Policies and Traits

Motivation

http://etutorials.org/Programming/Programming+Cpp/Chapter+8.+Standard+Library/8.4+Traits+and+Policies/

- A **policy** is a class or class template that defines an **interface as a service to other classes**.
  A **trait** is a class or class template that **characterizes a type**, possibly a template parameter.
- Traits define type interfaces, and policies define function interfaces, so they are closely related.
- Sometimes, a single class template implements traits and policies.

- The **typical application programmer** might never use traits and policies directly. Indirectly, however, they are used in the string class, I/O streams, the standard containers, and iterators **just about everywhere in the standard library**.

What's the idea?

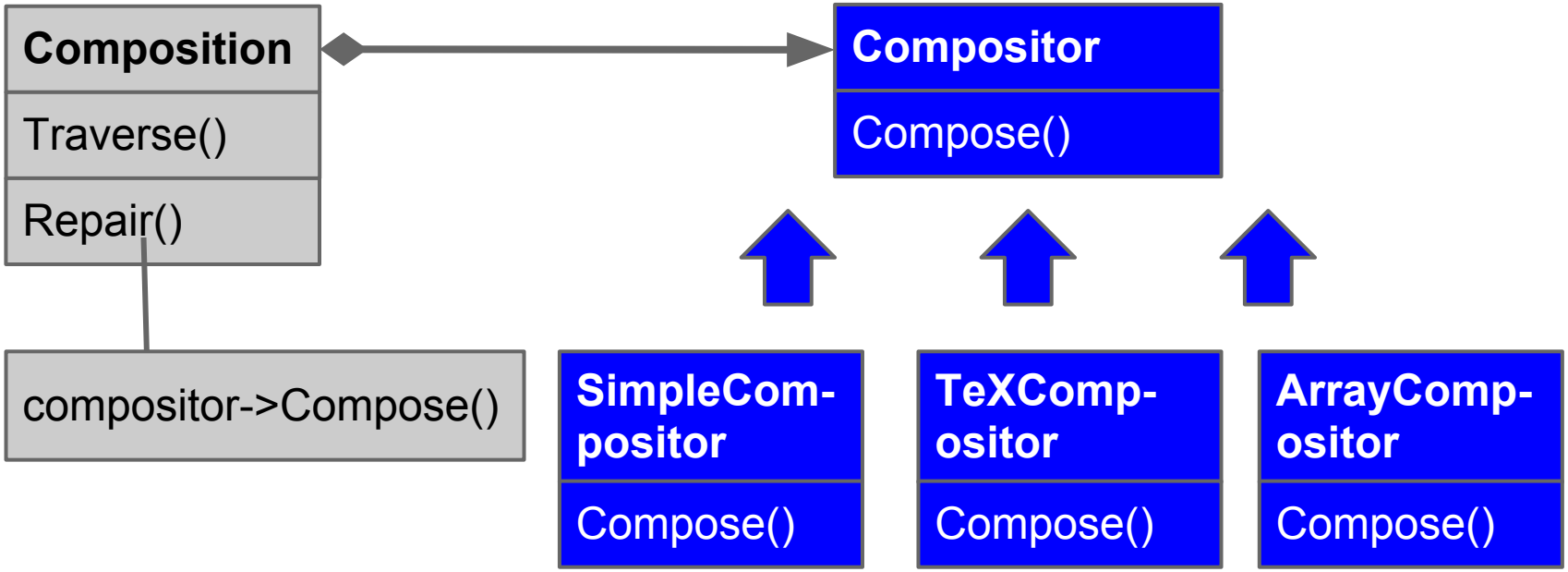
http://en.wikipedia.org/wiki/Policy-based_design
http://www.boost.org/doc/libs/1_52_0/libs/type_traits/doc/html/boost_typetraits/background.html

- **Policy-based design** [...] is a computer programming paradigm based on an idiom for C++ known as **policies**. It has been described as a **compile-time variant** of the **strategy pattern** [...].

- [...] there are times in generic programming when "generic" just isn't good enough - sometimes the differences between types are too large for an efficient generic implementation. This is when the traits technique becomes important - by **encapsulating those properties that need to be considered on a type by type basis inside a traits class, we can minimize the amount of code that has to differ from one type to another**, and maximize the amount of generic code.

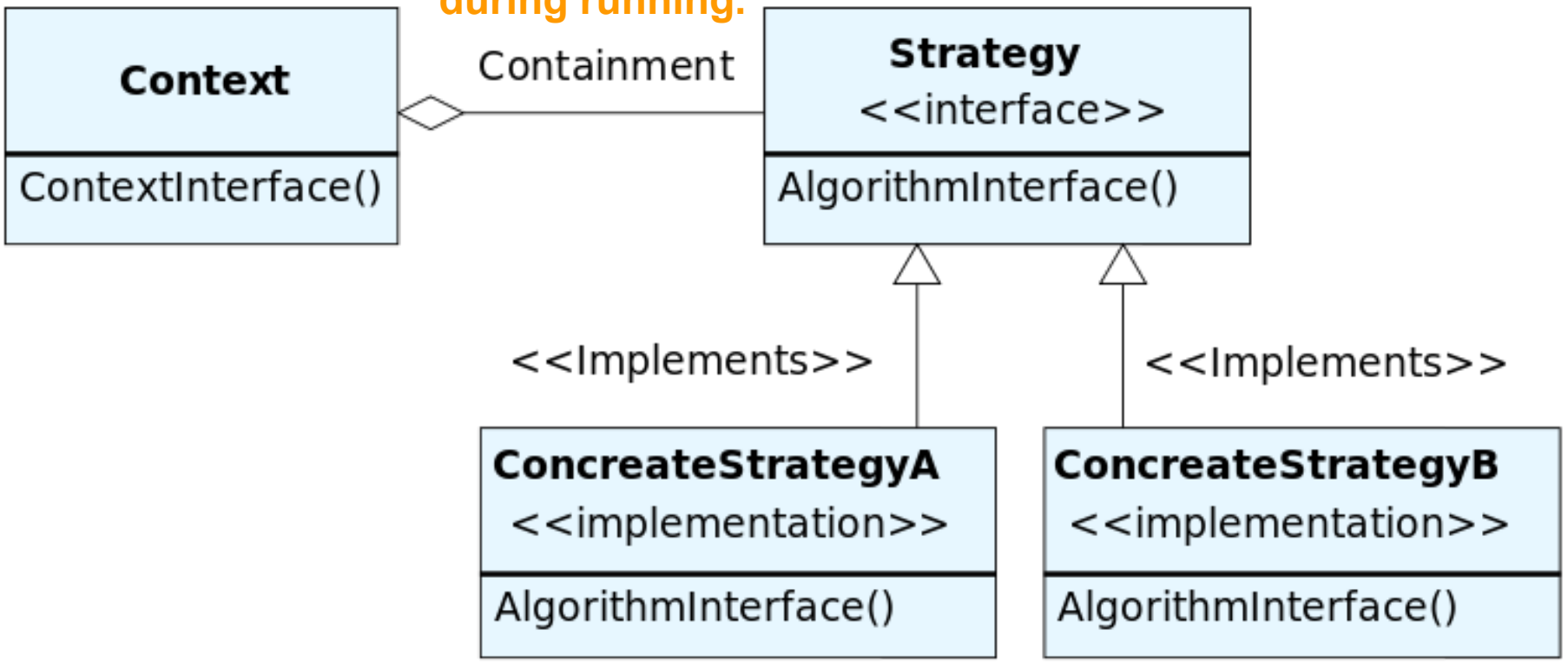
Strategy Pattern

- Assume you want to program an editor, and whenever something is changed, it checks, if it has to recompose line breaks.



Strategy Pattern - Structure

Depending on the type of object, the reference points to, the behaviour changes.
Behaviour can be dynamically changed during running.



When to Use Recommendations

E. Gamma et al.; Design Patterns

- Many related classes differ only in their behaviour (but are called in equal ways)
- You want to employ **Different Variants of an algorithm**, e.g. with different space/time constraints.
- You can **hide data structures** in specific strategy implementations.
- **Multiple conditional statements in a class** --> Move to Strategy class for greater flexibility and better overview.

Consequences

- *You create families of related algorithms*. Inheritance from the abstract Strategy may help factor out common behaviour.
- *Alternative to subclassing*. Allows dynamic switching of the algorithm, reasonable naming (you need a name for the algorithm, not the class, that uses it), and **no exponential increases of subclasses for combinations of orthogonal behaviour**.
- *Alternative to conditional statements*, which make code less easily maintainable, and may lead to branch misprediction etc.

More Consequences

- *Clients must be aware of different Strategies*. A default behaviour is not foreseen in this simple case. Clients need to choose something. Sometimes a reasonable default behaviour may be implemented, when there is no Strategy object at all.
- Strategy interface probably provides *very generous interface* with information for all types of concrete strategies. This may produce performance issues. Or the context may pass itself (see below).
- If you have stateless concrete strategy objects, they may be shared among several contexts (and can be flyweights; state information might be storable in the context).

```
void CDCRecoHit::setTranslators(ADCCountTranslatorBase*    const adcCountTranslator,
                                CDCGeometryTranslatorBase* const cdcGeometryTranslator,
                                TDCCountTranslatorBase*    const tdcCountTranslator)
/*
static void setTranslators(boost::shared_ptr<ADCCountTranslatorBase>    const& adcCountTranslator,
                           boost::shared_ptr<CDCGeometryTranslatorBase> const& cdcGeometryTranslator,
                           boost::shared_ptr<TDCCountTranslatorBase>   const& driftTimeTranslator)
 */
{
  s_adcCountTranslator    = adcCountTranslator;
  s_cdcGeometryTranslator = cdcGeometryTranslator;
  s_tdcCountTranslator    = tdcCountTranslator;
}
```

Function to set Translation Strategies for certain information, so that the CDCRecoHit can (re)calculate stuff. This can be changed on the fly.

```
CDCRecoHit::CDCRecoHit(const CDCHit* cdcHit)
  : GFAbsWireHit(c_nParHitRep)
{
  if (s_adcCountTranslator == 0 || s_cdcGeometryTranslator == 0 || s_tdcCountTranslator == 0) {
    B2FATAL("Can't produce CDCRecoHits without setting of the translators.")
  }

  // get information from cdcHit into local variables.
  m_wireID      = cdcHit->getID();

  m_tdcCount              = cdcHit->getTDCCount();
  m_driftLength           = s_tdcCountTranslator->getDriftLength(m_tdcCount, m_wireID);
  m_driftLengthResolution = s_tdcCountTranslator->getDriftLengthResolution(m_driftLength, m_wireID);

  m_adcCount    = cdcHit->getADCCount();
  m_charge      = s_adcCountTranslator->getCharge(m_adcCount, m_wireID);

  // forward wire position
  TVector3 dummyVector3 = s_cdcGeometryTranslator->getWireForwardPosition(m_wireID);
  fHitCoord(0) = dummyVector3.X();
  fHitCoord(1) = dummyVector3.y();
  fHitCoord(2) = dummyVector3.z();
  // backward wire position
  dummyVector3 = s_cdcGeometryTranslator->getWireBackwardPosition(m_wireID);
```

Abstract Strategy Class.
As you see, lots of parameters have to be put into the interface and perhaps it would have been useful to give a reference to the Hit instead, but unfortunately this would have as well created coupling issues.

```
class TDCCountTranslatorBase {
public:
  /** Constructor. */
  TDCCountTranslatorBase() {}

  /** Destructor. */
  virtual ~TDCCountTranslatorBase() {}

  /** Function for getting a drift length estimation.
   *
   *  @param tdcCount               This is the "drift time" saved in the CDCHit. Actually it can be the
   *                                such as trigger time jitter, propagation time of the signal in the wi
   *                                Actual translators should however get the appropriate drift length in
   *  @param wireID                 Object to identify hit wire.
   *  @param timeOfFlightEstimator  This is an estimator for the time, that lies between the event time/
   *                                of the particle and the time at which the ionisation happened, which
   *                                for the calculation of the drift length/actual drift time.
   *  @param ambiguityDiscriminator Information to resolve left/right ambiguity.
   *  @param z                      z-position for determining the in-wire-propagation time.
   *  @param theta                  Angle under which the particle moves through the drift-cell in r-phi.
   *
   *  @return  Best estimation of closest distance between the track and the wire.
   */
  virtual float getDriftLength(unsigned short tdcCount     = 0,
                               const WireID& wireID        = WireID(),
                               float timeOfFlightEstimator = 0.,
                               bool ambiguityDiscrimiator  = false,
                               float z = 0, float theta = static_cast<float>(TMath::Pi() / 2.)) = 0;

  /** Uncertainty corresponding to drift length from getDriftLength of this class.
   *
   *  @param driftLength            Output of the getDriftLength function.
   *  @param ambiguityDiscriminator Information to resolve left/right ambiguity.
   *  @param z                      z-position for determining the in-wire-propagation time.
   *  @param theta                  Angle under which the particle moves through the drift-cell in r-phi.
   */
  virtual float getDriftLengthResolution(float driftLength          = 0.,
                                         const WireID& wireID       = WireID(),
```

```
/** Translator mirroring the simple Digitization. */
class SimpleTDCCountTranslator : public TDCCountTranslatorBase {
public:
  /** Constructor, with the additional information, if propagation in the wire shall be considered. */
  SimpleTDCCountTranslator(bool useInWirePropagationDelay = false) :
    m_useInWirePropagationDelay(useInWirePropagationDelay), m_eventTime(0) {
  }

  /** Destructor. */
  ~SimpleTDCCountTranslator() {};

  /** If trigger jitter was simulated, in every event one has to give an estimate of the effect. */
  void setEventTime(short eventTime = 0) {
    m_eventTime = eventTime;
  }

  /** */
  float getDriftLength(unsigned short tdcCount,
                       const WireID& wireID        = WireID(),
                       float timeOfFlightEstimator = 0,
                       bool = false,
                       float z = 0, float = static_cast<float>(TMath::Pi() / 2.));

  /** Uncertainty corresponding to drift length from getDriftLength of this class.
   *
   *  Currently in the simple digitization just a Gaussian smearing is used.
   *
   *  @return Uncertainty on the drift length, currently 0.001.
   */
  float getDriftLengthResolution(float,
                                 const WireID&,
                                 bool,
                                 float, float);
```

Concrete Strategy class (in a separate shared object)

Policies

http://en.wikipedia.org/wiki/Policy-based_design

```
template <typename OutputPolicy, typename LanguagePolicy>
class HelloWorld : private OutputPolicy, private LanguagePolicy
{
    using OutputPolicy::print;
    using LanguagePolicy::message;

public:
    // Behaviour method
    void run() const
    {
        // Two policy methods
        print(message());
    }
};

class OutputPolicyWriteToCout
{
protected:
    template<typename MessageType>
    void print(MessageType const &message) const
    {
        std::cout << message << std::endl;
    }
};
```

```
class LanguagePolicyEnglish
{
protected:
    std::string message() const
    {
        return "Hello, World!";
    }
};

class LanguagePolicyGerman
{
protected:
    std::string message() const
    {
        return "Hallo Welt!";
    }
};
```

```cpp
int main()
{
    /* Example 1 */
    typedef HelloWorld<OutputPolicyWriteToCout, LanguagePolicyEnglish> HelloWorldEnglish;

    HelloWorldEnglish hello_world;
    hello_world.run(); // prints "Hello, World!"

    /* Example 2
     * Does the same, but uses another language policy */
    typedef HelloWorld<OutputPolicyWriteToCout, LanguagePolicyGerman> HelloWorldGerman;

    HelloWorldGerman hello_world2;
    hello_world2.run(); // prints "Hallo Welt!"
}
```

- You specify as template parameter from which class you inherit.
- Still more efficient than runtime-polymorphism, as you will usually not have pointers to the base type/ virtual functions etc.
- In contrast to usual object-oriented programming, inheritance is NOT an "is an" relationship. Often (but not necessarily always) private inheritance is appropriate (inheriting the implementation, NOT the interface).

- If the policy is inherited publicly, you can have additional methods.

- As stated before, policies work best, when they are fully orthogonal. If they are not, maybe traits can help you work around.

Traits

http://www.boost.org/doc/libs/1_52_0/libs/type_traits/doc/html/boost_typetraits/background.html

- The type-traits classes share a unified design; most classes inherit from the type
  - true_type if the type has the specified property
  - false_type otherwise.
  - but in principle other types, e.g. integer keys may be interesting.

```
template <class T, T val>
struct integral_constant
{
   typedef integral_constant<T, val>  type;
   typedef T                          value_type;
   static const T value = val;
};

typedef integral_constant<bool, true>  true_type;
typedef integral_constant<bool, false> false_type;
```

How does the Trait know which Type to choose?

- This depends on what exactly you want to know..., but generally from specializations... which follows somewhat strange rules... [this are implementation examples from boost]

```
template <typename T>
struct is_void : public false_type{};

template <>
struct is_void<void> : public true_type{};
```

is_void<T>::value is true if T is of type void, false otherwise

Such partial specialisation is as well allowed.

```
template <typename T>
struct is_pointer : public false_type{};

template <typename T>
struct is_pointer<T*> : public true_type{};
```

- Rule of Thumb: If you can overload like the following you can as well make different partial specialisation. (But the details differ)

```
void foo(T);
void foo(U);
```

```
template <typename T>
class c{ /*details*/ };

template <typename T>
class c<U>{ /*details*/ };
```

- You could as well make traits where you make full specialisations for some policies, that then are grouped together into one family.

```
template <typename T>
struct remove_extent
{ typedef T type; };

template <typename T, std::size_t N>
struct remove_extent<T[N]>
{ typedef T type; };
```

No limits to creativity.

Remove_extent<int[4][5]>::type would evaluate to the type int[5].

```
template<typename Iter1, typename Iter2>
Iter2 copy(Iter1 first, Iter1 last, Iter2 out);
```

Can be made fast with memcpy, if there are some conditions fullfilled, that can be tested with traits.