I believe that our design would support incremental development better than iterative development. The customer already came up with a well-defined software concept, so if we were to practice iterative development, we might run out of things to work on after several cycles. It’s hard to say that the software system is complete as currently defined, but I do not foresee enough new features or added capabilities to make iterative development a good idea.

On the other hand, incremental development makes a lot of sense for our product. The product can be split into many modular pieces pretty well, and each piece can be thoroughly tested before building on top of that. For example, if we were to utilize incremental development for our product, we would probably develop components in the following order:

1. Create a database to store info about users, areas, tasks, etc.
2. Develop user profiles, badges, points, etc.
3. Create a list of charities that donations can be sent to and build the functionality to select a charity and send money
4. Create a map server that pulls map data from the Google Maps API and provides it to the app
5. Overlay green/red/colorless areas that are stored in the database onto the map in the app
6. Create the ability to submit a task where a user reports that they cleaned an area
7. Build functionality for another user to confirm whether or not the first user did in fact clean the area
8. Add the ability to challenge users after submitting a task
9. Develop a way to determine when a user is fraudulently reporting areas as cleaned despite them not being cleaned

Each of these components is clearly defined and can be developed independently or on top of another component. Likewise, testing each individual component should not be too difficult to test. For this reason, I believe that our system design would be more favorable to an incremental approach as opposed to an iterative approach.

When determining which design patterns would be useful for implementing the system, we had to take a lot of details into consideration. How would our various components interact? How complex are the interactions between our server, database, and map and payment APIs? How many attributes and pieces will each individual component require?

These questions helped us determine that the Builder, Adapter, Façade, Memento, and Observer design patterns would all be useful for implementing the system.

The builder design pattern will help us develop the user profile and task entities. The user profile has many different aspects to it – a name, profile picture, attached social media accounts, points balance, favorite charities, completed tasks, sent challenges, medals and badges, etc. The user profile object will be quite complex in nature, especially since many of the attributes of the user entity are references to another entity in our class model. The builder design pattern makes it easier to develop and keep track of the user profile without getting lost in the complexity. Similarly, the task entity has several components as well: the user that submitted the task, the date and time, location, attached photos, the user that confirmed the area as cleaned, etc. Once again, several of these are references to other entities on our system. Using the builder design pattern helps us split the entities up into smaller pieces and build from the ground up.

The adapter design pattern will be useful when developing the interaction between our server and various payment APIs. Our product will accept payment in the form of credit/debit cards, PayPal, and Google/Apple Pay. These are all individual services that our system will need to be integrated with. Using the adapter design pattern allows us to send and receive data from these external components and do it in an efficient manner.

The façade design pattern will be used for a different API integration – the connection between our server and the maps API that we use to obtain map data. Getting the map from the API and converting it into the proper format to be viewed in the app will require a lot of code, certainly more than the payment integration. This integration will require information about the user’s location, nearby landmarks and areas, as well as all the map metadata that isn’t immediately visible to the user. For this reason, the façade design pattern makes the most sense for this component.

The memento design pattern will be useful to remember the state of the map when the user sees it and preserve it for some amount of time, which is yet to be determined. This amount of time may be dependent on various factors, such as app popularity in that area, how strong the user’s connection to the internet is, how many areas are located nearby, etc. Refreshing the map every second would be very inefficient and resource-heavy, so for this reason, having a way to maintain the state of the map would be very useful, and the memento design pattern helps us achieve that.

Last, the observer design pattern will help us implement two different functionalities – one where users have to report another area as cleaned or not cleaned, and another about the map. The map should trigger an update if there is a change to an area in a user’s immediate vicinity. We do not want users to clean an area that has just been cleaned, so the map ought to update in scenarios where the status of an area in the immediate vicinity was changed. If there isn’t a change nearby, then the map should continue to update on an interval as mentioned in the paragraph above.

The observer design pattern will also be useful so that a user is notified when there is a nearby area that needs to be confirmed for cleanliness, or if a user reported an area and is waiting for feedback from another user. When a second user confirms that the first user’s cleaned area was actually cleaned, the first user should be notified and the system will add the task to the user’s history and add badges/medals to their account. This sequence of events requires a trigger, which is why the observer design pattern will be useful.