Composite Pattern Exercise

Geographical Information Systems (GIS)

Geographical Information Systems (GIS)

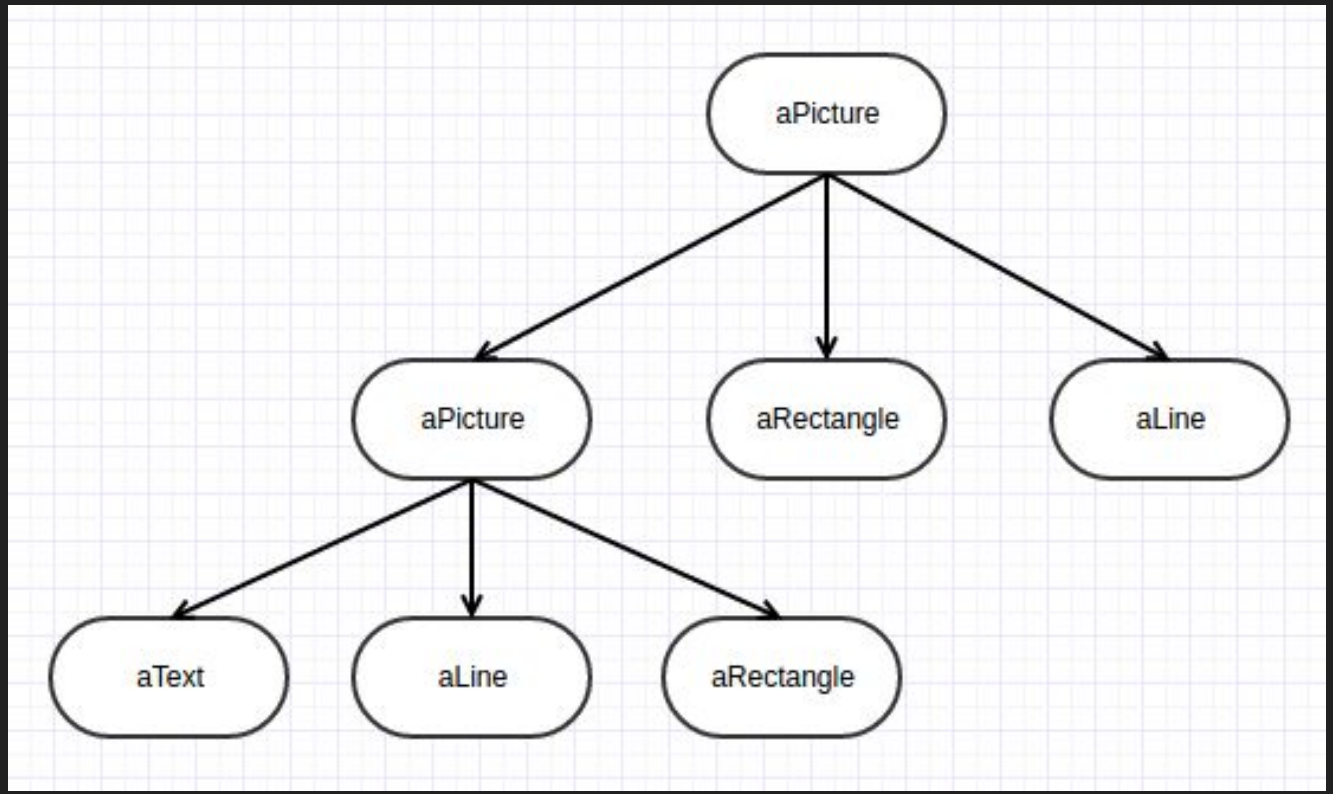
- Create a system that takes in point of interest (POI) information, terrain information, etc. and makes it easy and fast to search and visualize.
- **Example**: When you look at map software on your phone, it doesn’t load every point on earth just the subset you are looking at (and at different view levels)
- We should support multiple types of underlying data (POI, terrain, etc.) as well as a structure to speed up searches and make it easy to view different subsets of the map.

Some starting questions...

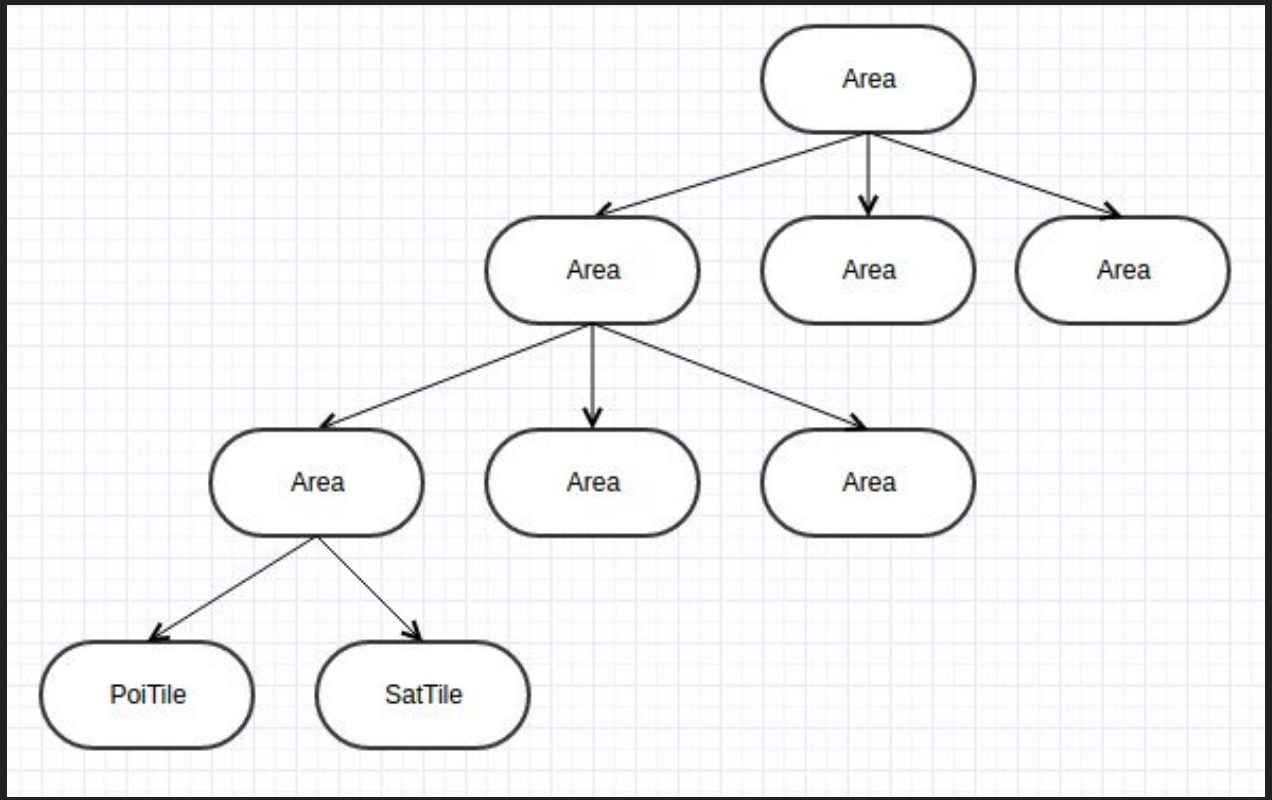
- What data do you need to support?
- Are there natural ways to separate this data into classes?
- Are there natural categories or hierarchies that we can use?
- What data interactions do we need?
- Which ones are between classes? Which ones are with the client?
- What interactions do we need between these categories?
- What interactions are common to all data? Which ones are specific?

How might you structure this system?

Remember this?



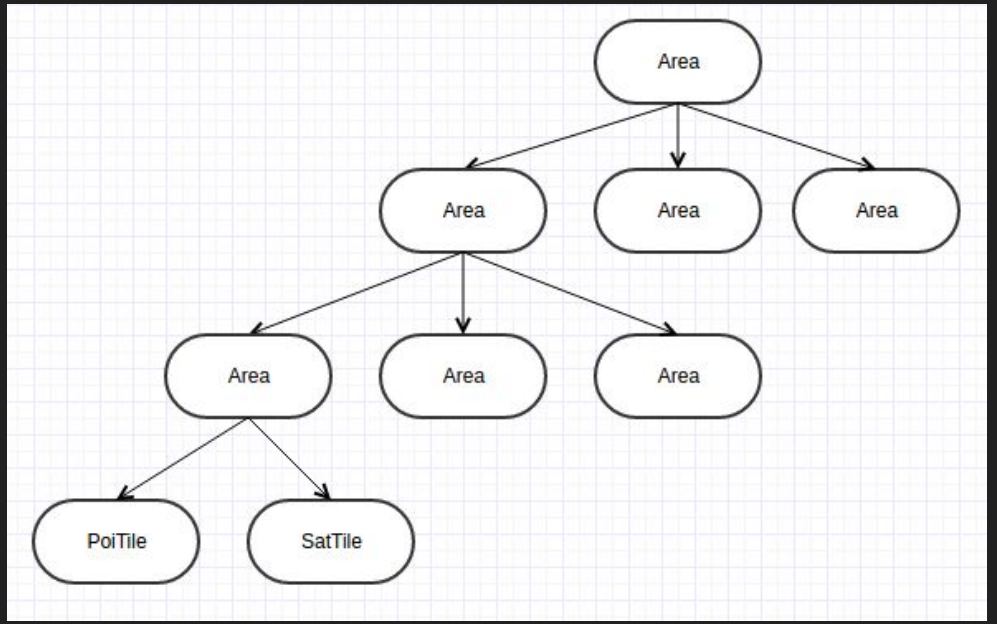
Object Hierarchy



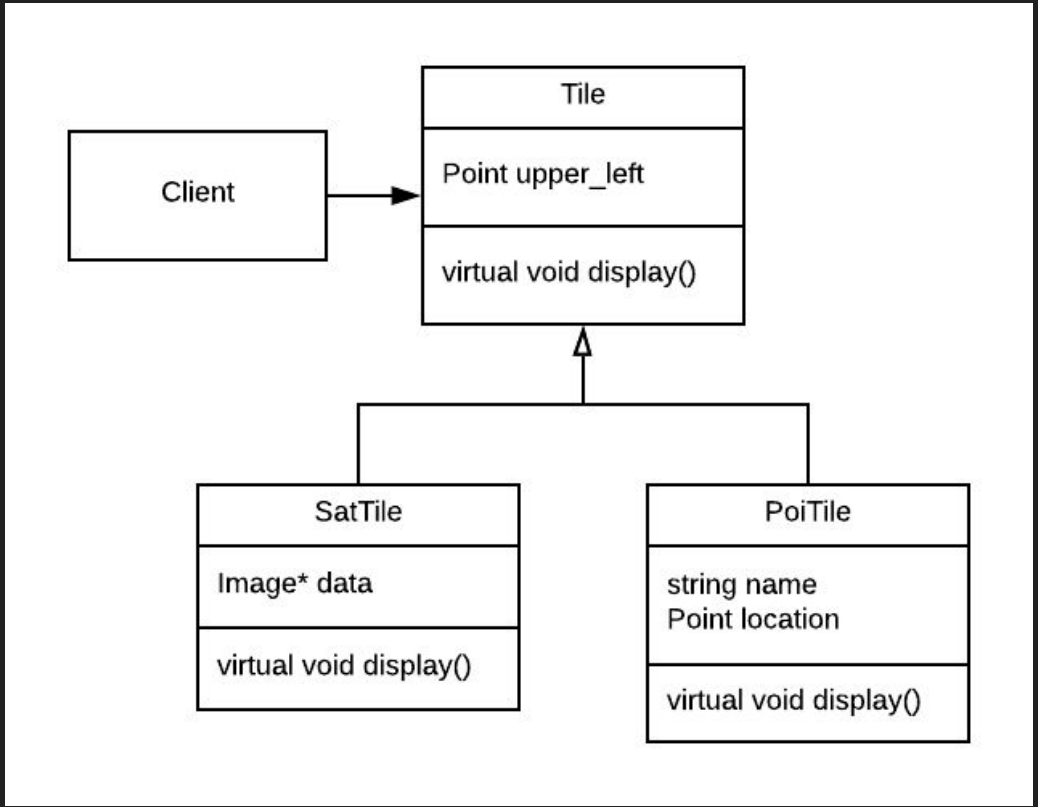
Now let’s create our **class diagrams**

Remember, the composite pattern has the following pieces:

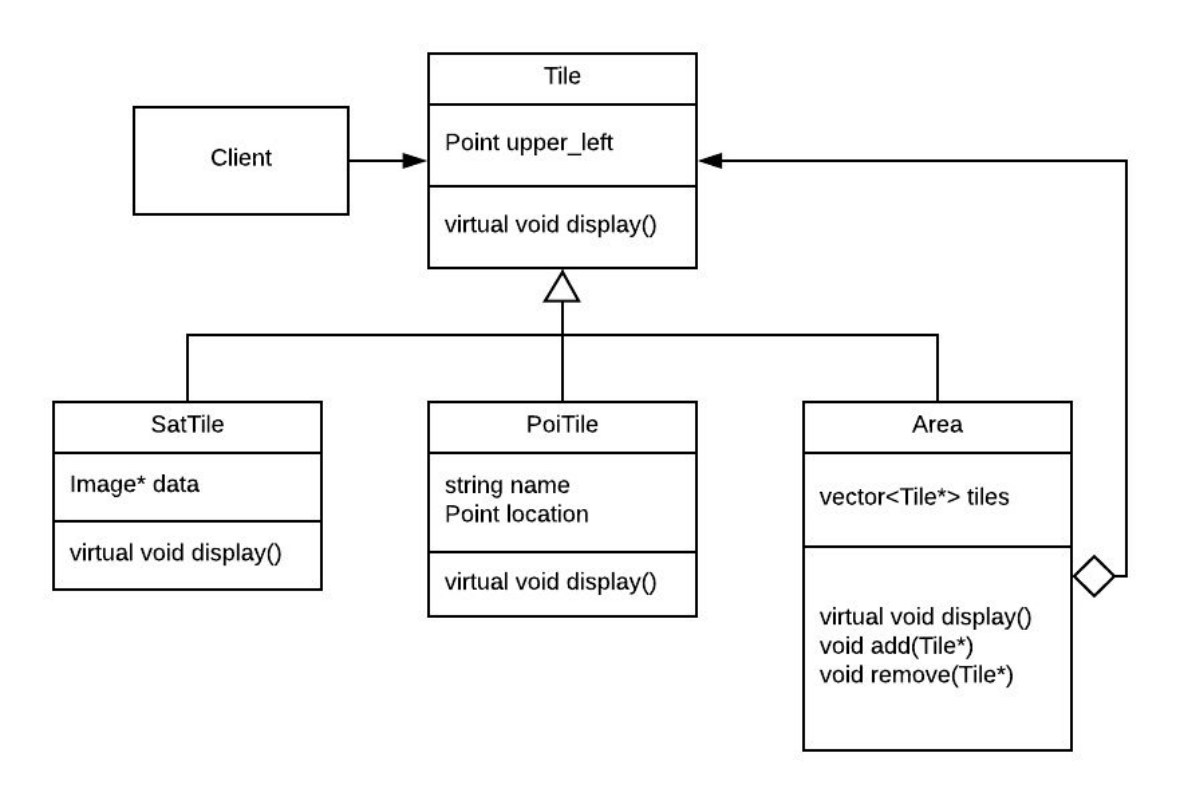
- A client which uses the interface to interact with the system
- A component defines the interface for our system
- Leaf nodes, which represent some function of data of interest
- Composite nodes, which represent collections and modifications of other composites and/or leafs



Base & Leaf Classes



Composite Class



Component Implementation

```
class Tile {
  private:
    Point upper_left;  // struct Point { double x; double y; };
  public:
    Tile(Point upper_left);
    virtual void display() = 0;  // pure virtual interface
};
```

Leaf Implementation

```
class SatTile : public Tile {  // terrain information
  private:
    Image* data;
  public:
    SatTile(Point upper_left, Image* data) : Tile(upper_left) { this->data = data; };
    void display();  // omitted, system dependent
};

class PoiTile : public Tile {  // point-of-interest information
  Private:
    string name;
    Point location
  public:
    PoiTile(Point location, string name) : Tile(upper_left) { this-> name = name; this->location = location };
    void display();  // omitted, system dependent
};
```

Area Implementation

```
class Area : public Tile {  // terrain information
  private:
    vector<Tile*> tiles;
  public:
    Area(Point upper_left) : Tile(upper_left) {};
    void add_tile(Tile* tile) { this->tiles.push_back(tile); };
    void display() {
        for(vector<Tile*>::iterator tile = this->tiles.begin(); tile != this->tiles.end(); tile++) {
            itr->display();
        }
    };
};
```

Now let’s ask a question of our data

“Scotty, show me the nearest Starbucks to UCR”

Assumptions (Given in this class):

- We know the name of the place we want to find
  - `string key = “Starbucks”;`
- We know the area we are looking near
  - `Point query_location = (double UCR_x, double UCR_y); // UCR’s location`
- We want to know the location of the nearest POI(s) matching our query

How do we formulate this into our interface?

What information needs to pass between the classes?

Tile find_poi()

```
Class Tile {
    ….
    Public:
        virtual Point find_poi(string name, Point location) = 0;  // we could also return a PoiTile object
    ….
};
```

PoiTile find_poi()

```
Class PoiTile : public Tile {
    ….
    Public:
    Point find_poi(string name, Point location) {
        if (this->name == name) {
            return this->location;
        }
        return Point(-1,-1); // representative of an invalid point
    };
    …
};
```

SatTile find_poi()

```
Class SatTile : public Tile {
    ….
    Public:
    Point find_poi(string name, Point location) {
        return Point(-1,-1); // the SatTile doesn’t have location data, so it's always invalid
    };
    …
};
```

Area find_poi()

```
Class Area : public Tile {
    ...
    Public:
        Point find_poi(string name, Point location) {
            Point current_closest = Point(-1,-1);
            double current_distance = DBL_MAX;
            for(unsigned i = 0; i < tiles.size(); i++) {
                Point new_location = this->tiles.at(i).find_poi(name, location);
                if (new_location != Point(-1,-1) {
                    if (distance(location,new_location) < current_distance || current_closest == (-1,-1)) {
                        current_closest = new_location;
                        current_distance = distance(location,new_location);  // we don't really need to save this
                    }
                }
            }
            return current_closest;
        };
    …
```

How could we use this structure to reduce the number of searches?

How could we apply these ideas to the display() function?