3F6 Software Engineering and Design: 2014 Solutions

Dr Elena Punskaya
21 January 2014

1. (a) The required class diagram is shown in Fig. 1. Any reasonable naming conventions can be used. Generalisation of photo and video classes should be used. Introduction of a class for Tags is desirable.

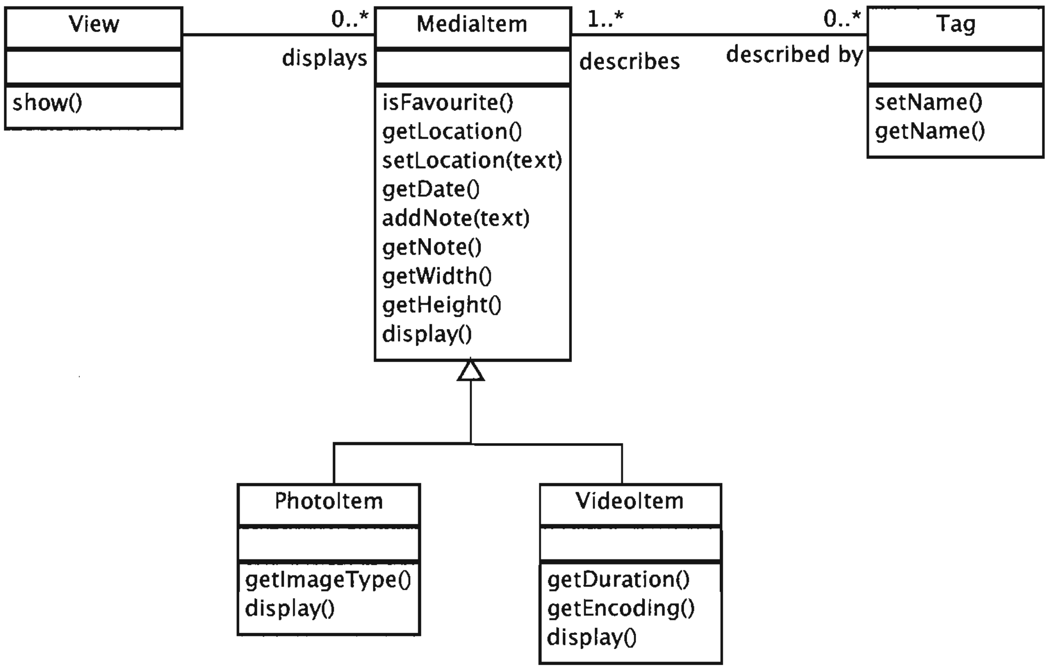


Fig. 1.

1. (b) The required class diagram is shown in Fig. 2. Any reasonable naming conventions can be used.

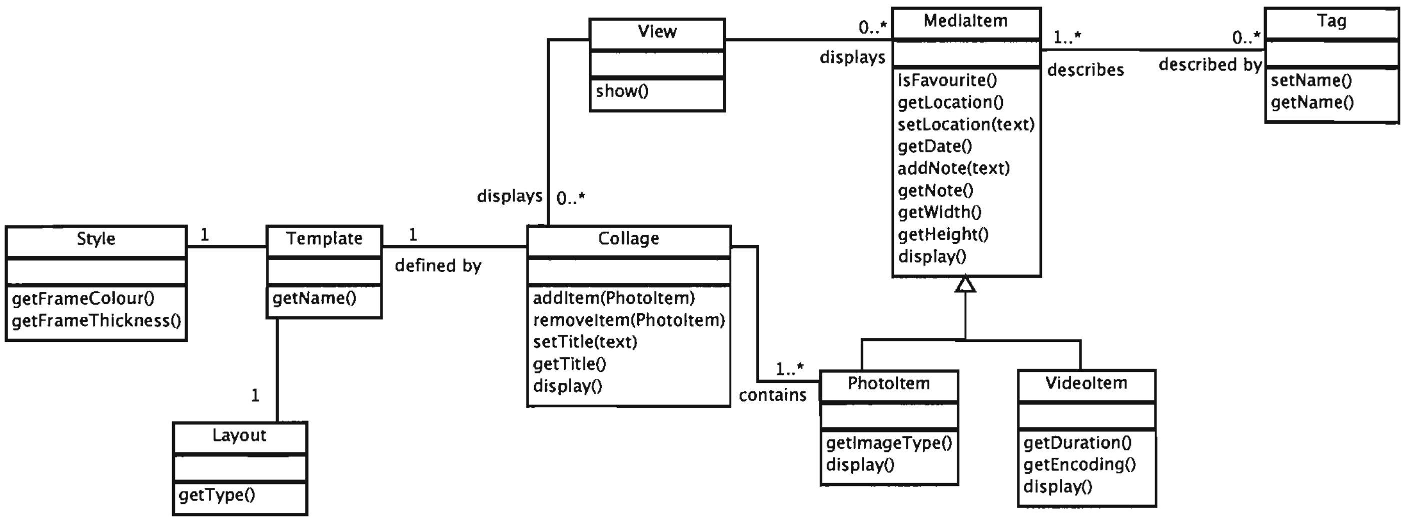


Fig. 2.

1. (c) The required class diagram is shown in Fig. 3. Any reasonable naming conventions can be used.

Use Composite design pattern to allow Collage class to include both simple components such as individual photos or videos and composite components such as collages.

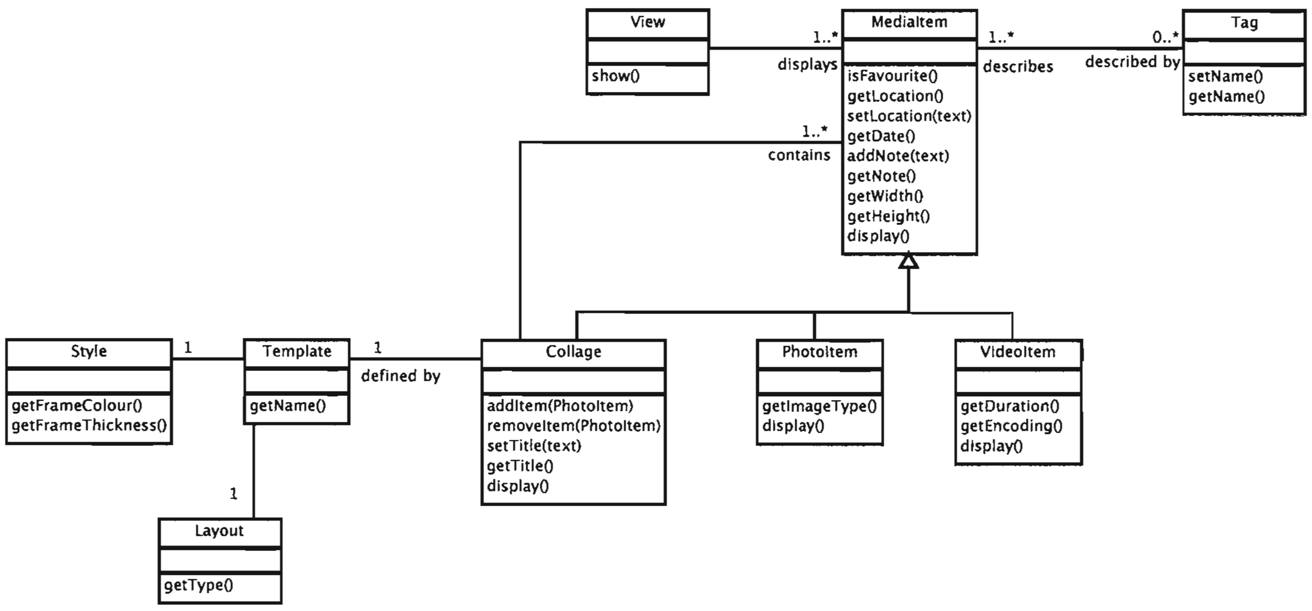


Fig. 3.

1. (d) The required class diagram is shown in Fig. 4. Any reasonable naming conventions can be used.

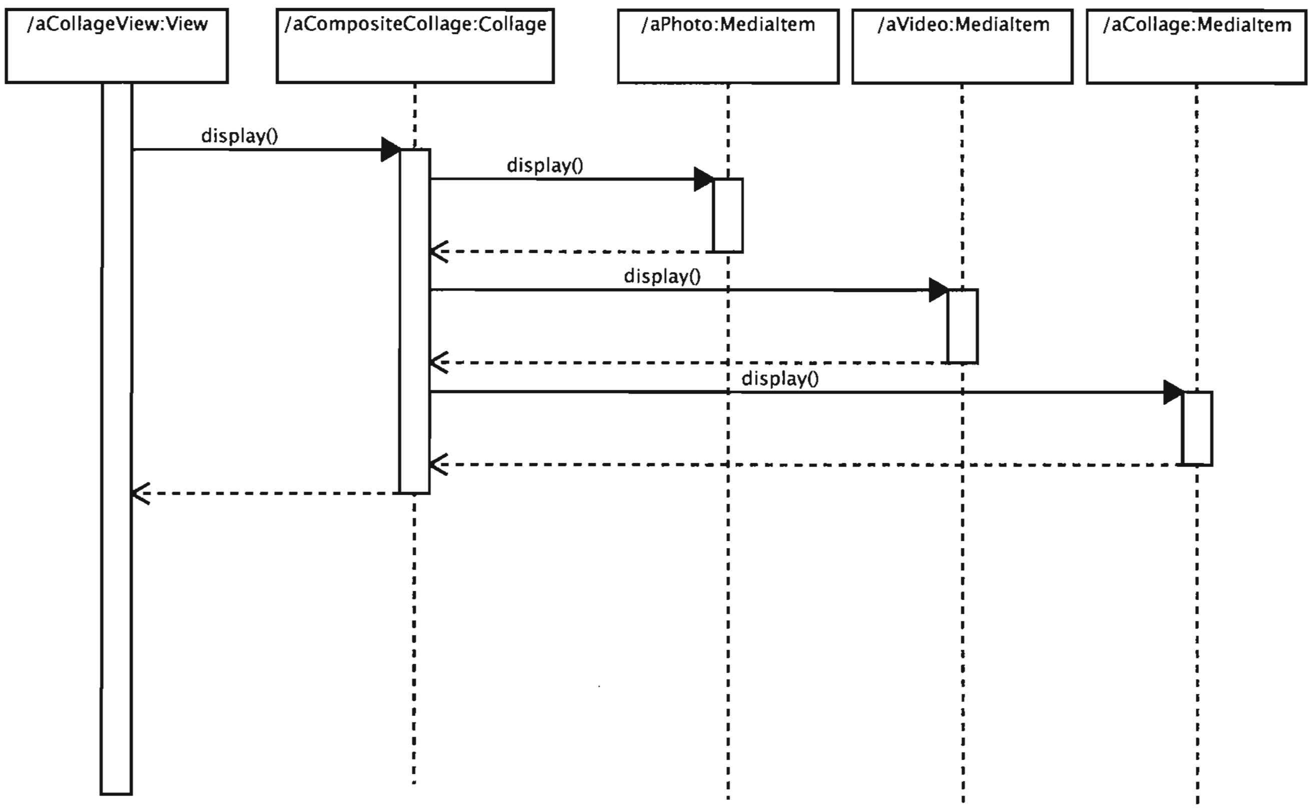


Fig. 4.

2. (a) Medical diagnostics image processing software is health critical and it is important to clarify the goals of the system, all the requirements and spend sufficient time on understanding the all the details prior to the start of the project. Since it is possible and advisable to define all the requirements in advance and the cost of introducing any changes is high the traditional waterfall model is most likely to be used for the task in this case. This model furthermore works well with the management tools, and spending a lot of time on system specification might be an advantage rather than disadvantage in this case.

   The waterfall model follows the sequential design process starting with the definition of the system requirements, followed by software requirements, analysis, program design, coding, testing and operations.

   The system could of course be improved by including iterations and prototyping.

   (b) The waterfall model, if it was followed, is not the most suited for the development of the community website. In this case, it is practically impossible to specify the system requirements in advance, in fact, it is clear from the start that there will be a need to experiment with different features and keep evolving or changing the system, it is not clear how popular the system would become and how quickly it might need to be extended. "Right" today for a community web site would never be "right" tomorrow thus one cycle, big bang approach might not be what is required. The waterfall model unfortunately does not allow for iterations, by the time the product is released there is very limited resource left for any changes or extensions, any changes become very costly and would take a long time - by the time they are actually implemented a completely different set of changes might already be required.

   (c) A more suitable software development methodology for the development of the community website would be an agile methodology. An example of such methodology could be extreme programming although a number of approaches could be used. Such lightweight approaches are based on the assumption that everything (requirements, business cases, technology, user base among others) changes and concentrates on improving the software responsiveness to change. It may include a number of different practices but is very likely to concentrate on automated builds set up, continuous integration techniques, incremental deployment, root cause analyses etc.

   Some of the advantages include being able to release important features quickly, frequent feedback from the users and thus the ability to understand the users better and keep changing the system according to users' changing requirements, which would result in a higher chance of success. Automating repeated tasks would allow to save time and resources and would also make system more reliable.

   The disadvantages however would include investment in automation, more effort required to deploy regularly, automated test and build scripts themselves would become a maintenance overhead and a more close communication between various teams would be required.

   (d) Since we expect a high number of changes an automated testing would be highly recommended in this case with the aim to cover the most practical number of test cases. It would typically include all unit tests, regression tests, UI testing if possible, etc.

   Testing as early as possible (not just before the release) would be recommended. Several examples might be provided such as an approach when the tests are written before the code (Test Driven Development), in particular, following the pattern "test - code' refactor". This approach would only work if a set of good unit tests would be designed which run very fast, concentrate on one thing at a time and clearly reveal its purpose etc.

   However, in addition to this there are various other risks that can be identified in the case of intriduction of the new community website, for example

      i. users may not find the system easily usable and may give up before they even had a chance to find out all the benefits it provides - usability studies could be introduced to mitigate this risk
      ii. the features requested by user might not be understood correctly or not function properly- beta testing to groups of users can be introduced to mitigate this risk
      iii. the website might suddenly become very popular and fail - performance testing, stress testing and testing under load can be introduced to mitigate this risk

3 (a) BOOK WORK: Database normalisation is the process by which the *attributes* and *relations* of a database are organised to avoid logical inconsistencies arising when the database is used. Normalisation aims to minimise redundancy and dependency in the database thereby improving *consistency* (since inconsistencies are prevented by the structure of the database), *extensibility* (since changes to the database structure will only affect parts upon which they are logically dependent), and *efficiency* (since redundant data is not stored). It is important to note that selective denormalisation *may* help for performance reasons.

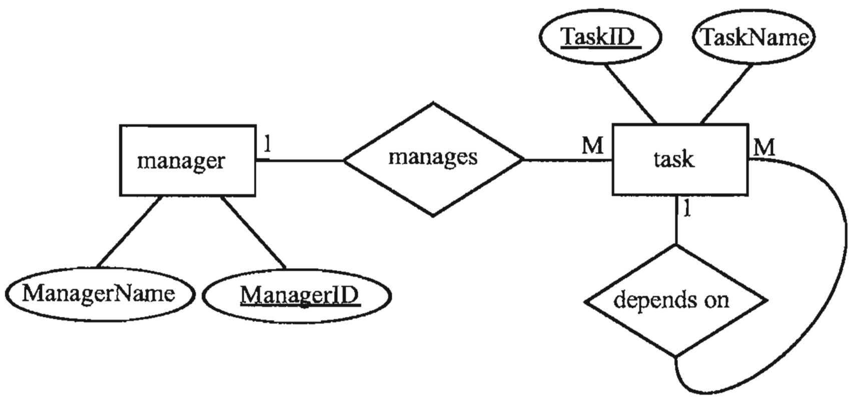
(b) (i) The database design is poor for the following reasons:

- manager names and task names are repeated many times in the relation (consistency/degeneracy/redundancy)
- the dependencies attribute contains lists of tasks which makes the database difficult and slow to query

The design can be improved by

- introducing `TaskID` and `ManagerID` attributes
- splitting the relation into four relations

The following ER diagram shows a redesign that addresses the problems mentioned above and includes the new attributes:



(ii) There are four tables in the updated database:

MANAGER	
<u>ManagerID</u>	ManagerName

<u>Attribute</u> = primary key
Attribute* = foreign key

TASK	
<u>TaskID</u>	TaskName

TASK-MANAGER	
<u>TaskID*</u>	<u>ManagerID*</u>

(from "manages" relationship)

TASK-TASK	
<u>TaskID*</u>	<u>TaskPR*</u>

(from "depends on" relationship)

Foreign Key References TASK.TaskID

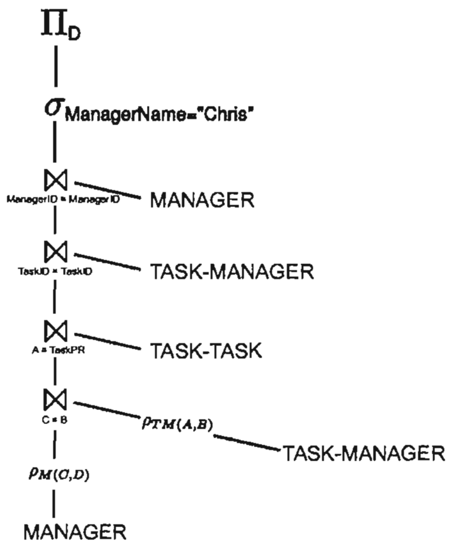
(iii)

$$\prod_{D} \sigma_{\text{ManagerName='Chris'}} \text{MANAGER} \bowtie \text{TASK-MANAGER} \underset{\text{TaskID = TaskID}}{\bowtie} \text{TASK-TASK}$$

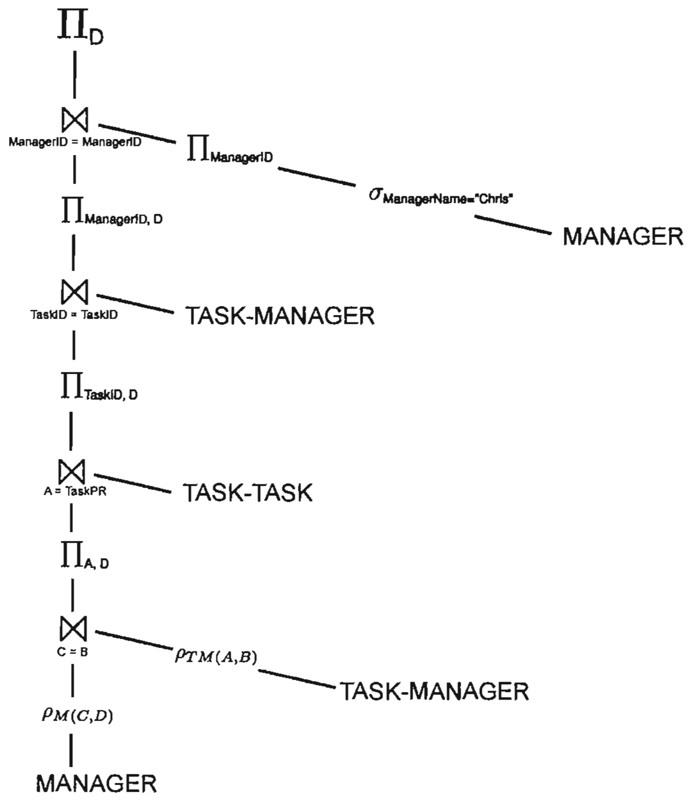
$$\underset{\text{TaskPR = A}}{\bowtie} \rho_{\text{TM(A,B)}} (\text{TASK-MANAGER}) \underset{\text{B = C}}{\bowtie} \rho_{\text{M(C,D)}} (\text{MANAGER})$$

```
SELECT B.ManagerName FROM MANAGER JOIN TASK-MANAGER ON MANAGER.Manage
= TASK-MANAGER.ManagerID JOIN TASK-TASK ON TASK-MANAGER.TaskID
= TASK-TASK.TaskID JOIN TASK-MANAGER AS A ON TASK-TASK.TaskPR
= A.TaskID JOIN MANAGER AS B ON A.ManagerID=B.ManagerID WHERE
MANAGER.ManagerName = 'Chris';
```

(iv) The expression tree for the query in question (iii) is:



After selection and projection pushing the tree is:



(v)

$$\prod_{\texttt{D}} \sigma_{\texttt{TaskName='excavate foundations'}}\texttt{TASK-TASK} \bowtie_{\texttt{TaskPR=A}} \rho_{\texttt{TT(A,B)}}(\texttt{TASK-TASK})$$
$$\bowtie_{\texttt{TaskID=TaskID}} \texttt{TASK} \bowtie_{\texttt{B=C}} \rho_{\texttt{T(C,D)}}(\texttt{TASK})$$

```
SELECT DISTINCT A.TaskName FROM TASK-TASK AS X JOIN TASK-TASK
AS Y ON X.TaskPR = Y.TaskID JOIN TASK AS Z ON Z.TaskID = X.TaskID .
JOIN TASK AS A ON A.TaskID = Y.TaskPR WHERE Z.TaskName = "excavate
foundations";
```

4 (a) BOOK WORK:

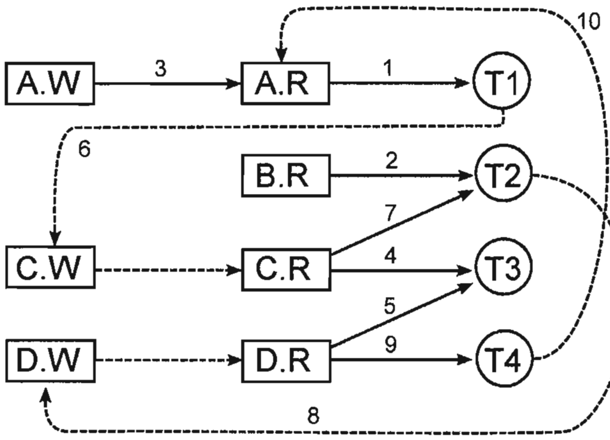
Atomicity: each transaction must be atomic, otherwise it would not be possible to recover cleanly from an aborted transaction.

Consistency: transactions must leave the system in a consistent state. This is essential since any sequence of transactions can be aborted at any point.

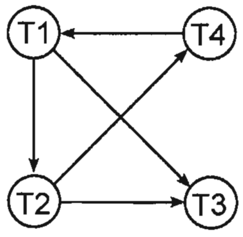
Isolation: results of an incomplete transaction must not be visible to any other transaction. Otherwise, another transaction might see an inconsistent state and produce erroneous results.

Durability: the system must not fail between a transaction committing and the results of the transaction being recorded in the system state.

b) The resource allocation graph is shown below up to and including step 10 at which point deadlock occurs (**a number of candidates forgot that the definition for deadlock is: the first time when a directed cycle appears in the resourse allocation or wait-for-graph**).



c) The corresponding wait-graph is shown below.



d) Recovery from deadlock involves:

* aborting a transation which will break the deadlock loop (called the victim)

* rolling back to the last check point
* redoing the remaining transactions
* restarting the victim at some later point in time

The victim is usually selected using the following criteria:

* has not been running for a long time
* has made few updates
* is blocking multiple transactions

In this case, the choice lies between T1, T2 and T4 (aborting T3 will not avoid the deadlock). T4 is the most sensible choice based on recency and fewer updates. T3 would complete first, followed by T2, and finally T1.