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

Design Retrospective:

Looking back to evolution 1, most of our design choices were beneficial in terms of implementing the evolution 3 requirements. As in evolution 2, we continued to reap the rewards of choosing Django as our framework, as it provided built-in support for sending emails, and its object-relational mapper simplified our implementation of bulk import and loans.

The one design decision from evolution 1 that continues to cause problems is our decision not to implement a REST API from the beginning of the project. Furthermore, our decision in evolution 2 to implement the required API as parallel to the rest of our application has caused us to neglect the API. Our flaw was in thinking of the API as an additional requirement to implement just for evolution 2, rather than as an integral part of the actual function of our application itself. As a result, we failed to update our API with the changing requirements and allowed serious bugs to develop. In hindsight, we should have spent the time to unify our application and API, or at least spent some time testing the API to ensure that it met the new requirements.

In evolution 2, we made what turned out to be one of our most important design decisions: we preserved our Request model and created a new top level Cart_Request model. The preservation of Request (now a subrequest of what the requirements consider a request) allowed us to touch each of the parts of a user's request even after the request had been made. As a result, in ev3 a "loan" constitutes a simple extension of the current Request model; instead of having three statuses (approved, denied, outstanding), we now have 5 statuses (approved for disbursement, denied, outstanding, approved for loan, returned). This allowed all the existing code, which already worked with Request instances, to be easily extended to handle the two new statuses and thus handle loans.

The only design decision from evolution 2 that may have hurt us was the design of our custom fields. We implemented custom fields by adding several database tables, one as a "table of contents" of the available custom fields and their data types, and individual tables for each type that store the values for given items. This design proved difficult to work with, particularly for the bulk import requirement. Perhaps it might have been better to just add one table for all custom fields and do data type validation in the backend instead of in the database itself.

Logistical Retrospective:

We made a lot of logistical improvements in evolution 2, and carried over our positive practices into evolution 3. Early in both evolutions, we made a plan with fine grained dates leading up to the due date, and then checked in at each of those checkpoints to determine whether we were ahead of, on, or behind schedule. We had 3 days for testing at the end, as we'd planned, so not only was the plan effective in guiding us along the way, but we were also able to better estimate the amount of time things actually would take during this evolution. We have also progressively improved our division of labor in each evolution, making it easier for

each individual team member to understand what is required of them and to finish their work on time.

Design Strengths:

Having subrequests allowed us to easily implement loaning, and will also allow us to easily implement backfill in ev4. Further, the loan handling code is already very extensible to any new kind of status we could add to Request; it only took about an hour to implement the basic backfill workflow by adapting the current code. That part of the manager workflow is pretty decently refactored out of the actual Django view methods and into what might be called “actual logic methods”, which makes it easy and quick to write multiple Django views which use the same logic.

We also feel that our Items are well-designed, as we don’t anticipate much difficulty implementing assets. In order to do so, we intend to simply extend our Item model so that Assets can refer to a parent item, much in the same way that our Requests refer to a parent Cart_Request. By extending Item to make our new Asset classes, we will be able to reuse the same code for both Items and Assets in many cases. This will reduce the amount of new code we have to write in the next evolution and will reduce the complexity of our project as a whole.

Design Weaknesses:

Our API is still a major weakness. During ev3, very little time was spent on the API, and the API code still runs parallel to the rest of the project (instead of meeting at a common point). We also barely tested the API. Due to all these factors, it constitutes a major weakness in the project. If we had paid off the technical debt to merge our application logic with our API logic, then this would not be an issue. Unfortunately, at this stage in the project, the technical debt that we have accrued in this regard is quite large and would be very difficult to pay off. In any case, we plan to make progress in this area by spending more time working on our API and refactoring it to call shared methods with the application code. In this way we can make progress towards a unified application and API even if we don’t eliminate all parallel code paths.

As mentioned in the Design Retrospective, our custom field design is also somewhat weak. This hurt us a bit in evolution 3 due to the bulk import requirement, and may also hurt in evolution 4 when we implement per-asset custom fields. However, we feel that the custom field requirement is one that demands more complexity than most other requirements, and therefore cannot be easily simplified. Every other design solution that we have considered would be at least as complex. The best alternative we thought of was to unify our per-datatype tables to simplify our database models, but that would just be a tradeoff of database complexity vs. backend validation complexity.

Logistical Evaluation:

As touched on in the logistical retrospective section, we were able to follow our schedule pretty closely during evolution 3. We scheduled no work to get done over spring break, in an effort to try to be realistic, and this was a good decision. We were able to create dates we could actually stick to, and by doing so allowed the project to move along smoothly until the end, when we had the desired time for testing.

The testing itself is still relatively weak though. Although we had 2-3 full days for testing, it wasn't really clear how to spend that time most efficiently. We have no automated testing of any kind, so we go through the requirements by hand and see if the required workflow happens error free with expected result on our project, but this doesn't feel very efficient. Still, for the main UI and workflow this allowed us to eliminate all the bugs by demo time. As mentioned earlier, though, the API, which was barely tested, barely worked. Looking ahead, we plan to test our API more extensively at the end of the next evolution in order to catch the kinds of bugs that we missed last time.

Contributions:

Johnny

- Loans (full stack)

Arthur

- Backups
- Backup Guide

Niklas

- Bulk Import (full stack):
    - UI
    - Validity Logic
    - Guide

Lucas

- Emails (full stack)