Basic Class Design

- **Goal of OOP**: Reduce complexity of software development by keeping details, and especially changes to details, from spreading throughout the entire program.

- **Actually, the same goal as software design concepts throughout**:
  - Originally, "structured programming" means using the information-hiding abilities of subroutines (functions) to help organize the code; a subroutine can be modified internally without requiring the calling code to change.
  - Then, "modular programming" means breaking up the program into separate modules that can be developed and modified separately, and perhaps even re-used - like a library. A library is just a generally useful module.

- **Definitions**
  - Client Code - the code that uses the classes under discussion.
  - Coupling - code in one module depends on code in another module
    - Change in one forces rewrite (horrible!), or recompile (annoying), of code in the other.

- **Two of the Four key concepts of OOP**
  - **Abstraction** - responsibilities (interface) is different from implementation. Distinguish between interface and implementation.
    - A class provides some services, takes on some responsibilities, that are defined by the public interface. How it works inside shouldn’t matter.
    - Client should be able to use just the public interface, and not care about the implementation.

  - **Encapsulation** - guarantee responsibilities by protecting implementation from interference
    - Developer of a class can guarantee behavior of a class only if the internals are protected from outside interference. Specifying private access for the internals puts a wall around the internals, making a clear distinction between which code the class developer is responsible for, and which code the client is not supposed to know or care about.

  - **Both are ways of decoupling the client code from the class implementation**:
    - Don’t need to know about the implementation;
    - Can’t accidentally interfere with it or depend on it.

  - **A class is a software component that consists of some data and some procedures, coupled together, with an interface that abstracts and encapsulates the data and procedures, insulating the client from the details of implementation.**

  - **Here, only concerned with "Concrete" classes - no inheritance or polymorphism involved.**
    - Objects interact with each other, contain each other, refer to each other.
Main program causes initial objects to be created, delegates work to objects as needed.

**Two kinds of classes:**

- *Objects that are in the problem domain.*
  - Managing Media: Records, Collections
  - Managing meeting schedule: Rooms, Meetings, Persons
  - Banking: Customers, accounts, banks

- *Objects that support the domain objects (useful to implement them).*
  - E.g. String, Ordered_list
  - Choose these to best get the work done.
  - But you have to understand what the real work is first!
Guidelines for Designing Individual Classes

Designing Domain classes

What kinds of objects are in the domain?
- Start with thinking about the objects, then group into classes.
- Which classes do they naturally belong to?

What characterizes each domain object?
- Member variables - the data
- Member functions - operations on the data

How are different kinds of objects related to each other?
- Inclusion versus association -
  - Part-of relation versus "using" or "interacts with"
- Relative lifetimes -
  - Do they exist independently of each other?

Design a class by choosing a clear set of responsibilities for it.

Make classes responsible for working with their own data.
- Main code should delegate the work down into the classes that have the information.

Deciding which class or module should be responsible for the work.
- Most general rule: who has the data? That component is probably the best one to do the work. Classes generally should be responsible for their data, and the work done with the data.
- If client code is doing the work, something is probably wrong - rethink it!

If class responsibilities can’t be made clear, then OOP might not be a good solution

Lots of problems work better in procedural programming than in OOP, so there is no need to force everything into the OO paradigm.
- Making this distinction is critical to understanding the difference between traditional procedural programming and OOP.

But often, you just need to think about it more carefully.
- Anthropomorphize it - e.g. in P2, imagine a person playing the role of each class object. What is her job? How does she interact with other objects?

Beware of classes that do nothing more than hold data, like a C struct type.
- Sometimes you need simple “holders of data” - no associated functions or operations - if so, then it should not be a class.

- Or, you’ve mis-assigned responsibilities - maybe this class should be doing the work, but some other component is doing it instead.

▼ Is it really a "Plain Old Data" object, like C struct, or did you overlook something?

- If it is a simple bundle of data, define it as a simple struct.

- If there are functions that operate on the data, maybe they should be member functions, and maybe these objects really are responsible for something.
A couple of General DO NOT rules

General Don’t: Don’t overengineer -

Overengineering - a more complex solution than necessary.

Often a result of anticipating future needs inappropriately.

- “Yes, it is more complex, but if I do it this way, then in the future, it will be easier to do yada-yada.” - but will this be needed?

- Problem: the code is harder to work with NOW, and you don’t actually know whether you will need to do the future thing.

- “YAGNI” principle - “you aren’t going to need it”.

- Wisdom of the gurus: If the code is a simple solution that is clear and well-designed, it will be easy to change it in the future if necessary.

- So design and code a current solution well, instead of making a mess trying to anticipate an unknown future.

- Another reason: Just getting complex without thinking through what the responsibilities of the classes really are - misdelegating, misassigning - the result is simply unnecessarily complex.

General Don’t: Don’t create heavy-weight, bloated, or "god" classes - prefer clear limited responsibilities.

- If a class does everything, it is probably a bad design. Either you have combined things that should be delegated to derived classes or peer classes, or you have misunderstood the domain.

Example: In project 2’s restore function, the work of reading and interpreting the data file, and creating the right objects or relationships, was delegated to the Record and Collection classes, which do all the work, and just signal a problem if they can’t. Only thing the main module knew is that Record data comes first, then Collection data. If a member variable was added to Record, only the Record class class would need to be changed.

- Contrast with a god-like main module that knows how Records and Collections are structured, and what the details of what data file looks like - it reads the data, validates it, creates objects, and stuffs the data into them from the outside while they just sit there passively. If a member variable is added to Record, both the Record declaration and the main module code would need to be changed.
Do the class design work at the level of the public interfaces, not the private implementations.

- Don’t get bogged down in implementation details like “I can do with this with a map container and a deque!”

Think only about what the class responsibilities are and what they do in their public interfaces:

- Class X is responsible for …., class Y for …. 
- When an X object needs … it calls the public member function … of the appropriate Y object with … as parameters, which returns …
- Try writing pseudo code just for the interactions between class objects through their public interfaces.
- Keep this up until you can’t stand it any more, then make implementation choices and write the code.

Continue design thinking until you have thought of at least two reasonable ways to solve each design problem.

- “Reasonable” here means “not obviously stupid.”
- Don’t just jump on the first design you think of and hack it out.
- All designs are imperfect - they all involve trade-offs. They are good in some ways, bad in others.
- A good design is good in the most important ways, and bad in the less important ways.
- But there might be more than one good design - just different in the specific tradeoffs.
- You can’t make an intelligent choice if you have only thought of one design - there could be another, better, simpler one.
Some specific rules for class design

- Make all member variables private in each class
  - Concept: Programmer of a class has to be able to guarantee that class will fulfill its responsibilities - do what he/she says it will do.
  - encapsulation - making member data private- is the basic step that makes this guarantee possible - prevents other code from tampering with the data.
  - No public member variables.
  - Beware of get_ functions that return a non-const pointer or reference to a private member variable - breaks the encapsulation!
  - Be careful if it is necessary to return a non-const reference or pointer to an item in a container member variable - even if client can't alter the container, might be able to alter the item in a way that violates design intent - or disorder the container!

- Put in the public interface only the functions that clients can meaningfully use.
  - Functions that are only helpers for the implementation should be private.
  - Resist the temptation to provide getters/setters for everything.
    - If you have to do this, something is probably wrong with your design -
      - why does somebody else have to put data in and pull data out of the object?
      - why aren't the class's member functions doing the work?

- Friend classes and functions are actually part of the public interface of the class, and belong with the class.
  - Friend class or function is part of the same module or component.
  - Most clear if declaration and implementation is in the same .h and .cpp files.
  - A class developer should declare a class or functions to be a friends only if he/she/they are also responsible for, and have control over, that class or function.
  - If class A uses class X for its internal work, and the client shouldn't have to see class X, then consider declaring class X as a private member of class A rather than have X be visible to the client with a friendship relation to A.
  - Make member functions const if they do not modify the logical state of the object.

- Make a class fully responsible for initializing itself with constructor functions.
  - It is error-prone and bad design if the client has to "stuff" initial data into the object.
  - Take care that all member variables get a good initial value.
    - Only supply these where necessary - if the member variable is a class type, the compiler will call its default constructor for you.
Some specific rules for class design:

- **Do not write constructors, assignment operators, or destructors when the compiler-supplied ones will work correctly.**
- Unnecessary code is simply places for bugs to hide!
  - Especially when revisions are made!
  - E.g., did you remember to fix the copy constructor when adding another member variable?
  - Let the compiler do the work as much as possible - it will automatically respond to changes in the code.