# Note TrackME Class Diagram

Non ci sono problemi di sincronizzazione dal momento che non c’è modifica concorrente degli stessi oggetti sui due tier: in particolare, non è possibile che un oggetto si trovi contemporaneamente su due tier diversi.

DataReader crea degli oggetti di classe Data e li salva localmente: periodicamente li invia al server. Dentro il server il router li inserisce nell’oggetto di classe User corrispondente, il data collector svuota il buffer della lista nello User e riempie il database ( pattern producer/consumer)- tra i metodi sendAllData and StoreInDB)

# Singleton pattern nel client per Thresholds

Il fatto di inserire I dati nello User e dare la possibilità di metterli nel database solo quando vuole il dataCollector ci permette di semplificare la gestione del database usando un buffer intermedio ( problemi di lock e sincronizzazione al dataabase, accessi concorrenti)

# Abbiamo pensato ad un’architettura fat client, perché evita una costante e dispendiosa comunicazione tra server e client dei dati conmtinuamente elaborati

Il compito di DataReadingService è quello di creare gli oggetti di classeData

CreateData va in loop: ogni volta che riceve dati crea un oggetto come sopra detto, se non lo riceve dopo un timer segnala il malfunzionamento

Abbiamo deciso di stare vaghi in preferences e nel database perché vogliamo lasciare all’implementazione fianle la scelta di quali possibilità di preference possano essere fornite alla third party, in dipendenza dalla facilità di fornirle nel linguaggio di implementazione scelto, da come il mercato risponde alle nostre release, etc.

NewDataSubscriber è observer di User, quando ci sono modifiche e nuovi dati vengono generati, lui fa richieste utilizzando il RequestsService

Bottom up Strategy, partendo dai componenti più essenziali (Emergency, DataCollector, Model)

Unit Testing dei componenti più importanti,

Le parti erano scollegate le une con le altre???

No chiaro match tra integration, implementation and testing: le parti non erano collegate tra loro e non era chiara la strategia effettiva

(che cosa non andava bene nel DD degli altri nel plan for the implementation???)

Ho separato le azioni di richiedere il permesso per gli individual Data e richiedere gli IndividualData stessi una volta avuto il permesso (è una scelta senza nessuno motivo specifico, mi è venuta così)

Dobbiamo dire che non scegliamo come implemetare il datatase se darlo in gestione esterna, perechè questa scelta dipenderà da quanti useranno l’app

Sta il problema del web Server: che cacchio fa? Io direi che converte i datapool di risposta alle groupedRequests in json da inviare al web client

State Pattern per la realizzazione dello Status: se lo stato è attivo, ci si può avvalere di thresholds comparator, altrimenti no

Visitor Pattern for History

Login e SigninService stanno sul Server percchè possano essere condivisi tra User e ThirdParty

Stateful connection: si perché i dati devono poter essere inviati continuamente): da quando ti logghi su di un dispositivo resi loggato per sempre: devi poterti loggare su tutti i dispositiv che vuoi.

DATA ELABORATOR SI OCCUPA ANCHE DI ANONIMIZZARE I DATI

Singleton pattern per emergencyNumber

NewData si mette come observer di tutti gli User, quando i dati vengono creati fa nuove richieste

Quando arrivano le richieste viene chiamato applyRequest, ogni request ha il suo algoritmo incorporato e lo applica chiamando i servizi di Requests Service. Si tratta di uno strategy pattern

DA RIPASSARE: LA DIFFERENZA TRA STRATEGY PATTERN E VISITOR PATTERN

Anche login deve interfacciarsi con il DB ( per i clienti? )

Dobbiamo spiegare come AutomatedSOS “ is built on top of Data4Help”, in che modo abbiamo gestito questa cosa

Potremmo dividere in due subsystems per dividdere correttamente tra Data4Help e AutomatedSOS

Parlare del bottleneck nel deployment diagram