**[Aggregate Objects via Components](http://handsomeliuyang.iteye.com/blog/1296738)**

**博客分类：**

* [Replica Island](http://handsomeliuyang.iteye.com/category/194253)

I used to make games like this:

class RenderableMovingCollidableGameObject extends RenderableMovingGameObject {

public void update() {

super.update(); // Parent classes implement rendering and movement.

// implement collision detection here

}

}

class PlayerObject extends RenderableMovingCollidableGameObject {

public void update() {

super.update(); // Run rendering, movement, and collision.

// update the player

}

}

...

// Main loop!

while (true) {

InputSystem.update(); // poll for input.

for (gameObject : ListOGameObjects) {

gameObject.update();

gameObject.draw();

}

}

This isn't a bad way to start making games, but it doesn't scale. Making a good game requires flexibility and the ability to iterate quickly, and this approach starts to break down when the game becomes medium-sized. What if we have more than one game mode? What if the behavior of a specific game object needs to change for a short period of time? What if Level 27's version of the player needs to be subtly different than Level 26's?  
  
In a nutshell, the problem with this approach is that the structure of the code producing a single frame of the game is hard-coded into the program. What I want is something that can change its structure between levels, or between game modes, or even on the fly. I need something more dynamic than a hard-coded for loop.  
  
**Another Approach**  
  
Let's look at game objects as an example. Game objects are entities in the game world like coins, enemies, moving platforms, and the player. They often have similar functionality, so one way to go about implementing a set of similar objects is to use an inheritance tree. We can create a base GameObject class, and from that derive RenderableGameObject, and from that derive RenderableMovingGameObject, and from that derive RenderableMovingCollidableGameObject, etc, etc, etc. Each level of derivation can add some common functionality until the leafs of this tree are specific entities, like the player.  
  
The real problem with inheritance trees is that features that don't need to be inter-related become dependent on each other because of the way the code itself is written. Given the example class structure above, it's not possible to make a GameObject that can move but doesn't need to render (short of littering the code with flags--don't do that). Because of inheritance, a dependency between movement and rendering has been created where none actually needs to exist. What if we could mix and match features on a per-instance basis rather than tying them all together in a class hierarchy? This is where object composition comes in.  
  
Object composition (or "object aggregation" depending on which design patterns book you read) is the idea that an object "has a" feature instead of "is a" feature. Rather than using inheritance or some other code-side method of collecting functionality together in a single object, we make the object manage a list of separate feature objects; the contents of that list can be different per instance.   
  
So for Replica Island, I have a GameObject class that contains a list of GameComponents. A game object with an empty list does nothing; it can't be drawn, or make noise, or hit things, or do anything else to affect the game. GameComponents implement all of those features, and they must be inserted into the GameObject for it to actually be able to act. Here's some psudeo-code of how GameObject and GameComponents work:

class GameObject {

private Array<GameComponent> mComponents;

public void addComponent(GameComponent component) {

mComponents.push(component);

}

public void update(float time) {

for (component : mComponents) {

component.update(time, this);

}

}

}

class GameComponent {

public void update(float time, GameObject parent) {

// ... functionality goes here

}

}

A GameObject just runs all the GameComponents it contains to produce its output each frame. GameComponents in Replica Island implement things like movement, physics, collision detection, sprite animation, rendering, animation selection, AI, player control, etc. Once a generic GameComponent is written it can be inserted in any number of objects; if a specific object requires special functionality, a new GameComponent can be written just for that object without disturbing anything else in the system.  
  
The beauty of this approach is that individual components can be compartmentalized features, and brand new objects in the game world can be created just by sticking different pieces of pre-existing code together in new ways. The result is also dynamic: unlike an inheritance-based object, Replica Island game objects can change their structure at runtime. For example, one of the Android robot's powers is to possess other robots. When an enemy robot is possessed, his AI component is removed and a new component that lets the player control him is inserted. All of a sudden the robot is being driven by the player; all the rest of his code for animation selection, physics, and collision detection, continues to work without realizing that anything has changed. When the player releases the robot, the original component structure can be restored (actually, in this case the robot blows up, taking out anything else near it, but you get the idea).  
  
I've made a couple of games now using components and I'm very happy with the result. In Replica Island I took a couple of shortcuts for speed that damage the goal of completely independent components, but I think it was worth it; the sacrifices I've made for frame rate haven't actually proved detrimental to the flexibility or extensibility of the system.