1. Name: Erin Ruby

2. Description: A membership service for members of a gym to schedule personal training sessions with trainers. There are three types of sessions that a member can schedule. Members use punch passes to buy sessions, and can buy more to add to their account.

3. Features that were implemented:

ID	Actor	Requirements
01a	Trainer	A new trainer can create an account so they can view the schedule, and cancel sessions.
01b	Trainer	A trainer can log into their account.
02a	Member	A new member can create an account so they can buy punch passes, view the schedule, and book sessions with a trainer.
02b	Member	A member can log into their account.
01c	Trainer	View session schedule. An output of each session that has been booked by a member.
01e	Trainer	A trainer can can cancel a session and remove it from the schedule.
02c	Member	A member can view how many punch passes they have on their account.
02d	Member	A member can buy punch passes that are used to book a session.
02e	Member	View session schedule. An output of each session that has been booked by a member.
02f	Member	Members can book sessions, with a type, trainer, day and time.
01f	Trainer	A trainer can log out of their account.
02g	Member	A member can log out of their account.

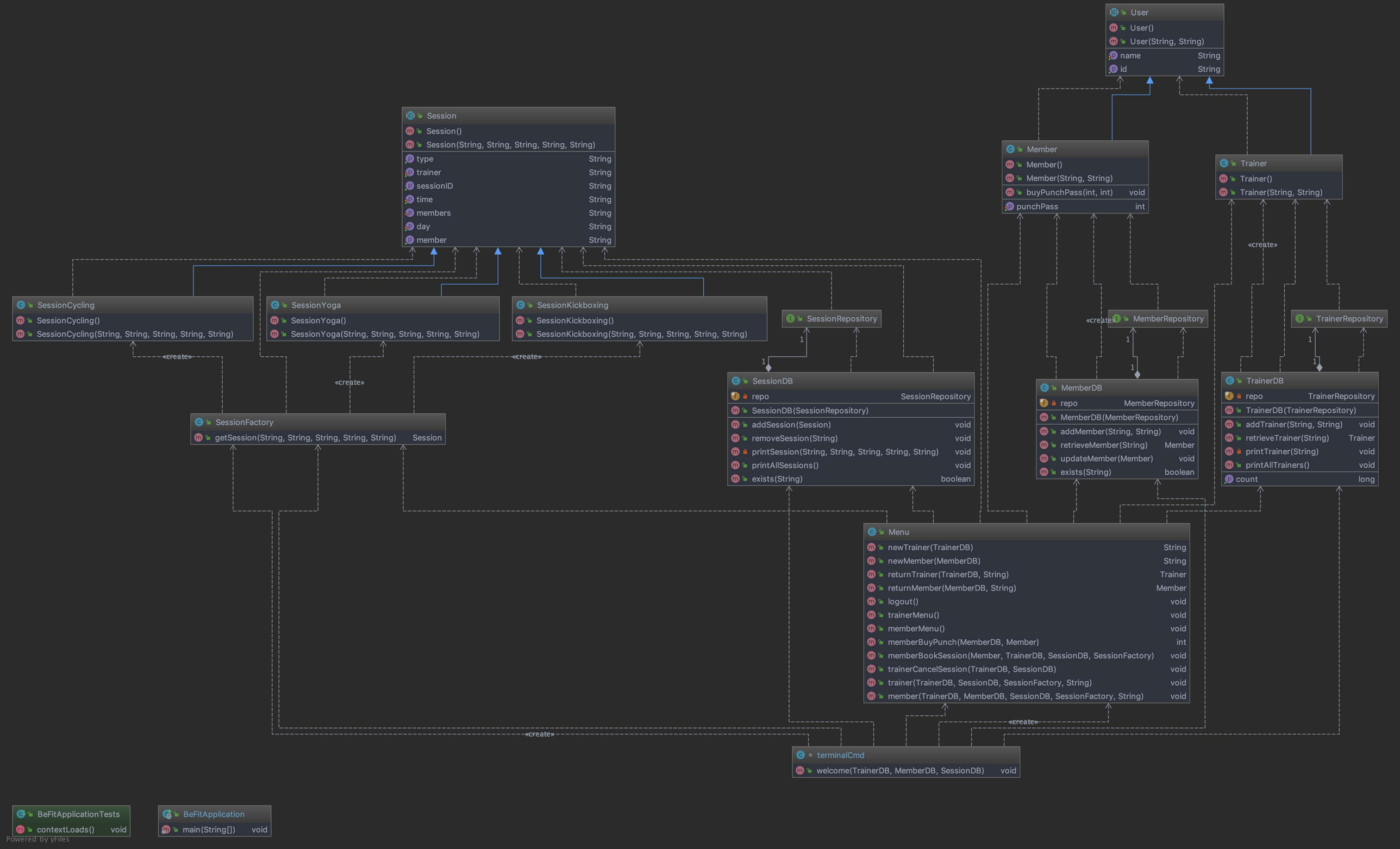
4. Features that were not implemented:

ID	Actor	Requirements
01d	Trainer	Edit a session. A trainer that wants to change the date, time or trainer for a session can update it.

5. Final Class Diagram

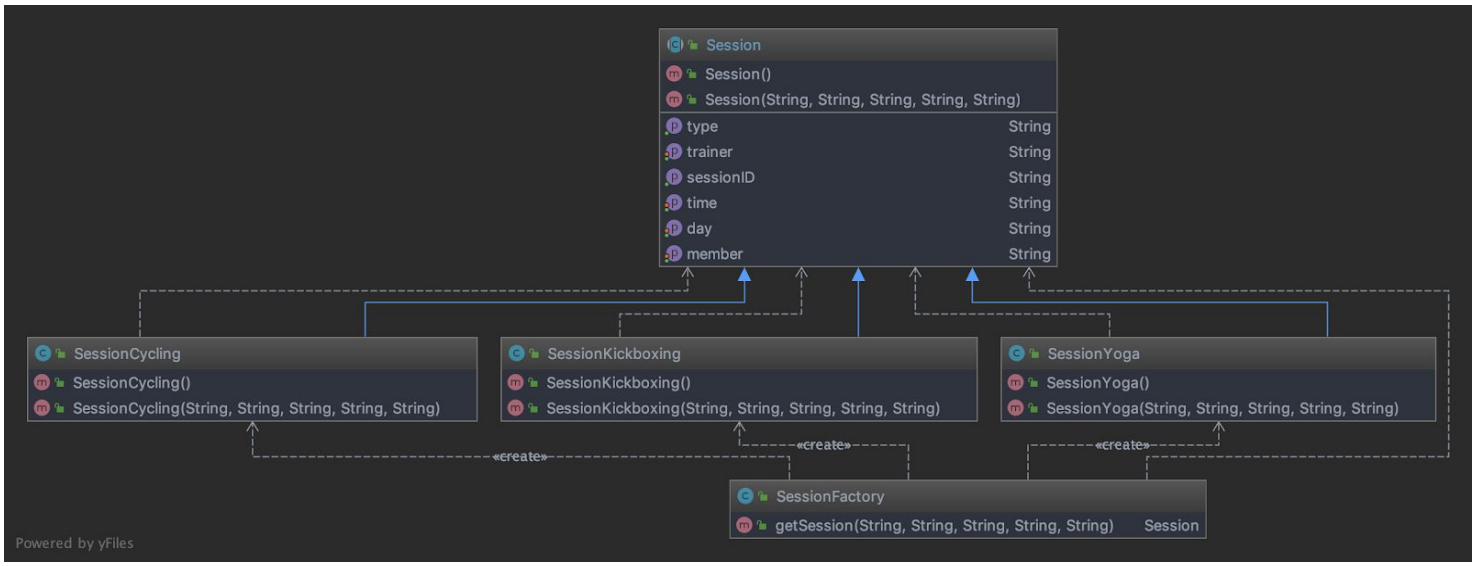
https://raw.githubusercontent.com/erru4017/4448_project/master/Ruby_BeFit_ClassDiagram.png

My class diagram changed drastically from my first one. I thought I would only need 5 classes, and I ended up creating 13 classes and 3 interfaces. I kept the abstract class *User* with *Member* and *Trainer* as subclasses from the first class diagram. If I had a better understanding of what my project was going to be at the beginning, the code would not have taken as long. I pivoted very frequently as the semester progressed. I needed to add a class for each type of session, *SessionYoga, SessionKickboxing, SessionCycling.* I also have an abstract class *Session* because all of the sessions have the same information stored about them. The type is what distinguishes each session from each other. Each collection in the database required a class and interface. I have a collection for trainers, a collection for members and a collection for sessions. Finally, I added a class *Menu* in order to refactor my code and get rid of duplicate code.

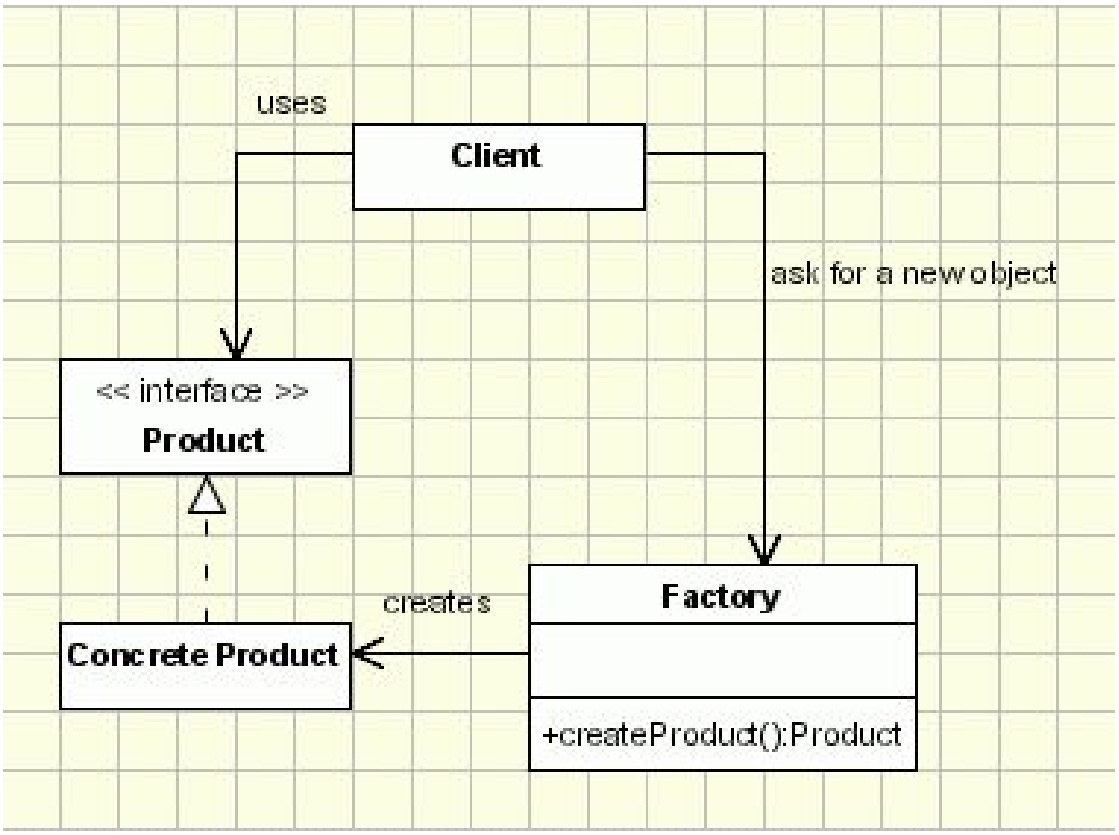

User
User()
User(String, String)
name String
id String
Session
Session()
Session(String, String, String, String, String)
type String
trainer String
sessionID String
time String
members String
day String
member String
Member
Member()
Member(String, String)
buyPunchPass(int, int) void
punchPass int
Trainer
Trainer()
Trainer(String, String)
SessionCycling
SessionCycling()
SessionCycling(String, String, String, String, String)
SessionYoga
SessionYoga()
SessionYoga(String, String, String, String, String)
SessionKickboxing
SessionKickboxing()
SessionKickboxing(String, String, String, String, String)
SessionRepository
MemberRepository
TrainerRepository
«create»
SessionDB
repo SessionRepository
SessionDB(SessionRepository)
addSession(Session) void
removeSession(String) void
printSession(String, String, String, String, String) void
printAllSessions() void
exists(String) boolean
MemberDB
repo MemberRepository
MemberDB(MemberRepository)
addMember(String, String) void
retrieveMember(String) Member
updateMember(Member) void
exists(String) boolean
TrainerDB
repo TrainerRepository
TrainerDB(TrainerRepository)
addTrainer(String, String) void
retrieveTrainer(String) Trainer
printTrainer(String) void
printAllTrainers() void
count long
SessionFactory
getSession(String, String, String, String, String) Session
Menu
newTrainer(TrainerDB) String
newMember(MemberDB) String
returnTrainer(TrainerDB, String) Trainer
returnMember(MemberDB, String) Member
logout() void
trainerMenu() void
memberMenu() void
memberBuyPunch(MemberDB, Member) int
memberBookSession(Member, TrainerDB, SessionDB, SessionFactory) void
trainerCancelSession(TrainerDB, SessionDB) void
trainer(TrainerDB, SessionDB, SessionFactory, String) void
member(TrainerDB, MemberDB, SessionDB, SessionFactory, String) void
terminalCmd
welcome(TrainerDB, MemberDB, SessionDB) void
BeFitApplicationTests
contextLoads() void
BeFitApplication
main(String[]) void
Powered by yFiles

6. Design Pattern - Factory:

- My class diagram



- Factory Design diagram



I chose to do the factory design pattern because the session class cannot instantiate the type of session it needs to create until the member specifies which session they want to book. The factory makes it clean and easy to create objects based on the user input.

7. What I have learned:
I have learned the importance of classes and how they make the code more organized. This was also my first time using Java, so I learned how to code in Java this semester. Creating UML diagrams are very helpful in understanding how a system gets implemented. Being able to see the code in a visual way gives us an easier way to design a complex system in a strategic way.