Implementing an Efficient Geometry Dispatcher in OpenTissue

Martin Parm*
Dept. of Computer Science, Copenhagen University

March 13, 2008

Abstract

In programming we some times need to choose between several implementations of the same functionality based on the run-time types of several arguments. This is called *Dynamic Multiple Dispatch*. One example of Dynamic Multiple Dispatch is the geometry dispatcher in OpenTissue, which takes a pair of geometry objects and selects a collision-detection function based on the objects type. Unfortunately `C++` does not support Dynamic Multiple Dispatch natively, and the previous implementation of the geometry dispatcher used static type enumeration, a static function lookup table and conversion through void pointers. In this project I have explored two methods for approximating dynamic multiple dispatch in C++ and I have compared their flexibility, limits and performance. Finally I have implemented the best suitable method in OpenTissue and compared it against the previous implementation.

*e-mail: parmus@diku.dk

	Single	**Multiple**
Static	Overloading	Overloading
Dynamic	Virtual Method	*Not supported*

Figure 1: Dispatching in C++

1 Introduction

In programmer, dispatching can basically be characterized in two ways: single/multiple and static/dynamic. The table in Figure 1 shows the four different types of dispatching and their equivalences in C++. The different between single single and multiple dispatching is whether the dispatching happen based a single or multiple parameters. The difference between static and dynamic dispatching is whether the dispatching can be resolved completely at compile-time or must be resolved at run-time. In static dispatching the types of the parameters are known at compile-time and the dispatching is handled by the compiler. In C++ both static single dispatching and static multiple dispatching are supported as overloaded functions. In dynamic dispatching the exact type of the parameters are not known until at run-time, so the dispatching has to be handled at run-time as well. Virtual functions in C++ are a kind of single dynamic dispatching, but there is no support for dynamic multiple dispatching at all.

1.1 Limitations and terminologi

In this project I will attempt to construct a mechanism which approximates dynamic multiple dispatch in C++. I will refer to such a mechanism as a *dispatcher*. The types, which the dispatcher can use in the dispatching, will be refered to as *dispatchable* and the actual functions, which the dispatcher chooses from, will be refered to as *dispatch functions*. I have choosen to stay within the limitation of C++, as C++ still is a defacto language in computer graphics. Many physics engines, graphics engines and physics engines, including OpenTissue, are written in C or C++. The project will be limited to dynamic multiple dispatch based on two variables. This particular form of dynamic multiple dispatch is sometimes called *Double Dispatch*. However, I will not use this term to avoid confusion with variations of the method described by [12], which is often also referred to as *Double Dispatch*. Functions capable of dynamic multiple dispatching is sometimes also referred to as *multimethods*, but I will avoid this term as well.

Finally I will use the well-known term *functor* to describe a *function object*. A functor is simply a class with an `operator()` member function, so instances of the class can be used as ordinary functions. One of the advantages of functors over ordinary functions is that functors can store instance-specific internal states as member variables.

1.2 Reader's guide

This project is rather technical and C++ specific, and during the entire project we will assume that the reader is familiar with C++, C++ templates and C++ terminologi. Reader's unfamiliar with C++ and C++ templates are encouraged to learn the basics before continuing.

In section 2 I begin by defining the requirements for a dispatcher. Section 3 outlines a few of the previous attempts to solve this problem, and explains why those solutions are inadequite for this project. Section 4 presents two different solutions to dynamic multiple dispatchin, which are evaluated and compared in section 5. Section 6 describes how to extend solutions to take an optional third variable, and section 7 briefly compares my

solutions to dispatching mechanism used in a few of the most well-known open source physics engines. Finally section 8 concludes the project and describe any future work.

During the entire project I will use UML diagrams to illustrate different aspects of the dispatchers. As a help to the reader, Appendix B contains a short reference of the subset of the UML syntax used in the project.

This project also include a CD-ROM with all the source code developed during this project. Appendix A contains an overview of it's contents.

2 Requirements and relaxations

In this section I begin by defining the requirements for a dispatcher.

Requirement **Rq.1**: The static type system in C/C++ is there for a reason; it helps programmers from making mistakes [see 17, pg.173-168]. When disabling the compiler's static type system e.g. by statically casting though void pointers, a simple typo or copy'n'paste error can result in segmentation faults or, even worse, very hard to find memory bugs. The solution should therefore be reasonable type-safe and preferably do as much of the type-checking at compile-time. Static type casts through void pointers and similar type-unsafe tricks should be avoided.

Requirement **Rq.2**: The solution should impose minimal implementation requirements on the programmer. If the programmer has to implement an huge interface to make his class dispatchable or add too much boilerplate to his dispatch functions, it will both increase the risk for introducing errors and discourage the programmer from using the solution.

Requirement **Rq.3**: The dispatchable classes should be decoupled from both the dispatcher and each other. This requirement is to prevent the need to alter any existing classes when adding new dispatchable classes or new dispatch functions. Of course the dispatch functions need to know about the dispatchable objects.

Requirement **Rq.4**: The solution should perform reasonably well both in terms of execution speed and memory overhead. Of course, the definition of reasonably well depends entirely on the task, but as this project is focused on dispatching geometry objects in OpenTissue, this will be the basis for comparison. I will return to this subject in section 5.

Requirement **Rq.1** and **Rq.2** comes experience with programming; both the author's own experience and other's [8, 17, see]. Requirement **Rq.3** and **Rq.4** in related to the geometry dispatcher in OpenTissue; the new solutions must comparable to the current dispatcher both with regards to performance and flex ibility.

In addition to these requirements I have also defines some relaxations compared to native dispatching:

Relaxation **Rl.1**: The solution is allowed to impose some common relation (e.g. a common base-class) or interface to the dispatchable classes. This is different from static multiple dispatching which allows dispatching on otherwise unrelated types e.g. the built-in type. However, it is perfectly reasonable compared with dynamic single dispatch, which also requires inheriting from a common base-class.

Relaxation **Rl.2**: The solution does not have to handle or even understand inheritance. This relaxation might seem to be in contrast to relaxation **Rl.1**, but it is done to simplify the solutions.

Relaxation **Rl.3**: The solution should be allowed to bind dispatch functions at runtime. This is both a relaxation and a feature. The relaxation comes from the fact that

the solution does not need to have all the information required for the binding/dispatching ready at compile-time. The feature comes from the fact that this allows the programmer to change binding during the executing of the program.

Relaxation **Rl.4**: The solution is allowed to fail at runtime if no suitable dispatch function is found. This is different compared with native dispatching. With static dispatching the compiler can verify the existence of a suitable dispatch function for all attempts to dispatch. Similarly with dynamic single dispatch the compiler can verify that any instantiated class has at least one actual implementation of any virtual function somewhere in the class hierarchy. However, due to relaxation **Rl.2** and **Rl.3** it will be impossible for the compiler to verify the dispatcher at compile-time, thus it must be allowed to fail at run-time.

Relaxation **Rl.5**: The solution is only required to support free functions. Dispatching to member functions and functors does not have to be supported.

Finally I will add a project-specific feature to the solutions, which I have named *Mirroring*, which simply means that the order of the parameters will be neglected when dispatching. Mirroring has nothing to do with dispatching, e.g. it does not have an equivalent in static multiple dispatching, and mirroring can not be generalized to dispatching based on more that 2 variable. It stems directly from the specific feature that some problems have, e.g. measuring the distance between two geometry objects. As this project is focused on the problem of doing collision detection between geometry objects, which has this feature, I have chosen to include it.

3 Related work

This project is not the first attempt to approximating dynamic multiple dispatch in C++. Nummerous other ideas and solutions have already been suggested, of which these are just a few:

1. Lorenzo Bettini et al. have developed a formal way for translating multiple dispatch into single dispatch and have implemented this method as a preprocessor for C++ [6, 7].
2. Dr. Carlo Pescio have described a method for doing dynamic multiple dispatch with two objects using templates and RTTI [16].

Both these methods violate requirement **Rq.3** by having the dispatch functions as a member function in one of the dispatchable classes. This means that at least one of these classes must know the declaration of the other. While it's still very easy to add new classes to the hierarchy, it is difficult to override existing dispatch functions, as this require the programmer to derive the whole class. This violates reguirement **Rq.2**.

4 Approximating dynamic multiple dispatch

In this section, I will describe two approaches to approximation dynamic multiple dispatching.

4.1 Solution 1: DTable

In our first attempt to make an approximation of dynamic multiple dispatch I will try to improve on the current dispatch implementation from OpenTissue. I will call this solution *DTable*; a contraction of *Dynamic Table*. This name will become clear later.

4.1.1 Auto generating unique class IDs

The first thing to improve is to replace the static type enumerator with a truly unique and auto generated class ID. This will both make it easier to add new classes, and it will releave the programmer from the responsibility of ensuring that the IDs are unique.

As the dispatchable classes might be compiled in separate independent units the IDs has to be generated at runtime.

One way of generating such unique class IDs is shown in Listing 1.

Listing 1: Auto generating a unique ID using an ID generator function and a static member function with a static variable.

```
const int GenerateID(void){
  static int nextID = 0;
  return nextID++;
};

class SomeClass{
public:
  static int ID(void){
    static int id = GenerateID();
    return id;
  }
};
```

The basic idea is to have the class ID stored in a static variable inside a static member function of the class. On the first invocation of this function the ID is initialized from some common ID generator, and on all subsequential invocations it will simply return this ID. The common ID generator is simply a function, which always returns a new ID. Listing 1 shows the simplest possible ID generator which simply returns an increasing integer, but the ID could be anything as long as it is unique.

`src/OpenTissue/utility/dispatchers/classid.h` on the included CD-ROM contains the complete implementation of unique class IDs used in the DTable solution. The main idea is still the same as before. These are the main points to notice:

- The ID generator has been wrapped in a private static member function, which is only accessible from the `ClassID` template, thus preventing the programmer from messing with it directly.
- The `ID()` static member function has been wrapped in a *Curiously Recurring Template Pattern* [see 18, pg. 295-298] for easy and automatic usage.
- The implementation also includes an pure abstract class, `ClassIDInterface`, with the interface for the ClassID. This class will be used later in the DTable solution.

Listing 2 illustrates how to use `ClassID` to autogenerate unique IDs for your classes.

Listing 2: Short example showing how to use `ClassID` to generate unique class IDs

```
#include <classid.h>

class Base: virtual public ClassIDInterface{};
class A: public Base, public ClassID<A>{};
class B: public Base, public ClassID<B>{};

void test(void){
  cout << "A's ID = " << A::ID()
       << ", B's ID = " << B::ID()
       << endl;
  A a();
```

```
  B b();
  Base& isA(a);
  Base& isB(b);
  cout << "isA's ID = " << isA.classID()
       << ", isB's ID = " << isB.classID()
       << endl;
};
```

The implementation also contains a new class, `Compositor`, which adds some support for inheritance. The class can be used instead of `ClassID` to add a new class ID to a class which inherits from parent that already have a class ID. Listing 3 illustrates the problem.

Listing 3: In this `Derived::ID()` becomes ambiguous because it inherites from both Base, which already have a class ID, and from `ClassID<Derived>`.

```
#include <classid.h>

class Base: public ClassID<Base>{};

class Derived: public Base, public ClassID<Derived>{};

void test(void){
  cout << Base::ID() << endl;     # OK
  cout << Derived::ID() << endl;  # Error! Ambiguous
};
```

The class `Derived` inherits an `ID()` member function from both `Base` and `ClassID` confusing the compiler. The `Compositor` class solves this ambiguousity and makes such inheritance legal. Listing 4 shows it is use.

Listing 4: The `Compositor` handles the multiply inheritance and solves the ambiguousity problem.

```
#include <classid.h>

class Base: public ClassID<Base>{};

class Derived: public Compositor<Base, Derived>{};

void test(void){
  cout << Base::ID() << endl;     # OK
  cout << Derived::ID() << endl;  # Now it is OK
};
```

4.1.2 Making the function table dynamic

The next thing to improve is to replace the static array of function pointers with some sort of dynamic table. This will allow the programmer to extend the dispatcher with new types without changing any code in the dispatcher. This is also the reason for the name *DTable*. The `boost::multi_array`[1] have been chosen in this implementation because of it is simplicity, but it could be replaced with any similar container.

4.1.3 Encapsulating type-safet issues

Next we will do something about the type-safety. With this approach, type casting can not be avoided completely as the dispatchable objects has to be type casted to their

true type at some point. The type casting can, however, be moved from the dispatch functions to the dispatcher inself, thus encapsulating the type casting and protecting the programmer from making mistakes. This can be done by wrapping the dispatch functions in a functor class template, `Functor`, which handles the type casting before calling the dispatch function. Listing 5 shows such a simple functor class template.

Listing 5: A simple functor class template, which encapsulates the type casting of the parameters to their true types.

```
// This base class provides the functors with a
// common interface.
class FunctorBase
{
public:
  virtual ReturnType operator()(Base& t1, Base& t2) const = 0;
};

// The functor class template
template <
    class T1
  , class T2
>
class Functor: public FunctorBase
{
private:
  typedef ReturnType FuncType(T1&, T2&);
  FuncType* const m_func;

public:
  // Wrap the dispatch function, func, in this functor
  Functor(FuncType* func)
    : m_func(func)
  {}

  // Type cast the parameters and call the dispatch function
  ReturnType operator()(Base& t1, Base& t2) const
  {
    return m_func(static_cast<T1&>(t2), static_cast<T2&>(t1));
  }
};
```

Next we want to make sure that the dispatch functions gets bound correctly to the dispatcher. In the geometry dispatcher from OpenTissue, the dispatch functions are bound explicitely to specific entries in the function table. However, this can be automated using a template function, which automatically creates the correct functor and inserts it in the function table. Listing 6 shows the overall structure of such a template function.

Listing 6: A simple template function for automatically wrapping a dispatch function in a functor and adding it to a function table. In the actual implementation this will be a member function of the dispatcher.

```
template <
    class T1
  , class T2
>
void bind(ReturnType (*f)(T1&, T2&))
{
  // Extract the class IDs from the two
```

```
  // dispatchable classes
  boost::multi_array_types::index idx1=T1::ID();
  boost::multi_array_types::index idx2=T2::ID();

  // Here, we should do some error checking and extending
  // the function table as needed.

  // Create a functor wrapping the dispatch function as insert
  // it function table using the class IDs as index.
  // In this example, m_functortable is assumed to be a
  // boost::multi_array<boost::shared_ptr<FunctorBase>, 2>
  m_functortable[idx1][idx2].reset(new Functor<T1,T2>(f));
  return;
}
```

While this encapsulement does not make DTable truly type-safe in a way the compiler can verify, the use of the functor class template and the `bind()` template function does protect and releave the programmer from most type-safety issues. Strictly speaking, as long as the programmer ensures that the dispatchable classes have unique class IDs, the DTable dispatcher should handle the rest automatically.

4.1.4 Adding support for mirroring

As a final detail, we extend the `DTable` template with a boolean template variable, which indicates whether mirroring is used. When mirroring is enabled the `bind()` member function is simply extended to bind the dispatch function to two entries in the function table; one with an ordinary `Functor` and one with a `Functor`, which swaps the parameters before executing the dispatch function. The optional swapping in the `Functor` is implemented simply by extending the `Functor` template with an extra boolean template variable, which indicated if the parameters should be swapped.

4.1.5 Using the solution

The complete source code for `ClassID` and `DTable` is on the included CD-ROM (see Appendix A), and Figure 2 shows a UML diagram of the solution.

To use the solution the programmer simply have to do 3 steps:

1. Create (or alter) a common base-class for the dispatchable classes, which inherits `ClassIDInterface`. The inheritance has to be `virtual` to avoid ambiguousity by inheriting `ClassIDInterface` multiple times.
2. Let all dispatchable objects inherite from the base class and from the `ClassID` template using themself as template parameter.
3. Create a dispatcher object using the DTableDispatcher template and bind the dispatch functions.

Listing 7 shows an example of this.

Listing 7: Demo program showing show to use the DTable solution

```
#include <iostream>
#include <OpenTissue/utility/dispatchers/classid.h>
#include <OpenTissue/utility/dispatchers/dtable.h>

using namespace std;

// Step 1: Create (or alter) a common base-class for the
// dispatchable classes, which inherits ClassIDInterface. The
```

```
// inheritance has to be virtual to avoid ambiguousity by inheriting
// ClassIDInterface multiple times.
class Base: virtual public OpenTissue::ClassIDInterface{};

// Step 2: Let all dispatchable objects inherite from the base class
// and from the ClassID template using themself as template
    parameter.
class A: public Base, public OpenTissue::ClassID<A> {};
class B: public Base, public OpenTissue::ClassID<B> {};


// Declare some dispatch functions
void myfuncAA(A& a1, A& a2){
  cout << "This is myfuncAA" << endl;
};
void myfuncAB(A& a1, B& b1){
  cout << "This is myfuncAB" << endl;
};
void myfuncBA(B& b1, A& a1){
  cout << "This is myfuncBA" << endl;
};
void myfuncBB(B& b1, B& b2){
  cout << "This is myfuncBB" << endl;
};

int main(int argc, char** argv){
  // Step 3: Create a dispatcher object using the DTableDispatcher
  // template and bind the dispatch functions.
  OpenTissue::DTableDispatcher<Base, false, void> dispatcher;
  dispatcher.bind(myfuncAA);
  dispatcher.bind(myfuncAB);
  dispatcher.bind(myfuncBA);
  dispatcher.bind(myfuncBB);

  // Now the dispatcher is ready for use
  A a;
  B b;
  Base& isA(a);
  Base& isB(b);
  dispatcher(isA, isB); // Prints "This is myfuncAB"

  return 0;
};
```

4.2 Solution 2: VTable

While the DTable solution protects the programmer from most type related issues, it's core is still a static type cast based on some user-defined class ID. And this type casting can still go horribly wrong, if the programmer fails to ensure that these class IDs are truly unique, so DTable is still not truly type-safe.

Another approach to the problem is inspired by the *Double Dispatch* technique [12]. Instead of doing trying to identify the true type of the dispatchable object from the outsite and doing the type casting explicitely, we could instead use the widely known *Visitor* pattern [14, see] on each dispatchable object to simulate a table-lookup. We will call this solution *VTable*; a contraction of *Virtual Table*.

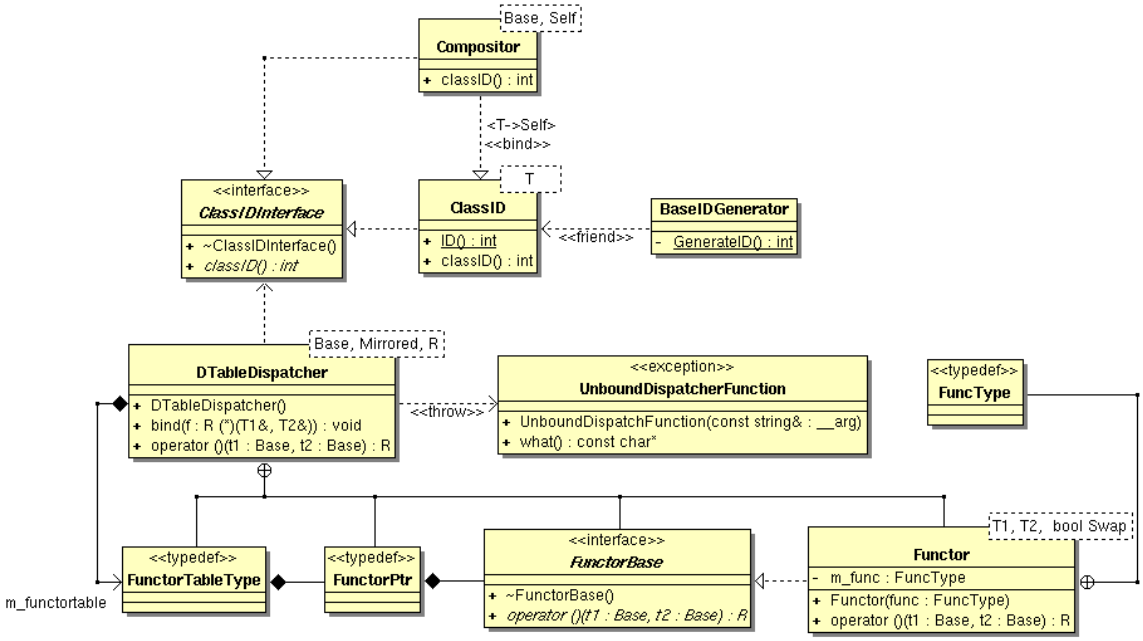

Figure 2: Complete UML diagram of the DTable solution

4.2.1 Using the Visitor pattern

The basic idea in the Visitor pattern is to ask the dispatchable object to call the visitor back passing itself as it's true type, thus selecting the correct member function based on it's type. In comparison to DTable, this is equevalent to a lookup in one-dimentional function table. Listing 8 shows a very simple implementation of this desing pattern.

Listing 8: Very simply implementation of the Visitor pattern. Notice the cyclic dependency between `Dispatchable` and `Vistor`.

```
class DispatchableInterface
{
public:
  virtual void accept(Visitor& v) = 0;
};

class Dispatchable;

class Visitor
{
public:
  void visit(DispatchableInterface& d)
  {
    // Ask the object about it's true type.
    b.visit(*this);
  }

  void visit(Dispatchable& d)
  {
    // Know we know it an instance of Dispatchable
```

```
};

class Dispatchable: public DispatchableInterface
{
public:
  virtual void accept(Visitor& v)
  {
    // Tell the visitor our true type
    v.visit(*this);
  }
};

int main(int argc, char** argv){
  DispatchableInterface* dispatchable = new Dispatchable();
  Visitor visitor;
  visitor.visit(*dispatchable);
};
```

By simply applying the Visitor pattern twice, once for each dispatchable object, we can acquire the true types of these objects in a type-safe fashion.

However, the Visitor pattern, have one major disadvantage: a cyclic dependency between the visitor and dispatchable class. In a fairly static class hierarchy this may not be a problem, but in our case it violates requirement **Rq.3**.

One possible solution to this cyclic dependency is to use the *Acyclic Visitor* pattern [13] instead. In the Acyclic Visitor pattern, the dispatchable class does not need to know the full declaration of the visitor, thus breaking the cyclic dependency. Instead the dispatchable class try to *cross cast* the visitor to an interface, which supports the dispatchable class. Listing 9 show a simple implementation of this desing pattern.

Listing 9: Very simply implementation of the Acyclic Visitor pattern. Notice the cross casting in `Dispatchable`, which relieves the class from knowing the full declaration of `AcyclicVisitor`.

```
class AcyclicVisitor;

template <
  class T1
>
class CanVisite{
public:
  virtual void visit(T1& d) = 0;
};

class DispatchableInterface
{
public:
  virtual void accept(Visitor& v) = 0;
};

class Dispatchable: public DispatchableInterface
{
public:
  virtual void accept(AcyclicVisitor& v)
  {
    // Try to cross cast the acyclic visitor to
    // to the right interface.
    CanVisit<Dispatchable>* cv;
```

```
    if ( cv = dynamic_cast<CanVisit<Dispatchable>* >(&v) ){
      // Tell the visitor our true type
      cv->visit(*this);
    }
  }
};

class AcyclicVisitor: public CanVisite<Dispatchable>
{
public:
  void visit(DispatchableInterface& d)
  {
    // Ask the object about it's true type.
    b.visit(*this);
  }

  virtual void visit(Dispatchable& d)
  {
    // Know we know it an instance of Dispatchable
};

int main(int argc, char** argv){
  DispatchableInterface* dispatchable = new Dispatchable();
  AcyclicVisitor visitor;
  visitor.visit(*dispatchable);
};
```

Using the Acyclic Visitor pattern, however, comes at a price. Crost casting is fairly expensive as it has to transverse the class hierarchy of the Acyclic Visitor at run-time, looking for the requested interface. To make matters worse, this performance penalty grows with the size of the hierarchy. In addition, the dispatchable class also have to be able to handle, if the cross casting fails.

4.2.2 Assembling the dispatcher

Ideally we would like the VTable dispatcher to be a class template taking all it's supported dispatchable classes as template parameters. The dispatcher should then inherited a `TypeDispatcher` interface for each of these templates and a single virtual template function would handle the implementation of these interfaces. Unfortunately there is two major problems, which makes this approach impossible. First of all, templates in C++ only support a fixed number of template parameters. Several techniques exist to emulate having an arbitrary number of template parameters by having a lange number of template parameters all assigned to some dummy type by default. However such techniques are often cumbersome and messy and in the end, the template will still have an upper limit set by the actual number of template parameters. The second problem with this approach is that C++ does not support implementing virtual functions as template function.

To overcome these limitations in the C++ language, we borrow some help from the C++ preprocessor and the *Boost Preprocessor Library* [2] instead. So instead of giving the supported dispatchable classes as template parameters, we assign them to a preprocessor variable and use the Boost Preprocessor Library to both add all the `TypeDispatcher` interfaces and build the implementations of these interfaces. This is not as elegant as the ideal approach would have been, but it works well in practice.

In order to support having several VTable dispatcher supporting different distatchable classes, we also define the exact type name of the VTable dispatcher as a preprocessor variable, thus enabling the preprocessor code to build several different VTable dispatchers

with different type names.

4.2.3 Two-step execution

Once the dispatcher have chosen the right functor, it also have access to the dispatchable objects by their true type and thus it could execute the functor immediately. However, it would also have to pass any return value from the dispatch function back through all the dispatchable objects. To avoid this we split the executing in two steps. Instead of executing the functor right away, the dispatcher just remembers the dispatchable objects and the functor using some internal book keeping variables, so they are accessable later on. Then the dispatcher can return from the dispatchable objects, read the functor from the internal book keeping and finally execute it. Figure 3 illustrates this process.

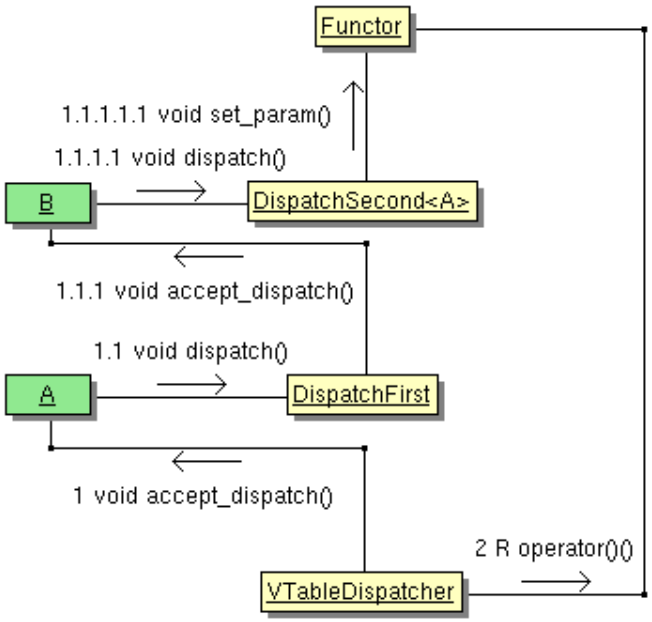

Figure 3: UML communication diagram illustrating two-step execution. Without two-step execution, any return value from `Functor` would have had to be passed back through `DispatchSecond<A>`, `B`, `DispatchFirst` and `A`.

4.2.4 Using the solution

Figure 4 and 5 shows UML diagrams of the VTable solution, when realized with two dispatchable classes, `A` and `B`. Some minor details, e.g. so me typedefs and minor helper classes, have been left out for clearity.

The steps required to use VTable are almost identical to the steps from DTable:

1. Create (or alter) a common base-class for the dispatchable classes, which inherits `DispatchableInterface`.
2. Let all dispatchable classes inherite from `Dispatchable` template using the base-class and themself as template parameters.
3. Create a dispatcher class using the `MULTIDISPATCHER_PARAMS` preprocessor definition. This definition must know of all the dispatchable classes.

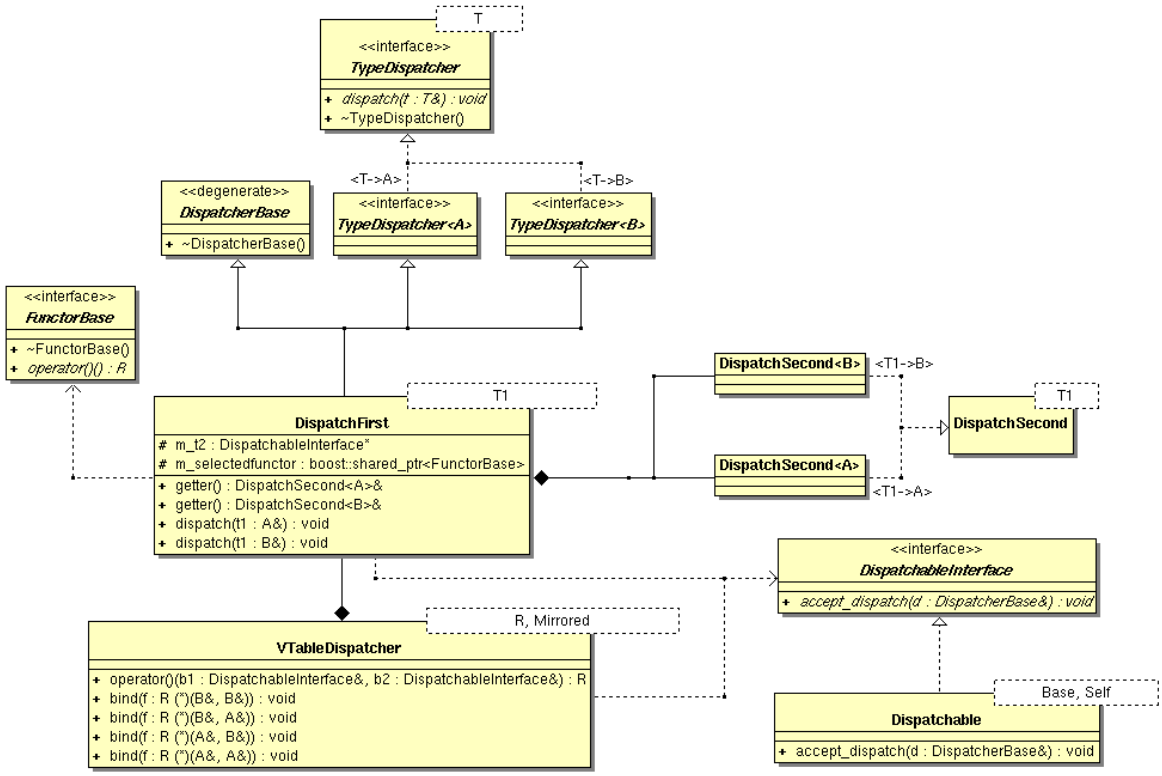

Figure 4: UML diagram of the first part of the dispatcher

4. Create a dispatcher objects using the dispatcher template just created and bind the dispatch functions.

Listing 10 shows an example of this.

Listing 10: This demo program shows how to use the VTable solution.

```
#include <iostream>
#include <OpenTissue/utility/dispatchers/vtable_interface.h>

using namespace std;

// Step 1: Create (or alter) a common base-class for the
// dispatchable classes, which inherits DispatchableInterface.
class Base: public OpenTissue::DispatchableInterface{};

// Step 2: Let all dispatchable classes inherite from
// Dispatchable template using the base-class and themself
// as template parameters.
class A: public OpenTissue::Dispatchable<Base, A>{};
class B: public OpenTissue::Dispatchable<Base, B>{};

// Declare some dispatch functions
void myfuncAA(A& a1, A& a2){
  cout << "This is myfuncAA" << endl;
};
void myfuncAB(A& a1, B& b1){
  cout << "This is myfuncAB" << endl;
```

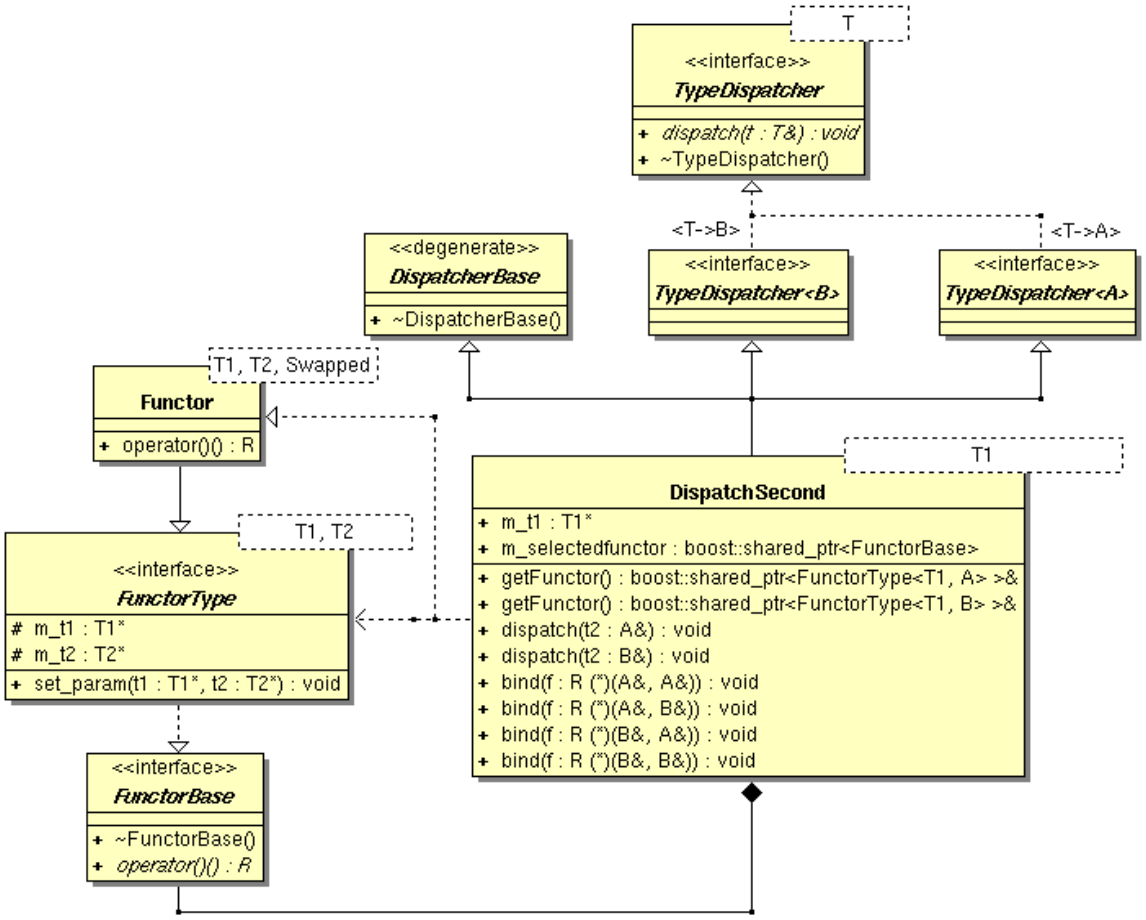

Figure 5: UML diagram of the second part of the dispatcher

```
};
void myfuncBA(B& b1, A& a1){
  cout << "This is myfuncBA" << endl;
};
void myfuncBB(B& b1, B& b2){
  cout << "This is myfuncBB" << endl;
};

// Step 3: Create a dispatcher class using the
// MULTIDISPATCHER_PARAMS preprocessor definition. This
// definition must know of all the dispatchable classes.
#define MULTIDISPATCHER_PARAMS (VTableDispatcher)((A)(B))
#include <OpenTissue/utility/dispatchers/vtable.h>

int main(int argc, char** argv){
  // Step 4: Create a dispatcher objects using the
  // dispatcher template just created and bind the
  // dispatch functions.
  OpenTissue::VTableDispatcher<void, false> dispatcher;
  dispatcher.bind(myfuncAA);
  dispatcher.bind(myfuncAB);
```

```
  dispatcher.bind(myfuncBA);
  dispatcher.bind(myfuncBB);

  // Now the dispatcher is ready for use
  A a;
  B b;
  Base& isA(a);
  Base& isB(b);
  dispatcher(isA, isB); // Prints "This is myfuncAB"

  return 0;
};
```

5 Comparison

In this section I will compare the two solutions with regards to performance, memory overhead, compile time overhead etc. All benchmark programs were compiled with the GNU gcc v4.1.2 compiler and executed on an AMD Athlon 64 3500+ processor in a 32 bit Linux environment.

5.1 Performance

First we compare the run-time cost of dispatching using any of the solutions. This is done using a simple benchmarking program, which meassures the execution time of using each solution. The program also meassures the run-time cost of Unsafe Cast for comparison.

In order to smooth out performance parameters such as cache hits, memory-lookups and etc. the program performs the dispatching on a large amount of objects with random type and then calculates the mean run-time cost. The program also tries to estimate and disregard the run-time cost of it is own loops to make the results more accuare

The graph in Figure 6 shows a comparison of run-time costs with regards to the number of dispatchable classes the dispatcher knows.

As we can see in the graph, DTable is a bit slower than Unsafe Cast, but is otherwise constant with regards to the number of dispatchable classes. This is to be expected as the heart of DTable is still just a lookup in a 2D table of functors. The extra run-time cost compared to Unsafe Cast comes from the extra steps DTable takes to ensure type-safety.

While the run-time cost of DTable is constant, the run-time cost of using VTable increases linearly with the number of dispatchable classes. This increase is due to the run-time cost of walking the class hierarchy of the dispatcher when doing cross-casting. While clearly the slowest of the solutions, it is still worth noticing that the dispatching still happens within a few microseconds.

5.2 Compile time overhead

The graph in Figure 7 shows the compile time of a minimal program, using each of the two solutions, with regards to the number of dispatchable classes the dispatcher knows.

As we can see the compile time increases almost exponentially with the number of dispatchable classes, though the VTable solution is clearly the worst. This exponential increase is due to the exponential increase in internal classes, which both solutions have:

- With n dispatchable classes, DTable have up to $n*n$ `Functor<T1,T2>` classes, one for each combination of two dispatchable classes, and n `ClassID<T>` classes. The actual number of classes depends on the number of dispatch functions actually bound to the dispatcher.

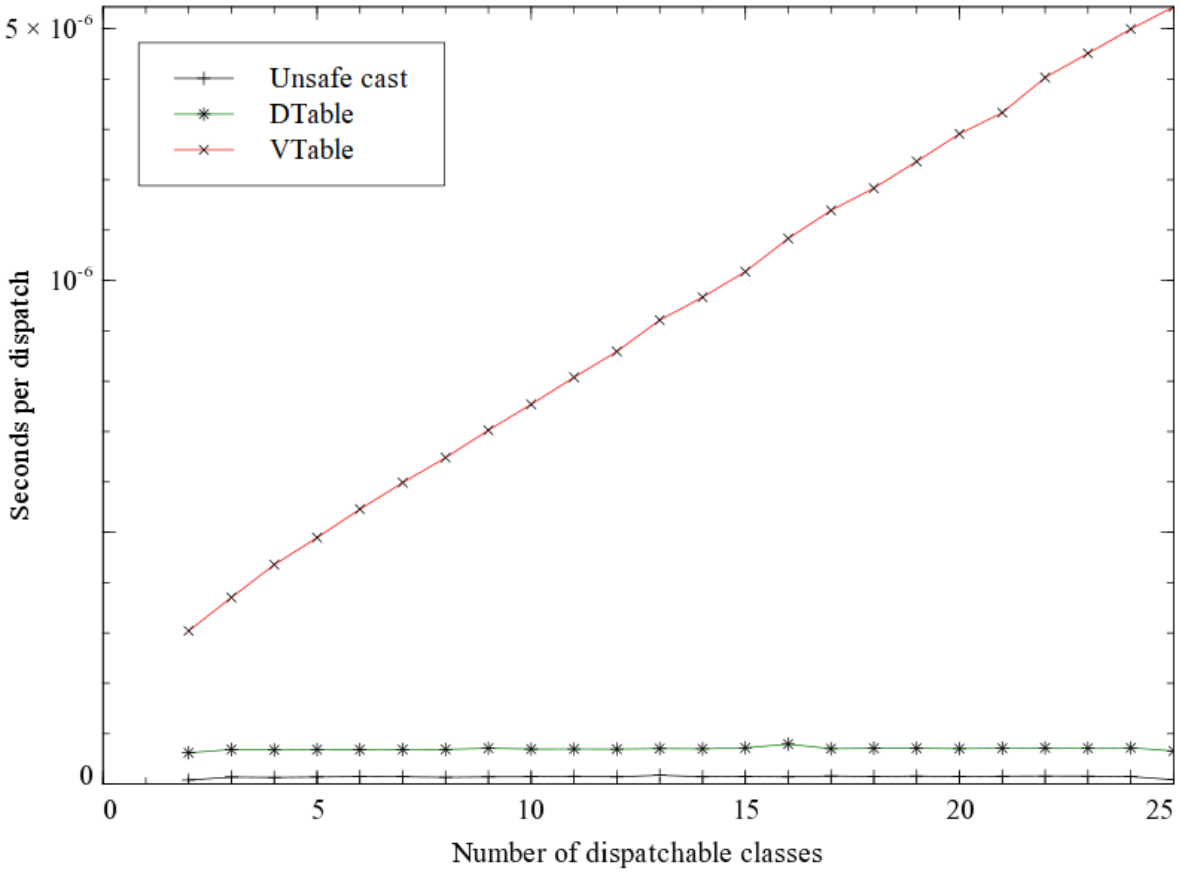

Figure 6: Run-time cost of dispatching with regards to the number of dispatchable classes the dispatcher knows.

- With n dispatchable classes, VTable have up to $n * n$ `Functor<T1,T2>` classes, just like DTable. However, it also have $n * n$ `FunctorType<T1,T2>` classes, n `DispatchSecond<T>` classes, n `TypeDispatcher<T>` classes and n `Dispatchable<T>` classes, which are independant of the number of bound dispatch functions.

5.3 Code overhead

The next thing to consider is how much extra code does the programmers have to write in order to use the solution? This is measurement is directly tied to requirement **Rq.2**.

Common base class: Both solutions require the programmer to add a common base class which the dispatchable classes must inherit. In the DTable solution the base class must inherite `ClassIDInterface` or provide a similar interface. In the VTable solution the base class must inherited `CastableInterface`. If the class hierarchy already have a common base class then this class must be altered to meet these requirements. In both solutions the base class itself may possibly be empty.

Both `ClassID_base` and `CastableInterface` can be used directly as a base class, but this is strongly discouraged as it hides the intent of the code. [8, pg. 295] describes this a *Cosmic Hierarchy* and argues strongly against it refering to the original C++ design decisions:

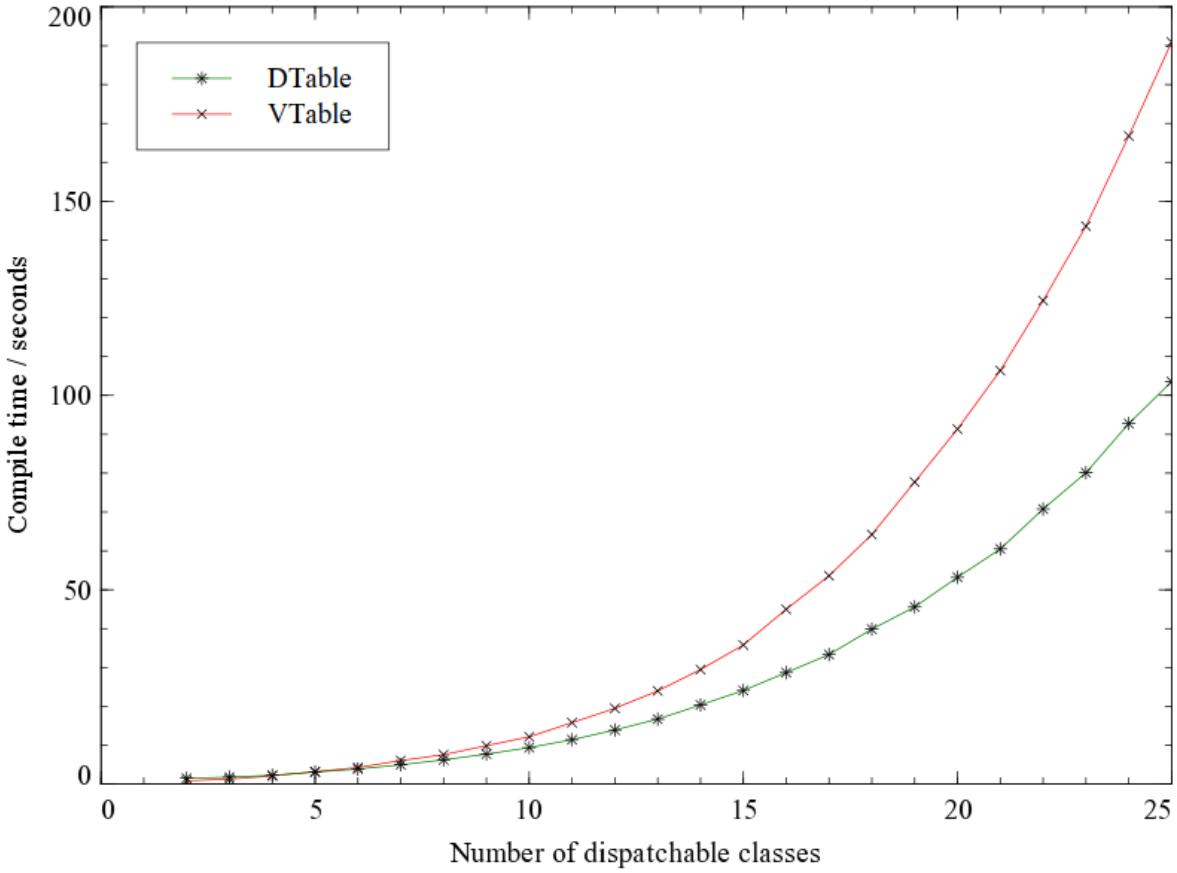

Figure 7: Comparing compile-time with regards to number of dispatchable classes.

> From a design standpoint, cosmic hierarchies often give rise to generic containers of "objects". The content of the containers are often unpredictable and lead to unexpected runtime behavior. Bjarne Stroustrup's classic counterexample considered the possibility of putting a battleship in a pencil cup - something a cosmic hierarchy would allow but that would probably suprise a user of the pencil cup.

.

Binding dispatch functions: Both solutions requires that the dispatcher functions are bound to the dispatcher at runtime. For n dispatchable classes this requires $n * n$ bindings, one for each possible combination. If the dispatcher uses mirroring the number of possible combination drops to $\frac{n*(n-1)}{2} + n$. Of course the developer might exploit some sort of preprocessor code, e.g. using the Boost Preprocessor Library, to avoid having to write all the binding code manually.

Declaring the dispatcher type: As a special requirement the VTable solution also needs the programmer to declare the dispatcher type. This is done with 2 simple lines of code:

Listing 11: These two lines declares the exact dispatcher type using some clever preprocessor code in `vtable.h`.

```
#define MULTIDISPATCHER_PARAMS (NameOfDispatcherClass)((Type1)(Type2
    )...)
#include <OpenTissue/utility/dispatchers/vtable.h>
```

5.4 Thread safety

Neither DTable nor VTable are thread-safe by themself. However, it is trivial to add thread-safety by guarding them with mutexes or similar syncronization mechanisms.

If `Boost Multi Array` is thread-safe with regards to concurrent lookups, then DTable is also thread-safe with concurrent dispatchings. However concurrent lookup and binding or several concurrent bindings are not thread-safe.

The VTable solution is not even thread-safe with regards to concurrent lookups as it uses several internal variables for bookkeeping.

It's worth noting, that the current geometry dispatcher in OpenTissue is inheritely thread-safe. The function table used is just a simple two-dimentional array, so concurrent lookups are thread-safe, and since the table is static, there are no binding issues. The dispatch functions used by the geomtry dispatcher is, however, not thread-safe, but that's be on our interest in this project.

5.5 External dependencies

One final parameter to consider is the external dependencies needed by the two solutions. This is measurement is indirectly tied to requirement **Rq.2**, as a solution with fewer dependencies is often considered more attractive from a programmers point of view.

- The DTable solution depends on Boost Shared Pointers[3] and Boost Multi Array[1].
- The VTable solution depends on Boost Shared Pointers and the Boost Preprocessor Library[2].

5.6 Conclusion on comparison

It is clear from the comparison that DTable is superior to VTable with regards to both run-time overhead, compile-time overhead and scalability. With regards to code overhead, the solutions are almost identical, except for the extra declaring needed by VTable.

While DTable is slightly more thread safe than VTable, neither of them is truly thread safe, so they both require some extra protection before being used in a multithreaded fashion.

Overall, DTable seems to be superior to VTable. Though DTable is not completely type safe, it does encapsulate the type issues very well, thus protecting the programmer from making mistakes. So from a practical point of view, DTable will be the preferred solution.

6 Adding extra parameters

In some situation it's impractical to return the result from a function call as a ordinary return value. This is especially true, when the result is a complex type or it needs to be added to an existing result. In such cases it's better to pass the function a pointer or reference to a preallocated object, in which the function can store it's results. The geometry dispatcher in OpenTissue is such an example where the result of a collision detected between two geometry objects is zero or more contact points, which needs to

be added to a contact point graph for all geomtry objects in a scene. It is possible to return such a result as an ordinary return value, by allocating the contact points along with some container, e.g. a vector, and return a pointer to the container. The caller can then add the contact points from this container to the contact point graph and deallocate the container. However this solution is a bit messy and it adds the overhead of allocating and deallocating the container for collision test. A muh better solution is to simply pass the contact point graph as a reference to the collision detection function as let it add the contact points directly. Thus we would like to add support for an optional third parameter to our dispatcher solutions.

In order to do this, we simply extend the dispatcher's template to include an extra template for the type of the third parameter. Unlike the two dispatchable parameters this third parameter should be passed on to the dispatch function unchanged, so the rest of the changes are trivial. Unfortunately there is still one problem left. Even when this third parameter is unused and set to be `void` or some empty dummy class, it still counts as a parameter for the dispatch function. So in order to correctly support dispatch functions with only two parameters, we are forced to add a partial template specialization for handling this special case. Thus we have to dublicate most of the solution for this specialization, which make the solution more difficult to maintain later on.

Ideally we would like at support an arbitrary number of extra parameters, however this is not possible as C++ templates currently only support a fixed number of template parameters. However variadic templates could solve this problem (see section 8.1).

Currectly only the preferred solution, DTable, have been extended to support the optional third parameter, but it would be trivial to extend VTable in a similar fashion.

7 Dispatchers in other physics engines

While I have primary focused on OpenTissue and it is current geometry dispatcher, it is also worth investigating similar dispatchers in other physics engines. I have investigated a few.

7.1 ODE v0.9

Dispatching in ODE is done by using a fixed sized 2D array of function pointers indexed with a static enumerator of geometry classes. Unlike OpenTissue it also have a bind function, `setCollider()`, so the programmer can register new dispatch functions and the type enumerator have 4 extra *user classes*, thus allowing the programmer to add 4 new types to the dispatcher without altering ODE. The dispatch functions are responsible for casting the geometry objects to their true types and the dispatch functions uses unchecked C-style static casting.

7.2 Bullet v2.66A

Dispatching in Bullet is done by using a fixed sized 2D array of dispatch functors indexed with a static enumerator of geometry classes. Unlike OpenTissue it also have a bind function, `btCollisionDispatcher::registorCollisionCreateFunc()`, so the programmer can register new dispatch functors. However, the geometry classes are fixed. The functors are responsible for casting the geometry objects to their true types and the built-in functors uses unchecked C-style static casting.

7.3 JigLib v0.84

Dispatching in JigLib is done by using a variable sized tables of functors created using nested vectors. Dispatch functions are registered in the table with the function

`RegisterCollDetectFunctor()`. The table is indexed with a static enumerator of the built-in geometry classes, but the programmer is free to add new geometry classes as long they get a unique ID. The functors are responsible for casting the geometry objects to their true types and the built-in functors uses unchecked C-style static casting.

7.4 Tokamak v1.0.4A

Dispatching in Tokamak is done is a nested `switch-case` using a static enumerator of geometry classes. The dispatch functions completely avoids casting by having all the geometry classes collected in a `union` in a single class. This solution is completely static as neither dispatch functions nor geometry classes can be altered without altering Tokamak.

7.5 Benefits of using DTable

All these 4 engines could benifit from the DTable solution presented in this project. Tokamak has a completely static dispatching technique, so the benefits are obvious, and ODE and Bullet both uses a technique similar to OpenTissue. All 4 engines could benefit from the improved type-safety that DTable provides.

8 Conclusion and future work

In the project I have presented two solutions, DTable and VTable, for approximating dynamic multi dispatching. I have evaluated the two solutions and compared them against the current geometry dispatcher in OpenTissue. I have concluded that DTable is the preferred solution in practice, when used within the problem domain of the geometry dispatcher, as it performs well in adding to protecting the programmer from most type related issues. VTable, on the other hand, have some nice theoretical features by being truly type safe. In this project, the dispatch functions are bound to the VTable dispatcher at run-time, as per relaxation **Rl.3**. However another interesting possibility, which could exploided in the future, would be to map dispatch functions to virtual member function in a single class, and then bind a whole dispatching object. By doing this, the compiler can verify that all dispatch functions are actually implemented, thus converting a run-time failure to a compile-time check.

This project have primarily been focused on the geometry dispatcher in OpenTissue, but I have also verified that dynamic multiple dispatching is a problem in other physics engines besides OpenTissue, and that these could also benefit from the solutions presented in this project.

One might notice that the solution in [16] can accomplish dynamic multiple dispatch based on two variables using a single dynamic cast, while VTable needs two. This difference stems from requirement **Rq.3** (see page 3). Both solutions uses dynamic casts to tell the true type of a dispatchable object to an otherwise unknown dispatch function. The solution in [16] only needs one dynamic cast because the dispatch function is implemented as a virtual function in one of the dispatchable classes and thus already knows the true type of it is own object. With requirement **Rq.3** both objects has to tell their true type to the dispatch function, thus the need for two dynamic casts. This is a trade off between flexibility and performance. In that sense, VTable can actually be consided as being a dynamic multiple dispatcher between 3 variables, if you count the dispatch function as a variable. Another performance overhead for this flexibility is an additional memory and compile-time overhead caused by the internal classes in the VTable dispatcher. In comparison, the solution is [16] has no such overhead.

8.1 Using Variadic Templates

Templates in the current C++ standard, *ISO/IEC 14882:2003*[4][1], is limited to handling a fixed number of template parameters. However this limitation will be handled in the upcoming C++ standard, *C++0x*[15], by introducing *Variadic Templates*.

Variadic templates introduces a new ellipsis notation indicating an arbitrary number of parameters, similar to the existing ellipsis notation for functions. Listing 12 shows a simple function template, `print()`, which demonstrates the new notation. The function, `print()`, takes an arbitrary number of parameters and print them to `std::cout` separated with commas and ending with a `std::endl`.

Listing 12: A simple print function template implemented with variadic templates. Notice how the parameters are peeled off in the recursive call in the first specialization, until only one parameter is left and the second specialization ends the recursion.

```
// Primary definition of the template function, which is
// actually never used.
template <
    typename... Types
>
void print(Types...);

// First template specialization, which takes two or more
// parameters. This specialization will print the first
// parameter and a comma.
template <
    typename T1
  , typename T2
  , typename... Types
>
void print<T1, T2, Types...>(T1 t1, T2 t2, Types... types>
{
  std::cout << t1 << ", ";
  print(t2, types...); // Recursively call itself.
};

// Second template specialization, which takes exactly one
// parameter. This specialization ends the recursion by
// printing it's parameter and an std::endl.
template<T1 t1>
void print<T1>(T1 t1){
  std::cout << t1 << std::endl;
};

// Third template specialization, which takes exactly zero
// parameters. This specialization is only defined to sack
// of completion.
template<>
void print<>(void){
  std::cout << std::endl;
};
```

The reader is encuraged to read [10] and [9] for a complete describtion of Variadic Templates in C++0x.

[1] Informally known as *C++98*

The directory `/Source/Future` on the included CD-ROM contains reimplementations of DTable and VTable, which shows variadic templates can be used to improve the solutions:

- Variadic templates can be used to generalize DTable to take any number of extra parameters by specifying the optional extra parameters using ellipsis notation. This both makes DTable more flexible and we avoid the code duplication from the partial template specialization, thus making the implementation both cleaner and easier to maintain.
- Variadic templates can replace all the preprocessor code in VTable by specifying the supported dispatchable types using ellipsis notation. These types can then be recursively unpacked in the assembly of the dispatcher, thus avoiding the extra unintuitive preprocessor declaration step needed by the current implementation.

Variadic Templates have already been implemented in GNU gcc v4.3, which was released on March 5, 2008.

8.2 Dynamic dispatching on more than two variables

Both solutiones presented in this project are trivial to extend to more than two variables. However, just as the need for dynamical multiple dispatch on two variables are relatively rare in practice, so is the need for dynamical multiple dispatch on more than two variables even more rare [12].

A Contents of included CD-ROM

`/Articles`		Constains copies of most of the articles referenced in the work.
`/Report`		Constain a copy of this project along with the result from an OpenTissue light-review for the code.
`/Source`		This directory contains all the source code developed during this project.
	`/Benchmark`	Constains the benchmarking programs used to evalute the dispatchers in this project.
	`/demo`	Contains demo applications, which illustrates how to use the dispatchers in a real program.
	`/Future`	Contains implementation of DTable and VTable using variadic templates.
	`/OpenTissue`	Contains the dispatcher implementations including all helper classes.
	`/unit_tests`	Contains unit testing programs for the dispatchers

B UML 2 reference

This appendix contains a quick reference of the subset of UML 2 used in this project. This reference is not a complete reference of UML 2. See [11] for a more complete reference of UML 2 and [5] for the exact specifications. This reference will only describe UML 2 with regards to C++.

B.1 Class diagrams

This first type of UML 2 diagrams used in this project are *class diagrams*, which are used to illustrate the static design of classes.

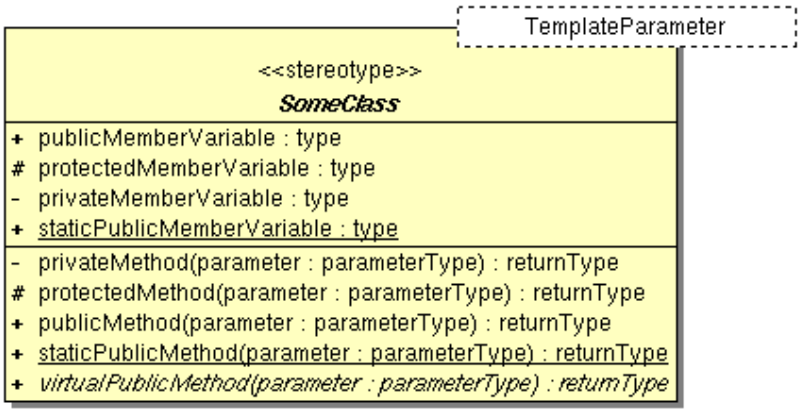

Figure 8: An example class illustrated in UML 2. This examples shows some of the different aspects of a class diagram.

In a class diagram, a class is represented by a box with three compartments. The top compartment constains the name of class and the prototype, if the class has one. A prototype is used to describes the overall role of a class in the design. The only two prototypes is used in this project is *interface*, used with pure abstract classes, and *degenerate*, used with classes, which contains absolutely no information. The middle compartment contains any member variables. These variables are prepended with either + (public), # (protected) or - (private) to indicate accesslevel. Futher more, static variables are underlined. Finally, the bottom compartment contains any member functions in the class. These are also prepended with either + (public), # (protected) or - (private) to indicate accesslevel. Futher more, static member functions are underlined and virtual functions are written in italic. In addition, if the class is a template class, the top right corner of the class representation is overlayed with another box containing the template variables.

Figure 8 shows an example of a class representation in UML 2.

A class diagram can also illustrate relationships between classes. Figure 9 shows the 5 types of relationships used in this project.

Nesting A nesting relationshipo simply indicate that a class is nested inside the namespace of another class.

Realization The realization relationship indicate that a class realises another, and is used in one of two situations. The realization can either be that a class implements some interface, a pure abstract class, or that a class is realized from a class template. In the later situation, the relationship has the *bind* prototype and the diagram will also the bindings between template parameters and types. In both situations the realization relationship indicate a strong *is-a* relationship.

Generalization A generalization relationship indicates that one class is derived from another. Unlike with realization relationship, the derived is expected to add, modify or specialize the base class. The generalization relationship also indicates a *is-a* relationship but it's weaker than with realization.

Composition A composition indicate a *has-a* relationship, where a class has one or more instances of another among it's member variables. This relationship also indicate a kind of ownership.

Dependency A dependency relationship indicates that one class uses another, e.g. calling one of it's member functions.

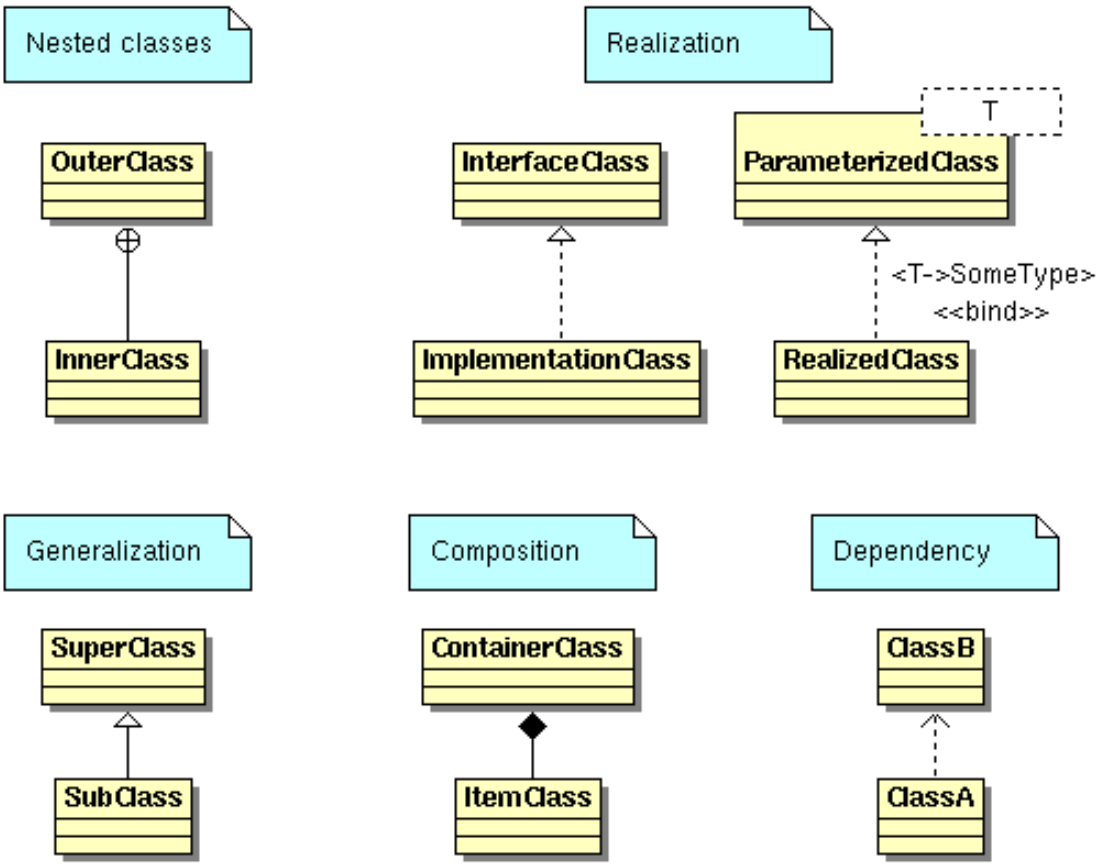

Figure 9: Examples of some of the different types of relationship in UML 2

UML 2 also have many other types of relationships, which can be used to reveal many more details about a design. See [5] for a complete description.

B.2 Collaboration diagrams

The second type of UML 2 diagrams used in are *collaboration digrams*, which are used to illustrate the interactions and flow of information between classes when performing a specific task.

Figure 10 shows such a collaboration diagram.

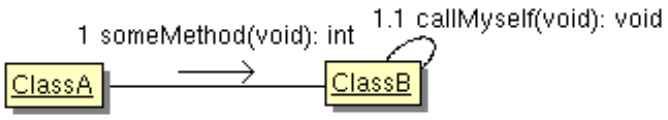

Figure 10: A collaboration diagram in UML 2. `ClassA` calls `ClassB::someMethod()`, which in turn calls `ClassB::callMyself()` before returning to `ClassA`.

In a collaboration diagram, boxes indicate objects and lines indicate collaboration between objects. Each message flowing between objects, usually by means of calling some member function, are illustrated with an arrow next to the lines. Each message is prepended with a sequence number, which indicate the order of execution. The sequence numbers can be multileveled, each level separated with a period character, to indicate subtasks. E.g. in Figure 10 message 1.1 is a subtask of message 1, which means that is has to complete before message 1 can complete.

Collaboration diagrams can also contains other notations to illustrate other details of a task, but those are not used in this project.

References

[1] *The Boost Multidimensional Array Library*, . URL `http://www.boost.org/libs/multi_array/doc/user.html`.

[2] *The Boost Library Preprocessor Subset for C/C++*, . URL `http://www.boost.org/libs/preprocessor/doc/index.html`.

[3] *Boost Smart Pointers*, . URL `http://www.boost.org/libs/smart_ptr/smart_ptr.htm`.

[4] Iso/iec 14882:2003, 2003. URL `http://webstore.ansi.org/`.

[5] Omg unified modeling language (omg uml), superstructure, v2.1.2, November 2007. URL `http://www.omg.org/docs/formal/07-11-02.pdf`.

[6] Lorenzo Bettini, Sara Capecchi, and Betti Venneri. Translating Double-Dispatch into Single-Dispatch. In *Proceedings of the Second Workshop on Object Oriented Developments (WOOD 2004)*, volume 138 of *ENTCS*, pages 59–78. Elsevier, 2005. URL `http://rap.dsi.unifi.it/ bettini/bibliography/files/doubledisp.pdf`.

[7] Lorenzo Bettini, Sara Capecchi, and Betti Venneri. Double Dispatch in C++. *Software - Practice and Experience*, 36(6):581 – 613, 2006. URL `http://rap.dsi.unifi.it/ bettini/bibliography/spelxb.ps.gz`.

[8] Stephen C. Dewhurst. *C++ Gotchas*. Addison-Wesley Professional Computing Series. Addison-Wesley, 2003. ISBN 0-321-125185-5.

[9] Douglas Gregor and Jaakko Järvi. Variadic templates for c++0x. *Journal of Object Technology*, 7(2):31–51, February 2008. URL `http://www.jot.fm/issues/issue_2008_02/article2.pdf`.

[10] Douglas Gregor, Jaakko Järvi, and Gary Powell. Variadic templates (revision 3). Technical Report N2080=06-0150, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, October 2006. URL `http://www.osl.iu.edu/~dgregor/cpp/variadic-templates.pdf`.

[11] Allan Holub. Allen holub's uml quick reference. Online Reference, August 2007. URL `http://www.holub.com/goodies/uml/`.

[12] Daniel H. H. Ingalls. A simple technique for handling multiple polymorphism. In *Proceedings of ACM OOPSLA Conference, Portland, OR*, November 1986. URL `http://portal.acm.org/citation.cfm?id=960112.28732`.

[13] Robert C. Martin. Acyclic visitor, January 1996. URL `http://www.objectmentor.com/resources/articles/acv.pdf`. Submitted to PLoPD-96.

[14] Robert C. Martin. *The Visitor Family of Design Patterns*, chapter 29, pages 525–558. Prentice Hall, 1 edition, February 2002. ISBN 978-0135974445. URL `http://www.objectmentor.com/resources/articles/visitor.pdf`.

[15] Alisdair Meredith. State of c++ evolution (pre-bellevue 2008 mailing), February 2008. URL `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2507.html`.

[16] Dr. Carlo Pescio. Multiple Dispatch - A new approach using templates and RTTI. *C++ Report*, June 1998. URL `http://www.eptacom.net/pubblicazioni/pub_eng/mdisp.html`.

[17] Herb Sutter and Andrei Alexanderscu. *C++ Coding Standards - 101 Rules, Guidelines and Best Practices*. C++ In-Depth Series. Addison-Wesley, 2005. ISBN 0-321-11358-6.

[18] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates - The Complete Guide*. Addison-Wesley, 2003. ISBN 0201734842.