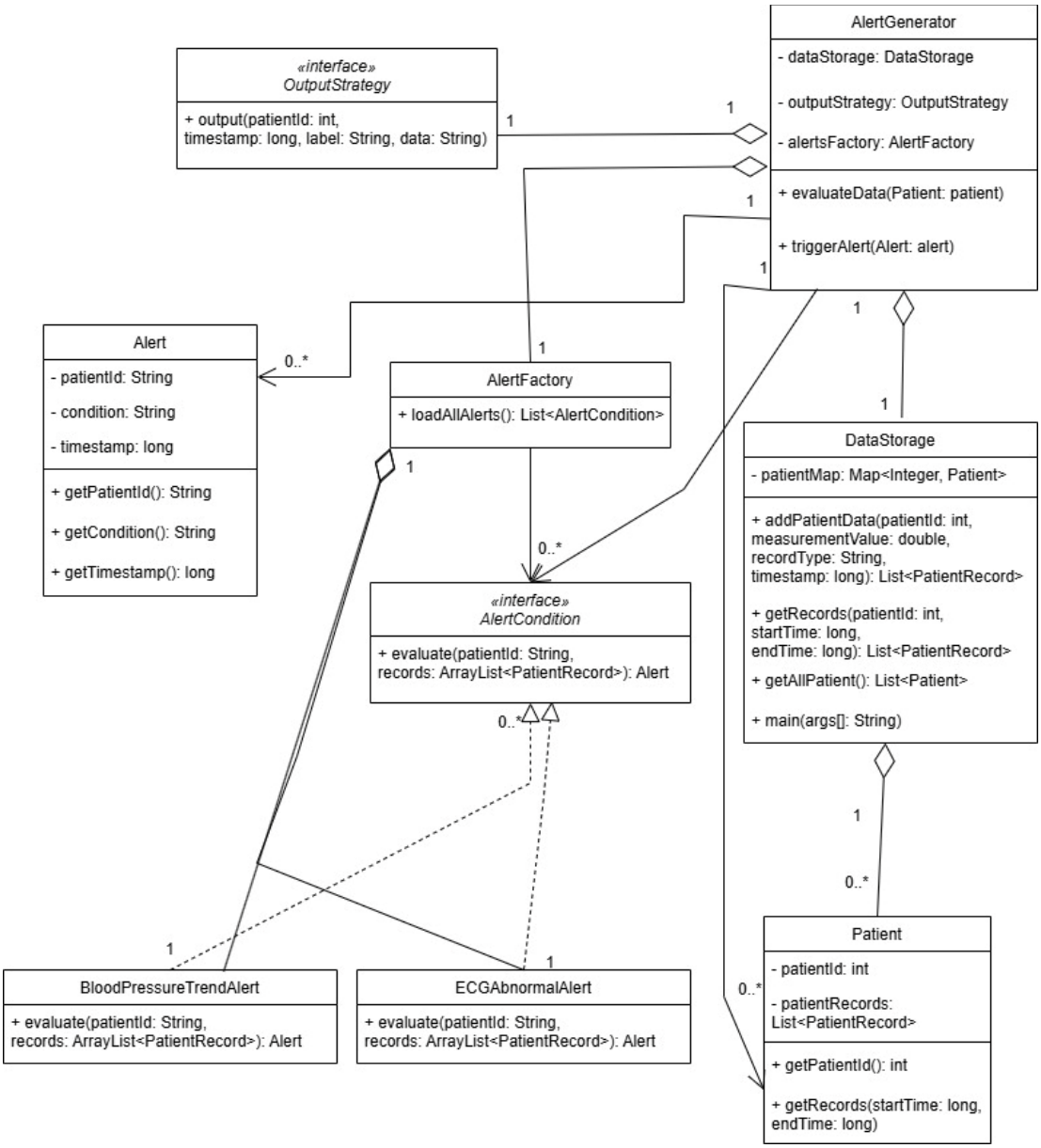
UML Diagrams - Updated

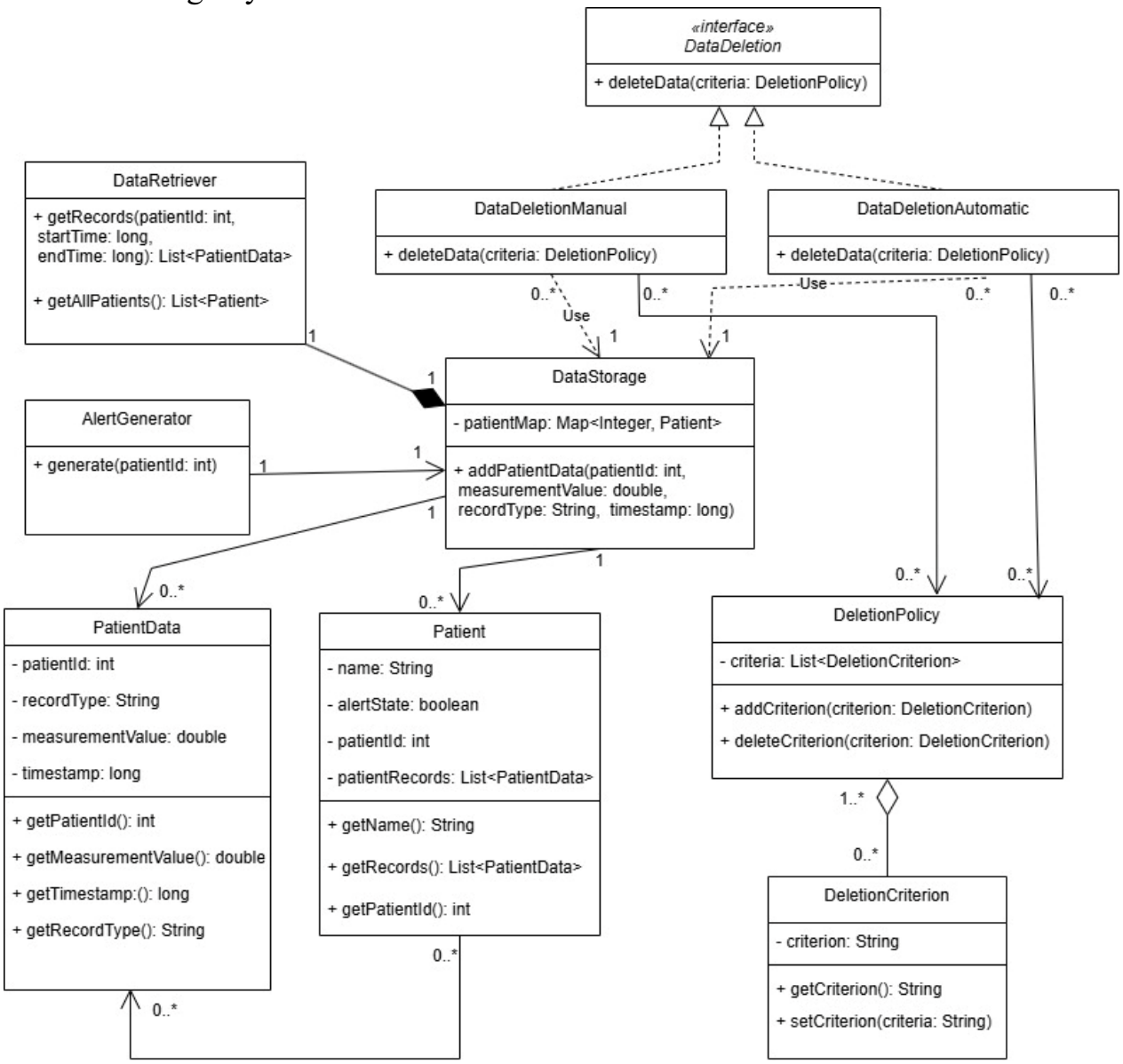
1. Alert Generation System



Information about all patients is stored at **DataStorage**. Each patient is considered as instance of **Patient** class, so each patient has Id number and list of records of data. DataStorage has main function through which it checks all patients for any alerts. There is interface *AlertCondition* which is standard for all alert conditions. It has only one function *evaluate* which return alert if needed or return null (nothing). Then, **AlertFactory** stores all alert conditions to be checked. For this example, only **BloodPressureTrendAlert** and **ECGAbnormalAlert** are shown, but it's very easy to add a new alert (just create new class implementing *AlertCondition* and add instance to **AlertFactory**). Then, AlertGenerator evaluates each patients going through all alert conditions

given by **AlertFactory**. If some alert required, the class sends information about alert condition according to **OutputStrategy** (so, it can go to logs, or to doctors, etc.).

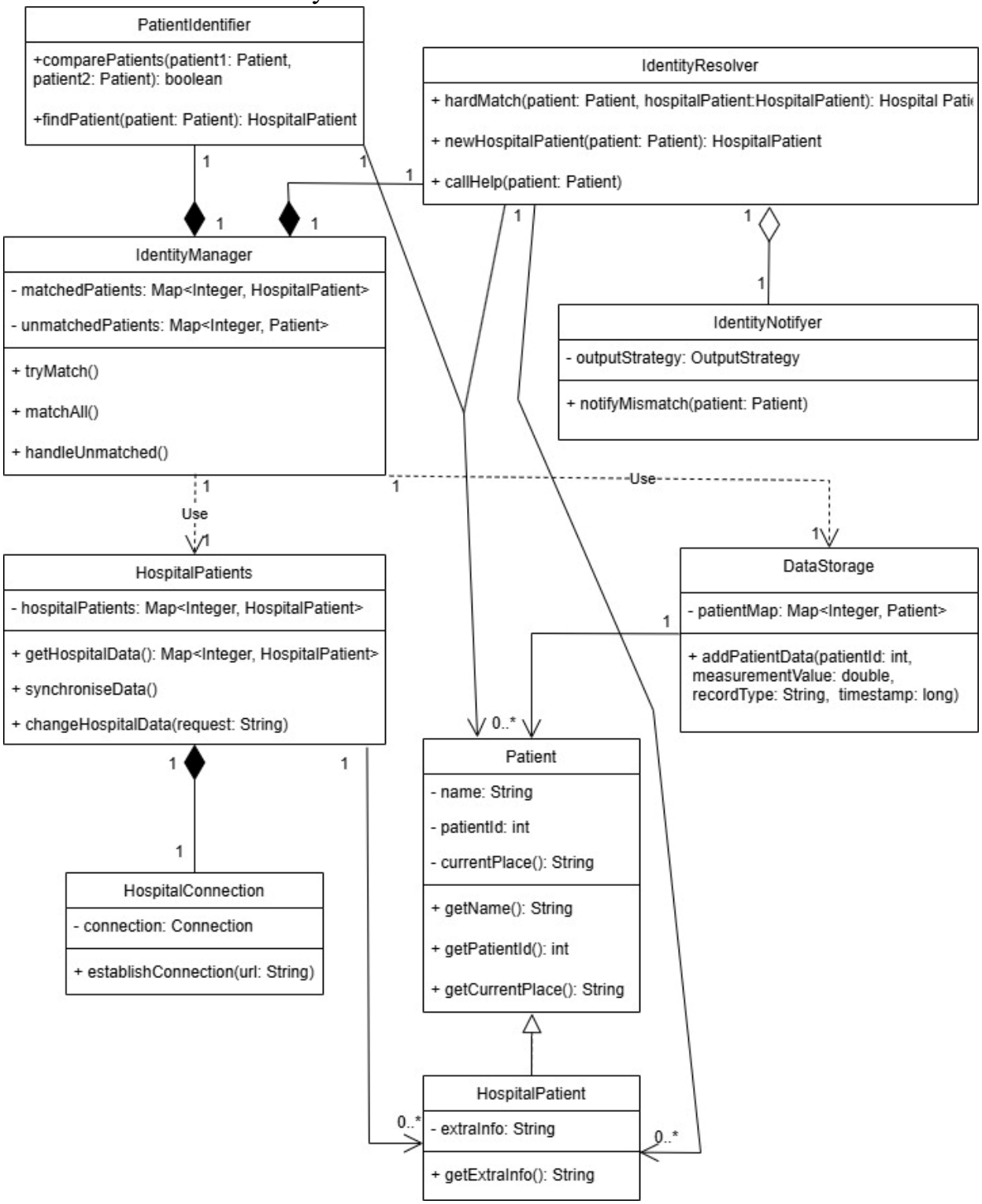
2. Data Storage System



AlertGenerator, discussed in part one, besides getting data and sending alerts, now also sends measurements for storage. It sends them to **DataStorage** class, which is singleton, contains hash table with all patients of the hospital. Using *addPatientData* function, it stores new measurements from **AlertGenerator**, assigned to specific patient. Considering each patient as instance of class, each patient has a list of all his/her records where each record contains *patientId*, *recordType*, *measurementValue*, and *timestamp* (instances of class **PatientData**). To retrieve data, medical staff will refer to **DataRetriever** class, using which it's possible to get list of patients (*getAllPatients*) or list or records of a specific patient (*getRecords*) through **DataStorage** instance. It's important that through this class it's impossible to occasionally delete any information. Now, getting to deletion topic. There is a class **DeletionCriterion**, instances of which can contain a criterion for future deletion. Those

criteria can be stored at instances of **DeletionPolicy** class. Next, there is interface ***DataDeletion*** which is implemented through 2 classes. The first class, **DataDeletionAutomatic**, allows to make criteria which will be automatically regularly checked and to delete data automatically (e.g. delete all data stored later than X days before). The second class, **DataDeletionManual**, allows medical staff to manually delete data in case of special situations (e.g. delete all data on blood pressure for day X because the measure device was broken and always showed wrong results). Hence, I decided that **DeletionPolicy** should not be singleton, so that staff can create different policies for different occasions. We may assume that through GUI, the access to criteria and deletion classes will be restricted to a smaller number of doctors than access to data retriever.

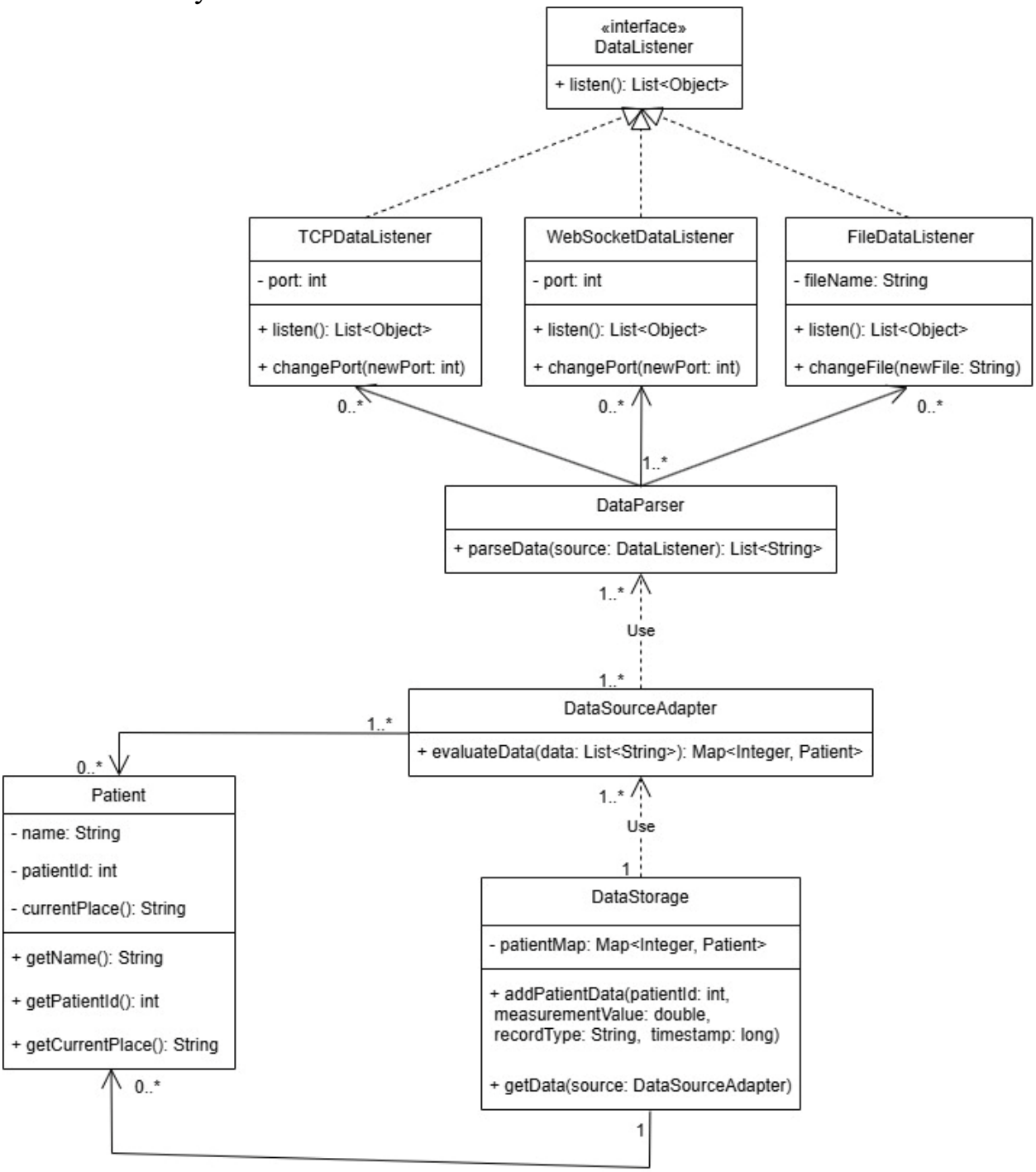
3. Patient Identification System



Using **HospitalConnection** class, we establish connection to a DB of the hospital. We can get data from DB, or change something in that data through **HospitalPatients** class. We save a hash map of all hospital patients. They are stored using **HospitalPatient** class, child of

Patient class. The reason for that is that real hospital patients may have some extra info rather than our simulated data, though they all have *name*, *patientId*, and *currentPlace* (in which room situated or not in hospital). Same as in the previous task, all our simulated data is stored in **DataStorage** class. The main class in this system is **IdentityManager** class. It stores which patients were matched with hospital ones and those who are not. It has *tryMatch* function, using which it tries to find matches with the help of *PatientIdentifier* class. This class can compare automatically data from 2 patients (no need to use *HospitalPatient* type since we have no way to compare extra information) or find a match of a patient in list of hospital patients. Another function is *matchAll*, using which in case of not-match the new hospital patient will be created. Finally, it can try to handle mismatches referring to **IdentityResolver** class. It can either hard match a patient with hospital patient, even if they are not totally same, or create a new hospital patient (used in *matchAll*), or refer to medical staff to resolve this problem. This is done by **IdentityNotifyer** class, which sends a notification about mismatch using *outputStrategy* and the medical staff decides themselves what to do next (e.g. use *hardMatch* from **IdentityResolver**).

4. Data Access Layer



We have an interface ***DataListener*** which assures that all listeners would have a function *listen* to get data from their way of getting information. In our example, we have 3 external data sources: TCP, WebSocket, or input file. As a result of listening, they get list of objects (most generic type). Then, **DataParser** transforms those lists into unified list of Strings where each string is same format and contains information of patients. Using **DataSourceAdapter**, it's easy from that list of strings transform it to map of patients,

which is the way how data is stored in the main **DataStorage**. This system is pretty flexible and doesn't require much changes in case of changes. For example, let's assume we want to add a new way of receiving information. For that, we will need to do only 2 things: create a class to a new listener, and maybe adjust a bit **DataParser**, so that it can transform gotten data into list of strings. Another example is if we assume since now we get more information about patients (e.g. their phone number). For doing that, we again need to change only 2 things: class Patient so that it stores 1 more parameter, and **DataSourceAdapter** which creates instances of **Patient**. As we can see, in each example we don't need to rebuild the whole system again, and most importantly, it doesn't affect the **DataStorage**, so it continues to properly work with other classes, mentioned in the previous UML diagram.