Composition verses Inheritance

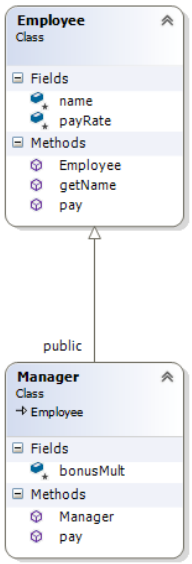
Think of composition as a has a relationship. A car "has an" engine, a person "has a" name, etc.
Think of inheritance as an is a relationship. A car "is a" vehicle, a person "is a" mammal, etc.

Prefer Composition over Inheritance (ALWAYS). Large (and deep) inheritance hierarchies are difficult to debug (you tend to jump around in the class structure) and are delicate (change a base class member overridden in derived class and you can break the whole thing.
Composition aggregates other objects and calls on them (delegation) when their services needed. Ex. I (a person) have a watch. If someone asks me the time, I defer to the watch. (see Composition_Intro project)

Inheritance implements IS_A

```cpp
#include "employee.h"
using namespace std;
Employee::Employee(string theName, float thePayRate)
{
  name = theName;
  payRate = thePayRate;
}

string Employee::getName() const
{
  return name;
}

float Employee::pay(float hoursWorked) const
{
  return hoursWorked * payRate;
}
```

```cpp
#include "manager.h"
using namespace std;

Manager::Manager(string theName,
                 float thePayRate,
                 int bonusMult)
  : Employee(theName, thePayRate), bonusMult(bonusMult)
{
}

float Manager::pay(float hoursWorked) const
{
 return bonusMult*(Employee::pay(hoursWorked));
}
```




```cpp
class Employee {
public:
  Employee(std::string theName, float thePayRate);

  std::string getName() const;
  float pay(float hoursWorked) const;

protected:
  std::string name;
  float payRate;
};
```

```cpp
#include "employee.h"
class Manager : public Employee {
public:
  Manager(std::string theName,
          float thePayRate,
          int bonusMult);

float pay(float hoursWorked) const;

protected:
  int bonusMult;
};
```

Manager inherits all base class members and data

Calling base class

Manager must call base class if needed

See constructor IL

Employee::pay(hoursworked)

(Scope it)

Pay is in 2 places do not even need pay in manager, but may want to change how pay works override pay to get diff behavior

Or just use base implementation (code reuse).

You must scope or get infinite recursion (remove Employee:: in manager)

If do not call base class employee in constructor. Compiler will attempt to create default constructor to call, if base does not have one will not compile.

Demo remove employee IL from manager

Do not need to call base class in destructor, compiler handles it.

Base class constructors are automatically called for you if they have no argument. If you want to call a superclass constructor with an argument, you must use the subclass's constructor initialization list. Unlike Java, C++ supports multiple inheritance (for better or worse), so the base class must be referred to by name, rather than "super()".

Protected:

Way for derived classes to get at innards of base class (member vars and functions) without exposing implementation details to world.

Why? Without it no derived class can get at base class members

Go to Composition_intro project

Composition:

When an object posseses something

I have a watch

I have a Pixel XL phone

Model it, create a human that takes a watch pointer (see above project)
add a method to give the human a watch and ask the human for the time, if he has a watch he gives the time, if not he says I dont have a watch

Which to Use:

Think of composition as a **has a** relationship. A car "has an" engine, a person "has a" name, etc.
Think of inheritance as an **is a** relationship. A car "is a" vehicle, a person "is a" mammal, etc.

In general, it's a good idea to prefer less inheritance. Use composition wherever possible, and inheritance only in the specific situations in which it's needed. Large inheritance hierarchies in general, and deep ones in particular, are confusing to understand and therefore difficult to maintain. Inheritance is a design-time decision and trades off a lot of runtime flexibility.

Virtual:
Go back to the Inheritance_intro project and add the following function

```
//want 1 function to handle all instances in my employee-manager inheritance
heiarchy.
//so I receive as a base class (Employee) pointer regardless of whether the
object is
//an Employee or a Manager
void outputPay(Employee *pEmp, int hw){
      // Assume all employees worked 40 hours this period.
      cout << "Employee "<<pEmp->getName()<<" has earned "<<pEmp->pay(40)<<"
dollars"<< endl;
}
```

and append this to main:

```
:
//lets see who gets paid what
//using pointers
outputPay(&e, hw);
outputPay(&m, hw);
```

We are now passing a reference to the Employee and Manager object to a function. The function receives the object as a pointer. And here is the 'slicing' problem;

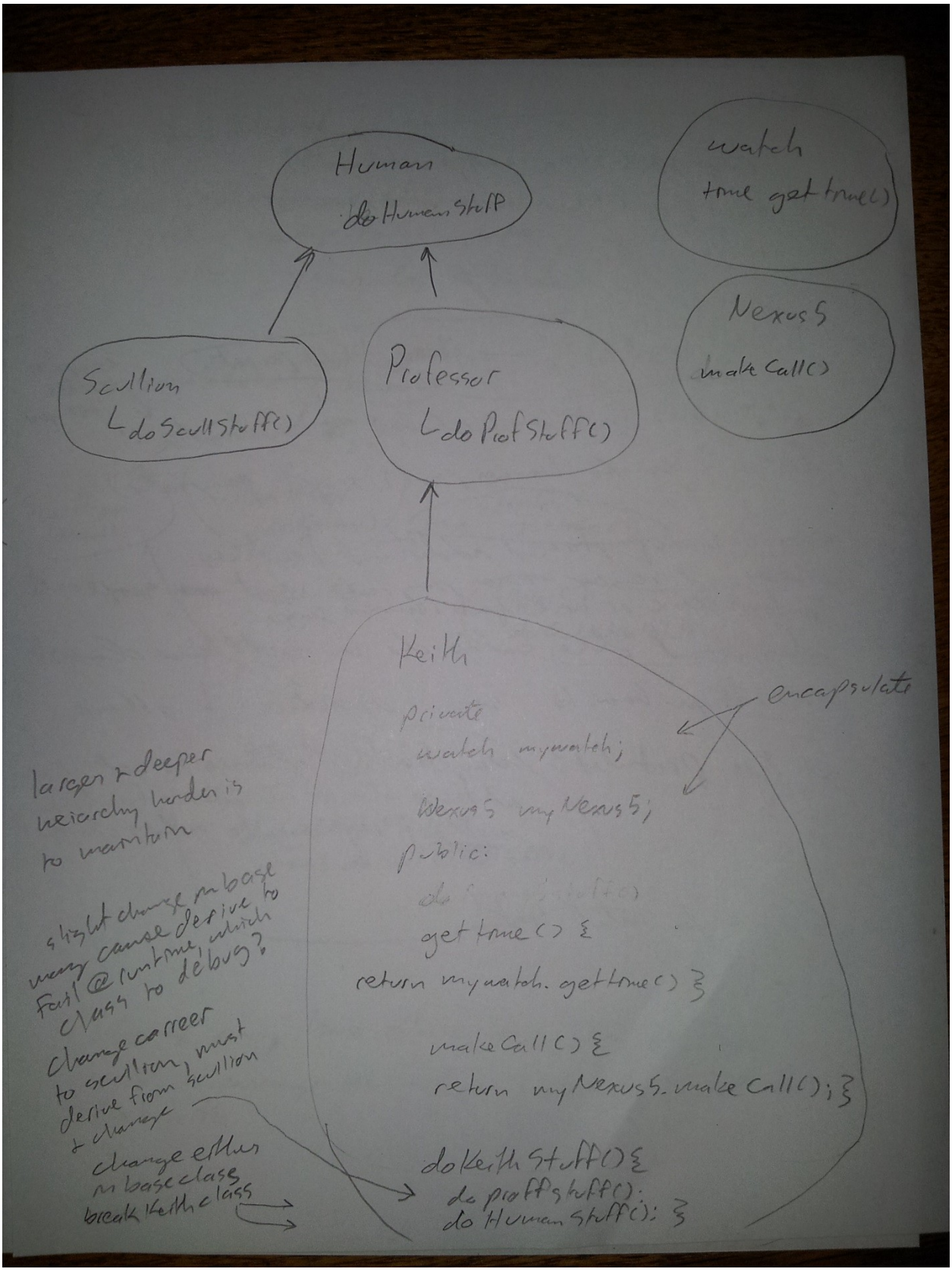
If pay(...) is not virtual, then you will always call the pay defined for the type of pointer passed in (employee::pay()). Manager::pay() will not be called.

The fix: Make the function virtual in header file of the base class, every class derived thereafter is virtual by default;
AS A COURTESY TO FUTURE DEVELOPERS PUT VIRTUAL IN ALL DERIVED CLASSES SO THEY DONT HAVE TO FIGURE IT OUT

No free lunch however: virtual functions require an additional lookup in something called a V-Table to find the most derived instance of a function.

Incidently C++ gives you a choice;
non virtual functions: faster function calls but vulnerable to slicing
virtual functions: slower function calls but invulnerable to slicing

Java BTW is always virtual.


Human
doHumanStuff
watch
time getTime()
Nexus5
makeCall()
Scullion
doScullStuff()
Professor
doProfStuff()
Keith
private
watch myWatch;
Nexus5 myNexus5;
public:
getTime() {
return myWatch.getTime() }
makeCall() {
return myNexus5.makeCall(); }
doKeithStuff() {
doProffStuff();
doHumanStuff(); }
encapsulate
larger + deeper
heirarchy harder is
to maintain
slight change in base
may cause derive to
fail @ runtime, which
class to debug?
change career
to scullion, must
derive from scullion
+ change
change either
in base class
break Keith class