CS-322 Lecture Notes

OBJECT-ORIENTED ANALYSIS
AND DESIGN

Walid S. Saba
School of Computer Science
University of Windsor

Accompanying Material
• Object-Oriented Design Heuristics, Arthur Riel, Addison Wesley
• UML specification (any source is fine)

Other Recommended Material
• Design Patterns, Gamma et al.
• Object-Oriented Analysis and Design, Grady Booch, Benjamin Cummings.
• Introduction to Object-Oriented Software Engineering (Ivar Jacobson et al.)

NOTES
• Do not use object-oriented programming languages books (such as C++, CLOS/Scheme, Java or Smalltalk) as your introduction to object-oriented analysis and design since they typically introduce concepts from the vantage point of a single language while these concepts should be understood independent of any language. The only exception here (that I am aware of) is Coplien’s book (Advanced C++ Porgramming Syles and Idioms). This book reflects Coplien’s rich experience in theory and practical SW development on large scale projects at Bell Labs. He also discusses C++ and OOP in the larger context of how to deal with situations where symbolic or applicative/functional programming seems to be superior.

• The best way to learn OO analysis and design is to work in groups on trying to come up with a design for some real-life problem. The discussions (some sessions might go beyond ‘discussions’) you will have when deciding on the key abstractions, on the object model, on the various relationships, or on the various positions of centralized vs. distributed control will teach you more on what OO analysis and design involves than you might first realize. Make up some groups and work on different designs. I am willing to referee.
## CONTENTS

### I. Preliminaries (2 Lectures)
1. Understanding technical features
2. Design tradeoffs
3. Why different programming paradigms?
4. Tracability and maintainability
5. The promise of the OO paradigm

### II. Building Blocks of the Object Model (2 Lectures)
1. Classes, objects, object identity and equality
2. Hierarchies ("isa" and "part-of")
3. Inheritance and polymorphism
4. Multiple inheritance and object construction
5. Containment, delegation and the "uses" relationship
6. Delegation and re-use
7. Some modeling heuristics
8. Functors: higher-order programming (basis of some design patterns)

### III. Object-Oriented Analysis and Design (8 Lectures)
1. Modeling and analysis
2. OO analysis
3. Modeling tools (summary using the UML notation)
4. Examples illustrating design tradeoffs
5. Invariants, semantic and cardinality constraints
6. A non-trivial application: an elevator system
7. Modeling dynamic behavior: state transition diagrams (STDs)
8. Some design heuristics
9. Refining a design
10. On concurrency
11. Design patterns
12. Persistence and data layers
13. An ATM Machine
14. The Elevator System

### IV. Related Topics (2 Lectures)
1. Hybrid object and rule-based design
2. CORBA and COM
3. Distributed component-based computing
4. A society of virtual agents

### V. Revision (1 Lecture)
Introduction

Designing software for large applications is a very complex task. It could arguably be said that there is no limit to how complex software design could get. The reason for this is that we generally design software to model some real-world phenomenon. The more realistic we aim to make our model the more complex the design, since in general real-world phenomena can be very complex. Take, for example, the modeling of activities on the stock market. To accurately represent all the parameters (variables) that might affect the behavior of a certain stock a system must take into account various statistical data (trends), as well as a complicated set of interrelationships between various pieces of data (for example, political instability in the middle east might affect the price of oil, which in turn might affect the inflation rate, which again, incidentally, might affect the price of oil.) How can such complex relationships be modeled? There are several reasons for the complexity here. The main source of the complexity in the modeling of the stock market is the fact that this model involves two distinct set of rules: (i) a set of analytic rules that can be well specified by expert actuaries and statisticians; and (ii) a set of commonsense rules and a number of heuristics that represent the "knowledge" of some experts. One challenge here is how to ultimately separate these two distinct sets of rules in the software, since the latter is a very dynamic set that could change on a daily basis.

If modeling the stock market is not complex enough, imagine the complexities involved in modeling the cognitive and reasoning capabilities of humans in building intelligent machines. Researchers in artificial intelligence (AI) are trying to emulate human perceptual, language, and reasoning capabilities. In addition to the conceptual breakthroughs, there are various computational challenges that arise when building such systems.

The fact that the logic of some applications can be very complex is one source of complexity in software design. The other source of complexity comes from the fact that software in the end is executed on machines and thus applications must be written with a computational environment in mind. While some of the constraints of the computational environment are not (and should not be) considered at the analysis phase, time and memory constraints, as well as issues that relate to concurrency, whether the system needs to be dynamic or static, add another level of complexity.

The point of all of this is that building large applications is a complex task and like engineers in other disciplines, software engineers need a good set of tools that can aid the analysis, the design and ultimately the implementation process. Indeed, over the years a number of programming tools, languages and paradigms where suggested. The aim of each paradigm was to help improve on one or more of the following:

- Rapid application development (RAD)
- Maintainability
- Performance
- Reuse

Among these, the most visible improvement is in the number of RAD tools that exist today. However, in many cases the tradeoff in using these tools is in performance.
KNOW THE TECHNOLOGY

• As a software engineer, how do you make a case for a certain technology, T?

<table>
<thead>
<tr>
<th>Must say, T can help us ...</th>
<th>as opposed to, T ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>build re-usable code</td>
<td>supports inheritance</td>
</tr>
<tr>
<td>build highly maintainable code</td>
<td>supports data abstraction</td>
</tr>
<tr>
<td>build concise and well-structured code</td>
<td>is polymorphic, has templates</td>
</tr>
<tr>
<td>build cross platform software</td>
<td>compiles to a virtual machine</td>
</tr>
<tr>
<td>modify online systems</td>
<td>is dynamic</td>
</tr>
<tr>
<td>build efficient code</td>
<td>is multi-threaded</td>
</tr>
<tr>
<td>in RAD and prototyping</td>
<td>is high-level, is declarative</td>
</tr>
<tr>
<td>?</td>
<td>is strongly-typed</td>
</tr>
<tr>
<td>?</td>
<td>is reflective</td>
</tr>
<tr>
<td>?</td>
<td>is higher-order</td>
</tr>
<tr>
<td>etc.</td>
<td>etc.</td>
</tr>
</tbody>
</table>

• A manager might not care if a certain language is dynamic, but you should, if you had to build a system the behavior (logic) of which might have to be updated at run-time.

• If you can not describe a feature (say, inheritance, polymorphism, etc.) without any reference to a specific programming language, then you probably do not fully understand (appreciate the value of) this feature.
DESIGN TRADEOFFS

• There is no ‘good design’ in the absolute sense. A design can only be measured against some context.

• Given a set of requirements and a number of environmental constraints, one can judge a design.

• Any design, therefore, is about making tradeoffs.

• Some common (and important) design tradeoffs:
  ➢ **Maintainability vs. Performance !!!**
  ➢ Time vs. Space (Storage) Complexity
  ➢ Static (Compiled) vs. Dynamic Environments
  ➢ Others requirements that might effect design decisions include: open system, real-time, fault-tolerance, distributed vs. centralized control, etc.

*Can anyone (formally) prove that there is always a tradeoff between highly intuitive/maintainable and highly efficient code? Can we not in theory/principle maximize both? Why or why not?*
ANALYSIS VS. DESIGN

- S/W construction is an iterative process.

- The analysis and design tasks are not different, except for the level of detail. This view minimizes the gap between “what” the system must do and “how” it is done!

- Thus, analysis and design are best thought of as two special states in a continuum.

- In the early stages we should pay little attention to implementation details. The final stage (final design) should be “the realization of the model within a certain (computing) environment.”

- In general the model at level \( n \) should be more **stable** than the model at level \( n+1 \). For example, early analysis is not so sensitive to change as a detailed design would be if the programming language, database or other environmental variables changed.
WHY DIFFERENT PROGRAMMING PARADIGMS?

Different paradigms interpret the ‘program=data+control’ equation differently.

Some paradigms try to achieve re-use by building re-usable data objects, some by building re-usable modules.

program = data + control (logic)  
⇒ \textbf{generic}(program) = \textbf{generic}(data + logic)

Therefore, to build generic (re-usable) programs we can generalize the data, the control, or, of course, both.

- Generic data objects are attained via \textbf{data abstraction}
- Generic modules are attained via \textbf{functional abstraction}
Different paradigms interpret the ‘program=data+control’ equation differently when building small modules.

There are two competing classes of paradigms: (i) data-centric and (ii) control-centric.

In the data-centric model the building blocks (modules) are data objects, and control is implemented as various services that the data objects promise to provide to clients.

In the control-centric model data flows in and out of small modules (functions, procedures, rules, etc. – see below) that are glued together to achieve a larger task.

In this sense the object-oriented paradigm does stand on its own as a completely different way of structuring (building) programs.
The procedural, functional, logic, etc programming paradigms differ substantially in how control is attained: function composition, procedure calls, rule-chaining, etc. (they also differ in whether or not they are: strongly-typed, high-order, reflective, static/dynamic, polymorphic, lazy/strict, multithreaded, etc.)

BUT, as far as the general philosophy of program design, they all belong to the same class, i.e., they are all control-centric (as opposed to data-centric)

In the OO paradigm we structure our programs around data objects and control (intelligence) is distributed as a set of services that each data object promises to provide.
SO WHAT IS THE SOFTWARE CRISIS?

• RAD (Rapid Application Development)
  ➢ Most programming paradigms emphasize the ease of software development
  ➢ This technology has advanced tremendously (we can now develop a prototype within hours and with minimal amount of coding.)

• Maintainability
  ➢ S/W Maintenance is more costly than S/W development!
  ➢ Maximizing cohesion of and minimizing coupling between modules is the key to building highly maintainable code
  ➢ Different paradigms approach this problem in various ways.

• Re-use
  ➢ Design block-box modules (software chips?)
  ➢ Can we ever build generic (re-usable) business objects?
  ➢ Is there more hope in building generic (re-usable) functions?
**TRACEABILITY AND THE MAINTENANCE PROBLEM**

- **Traceability**: where does (should) a certain piece of logic reside?
- If a module implements too many business rules, how do we update some without effecting the others?
- Maximizing cohesion of and minimizing coupling between modules is the key to highly maintainable code
- Clearly, to simplify the maintenance task, we must do a good job of distributing the intelligence across the system at design time.

These ideas are not entirely new!

- 3-tier architecture: UI, Control, Data (Ivar Jacobson)
- Multi-tier architecture: Control must also be cleanly separated into two important levels: application and business logic.
- In large systems the challenge is in the middle layer.
THE OBJECT-ORIENTED DESIGN PHILOSOPHY

- OO is NOT about polymorphism, encapsulation, inheritance, etc. OO is a new strategy for distributing intelligence across the system so as to maximize cohesion and minimize coupling!

By distributing intelligence across the system in a data-centric fashion, we implicitly build highly cohesive (& thus re-usable) modules, without introducing unnecessary coupling.

- The question now is how do we best distribute intelligence across the system? This is the object-oriented design problem!!!
THE OBJECT-ORIENTED DESIGN PHILOSOPHY

- In structured programming there is a centralized control. 'main' controls the sequencing (and the control flow) and owns all external and global data/information.

- In OOP, there is no centralized control. Ideally, all objects are created equal and control is an interaction between objects that collaborate to achieve a certain functionality.

- A First Example. An ATM machine.

```c
Main()
{
    while ( true )
    {
        acctNo = getAcctNo();
        pinNo = getPIN();

        if ( notValid(pinNo,acctNo) ) exit(CODE);
        case (choice) of
        {
            1: deposit(amt,acctNo);
            2: withdraw(amt,acctNo);
            3: return checkBalance(acctNo);
        }
    }
}

void withdraw(amt,acctNo)
{
    b = checkBalance(acctNo);
    if ( b >= amt )
    {
        updateBalance(SUB,amt,acctNo);
        giveMoney(amt);
    }
}
```
THE OBJECT-ORIENTED DESIGN PHILOSOPHY

A FIRST LOOK

- Each object maintains its relevant data (encapsulation)
- Each object publishes a public interface that implements its agreed-upon responsibilities (services it promises to provide to clients).
- Hidden (bookkeeping or control) functions need not be exposed and are defined as private.
- Objects may provide a service but not have the implementation (these services are provided by delegation). For example, the ATM object promises to respond to a `printBalance()` message. The ATM object will most likely delegate this function to a contained object (a printer). This detail should be hidden from the user. (more on this later!)

- Can any of the above objects be re-used in another application?
EXERCISES FOR PART I

• A typical telecom company has many systems, some real-time telephony systems that automate dialing, switching, etc. and some backend systems that are used for network management, etc. In the former performance is the main requirement, while in the latter it is not as crucial. What kind of design tradeoffs you think the designers of these systems have to make?

• Suggest examples where a dynamic environment is more suitable than a static (compiled) environment.

• Discuss the tradeoff between maintainability and performance. Think of the two extremes (say assembly language vs. very high-level declarative languages.)

• Give examples of details that should not be discussed in the analysis phase, but should be left to design. Why?

• Should a programming language or the computing environment be taken into consideration when in the design phase? Why, or why not?

• What makes the maintenance problem so critical? Is that what the ‘software crisis’ is? Or is development the main challenge? Discuss in relation to very complex systems.

• Any program satisfies ‘program=control+data’. To build generic programs, therefore, we can generalize the data, control (functions), or both. Which of the two (data and control) is easier to generalize? Discuss and give some examples of business objects and rules/logic.

• What are good examples of generic data objects that are used everyday by software developers. What are good examples of generic modules (functions)?
EXERCISES FOR PART I (CONT’D)

- Define/describe without referring to any specific programming language, and discuss with examples the advantages of each of the following:
  - Inheritance - polymorphism
  - data abstraction - higher-order
  - reflection

- How is the object-oriented paradigm different from all the other paradigms with respect to program design? Discuss.

- In the SW life cycle, programming paradigms often make promises regarding ease of development (RAD), and/or maintainability. Which paradigms facilitate RAD and which make promises regarding maintainability? In your opinion, where is the real problem in SW engineering, and where do you think OO stands on that?

- What is ‘traceability’? How is it related to maintainability?

- What are multi-tier architectures? Why are they useful? How is all of this related to maintainability and re-use?

- The main idea behind OO is a better distribution of logic across the system to minimize coupling and maximize cohesion. This produces highly maintainable and re-usable code. How is that? Discuss.

- How do structured (procedural) programs lend themselves to centralized control while object-oriented programs are more easily constructed with distributed control? How does that relate to coupling, cohesion, and maintainability?

- Describe the data abstractions and functional abstractions that are needed to write a sort function that sorts data of any type and in any order. (can this be achieved with one kind of abstraction only?)
ASSIGNMENT 1

1. A typical telecom company has many systems, some real-time telephony systems that automate dialing, switching, etc. and some backend systems that are used for network management, etc. In the former performance is the main requirement, while in the latter it is not as crucial. What kind of design tradeoffs you think the designers of these systems have to make?

2. Suggest examples where a dynamic environment is more suitable than a static (compiled) environment.

3. Discuss the tradeoff between maintainability and performance. Think of the two extremes (say assembly language vs. very high-level declarative languages.)

4. Give examples of details that should not be discussed in the analysis phase, but should be left to design. Why?

5. Should a programming language or the computing environment be taken into consideration when in the design phase? Why, or why not?

6. What makes the maintenance problem so critical? Is that what the software crisis is? Or is development the main challenge? Discuss in relation to very complex systems.

7. Any program satisfies ‘program=control+data’. To build generic programs, therefore, we can generalize the data, control (functions), or both. Which of the two (data and control) is easier to generalize? Discuss and give some examples of business objects and rules/logic.

8. What are good examples of generic data objects that are used everyday by software developers. What are good examples of generic modules (functions)?

9. Define/describe without referring to any specific programming language, and discuss with examples the advantages of each of the following:
   a. Inheritance- polymorphism
   b. data abstraction - higher-order languages
   c. reflection- dynamic languages

10. How is the object-oriented paradigm different from all the other paradigms with respect to program design? Discuss.

11. In the SW life cycle, programming paradigms often make promises regarding ease of development (RAD), and/or maintainability. Which paradigms facilitate RAD and which make promises regarding maintainability? In your opinion, where is the real problem in SW engineering, and where do you think OO stands on that?

12. What is ‘traceability’? How is it related to maintainability?

13. What are multi-tier architectures? Why are they useful? How is all of this related to maintainability and re-use?

14. The main idea behind OO is a better distribution of logic across the system to minimize coupling and maximize cohesion. This produces highly maintainable and re-usable code. How is that?

15. How do structured (procedural) programs lend themselves to centralized control while object-oriented programs are more easily constructed with distributed control? How does that relate to coupling, cohesion, and maintainability?

16. Describe the data abstractions and functional abstractions that are needed to write a sort function that sorts data of any type and in any order. (can this be achieved with one kind of abstraction only?)
FOUNDATIONS OF THE OBJECT MODEL

Abstract Data Types (ADT)

- The basic idea is to design a data structure with a certain expected behavior the details of which are hidden from its users.
- For example, a Stack object might be implemented in so many ways, as an array, a list, etc. All users need to know is that the operations they expect, namely Push(), Pop(), isEmpty() and isFull() are supported and they behave as expected.

Information Hiding (Encapsulation)

- Users of a class should not be concerned with internal representation of data members or implementation details of services. If any of these have changed, users of the class should not be effected.

ON HIDING DETAILS....

Can you provide the services/behavior of a queue if you only had stacks?

```java
class myQueue {
    private: Stack stack1;
    private: Stack stack2;

    Push(Object e) {
        if ( not stack1.isFull() ) { stack1.Push(e); return; }
        else CAN-NOT-ADD-ELEMENTS-TO-QUEUE;
    }

    Object Pop() {
        if ( not stack2.isEmpty() ) { return stack2.Pop(); }
        if ( stack1.isEmpty() ) { return QUEUE-EMPTY; }
        while ( (not stack1.isEmpty()) && (not stack2.isFull()) ) {
            stack2.Push( stack1.Pop() );
        }
        stack2.Pop();
    }

    Boolean isEmpty() { return stack1.isEmpty(); }
    Boolean isFull() { return stack2.isFull(); }
}
```
THE BUILDING BLOCKS OF THE OBJECT MODEL

- Classes

A class is a data abstraction of some concept/entity (physical or abstract). The abstraction is meant to capture the characteristics (properties/attributes) and behavior common to all instances of the class. For example, all ‘cars’ in the real world could be modeled by the following data abstraction:

<table>
<thead>
<tr>
<th>Car</th>
</tr>
</thead>
<tbody>
<tr>
<td>Make</td>
</tr>
<tr>
<td>Model</td>
</tr>
<tr>
<td>Year</td>
</tr>
<tr>
<td>Color</td>
</tr>
<tr>
<td>Start()</td>
</tr>
<tr>
<td>Drive()</td>
</tr>
<tr>
<td>ChangeSpeed()</td>
</tr>
<tr>
<td>Brake()</td>
</tr>
</tbody>
</table>

All cars have a make, a model, a year of make and a color. Also, we must be able to Start(), Drive(), ChangeSpeed(), Brake(), and Stop() all cars.
GENERALIZATION (DATA ABSTRACTION)

- When we recognize commonality in characteristics and behavior we abstract these commonalties out - we generalize. For example, what we assumed on ‘Car’ applies to all vehicles. Or does it?

- Note that as we go down in the hierarchy concepts become more specifically defined. As we go up in the hierarchy, concepts become more abstract. Can you visualize a picture of the general concept of vehicle? What about the concept ‘man-made-thing’?

- Unlike classes, which are mere abstractions, objects are unique instances of classes and are assumed to represent actual entities (physical or abstract). Objects have:
  - Identity
  - State
  - Behavior
**CLASS VS. INSTANCE**

- Objects (instances of classes) have attributes that can take on various values from a given domain/type.
- We can also specify class attributes: attributes who’s value is shared by all instances of a class. In a sense, a class attribute is a global variable within a single domain (concept space).
- These class attributes are useful, as they can be used to enforce some important semantic constraints.
- We can also define class-scope operations.

In languages like C++ and Java class variables are implemented as static data members.
In languages like Smalltalk and CLOS (Common LISP Object System) there are class variables (members) and instance variables (members) that are defined explicitly.

**Meta classes (abstract classes)**

- Classes that are defined as templates to other classes are called meta classes (or abstract) classes.
- These classes can not be instantiated but are sub-typed.
- The value of these classes is that they enforce a certain prototype (structure) and behavior (methods) that all their subtypes must support.
ON OBJECT IDENTITY AND EQUALITY

Importance of object identity

- Objects must have a unique identity. This could be just a memory address which defines the actual space where the state of that object is stored. However, sometimes an actual memory address is not enough since that could change with different executions.

- Suppose we are building a system for the maintenance of some equipment – say parts/components of an air fleet. Components typically have a life-time, and based on different sensory data they might be removed and replaced by other equipment. When some equipment are removed they are either taken for maintenance or simply scrapped. Equipment might enter the system again, and we must recognize when two equipment objects are the same.

- To maintain the state of an object beyond the scope of one execution objects must be persisted. (more on this and the issue of mapping objects to relational databases through a data layer later.)

Equality

- When are two objects equal? Clearly, if they point to the same memory address then they are the ‘same’. However, two objects could have two different memory addresses but still ‘denote’ the same actual object. Thus we have to define what it means for two objects to be ‘equal’. For example, we have to decide what it means for two Cars to be equal. This is clearly context/domain dependent.

Canonical Form

- Unlike primitive data types that come equipped with constructors, assignment, inequalities, etc. for the abstract data types that we define these operations must be defined. If these basic operations are defined, making these objects like any other object in the system, we say we have defined objects in canonical form. This is a safe practice!
The object-oriented paradigm is based on ideas that have been studied by logicians and philosophers for years. It was Aristotle who first suggested that the definition of a concept has two parts: the genus and the extension. The genus is the set of properties that are generally true of a certain category. It is what makes an object of a certain type. For instance, the genus of Human must contain ‘thinking’ since it is one of the main characteristics that define what a human is. In logic this has become known as the intension of a concept.

Thus, one could think of the intension of C as the set of properties that are generally true of C. The extension of C, on the other hand is the set of instances that the intension of C is true of. In fact, this is how sets are defined. For instance, the set \( \{ x \mid \text{Talk}(x) \& \text{Walk}(x) \} \) defines the set of things that walk and talk. The set is the extension, defined by two properties (intensions).

One could think of classes as sets defined by properties, and elements in the set as the objects (instances). The notion of Subset here becomes Subtype. In the object-oriented world, therefore, the intension is the class definition, and the extension is the set of objects (instances). For two objects to be equal they must be intensionally equal, i.e., they must have the same values of all the their properties.

Intensional equality is very important in Computer Science

We all learned in mathematics the following: \( \forall x ((f(x) = g(x)) \Rightarrow (f = g)) \). That is, if two functions produce the same output for the same input then they are the same functions. Is this the whole story? Consider:

\[
\forall s (\text{quickSort}(s) = \text{insertionSort}(s))
\]

That is, for any input sequence s, both insertion sort and quick sort produce the same output. Are these two objects the same? Clearly not. While they are extensionally the same, intensionally the differ: they have different PROPERTIES. For example, they have different definitions, different time and space complexities, etc. Thus, two objects are equal if they are ‘intensionally’ equal.
HIERARCHIES

A Hierarchy is a ranking or an ordering of abstractions (Booch)

The “isa” (or Inheritance) Hierarchy

- If X “isa” (kind of) Y then X inherits the properties and behavior of Y.
- X is said to be a specialization of Y (Y is a generalization of X)
- X is specialized by adding properties and/or overriding behavior.
- The “isa” relationship is transitive: isa(a,b) & isa(b,c) => isa(a,c)

Inheritance is one way we can achieve polymorphism, which is the ability to treat a set of objects of similar types the same.

We need to able to treat a collection (an array, a list) of employees the same, regardless of the specific type of each employee.

Engineer inherits all properties and behavior of Employee while ResearchStaffMember adds properties and Manager adds properties and overrides behavior.
INHERITANCE AND POLYMORPHISM

- The whole idea behind inheritance and polymorphism is to be able to write simple and intuitive (thus maintainable) code.
- For example, consider the following pseudo code that iterates through all employees and decides who should be a candidate for a new director opening:

```java
List<Employee> getListOfCandidatesForDirector( List<Employee> allEmployees )
{
    List<Employee> candidates = makeNewList( );

    ForEach( emp in allEmployees )
    {
        if ( emp->Position != DIRECTOR )
        {
            if (emp->SuggestPromotion( ) == DIRECTOR )
                candidates->addElement( emp );
        }
    }

    return candidates;
}
```

- Note the code assumes each type of object “knows” how to compute its own `SuggestPromotion()` function/method.
- For objects of type `Engineer` and objects of type `ResearchStaffMember` the default method defined in `Employee` will be executed. For objects of type `Manager`, on the other hand, a different method is executed. All these details however are not our concern in the code above.

Where’s the big advantage here?
- If the designers of the `Engineer` class decided that engineers should have a different (special) way of computing `SuggestPromotion()` they could override the default method and the above code will not be effected at all!!!
Which `ComputeSalaryIncrease()` method is executed in this code?

```cpp
TechnicalManager* dan = new TechnicalManager();
dan->ComputeSalaryIncrease();
```
MULTIPLE INHERITANCE (cont’d)

- When objects are constructed an instance of a class (which is simply a template definition) is created. This involves allocating the appropriate memory and creating all the inheritance layers.
- To achieve inheritance, the memory allocated is layers of all of the structures in the type hierarchy.

- D* d = new D();
- How many objects have been created here?
- Note that we expect all these to be valid:

  d->fb(); // since d inherits from B
  d->fc(); // since d inherits from C
  d->fa(); // since d inherits from C and B
  d->ga(); // since d inherits from C and B

- Which method is executed here d->fa();
- Since an object needs the structure of its superclass to achieve inheritance, the construction of an object triggers the construction of an object of its superclass and so on until the root object.

Questions

- Now, how many objects would be created if we construct an object of type D.
- Also, how do we resolve the ambiguity?
- Different languages deal with the problem of multiple inheritance differently!
- Is multiple inheritance really needed?
Like inheritance, containment, which is a relation between objects, forms a hierarchy known as the “part-of” hierarchy.

Also like the “isa” relationship, the “part-of” relation is transitive: partOf(a,b) & partOf(b,c) => partOf(a,c)

Unlike inheritance, the semantics of this relationship are not supported directly by OO languages. Programmers must code the corresponding logic!

**Physical Containment (Composition) vs. Containment by Reference**

- If B can be referred to outside of A, A should contain B by reference
- If B is not known outside of A, it should be physically contained in A
- If A is not an A without B, we probably have physical containment

Note that when an object is contained by reference it can be removed without changing what the containing object is. The same is not true in the case of physical containment (a phone is not a phone without a dialer; a car is not a car without an engine; etc. but a plane is still a plane even when the tape recorder is removed.)
Unless overridden, we like to assume the same font, margins, etc of the document object in all the contained objects. This passing of information is some form of inheritance.

Unlike inheritance that comes with the “isa” hierarchy the logic of the containment relationship is not built into OOPLs.

This is typically a function of the application and the semantics of the attributes. For example, the location attribute of a Car object and that of its Engine must be the same. A Car and its Engine need not have the same weight, or manufacturer, however.

Note the recursive containment (more on this later…)

We’ll discuss cardinality constraints (and other kinds of semantic constraints) below....
LETTHEREBEOBJECTS

Basics
• Objects come to life using a special type of function/method, called
the constructor, which is defined on every class.
• A constructor must properly initialize the state of the object
• There can be several constructors each assigning the initial state of
the object based on the context.
• There must always be a default constructor (an empty context), since
temporary objects might be created implicitly by function calls
without explicit call for a “new”.
• A constructor can be used to enforce some invariants (predicates or
conditions on attribute values.)
• Invariants can be used to enforce some semantic constraints.

Interaction with Containment and Inheritance
• An object’s constructor must force a cascade construction on all
objects that it ‘physically’ contains. For example, when someone
constructs a Phone object they should be able to safely assume that
the phone’s Dialer is not null, but that it has been created. One
should not assume that an airplane’s TapeRecorder has been
instantiated by simply constructing a Plane object.
• The constructor of an object can initialize the state of the object by
‘partly’ using the constructor of its superclass.

Sadly, we still have things like ....
• In languages like C++, where the programmer allocates and cleans-
up (deletes) memory explicitly (we haven’t advanced much have we?),
objects must also have a destructor.
• Worse yet, in C++ destructors must (sometimes) be declared ‘virtual’
to ensure that the object itself and not only its base class are deleted.
• For example, we do not want the delete method of Employee to be
called in ForEach ( emp in List<Employee> ) emp->delete but the specific
delete of each type of Employee. (this is C++ specific and need not
concern us too much but it makes us understand how polymorphism
was “hacked” in some languages!)
DELEGATION AND THE "USES" RELATIONSHIP

- To perform a service an object might need to "use" the services of one or more object(s).
- An object might even promise a service that is provided (i.e., implemented) entirely by another object. (this is called delegation)
- The "uses" relationship is very important and objects can use the services of other objects in various ways.
- For an object to use another it must have access to it.

- In performing f() ObjectX uses the service g() provided by ObjectY.
- Here we have access through containment (physical or by reference).

```java
ObjectX
ObjectY y;
Etc...
f()
{
    y.g();
}
```

```java
ObjectY
A a;
Etc...
g();
```

```java
Car
BrakeSystem b;
Brake() {
    b.DoBarke();
}
```

```java
BrakeSystem
DoBrake() {
    // implementation of brake
}
```

```java
Document
PrintDriver& p;
Print() {
    p.DoPrint();
}
```

```java
PrintDriver
DoPrint() {
    // implementation of printing
}
```

```java
Image
PrintDriver& p;
Print() {
    p.DoPrint();
}
```

```java
uses
```
MORE ON THE "USES" RELATIONSHIP

Various ways of implementing "uses"

- Besides containment (physical or by reference), an object can have access to another object by parameter passing, through global data (not good!), or by locally creating instances, using them, and then deleting them (this is really using!)

INHERITANCE and DELEGATION

- These two notions are often confused since in both cases an object claims it can provide a service it does not actually implement but is a service that is implemented elsewhere.
- This can be done in two ways: an object can provide a service that it does not implement if it can obtain by inheritance or if it can delegate the work to some contained (proxy/agent) object.
DELEGATION AND RE-USE (LOOKING AHEAD)

- Why did we implement some behavior outside of an object although conceptually the object itself provides the service?

  The reason is modularity and re-use.
  Taking the well cohesive functionality of printing outside of the Document and Image objects results in a re-usable printing object.
  The same is true in the case of a BrakeSystem which could be used now by other objects.
  The design question, however, is what is the correct ‘granularity’ at which functionality should be factored out for re-use? And what is the tradeoff of this factoring-out?
  The tradeoff is that, as far as the conceptual model, too many collaborations between objects increase coupling (dependency).
  Computationally, too many collaborations increase the number of messages sent back and forth and thus effects performance.
  We will turn to these design tradeoffs later....
Some Modeling Heuristics

- (2.1) All data should be hidden within its class.
- (2.2) Users of a class should be dependent on its public interface, but a class should not be dependent on its users.
- (2.8) A class should capture one key abstraction (cohesion)
- (2.9) Keep related data and behavior in one place (same as above)

- (3.1) Distribute system intelligence horizontally and vertically as uniformly as possible. (?) (It might be up to the model!)
- (3.2) Do not create God classes. (discuss in class)
- (3.6) Model the real world whenever possible
- (3.9) Do not turn an operation into a class (in practice, a good heuristic but sometimes it is a good techniques.)

For example, by wrapping functions in class definitions we can pass functions around as arguments and returns them as values. This allows us to write higher-order methods (functional abstraction).

```java
class LE {
    boolean run(int i, int j) { return (i <= j); }
}
class GE {
    boolean run(int i, int j) { return (i >= j); }
}
LE le = new LE();
GE ge = new GE();
sort( List<int>, le );
sort( List<String> ge );
```
**Functors (Basis of Some Design Patterns)**

Functors are ‘Function Objects’

```java
class BinaryOperation {
    public Object apply(Object o1, Object o2) {return new Object();}
};

class And extends BinaryOperation {
    public Object apply(Object o1, Object o2) {
        Boolean b1 = (Boolean) o1;
        Boolean b2 = (Boolean) o2;
        return (new Boolean(b1.booleanValue() && b2.booleanValue()) );
    }
};

class Plus extends BinaryOperation {
    public Object apply(Object o1, Object o2) {
        Integer v1 = (Integer) o1;
        Integer v2 = (Integer) o2;
        return (new Integer(v1.intValue() + v2.intValue()) );
    }
};

// this higher-order function can be used to perform n
// binary operation on various types of objects in a vector
public Object foldr(Vector objects, BinaryOperation op, Object id) {
    Object result = id;
    for (int i=0; i<objects.size(); i++)
        result = op.apply(result, (Object)objects.elementAt(i));
    return result;
}

Integer plusID = new Integer(0);
Boolean andID  = new Boolean(true);
Object andList = functor.foldr(bools, and, andID);
Object sumList = functor.foldr(numbers, plus, plusID);
```
MORE MODELING HEURISTICS

- Later we will see a design pattern (the “Strategy Pattern”) that is suggested specifically for cases where we need functional abstraction. This should tell us that OO is not a science, as some good design heuristics and design patterns might say conflicting things. The trick is to know what is best in a certain situation!

- (4.1) Minimize number of classes with which a class collaborates (removes unnecessary coupling)
- (4.2) (if 4.1 an not be avoided) Minimize the number of message sends between a class and its collaborator
- (4.5) If X contains Y messages should be sent from X to Y not the other way (i.e., containment should always imply “uses”.)
  note: for performance we sometimes link objects both ways! (recall our tradeoffs?)

- (4.13) A class should know (about) what it contains but it should not know who contains it. (very important for re-use)

- (5.1) Inheritance should be used only to model specialization
- (5.2) Base classes should not know about their derived classes

(5.2) is questionable, both conceptually and computationally! The whole idea behind a generalization is that it is borne out of a realization of commonalities between classes, thus the base concept is designed with those classes in mind.) Nevertheless, the point of the heuristic is that when designing a class you should worry about capturing the right abstraction and not how some other class might use this class!

This next heuristic says exactly that, which somehow contradicts (5.2)

- (5.8) Factor the commonality of data, behavior, and/or interface as high as possible is the inheritance hierarchy.
When we classify concepts we usually do so based on some attributes. For example, we can classify humans by gender, creating two subtypes Male and Female, or by age creating subtypes such as Child, Adult, etc... The decision as to what attribute should the classification be based on is usually application-dependent. Discuss this issue with an example (pick a general concept and classify in two or more ways and justify each classification.)

Discuss the importance of an object’s identity in real-world applications.

Discuss equality and other canonical operations between objects. For example, what might these mean: car1=car2. How about car1 \geq car2, car1+car2, etc?

How are the two concepts of inheritance and polymorphism related? Can we have one without the other in a language like C++? How about in general?

Give an example of how inheritance and polymorphism can be used to write intuitive and maintainable code.

What is multiple inheritance and what are some of the technical problems associated with it? Is it necessary to have multiple inheritance?

What is containment? Discuss the two different kinds of containment.

How is containment related to delegation and the “uses” relationship. Give examples.

Discuss with examples different ways the “uses” relationship can be implemented.

Discuss with an example the relationship between delegation and re-use.

Objects have data (a state) and operations should not, in general, be turned into classes. Can you give an example that provides an exception to this heuristic?

How is heuristic (4.13) important for re-use?
ASSIGNMENT 2

Below you are given requirements for a new design, some design constraints, and the deliverables that you must provide on the due date. Read the requirements carefully as well as the set of design constraints. You’ll be judged by how well your design meets the requirements according to the given constraints.

Requirements

You are to design an elevator system for large buildings. These buildings have a large number of floors, say $F$, and a number of elevators, $E + 1$, where $E$ elevators are divided among $F$ floors and one elevator is used as a service elevator and in emergency situations only. For example, we might have a building with 20 floors and 5 elevators. Elevator 1 might be assigned to floors 1 through 5, elevator 2 is assigned floors 6 through 10, etc. Our goal is to design a reliable/fault-tolerant system that also optimizes the average waiting time of passengers. These elevators will be used in large office complexes where peak hours are mornings (where most requests are from first going up), lunch time and between 4:00 and 7:00pm (where most requests are from upper levels to first floor).

An emergency situation is one where an elevator goes down. The service elevator goes to all floors, and if more than one elevator is down, the service elevator can be used for all floors that are serviced by these elevators. Clearly, however, priority should be given to the upper-level floors and for requests going in the 'up' direction (taking the stairs down is easier than taking the stairs up, and walking from floor 1 to floor 5 is easier than walking from floor 1 to floor 20!)

Heavy items (e.g., equipment, purchases, office furniture, etc.) are typically delivered during off hours and on Weekends, but could sometimes be delivered during working hours. Maintenance and service during working hours is done using the service elevator, although during off hours other elevators might be used.

Finally, although this particular system is designed for large office buildings, some aspects of this system might be used in commercial buildings where off hours and Weekends are actually the busiest (the role of the service elevator is reversed here.)

Constraints

- The budget allows for elevators to be equipped with specialized hardware.
- Performance (average waiting time) and reliability are major concerns.
- Data storage is not an issue.
- Any special technology that facilitates meeting the requirements can be used.
- Components of the design might used in other settings where the above might not all be available, so your design should assume a reasonable common denominator.

Deliverables

- A detailed object model, showing all the key abstractions and the important (class and object) relationships. Cardinality and key semantic constraints must be shown.
- A detailed description of the structure of all key abstractions.
- A detailed CRC (Class Responsibility and Collaboration) card for all the key abstractions. This must detail the behavior of objects and include all collaborations (client/server).
- At least 5 use cases (here’s an obvious one: ‘passenger makes a request’).
- A detailed trace of 3 use cases in an interaction (sequence) diagram, including one for the ‘passenger makes a request’ use case.
- A state diagram for an elevator object.
A model is an abstract/conceptual view/representation of some part of the world.

ALL of what we do in computer science is build models, since all the systems we build are mere representations of the real thing. In a machine, all we have are 0’s and 1’s. There are no trains, missiles, corporations, alarms, etc. What we have are models/representations of these.

Are there limits to what we can model in a digital machine?
What concepts/notions are difficult to model?

I claim that this symbol encodes all the knowledge in the world:

\[ \ell \]

Models can be
- very abstract, or very detailed
- formal or intuitive/informal

Examples of formal modeling languages:
- Logic (first-order, modal, temporal, fuzzy, default, etc.)
- Set theory
- Probability theory
- Vector algebra
- Boolean algebra
- Relational algebra
- Geometry

Some of these formal languages (in reality a combination of them) are used to model things like information/data, electronic circuitry, businesses, economies, cars, traffic lights, even the human mind.
ON FORMAL METHODS IN S/W ENGINEERING

- Formal modeling techniques clearly have an advantage over informal modeling techniques in that they allow us to
  - Systematically develop algorithms to implement the model
  - Automate the development and verification/testing of the system
  - Transform from one representation to another using meaning-preserving transformations between two formal systems
  - Analyze and study various properties about the system more easily, such as complexity, potential for concurrency, etc.

- However, the formal systems that are at our disposal today are very limiting in terms of their expressive power to model a wide variety of natural phenomena.

A dilemma!

One reason object-oriented database systems have not been able to overtake relational database systems (with all the academic, commercial and business backing of object technology) is that RDBMS have a formally simple and elegant query language, namely relational algebra (SQL). No similar language exists in the object world. That is, there’s no well-defined algebra for objects! In many ways, OODBMS are a step backward, they take us back to the days of networked models where we processed a query by traversing the physical structure of the database (pointers and all!)

So where do we stand?

The truth of the matter is that a lot of S/W engineering is still in the realm of magic and witchcraft. We are still producing code that can not be reasoned with (that is the maintenance/software crisis).

Formal methods are superior, but we do not (yet) have systems that are expressive enough for general purpose S/W development. So for now all we have are heuristics, guidelines, methodologies, etc. God (who says what’s right and wrong) is not logic and mathematics today, but the gurus, the methodologists, and there are plenty of those Gods around!
Object Modeling and Analysis

- Object modeling is not based directly on any formal system, but has elements from a number of modeling techniques (see Booch’s attachment for a historical look on how the object model was based on ideas developed in psychology, cognitive science, artificial intelligence, database systems, and type theory.)

- “In analysis we seek to model the world by discovering the classes and objects that form the vocabulary of the problem domain” (Booch)

- Suggested places to look for candidate classes and objects in the problem domain (extract from all nouns things like):
  - Tangible things
  - Roles
  - Events
  - People
  - Places

- Identifying classes and objects is not an easy task. This process involves discovery (from domain), invention (agent objects that might be used for control) and elimination (removal of previously identified objects that turn out to be irrelevant to the system.)

- Two very powerful tools that are used early in the analysis phase are Use Cases and what are known as CRC Cards (CRC stands for Class Responsibilities and Collaborators)
USE CASE ANALYSIS

- A use case is any pattern of usage or scenario that begins when some user of the system (called **actor**) initiates some transaction, request or functionality that the system is supposed to provide.

- Actors are users of the system. They could be other sub-systems.
- Use cases must cover all what the system must do.
- Selecting use cases with the right granularity is more difficult than it might first appear.
- Use cases can also be related to each other: a use can use or specialize another use case.
- Too general (too encompassing) use cases tend to make the system broken down in an action-oriented (procedural) fashion.
- Too specific use cases are unmanageable.

- We will see how uses cases can be a useful tool and discuss some heuristics to decide on the right granularity of a use case.
CRC CARDS

- CRC Cards are a very powerful tool that can help us distribute intelligence (functionality/control/logic) across the system based on the contract model.

- For each class we define a CRC Card that looks like the following:

<table>
<thead>
<tr>
<th>class</th>
<th>responsibilities</th>
<th>collaborations</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>Method/Service 1</td>
<td>A (Client), B (Server)</td>
</tr>
<tr>
<td></td>
<td>Method/Service 2</td>
<td>C (Client), B (Server)</td>
</tr>
<tr>
<td></td>
<td>Method/Service 3</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>Etc...</td>
<td>Etc...</td>
</tr>
</tbody>
</table>

- This CRC Card already achieved the following:
  - The designers of class G know what services they must provide (i.e., implement)
  - They designers of class G know that in order to provide service 1, for example, they have to understand the public interface provided by B since they have to ask for a service that B provides.

Recall that the process is iterative

- The methods/services that G must provide need not be specified in detail at first. Gradually, however, one starts to specify the input arguments and return values as well as their types.

- If done right a CRC Card can be a complete specification that is sufficient for a programmer to implement the class!!!

- The designers of G can also assume that their servers will provide the services promised and thus completely test “their” design.
**Sequence Diagrams**

- Also called Event Traces, Sequence Charts, are used to trace (incrementally in some detail) the execution of a “use case” as a collaboration between objects.

```
class Obj1 {
    Objn obj;
    obj → func(int i, A a)
};

class Objn {
    public Obj func(int i, A a)
};
```
With the tools at our disposal, let us suggest the following process:

- Identify as many key abstractions (classes) as possible
- Fill a CRC card for each class
- Identify a number of use cases
- For each use case, develop a sequence diagram

**An Example**

We are to design a very simple watch that has three controls. The watch shows the time and the date and has a timer. The first control (button) is used as follows: 1 press toggles between time and date, and 2 consecutive presses are used to switch to edit/set mode. Once in edit mode, a single press switches from hour to minute to seconds in a circular fashion. Button 2 is used to increment, while button 3 is used to set the timer.

How to proceed?

- Some obvious abstractions: **Watch, Timer, Button, Time, Date**
- Draw the object model (identify possible collaboration)
- Some obvious use cases:
  - users sets the time
  - users sets the date
  - users sets the timer
- Fill-up the CRC Cards of the key abstractions
- Re-iterate, if needed
- Start tracing the use cases in sequence diagrams
- Iterate and specify more details in each iteration
Do we need a Display object?

We do not need to decide at this point whether or not the containment we have is physical containment (composition) containment or by reference.

We can identify most sources of coupling at this point. For example, TimeManager must somehow interact with Timer, Button must interact with Watch, etc.

The specific nature of this interaction can be left for later.
CRC CARDS

**Watch**

**Responsibilities**
- SetHour()
- SetMinute()
- SetYear()
- SetMonth()
- SetDay()

**Collaborators**
- TimeManager (server), Button (client)
- TimeManager (server), Button (client)
- DateManager (server), Button (client)
- Etc...

**TimeManager**

**Responsibilities**
- IncrementHour()
- IncrementMinute()
- IncrementSecond()

**Collaborators**
- ? DateManager (server), Watch (client)
- ?

**DateManager**

**Responsibilities**
- IncrementYear()
- IncrementMonth()
- IncrementDay()

**Collaborators**
- ?
- ?

**Timer**

**Responsibilities**
- Reset()
- Start()

**Collaborators**
- ?
- ?

**Button**

**Responsibilities**
- Press()
- DoublePress()

**Collaborators**
- Watch (?)
- User (?).
Let us try to trace the use case “user sets time”

- User presses button 1 to select time
- The watch object must go to time manager to get and display time
- A timer is set to wait for a double press
- User double presses button 1 to go to edit mode
- Watch is now in edit mode of time
- User now double presses button 2 to select hour, minute, or second
- User now in a loop continuously single presses button to increment
- With each increment, Watch sends a message to TimeManager

NOTES

- Many details are missing. But, remember, this is the first iteration!
- Was the use too general? If the complete sequence diagram is too complicated (too many objects and too many messages) than yes. This is not yet ready for coding.
- Later, arguments, return values and their types, as well as timing issues, exceptions, etc. must be specified in detail.
SOME DESIGN TRADEOFFS

- It makes sense to build a `TimeAndDateManager` object, instead of the two objects `TimeManager` and `DateManager`.
- This will simplify the model, reduce traffic, and improve performance.
- Are there any tradeoffs here? What do we lose by choosing this route?

HINTS

- If we were to build another watch, can we use any existing components?
- What if we like to use `TimeManager` in the design of a radio alarm?
- Which design is easier to maintain?

OTHER QUESTIONS

- Should we have defined a display object? The functionality of the display is now buried with that of the Watch. We can not re-use that functionality, which might be useful in other watches or even in other systems (alarm clocks, etc.)
- Should we a different class for each button, like we suggested here?
- Can these buttons be re-used?
- Is the Watch object a “God Class” - i.e., does it contain most of the logic while other objects are merely data structures with no behavior? If so, then we ended up with a structured design.
- What about combining the `TimeManager` and `DateManager` objects?

All these questions can now be asked, and depending on what our criteria is (requirements, preference for performance over maintainability or vice versa, etc.) we can re-iterate, make some changes, and/or spell out some more details!!!
EVALUATING YOUR INITIAL MODEL

In most cases iterating over your model involves re-visiting your object model. In OO systems, the object model drives the distribution of logic in the system and thus is the basis of your design.

Some Classification Heuristics (5.12, 5.13)

- When classifying objects remember to classify based on behavior
- If you have case statements on the type of objects, re-think the hierarchy
- In general, if you have many case statements on any attribute, then that attribute should probably be the basis of your classification

```
Employee::promote()
{
    case (employeeTYPE)
    {
        TECHNICAL : {...}  
        MANAGEMENT : {...} 
        ETC. 
    }
};
```
More Modeling and Analysis Tools

Monitoring the States of an Object: State Transition Diagrams

- The state of the object is a snapshot of the object’s data at some point in time.

![State Transition Diagram for an Elevator](image)

- Note that it is up to us to capture the states of interest. Here we took the level (the elevator is on) and the directionOfMovement (say it is 0=idle, 1=up, 2=down) to be important.

- Other state information (i.e., attributes), e.g. number of passengers currently in the elevators, number of pending requests, etc. are not important here, although they could be in some other situation!!!
MORE ON THE STATE OF AN OBJECT

- The state of an object should only be changed through the public interface, since otherwise the behavior of the system can not be properly predicted/analyzed.

- A generally excepted heuristic: we should not classify based on the dynamic behavior of the objects (i.e., based on some specific states).

  ![Diagram](Diagram.png)

- However if the entire logic in the system is based on the state of an object then the classification should be based on that state.

- For example, in some decision support (expert) systems developed to monitor and diagnose some equipment the entire logic might be a function of the current state of an object (equipment.)

- There are two techniques to deal with these situations: a design pattern called the ‘state pattern’ and integrating data-driven rule-based techniques with the object model.

- We will discuss these examples later.
AN EXERCISE IN ANALYZING DYNAMIC BEHAVIOR

- Draw a state diagram to describe all the state transitions of this Dialog object, from which another dialog might be opened.

- Any window can be closed by pressing OK or Cancel
- Cancel is basically do nothing
- OK in window 2 should not force a save, since we can cancel in 1
- The semantics of this object are quite simple. However, we could have a number of other fields that might also depend on each other.
- Consider both cases where window 2 is modal and non-modal.
INVARIANTS AND SEMANTIC CONSTRAINTS

Semantic Constraints

- The semantics of an object are the meanings we associate with the attributes (or data members) of an object and the semantics of the object’s methods.
- In some applications it is necessary to define a number of constraints on the data members of an object.

```java
class Employee {
    ...
    int mAge;
    String mName;
    float mSalary;
    Address mAddress;
    int mYearsOfExp;

    Employee(int age, String name, Address& address )
    {
        // enforces a semantic (integrity) constraint
        if ( age < 16 )
        {
            cerr << "Invalid age....";
            raiseException(this);
            ...
        }
    }

    ....
    public setSalary( float salary )
    {
        // enforces a business/corporate policy
        if ( (mYearsOfExp > 5) && (salary < 35000.0) )
        {
            raiseException(this); etc...
        }
    }
};
```
INVARIENTS

- An invariant is a condition/relation that must always hold (be true).

```java
class Rectangle {
    ...
    int   mHeight
    int   mWidth;
    int   mArea;

    // the area of a rectangle must always be height*width
    // whenever we update either one we must re-compute the area

    public void setHeight( int h )
    {
        mHeight = h;
        mArea = mHeight * mWidth;
    }

    public void setWidth( int w )
    {
        mWidth = w;
        mArea = mHeight * mWidth;
    }
};
```

- Maintaining these invariants complicates the code.
- Note also that we are assuming that we have captured all the ways someone can get at the attributes. For example, if someone can update mHeight or mWidth above without going through the public interface we can not guarantee that the invariant is enforced!
- The really big problem however is that invariants might involve not just simple attributes, but attributes that are themselves objects.

- We will discuss these more complicated situations below....
CARDINALITY CONSTRAINTS

Manager
List<Employee> mEmployees;
List<Project> mProjects;
Department& mDepartment;

Manager( List<Employee> emps,
List<Project> projs,
Department& dept )
{
    if ( (emps.size() < 3 )
    {
        // raise an exception since a manager
        // must have at least three employees
    }
    if ( (projs.size() < 1) || (projs.size() > 5) )
    {
        // raise an exception since a manager must
        // have at least one but at most 5 projects
    }
}

public void addEmployee(Employee& emp) { enforce constraint }
public removeEmployee(Employee& emp) { enforce constraint }
public void addProject(Employee& emp) { enforce constraint }
public removeProject(Employee& emp) { enforce constraint }

- You probably observed by now that enforcing invariants, semantic
  and cardinality constraints and other complex business rules
  complicate the code considerably and in a non-declarative nature.
- Moreover, there are no disciplined way to ensure that we have
  captured all the possible places that the constraints must be checked.
- Fortunately, there are ways to clean up this mess, but we have to
  combine another paradigm with OO (more on this later!)
AN INTERIM RECAP

Let us put what we know so far into use...

Let us design an elevator system that services a large building. Say we have some E elevators servicing F floors. The elevators are divided into groups that service specific floors. For example, elevators 1, 2 and 3 service floors 1 to 5 only. Another group will service floors 6 to 10, etc. Of course, all elevators go to first, but some do not go beyond a certain floor. The goal is a reliable, fault-tolerant elevator system that minimizes the waiting and the service time of every request.

Here are some requirements/constraints that place some boundaries on your design tradeoffs:

- It is ok to assume a design that will result in some expensive processors installed on any (or even all) elevators. Performance and reliability/fault-tolerance are more important.
- You can assume that one elevator is used only as a service elevator which could be relied upon in emergencies.
- You can assume that space for storage of data is not an issue, but you have to remember that message traffic is expansive and can hinder performance, which IS an issue!
- Data on the building, its population and the traffic patterns at different times during the day (for the past several years) ‘will be’ supplied.

Hints

- Remember the design tradeoffs.
- Exploit all that has been granted for you to the limit (its paid for!)
- Ask a lot of ‘what if’ and ‘why not this’ questions.

Deliverable: Object Model, Use Cases, and Sequence Diagrams

Deadline: YESTERDAY!
THE ELEVATOR SYSTEM

The following are things you should come up with on your lunch break. You could start writing these things (like any other software engineer) on high-tech media such as napkins or the back of envelops. That’s the genesis of your design.

Key Abstractions:
- Elevator
- Floor
- Passenger
- FloorRequestPanel
- ElevatorSelectionPanel
- Etc...

Obvious Use Cases:
- Passenger requests an elevator
- Passenger selects (a) floor(s)
- Elevator stops at floor no. x
- Passenger blocks door
- Passenger stops elevator
- Etc...

Obvious Relations:
- Every floor has an elevator-request panel
- Every elevator has a floor-selection panel
- Etc...

Obvious Object Attributes (the case of an Elevator):
- Moving/Idle?
- Direction (if moving!)
- Number of passengers
- Number of pending requests (is that an elevator attribute?)
- Etc...
EXERCISES FOR PART III

- Images, sounds, pictures, poems, paintings, books, scripts, just about everything that the human mind has created can be digitized and stored in machines (some digital library projects are trying to do exactly that!) This means that all human knowledge can be stored in binary (i.e., as a very, very large sequence of 0's and 1’s). This large sequence is essentially a number, albeit a very large number. If we represent this number by some symbol, say \( \partial \), then we have encoded (or represented) all of human knowledge by one symbol.

Do you agree with this argument? Why or why not? Explain your position on what can or can not be represented and modeled.

- Suppose we model a class as a set. Than a subclass can be though of as a subset and an instance as an element of a set. For example, the Vehicle class and its subclasses could modeled as follows:

\[
\begin{align*}
Vehiclet & = \{ v \in U | \exists m(Make(v, m)) \land \exists c(Color(v, c)) \land \exists y(Year(v, y)) \} \\
Car & = \{ c \in Vehicle \mid C(c) \} \\
Truck & = \{ t \in Vehicle \mid T(c) \}
\end{align*}
\]

That is, a vehicle is the set of things that have a make, a color, and year of make. A car is a set of vehicles that satisfy some additional properties that we group as \( C \), and so on. This might be one way to formally describe an object model. However, so far we have only modeled the static part of an object model. Suggest how we might represent the behavior (services/methods) of classes in this scheme.

- Selecting very general use cases could result in a structured (action-oriented) design. How is that? What does selecting very specific use cases result in?
EXERCISES FOR PART III (CONT’D)

- Complete the Watch example we discussed above. Provide a detailed object model with all relationships identified, the uses cases, the CRC cards and the sequence diagrams for every use case. Make notes of some alternative design decisions that you abandoned – they could be important to look at when evaluating your final design.

- Provide a detailed state transition diagram (STD) for a typical elevator from the elevator system example. Is the state model important here, or is the sequence (or collaboration) diagram a more important tool in this example?

- What are all the semantic constraints and invariants that in real-life might be placed on the various objects in the elevator example? (Hint: number of passengers or total weight in an elevator, etc.) How would you enforce all these constraints?

- Suggest what the cardinality constraints on the various relationships in the elevator example should be.

- In the same example, can you come up with some more complicated rules that must be enforced involving attributes in more than object? How would you define those?
SOME DESIGN HEURISTICS

• A class must know of objects it contains, but it should never be aware of who might contain it.

![Diagram of object relationships]

If this object has logic that is based on the fact that it is contained in a watch then it would not be easy to (re)use this object in an alarm clock or a calendar, or simply another watch.

Here’s a design problem:

• We suggested that in order to build cohesive and re-usable modules Date and Time must belong to two separate abstractions.

• Also, according to what we said earlier about containment, the Watch object must physically contain TimeManager, since a Watch is not Watch without a TimeManager. However, the same is not true between Watch and DateManager and thus the DataManager object must be placed outside the scope of Watch.

• In a watch that maintains the date, the TimeManager must somehow trigger the change of date. However, we would not like the TimeManager to know about a DateManager since in general we like the TimeManager object to built in such a way so that it can be reused in a Watch that does not maintain the date. How do we do that?

• In other words, how do we remove the coupling between TimeManager and DateManager?
SOME DESIGN HEURISTICS (cont’d)

Here’s the problem:

- The TimeManager object must not know of any DateManager object, again for re-use purposes.
- In fact, according to the first (and sound) heuristic mentioned above, the TimeManager must not have any knowledge of the structure of its containing object, and thus should not be aware it is part of a watch that maintains the date.
- How then, can the TimeManager trigger a date increment?

Here’s the solution:

- The meta-knowledge about the structure of the Watch must be part of the Watch object only.
- Therefore, it is the Watch object that is responsible for updating the date based on data obtained from the TimeManager.

```java
class Watch {
    TimeManager tm;  // physical containment
    DateManager& dm;  // containment by reference

    void StartWatch()
    {
        while ( true )
        {
            time = tm.getTime();
            reDisplayTime(time);
            if ( time == <24:00:00> ) dm.incrementDate();
        }
    }
};
```

- In general, these kinds of design problems can not be solved this easily. There are however design patterns and rule-based triggering mechanisms that provide more elegant solutions to these problems.
All of this to Minimize Coupling?

- Undisciplined and especially unnecessary coupling is the source of the maintenance problem.

- Objects can be coupled by:
  - containment (must be model-driven)
  - reference (do not introduce just to improve performance)
  - argument passing (very hard to detect/debug)
  - local variables (more disciplined)
  - globals (statics) (the **WORST** kind of coupling)

- A big question is when do we know if two objects must be coupled at all. That is, is the introduction of coupling data-driven (i.e., model-driven) or is it behavior-driven (“because A needs B to do ... it must have access to it)?
- In fact these are two competing views among the “methodologists”.
- The best answer to that is Riel’s: start with data-driven modeling but keep in mind that you will not produce the final model until you consider behavior.
- Coupling should be minimized at the very later stages. Initially the model should drive the associations.
Refining a Design

Some tasks that occur at later stages in the design process

- Minimize coupling by introducing agent/control objects
- This involves the use of some very common design patterns that we will discuss below.
- Refine your sequence diagrams so that methods have as complete a prototype as possible (i.e., specification of arguments return values and their a prototype as possible (i.e., specification of arguments return values and their *types.*)
- Consider timing issues, possible exceptions and try to find opportunities for concurrency
ON CONCURRENCY

A Multi-Threaded Elevator System

• The main control receives a request: from floor in direction
• Control can broadcast the request, in parallel to all elevators
• Each elevator can (in parallel) compute some function that measures how suitable it is to service the request.
• Acknowledgements are returned to control (also in parallel)
• The main control decides which elevator must service the request
• This assumes every elevator has its own processor and data
• Timeouts to be placed on eva(); an elevator might be out-of-service
• We will have a complete Java implementation of the Elevator system

Concurrency

• Concurrency (whether multi-threading or multi-processing) is supported by many languages.
• The difficulty, however, is in identifying the fragments of code that can potentially be processed concurrently.
• This is sometimes called vectorization, since it involves slicing the entire system into vectors of expressions that can safely run in parallel.
• For example, in the above concurrent implementation of the elevator system we have assumed that the various eva() functions that are computed by each elevator in parallel have no side effects.
• In languages that have side effects this is very difficult.
• Modeling the semantics of dynamic systems supporting concurrency is a very involved area of study.
• In concurrent systems the meaning of an expression is not just a function of what sub-expressions it is computed from, but also when.
• Usually these systems are modeled using modal and temporal logics.
• Concurrency/parallelism is more promising in declarative languages.
• Do not confuse concurrency/parallelism with distributed systems.
Note that there is a limit to concurrency. Sometimes the nature of the problem is such that we can only have sequential processing (something has to wait before it can proceed since it depends on data that has not yet been provided/computed!)

For example, in the above model select must wait for all eval() functions to complete. serviceRequest() also waits for select(), etc.
DESIGN PATTERNS

- Over the years the object-oriented paradigm has not quite delivered on its main promise: re-use.
- The difficulty was that building re-usable business objects proved difficult. (Think of our Watch example and the discussion about the design of the TimeManager object.)
- Beyond traditional well-defined libraries (such as GUI objects and standard foundation classes, such as vectors, lists, etc.) there are very little examples of re-use.

Going back to square one...

- Recall our discussions on data abstraction vs. functional abstraction, and the ‘program=control+data’ discussion.
- It seems that in that equation building generic modules/solutions is more promising than building generic business objects.
- Design patterns are an attempt at building generic (thus re-usable) solutions/strategies.

What is a design pattern?

A design pattern is an abstraction (a generalization) of a number of similar problems that require the same set of structures, designs and ultimately solution.

The concept of a design pattern is very simple, and we have all used design patterns before. So do not be overwhelmed by the buzz word ‘Design Patterns’ ...

Let us see what design patterns are...
**Examples of Simple Design Patterns**

**Iterator Pattern**

It sounds very exotic, but essentially it is a pattern of computing that we have all used before. It is the simple pattern of iterating over a collection of data objects (an aggregate of some sort) and performing a certain function on each.

```
List<Obj> collection;
Iterator iter = new Iterator( collection );
for ( obj = iter.First(); ! iter.IsDone( ); iter.Next( ) )
{
    .... obj ...
}
```

**Structure of Iterator Pattern**

```
Aggregate
  CreateIterator( )
```

```
Iterator
  First( )
  Next( )
  IsDone( )
  CurrentItem( )
```

This is indeed a very powerful functional abstraction

```
map f [ ] = [ ]
map f (obj:objs) = (f obj) : (map f objs)
```
SOME OTHER SIMPLE DESIGN PATTERNS

The Broadcast Pattern

Again, this is a very common programming pattern that we have used in many situations. The abstraction here can be graphically viewed as follows:

This also is a very powerful abstraction that is useful in many situations. Note that this pattern is quite similar to the iterator pattern. However, in the iterator pattern it is generally assumed that the aggregate is a collection of objects of the same (polymorphic) type, while in broadcasting the list of objects that might receive the message do not have to have the same general type.

```
map f [ ] = [ ]
map f (obj:objs) = (f obj) : (map f objs)

broadcast f (objs1, objs2, ....) = (map f objs1, map f objs2, ...)
```
AN EXAMPLE OF USING DESIGN PATTERNS

Observer Pattern

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

- Instead of coupling the various objects that depend on each other’s state, one object is made an observer of any change.
- Modifications are made through the observer and the change is broadcasted into the various observers.
- Minimizes coupling and thus simplifies change.
- Very common pattern.
THE STRUCTURE OF THE OBSERVER PATTERN

Subject
- Attach(Observer)
- Detach(Observer)
- Notify(

Observer
- Update(

for all o in observer {
  o -> Update()
}

ConcreteObserver
- ObserverState
  - Update(
    observerState = subject->GetState()
  
ConcreteSubject
- subjectState
  - GetState(
    return subjState;
  )
  - SetState()

aConcreteSubject
- SetState()
- Notify()
- Update()
- GetState()
aConcreteObserver
- Update()
- GetState()
aConcreteObserver
- Update()
- GetState()
THE STATE AND STRATEGY DESIGN PATTERNS

TCPConnection
Open()
Close()
Acknowledge()

TCPState
Open()
Close()
Acknowledge()

state
TCPEstablished
Open()
Close()
Acknowledge()

TCPListen
Open()
Close()
Acknowledge()

TCPClosed
Open()
Close()
Acknowledge()

Elevator
Strategy & str;
Context context
eval() { str->Execute() }
switch()
{
  if ( context == .... )
    str = new Random()
  else ....

Strategy
Execute()

Statistical
Execute()

Random
Execute()

RuleBased
Execute()
class Device
{
    List<Device*>    parts;
    List<Alarm>      alrams;
    List<Device*>    standBy;
    List<Device*>    dependsOn;
    State            state;

    public void newAlarmMessage(Alarm& a)
    {
        if ( state == SOAK )
        {
            if ( alrams.length() > thresh1 ) { g1(); }
        }
        elseif ( state == OOS1 )
        {
            if ( standByOOS(standBy) == standBy.length() ) { g2(); }
            else { g3(); }
        }
        elseif ( )
        {
            //
        }
    }
}
More on the State Patterns

```
State state;
if ( condition1 ) { state = new SoakState(); }
elseif ( condition2 ) { state = new OOS1State(); }
elseif ( condition3 ) { state = new ALERTState(); }
etc...
state->handle();
```
MORE ON THE STATE PATTERNS

class EControl
{
    private List<Elevator*> elevators;
    private Time time;
    private Building building;
    // etc...

    void handleNewRequest( Request req )
    {
        List<float> results;

        ForEach( e in elevators )
        {
            results.insert
            ( e→handleRequest( req, time, building ) );
        }

        Elevator* e = pickElevator( results );
    }

    class Elevator
    {
        private int numOfPendingRequests;
        private State state;
        // etc...

        public float handleRequest( Request r, Time t, Building b )
        {
            if ( ( t == ... ) && (b == ... ) && (r == ... ) )
            {
                return applyStrategy1();
            }
            elseif ( )
            {
                return applyStrategy2();
            }
            elseif ( )
            {
                "-+-"
**Relation Between State and Strategy Design Patterns**

**Problems:**
- Logic in elevator objects is too complicated
- Special kinds of elevators need to be subtyped (different behavior)
- Why should an elevator object be coupled with building, etc.
- What if the strategy applied also depends on whether or not an elevator is down. How would that be accomplished. More data must also be passed from Control – data that does not belong in Elevator!

**Solution**
- What strategy is applied is a function of global data that must only be owned by the EControl object.
- The EControl object should alone decide what strategy must be applied.
- This also allows the EControl object to force different elevators to apply different strategies based on the same context!
MORE ON THE STATE PATTERNS

Generalizing the STRATEGY Pattern
A simple technique that allows us to add/change class behavior at run-time

```
public Type1 method1();
public Type2 method2();
public Type3 method3();
```

```
Object
private Type1 attribute1;
private Type2 attribute2;
private Type3 attribute3;

public List<Operation> operations;

public Type1 method1()
{
    Operation operation1=find("operation1",operations);
    if(operation1 != null) return operation1->apply();
    else
    {
        // default logic....
    }
}
```

```
Operation
virtual Object apply();
```

```
Operation1
Object apply();
```
```
Operation2
Object apply();
```
```
OperationN
Object apply();
```
GENERALIZING THE STRATEGY PATTERN

class Dynamic {
    private Integer x;
    private Integer y;
    public Operation op1;

    Dynamic( Integer a, Integer b)
    {
        x = a;
        y  =  b ;
    }

    public Integer method1( )
    {
        if ( op1 != null ) return op1.apply(x,y);
        else return x+y;
    }
};

class Operation {
    virtual Object apply(Object o1, Object o2)
    {
        return null;
    }
};

class Add extends Operation {
    public Integer apply(Integer o1, Integer o2)
    {
        return o1+o2;
    }
};

class Multiply extends Operation {
    public Integer apply(Integer o1, Integer o2)
    {
        return o1*o2;
    }
};

Dynamic d = new Dynamic(3, 2);
d.Operation = new Add();
d.method1(); // returns 5
d.Operation = new Multiply();
d.method1(); // returns 6

class mostGenericAndDynamicJavaObject {
    List<Object> attributes;
    List<Object> operations;
};

**VISITOR PATTERN (ACCESS CONTROL)**

A simple technique that allows objects to dynamically decide *when* another object/visitor can visit/access them, and *what* they can access. This is preferred to hardwired (static) coupling and enforces formal contracts.

```java
class Visitor {
    ObjectA* agent = new ObjectA();
    Key key;
    public void visitObjectA(ObjectA* a, Key k) {
        agent = a;
        key = k;
    }
    public TYPE useAgencyA() {
        if (agent != NULL) return agent->fa(key, this);
    }
    public void register() { agent->registerMe(this); }
};
class ObjectA {
    List<Visitor> visitors;
    List<Key> keys;
    private void acceptVisits(Visitor& v) { v.visitObjectA(this, getKey(v)); }
    private void stopVisits(Visitor& v) { v.visitObjectA(NULL, eraseKey(v)); }
    //actual functionality is made private
    private TYPE fa() { // code } // public method does nothing unless caller is registered and accepted
    public TYPE pfa(Key k, Visitor* v)
```
{  
    if (validKey(k,v) return fa();  
    else return new TYPE(0);  
}

public void registerMe( Visitor& v ) { if (....) acceptVisits(v); }
};

• Note that the decision to accept or stop visits is made (privately) by the
  object itself. Otherwise, the whole approach is meaningless.

• Potential visitors can try to access an object, and depending on the object’s
  latest decision, they may, or may not be able to visit (access). That is, an
  object can accept or stop visitors as it wishes.

This simple technique can be used to control concurrency, control coupling,
 maintain the state of an object, enforce contracts, etc.

**INTERPRETER PATTERN**

Many applications involve building an interpreter, which is a
context-free grammar plus some semantics.
Persistence and the Data Layer

- In most applications objects have to persisted (saved) before they are destroyed. Objects might again enter the system and some mechanism for creating objects and maintaining an object’s identity must be developed (recall that objects must have a unique identifier)
• If the database to which objects are being persisted in an object-oriented database than the persistence and subsequent retrieval is straight-forward. However, in OODBMS queries are quite complex since one has to be aware of the physical structure and the various access paths to perform queries.

• In most cases objects are persisted into a RDBMS. Here there are some technical challenges. Flat objects, i.e., objects that have attributes of primitive types only can be mapped into tables very easily. However in most applications objects have links to other objects and these references must be translated into and from an appropriate database structure.

• To hide the details of these mappings and to isolate the application from changes in the database or its structure we build a data layer between the application and the database.

• The User Interface should never access data without going through a data layer.

• In most applications the main problem is changes to the application whenever the changes were in the database and/or the user interface. Ideally, these the internal details of these three layers should be hidden and communication should be done through a mapping-layer which is the only place where changes are made.
ISSUES WITH PERSISTENCE OF OBJECTS

• **REPRESENTATION**

  How are objects represented? How are they mapped from and to the DB?

• **ACCESS, SYNCHRONIZATION AND STATE UPDATE**

  Maintaining an object’s identity and synchronizing the state of the objects in memory with the data in the DB.

• **ISOLATING APPLICATION FROM DATABASE**

  The above issues must be resolved in such a way that would could the application isolated from changes in the database model, or even the database system itself.
**On Object-Oriented Databases (OODBs)**

With all the (super) hype surrounding Object Technology, why haven’t object-oriented databases really pick up? Short Answer: **ALGEBRA**

- As a modeling tool the object model is much richer than the entity-relationship (ER) model (actually, any ER diagram can be defined by an object model but not the other way around.)

- However, with this expressive power comes a price. Unlike relational models (where we deal with entities and relations between them), the object model does not have a formal semantics like the relational algebra.

- An algebra is simply a set objects from a domain and a set of operations that produce the objects in the same domain.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Operations</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic</strong></td>
<td>+, *, /, -</td>
<td>Num+Num→Num</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Num*Num→Num</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Num/Num→Num</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-Num→Num</td>
</tr>
<tr>
<td><strong>Boolean Algebra</strong></td>
<td>∧, ∨, ⇒, ⇔, ¬</td>
<td>Bool∧Bool→Bool</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bool⇒Bool→Bool</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Etc...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>¬Bool→Bool</td>
</tr>
<tr>
<td><strong>Fuzzy Logic</strong></td>
<td>∧, ∨, ⇒, ⇔, ¬</td>
<td>FBool∧Fbool→FBool</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Etc...</td>
</tr>
<tr>
<td><strong>Vector Algebra</strong></td>
<td>+v1+v2, 3*v1</td>
<td>Join, Select, Project, Etc...</td>
</tr>
<tr>
<td><strong>Relational Model</strong></td>
<td>Relations</td>
<td>Join R1 R2→Rel, Project cols R→Rel, Select conds R→Rel</td>
</tr>
<tr>
<td><strong>Object Model</strong></td>
<td>Objects</td>
<td>???????</td>
</tr>
</tbody>
</table>

In many ways OODBs are a step backwards to the days of network and hierarchical DBs (like IBM’s IDMS) since to query an OODB we have to traverse pointers between objects; i.e., we have to know the physical structure of the database.
FROM OBJECTS TO TABLES

1. Assign one table $T$ for each class

2. Each primitive attribute will become column in the table $T$

3. For a complex attribute (one that has a structure of its own) we can either
   - Add an additional table (after all, the attribute refers to a class)
   - Split the attribute over several columns in the class (bad normalization and results and many duplications.)

4. The primary key of the table should be a unique object identifier. The key must invisible from the user (should be private), and should be machine generated.

5. Each instance of the class is now represented as a row in table $T$

6. An association with a cardinality greater than 1 will result in a new table $B$ that is connected to $T$ by the primary key
FROM OBJECTS TO TABLES (SIMPLE CASE)

```
Person
private String id;
private String name;
private int age;
private String occupation;
private String address;
```

<table>
<thead>
<tr>
<th>id</th>
<th>Name</th>
<th>Age</th>
<th>Occupation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1969X000</td>
<td>Jimi Hendrix</td>
<td>27</td>
<td>Eccentric Engineer</td>
<td>Blue Sky, Arizona</td>
</tr>
<tr>
<td>1782YE12</td>
<td>Marvin Minsky</td>
<td>55</td>
<td>Media Consultant</td>
<td>Minefield, New York</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- id="1969X000"
- name="Jimi Hendrix"
- age=27
- occupation="Eccentric Engineer"
- address="Blue Sky, Arizona"

- id="1782YE12"
- name="Marvin Minsky"
- age=55
- occupation="Media Consultant"
- address="Minefield, New York"
FROM OBJECTS TO TABLES (NON-TRIVIAL CASE)

<table>
<thead>
<tr>
<th>Person</th>
<th>Name</th>
<th>Age</th>
<th>Occupation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1969X000</td>
<td>Jimi Hendrix</td>
<td>27</td>
<td>Eccentric Engineer</td>
<td>?</td>
</tr>
<tr>
<td>1782YE12</td>
<td>Marvin Minsky</td>
<td>55</td>
<td>Media Consultant</td>
<td>?</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Address

<table>
<thead>
<tr>
<th>Address</th>
<th>id;</th>
<th>street;</th>
<th>city;</th>
<th>province/state;</th>
<th>code;</th>
</tr>
</thead>
<tbody>
<tr>
<td>id=&quot;1969X000&quot;</td>
<td>name=&quot;Jimi Hendrix&quot;</td>
<td>age=27</td>
<td>occupation=&quot;Eccentric Engineer&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>id=&quot;1782YE12&quot;</td>
<td>name=&quot;Marvin Minsky&quot;</td>
<td>age=55</td>
<td>occupation=&quot;Media Consultant&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>id=&quot;6354GEE1&quot;</td>
<td>street=&quot;2000 Bank&quot;</td>
<td>city=&quot;Windsor&quot;</td>
<td>province=&quot;ON&quot;</td>
<td>code=&quot;N9E 1X3&quot;</td>
<td></td>
</tr>
</tbody>
</table>
This is a bad solution since accessing/updating person or address will always involve a projection. Also, if the cardinalities are such that a person can have more than one address or many people can have the same address there will be a lot of data duplication (basic normalization principles tell us that this an extremely bad model).
When a ‘Save Yourself’ or a ‘Refresh Yourself’ message is sent to an object, the object says I don’t do this stuff, but I’ll delegate this task to a proxy object (a db clone object) that I know I have.

**Important Issues to Consider in Complex Applications:**

- How deep in the containment hierarchy should we `save()` and `refresh()`? This depends on the application, the nature of the data and whether the system has multi-user access to the database (which is almost always the case!)
- When do we trigger a refresh? This is important since in a multi-user environment several clients might be looking at the same database, however, some might have obsolete data as others might have made a new `save()`.
- The Observer pattern is very useful here when making notification to `refresh()`, which is essentially an `UpdateState()` operation.
• Recall the importance of object identity!

• Domain objects do not make ‘new’ objects. All object creation is done through a `make()` request that goes through the Factory.

• If a new instance actually needs to be created, factory gets the data from the database and makes a ‘new’ instance otherwise it will return the instance it has a handle on.

• To see how complicated things could get, consider now concurrency and/or a distributed environment.
PRACTICAL ISSUES

1. When creating an instance of the student object we need to go to the DB and pull the data from the PERSON table.
2. What attributes do we populate?
3. Do we get all contained collections? That is, do we perform a shallow dbGet( ) or a deep dbGet( )?
4. What practical heuristics are there?
5. Same reasoning applies in reverse. When we perform a dbSet( ), do we dbSet( ) all objects in the containment hierarchy or do we assume that each object has its own dbSet( )
6. How do we ensure that no two instances of the same object are active? This must be done if we need to maintain consistency.

    {
        ...
        Object o1 = new Object( "JAN012000" );
        o1.f( );
        o1.g( );
        ...
    
    ...
    
    Object o2 = new Object( "JAN012000" );
    o2.f( );
    o2.g( );
    o2.dbSet( );
    }

Basically ’new’ should now be replaced by factory→getOrCreate( )
ON GET() AND SET()

- What about special rooms?
- What about special circumstances?
- To handle these situations, the functionality in the **HeatFlowRegulator** will have to be cluttered with very complicated and unreadable if-then-else logic to cover all cases (including addition of new rooms types, and all exceptional cases).
What are the advantages of this design?
ON GET() AND SET()

- What are the disadvantages of this design?
- Aren’t the classes 'Course' and 'Student' simply structures with no significant 'behavior'?
- Isn’t the 'Employment' class a God class where we have all the logic embedded in one place?
Course
Vector prereqs;
GetPrerequisites();

Student
Vector courses;

boolean Check(Vector prereqs)
{
    return subset(prereqs, courses);
}

Enrolment

NewEnrolment(Course c, Student s)
{
    prereqCourses = c.GetPrerequisites();
    if (s.Check(prereqCourses))
    {
        // accept...
    }
    else
    {
        // reject....
    }
}

Course
Vector prereqs;

boolean Check(Vector courses)
{
    return subset(prereqs, courses);
}

Student
Vector courses;

GetCourses();

Enrolment

NewEnrolment(Course c, Student s)
{
    studentCourses = s.GetCourses();
    if (c.Check(studentCourses))
    {
        // accept...
    }
    else
    {
        // reject....
    }
}
**BEST ALTERNATIVE**

<table>
<thead>
<tr>
<th>Course</th>
<th>Student</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector prereqs;</td>
<td>Vector courses;</td>
</tr>
<tr>
<td>boolean Check(Student s)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>return subset(prereqs,s.courses);</td>
<td>return subset(c.prereqs,courses);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Enrolment</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewEnrolment(Course c, Student s)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>if ( s.Check(c))</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>// accept ...</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>// reject ...</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

- Every object ’owns’ the logic it is responsible for enforcing.
- The ’Enrolment’ object can now be updated in various ways with minimal changes.
AN ATM MACHINE (CONT’D)

•
THE ELEVATOR SYSTEM
THE ELEVATOR SYSTEM (CONT’D)
EXERCISES FOR PART IV
HYBRID OBJECT AND RULE-BASED DESIGN

- All the design patterns and heuristics we have seen have one common theme: reduce coupling and maximize cohesion so as to build highly maintainable and re-usable modules!
- A big problem is how to best distribute the logic of the system in such a way so as to have clear handle on “traceability”.

Rules to the Rescue!

- We Can Use Rules to
  - **Enforce Invariants** i.e., things that should always be true
  - **Detect Constraint Violations** i.e., things that should never be true
  - **Monitor and React to Important Object States** Rules are a good ‘watchdog’ as they are triggered by any relevant change.
  - **Express Domain Knowledge** e.g., business policies, engineering rules, situation-action heuristics, etc.
  - **Eliminate some of the Burden of Procedural Control** Use rules as “automatic member functions” Replaces many ‘if (P) then Q’ by one ‘whenever(P) then Q’ (cleaner code!)
WHAT ARE BUSINESS RULES?

• A business rule is "a statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control or influence the behavior of the business."

• When decomposed to their most elementary form, a business rule is as atomic (indivisible) as possible while still being inclusive enough to be one complete thought.”

• Identification of the objects, properties, derived facts, and constraints of a business statement is essential to creating a model of the enterprise process.

• As important as identifying the constituent parts of business rules, is an understanding of what is not to be found within these rules.

  ➢ Rules do not contain process or control statements.
  ➢ They do not define objects, but they reference the objects of an application.
  ➢ Rules do not instigate any action. If actions are to be taken, they are carried out by the responsible business object.
**BUSINESS OBJECTS VS. BUSINESS RULES**

- Business rules and business objects are easily confused, and it is important to recognize that rules are distinct from objects.

- However, complex enterprise applications must be modeled using both.

- Objects represent real world entities like customers, products, orders, revenues, and assets. They encapsulate the data and behavior needed to perform business functions.

- Rules, on the other hand, implement the policies and practices of an organization. They control how objects perform business functions.

- Historically, there has been little or no distinction. Objects were considered to contain 'business logic.'

- Many development methodologies promote the misperception that 'business logic' was properly the domain of software developers.

- To control business decisions and react to change a new methodology is needed to bridge the gap between 'business logic' and 'application logic'.

BUSINESS RULES & MULTI-TIER DESIGNS

From RuleMachines Web Site

http://www.rulemachines.com
AN EXAMPLE WHERE & WHY RULES ARE NEEDED

- Business objects provide ”services” or ”functionality”. But these services are not always activated. Rules, on the other hand, are always active (i.e., must always be enforced!!!)
- Here’s an example. In a student registration system we might have:

**Business Objects**

- Student
- Course
- CourseOffering
- Instructor
- Room
- Degree
- Etc...

**Business Rules**

R₁: A student can not take more than 6 courses in 1 term
R₂: Instructors should not teach more than 200 students per term
R₃: Courses can not be taught in the same room at the same time
R₄: To graduate a student must complete some degree requirements
R₅: TAs for a COURSE must have obtained at least a B+ in COURSE
R₆: 2 TAs can be assigned for every [1,...,60] students
R₇: An instructor can teach at the most three offerings per term

Etc...

- Note that the business rules are different from some behavior (or functionality) that the objects might provide.
- Unlike responsibilities, which could be distributed among objects, rule often involve several objects. This presents a major challenge, namely ”where should the rules be defined?” or ”where must the business logic be enforced?”
Recall that the main issue with maintainability is to have handle on traceability. This means that care must be made in distributing the logic across the system. How are we to distribute rules, which naturally involve several objects?

### Enforcing Rules in an OO Design

```java
class SCourse {
    Student& student;
    Course& course;
    String status;
};

class Student {
    private String studentNo;
    private List<SCourse&> courses;
    private Degree& major;

    // enforces R1
    public void addCourse(Course& crs)
    {
        int cnt = 0;
        ForEach( c in courses )
        {
            if ( c.course.status == IP )
                cnt++;
        }
        if ( cnt >= 5 ) { outputMsg("Can not add course."); }
        else courses.addCourse( crs );
    }
};

class Offering {
    private String sectionNo;
    private Course& course;
    private Time time;
```
### ENFORCING RULES IN AN OO DESIGN (CONT'D)

```
class Instructor {
    private String employeeNo;
    private List<Offering&> offerings;
    private Department& department;

    // enforces R2 (or DOES IT????)
    public void addOffering(Offering& offer) {
        int cnt = 0;
        ForEach( o in offering ) {
            cnt = cnt + o.getNumberRegistered();
        }
        if ( (cnt + offer.getNumberRegistered()) > 200 ) {
            outputMsg("Can not add course.");
        } else offerings.addOffering( offer );
    }
};
```
What are the problems?

First, the rule **An instructor should not teach more than 200 students each term** must be enforced in more than one block of code, and, worse yet, it might have to be enforced in more than one object!!!

For example, if we added students to some offering that has already been added to an instructor, we now have to check if the instructor teaching this offering can still take students!! So, in this case, the rule must be enforced in (at least) two places...

```java
// ENFORCING RULES IN AN OO DESIGN (CONT'D)

class Instructor {
    private String employeeNo;
    private List<Offering*> offerings;
    private Department& department;

    // enforces part of R2
    public void addOffering(Offering* offer) {
        int cnt = 0;
        ForEach( o in offering ) {
            cnt = cnt + o->getNumberRegistered();
        }
        if ( (cnt + offer->getNumberRegistered()) > 200 ) {
            outputMsg("Can not add course.");
        }
        else offerings.addOffering( offer );
    }
};

class Offering {
```
private String   sectionNo;
private Instructor*  instructor;
private Department&  department;
private List<Student*> students;

public void addStudent(Student* student)
{
  // first enforce part of R2
  if ( checkInstructorLoad() ) {
    students.addElement(student);
  }
  else { ... }
}

(SERIOUS) PROBLEMS WITH THIS APPROACH
• The same rule is enforced in several places (traceability & thus maintainability is bad)
• Ironically, enforcing rules can not be done in traditional OO without violating
  encapsulation. For example, why should the designer of the Offering course ever be
  concerned with load on an instructor???
• Finally, since we are explicitly enforcing rules, we have to make sure we cover all
  cases, spreading snippets of code all over the place and this will inevitably lead to
  errors.
HYBRID OBJECT AND RULE-BASED DESIGN

Rules to the Rescue!

- We Can Use Rules to
  - **Enforce Invariants** i.e., things that should always be true
  - **Detect Constraint Violations** i.e., things that should never be true
  - **Monitor and React to Important Object States** Rules are a good ‘watchdog’ as they are triggered by any relevant change.
  - **Express Domain Knowledge** e.g., business policies, engineering rules, situation-action heuristics, etc.
  - **Eliminate some of the Burden of Procedural Control** Use rules as “automatic member functions” Replaces many ‘if (P) then Q’ by one ‘whenever(P) then Q’ (cleaner code!)
EXAMPLES OF RULES+OBJECTS

Enforcing Invariants (Semantic Constraints)

class Rectangle {
  private:
    monitored int height;
    monitored int width;
    int area;
  public:
    Rectangle(int h, int w)
      : height(h), width(w) {}

    rule update_area;

    etc..
};

Rules add automatic (non-procedural) behavior to a class.

rule Rectangle::update_area {
  height > 0 &&
  width > 0
  =>
  area = height * width
}
A HANDLE ON TRACEABILITY AND MAINTAINABILITY

Here’s an example of a real-world engineering rule:

“if a new alarm message is received on a device and its standby is in an Out-of-Service state, then send an alert.”

How can such a business rule be coded in one place for traceability and future maintenance?

```c++
rule Device::alert_if_standby_oos {
    fsm
    &&
    Device *stdby = standby
    &&
    State *st = stdby->state
    &&
    st->name == OOS
    &&
    Alarm *mx @ self_msg
    &&
    current_time <= mx-timestamp
    =>
    Time time = make_time("now");
    Alert *new_alert = new Alert(...);
    new_alert->send();
}
```

We can track each business rule/policy into a first-class object (a rule) that implements the logic in a single place!

The above fragment is an actual code in the language R++ which is a rule-based extension to C++ developed at AT&T Laboratories. (http://www.research.att.com/tools/)
class Instructor {
    private String employeeNo;
    private List<Offering*> offerings;
    private Department& department;

    // ALWAYS enforcing R2
    rule checkInstructorLoad;
};

rule Instructor::checkInstructorLoad {
    offerings.size() > 0
=>
    int cnt = 0;
    forEach (o in offerings) {
        cnt += o.students.size();
    }
    if (cnt > MAX)
    { // exception... must remove some students }
    else { // do nothing }
}

class Offering {
    private String sectionNo;
    private List<Offering*> offerings;
    private Department& department;

    // ALWAYS enforcing R2
    rule checkInstructorLoad;
};

rule Instructor::checkInstructorLoad {
    ! (instructor.canHandleLoad(students.size()))
=>
    { // do not commit the class addition ....
        // or, remove one according to some priority rules...
    }
}
By the way, here is an example of not using `get()` and hiding the details of an object. For example, we could have replaced the call `instructor→canHandleLoad(students.size())` by 

```
(students.size()>instructor→getMaxLoad())
```

However, this exposes the object and the logic of how an instructor determines if s/he can handle a certain load. This should be private, and we should only ask the instructor if they `canHandleLoad()`
class Student
{
    private String     studentID;
    private List<Course*> completedCourses;
    private Degree*    degree;

    // enforce a rule as to when a student can graduate

    rule graduationRule;
};

rule Student::graduationRule
{
    subset(  degree→requiredCourses(),
             completedCourses
         )
=>
    canGraduate == TRUE;
}

NOTE

- This rule checks if a student can graduate WHENEVER the student courses have changed, and/or the degree requirements have changed.
- What is the alternative???
- If we did not have such rules, then if the requirements of a certain degree have changed we also have to visit all students and check if they can now graduate.....
MORE ON RULES
COMPONENT-BASED PROGRAMMING (AGENTS)
A Society of Virtual Agents
EXERCISES
You must know the following:

- Design tradeoffs
- Essential difference between OOP and other paradigms
- Relation between traceability and maintainability
- The object model: classes, objects, relationships
- Object identity, construction and canonical forms
- Difference between “isa” and “part-of” hierarchies
- Relation between inheritance and polymorphism
- Multiple inheritance and associated technical difficulties
- Containment and the uses relationship
- Containment and delegation
- Compare “uses” via inheritance and “uses” via delegation
- Modeling and analysis heuristics (classification, coupling, etc.)
- The UML notation as modeling notation
- CRC crads, use cases, state transition and sequence diagrams
- Invariants, semantic and cardinality constraints
- Design heuristics
- Refining a design: timing, concurrency, etc.
- Design patterns
- Hybrid object and rule-based design
- Distributed OOD

- Develop a somewhat detailed model for a non-trivial example
Sources

Professional and Academic Textbooks


Periodicals

SIGS (JOOP, C++ Report, etc.): [http://www.sigs.com/](http://www.sigs.com/)
ACM: [http://www.acm.org/](http://www.acm.org/)

Relevant Web Sites

UML documents: [http://www.rational.com](http://www.rational.com)
ECOOP: [http://www.ecoop.org/](http://www.ecoop.org/)
OOPSLA: [http://www.acm.org/sigs/sigplan/oopsla/](http://www.acm.org/sigs/sigplan/oopsla/)
Midterm Test 1

Part I

1. The most important design tradeoff is the tradeoff between _______ and _______.

2. In the object-oriented paradigm control is implemented as _______ between _______.

3. The challenge in object-oriented design is in distributing intelligence across the system so as to maximize _______ and minimize _______.

4. An object can provide a service it does not implement either by _______ or _______.

5. Factoring out specialized (highly cohesive) functionality facilitates _______. If done to extreme, this can increase the _______ the system.

Part II

1. Delegation is often achieved using the containment relationship _______.

2. Only in objects (instances) and not classes do we assign values to attributes _______.

3. Ambiguity in multiple inheritance must be explicitly resolved by the designer _______.

4. Pointers (or references) can be used as keys to object identifiers _______.

5. To generalize programs both data and functionality must be generalized _______.
## Midterm Test 1 (cont'd)

<table>
<thead>
<tr>
<th>PART III</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>There are 10 questions in part III, each worth 2% for a total of 20%</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Circle the most appropriate answer to each of these questions.</strong></td>
<td></td>
</tr>
</tbody>
</table>

1. The relation between an abstract data type (ADT) and a class can best be described by the following:
   a. Both classes and abstract data types support inheritance
   b. A class is an abstract data type, but an abstract data type is not a class.
   c. An abstract data type is similar to an abstract class in object-oriented programming
   d. Abstract data types and classes are exactly the same concepts.

2. Inheritance is often used:
   a. When we like to re-use the services provided in a base class.
   b. When several types can be substituted for each other in some context (to catch a ride, any vehicle will do).
   c. When modeling the real-world suggests the use of such a relation.
   d. When any of the above situations arise.

3. Delegation, containment and re-use are related as follows:
   a. We introduce containment to delegate functionality from a container to a contained object.
   b. Delegation often happens as a result of factoring out an object for re-use purposes.
   c. Container objects often delegate some functionality to their contained objects.
   d. b and c but not a.

4. Delegation and inheritance are related as follows:
   a. Both represent a situation where a class provides a service it does not actually implement.
   b. Both introduce coupling between objects.
   c. Both implement the "uses" relationship.
   d. All of the above.

5. A good heuristic suggests that there is some error in the class hierarchy if:
   a. The class hierarchy is too deep.
   b. The class hierarchy has too many situations of multiple inheritance.
   c. The resulting code has too many case statements on some attribute.
   d. Only b and c.

6. The source of the so-called "software crisis" is:
   a. Slow progress in the development of Rapid Application Development (RAD) tools.
   b. The difficulty in the design and development of re-usable code.
   c. Traceability and maintainability.
   d. All of the above.

7. A user-defined data type is said to be in canonical form if:
   a. All operations that primitive data types have are defined on that type.
   b. Data and behavior are well encapsulated in the user-defined data type.
   c. If the internal structure of the data type is not exposed by get and set methods.
   d. All of the above.

8. If an object of type A physically (not by reference) contains a B object, then the following is true:
   a. An object of type B must be constructed whenever an A object is constructed.
   b. The B object must be destroyed whenever the A object is destroyed.
   c. An A object and a B object have the same lifetime and they are coupled throughout their lifetime.
   d. All of the above.

9. A dynamic (as opposed to a static/compiled) environment is suitable for:
   a. Environments that are constantly changing.
   b. Environments where unpredictable situations that might arise must be handled.
   c. Real-time systems where very low-level tasks are performed (in debuggers, schedulers, etc.).
   d. a and b.
1. On Classification
We are given the task of building an object-oriented system for an online digital library that catalogues various types of multimedia objects (such as books, journals, newspapers, thesis, reports, scientific papers, conference proceedings, videos, audio tapes, images, etc.). Papers exist on their own, or they might be "contained" in a book or a conference proceeding (both of which, along with journals, are a collection of papers). Multimedia objects, such as audio, image or video objects, can also be part of another content object. Your task here is to simply deliver the object model. Identify the key abstractions (classes) and draw the class diagram (or the object model), showing all the relationships, attributes and methods that you think will be needed to properly navigate, search and manage this digital library.

```
class ContentObject:
    String title;
    Date date;
    List<Person> authors;
    String publisher;
    int size;
    int refNumber;
    List<String> keywords;
    List<String> subjects;
    Query(BooleanExpr);
    InvariantKeywords();
```

```
class MultimediaContentObject:

class TextualContentObject:

class Book, Journal, Proceeding, Paper:

class EditedBook:
```
2. On Containment

Consider the simple class diagram given below, which models a document as a recursive structure of object containment where the simplest object is a word object. Objects at all levels in the document point to a graphics context (GC) object which defines a set of default graphic attributes, as shown below. These attributes apply to each object in the part of hierarchy unless locally overridden. Identify the appropriate attributes and methods needed to propagate a change in the GC of the document (top level object) to all the objects in the containment hierarchy. Write pseudo code that shows what happens when a method such as Document::setGC( GC gc ) is called (assume every object has a method void DocObj::redisplay(GC gc ) ).
3. On the "uses" Relationship, Delegation and Reuse

In the model below we decided to factor a TimeManager and DateManager objects out of the functionality of a watch so that these objects could be used in another system where the maintenance of time and/or date only are needed. However, in this model the TimeManager interacts (is coupled with) the DateManager object since a change in time might cause a change in date. This makes the TimeManager less re-usable, since some of its logic now assumes the existence of a DateManager, which means it can not be re-used in systems date is not maintained. How can you solve this problem (how can you decouple time and date, yet still maintain time and date)? Draw a new diagram and show the pseudo code for the method void Watch::incrementMinute( )

```
void Watch::incrementMinute( )
{
    timeManager.incrementMinute( );
}

// although the next method is similar, this
// is needed for the outside interface
void Watch::incrementDate( )
{
    dateManager.incrementDate( );
}

void Watch::endOfDay( )
{
    dateManager.incrementDate( );
}

void TimeManager::incrementMinute( )
{
    minute++;
    if ( minute > 60 ) incrementHour( );
}

void TimeManager::incrementHour( )
{
    hour++;
    if ( hour == 24 ) observer.endOfDay( );
}
5. On Inheritance and Polymorphism

Companies have several types of employees, such as engineers, administrators, managers, secretaries and executives. Draw a class diagram showing the attributes and methods needed to write a polymorphic method `totalNumberOfVacationDaysRemaining()` that determines the number of vacation days all employees can still have at some point in time. Although the actual function is not important, in a given year the number of vacation days an employee can have is based on the type of the employee, number of years in service, and on how many days have already been taken off (employees can anytime take days off against their vacation).

```cpp
class Employee {abstract}
int yearsInService;
int vacationDaysTaken;
//int vacationDaysRemaining;

int vpr(); {virtual}

class Engineer
int vpr();

class Administrator
int vpr();

class Manager
int vpr();

class Secretary
int vpr();

class Executive
int vpr();

int Object::totalNumberOfVacationDaysRemaining( List<Employee> employees )
{
    int result = 0;
    ForEach( emp in employees )
    {
        result = result + emp.vpr();
    }
    return result;
}
```
PART V

- There are 4 questions in part V, each worth 5% for a total of 20%
- In the space provided explain the heuristic and what violating it means.

1. **An object should know of objects it contains, but an object should not know what contains it.**

   If an object knows who contains it then it must have some logic reflecting this fact. This means that the contained object is less likely to be reusable in other situations. For example, if a generic BrakeSystem should not have only logic that is specific to any kind of vehicle. This way it can be used by (plugged in) a different types of Car, Truck, or Van objects.

2. **Keep related data and behavior in one place.**

   This heuristic is related to the problem of distributing intelligence across the system. The heuristic says that logic that manipulates some data should be in the same place where the data resides. That is, it should be the responsibility of an object that maintains that data to maintain that relevant logic. A good indication that this heuristic is being violated is when there are many get() and set() methods. Why would any object get() the data if another object if it was not implementing some logic on its behalf?

3. **A class should capture one and only one key abstraction.**

   This is a heuristic to create cohesive objects. A class that captures more than one key abstraction is not a cohesive class since it will end up containing functionality that implements several unrelated things. For example, in the digital watch problem, the TimeManager object is a very cohesive object since it does one specific functionality. It is therefore a good abstraction. We could have designed one object that maintains time and date, but the result would have been a less cohesive object, since it is now an object that does several things. Moreover, the bigger and less cohesive object is less reusable, since systems that only need to maintain time can now use the well cohesive TimeManager object. The other extreme is to have a MinuteManager, an HourManager, etc. objects. This is not needed for several reasons (discuss in class).

4. **Do not create god classes/objects in your system.**

   A god class is a class that contains most (if not all) of the functionality/behavior of the system. When we have such classes the rest of the objects in the system end up being data structures with no behavior. The problem with this very common error is that the system usually ends up being designed as a structured program, since the logic in this big class can only be broken down in an action-oriented (i.e., procedural fashion) as opposed to the contract or CRC method.

   Control objects, although sometimes needed, especially for some good design patterns, must be carefully crafted so as not to end up designing them as god classes.
<table>
<thead>
<tr>
<th>PART VI</th>
<th></th>
</tr>
</thead>
</table>
| • There is one question in part VI that is worth 10%  
• Answer in the space provided (use the back of the pages if needed) | Page 10 |

Your are involved in the design of an ATM (automatic teller machine) system. An ATM is used by the bank’s customers to do most of their banking. Customers typically use it to withdraw or deposit money as well as to check and/or update their balance. The bank’s customers as well other customers with a major credit card can also use the ATM to withdraw money against accounts that do not reside in the bank.

Draw the use case diagram (showing at least three obvious use cases), identify the key abstractions (at least four) and draw the object diagram showing at least two attributes that each object must have. For each key abstraction provide a “class responsibilities and collaboration” (CRC) card, showing at least two services (methods) that each object must provide. Trace in an interaction diagram one use case and for one object show the state transition diagram (use a UML-like notation).
SAMPLE TEST
NOTES