Designing Object Oriented Software
Exam I
Mon May 7, 2007 / Ma 7.5.2007

It is allowed to use the following book during the examination:

Robert C. Martin:
Agile Software Development: Principles, Patterns, and Practices,
Prentice-Hall, 2003.

Any other extra materials (e.g. lecture notes, slides, extracts, copies, or handouts) are *not* allowed.

• Each question gives at most 10 exam points. Because the digressions from the topic will reduce the received exam points, please, stick to the subject. • There are three questions to be answered. • To receive the exam points You are expected to include Your name, student number and the course title ("Designing Object Oriented Software") on the returned paper sheets.

---

1. Let us consider the most widely used object-oriented programming languages (OOPLs) such as C++ or Java. These OOPLs provide the concepts 'interface' and 'class' that can be used for describing the modular design of a software system both at definition time and at runtime. However, the 'interface' and 'class' mechanisms as such are not sufficient for defining modules completely. Discuss what other modularity properties these kinds of OOPLs should provide or support more directly. *Hint: Base Your arguments, for example, on the modularity concepts presented by Robert C. Martin and Bertrand Meyer.*

2. Design patterns Bridge, State, and Strategy have similar class diagram structures but their intentions are quite different. In what kind of use contexts each of these pattern is more suitable than the other ones? Describe the use contexts of these patterns in general terms and concretize Your comparison by giving some examples.

3. Let us assume that we have a software (Sw) system called Appl that is build from three packages ApplState, ApplRendering, and ApplControl as presented in Figure 1. This packaging structure is derived from the following rationales.

MORE ON THE NEXT PAGE —

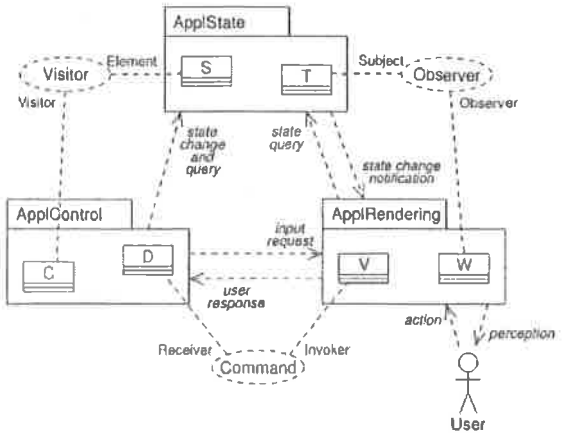

Figure 1: The organization of the packages in system Appl. A dashed arrow depicts a dependency (the attached text in italics describes an example for it). A dashed oval denotes a design pattern (the attached lines bind its collaborating roles to the concrete interfaces).

- ApplState: The package defines how to encapsulate and preserve the state data of the system. Here, 'state data' means the information that define the current state of Appl at the system level (i.e. *not* the member variables in the class definitions). This package enables us to replace the existing state implementation, for example, by a database. Also, we can relocate the whole state data to a remote computer in future. [Analogous example: In a weather monitoring system (WMS) this kind of package would include the classes that keep up the weather measurements and uphold the user accounts.]
- ApplRendering: The responsibility of this package is to define how to render the necessary state data to the users of the system. Furthermore, it encapsulates and hides all the user interface (UI) issues. Now, the rest of the system does not need to know if it is used via a graphical window or via a command prompt terminal. This allows us to adapt the system to many operating systems and platforms. [Analogous example: In a WMS this kind of package includes the classes that handle the user interaction interface.]
- ApplControl: This package defines the control logic (i.e. the behaviour) of the system. Also, the state data is changed only by the objects instantiated from the classes of this package. For example, the system's integrity rules and error recovery logic are located here. This makes it easier to have many levels of product features; the more the customer pays the more advanced features are delivered. [Analogous example: In a WMS this kind of package includes the classes that drive the measurements and their timings, automatically calibrate the instruments, and alert the faults.]

Figure 1 outlines how the objects from the different packages collaborate with each others. For example, ApplRendering objects can be observers of ApplState objects, ApplControl objects can hook their functionality into ApplRendering objects by Command pattern, and data objects of ApplState can be operated from ApplControl objects by Visitor mechanism. NOTE, that the classes S, T, V, W, C, and D are presented only for illustration purpose; the true classes and interfaces of these packages are not shown.

Evaluate the packaging structure of Appl system at least from the following perspectives: Sw reusability, Sw release policy, change in Sw requirements, and concurrent Sw development (one team dedicated to one package). Identify the most serious problem in the packaging structure and solve it under the following restriction: We want to keep packages ApplState, ApplRendering, and ApplControl — each has their own solid justification in the forthcoming evolution of Appl.