How to restructure this code hierarchy (relating to the Law of Demeter)

Asked 15 years, 11 months ago Modified 15 years ago Viewed 629 times

3

I've got a game engine where I'm splitting off the physics simulation from the game object functionality. So I've got a pure virtual class for a physical body

```
class Body
```

from which I'll be deriving various implementations of a physics simulation. My game object class then looks like

```
class GameObject {
public:
   // ...
private:
   Body *m_pBody;
};
```

and I can plug in whatever implementation I need for that particular game. But I may need access to all of the `Body` functions when I've only got a `GameObject`. So I've found myself writing tons of things like

```
Vector GameObject::GetPosition() const { return m_pBody->GetPosition(); }
```

I'm tempted to scratch all of them and just do stuff like

```
pObject->GetBody()->GetPosition();
```

but this seems wrong (i.e. violates the Law of Demeter). Plus, it simply pushes the verbosity from the implementation to the usage. So I'm looking for a different way of doing this.

c++ oop refactoring law-of-demeter

Share
Improve this question
Follow

edited Jan 22, 2009 at 6:43
Svante
51.4k ● 11 ● 83 ● 124

asked Jan 22, 2009 at 5:00
Jesse Beder
34k ● 22 ● 110 ● 146

4 Answers

Sorted by: Highest score (default)

3

The idea of the law of Demeter is that your `GameObject` isn't supposed to have functions like `GetPosition()`. Instead it's supposed to have `MoveForward(int)` or `TurnLeft()` functions that may call `GetPosition()` (along with other functions) internally. Essentially they translate one interface into another.

If your logic requires a `GetPosition()` function, then it makes sense turn that into an interface a la Ates Goral. Otherwise you'll need to rethink why you're grabbing so deeply into an object to call methods on its subobjects.

Share
Improve this answer
Follow

edited Dec 22, 2009 at 23:34

answered Jan 22, 2009 at 8:37
Max Lybbert
20k ● 5 ● 49 ● 69

The GameObject definitely needs stuff like GetPosition(), since it might be, say, a ball (for which MoveForward doesn't make sense). – Jesse Beder Jan 22, 2009 at 15:57

In that case "your logic requires a GetPosition() function" and so "it makes sense turn that into an interface a la Ates Goral." But you should "rethink why you're grabbing ... into an object" (i.e., determine if you need to redesign with, say, ImmovableGameObjects and MovableGameObjects) – Max Lybbert Jan 22, 2009 at 20:07

How would that redesign work? Also, see my comment to Ates Goral. – Jesse Beder Jan 22, 2009 at 20:56

I changed my mind - this is actually what I'm going with. What confused me earlier was that MoveForward and TurnLeft don't make sense in my context, but other functions (with a similar idea) do. – Jesse Beder Jan 23, 2009 at 5:37

3

One approach you could take is to split the `Body` interface into multiple interfaces, each with a different purpose and give `GameObject` ownership of only the interfaces that it would have to expose.

```
class Positionable;
class Movable;
class Collidable;
//etc.
```

The concrete `Body` implementations would probably implement all interfaces but a `GameObject` that only needs to expose its position would only reference (through dependency injection) a `Positionable` interface:

```
class BodyA : public Positionable, Movable, Collidable {
    // ...
};
```

```
class GameObjectA {
private:
    Positionable *m_p;
public:
    GameObjectA(Positionable *p) { m_p = p; }
    Positionable *getPosition() { return m_p; }
};

BodyA bodyA;
GameObjectA objA(&bodyA);

objA->getPosition()->getX();
```

Share

Improve this answer

Follow

edited Jan 22, 2009 at 5:21

answered Jan 22, 2009 at 5:11

Ates Goral

140k ●27 ●141 ●191

This just seems to push the problem on to Positionable - we still have to use a double chain to get any relevant information. – Jesse Beder Jan 22, 2009 at 15:56

From what I understand, the idea is that you get rid of m_pBody altogether. Instead you have the Positionable interface provide GetPosition() and related methods. Then you implement those methods in GameBody. The main point of the Law of Demeter is to make sure things that should be hidden are. – Max Lybbert Jan 23, 2009 at 1:55

But what's the difference between m_pBody and m_p? (Really, most things that I'd want to do to/read from a Body, I'd also want to do the same for a GameObject. This whole Positionable/Movable/Collidable thing seems just to skirt around the issue. – Jesse Beder Jan 23, 2009 at 5:35

1

Game hierarchies should not involve a lot of inheritance. I can't point you to any web pages, but that is the feeling I've gather from the several sources, most notably the game gem series.

You can have hierarchies like ship->tie_fighter, ship->x_wing. But not PlaysSound->tie_fighter. Your tie_fighter class should be composed of the objects it needs to represent itself. A physics part, a graphics part, etc. You should provide a minimal interface for interacting with your game objects. Implement as much physics logic in the engine or in the physic piece.

With this approach your game objects become collections of more basic game components.

All that said, you will want to be able to set a game objects physical state during game events. So you'll end up with problem you described for setting the various pieces of state. It's just icky but that is best solution I've found so far.

I've recently tried to make higher level state functions, using ideas from Box2D. Have a function SetXForm for setting positions etc. Another for SetDXForm for velocities and angular velocity. These functions take proxy objects as parameters that represent the various parts of the physical state. Using methods like these you could reduce the number of methods you'd need to set state but in the end you'd probably still end up implementing the finer grained ones, and the proxy objects would be more work than you would save by skipping out on a few methods.

So, I didn't help that much. This was more a rebuttal of the previous answer.

In summary, I would recommend you stick with the many method approach. There may not always be a simple one to 1 relationship between game objects and physic objects. We ran into that where it was much simpler to have one game object represent all of the particles from an explosion. If we had given in and just exposed a body pointer, we would not have been able to simplify the problem.

Share Improve this answer Follow

answered Jan 22, 2009 at 6:08

deft_code

59.1k ● 31 ● 144 ● 224

I may simply go with keeping all of the duplicate methods. But it's pretty frustrating.
– Jesse Beder Jan 22, 2009 at 18:59

0

Do I understand correctly that you're separating the physics of something from it's game representation?

i.e, would you see something like this:

```
class CompanionCube
{
    private:
        Body* m_pPhysicsBody;
};
```

?

If so, that smells wrong to me. Technically your 'GameObject' *is* a physics object, so it should derive from Body.

It sounds like you're planning on swapping physics models around and that's why you're attempting to do it via aggregation, and if that's the case, I'd ask: "Do you plan on swapping physics types at runtime, or compile time?".

If compile time is your answer, I'd derive your game objects from Body, and make Body a typedef to whichever physics body you want to have be the default.

If it's runtime, you'd have to write a 'Body' class that does that switching internally, which might not be a bad idea if your goal is to play around with different physics.

Alternatively, you'll probably find you'll have different 'parent' classes for Body depending on the type of game object (water, rigid body, etc), so you could just make that explicit in your derivation.

Anyhow, I'll stop rambling since this answer is based on a lot of guesswork. ;) Let me know if I'm off base, and I'll delete my answer.

Share Improve this answer Follow

answered Jan 22, 2009 at 6:53

Eddie Parker
4,888 ● 4 ● 37 ● 45

It's sort of a mix. I can't derive game objects from Body because some objects may use different types of Body (like you suggested); but it's not as rigid as "Water should derive from WaterBody" since we may want WaterBody1 and WaterBody2, and choose between those either at runtime or compile-time. – Jesse Beder Jan 22, 2009 at 18:56

Also, the problem with writing a 'Body' class that switches internally is that it pushes the same problem to the *implementation* of Body (since then we'll have similarly derived classes from, say, BodyImpl and we'll want to forward all of the BodyImpl functions to Body).
– Jesse Beder Jan 22, 2009 at 18:58