

CS 327 Exam 1 – Fall 1994 – Steven A. Demurjian

Name: _____

Problem	Part	Points	Score
1	a.	10	
	b.	10	
	c.	10	
	d.	10	
	e.	10	
	f.	10	
	Subtotal	60	
2		20	
3		20	
	Total	100	

Use only one side of the paper and start each problem on a new page!!

Please show all work to receive ANY credit!!!!

Note that you should manage your time effectively, and play close attention to point amounts, so that you're not working on excessive answers!

Roughly equate 10 points to fifteen (15) minutes of time/effort.

1. (60 points total) Unit of Abstraction for Object-Oriented Paradigm

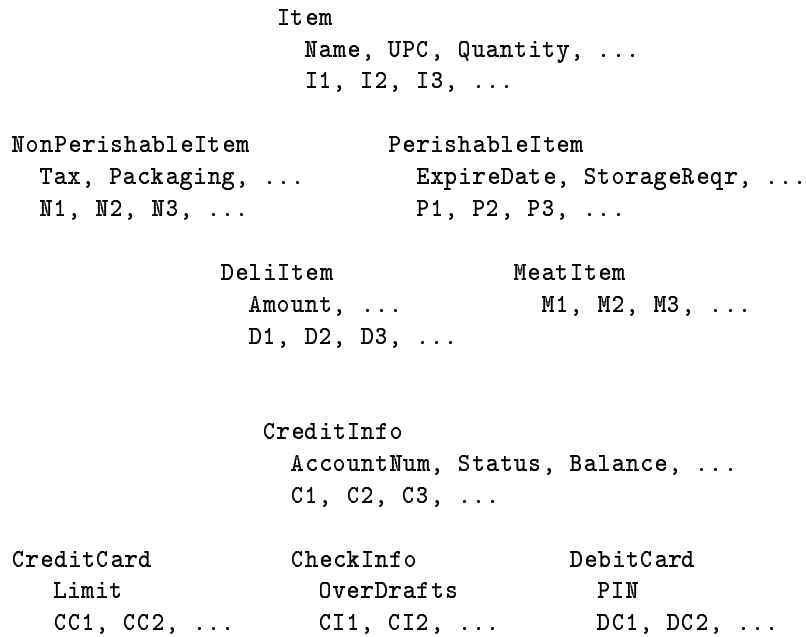
The public interface of an object type (OT)/class has evolved from an abstract data type (ADT), which in the past has represented the unit of conceptual abstraction. That is, when ADTs were first introduced by Liskov, the unit of abstraction was a single data type. Classical examples of ADTs, Stack, Queue, etc., which are still widely utilized today, focused on a single unit of well-defined functionality. At that time, having a unit of abstraction as a single type made sense, since the complexity and capabilities of computers, software, and applications was primitive compared to today. But, are these examples reflective of the current practice? Is an ADT *yesterday's* unit of conceptualization? Since object-oriented concepts are often differentiated from ADTs via inheritance, is the ADT as a unit of abstraction still relevant?

Inheritance as one of the defining components of the object-oriented paradigm, with its support for extensibility and evolution, must be part of the discussion when trying to identify an appropriate current unit of abstraction. The overriding reality is that most, if not all, object-oriented applications, are firmly rooted in an OT/class library design and implementation, as has been stressed when discussing object-oriented concepts in both Chapters 2 and 4. These libraries are characterized by the presence of one or more inheritance hierarchies. What role does such a hierarchy play in this situation? There is strong evidence that a hierarchy is serving as a *larger* unit of conceptualization than its ADT ancestor. In the late 1970s, given the state, size, and capabilities of computing hardware and software, developing applications with multiple ADTs was both reasonable and feasible, since each ADT would represent a significant portion of the functionality at that time. However, as the mid-1990's approach, the complexity of

applications and their underlying platforms has exploded! Today's class libraries are intended to capture all aspects of an application's functionality, with individual inheritance hierarchies representing substantial portions. Collectively, the hierarchies that comprise an application divide the functionality into meaningful subsets. Thus, each hierarchy represents a significant conceptual abstraction (a macroscopic view), which can be decomposed microscopically to individual OTs/classes, where each characterizes the 'older' ADT perspective.

Moreover, the individual hierarchies are used as an entire conceptual unit by different aspects of the application. That is, when a subset or tool of the application utilizes a hierarchy, it is likely to need access to most or all of its constituent OTs/classes. It would be unusual for a tool to only use one OT/class of either one or multiple hierarchies; this often indicates a poor design. Thus, an inheritance hierarchy is a conceptual unit, since it was formed on the basis of a known set of commonalities and similarities of data and/or behavior (methods) and/or usage, intended to serve specific, well-defined, and consistent purposes in the overall application.

For example, in HTSS, an overriding concern in the design and development process as discussed in Chapters 2, 3, and 4, has focused on a type hierarchy for items. Another hierarchy, that has not been discussed in detail, might involve payment methods for a customer's order. Both hierarchies are shown below.



In both hierarchies, a subset of relevant data has been indicated. When one considers the way that HTSS utilizes each hierarchy, it is clearly evident that usage occurs based on access to specific portions of particular hierarchies. For example, the `Locator` requires access to the entire `Item` hierarchy, since in addition to printing out 'where' an item is, it will also print out information on the item. The `CashRegister` requires access to to both hierarchies, to facilitate the tasks of generating a receipt and taking payment. The `DeliOrderer` interface would focus on only the `PerishableItem` subtree for supporting its functionality. Clearly, there are other examples in HTSS that might only use one class in a given hierarchy. However, in general, one can conclude that different aspects of the application will typically require access to significant portions of one or more different inheritance hierarchies.

This question asks you to further explore the ideas and issues related to the unit of abstraction for the object-oriented paradigm in general, and its impact on the different concepts and topics that have been discussed in Chapters 1 to 4. The basic assumption is: Suppose that instead of single OTs/classes, the unit of abstraction was an entire inheritance hierarchy. Your answers should address the impact (either positively or negatively) of the assumption.

- (a) Discuss the impact of changing the unit of abstraction to an inheritance hierarchy on basic object-oriented concepts and features, namely, the public interface and the private implementation (thereby including encapsulation and hiding). To assist in formulating your answer, the following questions might be useful:
 - In HTSS, can a descendant of `Item` access and change its private data directly? Why or why not?
 - Is the public interface defined for the entire hierarchy as a combination of all public methods? If so, what are the implications of such an approach? If not, why not?
 - Do actions occur at only leaf nodes?
- (b) A more critical concern is the impact of revising the unit of abstraction on extensibility, namely, software reuse and evolution. Are reuse and evolution impacted by a change in the unit of abstraction? If so, how? If not, why not? Note: Your answer in part (a) might assist you in formulating your answer to this part of the question.
- (c) Generics are important to object-oriented development, since they provide a means for a IE to develop a single piece of code that can be customized based on type, thereby meeting multiple needs with a single effort. The majority of our discussion has considered a generic for a single OT/class. Does the concept of a generic inheritance hierarchy make sense? Why? If so, provide a generic hierarchy and indicate its expected utility. If not, make sure that you have a strong argument with supporting reasons. Note: Again, answers to parts (a) and (b) may be helpful.
- (d) Recall the reference model as presented in Section 6 of Chapter 2 and in ZM (pgs. 13-23). Two features of the reference model that we discussed were: persistence by reachability and versions. Are these two features impacted by the inheritance hierarchy as a unit of abstraction? If so, how? If not, why not?
- (e) In Chapter 4, the ECRC approach for high-level design was presented as an abstraction and conceptualization technique to transition from the specification to a detailed design. Which, if any, of the characteristics of individual decks and/or hands and/or cards must be changed and/or modified to support an inheritance hierarchy as a unit of abstraction? Do any of these changes have an impact on the specification process as discussed in Chapter 3? Again, make sure that you provide a cohesive argument in your answer.
- (f) One of the other strengths of the object-oriented paradigm is its support for the testing process. The premise is that an object type can be designed, implemented, and tested in a manner that is independent of other portions of an application, and in a fashion that is more comprehensive. The claims are that the resulting implementation is more robust and less prone to maintenance. If you accept the premise/claims, the next question is on the impact of a change of a unit of abstraction on testing. Is there an impact? If so, is the testing process enhanced or degraded? If not, why not?

Bonus: Argue for or against the original idea: Should we upgrade the unit of abstraction to be an entire inheritance hierarchy rather than a single OT?

In answering each part of this question, don't hesitate to utilize examples from HTSS or other application domains that illustrate any points that you make.

2. (20 points total) Specification and Performance

The Performance section (Chapter 3, Section 2.1.7) of a specification appears to consider not only the time, space, and consistency considerations for the system, but also addressed questions related to fault tolerance and robustness. An argument can be made for differentiating between different aspects of performance by reorganizing the questions into subsets, related to, for example: time and space; fault tolerance, and robustness. Consider the following issues:

- (a) Propose a reorganization of the questions into multiple subsets to more adequately reflect commonalities and differences, i.e., capture different perspectives of performance.
- (b) For each of your subsets, propose questions that should be added to capture performance considerations that have been overlooked.
- (c) Once parts a. and b. have been completed, examine the dependencies of your "new/revised" Performance section to other sections in the specification.

3. (20 points total) Searching for Silver Bullets

Harel's article on "Biting the Silver Bullet" contains a number of interesting ideas and concepts that are, in general, contrary to the approach and philosophy that has been taken in class. The article begins by establishing a focus on "reactive" systems (pg. 10), which includes applications for embedded, concurrent, or real-time computing, and not ones that are data intensive such as database or management information systems. It has been argued in Chapters 1 and 3 of the textbook, that application design, particularly at the specification level, must be all inclusive concerning different application requirements. It has also been strongly advocated that databases are norms rather than exceptions in advanced applications. In fact, other than embedded computing, where space is often limited, both concurrent and real-time computing are often highly data intensive, seeking either parallel or distributed solutions as a means to increase performance. Briefly comment on the following issues that have been raised in Harel's article. Each part of the question is worth 5 points.

- (a) In the section on "Modeling Behavior" (pgs. 11-12), Harel focuses on state and transition diagrams as a means to capture and represent control flow for reactive systems. What features of the object-oriented paradigm, at both design and implementation levels, also support behavior? Explain.
- (b) How does Harel's view on "Strata of Conceptual Models" (pg. 12) compare against the overall design and development process as we have examined in class? That is, compare and contrast his specific views of models against the one discussed in class.
- (c) Object-oriented concepts are considered only briefly (paragraph on pg. 13 that begins "Some methods ..."). If one examines any recent OOPSLA (Object-Oriented Programming, Systems, Languages, and Applications) proceedings, it becomes quickly evident that many researchers and companies are applying object-oriented concepts and techniques to reactive systems. Do you believe that Harel's dismissal of object-oriented concepts for reactive systems is justified? Why or why not?
- (d) Examine Harel's view of "Analyzing the Model" (pgs. 14-18) against our approach to adding engineering rigor. Provide one similarity and one difference.