ECE 458
Project Report
Lucas Donaldson, Johnny Kumpf, Arthur Schweitzer, Niklas Sjöquist

Design Retrospective:

Looking back on evolution 1, many of our design decisions helped us in evolution 2, although there were some decisions that caused issues. The choice that paid off in the most obvious way was our choice of Django as a web framework. Django's built-in support for authentication, permissions, and object-relational mapping made most of the new requirements much easier to implement. We were able to provide support for Duke Oauth simply by extending Django's authentication system. Django's User model also allowed us to easily separate unprivileged, manager, and admin users. Django's object-relational mapper made it easy to implement logging by extending the save and delete functions of database objects. Furthermore, the Django Rest Framework add-on simplified our implementation of a ReST API.

As mentioned in our first project report, in evolution 1 we defined requests as having one corresponding item. While this may seem counterintuitive to the evolution 2 requirement for a "shopping cart" it actually made this extension relatively simple. In order to implement the shopping cart, we created the abstraction of a "Cart_Request", which is composed of one or more of our original Requests. In this way we were able to easily add support for the shopping cart on top of our existing implementation, while also make it easier to add support for partial approval/denial in the future, because admins can address individual Requests within the parent Cart_Request.

One weakness from our evolution 1 decisions was our use of Django's built-in "admin panel". While most of Django's built-ins have been helpful, this particular interface turned out to be too restrictive for our needs. As a result, we created a new "admin" app within our Django project to facilitate all of the admin-specific functionality. In addition, we created a "manager" app for manager-specific workflows, which allowed us to separate out different views and controllers into a more intuitive project structure.

We also rectified our lack of a ReST API from evolution 1, albeit mostly due to the necessity of the evolution 2 requirement. As a result, our project is more accessible to developers and is therefore a more industry-standard system.

Our decisions to use PostgreSQL and Apache for database management and deployment, respectively, seem to have worked out well. Nothing from evolution 2 was particularly complicated due to these choices.

In evolution 1, we saw our UI as a major weakness. While we have definitely improved our front-end by using more bootstrap components and spending more time adjusting the appearance of our website, we still focused more on functionality than aesthetics in evolution 2. We were able to fix the major style bugs where entire css files were missing from our production deployment. The issue was simply a typo in the name of the folder to look for these staticfiles.

Logistical Retrospective:

Evolution 1 served as a strong wakeup call to get our logistical strategy in order. We saw firsthand the inefficiency of our work, and knew that we needed to add more structure and more

face time with the group if we wanted to have a successful Evolution 2. Our main mistakes consisted of scheduling granularity, not enough face-to-face meetings with the group/pairs, and not planning enough time for testing.

Looking back at the previous evolution, we can see that our schedule was nearly unusable. This is mainly due to the tasks being too coarsely defined; instead, we should have split up our tasks into smaller chunks, and assign people to each little chunk. Additionally, tasks like “learn Django” were far too broad and unspecific to be usefully planned; the lack of a distinct plan for learning the framework led to lots of inefficiently used time. Another logistical error we made was not schedule enough group meetings. We rarely met face to face to get a chance to discuss our progress and design decisions along the way, which we clearly saw did not work. It is important to meet regularly so that everyone stays on track and on the same page. If group meetings are hard to schedule, it could also be beneficial to meet up in pairs and program that way. Finally, we gave ourselves little to no time to test because we had to spend much of the time leading up to the deadline implementing additional features for the requirement. This definitely hurt us in the end, as there were multiple bugs that could have been found from the simplest of user testing.

We improved upon pretty much all of our logistical weaknesses as we began ev2. Not only did we meet face to face more frequently, our first meeting was dedicated to pragmatic planning. We set out realistic, specific goals for the 2 weeks we had left for ev2, goals people felt they could meet and had a clear path forward to meeting, unlike the nebulous “learn Django” type goals of ev1. We explain how the plan affected our ev2 work in the Logistical Evaluation section below.

Design Strengths:

The decision to preserve our Request model, and add a new Cart_Request model on top of it proved very useful. With this design, it will be easy to implement the loaning of individual items, by altering the status of sub-requests (Request) attached to a Cart_Request.

Splitting the project into 3 apps, home, manager, and administrator, was useful for readability and documentation purposes, but given the ev3 requirements we wouldn’t have gotten punished very much for leaving the project as one huge monolith. Still, the manager/views.py file and the home/views.py file are around 500 lines of code each, which is already large, so having one views.py file in one app with 1000+ is certainly not ideal. Even just for the sake of a person orienting himself with the code, I believe the app refactoring was a good idea.

The “table of contents” aspect of our Custom Fields implementation is strong. By having the CustomFieldEntry table to tell the program which fields exist, we avoid having to create more than 1 new database entry when a new field is created; we can simply add one key value pair to the CustomFieldEntry table. Then, we query that table whenever we want to display custom fields. This means, with regards to the actual value of the custom field for a certain item, say, “location” for “item5”, we don’t have to load in a blank value before the user enters one; our program understands that if no data exists, the field value for that item hasn’t been specified.

Probably of smaller significance, we created a few helper methods in this evolution to pull large quantities of code out of our view methods. This “programming in the small” technique allowed for a much more readable code base. For example, taking the create_item code out of the view which creates an item, and into a helper method makes both the outer view and the create_item code itself much easier to debug. Little things like this throughout the project should make it easier to add future requirements for which we need to touch old code.

Design Weaknesses:

While the CustomFieldEntry “table of contents” constitutes a strength of the custom field design, the actual storing of the custom field data constitutes a weakness. By creating four separate tables for each of the field types, we locked ourselves into having to create a 4 branch if-else tree everywhere we want to read, create, or modify a custom field. This is messy, and difficult to extend. If we were to add a new kind of field, we would have to touch many places in the code to make it work with our current design.

Another current design weakness is how we decided to split up permissions. Since evolution 2 only required 3 unique privileges (user, manager, admin) we were able to accomplish this by essentially using Django’s built-in ‘superuser’ and ‘staff’ types in order to distinguish between users. However, in the case that we needed to add another level of permissions, this might prove to be slightly more difficult, given our current design.

As mentioned above, we refactored our project into three apps, which reduced the size of our biggest files. However, our views files are still very large and difficult to read through. They would also surely be difficult for a new developer to orient themselves to. This is a case where our “programming in the large” is well-designed with a reasonable division of major project areas, but our “programming in the small” is still relatively poor. In the next evolution, we can help solve this deficiency by continuing to refactor our project. One possible solution would be to create utility methods for common behaviors. However, in many cases there are subtle differences even between similar lines of code that might make this option difficult or even impossible. Another option would be to simply extract methods into more narrow files. Instead of one manager/views.py, we could have multiple views files in our manager app. One such file could be specifically for adding/creating items, another could be for adding/modifying users, and another could be for disbursements/servicing requests.

Furthermore, our front-end design, which relies on the usage of Django templates (many of which includes redundancies), could be improved, since the usage of templates in order to generate HTML scripts is somewhat restrictive. On the other hand, if we utilized a built-in library like React JS, we would most likely have slightly more flexibility and robustness in our user interface, since React’s components are highly reusable and quite easily to implement.

Logistical Evaluation:

Our schedule worked well. Most of the components were finished on time, and we were able to quickly decipher and fix the places where we were behind schedule. By having the schedule laid out in detail, we were able to reassign labor according to various delays, and always still feel confident we were on a path to finish in time. Ultimately, we ended up losing 1 day of testing, ending up only having 1 day instead of the 2 we’d planned, but that was a

necessary choice based on timing. The schedule we made kept us on track early and allowed us to adapt late, so we'll be making a similar one for evolution 3, with the goal of making enough progress early on that we have several days left for testing at the end.

Contributions:

Johnny - Servicing Cart_Requests, making Cart_Requests (pair programmed with Lucas), All the permission-protected stuff (making/deleting/modifying items, tag manipulation, cf manipulation), Custom Fields (adding/modifying/deleting/displaying)

Arthur - User testing for ev2 requirements (pair programmed with Niklas on his computer), Supposed to work on Cart_Request view and servicing of requests... (however, system crashed and project became unworkable on my system after some updates during the last week…. Tried trashing and repopulating the DB, but all migrations would break before completing due to internal issue and even deleting and recreating the project didn't work, so progress was halted for last few meetings and was forced to work in pairs on other tasks… NEED TA help to get my system back to life)

Niklas - API calls/views (some pair programming with Lucas); improved UI of the item detail/list views, especially the addition of custom fields to the details; ran user tests of all the basic requirements, in addition to further testing on the API and Item views

Lucas - API backend, debugger(some pair programmed with Niklas), and token authentication, Duke OAuth implementation, Logging backend and frontend, User creation and permission modification, direct disbursement for managers and shopping cart for users(pair programmed with Johnny), API guide