Templates and Patterns?

David Hibler
Language specific patterns?

- What if language is not object oriented?
- Do “Gang of Four” patterns apply?
- What if language has more facilities?
- Patterns/idioms?
Templates

- Not the same as Java generics
- Not a syntactic cover for Object
- Compiler generates code at compile time
- Logically equivalent to an interpreter at compile time
- Tricky – debugging etc.
Multiple Inheritance

- Do interfaces really replace multiple inheritance?
- Complicated
- Name problems
- Same base class – virtual base
- Hard on compiler writer
  - Duplicate code from multiple files etc.
Combine BOTH features

- Multiple inheritance
- Templates
- Templated inheritance
template<typename T>
class MyClass : public T {
  ...
};

MyClass<Policy1> m1;
MyClass<Policy2> m2;
Multiple Policies

template<type T1, type T2, ...>
class MyClass : public T1, T2, ... {
    ...
}
}
Changing virtual functions

Class A { public: virtual void f(); }
Class B { no f() }

template<type T>
class MyClass public: T { 
    Public: void f();
}

Example

MyClass<A> m1;
MyClass<B> m2;
m1.f() is virtual
m2.f() is not
Policies

Derived class is host (shell)
Class built from policies
Host glues policies together
Smart Pointer example
Defaults can be used to simplify
Generic Add

template<typename T>
T accum (T const* beg, T const * end) {
    T total = T();
    while (beg != end) {
        total += *beg;
        ++beg;
    }
} return total
A templated addition

- char name[] = “templates”;
- int num[] = {1, 2, 3, 4, 5};
- Add characters in name using a template function divide by 5 to get average.
- Add integers in num using template function divide by 9 to get average.
- Java can’t do this.
What is result?

- Average of integers is 3
- Average of characters is -6
Traits

template<typename T>
Class AccumulationTraits;
template<>

class AccumulationTraits<char> {  
public: typedef int AccT;
};

template<>  
class AccumulationTraits<int> {  
public: typedef long AccT;
}
Fix of template function

template<typename T>
typename AccumulationTraits<T>::Acct accum(T const* beg, T const* end)
{
    AccT total = ...
    return total;
}
Curiously Recurring Template Pattern (CRTP)

template <typename Derived>
class CuriousBase {
    ...
};

class Curious: public CuriousBase<Curious> {
    ...
};
Why?

- Base class knows type of derived class
- Can add functions appropriate for the derived class