Class responsibilities & relationships
Responsibilities

• Responsibilities describe a class’s purpose in terms of its functionality

• Responsibilities include:
  – Knowledge an object maintains
  – Actions an object performs
  – In general, all services an object provides
Clients and Servers

• An object acts as a server when it fulfills a request made by another object (the client)

• A contract between 2 classes represents the list of services one object can request from another

• Responsibilities support contracts, so they represent only publicly available services
Define what, not how

• Keep definition of responsibilities at an implementation-independent level
• A client/server contract should specify what is done, not how the task is accomplished
Finding Responsibilities

• Go back to requirements specification & re-read carefully
• List or highlight verbs, using common sense to determine which of these represent actions some object must perform
• Make note of information mentioned - an object or objects will need to maintain it
Responsibilities & Use Cases

• Use cases provide excellent source for finding responsibilities, allowing you to:
  – Imagine how system will be invoked
  – Go through variety of scenarios using as many system capabilities as possible
  – Look for places where something must occur as result of input to system
Responsibilities & Preliminary Class List

• Use previously identified classes as a source of information
  – Class name usually suggests at least one responsibility, maybe more
  – Statement of purpose may suggest additional responsibilities
Assigning Responsibilities

- Assign each identified responsibility to the class or classes to which it logically belongs.
- Clearest source for this information is the spec - examine context in which you identified each responsibility.
Assigning Responsibilities

• Follow the guidelines below for responsibilities that don’t fit obviously with a specific class:
  – evenly distribute system intelligence
  – state responsibilities as generally as possible
  – keep behavior with related information
  – keep information about one thing in one place
Even Distribution of Intelligence

• Intelligence: how much a class knows or can do, and how many other objects it can affect

• A tempting (but erroneous) approach is to centralize control by minimizing the number of intelligent classes
  – hard-wires system behavior, making it harder to modify
  – less intelligence means less behavior - takes more “dumb” classes to implement system
Even Distribution of Intelligence

- Even distribution means designing all classes to be equally intelligent
  - each object knows relatively few things
  - promotes flexibility, ease of modification
  - requires relatively fewer classes with comparable functionality
Checking for Even Distribution

• Compare list of responsibilities for each class

• If a particular class appears too “smart” relative to all others, stop & examine it
  – check that all responsibilities are stated with the same level of detail - if not, re-write
  – if necessary, split the class
State responsibilities generally

- It’s easier to find common responsibilities between classes if they’re stated generally
- Common responsibilities are responsibilities that can be assigned to an abstract superclass and inherited by its subclasses
Keep behavior with related info

• The heart of the encapsulation principle is that like things should be kept together
• If an object maintains certain information, it makes sense to assign it responsibilities to perform operations on that information
• Conversely, if an object already manipulates certain information in some way, can give it the responsibility of maintaining the information
Keep info about 1 thing in 1 place

• Should not share responsibility for maintaining info between 2 or more classes
• Such duplication can lead to inconsistency, loss of data integrity
Keep info about 1 thing in 1 place

• If 2 or more objects must know same information to perform their actions:
  – can assign maintenance responsibility to one of the objects involved - others can get access by sending messages to this one
  – can create new object to be sole repository of info
  – can collapse objects requiring the info into a single object
Share responsibilities

• May find that an identified responsibility is really several responsibilities & should be divided among 2 or more objects
  – More complete understanding of how model should function may lead to determination that responsibility should be shared
  – Split responsibility into smaller, more specific responsibilities & assign each to the most appropriate class
Dealing with “leftover” responsibilities

• Class may be missing from design - can create new class to encapsulate related set of unassigned responsibilities

• Can arbitrarily assign to existing class - if 2 candidates seem equally likely, just pick one - can always change later
Recording Responsibilities

• List responsibilities assigned to a class succinctly on its CRC card
• If all responsibilities won’t fit on one card, you may be including too much detail or the class may be too “smart”
• Don’t list superclass responsibilities on subclass cards
• Responsibilities should be stated generally; there is not necessarily a one-to-one correspondence between responsibilities and methods
Examining Relationships Between Classes

• Can find additional responsibilities by looking at relationships between classes, especially the following:
  – “uses” (example - a driver uses a key to start a car - dependency relationship)
  – “is part of” or “has” (example: the ALU is part of the CPU, a car has an engine - aggregation)
  – “is a kind of” or “is a” (example: a square is a kind of rectangle, a lion is a mammal - inheritance)
Dependency

• One class depends on another if it manipulates object of the other class
• A class is independent if it can carry out all its operations without being aware of the existence of other classes
• In the voice mail example, Mailbox is dependent on the Message class, since a Mailbox stores Messages; on the other hand, a Message can carry out its operations without being aware of the Mailbox that stores it
Minimizing dependency

• A certain degree of dependency between classes is necessary, and even desirable; however:
  – Dependency relationships make software more difficult to modify
  – Change in the server class may necessitate changes in client
Minimizing dependency – example

• In C++, we can overload operators, and we often overload the `<<` operators so that we can output objects of new class types

Consider the following functions; which is more versatile, and why?

```cpp
void Obj::output () const
{
    cout << member1
    << member2
    << member3;
}

ostream& operator << (ostream &outs, const Obj& o)
{
    outs << o.member1
    << o.member2
    << o.member3;
    return outs;
}
```
Minimizing dependency - example continued

• Java doesn’t provide the ability to overload operators; a similar Java Obj class would have to rely on two classes (System and PrintStream) to send output to the screen, and there is no mechanism for sending output from a new class to a generic output stream

• We can make the Java version of Obj less dependent by having a member method that returns a String object, since Strings can be output to a variety of streams

• The C++ version of this idea is shown on the next slide
Minimizing dependency - example continued

```cpp
string Obj::getText () const {
    string s = "";
    s = s + member1 + member2 + member3;
    return s;
}
```

- Uses string concatenation to build result string
- Result string can be output to whatever stream is desired - or used for purposes other than output
Aggregation

- An aggregation is a class that contains, or is composed of, objects of another class
- This is really a special kind of dependency; the container needs to be aware of what it contains
- In the voice mail example, the Mailbox class can be considered an aggregation because it contains instances of the Message class
- In the ATM example, Keypad is an aggregation, since it contains Keys
Inheritance

• A class inherits from another if it incorporates the behaviors and attributes of the other class
• This usually means that the objects of the inheriting (child or sub-) class are specialized forms of the original (parent or super-) class objects
Inheritance

- Subclass objects can always be used in place of superclass objects; for example, if the superclass is “Language” then subclasses “French” or “Java” should be valid arguments to a method requiring a “Language” parameter
- Superclasses may be abstract or concrete; the lowest level of subclass is always concrete
- Inheritance is useful, but less commonly used than dependency or aggregation
Class responsibilities & relationships