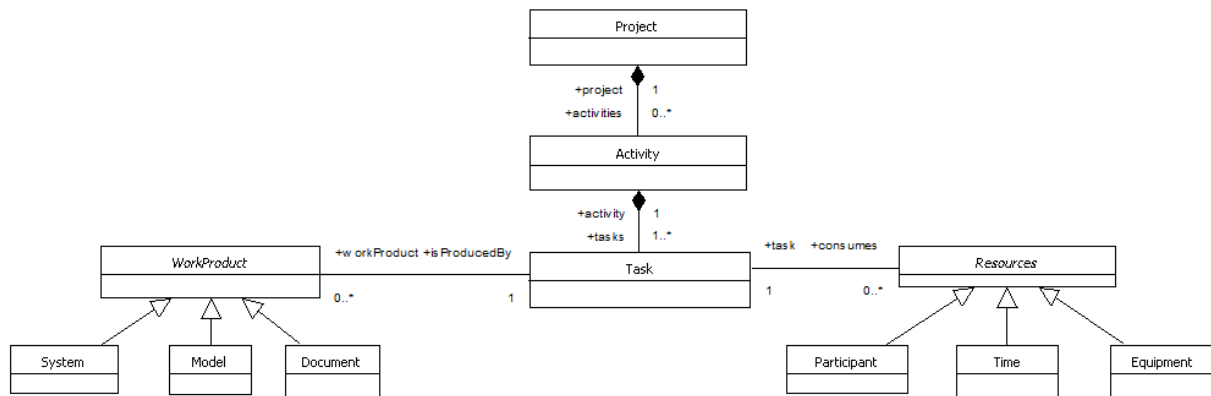


SE midterm exam - June 1, 2014 - solutions

In Software Engineering (SE) a project, whose purpose is to develop a software system, is composed of a number of activities. Each activity is in turn composed of a number of tasks. A task consumes resources and produces a workProduct. A workProduct can be a system, a model, or a document. Resources are participants, time, or equipment. 2.5 pt

- Using the UML, please represent the architectural model of the SE concepts mentioned above. Please specify in OCL an invariant stating that for each workProduct there is allocated at least one resource. 0.5 pt
- In the context of WorkProduct, please specify an observer returning a tuple with 2 components: a workProduct and the size of consumed resources. The number of resources allocated to the workProduct of the tuple has to be the maximum in the system. 1 pt



context WorkProduct

inv atLeastOneResourceAllocated:

self.isProducedBy.consumes->size >= 1

def theMostResourcesConsumer:

```
let theMostResourcesConsumer: TupleType(w:WorkProduct, rn:Integer) =  
    WorkProduct.allInstances->collect(wp:WorkProduct | Tuple{w = wp,  
    rn = wp.isProducedBy.consumes->size})->sortedBy(t | t.rn)->last
```

2. Using your own words, please describe what do you mean by requirement elicitation? Why this activity is important and why is it challenging? 1.5 pt

A requirement is a feature that the system must have or a constraint that it must satisfy to be accepted by the client. **Requirements engineering** aims at defining the requirements of the system under construction. **Requirements engineering** includes two main activities; *requirements elicitation*, which results in the specification of the system that the client understands, and *analysis*, which results in an analysis model that the developers can unambiguously interpret. **Requirements elicitation is the more challenging of the two because it requires the collaboration of several groups of participants with different backgrounds.** On the one hand, the client and the users are experts in their domain and have a general idea of what the system should do, but they often have little experience in software development. On the other hand, the developers have experience in building systems, but often have little knowledge of the everyday environment of the users.

3. By means of an object diagram, please represent the elements (models/views) of an analysis model and the kind of UML diagrams associated to these models. 1 pt

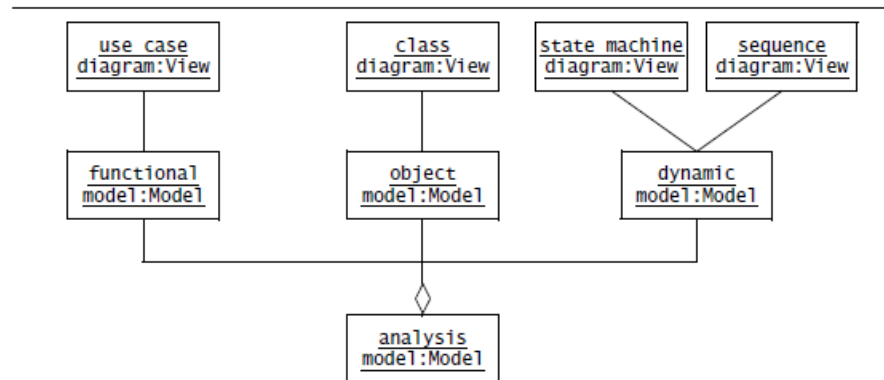
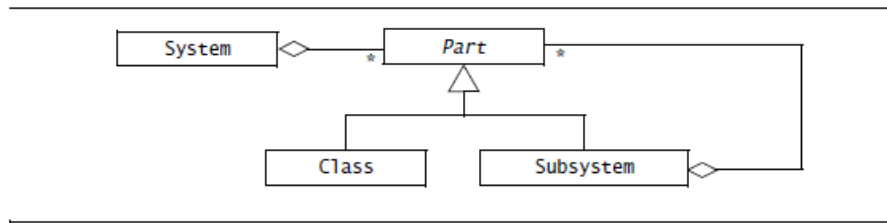


Figure 5-3 The analysis model is composed of the functional model, the object model, and the dynamic model. In UML, the functional model is represented with use case diagrams, the object model with class diagrams, and the dynamic model with state machine and sequence diagrams.

4. The class diagram below presents system decomposition. Please mention which are the main rationales of this decomposition? What's the role of Part? If this architecture complies with a design pattern, please name it and describe the main advantages of using it. 1 pt



The system decomposition

To reduce the complexity of the solution domain, we decompose a system into simpler parts, called “subsystems,” which are made of a number of solution domain classes. A subsystem is a replaceable part of the system with well-defined interfaces that encapsulates the state and behavior of its contained classes. A subsystem typically corresponds to the amount of work that a single developer or a single development team can tackle. By decomposing the system into relatively independent subsystems, concurrent teams can work on individual subsystems with minimal communication overhead. In the case of complex subsystems, we recursively apply this principle and decompose a subsystem into simpler subsystems (see Figure above). **The composite pattern supports a uniform access to both containers and contained elements by means of Part.**

Please describe shortly the Adapter Design Pattern and represent its architecture by means of a class diagram. 1.5 pt

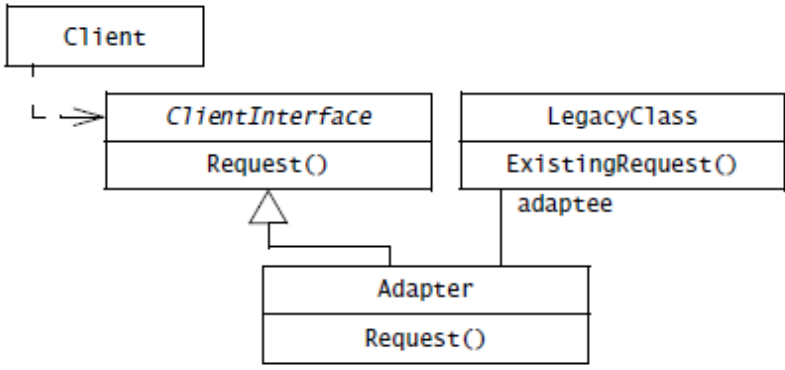
<i>Name</i>	Adapter Design Pattern
<i>Problem description</i>	Convert the interface of a legacy class into a different interface expected by the client, so that the client and the legacy class can work together without changes.
<i>Solution</i>	<p>An Adapter class implements the <code>ClientInterface</code> expected by the client. The Adapter delegates requests from the client to the <code>LegacyClass</code> and performs any necessary conversion.</p>  <pre> classDiagram class Client class ClientInterface { Request() } class LegacyClass { ExistingRequest() } class Adapter { Request() } Client --> ClientInterface Adapter -- > ClientInterface Adapter --> LegacyClass : adaptee </pre> <p>The diagram illustrates the Adapter Design Pattern. It features four classes: <code>Client</code>, <code>ClientInterface</code>, <code>LegacyClass</code>, and <code>Adapter</code>. <code>Client</code> has a dependency on <code>ClientInterface</code>, indicated by a solid line with an open arrowhead. <code>ClientInterface</code> defines a <code>Request()</code> method. <code>LegacyClass</code> defines an <code>ExistingRequest()</code> method. <code>Adapter</code> is a subclass of <code>ClientInterface</code>, as shown by a solid line with an open triangle arrowhead. <code>Adapter</code> also has a dependency on <code>LegacyClass</code>, indicated by a solid line with an open arrowhead, labeled "adaptee". The <code>Adapter</code> class implements the <code>Request()</code> method.</p>
<i>Consequences</i>	<ul style="list-style-type: none"> • <code>Client</code> and <code>LegacyClass</code> work together without modification of neither <code>Client</code> nor <code>LegacyClass</code>. • <code>Adapter</code> works with <code>LegacyClass</code> and all of its subclasses. • A new <code>Adapter</code> needs to be written for each specialization (e.g., subclass) of <code>ClientInterface</code>.

Figure 8-5 An example of design pattern, Adapter (adapted from [Gamma et al., 1994]).