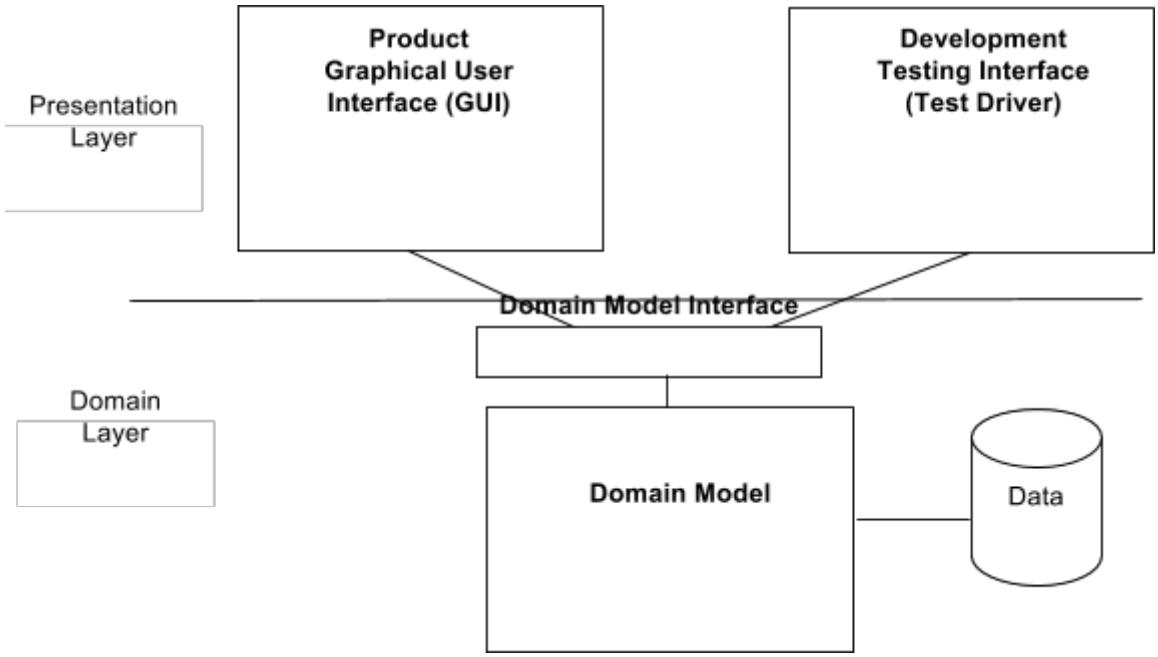
Product Design

Team Group 1 - To The Moon!

Architectural Model

This diagram represents the major subsystems of the product. Initially focus on the domain layer and its components before decomposing the user interface component. Note that a common interface allows both the GUI and a Command Line Interface to access the domain model in the same manner without regard to the type of presentation technique.



Components and Functions

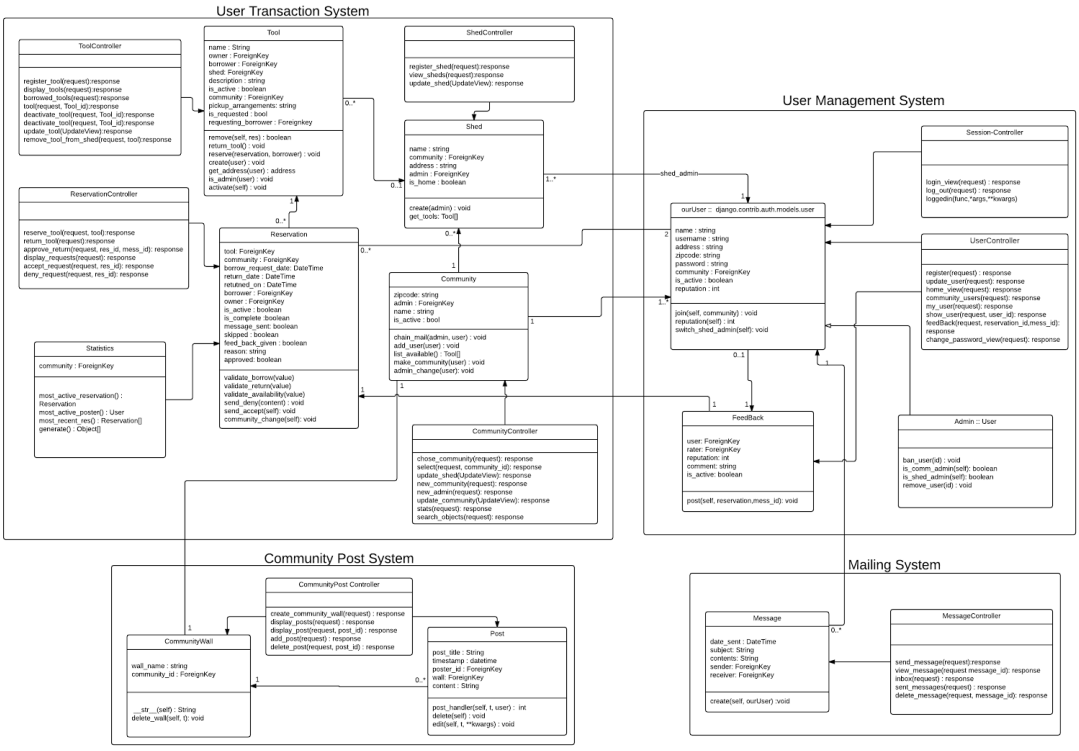
Messaging System	Component state ● This system is responsible for the storage of private messages ● It holds the status of each user's unique inbox; users don't share inboxes ● The system is responsible for queuing messages to be sent Component behavior ● This system is able to send messages between users ● This system is able to send messages to user ● Confirming for user that a message sent ● It is responsible for rendering the inbox page, read message page, and the send message page
Community Post System	Component state ● This system is responsible for the storage of community messages ● It holds the community messages for each community's unique wall ● The system is responsible for queuing posts to be posted Component behavior ● This system allows users post/remove their own public messages to the community wall ● This system allows users to view messages on a community wall ● It is responsible for rendering the community wall page, submit a post page, and a view post page
User Transaction System	Component state ● Holds the information needed to make transactions between users ○ State of a community (a collection of users) ■ Holds name, admin, zipcode ○ State of a tool ■ Holds name, owner, description, pickup arrangements, if it's available ○ State of a shed (a collection of tools in a specific part of a community) ■ Holds name, community it's in, address, admin, what type of shed it is ○ State of a reservation ■ Hold tool that's being reserved, the community it's in, the borrow request and return dates, the owner and borrower, the reason for the reservation, and state that checks if the reservation has been carried out and completed ● Hold the information needed to analyze transactions ○ Keeps track of all in a community reservations for analysis by the statistics system Component behavior ● System supports reservation action ○ Reservations can not overlap. A tool can only be borrowed by one user in their reservation time ○ Owner can set blackout dates on their own tools so that other users cannot borrow at that time ○ Current and future reservations of tools (borrower request and owner approval). ■ Reservations need owner the approval of borrowing and assurance of return when sharing from owner's home ■ Reservations can happen without owner or shed admin approval when in a community shed.

	● System supports tool actions ○ Registration of a tool ○ Updating tool information ○ Removal of a tool from the system ● System supports shed actions ○ Creation of a shed ○ Adding tools in a shed ○ Removing tools from a shed if user owns tools or is the shed admin ○ Sharing a tool from home ● System supports community actions ○ Creation of a community ○ Updating of community information ○ Removal of a community from the system ○ Aggregating stats on a community ○ Searching function for tools and users in community ○ Sorting of tool information and users in tables alphabetically ● System supports admin actions ○ Banning or removing users ○ Promoting users ○ Making communities ○ Shed admins can manage tools in shed ● System is responsible for rendering all needed pages for the previously stated behaviors in this system
User Management System	Component state ● Hold reputation status (an aggregate of reservation ratings applied to a user) ● Holds user information ○ Extends built in django User class, contains all state in that class ■ Password and username ○ User information specific to ToolShare ■ User's address, birth date, zip code stored Component behavior ● System supports new user creation ○ Responsible for registering a new user ■ Makes sure new user is older than 16 to allow them to register ● System supports user and reservation action ○ Joining a community ○ Editing user's account info (zipcode, password, and address) ○ Adding/removing a tool to share ○ Making sheds ○ Sharing tools (through the reservation system) ○ Rating a reservation interaction ○ Admin functions when applicable (managing sheds, managing communities, banning users) ○ Posting on the community board (through post system) ● System is responsible for rendering all needed pages for the previously stated behaviors in this system

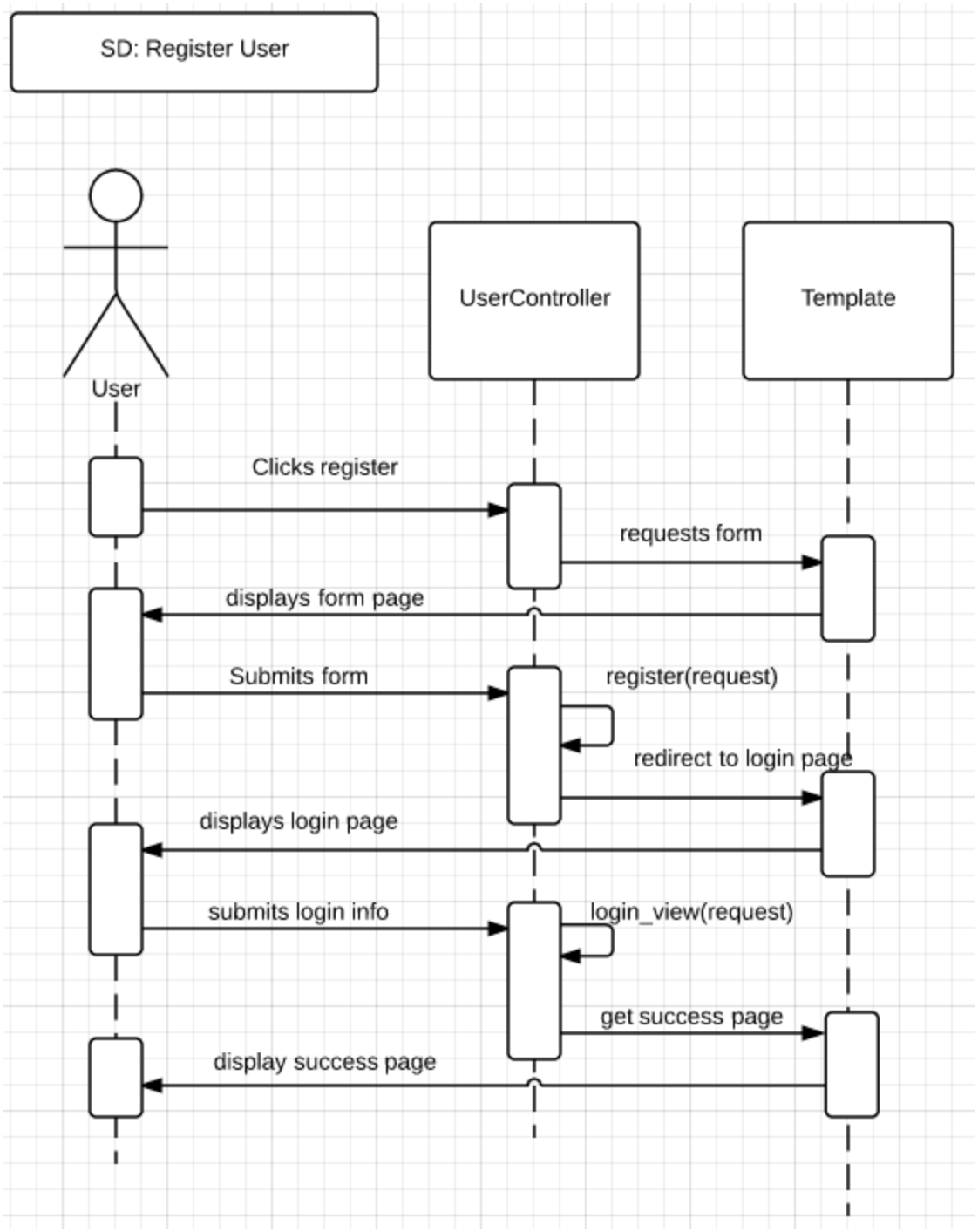
Statistics System	Component state ● Hold the community on which statistics can be aggregated ● Is responsible for only generating statistics on a community Component behavior ● Subsystem supports queries on communities ○ Can display most active borrower ○ Can display most active poster ○ Can display most resent reservations
Authentication System	Component state ● This subsystem is responsible for authentication for each time the system is used ○ Responsible for user login ○ Responsible for keeping track of who is logged in so a user can only access information allowed by their privileges ○ Responsible for user logout ○ For updating user information Component behavior ● System supports user login ● Allows changing password (make sure it passes system requirements)

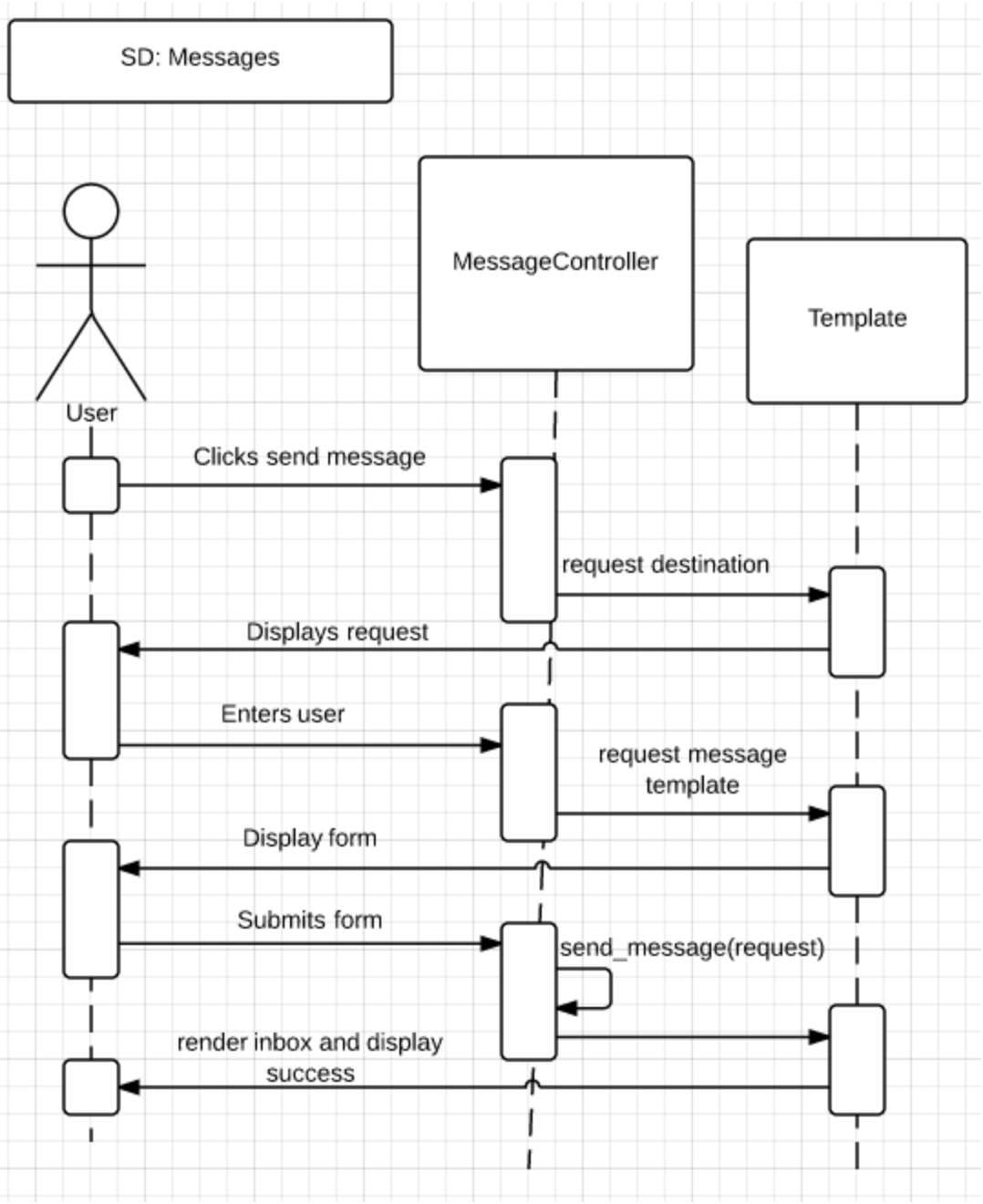
Class Diagram

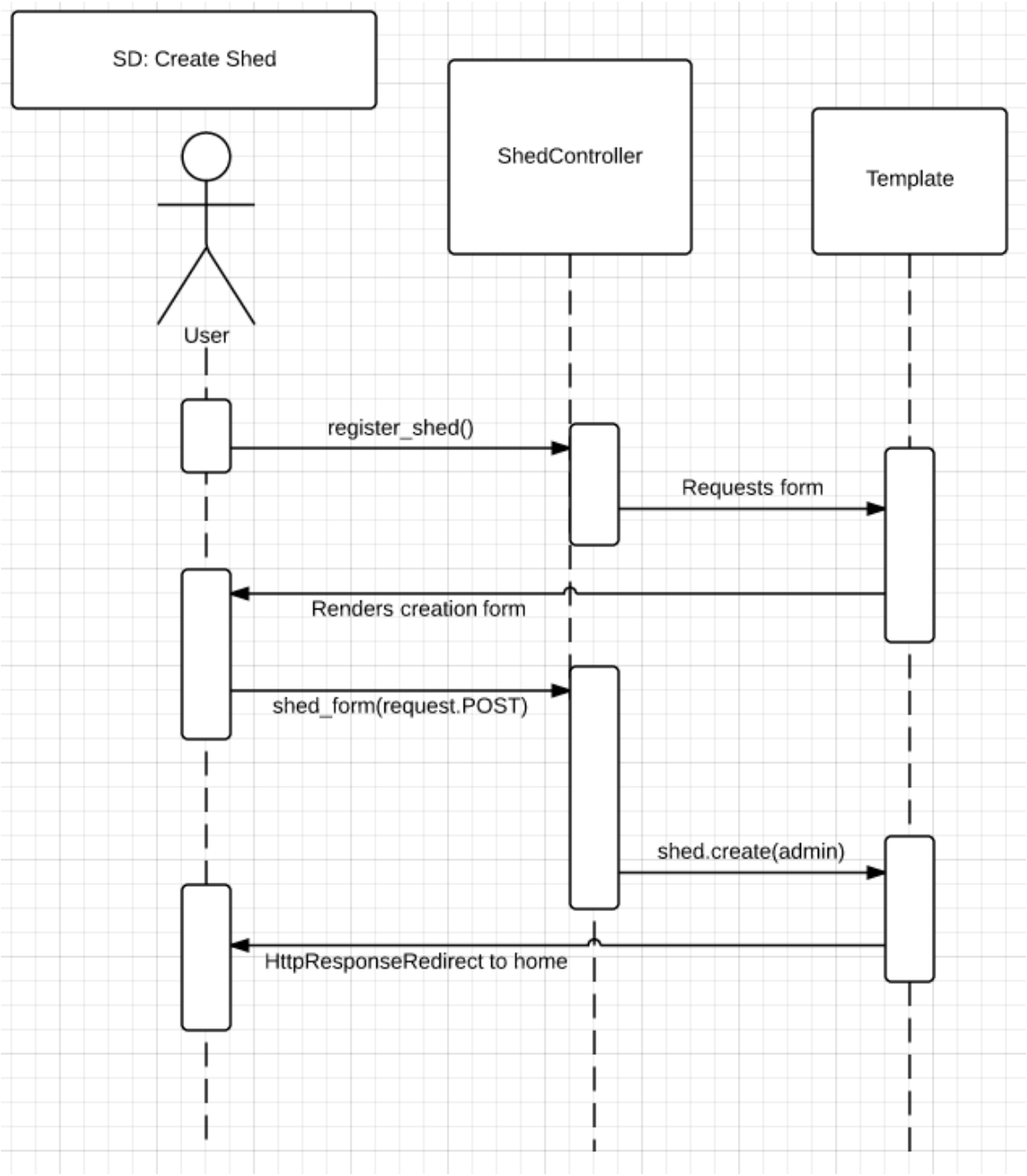
see attached “ToolShare Class Diagram.pdf” for enhanced viewing

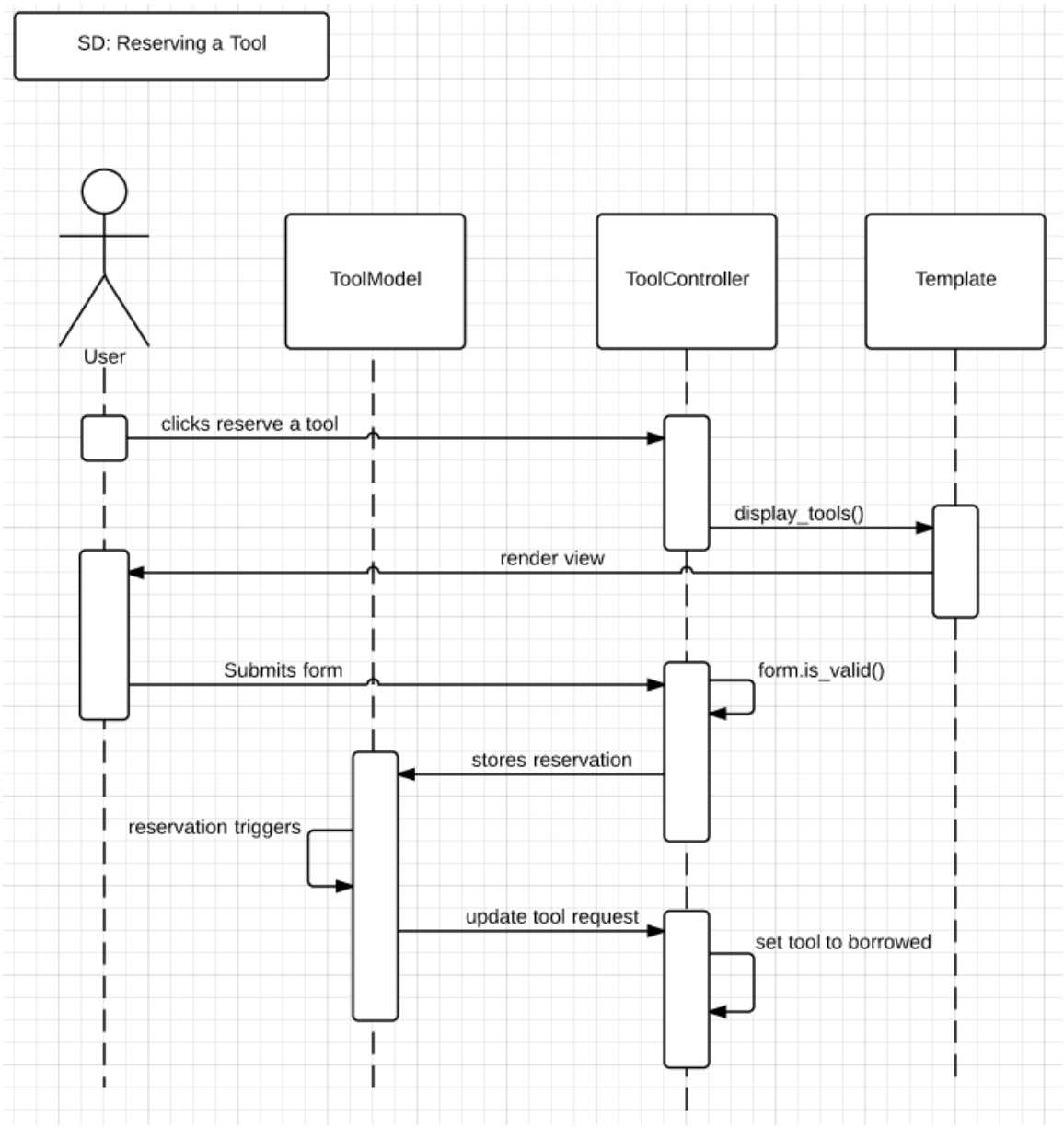


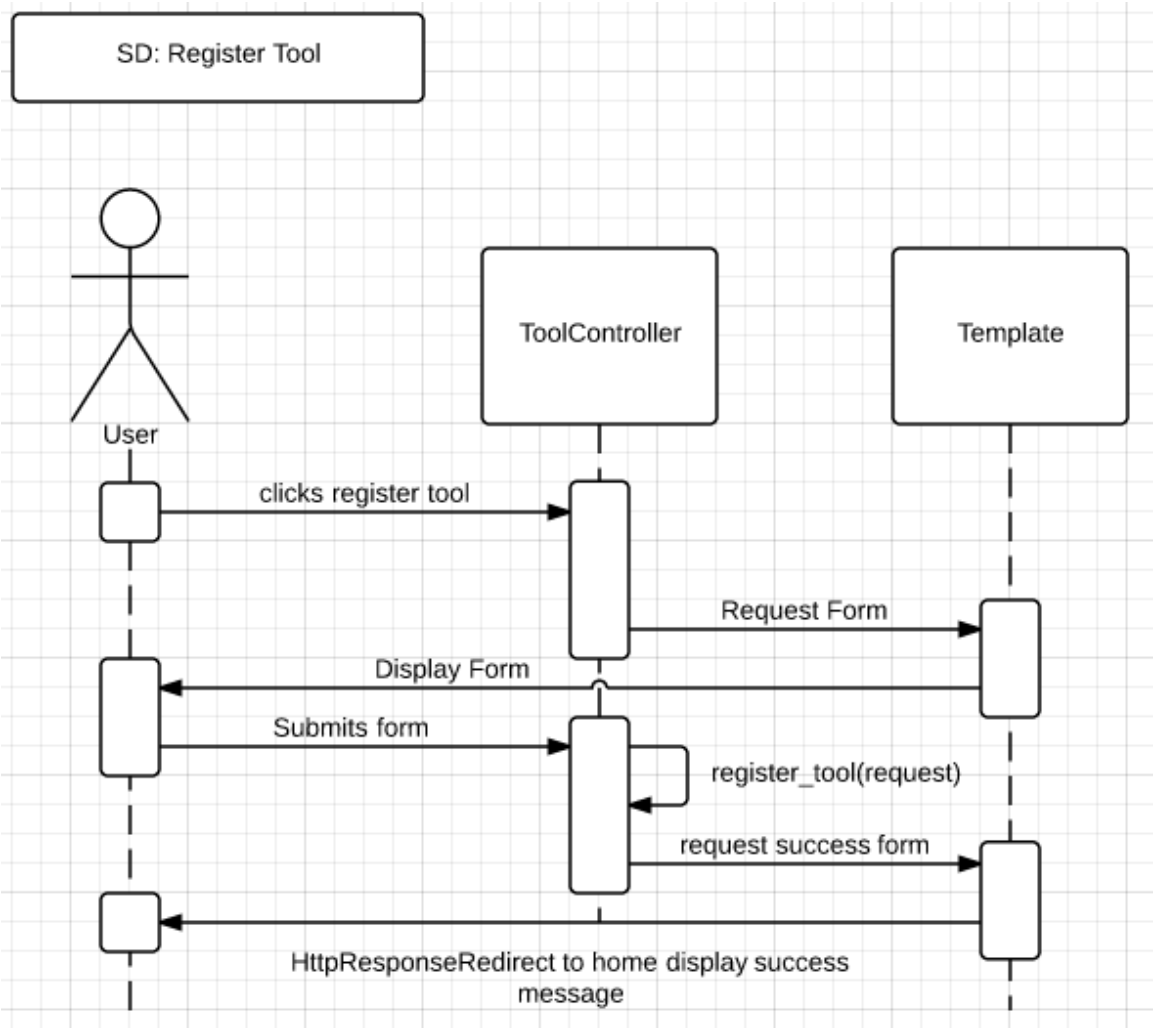
Sequence Diagrams




SD: Messages
MessageController
Template
User
Clicks send message
request destination
Displays request
Enters user
request message template
Display form
Submits form
send_message(request)
render inbox and display success


SD: Create Shed
ShedController
Template
User
register_shed()
Requests form
Renders creation form
shed_form(request.POST)
shed.create(admin)
HttpResponseRedirect to home


SD: Reserving a Tool
User
ToolModel
ToolController
Template
clicks reserve a tool
display_tools()
render view
Submits form
form.is_valid()
stores reservation
reservation triggers
update tool request
set tool to borrowed


SD: Register Tool
User
ToolController
Template
clicks register tool
Request Form
Display Form
Submits form
register_tool(request)
request success form
HttpResponseRedirect to home display success message

Design Rationale

Rationale During R1

Our first design for Toolshare was not logical or efficient in terms of how the database and the system as a whole worked together. The main idea of the design was as follows:

- We had models that we connected through one-to-many, one-to-one, and many-to many relationships.
- We planned to use Django's built in user model.

With our first design we had the following faults:

- We wanted to hold the the tool sharing logic (what is where, who is the owner, who is the lender) to be held by both the user and the tool itself.
    - This was bad design, as the coupling between the information for tools and information for users was tight. It required a lot of coordination and a higher information flow to stay accurate.
- We wanted the community post system and the mailing system to be connected
    - This was because we thought the logic for both was just the sending of messages. Our plan was to have post and message be one message class that both the community wall and the mailbox could use.
        - This was not a good way to build our system because the information we needed for posts and messages were different.
        - This also made the system design connect two separate components in a way that was needless. We decided it would be more work to connect the logic of both than it would be to separate the components.
- We ignored controllers when designing the system
    - This is bad because it caused us to have lots of different/unnecessary connections between models.
    - We determined we had unnecessary methods distributed throughout the models that would be handled by controllers instead.

The next iteration of our design loosely incorporated the controllers and also removed the connection between the Community Post System and Mail System. We also added a SharingLog model to keep track of the sharing interactions. We added the SharingLog to reduce the coupling between users and tools. This allowed us to think a little more about how sheds, tools, and users interacted and influenced later design.

- The incorporation of a SharingLog allowed for a Stats Controller to exist. A SharingLog allowed for a controller to look at that log and aggregate information easily
    - A large issue we didn't consider with making the SharingLog was the validity of the information inside of it. We assumed that users would return tools on time no matter what. Which, when thinking of the real world, is a huge assumption
- Separating the Mailing system and the Community Post system made sense to us, since even though they both deal with messaging, their purpose is different

The following design iteration added a ToolHolder Class that both the Shed class and the User class would inherit from. We did this because there is a connection between the functionality of a shed and a user. Since tools can be shared from a user or a shed, we wanted to have a superclass to use for identification in our SharingLog. At this time we also separated all of our models and controllers into logical subsystems.

The next time we changed the design was to try to handle the validity of information in the SharingLog. We added a Reservation Class and a Transaction Class in place of the SharingLog.

- The Reservation Class was made so users could try to borrow a tool at any time that the tool was available.
- The Transaction Class was made to keep track of what was actually happening.
    - We separated it like this so we could see if a reservation was not able to be made into a transaction (like if the borrower from the last transaction didn't return a tool).

We changed the design again when we realized that the information between the Reservation Class and the Transaction Class was coupled too much. At this time we also removed the ToolHolder Class because of advice from Andy Meneely. In retrospect, the added ToolHolder Class was not needed and caused us extra work.

- We removed the Transaction Class and kept the Reservation Class. This way we can still see reservations and use the information in Reservations to see if a tool was not returned, but we're not, in a sense, duplicating information.
    - If we kept both classes, we would have a lot of duplicate information flow between Reservations and Transactions

Our current design has three main components: a Mailing Subsystem, an User Transaction Subsystem, and a Community Post Subsystem.

- The Community Post System is the same as earlier iterations. It keeps track of community posts and is connected to communities by a community wall.
- The Mailing System is also the same as past iterations. It keeps track of messages between users by a unique inbox for each user
- The User Transaction System was only altered slightly from the last iterations
    - We added an Admin Class that extends User. This allows for admin functionality in our system
    - We also moved our Reputation class from just connecting to the user to connecting to the user through a reservation.
        - The idea behind this was that a lender wouldn't want to go to another borrower's page to rate them, they would want to do it after a transaction.

We landed on our current design because the system was more complex than we thought it was originally. Our original design was only the basic idea of the system and we weren't able to handle some of our requirements with it. We thought that past designs would be easier to code because we were only dealing with a few models. Over time we realized breaking up the system into smaller parts might be more up front work, but it should cause less problems. The system we have now has less coupling that past iterations, which should help reduce the maintenance requirements of the database and the code.

Rationale During R2

Our current design has 4 main systems: a User Transaction System, a User Management System, a Mailing System, and a Community Post System. The functionality of the system is very similar; the biggest difference is added features (admin abilities, reputation with feedback form, searching functionality). We separated out the User Management System from the User Transaction System because keeping them together would have created fairly unmanagable files.

When we came to the reputation section of the project we made a decision to tie reputation not only to reservations with a one to one field but also to tie it directly to the user. This allows for easier

access to reputation because getting the user object in any instance is really easy. To calculate the reputation we added a function on user that would get all the feedback objects that are active and are attached to that user. It then averages them together and return the number value. This function can be called from templates which makes it really easy to display to the user.

To make sure communities for formed well we made it so when a new user creates an account we have it so they are able to pick an existing community or they can create a new community. If they create a new one they become the admin of that community so that they can manage all the other users that may join their community.

Once a user is in a community they have the option to change communities. This cancels all future reservation made by them or for any of their tools. If that user is the admin of the community the admin is given to the user in the community with the high reputation. We did this to save time and complexity. If the user is an admin of a shed the the community admin will become the admin of there shed. You are given the option to either join an existing community or make a new one. If you are the last person in the community when you leave the community closes and no one is able to join that community any more.

While we were developing reservation and the ability for a user to make future reservation we ran into the problem of the system will know when a reservation should be activated. So we decided that we should make a background worker that just keeps track of what happened with reservation. It has the job of activated reservation when it is their turn and it has the job of notifying users when there reservation has expired for a certain tool. This background thread simplified future reservations for us and allowed us to create the concept of what happens if someone doesn't return a tool on time and a reservation gets skipped.

While we were developing we noticed that if you weren't logged in you could still go to a lot of pages on the website and potentially break things. To fix this we adopted the decorator design pattern and wrote a decorate that checks if a user is logged in before allowing them to navigate to any page. If they are not logged in they are redirected to the login page.