# Software design

- In earlier courses you've been introduced to concepts of modularity and top down design
- Both ideas allow you to  focus on design of one part, while abstracting away the inner workings of the other parts
- e.g. while working on 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

# 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

# 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
- Will also look at cohesion and coupling as ways to evaluate the effectiveness of our decomposition

# 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
- When picking a design, need to think about 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

# Coupling and cohesion

- Two common metrics for evaluating a decomposition
- Coupling measures how tightly interconnected different components are: the more tightly interconnected, the more difficult it is to change one without also needing to change the other (i.e. we want loose coupling)
- Cohesion is how much the elements of a component logically belong together: does everything in the component truly have some unifying purpose (i.e. we want tight cohesion)

# Cohesion

- Often easier to recognize things that *aren't* cohesive: components that have some chunk of code stuck in just because we didn't really have a good idea of where else to put it
- Cohesion can be based on a variety of different unifying concepts: e.g. code that works on shared data may be grouped into a component, code that happens in a tight time sequence might be grouped together, code that interacts with a common user or other system may be grouped together, etc

# Design for the project priorities

- Our design should reflect the long term priorities of the project
- If the system is expected to be maintained for years, and to evolve/change substantially, then we need to account for that in our early design – think about adaptability
- If the system is expected to be fairly static but very resource limited (e.g. low memory, or poor connectivity, or limited processing power) then our design priority may sacrifice coupling/cohesion for efficiency

# Design documents

- As with requirements and specifications, designs are usually carefully documented in (surprise) a design document
- These will also be a mix of plain text, diagrams, examples, and pseudo-code, meant to give a clear and complete picture of the design
- As with requirements and specifications, it is important that design documents include the rationale/priorities upon which the design is based
- Clearly, design documents should also be thoroughly reviewed, and kept up to date as the design evolves

# Data flow diagrams (DFDs)

- Data flow diagrams are heirarchical collections of diagrams showing the top-down decomposition of a system into smaller and smaller components
- Typically they show each component, each data store (file, database, etc), each external system/user the component interacts with, and arrows indicating the flow of data between the elements of the DFD
- DFDs will typically be accompanied by detailed supporting text descriptions