```
   // if the error code isn't valid, notify the user that an
   // internal error has been detected

   // return status information
}
```

This is a start on the code. The variable *errorMessage* is used, so it needs to be declared. If you were commenting after the fact, two lines of comments for two lines of code would nearly always be overkill. In this approach, however, it's the semantic content of the comments that's important, not how many lines of code they comment. The comments are already there, and they explain the intent of the code, so leave them in.

The code below each of the remaining comments needs to be filled in:

C++ Example of a Complete Routine Created with the Pseudocode Programming Process

```
/* This routine outputs an error message based on an error code
supplied by the calling routine. The way it outputs the message
depends on the current processing state, which it retrieves
on its own. It returns a value indicating success or failure.
*/

Status ReportErrorMessage(
   ErrorCode errorToReport
   ) {
   // set the default status to "fail"
   Status errorMessageStatus = Status_Failure;

   // look up the message based on the error code
   Message errorMessage = LookupErrorMessage( errorToReport );

   // if the error code is valid
   if ( errorMessage.ValidCode() ) {
      // determine the processing method
      ProcessingMethod errorProcessingMethod = CurrentProcessingMethod();

      // if doing interactive processing, display the error message
      // interactively and declare success
      if ( errorProcessingMethod == ProcessingMethod_Interactive ) {
         DisplayInteractiveMessage( errorMessage.Text() );
         errorMessageStatus = Status_Success;
      }

      // if doing command line processing, log the error message to the
      // command line and declare success
      else if ( errorProcessingMethod == ProcessingMethod_CommandLine ) {
         CommandLine messageLog;
         if ( messageLog.Status() == CommandLineStatus_Ok ) {
            messageLog.AddToMessageQueue( errorMessage.Text() );
            messageLog.FlushMessageQueue();
            errorMessageStatus = Status_Success;
         }
```

The code for each comment has been filled in from here down.

This code is a good candidate for being further decomposed into a new routine: *DisplayCommandLine-Message()*.

This code and comment are new and are the result of fleshing out the *if* test.

This code and comment are also new.

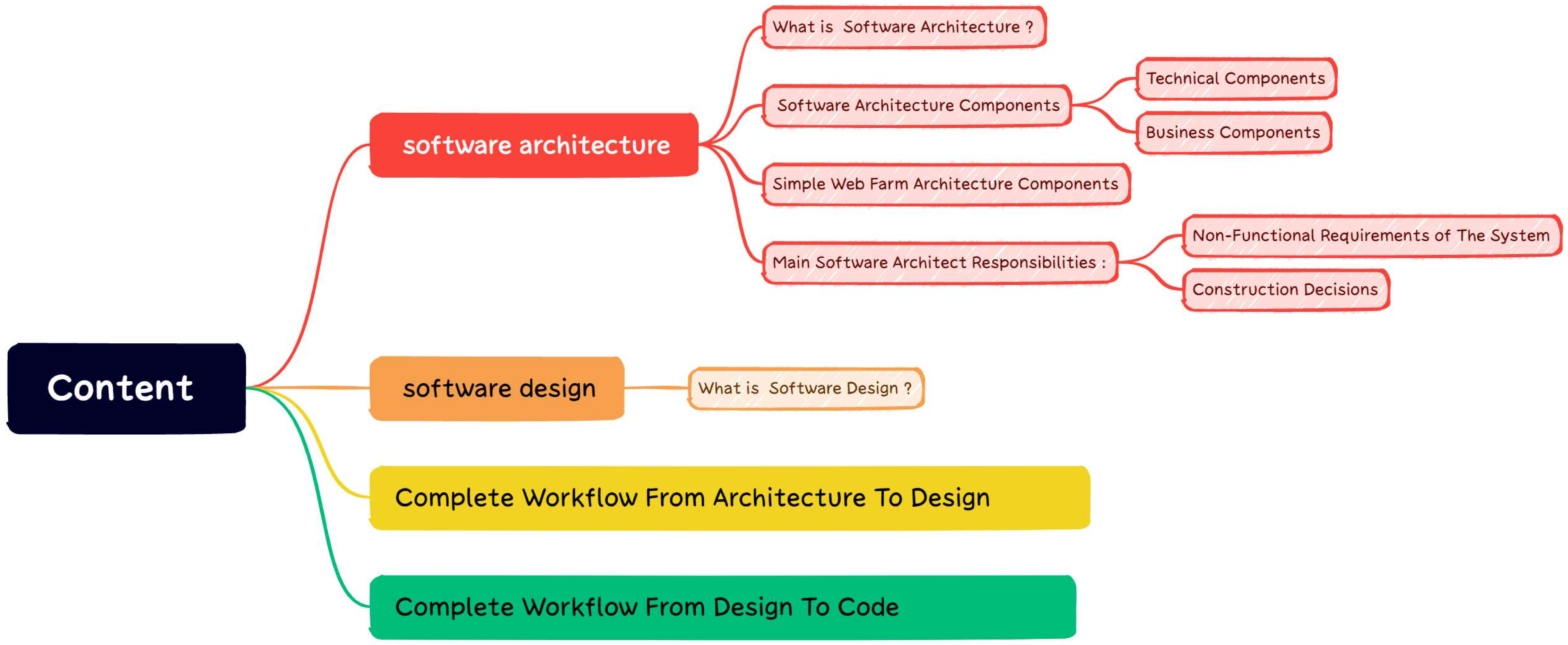
```
         else {
            // can't do anything because the routine is already error processing
         }
      else {
         // can't do anything because the routine is already error processing
      }
   }

   // if the error code isn't valid, notify the user that an
   // internal error has been detected
   else {
      DisplayInteractiveMessage(
         "Internal Error: Invalid error code in ReportErrorMessage()"
      );
   }

   // return status information
   return errorMessageStatus;
}
```

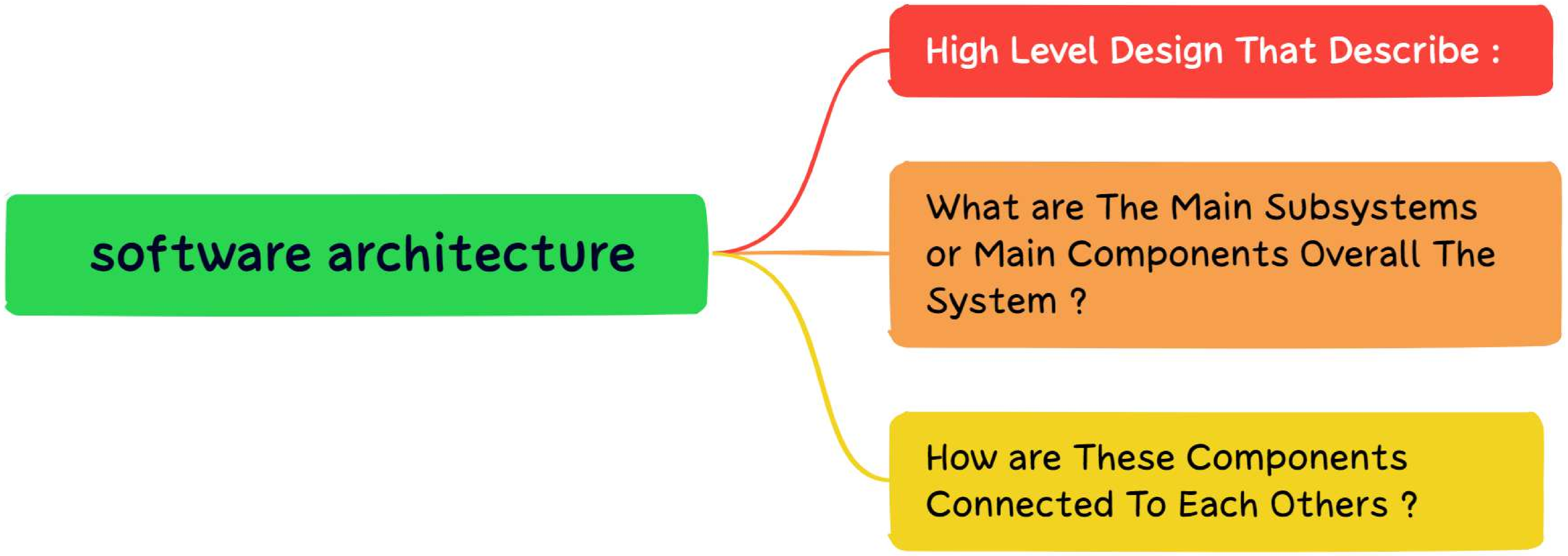
Each comment has given rise to one or more lines of code. Each block of code forms a complete thought based on the comment. The comments have been retained to provide a higher-level explanation of the code. All variables have been declared and defined close to the point they're first used. Each comment should normally expand to about 2 to 10 lines of code. (Because this example is just for purposes of illustration, the code expansion is on the low side of what you should usually experience in practice.)

Now look again at the spec on page 221 and the initial pseudocode on page 224. The original five-sentence spec expanded to 15 lines of pseudocode (depending on how you count the lines), which in turn expanded into a page-long routine. Even though the spec was detailed, creation of the routine required substantial design work in pseudocode and code. That low-level design is one reason why "coding" is a nontrivial task and why the subject of this book is important.

Part-1 software architecture and design

- Ch-1 Basics Of software architecture and design


Content
software architecture
What is Software Architecture ?
Software Architecture Components
Technical Components
Business Components
Simple Web Farm Architecture Components
Main Software Architect Responsibilities :
Non-Functional Requirements of The System
Construction Decisions
software design
What is Software Design ?
Complete Workflow From Architecture To Design
Complete Workflow From Design To Code



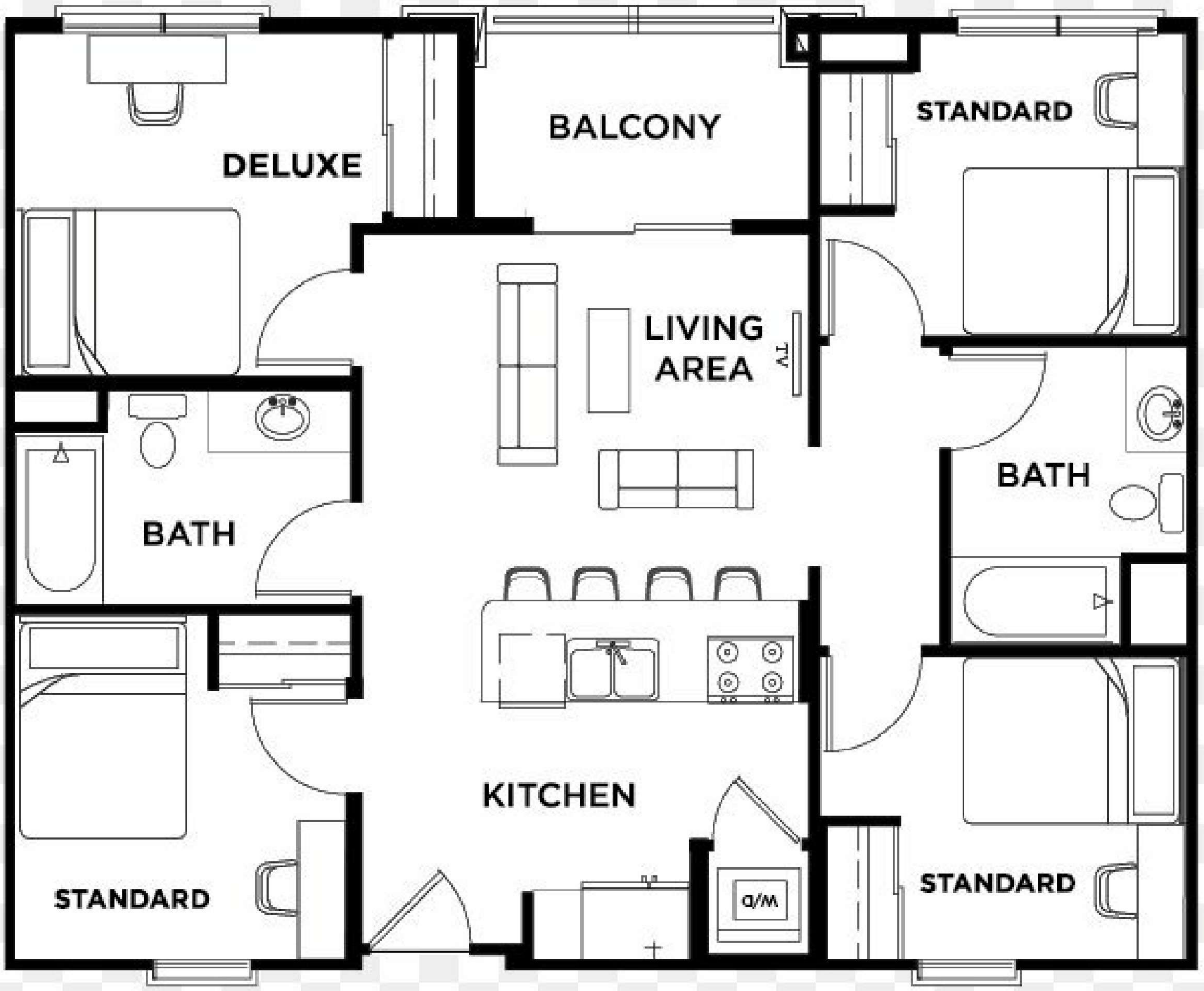
software architecture

High Level Design That Describe :

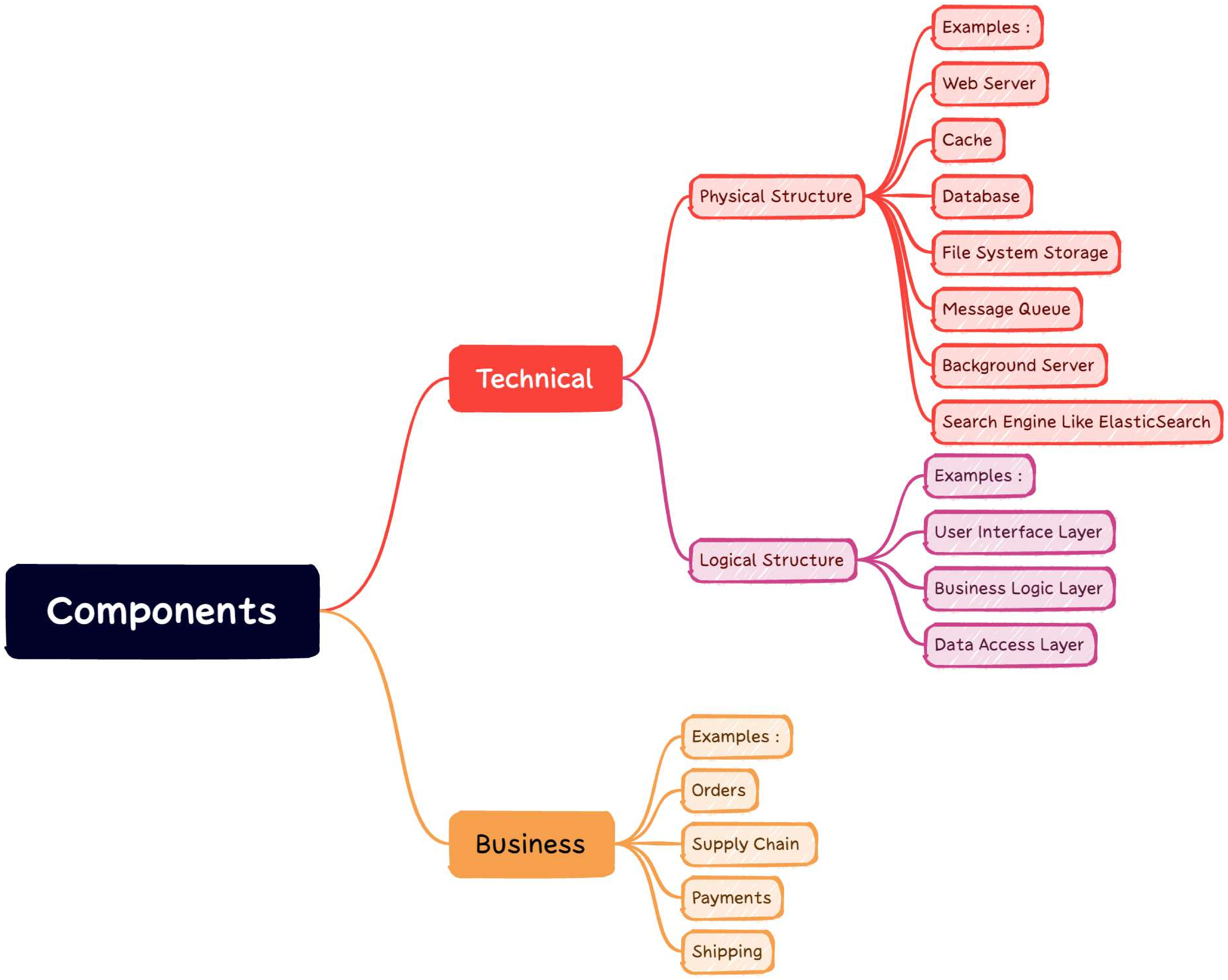
What are The Main Subsystems or Main Components Overall The System ?

How are These Components Connected To Each Others ?

High Level Details : Software Architecture



Low Level : Software Design

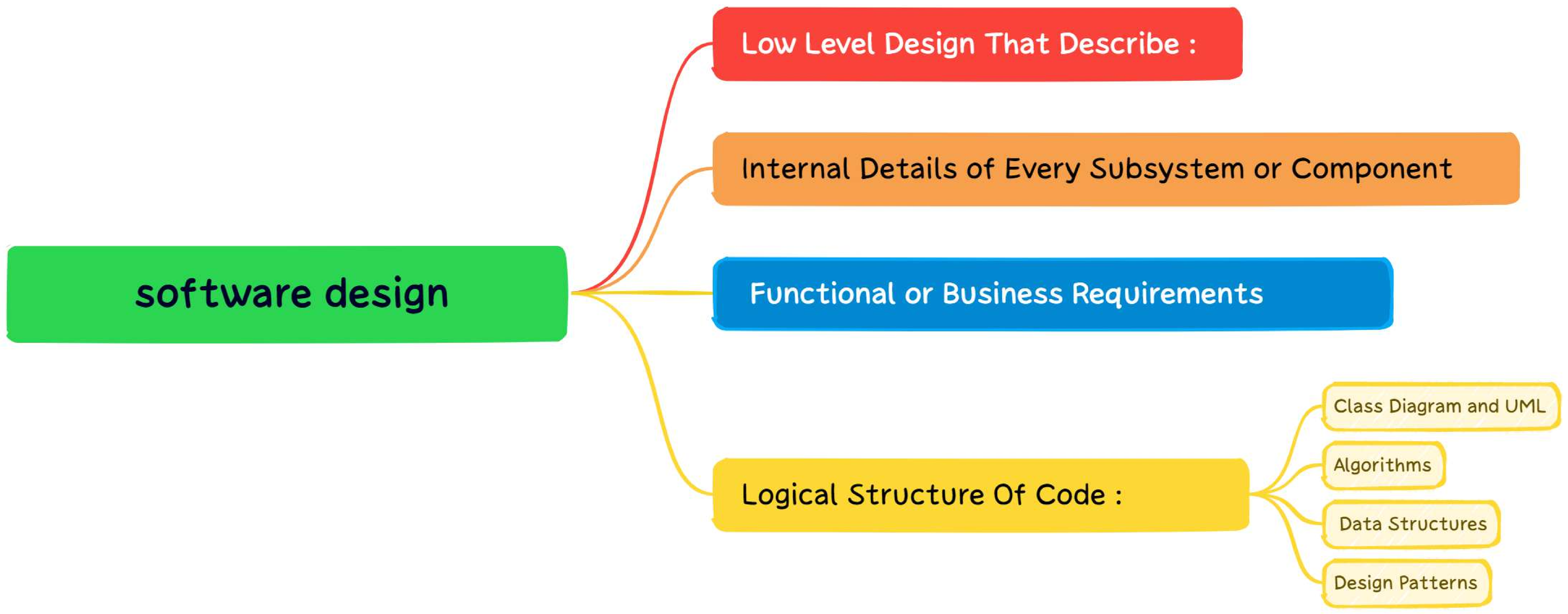

Components
Technical
Physical Structure
Examples :
Web Server
Cache
Database
File System Storage
Message Queue
Background Server
Search Engine Like ElasticSearch
Logical Structure
Examples :
User Interface Layer
Business Logic Layer
Data Access Layer
Business
Examples :
Orders
Supply Chain
Payments
Shipping

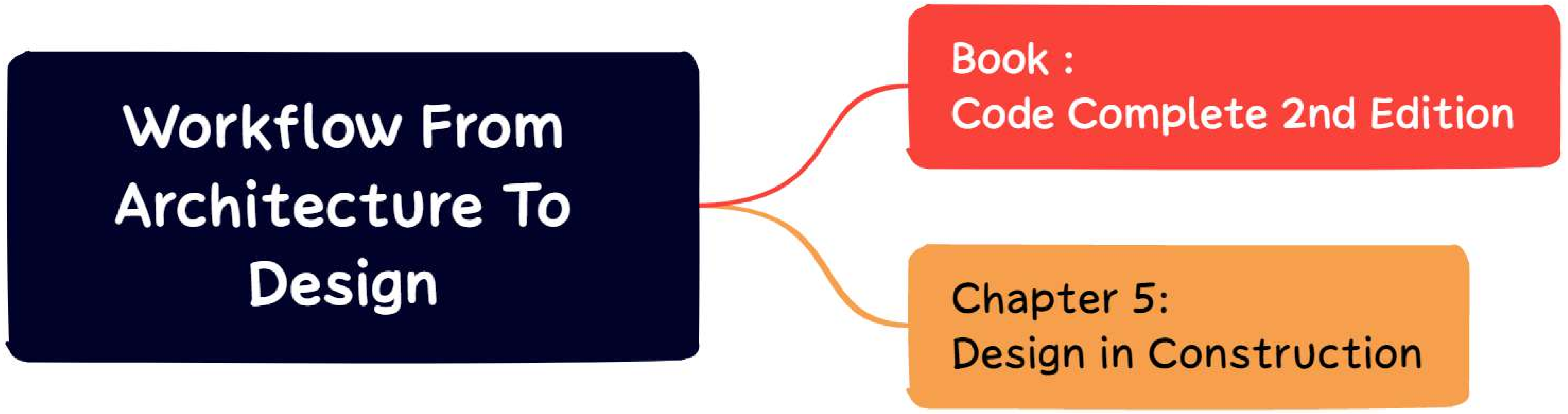
Simple Web Farm Architecture Components

android
iPhone
Load Balancer
Web Server 1
User Interface
Business Logic
Data Access
Web Server 2
User Interface
Business Logic
Data Access
File System Storge
Cache
Database
Elastic Search
Message Queue
Consumer 1
Consumer 2
Background Server

- **Main Software Architect Responsibilities**
  - **Non-Functional Requirements of The System**
    - Examples :
    - Latency
      - The Time Needed By The System To Do Single Operation
      - Ex : User Must Be Can Publish Post On Facebook With Time Less Than One Sec At Max
    - Throughput
      - The Total Number Of Operations That The System Can Do At The Same Time
      - Ex :
        Throughput : System Can Handle 100,000,000 Posts At One Sec
    - Bandwidth
      - The Total Size Of Data That System Can Transfer or Process At The Same Time
      - Ex : Total Bandwidth = 200,000,000 Megabytes So 100,000,000 Users Can Publish Posts With Max Size For Every Post Equal 2 Megabyte.
    - Capacity
      - The Total Size Of Data That The System Can Stored
      - Ex :
        Active Users : 100,000,000 User
        Max Post Size : 2 Megabyte
        Max Number Of Posts By User Per Day : 10
        Total Capacity Needed By The System For Every One Month = 100,000,000 * 2 * 10 * 30 = 60,000,000,000 Megabytes
    - Scalability
      - How The Systems Can Be Scale To Serve More Active Users ?
        Current Total Active Users = 100,000,000
        New Users Per Day = 10,000
        Total Active Users After One Month = 100,000,000 + (30 * 100,000) = 103,000,000 Users
      - How The Systems Can Be Scale To Store More Data (capacity) ?

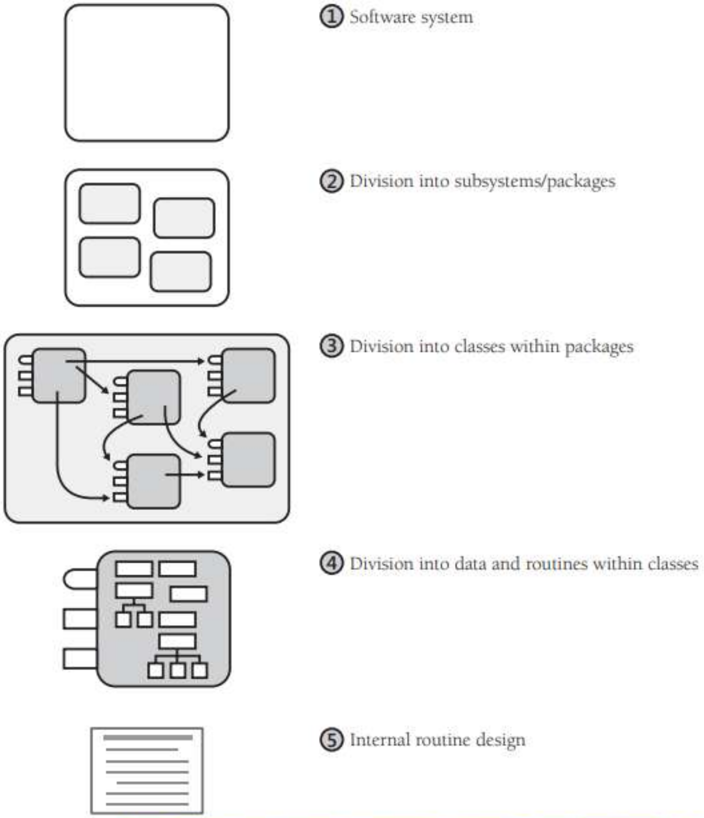
Main Software Architect Responsibilities
Construction Decisions
Tools :
examples :
Programming Languages
ex : c# vs python
UI Frameworks
ex : Angular vs React
Database
ex : SQL vs NoSQL
Message Queue
ex : RabbitMQ VS Kafka
Background Server
ex : Hangfire vs Quartz.Net
Software Architecture Styles and Patterns
examples :
monolithic vs distributed
database centric vs domain centric
Synchronous Communications vs Asynchronous Communications
We Will Discuss Software Architecture Styles and Patterns in Details Later


software design
Low Level Design That Describe :
Internal Details of Every Subsystem or Component
Functional or Business Requirements
Logical Structure Of Code :
Class Diagram and UML
Algorithms
Data Structures
Design Patterns


Workflow From Architecture To Design
Book : Code Complete 2nd Edition
Chapter 5: Design in Construction

Levels of Design

Design is needed at several different levels of detail in a software system. Some design techniques apply at all levels, and some apply at only one or two. Figure 5-2 illustrates the levels.



① Software system
② Division into subsystems/packages
③ Division into classes within packages
④ Division into data and routines within classes
⑤ Internal routine design

Figure 5-2 The levels of design in a program. The system (1) is first organized into subsystems (2). The subsystems are further divided into classes (3), and the classes are divided into routines and data (4). The inside of each routine is also designed (5).

Level 1: Software System

The first level is the entire system. Some programmers jump right from the system level into designing classes, but it's usually beneficial to think through higher level combinations of classes, such as subsystems or packages.

Level 2: Division into Subsystems or Packages

The main product of design at this level is the identification of all major subsystems. The subsystems can be big: database, user interface, business rules, command interpreter,

report engine, and so on. The major design activity at this level is deciding how to partition the program into major subsystems and defining how each subsystem is allowed to use each other subsystem. Division at this level is typically needed on any project that takes longer than a few weeks. Within each subsystem, different methods of design might be used–choosing the approach that best fits each part of the system. In Figure 5-2, design at this level is marked with a 2.

Of particular importance at this level are the rules about how the various subsystems can communicate. If all subsystems can communicate with all other subsystems, you lose the benefit of separating them at all. Make each subsystem meaningful by restricting communications.

Suppose for example that you define a system with six subsystems, as shown in Figure 5-3. When there are no rules, the second law of thermodynamics will come into play and the entropy of the system will increase. One way in which entropy increases is that, without any restrictions on communications among subsystems, communication will occur in an unrestricted way, as in Figure 5-4.

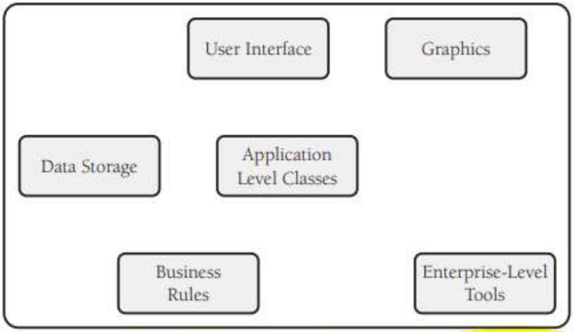


Figure 5-3 An example of a system with six subsystems.

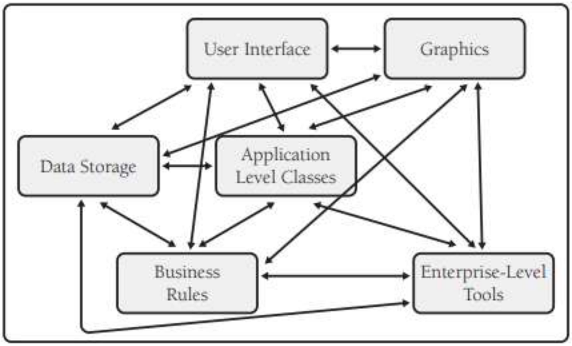


Figure 5-4 An example of what happens with no restrictions on intersubsystem communications.

As you can see, every subsystem ends up communicating directly with every other subsystem, which raises some important questions:

- How many different parts of the system does a developer need to understand at least a little bit to change something in the graphics subsystem?
- What happens when you try to use the business rules in another system?
- What happens when you want to put a new user interface on the system, perhaps a command-line UI for test purposes?
- What happens when you want to put data storage on a remote machine?

You might think of the lines between subsystems as being hoses with water running through them. If you want to reach in and pull out a subsystem, that subsystem is going to have some hoses attached to it. The more hoses you have to disconnect and reconnect, the more wet you're going to get. You want to architect your system so that if you pull out a subsystem to use elsewhere, you won't have many hoses to reconnect and those hoses will reconnect easily.

With forethought, all of these issues can be addressed with little extra work. Allow communication between subsystems only on a "need to know" basis—and it had better be a *good* reason. If in doubt, it's easier to restrict communication early and relax it later than it is to relax it early and then try to tighten it up after you've coded several hundred intersubsystem calls. Figure 5-5 shows how a few communication guidelines could change the system depicted in Figure 5-4.

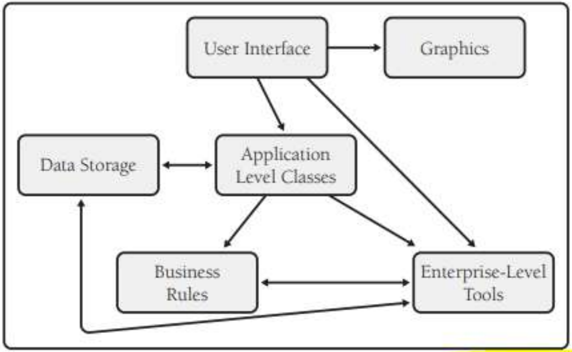


Figure 5-5 With a few communication rules, you can simplify subsystem interactions significantly.