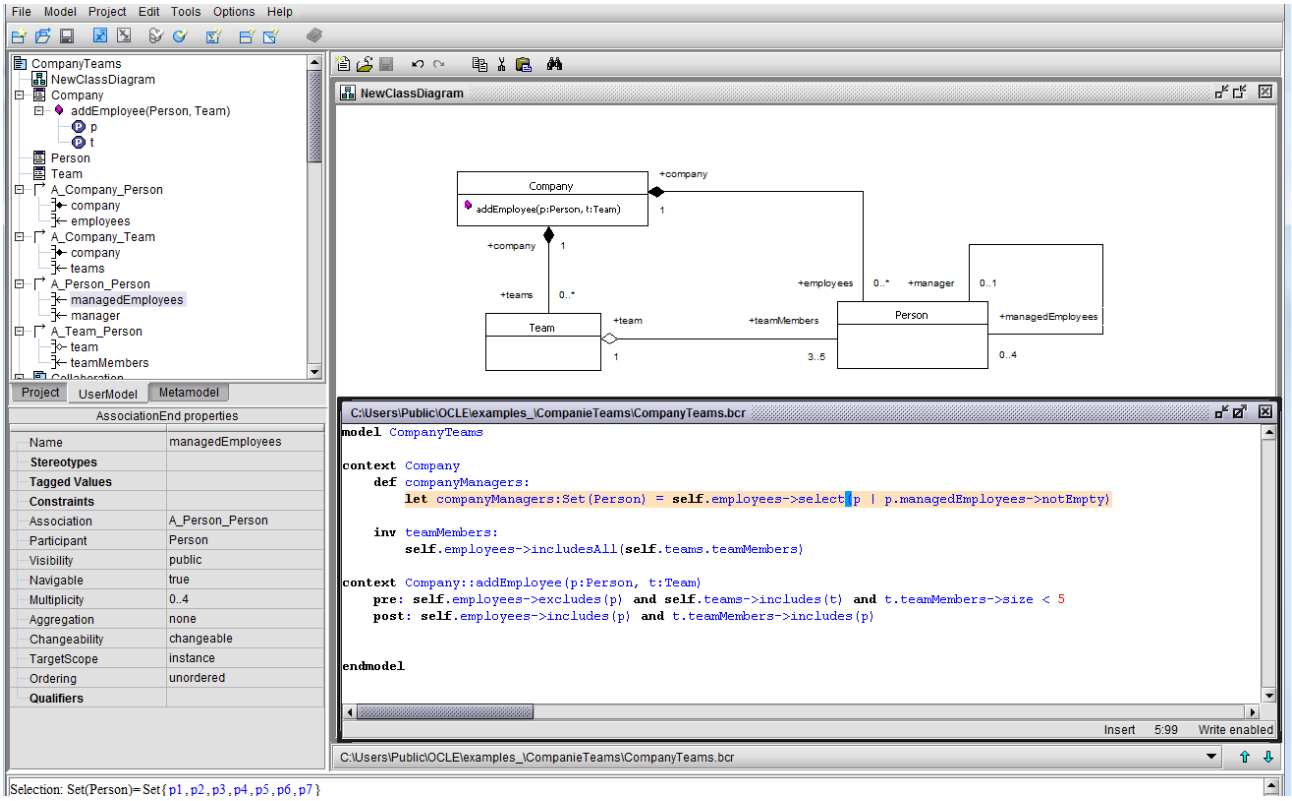
SE exam - 15th of June, 2015 - solutions

I. A company has a set of employees. Each employee may be member of a team having from 3 to 5 members. Each team has a manager, member of that team. In turn, team managers may have another manager (leader) who is not a member of any team.

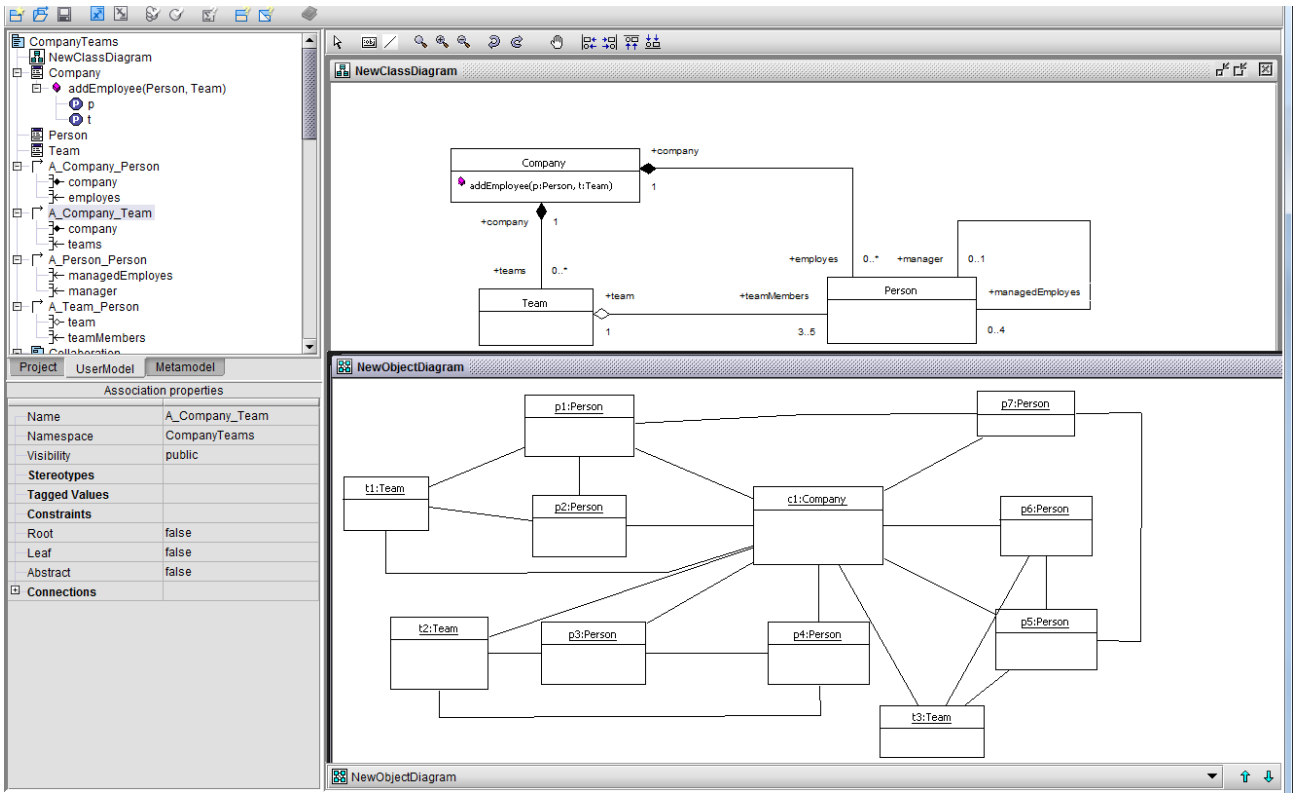
a. By means of a UML class diagram, please describe the structure of a model which structure complies with the above requirements. 1.5 pt

b. Using OCL please specify in the context of Company an observer computing the set of managers for the current company. Also, in the same context, please specify an invariant stating that all teamMembers of all the company teams are included in the set of employees of that company. 0.5 pt

c. In the context of the operation Company::addEmployee(p:Person, t:Team) that adds a new p employee to the company, as member of the team t; please specify appropriate pre and postconditions ensuring that the value of the above invariant will remain true after executing an addEmployee(...) message. 1 pt



II. By means of an object diagram (snapshot) please represent a model instantiation in which a company c1 has 7 employees: p1,...,p7 and 3 teams t1, t2 and t3. p1 and p2 are t1 members, p3 and p4 are t2 members and p5 and p6 are t3 members. p1 is p2 manager, p3 is p4 manager, p5 is the manager of p6. p7 is the manager of p1 and p5. Does this snapshot comply with the UML model? Justify your answer. 1 pt



The snapshot doesn't comply with the model simply because each team must have at least 3 members and not 2 like in our case.

III. Please mention the "types"(kinds) of objects used in analysis models and describe shortly the role of each "type" and the rationale of using this distinction in describing analysis models. What's the mechanism/manner of distinguishing the types in a class diagram describing the structure of the analysis model? 1 pt

The analysis object model consists of **entity**, **boundary**, and **control objects** [Jacobson et al., 1999]. **Entity objects** represent the persistent information tracked by the system. **Boundary objects** represent the interactions between the actors and the system. **Control objects** are in charge of realizing use cases.
Modeling the system with entity, boundary, and control objects provides developers with simple heuristics to distinguish different, but related concepts. For example, the time that is tracked by a watch has different properties than the display that depicts the time. **Differentiating between boundary and entity objects forces that distinction.**

The three object-type approach also leads to models that are more resilient to change: the interface to the system (represented by the boundary objects) is more likely to change than its basic functionality (represented by the entity and control objects). By separating the interface from the basic functionality, we are able to keep most of a model untouched when, for example, the user interface changes, but the entity objects do not.
To distinguish between different types of objects, UML provides the stereotype mechanism to enable the developer to attach such meta-information to modeling elements.

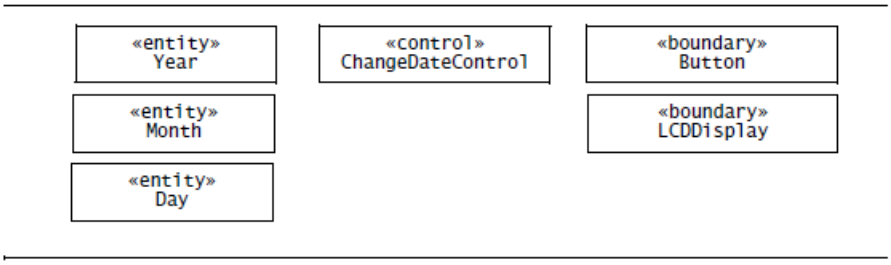


Figure 5-5 Analysis classes for the 2Bwatch example.

IV. Please explain the concept of model transformation. What's code generation and which are the terms used for referring code generation? Which are the advantages that can be obtained by code generation? Please describe shortly which parts of code can be generated for each kind of diagram and complementary specifications. 2.5 pt

A **transformation** aims at improving one aspect of the model (e.g., its modularity) while preserving all of its other properties (e.g., its functionality). Hence, a transformation is usually localized, affects a small number of classes, attributes, and operations, and is executed in a series of small steps. These transformations occur during numerous object design and implementation activities. We focus in detail on the following activities:

- *Optimization* that addresses the performance requirements of the system model. This includes reducing the multiplicities of associations to speed up queries, adding redundant associations for efficiency, and adding derived attributes to improve the access time to objects.
- *Realizing associations* map associations to source code constructs, such as references and collections of references.
- *Mapping contracts to exceptions* that describe the behavior of operations when contracts are broken. This includes raising exceptions when violations are detected and handling exceptions in higher level layers of the
- system.
- *Mapping class models to a storage schema.* During system design, we selected a persistent storage strategy, such as a database management system, a set of flat files, or a combination of both. During this activity, we map the class model to a storage schema, such as a relational database schema.

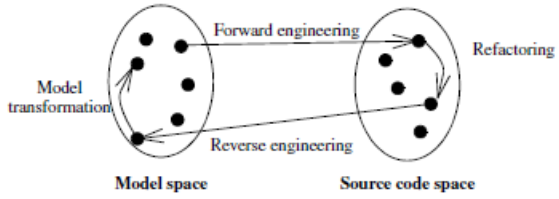


Figure 10-1 The four types of transformations described in this chapter: model transformations, refactorings, forward engineering, and reverse engineering.

V. Please describe the concepts related to software testing and represent these by means of a class diagram. 1.5 pt

Testing Concepts

- A **test component** is a part of the system that can be isolated for testing. A component can be an object, a group of objects, or one or more subsystems.
- A **fault**, also called *bug* or *defect*, is a design or coding mistake that may cause abnormal component behavior.
- An **erroneous state** is a manifestation of a fault during the execution of the system. An erroneous state is caused by one or more faults and can lead to a failure.
- A **failure** is a deviation between the specification and the actual behavior. A failure is triggered by one or more erroneous states. Not all erroneous states trigger a failure.
- A **test case** is a set of inputs and expected results that exercises a test component with the purpose of causing failures and detecting faults.
- A **test stub** is a partial implementation of components on which the tested component depends.
- A **test driver** is a partial implementation of a component that depends on the test component.

Test stubs and drivers enable components to be isolated from the rest of the system for testing.

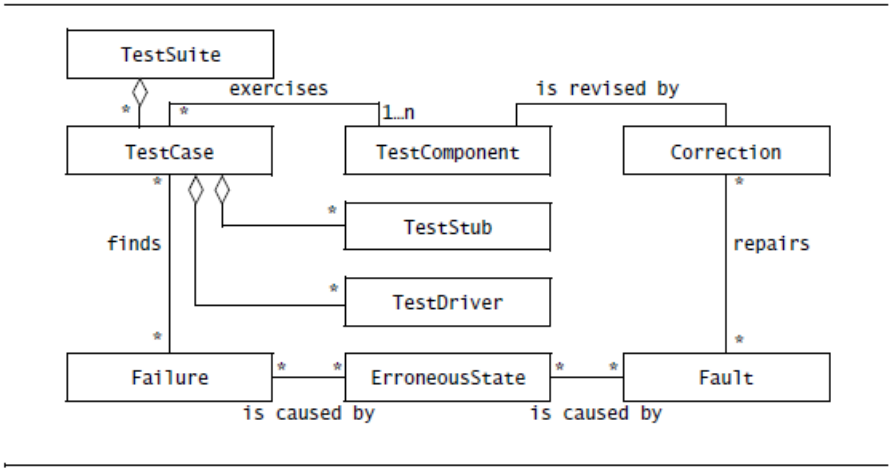


Figure 11-2 Model elements used during testing (UML class diagram).

Total 9 pt
1 pt by default