Software Design Fundamentals

CSCE 247 - Lecture 12 - 02/27/2019

Today’s Goals

- Define design
- Introduce the design process
- Overview of design criteria
  - What results in a good design?

What is Design?

Design is the creative process of transforming a problem into a solution.

- In our case, transforming a requirements specification into a detailed description of the software to be implemented.

- Requirements - *what* we're going to build.
- Design - *how* to build it. A description of the structure of the solution.

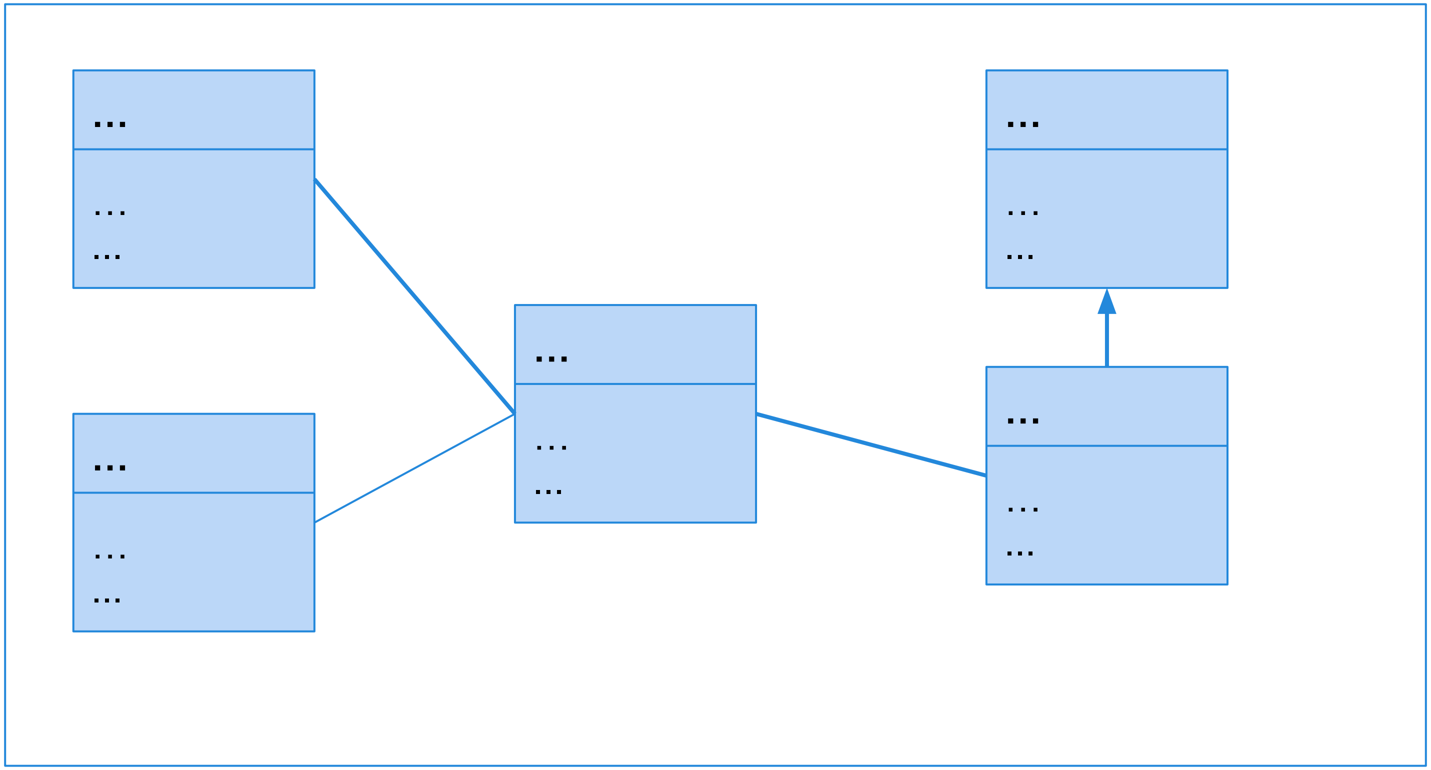
What is Design?

Design is the process of going from this:

Software

What is Design?

… to this:

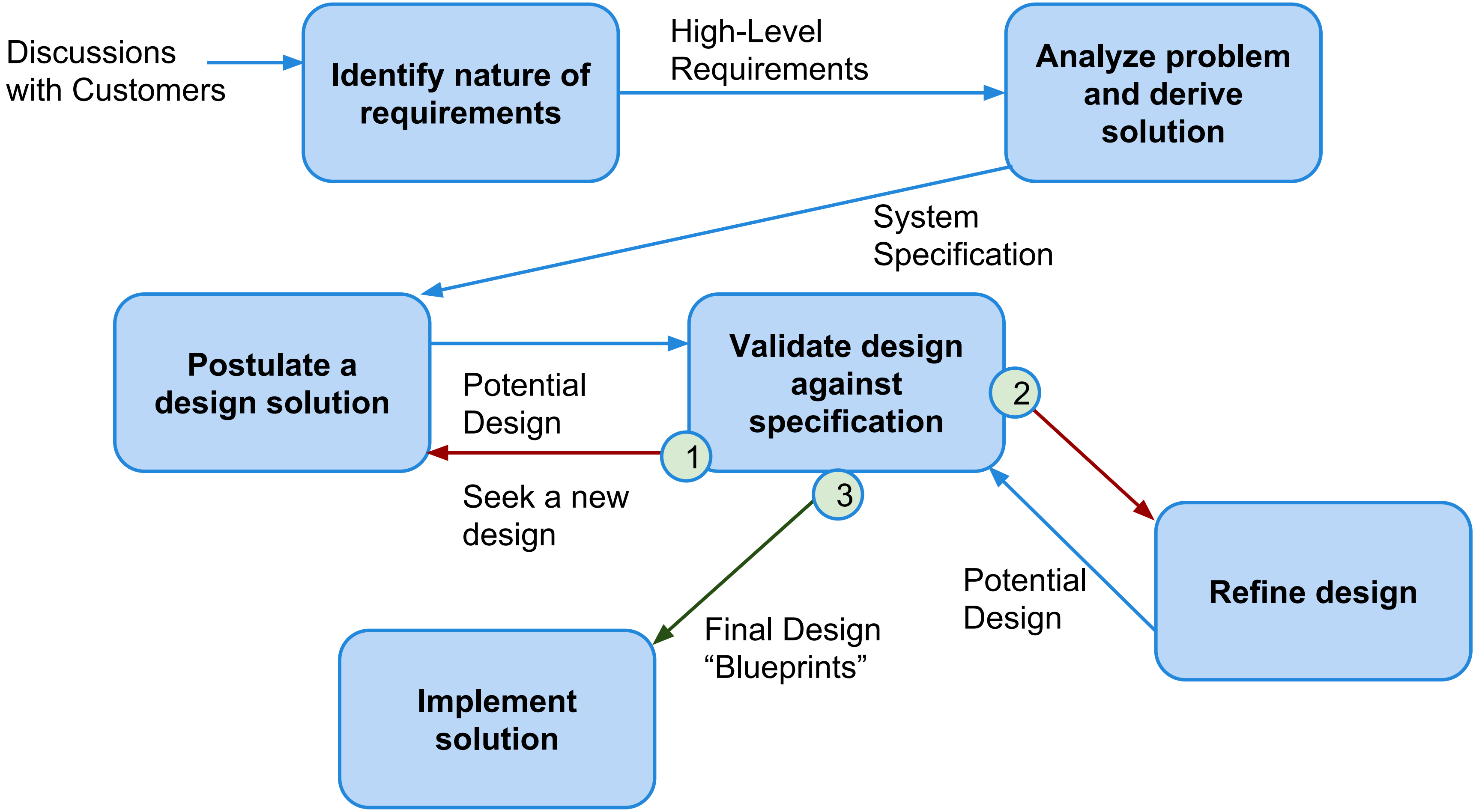


What is Design?

Design is the process of defining the *structure* of the software.

- What *units* make up the codebase?
- How do those units connect to perform the required functions?

General Design Stages



Stages of Design

Three repeating stages:

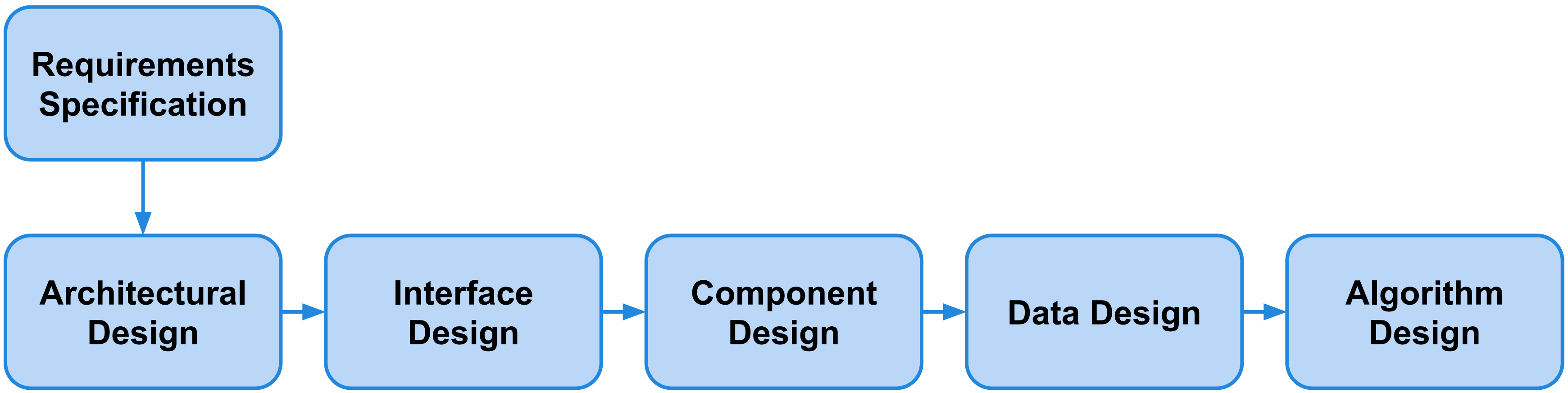
- Problem Understanding
  - Look at the problem from different angles to discover what needs the design needs to capture.
- Identify Solutions
  - Evaluate possible solutions and choose the most appropriate in terms of available resources.
- Describe and Document Chosen Solution
  - Use graphical, formal, or other descriptive notations to describe the components of the design.

Stages of Design

Design is performed at multiple levels of granularity:

- Architecture
  - How is the system structured into *subsystems*?
  - How do those subsystems work together?
- Unit
  - What *units* make up these subsystems?
  - How do these units work together?
- Low-Level
  - What algorithms will be employed?
  - What data structures will be used?

Design Activities



System architecture (high-level breakdown of system)

How subsystems interact with each other and how external users and systems interact with your system

A listing of the individual classes within your system

The format of data that is produced and consumed by your system

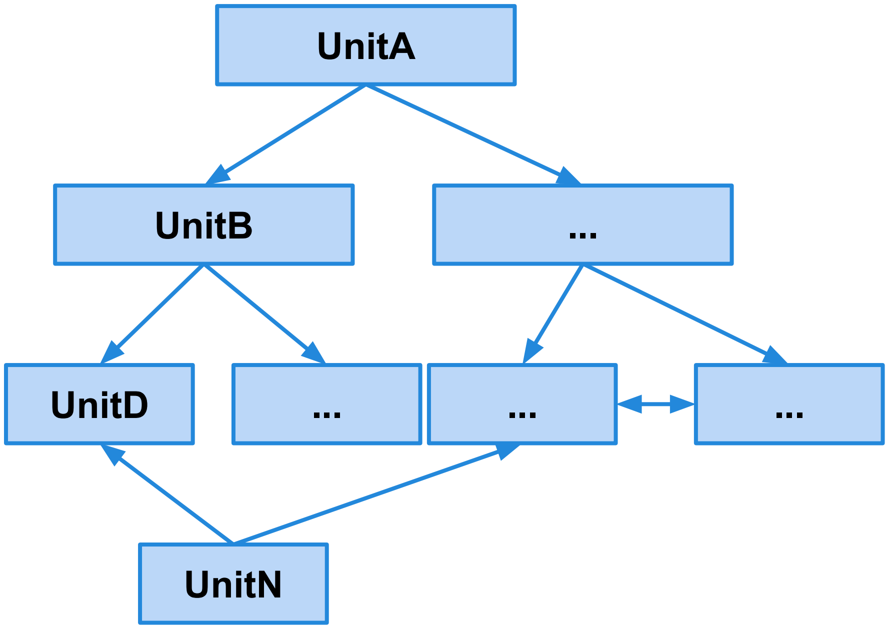
Algorithms used to implement system functionality.

The Design Process

- Design takes place in overlapping stages.
    - It is artificial to separate them into distinct phases. Some separation occurs, but these phases take place largely at the same time.
- In practice - design is an exercise filling in the missing details.
    - However, don’t forget about the big picture. Keep looking at all levels of abstraction to make sure you’re designing the right solution.

Basic Design Strategies

Design Strategies



Systems are typically designed as a hierarchy.

- UnitN provides a service used by UnitD.
- UnitD provides a service used by UnitB.
- UnitB provides a service used by UnitA.

Design strategies dictate how these units and their connections are laid out.

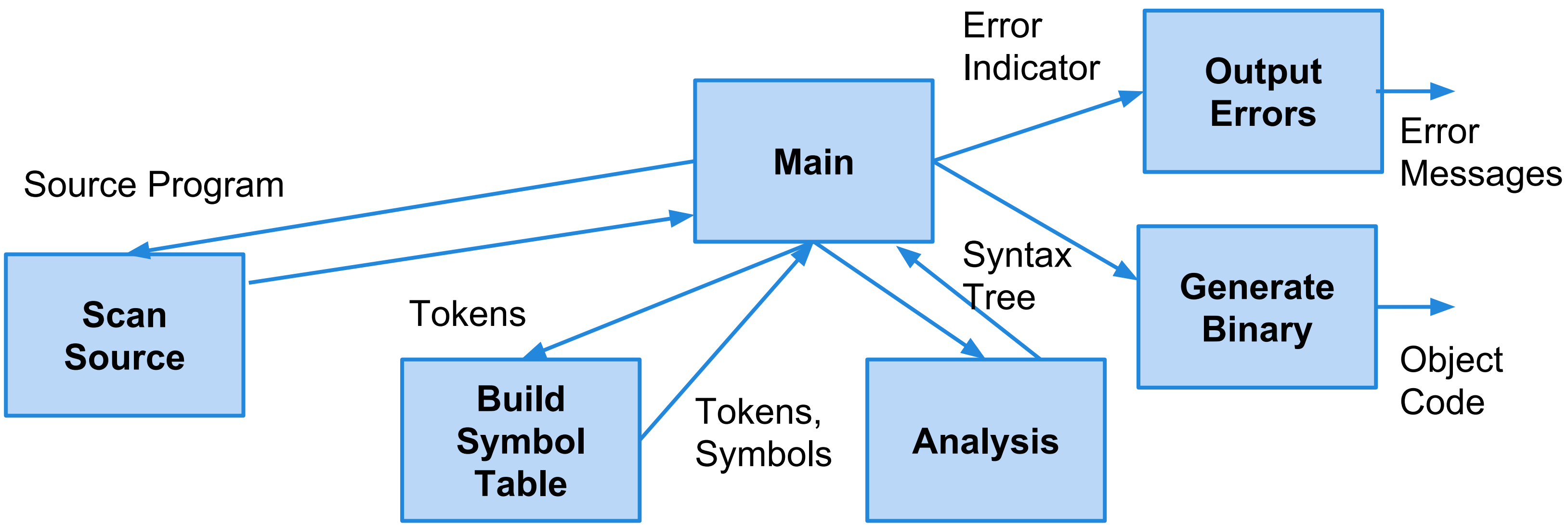
Centralized Design

- System is designed from a functional viewpoint: call and return model.
  - Typical in C and non-OO languages.
- Execution is controlled from a central point in the system.
  - A method is called, the result is passed back to the controlling location, then that is passed into the next method.
  - System is designed as a set of independent services that communicate only with a central master component.

Centralized Design

- The system state is centralized and shared between the functions operating on that state.
  - All data is stored by the master component.
  - Each called component receives all data it needs from the master.

Centralized View of a Compiler

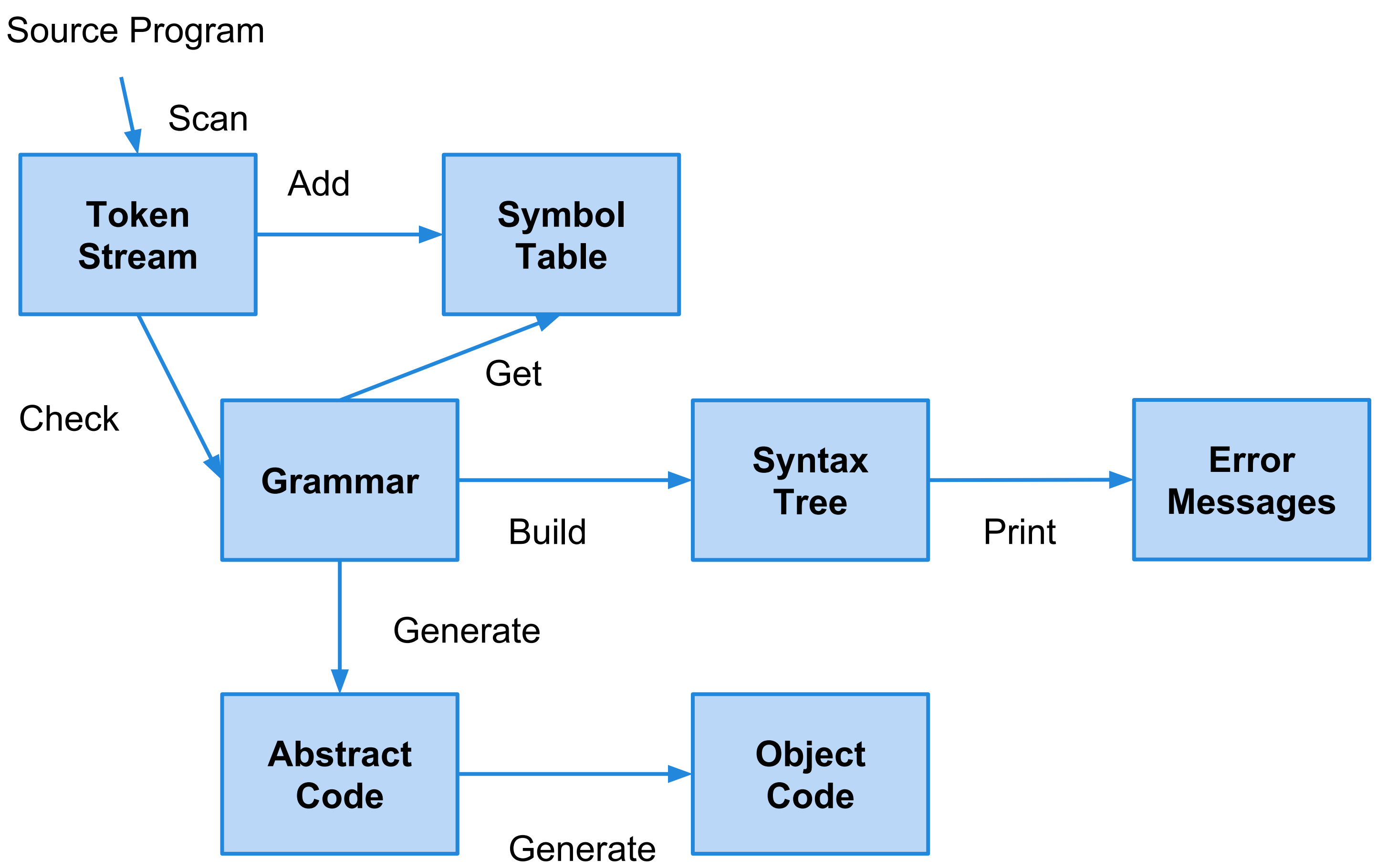



```
tokens = scanSource(program);
symbols=buildSymbolTable(tokens);
try{
    tree=analysis(tokens,symbols);
    generateBinary(tree);
catch(errors){
    print errors
}
```

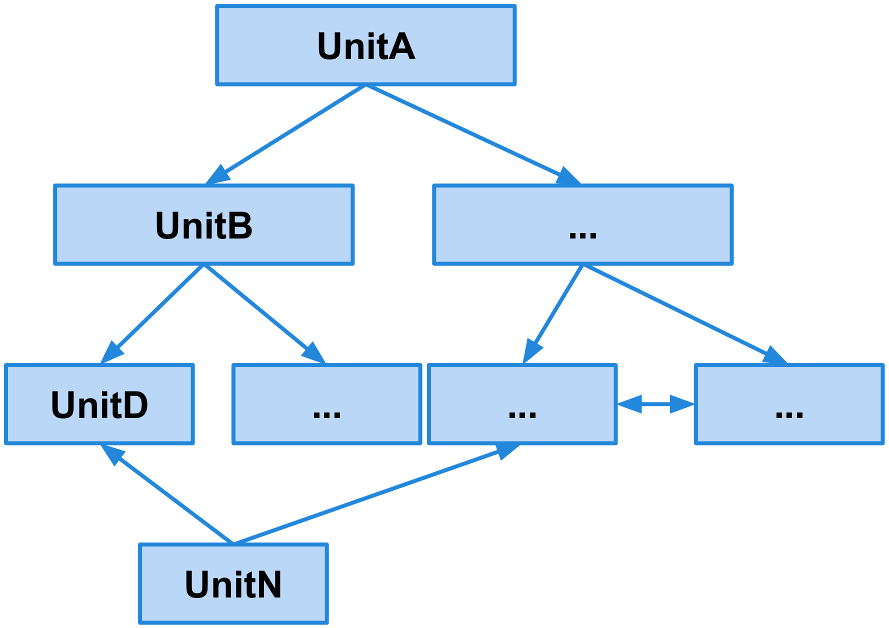
Decentralized Design

- Basis of object-oriented design
- System is designed as a collection of interacting components.
- System state is decentralized and each component manages its own data.
- Multiple instances of an component may exist and communicate.
- How most modern systems are designed.
  - Easier to isolate errors in one component.

Decentralized View of a Compiler



Design Strategies



Systems are typically designed as a hierarchy.

- Higher-level units make use of many lower-level units.
- Lower-level units tend to stand alone.
    - Small, self-contained, rarely call other components.

Top-Down Design

- In principle, top-down design involves starting at the uppermost components, design those, and work down the hierarchy level-by-level.

- Choose a major system function.
- Decide how to break it into components.
- Decide how to break those components into smaller subcomponents.

Top-Down Design

- In practice, large system design is never truly top-down.
  - Some branches are designed before others.
  - Designers reuse experience (and sometimes components) during the design process.
  - Sometimes, the lower levels need to be designed for the top-level to be completed.

Bottom-Up Design

- In principle,bottom-up design involves starting with standalone components, then assembling them into a complete system.

- In practice, large system design is never truly bottom-up.
  - An efficient system cannot be designed without planning for integration. The complete picture must be kept in mind.

Key Points

- Design is the process of deciding what components make up the software, and how they connect.
  - The *structure* of the software.
- Design activities include architectural design, interface design, component design, data design, and algorithm design.
  - But this is a messy process where phases overlap and activities cycle.

What are the criteria for a “good” design?

Design Quality

- No simple answer.
- Design quality is an elusive concept.
    - Depends on organizational priorities, and involves balancing competing objectives.
- A “good” design may be the most efficient, the cheapest, the most maintainable, the most reliable, etc…

Design Quality

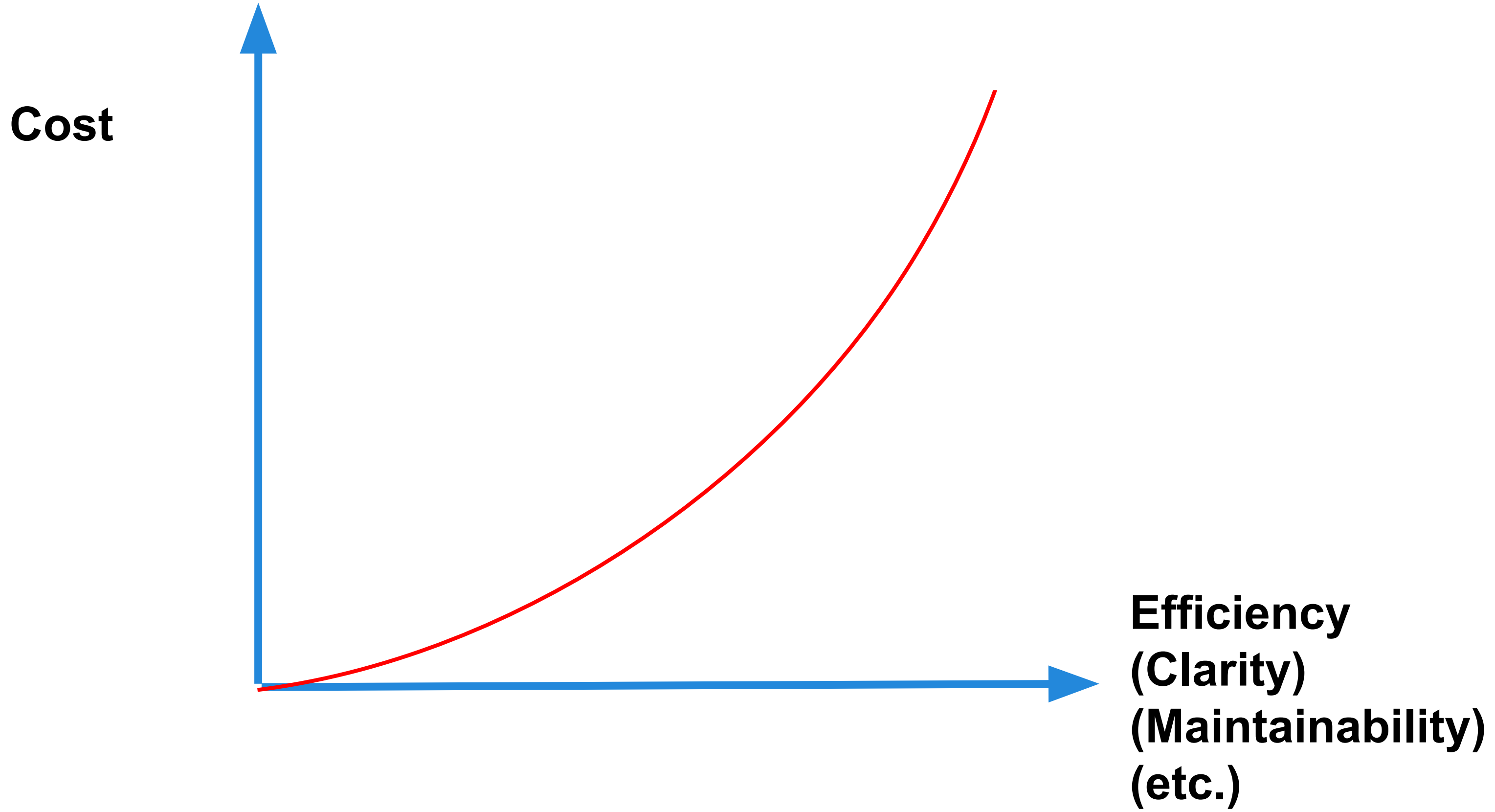
- A good design results in efficient software.
- Even more important...
- Software will change over time.
  - During implementation, after release.
- A good design **allows changes** to be made.
  - While also **protecting what works** from any side effects of those changes.

Design Attributes

- Simplicity
- Modularity
  - Low Coupling
  - High Cohesion
  - Information Hiding
  - Data Encapsulation
- Other “abilities”
  - Adaptability
  - Traceability

Expensive to Maximize Attributes

Costs rise exponentially if very high levels of an attribute are required.



Modularity

A complex system must be broken down into smaller modules.

Three goals of modularity:

- Decomposability
  - Break the system down into understandable modules.
- Composability
  - Construct a system from smaller pieces.
- Ease of Understanding
  - The system will change, we must understand it.

Modularity Properties

- Cohesion = The degree to which modules are compatible.
- Coupling = The degree of interdependence between modules.

We want **high** cohesion and **low** coupling.

Cohesion

- The degree to which modules are compatible. A measure of how well a component “fits together”.
- A component should implement a single logical entity or feature of the software.
- A high level of cohesion is a desirable design attribute because changes are localized to a single, cohesive component.

Types of Cohesion

- **Logical Cohesion (weak)**
  - Components that perform similar functions are grouped.
- **Temporal Cohesion (weak)**
  - Components that are activated at the same time are grouped.
- **Procedural Cohesion (weak)**
  - The elements in a component make up a single control sequence.
- **Sequential Cohesion (medium)**
  - The output for one part of a component is the input to another part.

Levels of Cohesion

- **Communicational Cohesion (medium)**
  - All of the elements of a component operate on the same input or produce the same output.
- **Functional Cohesion (strong)**
  - Each part of a component is necessary for the execution of a single system feature.
- **Object/Data Cohesion (strong)**
  - Each operation modifies or allows inspection of stored object attributes.
  - The class stores data and all operations performed on that data.

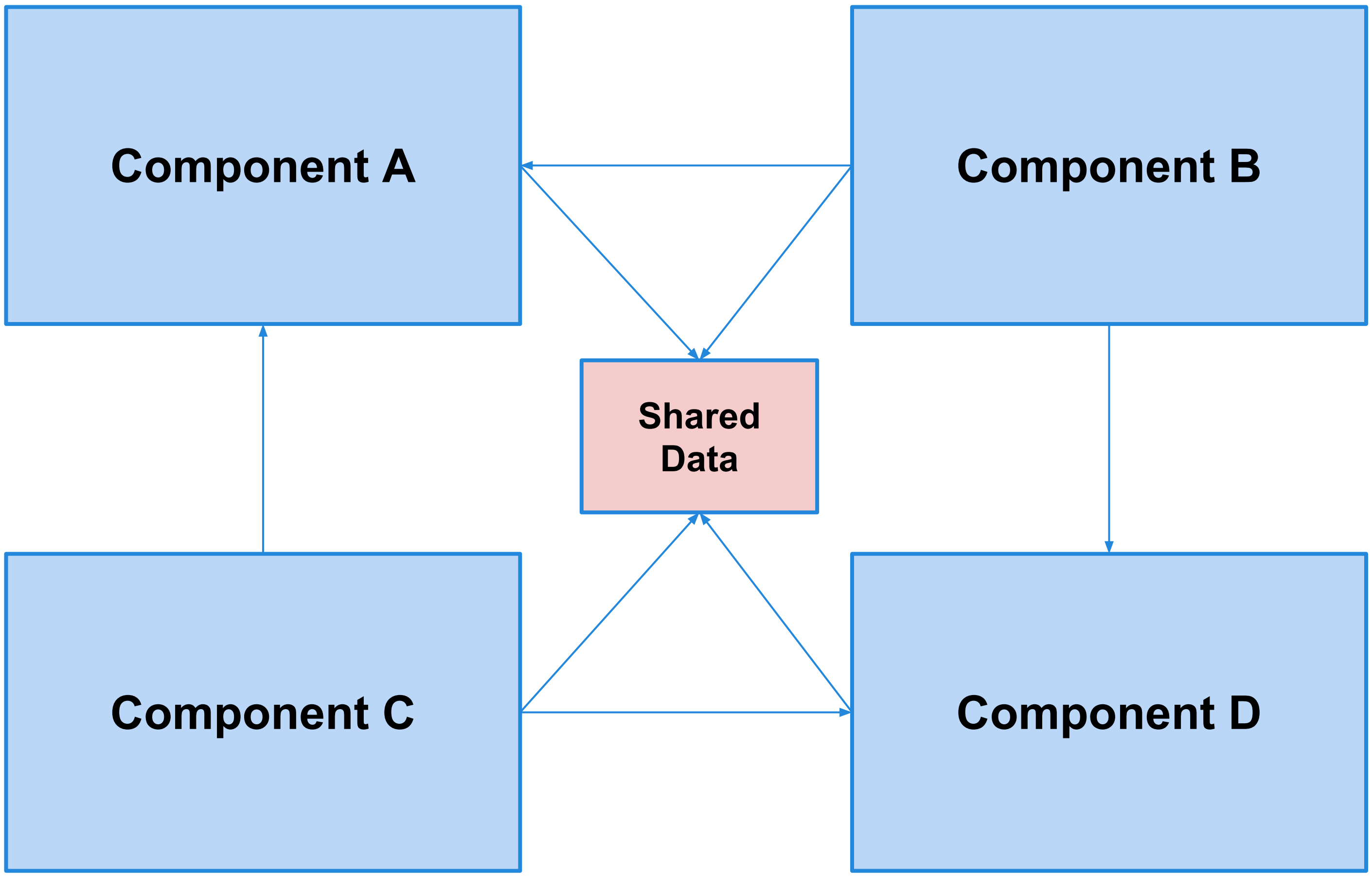
Cohesion as a Design Attribute

- Not well-defined.
  - Despite guidelines, cohesion is subjective and can't be easily measured.
  - Often very difficult to figure out what is related.
    - Some code is used by multiple classes.
- Inheriting attributes from super-classes weakens cohesion.
  - To understand a component, the super-classes as well as the component class must be examined.

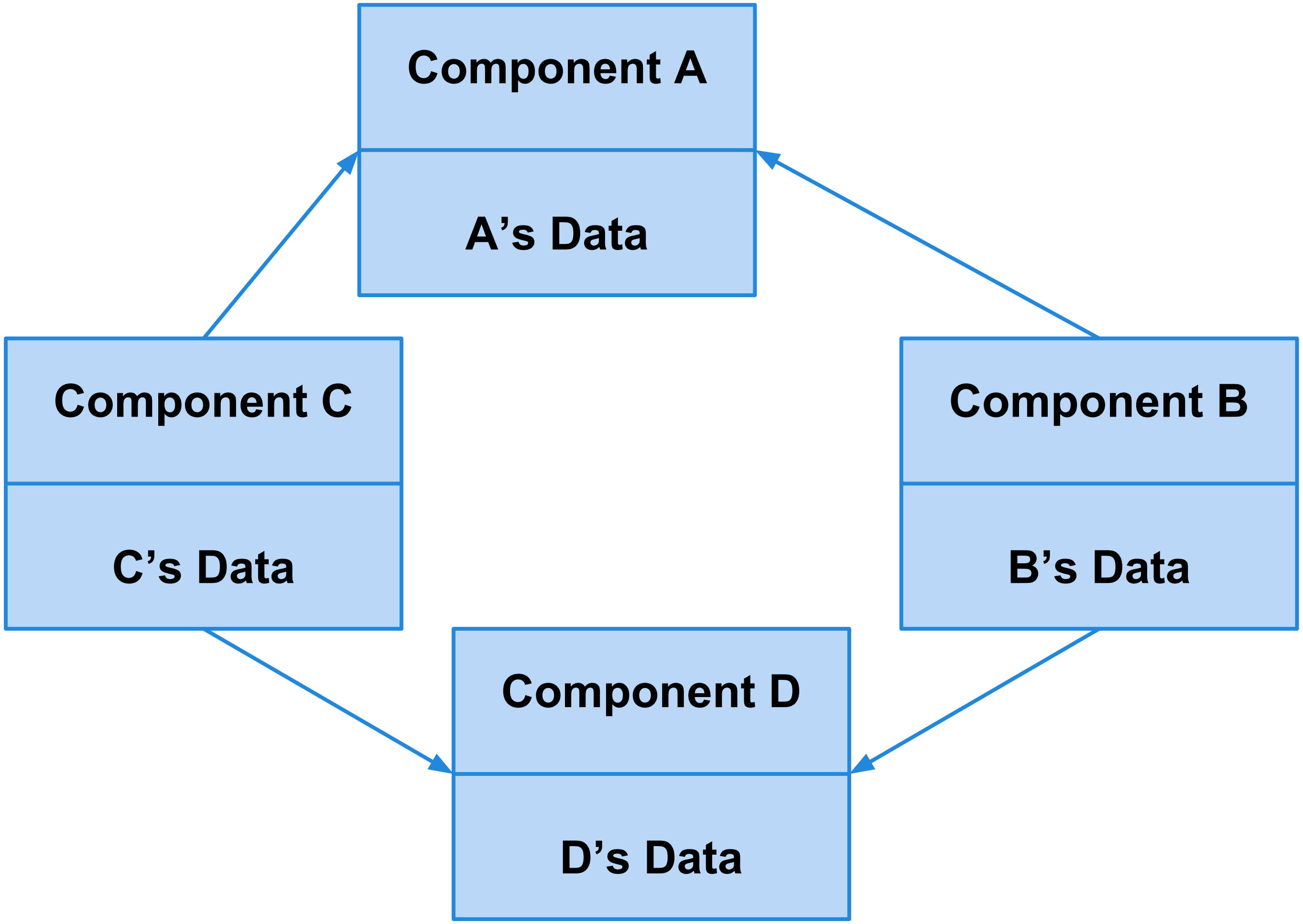
Coupling

- The degree of interdependence between modules. A measure of the strength of the interconnections between components.
  - Is code from another class called often?
  - How much data is passed during those calls?
- Loose coupling means component changes are unlikely to affect other components.
  - Loose coupling can be achieved by storing local data in objects and communicating solely by passing data through component's parameters.

Tight Coupling



Loose Coupling



Food for Thought

- How does an OO language like Java or C++ support low coupling and high cohesion?
  - How can we mess it up?
- How do global variables affect coupling?
- How about complex data structures?
  - … and pointers?
- What does inheritance do to coupling and cohesion?

Coupling and Inheritance

- Object-oriented systems can be loosely coupled because there is no need for shared state and objects communicate using message passing.
- However, an object class is coupled to its super-classes.
  - Changes made to the attributes or operations in a super-class propagate to all sub-classes. Such changes must be carefully controlled.

Information Hiding

- Put the complexity inside of a “black box”
  - Hide it from the components that use that “box”.
  - The user does not need to know *how* the box works, just *what* it does.
- Greatly reduces the amount of information the designer needs to understand at once.
- Examples:
  - Functions, Interfaces, Classes, Libraries
- If used properly, helps ensure loose coupling.

Information Hiding Example

```
int[] sortAscending(int[] unsorted, int length);
```

- We do not know what sort routine is used.
- All we know is what the interface is and what the module accomplishes.
- Allows designers to focus on one part of the system at a time, without worrying about other components.

Data Encapsulation

- Encapsulation is the principle of building a barrier around a collection of items.
- Encapsulate the data a module is working on.
  - Protect the data from unauthorized access.
  - Nobody else can mess with the data.
  - If it gets corrupted, it must have been the fault of this component.
- Makes the design more robust.

Encapsulation Example

Version 1:

```
class Adder{

      int total;
      void addNum(int number){
            total += number;
      }
};


int main( )
{
   Adder a;

   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.total <<endl;
   return 0;
}
```

Version 2:

```
class Adder{

      private int total;
      void addNum(int number){
            total += number;
      }

      int getTotal(){

            return total;

      }
};
int main( )
{
   Adder a;
   a.addNum(10);
   a.addNum(20);
   a.addNum(30);
   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

Understandability

The design should be understandable by the developers - unambiguous and easy to follow. Related to many component characteristics:

- Cohesion
  - Can each component be understood on its own?
- Naming
  - Are meaningful component (class, method, variable) names used?
- Documentation
  - Is the design well-documented? Are decisions justified? Rationale noted?
- Complexity
  - Are complex algorithms used?

Understandability

- High complexity means many relationships between different entities in the design.
  - Hence, the design is hard to understand.
- Most “measurements” of design quality measure the complexity.
  - They tell you to avoid high complexity (high number of relations between components).
  - These metrics tend to be of little use - the number is irrelevant - instead, be careful to only include necessary relations.

Adaptability

- A design is adaptable if:
  - Its components are loosely coupled.
  - It is well-documented and the documentation is kept up to date.
  - There is an obvious correspondence between design levels (interface, components, data, etc).
  - Each component is a self-contained entity (strong cohesion).
- To adapt a design, it must be possible to trace links between components so that change consequences can be analyzed.

Adaptability and Inheritance

Inheritance improves adaptability.

- Components may be expanded without change by deriving a sub-class and modifying that derived class.
- However, as the depth of the inheritance hierarchy increases, so does complexity.
  - Complexity must be periodically reviewed and restructured.

Design Traceability

For a design to be adaptable and understandable, we must be able to link:

- Components to their data.
- Components to their related components.
- Data to related data.
- Components to their requirements.
- Components to their test cases.

We Have Learned

- Design is the process of deciding what components make up the software, and how they connect.
  - The *structure* of the software.
- A good design allows change while protecting unchanged components.
- Coupling and cohesion are central to good software design.
  - Always keep these in mind.

Next Time

- Midterm Review!
  - Practice midterm on site - try it out.
  - We will go over answers next time.

- Homework 2 due March 3rd
  - Questions?