Top down design, modularity, & abstraction

- one key design focus is decomposition of overall problem into subproblems that can be solved independently
- allows developer to focus on design of one part, abstracting away the inner workings of the other parts
- while implementing a high level component, you think of the abstract view of what the lower level components do – ignoring the gory details of how they do it
- while implementing a lower level component, you ignore what the higher level component might be using it for

Abstraction

- modeling in a way that provides “just enough” detail to solve the immediate problem
- control abstraction (e.g. functions, methods):
  - gives a logical name to a sequence of actions and describes its externally-observable effects
  - identifies any necessary inputs/outputs (including parameters)
- data abstraction (e.g. structs or classes):
  - gives a logical name to a data type
  - describes the nature of the information stored
  - describes the publicly-visible operations that can be used to manipulate the information stored

Top down decomposition

- For system as a whole, we think of who (users/other systems) it interacts with – what data it takes in, what processing it does, what data it pushes out
- We then think of it as a small number of key subsystems, and how they interact with one another
- Then we carry out the decomposition process on each subsystem, stopping the decomposition “tree” when we reach components that are simple enough to implement directly

Decomposition into components

- in first CS course (e.g. 160) we often think of top down design purely in terms of division into functions
  - main routine calls several functions to perform core/major tasks
  - each of those may call multiple other functions to perform smaller parts of their specific tasks
  - etc
- in larger programs, we might need to divide the overall program into collections of data types/functions, where the entire collection is needed to handle major subproblems

Subsystems, modules,etc

- terminology varies from developer to developer, but we often refer to different groupings of program components by size, e.g.
  - system: the whole thing (all files, functions, structs, etc)
  - subsystems: division of system into large collections of files, data structures, functions, etc
  - modules: division of an individual subsystem into smaller collections, for a specific set of tasks within the subsytem

Deciding how to decompose

- For each item we decompose, we need to consider the best place to do different parts of the task:
  - Where does data input, error checking take place
  - Where does data storage take place
  - Where does data transformation take place
  - Where does data output take place
- Want to get good balance of data processing, storage, and transmission based on the resources available
- Want to decompose in a way that is intuitive for the developer, so if they need to modify a feature it is easy to predict which components contain the relevant code

Balancing loads

- Many complex systems have both server-side and client-side processing
- Server-side might involve web servers, database servers, various processing servers (and possibly layers of gateways and mirrors)
- Client-side might involve running apps or programs on user devices, in browsers, etc
- Need to consider the storage and processing power of the different components, how much data we need to transfer between the components, and how sensitive the data is