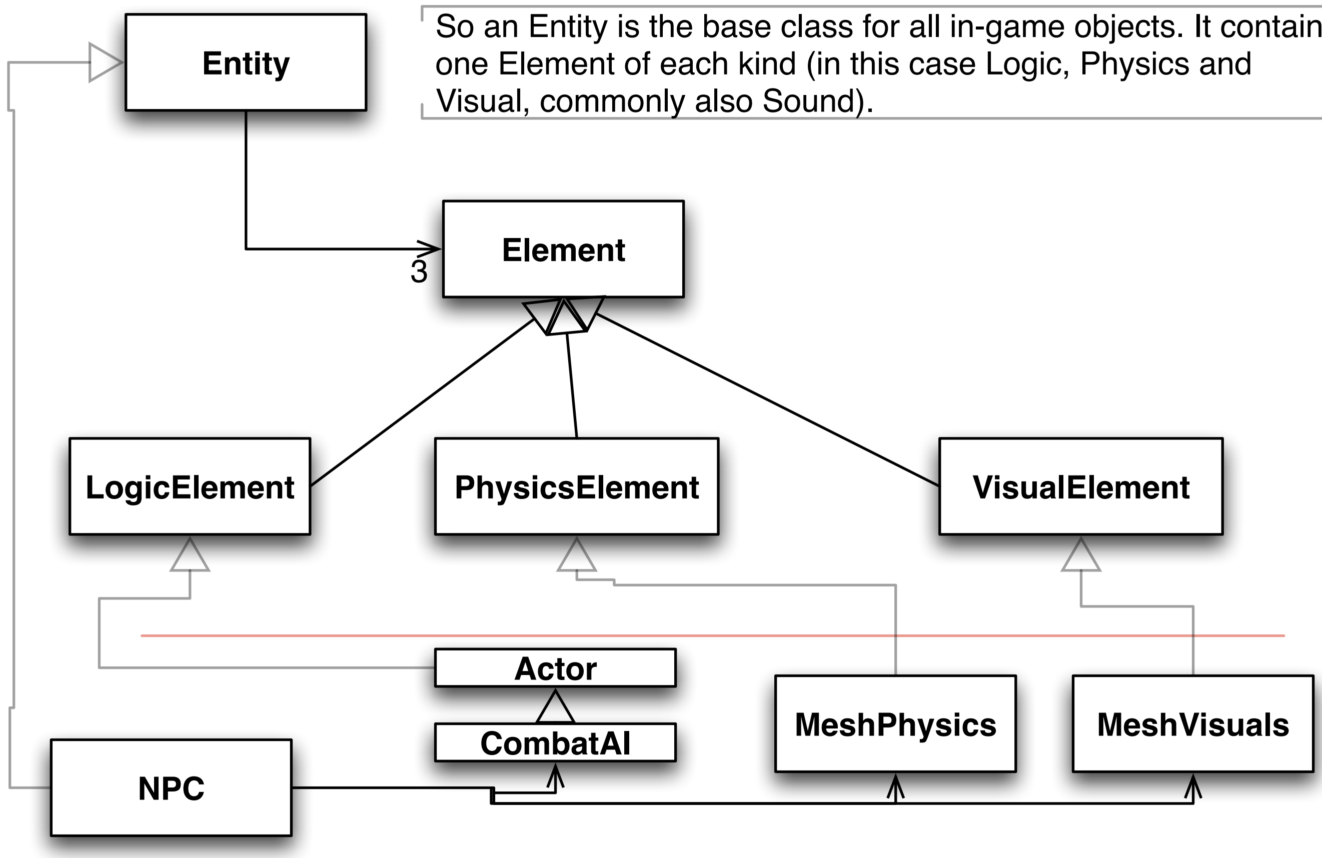
CCP Test

Question 5: Designing a game engine.

Answers by Joachim Bengtsson <joachimb@gmail.com>

I'm a big fan of using composition for building complex objects, and avoiding deep inheritance hierarchies. I've been working on such a model for three game engines now (the released ones are at robot-game.com and friendlystapler.se). Even after three iterations, however, my implementations are far from my visions. That *might* be related to time shortages and rushed implementations, though...

So an Entity is the base class for all in-game objects. It contains one Element of each kind (in this case Logic, Physics and Visual, commonly also Sound).



An NPC, for example, might use the three Elements AI, MeshPhysics and MeshVisuals. NPC's constructor would just be a convenience method for building an Entity from these elements, like so:

```
class NPC(Entity):
  def __init__(self, isFriendlyTowards = None, mesh = Mesh.UnknownMesh):
    self.logics = CombatAI(isFriendlyTowards)
    self.visuals = MeshVisuals(mesh)
    self.physics = MeshPhysics(mesh.variation("simple"))
```

Entity might have methods for storing and setting states, which the mesh could listen to and be triggered from logics. For example, in Actor (the base for eg CombatAI),

```
class Actor(LogicElement):
  ...
  def damage(self, amount):
    self.health -= amount
    self.entity.setState("hit")
    # The above call will tell the parent's visuals' mesh to morph to the
    # animation called 'hit'. Since it shares mesh with the physics,
    # physics will also notice this change.
```

A note about clients and servers: On a client, the Logic and Physics element would be different, potentially even non-existent. The constructors for AI and MeshPhysics might actually be factory methods.

The Crowd NPC is pretty similar to the combat AI above:

```
class CrowdNPC(Entity):
  def __init__(self, meshes = (Mesh.UnknownMesh,)):
    self.logics = RoamingAI()
    mesh = meshes.random()
    self.visuals = MeshVisuals(mesh)
    self.physics = None
```

There'd be an internal event system within Entities, that signals events to all elements. An example relevant to the damage method on the previous page, if a collision occurs with some entity, an event is broadcasted from the Physics element. Some base of AI and Player might listen to this event, and see how hard the collision was, or if the collided-with entity was of some dangerous type, and call damage on self.

This event system could be used also between entities; for example, with the Door entity, a Player might send an Action event whenever he presses the 'E' keyboard key, and it'd get broadcasted to all entities within collision range. When received, the door could open or close.

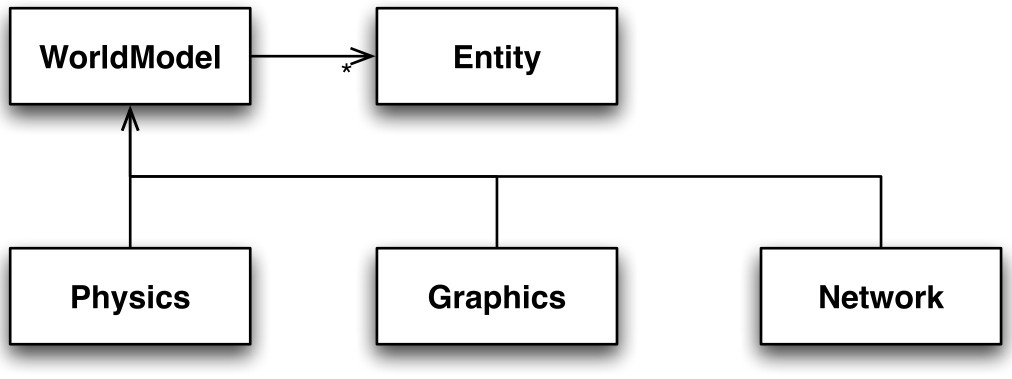
```
class Door(Entity):
  class DoorLogics(Actor):
    ...
    def event(self, evt):
      if evt is PlayerAction and not self.entity.state("destroyed"):
        if self.entity.state("open"):
          self.entity.unsetState("open")
        else:
          self.entity.setState("open")

    def destroyed(self): #called when health =< 0
      self.entity.setState("destroyed")

  def __init__(self):
    self.logics = DoorLogics()
    mesh = Mesh("door.mesh", defaultState = "closed")
    self.visuals = MeshVisuals(mesh)
    self.physics = MeshPhysics(mesh.variation("simple"))
```

Whenever the 'open' state isn't available, the 'closed' animation state will be set. This will affect everything using this mesh, including the physics. Anyone trying to go through the door when it's closed would collide with its MeshPhysics.

The Player Entity would be pretty similar to the NPC Entity. The difference would once again be the Logics element. The server-side Logics would listen to events sent through the network from a Logics element with the same ID on a client. These would be translated into movements and actions in the game world, affecting the physics element, visuals and other entities.

The client-side LogicElement would be non-existent on clients that don't own the entity. On the client that owns the entity, the LogicElement would be almost the same as on the server (for interpolation purposes9, *plus* the sending of user actions part (MoveForward, FirePrimary, etc events).

The client of these entities would be the big sub-engines; Physics, Graphics (on the clients) and Network (and if it had existed, Sound). The container for them would be the WorldModel.



Each sub-engine holds a WorldModel or is regularly handed an appropriate WorldModel each frame, to do its thing on it. Physics takes the entities as input, and modifies their positions, etc, through logic in the engine, or logic in the physics elements of the entities. Graphics might take all entities visible from the player, and render them without affecting them. Network on the server side might pick entities that are flagged as changed and broadcast them through the network. Network on the client side would inject these changes into WorldModel.

There would also be the mother class of these that coordinate their actions, normally called just "ServerApp" and "ClientApp".

I'm also dreaming about a visual editor for building game worlds with this structure. Specific entities like CombatNPC or Door might not even exist in code, but be built visually from the building blocks that are elements; think of it as a cross between Hammer, Interface Builder and a free-form grapher.

I'd love to elaborate further on this subject, but my time is up. Thanks for listening.