Object Oriented Programming Tips

By Carey Nachenberg

Before designing your base and derived classes for Project 3 (or for that matter, any other school or work project), make sure to consider the following best practices. These tips will help you not only write a better object-oriented program, but also help you get a better grade on P3! Try your best to leverage the following best practices in your program, but don't be overly obsessive–it's rarely possible to make a set of perfect classes. That's often a waste of time. Remember, the best is the enemy of the good (enough). Here we go!

1. Avoid using dynamic cast to identify common types of objects. Instead add a function to check for a kind of behavior:

Don't do this:

```
void decideWhetherToAddOil(Actor* p)
{
  if (dynamic_cast<BadRobot*>(p) != nullptr || dynamic_cast<GoodRobot*>(p) != nullptr ||
    dynamic_cast<ReallyBadRobot*>(p) != nullptr || dynamic_cast<StinkyRobot*>(p) !=
    nullptr)
  p--->addOil();
}
```

Do this instead:

```
void decideWhetherToAddOil(Actor* p)
{
// define a common function, have all Robots return true, all biological
// organisms return false
  if (p--->requiresOilToOperate())
    p--->addOil();
}
```

2. Always avoid defining specific isParticularClass() functions for each type of object. Instead add a function to check for a kind of common behavior that spans multiple classes:

Don't do this:

```
void decideWhetherToAddOil(Actor* p)
{
  if (p--->isGoodRobot() || p--->isBadRobot() || p--->isStinkyRobot())
    p--->addOil();
}
```

Do this instead:

```
void decideWhetherToAddOil(Actor* p)
{
// define a common function, have all Robots return true, all biological
// organisms return false
  if (p--->requiresOilToOperate())
    p--->addOil();
}
```

3. If two related subclasses (e.g., BadRobot and GoodRobot) each directly define a member variable that serves the same purpose in both classes (e.g., m_amountOfOil), then move that member variable to the common base class and add any necessary accessor and mutator functions for it to the base class. So the Robot base class should have the m_amountOfOil member variable defined once, with, say, getOil() and addOil() functions, rather than defining this variable directly in both BadRobot and GoodRobot.

Don't do this:

```
class SmellyRobot : public Robot
{
  …
  private:
    int m_oilLeft;
};
class GoofyRobot : public Robot
{
  …
  private:
    int m_oilLeft;
};
```

Do this instead:

```
class Robot {
  public:
    void addOil(int oil)
    {
      m_oilLeft += oil;
    }
    int getOil() const
    {
      return m_oilLeft;
    }
  private:
    int m_oilLeft;
};
```

4. Never make any class's data members public or protected. You may make class constants public, protected or private.

5. Don't introduce a public function if it is to be used directly only by other functions within the same class that declares it. Make it private or protected instead.

6. Your StudentWorld functions should never return a vector, list, or iterator to StudentWorld's private game objects or pointers to those objects. Only StudentWorld should know about all of its game objects and where they are. Instead, StudentWorld should do all of the processing itself if an action needs to be taken on one or more game objects that it tracks.

Don't do this:

```
class StudentWorld
{
  public:
    vector<Actor*> getActorsThatCanBeZappedAt(int x, int y) {
      …
      // create a vector with actor pointers and return it
    }
};
class NastyRobot
{
  public:
    virtual void doSomething()
    {
      …
      vector<Actor*> v;
      vector<Actor*>::iterator p;
      v = studentWorldPtr--->getActorsThatCanBeZappedAt(getX(), getY());
      for (p = actors.begin(); p != actors.end(); p++)
        p--->zap();
    }
};
```

Do this instead:

```
class StudentWorld
{
  public:
    void zapAllZappableActors(int x, int y) {
      for (p = actors.begin(); p != actors.end(); p++)
        if (p--->isAt(x,y) && p--->isZappable())
          p--->zap();
    }
};
```

```
class NastyRobot
{
  public:
    virtual void doSomething()
    {
      …
      studentWorldPtr ---> zapAllZappableActors(getX(), getY());
    }
};
```

7. If two subclasses have a function that shares some common functionality, but also has some differing functionality, use an auxiliary method to factor out the differences:

Don't do this:

```
class StinkyRobot : public Robot
{
  …
  virtual void takeAction()
  {
    doCommonThingA();
    passStinkyGas();
    pickNose();
    doCommonThingB();
  }
};

class ShinyRobot : public Robot
{
  …
  virtual void takeAction() {
      doCommonThingA();
      polishMyChrome();
      wipeMyDisplayPanel();
      doCommonThingB();
    }
};
```

Do this instead:

```
class Robot
{
  public:
    virtual void takeAction()
```

```
  {
    doCommonThingA(); // all robots do this
    doDifferentiatedStuff(); // each kind of robot does this differently
    doCommonThingB(); // all robots do this
  }
 private:
  virtual void doDifferentiatedStuff() = 0;
};

class StinkyRobot : public Robot
{
 ...
 private:
  // define StinkyRobot's version of the differentiated function
  virtual void doDifferentiatedStuff()
  {
    passStinkyGas();
    pickNose();
  }
};

class ShinyRobot : public Robot
{
 ...
 private:
  // define ShinyRobot's version of the differentiated function
  virtual void doDifferentiatedStuff(){
    // only Shiny robots do these things
    polishMyChrome();
    wipeMyDisplayPanel();
  }
};
```

Notice that *doDifferentiatedStuff()*is private, because no functions in other classes should call it directly. It should be private, not protected, in the base class; derived classes's overriding a virtual function is not considered to be using the base class version in a way that the public/protected/private distinction matters.

Good luck!