map>

namespace PSF
{
    typedef std::map<Loki::TypeInfo,std::wstring> TypeMap;
}
```

Listing 50: "Checker.cpp"

```
#include "Loki/TypeInfo.h"
#include <map>

namespace PSF
{
    typedef std::map<Loki::TypeInfo,std::wstring> TypeMap;
}
```

B Test Source listings

B.1 Test Program

Listing 51: "TestBase.hpp"

```
#ifndef PSF_TESTBASE_HPP
#define PSF_TESTBASE_HPP

#include "Compound.hpp"
#include "Serializer.hpp"
#include "DeSerializer.hpp"

#include <list>
#include <vector>
#include <map>
#include <set>
#include <string>

typedef TYPELIST_3(int,bool,float) SimpleTypes;
class CSimple : public PSF::TCompound<SimpleTypes,CSimple>
{
public:
    ACCESSORS_3( Int, Bool, Float);
};

typedef TYPELIST_2(CSimple,std::list<CSimple>) ComplexTypes;
class CComplex : public PSF::TCompound<ComplexTypes,CComplex>
{
public:
    ACCESSORS_2( Simple, List);
};

typedef TYPELIST_3(int,std::wstring,float) BaseTypes;
class CBase : public PSF::TCompound<BaseTypes,CBase>
{
public:
    ACCESSORS_3( BaseMember1, BaseMember2, BaseMember3);
};

typedef TYPELIST_2(CSimple,int) SuperTypes;
class CSuper : public PSF::TCompound<SuperTypes,CSuper,CBase>
{
public:
    ACCESSORS_2( Member1, Member2);
};

class CTestBase
{
public:
    void DefaultInit();
    void DoSerialize( PSF::ISerializer& i_serializer);
    void DoDeSerialize( PSF::IDeSerializer& i_deserializer);
    void DoInverseDeSerialize( PSF::IDeSerializer& i_deserializer);

    // Various data types used for testing
    // POD like types
    std::wstring m_wstr;
    int m_nInt;
    unsigned int m_nUnsignedInt;
    float m_fFloat;
    bool m_bBool;

    // Simple class
    CSimple m_simple;

    // Complex class
    CComplex m_complex;

    // class that inherits from another compound class
    CSuper m_super;

    // standard containers
    std::list<int> m_list;
    std::vector<int> m_vector;
    std::map<int,int> m_map;
    std::set<int> m_set;
};

#endif
```

Listing 52: "TestBase.cpp"

```
#include "TestBase.hpp"
#include "Serialize.hpp"
#include "DeSerialize.hpp"

const wchar_t* CSimple::ms_namelist[] = { L"Int", L"Bool", L"Float", 0};
const wchar_t* CComplex::ms_namelist[] = { L"Simple", L"List", 0};

const wchar_t* CBase::ms_namelist[] = { L"BaseMember1", L"BaseMember2", L"BaseMember3", 0};
const wchar_t* CSuper::ms_namelist[] = { L"Member1", L"Member2", 0};

void CTestBase::DefaultInit()
{
    // POD like types
    m_wstr = L"Text";
    m_nInt = -15;
    m_nUnsignedInt = 42;
    m_fFloat = 3.14;
    m_bBool = true;

    // Simple class
    m_simple.SetInt( 84);
    m_simple.SetBool( false);
    m_simple.SetFloat( 2.3025f);

    // Complex class
    m_complex.Simple() = m_simple;
    m_complex.List().insert( m_complex.List().end(), 3, m_simple);

    // Super class
    m_super.BaseMember1() = 16;
    m_super.BaseMember2() = L"Base Text";
    m_super.BaseMember3() = 32;
    m_super.Member1() = m_simple;
    m_super.Member2() = 256;

    // standard containers
    int nInts[] = {1, 2, 3, 4, 5, 6, 7};
    m_list.insert( m_list.end(), nInts, nInts + 7);
    m_vector.insert( m_vector.end(), nInts, nInts + 7);
    m_set.insert( nInts, nInts + 7);
    m_map[1] = 1;
    m_map[2] = 2;
    m_map[3] = 3;
    m_map[4] = 4;
    m_map[5] = 5;
}

void CTestBase::DoSerialize( PSF::ISerializer& i_serializer)
{
    // Serialize POD types
    PSF::Serialize( m_wstr, i_serializer, L"string");
    PSF::Serialize( m_nInt, i_serializer, L"int");
    PSF::Serialize( m_nUnsignedInt, i_serializer, L"unsigned_int");
    PSF::Serialize( m_fFloat, i_serializer, L"float");
    PSF::Serialize( m_bBool, i_serializer, L"bool");

    // serialize simple class
    PSF::Serialize( m_simple, i_serializer, L"Simple");

    // serialize complex class
    PSF::Serialize( m_complex, i_serializer, L"Complex");

    // serialize super class
    PSF::Serialize( m_super, i_serializer, L"Super");

    // serialize list
    PSF::Serialize( m_list, i_serializer, L"number_list");

    // serialize vector
    PSF::Serialize( m_vector, i_serializer, L"number_vector");

    // serialize map
    PSF::Serialize( m_map, i_serializer, L"number_map");

    // serialize set
    PSF::Serialize( m_set, i_serializer, L"number_set");
}

void CTestBase::DoDeSerialize( PSF::IDeSerializer& i_deserializer)
{
    // Deserialize POD types
    PSF::DeSerialize( m_wstr, i_deserializer, L"string");
    PSF::DeSerialize( m_nInt, i_deserializer, L"int");
    PSF::DeSerialize( m_nUnsignedInt, i_deserializer, L"unsigned_int");
    PSF::DeSerialize( m_fFloat, i_deserializer, L"float");
    PSF::DeSerialize( m_bBool, i_deserializer, L"bool");

    // Deserialize simple class
```

```
    PSF::DeSerialize( m_simple, i_deserializer, L"Simple");

    // Deserialize complex class
    PSF::DeSerialize( m_complex, i_deserializer, L"Complex");

    // Deserialize super class
    PSF::DeSerialize( m_super, i_deserializer, L"Super");

    // Deserialize list
    PSF::DeSerialize( m_list, i_deserializer, L"number_list");

    // Deserialize vector
    PSF::DeSerialize( m_vector, i_deserializer, L"number_vector");

    // Deserialize map
    PSF::DeSerialize( m_map, i_deserializer, L"number_map");

    // Deserialize set
    PSF::DeSerialize( m_set, i_deserializer, L"number_set");
}

void CTestBase::DoInverseDeSerialize( PSF::IDeSerializer& i_deserializer)
{
    // Deserialize set
    PSF::DeSerialize( m_set, i_deserializer, L"number_set");

    // Deserialize map
    PSF::DeSerialize( m_map, i_deserializer, L"number_map");

    // Deserialize vector
    PSF::DeSerialize( m_vector, i_deserializer, L"number_vector");

    // Deserialize list
    PSF::DeSerialize( m_list, i_deserializer, L"number_list");

    // Deserialize super class
    PSF::DeSerialize( m_super, i_deserializer, L"Super");

    // Deserialize complex class
    PSF::DeSerialize( m_complex, i_deserializer, L"Complex");

    // Deserialize simple class
    PSF::DeSerialize( m_simple, i_deserializer, L"Simple");

    // Deserialize POD types
    PSF::DeSerialize( m_bBool, i_deserializer, L"bool");
    PSF::DeSerialize( m_fFloat, i_deserializer, L"float");
    PSF::DeSerialize( m_nUnsignedInt, i_deserializer, L"unsigned_int");
    PSF::DeSerialize( m_nInt, i_deserializer, L"int");
    PSF::DeSerialize( m_wstr, i_deserializer, L"string");
}
```

Listing 53: "Main.cpp"

```
// Render test.cpp : Defines the entry point for the application.
//


//#include "Render test.h"
#include <iostream>
#include <fstream>
#include <list>
#include <map>
#include "Ghost/System/StreamRedirect.h"

#include "XMLSerializer.hpp"
#include "XMLDeSerializer.hpp"
#include "BinarySerializer.hpp"
#include "BinaryDeSerializer.hpp"
#include "PrettyPrintSerializer.hpp"

#include "TestBase.hpp"
#include "Serialize.hpp"


#define WIN32_LEAN_AND_MEAN     // Exclude rarely-used stuff from Windows headers
// Windows Header Files:
#include <windows.h>



/////////////////////////////////////////////////////////////////////////////
// Initialize data and output as xml to 'out.xml'
/////////////////////////////////////////////////////////////////////////////
SerializerTypeMap CreateTypeMap()
{
    SerializerTypeMap mapType;
```

```
    mapType[typeid(int)] = L"int";
    mapType[typeid(unsigned int)] = L"unsigned_int";
    mapType[typeid(float)] = L"float";
    mapType[typeid(bool)] = L"bool";
    mapType[typeid(char)] = L"char";
    mapType[typeid(wchar_t)] = L"wide_char";
    mapType[typeid(std::basic_string<void>)] = L"string";
    mapType[typeid(std::list<void>)] = L"list";
    mapType[typeid(std::vector<void>)] = L"vector";
    mapType[typeid(std::map<void,void>)] = L"map";
    mapType[typeid(std::set<void>)] = L"set";
    mapType[typeid(std::pair<void,void>)] = L"pair";
    mapType[typeid(CSimple)] = L"simple";
    mapType[typeid(CComplex)] = L"complex";
    mapType[typeid(CSuper)] = L"super";
    return mapType;
}

SerializerTypeMap CreateIncompleteTypeMap()
{
    SerializerTypeMap mapType;
    mapType[typeid(int)] = L"int";
    mapType[typeid(unsigned int)] = L"unsigned_int";
    //mapType[typeid(float)] = L"float";
    mapType[typeid(bool)] = L"bool";
    mapType[typeid(char)] = L"char";
    mapType[typeid(wchar_t)] = L"wide_char";
    mapType[typeid(std::basic_string<void>)] = L"string";
    mapType[typeid(std::list<void>)] = L"list";
    mapType[typeid(std::vector<void>)] = L"vector";
    mapType[typeid(std::map<void,void>)] = L"map";
    mapType[typeid(std::set<void>)] = L"set";
    mapType[typeid(std::pair<void,void>)] = L"pair";
    mapType[typeid(CSimple)] = L"simple";
    mapType[typeid(CComplex)] = L"complex";
    mapType[typeid(CSuper)] = L"super";
    return mapType;
}

///////////////////////////////////////////////////////////////////////////
// Initialize data and output as xml
///////////////////////////////////////////////////////////////////////////
void Test_ToXML( std::wstring i_wstrFilenameXML)
{
    CTestBase test;
    test.DefaultInit();

    SerializerTypeMap mapType = CreateTypeMap();

    // Construct XML serializer
    PSF::CXMLSerializerProxy serializer( i_wstrFilenameXML, L"root", mapType);
    test.DoSerialize( serializer);
}

///////////////////////////////////////////////////////////////////////////
// Initialize data and output as binary
///////////////////////////////////////////////////////////////////////////
void Test_ToBinary( std::ostream& i_ostream)
{
    CTestBase test;
    test.DefaultInit();

    SerializerTypeMap mapType = CreateTypeMap();

    // Construct XML serializer
    PSF::CBinarySerializer serializer( i_ostream);
    test.DoSerialize( serializer);
}

///////////////////////////////////////////////////////////////////////////
// Deserialize data from xml file and output pretty printed to stream
///////////////////////////////////////////////////////////////////////////
void Test_XMLToPretty( std::wstring i_wstrFilenameXML, std::ostream& i_ostream)
{
    CTestBase test;

    SerializerTypeMap mapType = CreateTypeMap();

    // Construct XML Serializer
    PSF::CCheckerAssert checker;
    PSF::CXMLDeSerializerProxy deserializer( i_wstrFilenameXML, mapType, checker);

    test.DoDeSerialize( deserializer);

    // Construct pretty print serializer
    //std::wfstream fileTxt( "out.txt", std::ios::out);
    PSF::TPrettyPrintSerializer<std::ostream::char_type> serializer( i_ostream);
```

```
    test.DoSerialize( serializer);
}

////////////////////////////////////////////////////////////////////////
// Deserialize data from xml file inverted and output pretty printed to stream
////////////////////////////////////////////////////////////////////////
void Test_InverseXMLToPretty( std::wstring i_wstrFilenameXML, std::ostream& i_ostream)
{
    CTestBase test;

    SerializerTypeMap mapType = CreateTypeMap();

    // Construct XML Serializer
    PSF::CCheckerAssert checker;
    PSF::CXMLDeSerializerProxy deserializer( i_wstrFilenameXML, mapType, checker);

    test.DoInverseDeSerialize( deserializer);

    // Construct pretty print serializer
    //std::wfstream fileTxt( "out.txt", std::ios::out);
    PSF::TPrettyPrintSerializer<std::ostream::char_type> serializer( i_ostream);

    test.DoSerialize( serializer);
}

////////////////////////////////////////////////////////////////////////
// Deserialize data from binary stream and output pretty printed to stream
////////////////////////////////////////////////////////////////////////
void Test_BinaryToPretty( std::istream& i_istream, std::ostream& i_ostream)
{
    CTestBase test;

    SerializerTypeMap mapType = CreateTypeMap();

    // Construct Binary Serializer
    PSF::CBinaryDeSerializer deserializer( i_istream);

    test.DoDeSerialize( deserializer);

    // Construct pretty print serializer
    PSF::TPrettyPrintSerializer<std::ostream::char_type> serializer( i_ostream);

    test.DoSerialize( serializer);
}

////////////////////////////////////////////////////////////////////////
// Deserialize data from binary stream inverted and output pretty printed to stream
////////////////////////////////////////////////////////////////////////
void Test_InverseBinaryToPretty( std::istream& i_istream, std::ostream& i_ostream)
{
    CTestBase test;

    SerializerTypeMap mapType = CreateTypeMap();

    // Construct Binary Serializer
    PSF::CBinaryDeSerializer deserializer( i_istream);

    test.DoInverseDeSerialize( deserializer);

    // Construct pretty print serializer
    PSF::TPrettyPrintSerializer<std::ostream::char_type> serializer( i_ostream);

    test.DoSerialize( serializer);
}

////////////////////////////////////////////////////////////////////////
// Deserialize data from xml files with errors to test xml deserializers
////////////////////////////////////////////////////////////////////////
void Test_XMLError( std::ostream& i_ostream)
{
    CTestBase test;

    PSF::CCheckerException checker;
    SerializerTypeMap mapType = CreateTypeMap();
    SerializerTypeMap mapTypeIncomplete = CreateIncompleteTypeMap();

    // Testing TypeMissing
    try
    {
        PSF::CXMLDeSerializerProxy deserializer( L"MissingType.xml", mapType, checker);
        test.DoDeSerialize( deserializer);
    }
    catch ( std::runtime_error& ex)
    {
        i_ostream << "An exception occurred while deserializing MissingType.xml" << std::endl;
        i_ostream << "Expected error message : Type missing" << std::endl;
        i_ostream << "Actual error message   : " << ex.what() << std::endl;
    }
```

```
    // Testing TypeMismatch
    try
    {
        PSF::CXMLDeSerializerProxy deserializer( L"WrongType.xml", mapType, checker);
        test.DoDeSerialize( deserializer);
    }
    catch ( std::runtime_error& ex)
    {
        i_ostream << "An exception occurred while deserializing WrongType.xml" << std::endl;
        i_ostream << "Expected error message : Type mismatch" << std::endl;
        i_ostream << "Actual error message   : " << ex.what() << std::endl;
    }

    // Testing TypeUnknown
    try
    {
        PSF::CXMLDeSerializerProxy deserializer( L"out.xml", mapTypeIncomplete, checker);
        test.DoDeSerialize( deserializer);
    }
    catch ( std::runtime_error& ex)
    {
        i_ostream << "An exception occurred while deserializing out.xml" << std::endl;
        i_ostream << "Expected error message : Type unknown" << std::endl;
        i_ostream << "Actual error message   : " << ex.what() << std::endl;
    }

    // Testing SemanticError
    try
    {
        PSF::CXMLDeSerializerProxy deserializer( L"SemanticError.xml", mapType, checker);
        test.DoDeSerialize( deserializer);
    }
    catch ( std::runtime_error& ex)
    {
        i_ostream << "An exception occurred while deserializing SemanticError.xml" << std::endl;
        i_ostream << "Expected error message : Sematic error" << std::endl;
        i_ostream << "Actual error message   : " << ex.what() << std::endl;
    }

    // Testing RangeSizeError
    try
    {
        PSF::CXMLDeSerializerProxy deserializer( L"RangeError.xml", mapType, checker);
        test.DoDeSerialize( deserializer);
    }
    catch ( std::runtime_error& ex)
    {
        i_ostream << "An exception occurred while deserializing RangeError.xml" << std::endl;
        i_ostream << "Expected error message : Range size error" << std::endl;
        i_ostream << "Actual error message   : " << ex.what() << std::endl;
    }
}

void Test()
{
    // Create XML file
    std::clog << "Creating xml file" << std::endl;
    Test_ToXML( L"out.xml");

    // Create Binary file
    {
        std::clog << "Creating binary file" << std::endl;
        std::fstream fileBin( "out.bin", std::ios::out | std::ios::binary);
        Test_ToBinary( fileBin);
    }

    // Deserialize XML and binary and pretty print
    std::clog << std::endl << "Outputting deserialized xml data as pretty printed text" << std::endl;
    Test_XMLToPretty( L"out.xml", std::clog);
    std::clog << std::endl;

    std::clog << std::endl << "Outputting deserialized binary data as pretty printed text" << std::endl;
    {
        std::fstream fileBin( "out.bin", std::ios::in | std::ios::binary);
        Test_BinaryToPretty( fileBin, std::clog);
    }
    std::clog << std::endl;

    // Deserialize XML and binary inverted and pretty print
    std::clog << std::endl << "Outputting inverse deserialized xml data as pretty printed text" << std::endl;
    Test_InverseXMLToPretty( L"out.xml", std::clog);
    std::clog << std::endl;

    std::clog << std::endl << "Outputting inverse deserialized binary data as pretty printed text" << std::endl;
    {
        std::fstream fileBin( "out.bin", std::ios::in | std::ios::binary);
        Test_InverseBinaryToPretty( fileBin, std::clog);
```

```
    }
    std::clog << std::endl;

    // Testing various type errors
    Test_XMLError( std::clog );
}


int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPTSTR    lpCmdLine,
                     int       nCmdShow)
{
    std::ofstream streamError( "Error.txt");
    System::TStreamRedirect<std::ios::char_type,std::ios::traits_type> streamRedirect_cerr( std::cerr, streamError);

    std::ofstream streamLog( "Log.txt");
    System::TStreamRedirect<std::ios::char_type,std::ios::traits_type> streamRedirect_clog( std::clog, streamLog);

    #if !defined(_DEBUG)
    try
    #endif
    {
        Test();


    }
    #if !defined(_DEBUG)
    catch( std::exception& ex)
    {
        std::cerr << "Std exception : " << ex.what() << std::endl;
    }
    catch(...)
    {
        std::cerr << "Unhandled exception" << std::endl;
    }
    #endif
    streamError.close();
    streamLog.close();

    return 1;
}
```

B.2 Results

Listing 54: "out.xml"

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<root>

  <string size="4" type="string">Text</string>

  <int type="int">-15 </int>

  <unsigned_int type="unsigned_int">42 </unsigned_int>

  <float type="float">3.14 </float>

  <bool type="bool">true </bool>

  <Simple type="simple">
    <Int type="int">84 </Int>
    <Bool type="bool">false </Bool>
    <Float type="float">2.3025 </Float>
  </Simple>

  <Complex type="complex">
    <Simple type="simple">
      <Int type="int">84 </Int>
      <Bool type="bool">false </Bool>
      <Float type="float">2.3025 </Float>
    </Simple>
    <List size="3" type="list">
      <Compound type="simple">
        <Int type="int">84 </Int>
        <Bool type="bool">false </Bool>
        <Float type="float">2.3025 </Float>
      </Compound>
      <Compound type="simple">
        <Int type="int">84 </Int>
        <Bool type="bool">false </Bool>
        <Float type="float">2.3025 </Float>
      </Compound>
      <Compound type="simple">
        <Int type="int">84 </Int>
        <Bool type="bool">false </Bool>
        <Float type="float">2.3025 </Float>
      </Compound>
    </List>
  </Complex>

  <Super type="super">
    <BaseMember1 type="int">16 </BaseMember1>
    <BaseMember2 size="9" type="string">Base Text</BaseMember2>
    <BaseMember3 type="float">32 </BaseMember3>
    <Member1 type="simple">
      <Int type="int">84 </Int>
      <Bool type="bool">false </Bool>
      <Float type="float">2.3025 </Float>
    </Member1>
    <Member2 type="int">256 </Member2>
  </Super>

  <number_list size="7" type="list">1 2 3 4 5 6 7 </number_list>

  <number_vector size="7" type="vector">1 2 3 4 5 6 7 </number_vector>

  <number_map size="5" type="map">
    <Pair type="pair">
      <First type="int">1 </First>
      <Second type="int">1 </Second>
    </Pair>
    <Pair type="pair">
      <First type="int">2 </First>
      <Second type="int">2 </Second>
    </Pair>
    <Pair type="pair">
      <First type="int">3 </First>
      <Second type="int">3 </Second>
    </Pair>
    <Pair type="pair">
      <First type="int">4 </First>
      <Second type="int">4 </Second>
    </Pair>
    <Pair type="pair">
      <First type="int">5 </First>
      <Second type="int">5 </Second>
    </Pair>
  </number_map>
```

```
  <number_set size="7" type="set">1 2 3 4 5 6 7 </number_set>

</root>
```

Listing 55: "Log.txt"

```
Creating xml file
Creating binary file

Outputting deserialized xml data as pretty printed text

string : Text
int : −15
unsigned_int: 42
float : 3.14
bool : true
Simple :
    Int : 84
    Bool : false
    Float : 2.3025
Complex :
    Simple :
        Int : 84
        Bool : false
        Float : 2.3025
        List :
            Compound :
                Int : 84
                Bool : false
                Float : 2.3025
            Compound :
                Int : 84
                Bool : false
                Float : 2.3025
            Compound :
                Int : 84
                Bool : false
                Float : 2.3025
Super :
    BaseMember1 : 16
    BaseMember2 : Base Text
    BaseMember3 : 32
    Member1 :
        Int : 84
        Bool : false
        Float : 2.3025
    Member2 : 256
number_list : 1 2 3 4 5 6 7
number_vector : 1 2 3 4 5 6 7
number_map :
    Pair :
        First : 1
        Second : 1
    Pair :
        First : 2
         Second : 2
     Pair :
        First : 3
        Second : 3
     Pair :
        First : 4
        Second : 4
     Pair :
        First : 5
        Second : 5
number_set : 1 2 3 4 5 6 7

Outputting deserialized binary data as pretty printed text

string : Text
int : −15
unsigned_int : 42
float : 3.14
bool : true
Simple :
    Int : 84
    Bool : false
    Float : 2.3025
Complex :
    Simple :
        Int : 84
        Bool : false
        Float : 2.3025
    List :
        Compound :
            Int : 84
            Bool : false
```

```
            Float : 2.3025
        Compound :
            Int : 84
            Bool : false
            Float : 2.3025
        Compound :
            Int : 84
            Bool : false
            Float : 2.3025
Super :
    BaseMember1 : 16
    BaseMember2 : Base Text
    BaseMember3 : 32
    Member1 :
        Int : 84
        Bool : false
        Float : 2.3025
    Member2 : 256
number_list : 1 2 3 4 5 6 7
number_vector : 1 2 3 4 5 6 7
number_map :
    Pair :
        First : 1
        Second : 1
    Pair :
        First : 2
        Second : 2
    Pair :
        First : 3
        Second : 3
    Pair :
        First : 4
        Second : 4
    Pair :
        First : 5
        Second : 5
number_set : 1 2 3 4 5 6 7

Outputting inverse deserialized xml data as pretty printed text

string : Text
int : -15
unsigned_int : 42
float : 3.14
bool : true
Simple :
    Int : 84
    Bool : false
    Float : 2.3025
Complex :
    Simple :
        Int : 84
        Bool : false
        Float : 2.3025
    List :
        Compound :
            Int : 84
            Bool : false
            Float : 2.3025
        Compound :
            Int : 84
            Bool : false
            Float : 2.3025
        Compound :
            Int : 84
            Bool : false
            Float : 2.3025
Super :
    BaseMember1 : 16
    BaseMember2 : Base Text
    BaseMember3 : 32
    Member1 :
        Int : 84
        Bool : false
        Float : 2.3025
    Member2 : 256
number_list : 1 2 3 4 5 6 7
number_vector : 1 2 3 4 5 6 7
number_map :
    Pair :
        First : 1
        Second : 1
    Pair :
        First : 2
        Second : 2
    Pair :
        First : 3
        Second : 3
```

```
    Pair :
        First : 4
        Second : 4
    Pair :
        First : 5
        Second : 5
number_set : 1 2 3 4 5 6 7

Outputting inverse deserialized binary data as pretty printed text

string : Text
int : −15
unsigned_int : 42
float : 3.14
bool : true
Simple :
    Int : 84
    Bool : false
    Float : 2.3025
Complex :
    Simple :
        Int : 84
        Bool : false
        Float : 2.3025
    List :
        Compound :
            Int : 84
            Bool : false
            Float : 2.3025
        Compound :
            Int : 84
            Bool : false
            Float : 2.3025
        Compound :
            Int : 84
            Bool : false
            Float : 2.3025
Super :
    BaseMember1 : 16
    BaseMember2 : Base Text
    BaseMember3 : 32
    Member1 :
        Int : 84
        Bool : false
        Float : 2.3025
    Member2 : 256
number_list : 1 2 3 4 5 6 7
number_vector : 1 2 3 4 5 6 7
number_map :
    Pair :
        First : 1
        Second : 1
    Pair :
        First : 2
        Second : 2
    Pair :
        First : 3
        Second : 3
    Pair :
        First : 4
        Second : 4
    Pair :
        First : 5
        Second : 5
number_set : 1 2 3 4 5 6 7
An exception occurred while deserializing MissingType.xml
Expected error message : Type missing
Actual error message   : Type missing
An exception occurred while deserializing WrongType.xml
Expected error message : Type mismatch
Actual error message   : Type mismath
An exception occurred while deserializing out.xml
Expected error message : Type unknown
Actual error message :   Type unknown
An exception occurred while de serializing SemanticError.xml
Expected error message : Sematic error
Actual error message   : Sematic error
An exception occurred while deserializing RangeError.xml
Expected error message : Range error
Actual error message   : Range size error
```

B.3 Data

Listing 56: "MissingType.xml"

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<root>

  <string size="4">Text</string>

  <int type="int">-15 </int>

  <unsigned_int type="unsigned_int">42 </unsigned_int>

  <float type="float">3.14 </float>

  <bool type="bool">true </bool>

  <Simple type="simple">
    <Int type="int">84 </Int>
    <Bool type="bool">false </Bool>
    <Float type="float">2.3025 </Float>
  </Simple>

  <Complex type="complex">
    <Simple type="simple">
      <Int type="int">84 </Int>
      <Bool type="bool">false </Bool>
      <Float type="float">2.3025 </Float>
    </Simple>
    <List size="3" type="list">
      <Compound type="simple">
        <Int type="int">84 </Int>
        <Bool type="bool">false </Bool>
        <Float type="float">2.3025 </Float>
      </Compound>
      <Compound type="simple">
        <Int type="int">84 </Int>
        <Bool type="bool">false </Bool>
        <Float type="float">2.3025 </Float>
      </Compound>
      <Compound type="simple">
        <Int type="int">84 </Int>
        <Bool type="bool">false </Bool>
        <Float type="float">2.3025 </Float>
      </Compound>
    </List>
  </Complex>

  <Super type="super">
    <BaseMember1 type="int">16 </BaseMember1>
    <BaseMember2 size="9" type="string">Base Text</BaseMember2>
    <BaseMember3 type="float">32 </BaseMember3>
    <Member1 type="simple">
      <Int type="int">84 </Int>
      <Bool type="bool">false </Bool>
      <Float type="float">2.3025 </Float>
    </Member1>
    <Member2 type="int">256 </Member2>
  </Super>

  <number_list size="7" type="list">1 2 3 4 5 6 7 </number_list>

  <number_vector size="7" type="vector">1 2 3 4 5 6 7 </number_vector>

  <number_map size="5" type="map">
    <Pair type="pair">
      <First type="int">1 </First>
      <Second type="int">1 </Second>
    </Pair>
    <Pair type="pair">
      <First type="int">2 </First>
      <Second type="int">2 </Second>
    </Pair>
    <Pair type="pair">
      <First type="int">3 </First>
      <Second type="int">3 </Second>
    </Pair>
    <Pair type="pair">
      <First type="int">4 </First>
      <Second type="int">4 </Second>
    </Pair>
    <Pair type="pair">
      <First type="int">5 </First>
      <Second type="int">5 </Second>
    </Pair>
  </number_map>
```

```
  <number_set size="7" type="set">1 2 3 4 5 6 7 </number_set>

</root>
```

Listing 57: "WrongType.xml"

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<root>

  <string size="4" type="int">Text</string>

  <int type="int">-15 </int>

  <unsigned_int type="unsigned_int">42 </unsigned_int>

  <float type="float">3.14 </float>

  <bool type="bool">true </bool>

  <Simple type="simple">
    <Int type="int">84 </Int>
    <Bool type="bool">false </Bool>
    <Float type="float">2.3025 </Float>
  </Simple>

  <Complex type="complex">
    <Simple type="simple">
      <Int type="int">84 </Int>
      <Bool type="bool">false </Bool>
      <Float type="float">2.3025 </Float>
    </Simple>
    <List size="3" type="list">
      <Compound type="simple">
        <Int type="int">84 </Int>
        <Bool type="bool">false </Bool>
        <Float type="float">2.3025 </Float>
      </Compound>
      <Compound type="simple">
        <Int type="int">84 </Int>
        <Bool type="bool">false </Bool>
        <Float type="float">2.3025 </Float>
      </Compound>
      <Compound type="simple">
        <Int type="int">84 </Int>
        <Bool type="bool">false </Bool>
        <Float type="float">2.3025 </Float>
      </Compound>
    </List>
  </Complex>

  <Super type="super">
    <BaseMember1 type="int">16 </BaseMember1>
    <BaseMember2 size="9" type="string">Base Text</BaseMember2>
    <BaseMember3 type="float">32 </BaseMember3>
    <Member1 type="simple">
      <Int type="int">84 </Int>
      <Bool type="bool">false </Bool>
      <Float type="float">2.3025 </Float>
    </Member1>
    <Member2 type="int">256 </Member2>
  </Super>

  <number_list size="7" type="list">1 2 3 4 5 6 7 </number_list>

  <number_vector size="7" type="vector">1 2 3 4 5 6 7 </number_vector>

  <number_map size="5" type="map">
    <Pair type="pair">
      <First type="int">1 </First>
      <Second type="int">1 </Second>
    </Pair>
    <Pair type="pair">
      <First type="int">2 </First>
      <Second type="int">2 </Second>
    </Pair>
    <Pair type="pair">
      <First type="int">3 </First>
      <Second type="int">3 </Second>
    </Pair>
    <Pair type="pair">
      <First type="int">4 </First>
      <Second type="int">4 </Second>
    </Pair>
    <Pair type="pair">
      <First type="int">5 </First>
      <Second type="int">5 </Second>
    </Pair>
```

```
  </number_map>

  <number_set size="7" type="set">1 2 3 4 5 6 7 </number_set>

</root>
```

Listing 58: "SemanticError.xml"

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<root>

  <string size="4" type="string">Text</string>

  <int type="int">abc </int>

  <unsigned_int type="unsigned_int">42 </unsigned_int>

  <float type="float">3.14 </float>

  <bool type="bool">true </bool>

  <Simple type="simple">
    <Int type="int">84 </Int>
    <Bool type="bool">false </Bool>
    <Float type="float">2.3025 </Float>
  </Simple>

  <Complex type="complex">
    <Simple type="simple">
      <Int type="int">84 </Int>
      <Bool type="bool">false </Bool>
      <Float type="float">2.3025 </Float>
    </Simple>
    <List size="3" type="list">
      <Compound type="simple">
        <Int type="int">84 </Int>
        <Bool type="bool">false </Bool>
        <Float type="float">2.3025 </Float>
      </Compound>
      <Compound type="simple">
        <Int type="int">84 </Int>
        <Bool type="bool">false </Bool>
        <Float type="float">2.3025 </Float>
      </Compound>
      <Compound type="simple">
        <Int type="int">84 </Int>
        <Bool type="bool">false </Bool>
        <Float type="float">2.3025 </Float>
      </Compound>
    </List>
  </Complex>

  <Super type="super">
    <BaseMember1 type="int">16 </BaseMember1>
    <BaseMember2 size="9" type="string">Base Text</BaseMember2>
    <BaseMember3 type="float">32 </BaseMember3>
    <Member1 type="simple">
      <Int type="int">84 </Int>
      <Bool type="bool">false </Bool>
      <Float type="float">2.3025 </Float>
    </Member1>
    <Member2 type="int">256 </Member2>
  </Super>

  <number_list size="7" type="list">1 2 3 4 5 6 7 </number_list>

  <number_vector size="7" type="vector">1 2 3 4 5 6 7 </number_vector>

  <number_map size="5" type="map">
    <Pair type="pair">
      <First type="int">1 </First>
      <Second type="int">1 </Second>
    </Pair>
    <Pair type="pair">
      <First type="int">2 </First>
      <Second type="int">2 </Second>
    </Pair>
    <Pair type="pair">
      <First type="int">3 </First>
      <Second type="int">3 </Second>
    </Pair>
    <Pair type="pair">
      <First type="int">4 </First>
      <Second type="int">4 </Second>
    </Pair>
    <Pair type="pair">
      <First type="int">5 </First>
```

```
      <Second type="int">5 </Second>
    </Pair>
  </number_map>

  <number_set size="7" type="set">1 2 3 4 5 6 7 </number_set>

</root>
```

Listing 59: "RangeError.xml"

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<root>

  <string size="9" type="string">Text</string>

  <int type="int">-15 </int>

  <unsigned_int type="unsigned_int">42 </unsigned_int>

  <float type="float">3.14 </float>

  <bool type="bool">true </bool>

  <Simple type="simple">
    <Int type="int">84 </Int>
    <Bool type="bool">false </Bool>
    <Float type="float">2.3025 </Float>
  </Simple>

  <Complex type="complex">
    <Simple type="simple">
      <Int type="int">84 </Int>
      <Bool type="bool">false </Bool>
      <Float type="float">2.3025 </Float>
    </Simple>
    <List size="3" type="list">
      <Compound type="simple">
        <Int type="int">84 </Int>
        <Bool type="bool">false </Bool>
        <Float type="float">2.3025 </Float>
      </Compound>
      <Compound type="simple">
        <Int type="int">84 </Int>
        <Bool type="bool">false </Bool>
        <Float type="float">2.3025 </Float>
      </Compound>
      <Compound type="simple">
        <Int type="int">84 </Int>
        <Bool type="bool">false </Bool>
        <Float type="float">2.3025 </Float>
      </Compound>
    </List>
  </Complex>

  <Super type="super">
    <BaseMember1 type="int">16 </BaseMember1>
    <BaseMember2 size="9" type="string">Base Text</BaseMember2>
    <BaseMember3 type="float">32 </BaseMember3>
    <Member1 type="simple">
      <Int type="int">84 </Int>
      <Bool type="bool">false </Bool>
      <Float type="float">2.3025 </Float>
    </Member1>
    <Member2 type="int">256 </Member2>
  </Super>

  <number_list size="7" type="list">1 2 3 4 5 6 7 </number_list>

  <number_vector size="7" type="vector">1 2 3 4 5 6 7 </number_vector>

  <number_map size="5" type="map">
    <Pair type="pair">
      <First type="int">1 </First>
      <Second type="int">1 </Second>
    </Pair>
    <Pair type="pair">
      <First type="int">2 </First>
      <Second type="int">2 </Second>
    </Pair>
    <Pair type="pair">
      <First type="int">3 </First>
      <Second type="int">3 </Second>
    </Pair>
    <Pair type="pair">
      <First type="int">4 </First>
      <Second type="int">4 </Second>
    </Pair>
```

```
    <Pair type="pair">
      <First type="int">5 </First>
      <Second type="int">5 </Second>
    </Pair>
  </number_map>

  <number_set size="7" type="set">1 2 3 4 5 6 7 </number_set>

</root>
```