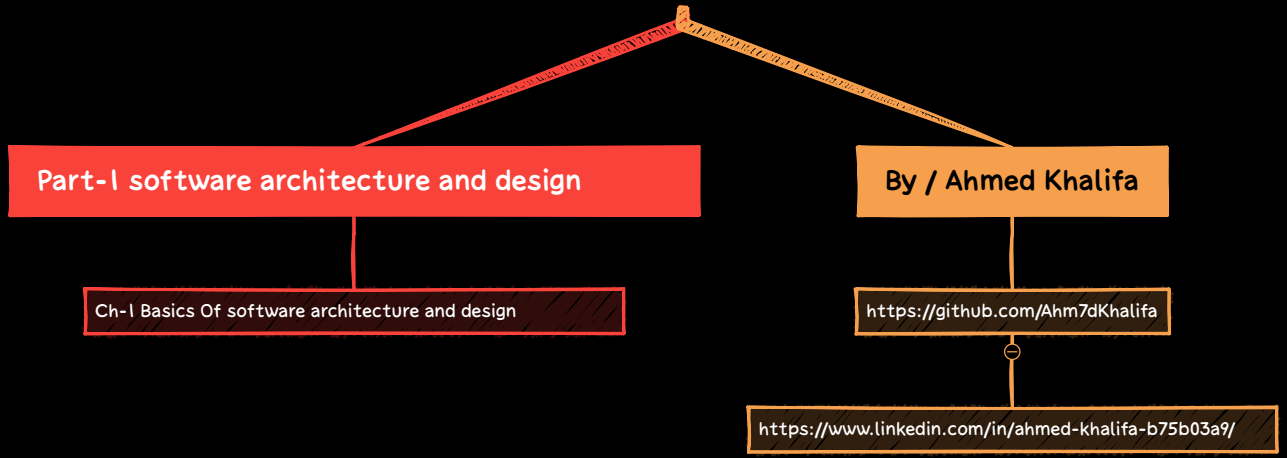
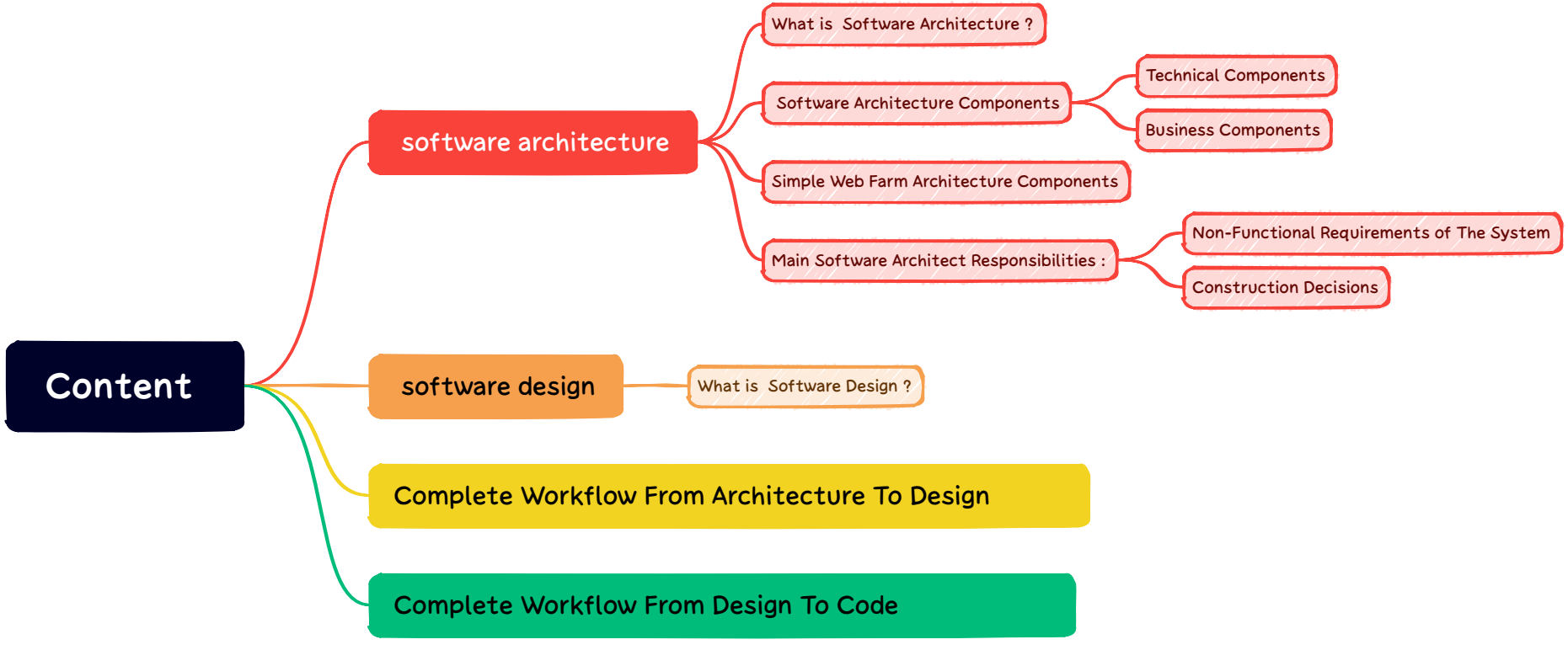
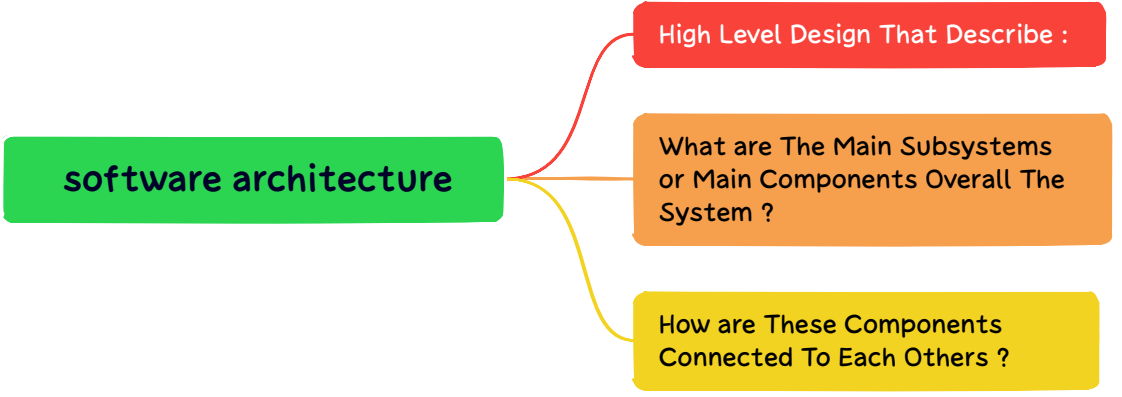
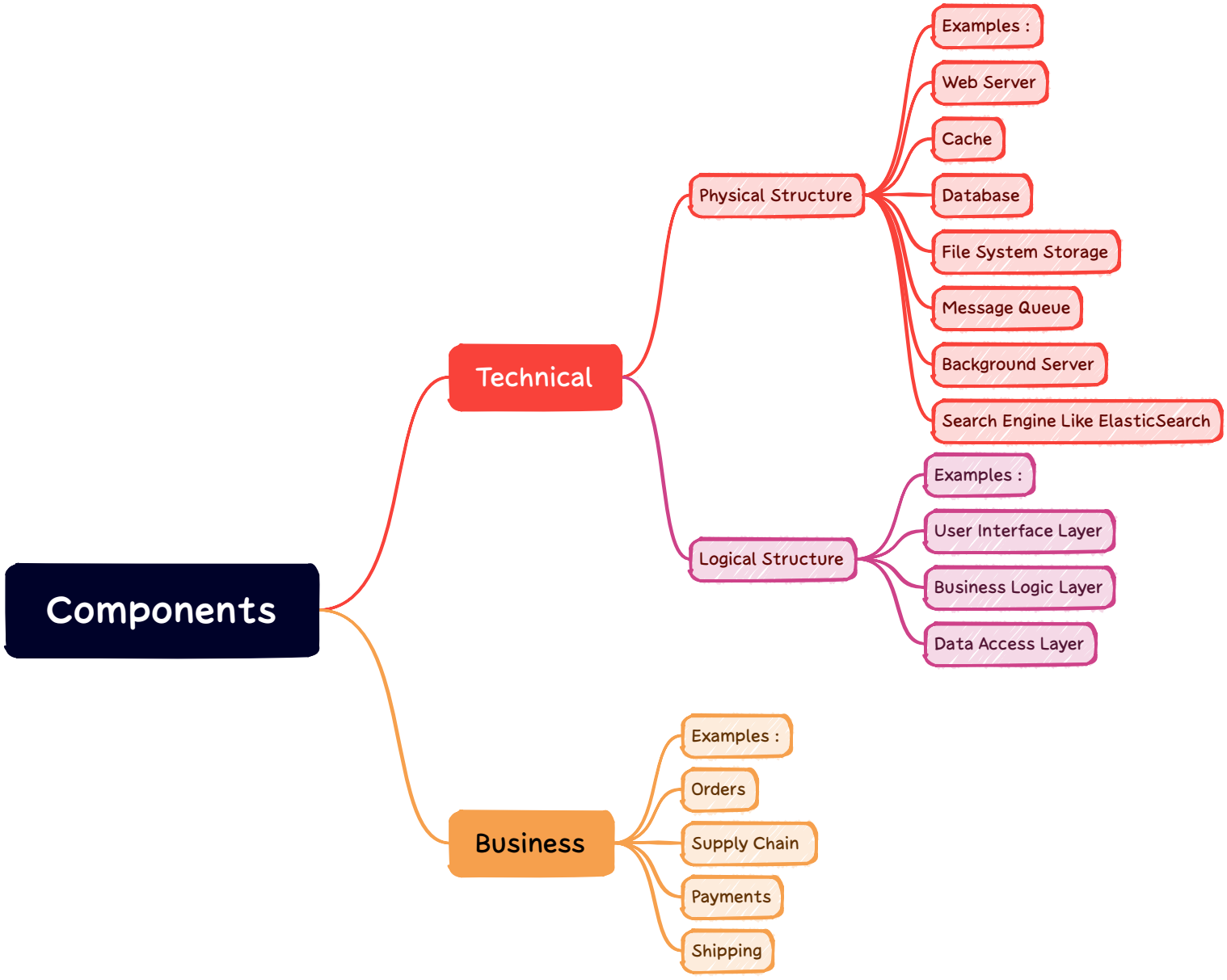
System Design

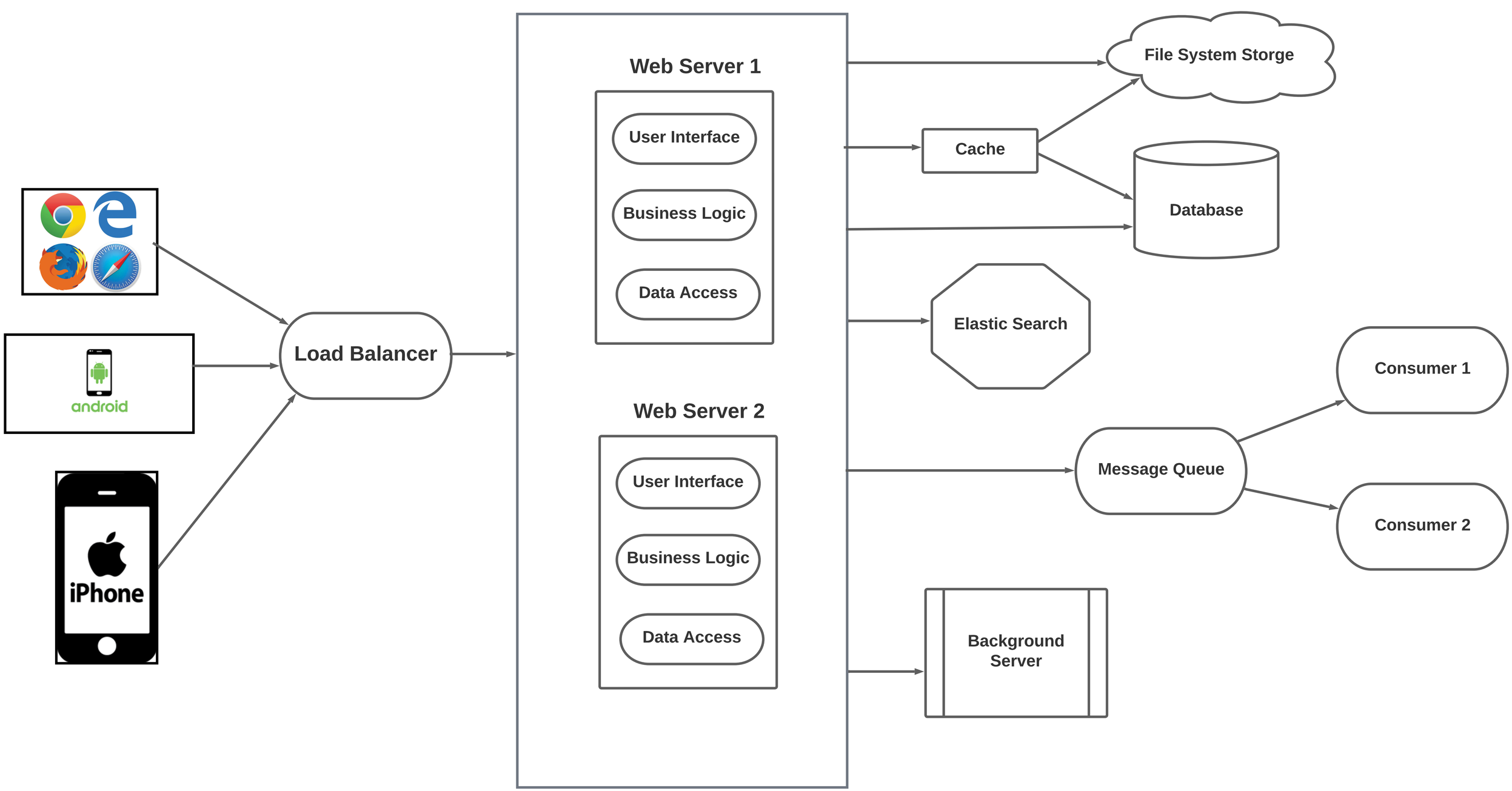








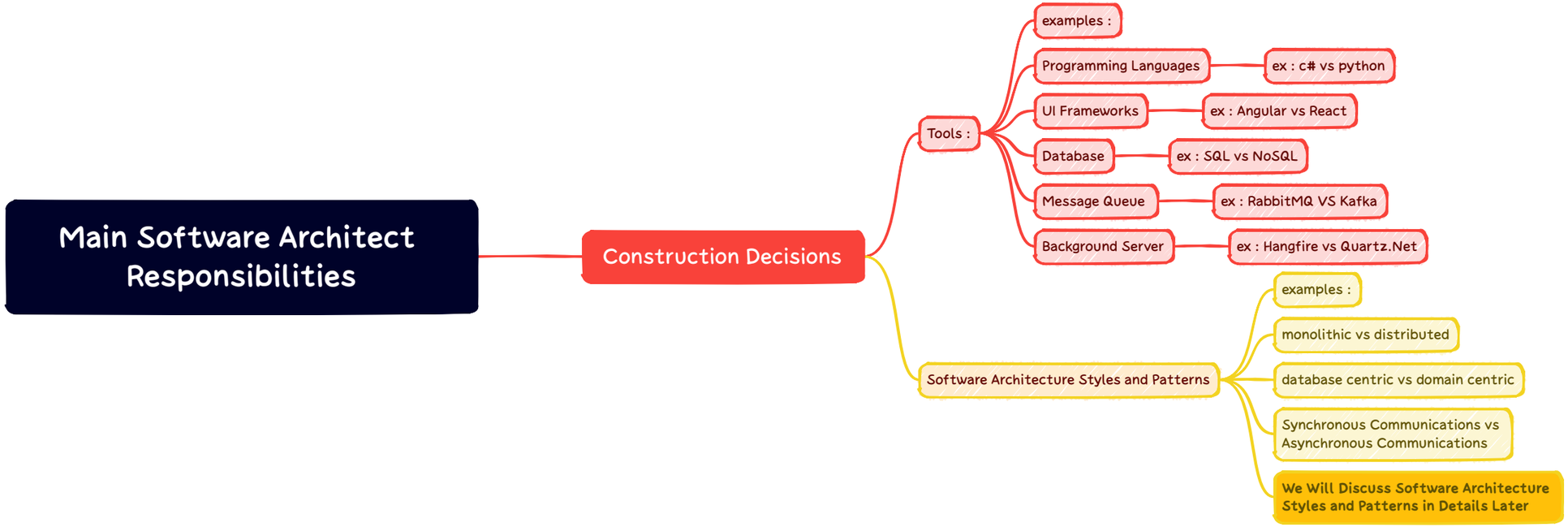
Simple Web Farm Architecture Components

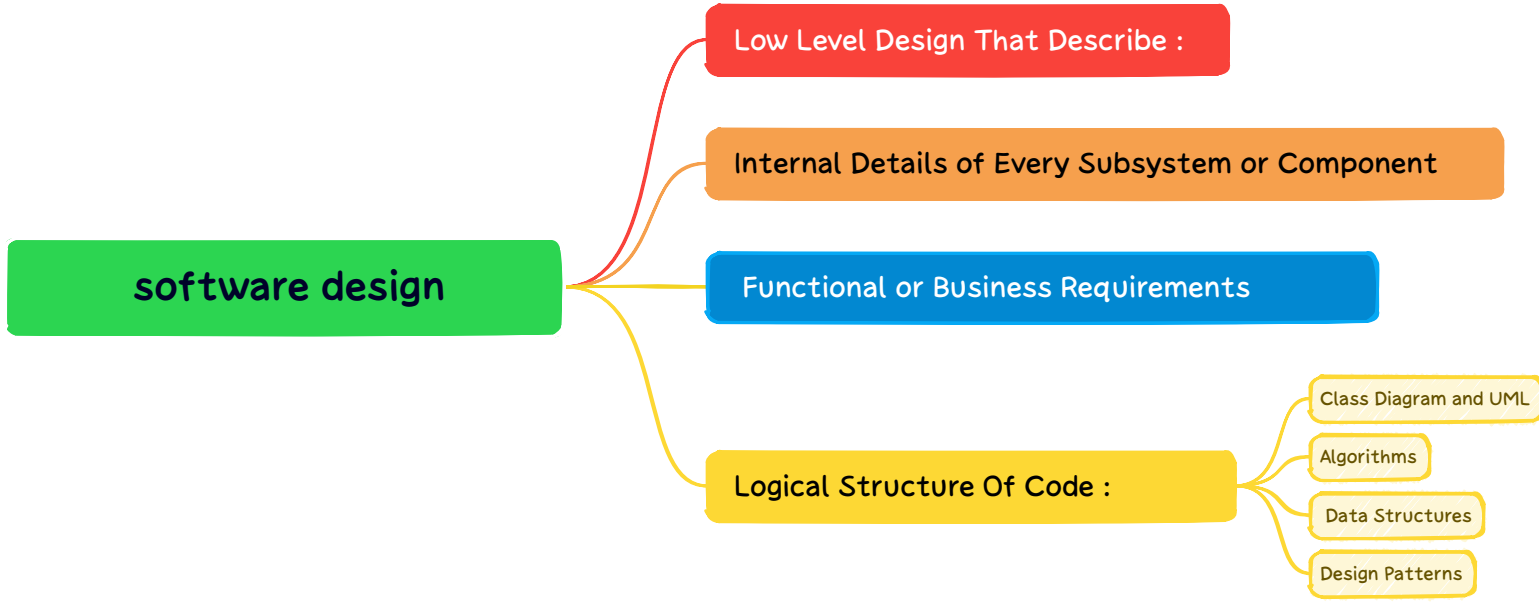


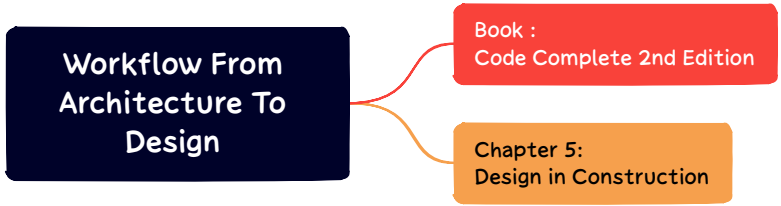
Main Software Architect Responsibilities

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) ?







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.

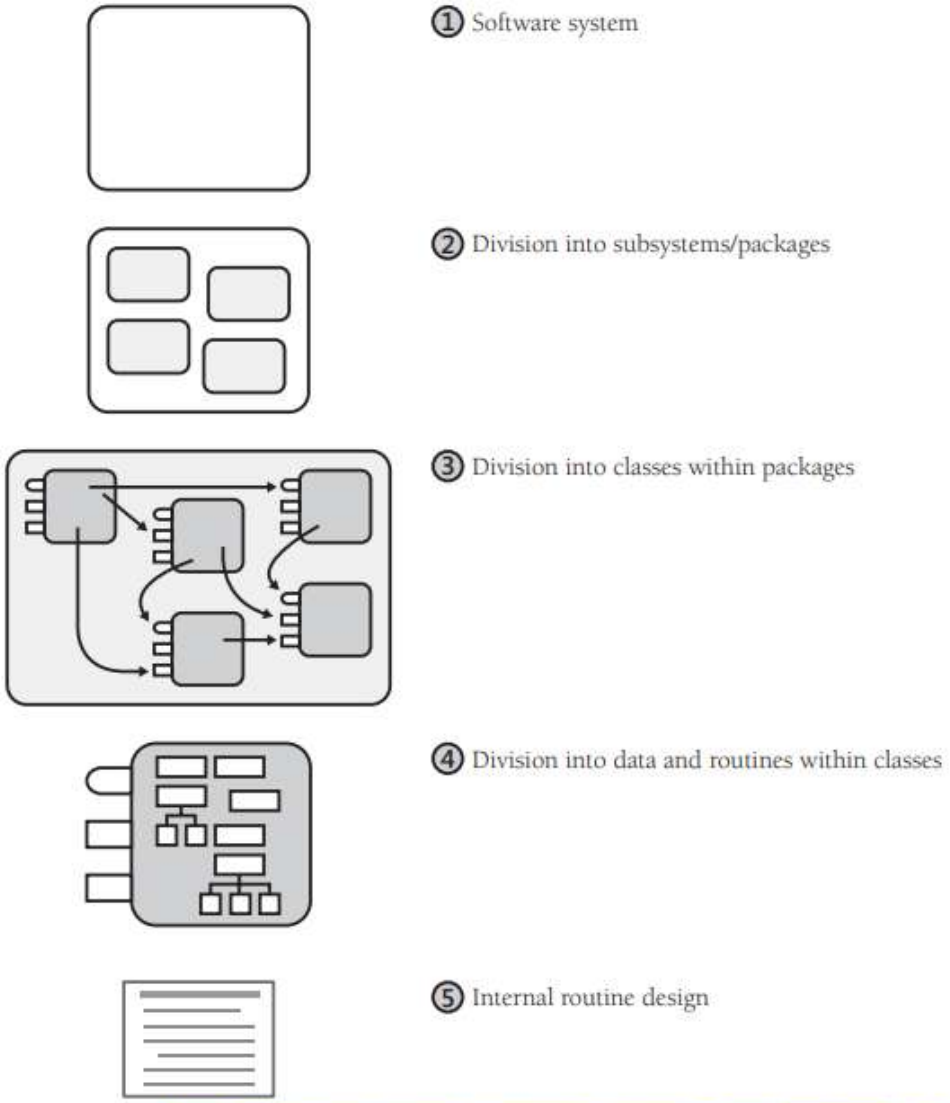


Figure 5-2 The levels of design in a program. The system (1) is first organized into subsystems (2). The subsystems are further divided into classes (3), and the classes are divided into routines and data (4). The inside of each routine is also designed (5).

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.

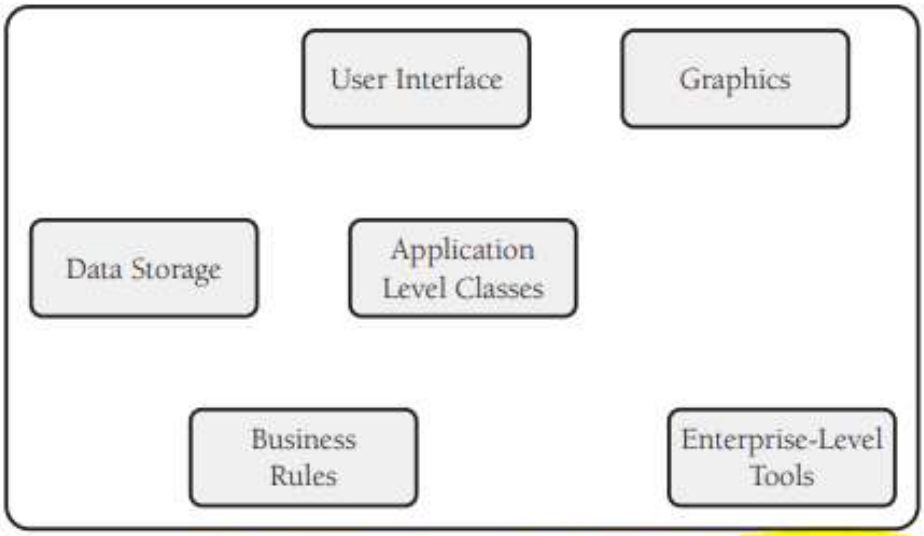


Figure 5-3 An example of a system with six subsystems.

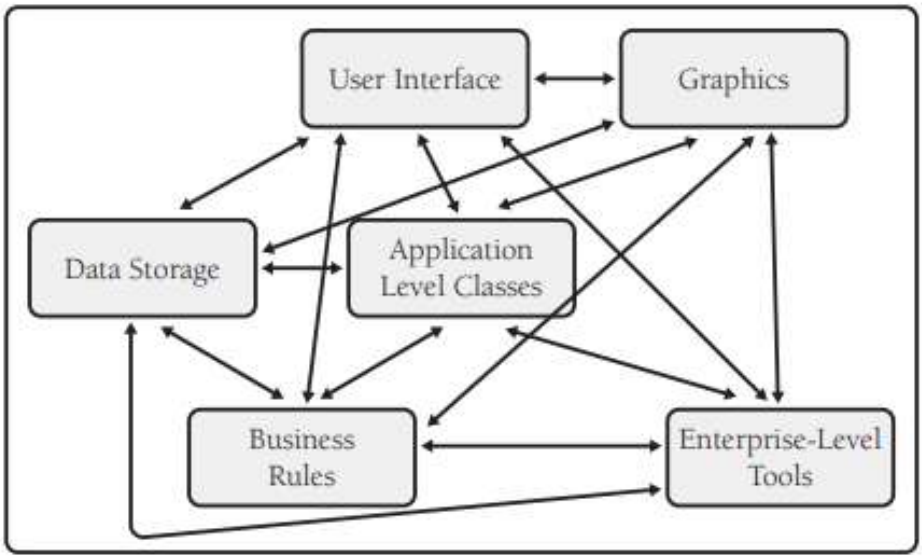


Figure 5-4 An example of what happens with no restrictions on intersubsystem communications.

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.

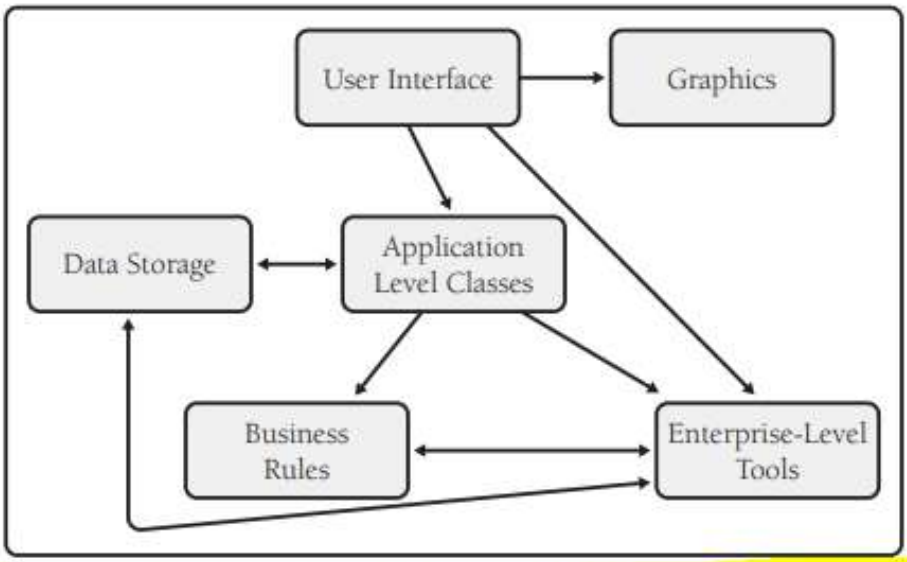


Figure 5-5 With a few communication rules, you can simplify subsystem interactions significantly.

To keep the connections easy to understand and maintain, err on the side of simple intersubsystem relations. The simplest relationship is to have one subsystem call routines in another. A more involved relationship is to have one subsystem contain classes from another. The most involved relationship is to have classes in one subsystem inherit from classes in another.

A good general rule is that a system-level diagram like Figure 5-5 should be an acyclic graph. In other words, a program shouldn't contain any circular relationships in which Class A uses Class B, Class B uses Class C, and Class C uses Class A.

On large programs and families of programs, design at the subsystem level makes a difference. If you believe that your program is small enough to skip subsystem-level design, at least make the decision to skip that level of design a conscious one.

Common Subsystems Some kinds of subsystems appear again and again in different systems. Here are some of the usual suspects.

Cross-Reference For more on simplifying business logic by expressing it in tables, see Chapter 18, "Table-Driven Methods."

Business rules Business rules are the laws, regulations, policies, and procedures that you encode into a computer system. If you're writing a payroll system, you might encode rules from the IRS about the number of allowable withholdings and the estimated tax rate. Additional rules for a payroll system might come from a union contract specifying overtime rates, vacation and holiday pay, and so on. If you're writing a program to quote automobile insurance rates, rules might come from government regulations on required liability coverages, actuarial rate tables, or underwriting restrictions

User interface Create a subsystem to isolate user-interface components so that the user interface can evolve without damaging the rest of the program. In most cases, a user-interface subsystem uses several subordinate subsystems or classes for the GUI interface, command line interface, menu operations, window management, help system, and so forth.

Database access You can hide the implementation details of accessing a database so that most of the program doesn't need to worry about the messy details of manipulating low-level structures and can deal with the data in terms of how it's used at the business-problem level. Subsystems that hide implementation details provide a valuable level of abstraction that reduces a program's complexity. They centralize database operations in one place and reduce the chance of errors in working with the data. They make it easy to change the database design structure without changing most of the program.

System dependencies Package operating-system dependencies into a subsystem for the same reason you package hardware dependencies. If you're developing a program for Microsoft Windows, for example, why limit yourself to the Windows environment? Isolate the Windows calls in a Windows-interface subsystem. If you later want to move your program to Mac OS or Linux, all you'll have to change is the interface subsystem. An interface subsystem can be too extensive for you to implement on your own, but such subsystems are readily available in any of several commercial code libraries.

Level 3: Division into Classes

Design at this level includes identifying all classes in the system. For example, a database-interface subsystem might be further partitioned into data access classes and persistence framework classes as well as database metadata. Figure 5-2, Level 3, shows how one of Level 2's subsystems might be divided into classes, and it implies that the other three subsystems shown at Level 2 are also decomposed into classes.

Details of the ways in which each class interacts with the rest of the system are also specified as the classes are specified. In particular, the class's interface is defined. Overall, the major design activity at this level is making sure that all the subsystems have been decomposed to a level of detail fine enough that you can implement their parts as individual classes.

The division of subsystems into classes is typically needed on any project that takes longer than a few days. If the project is large, the division is clearly distinct from the program partitioning of Level 2. If the project is very small, you might move directly from the whole-system view of Level 1 to the classes view of Level 3.

Level 4: Division into Routines

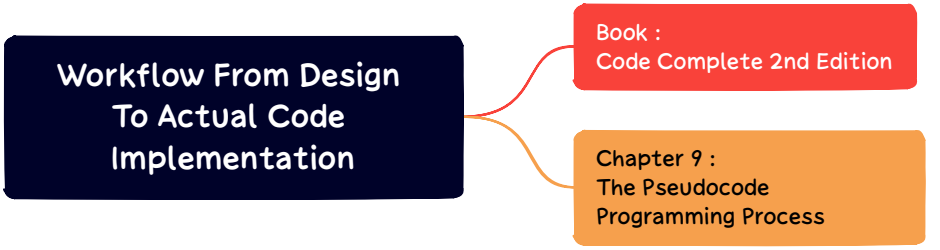
Design at this level includes dividing each class into routines. The class interface defined at Level 3 will define some of the routines. Design at Level 4 will detail the class's private routines. When you examine the details of the routines inside a class, you can see that many routines are simple boxes but a few are composed of hierarchically organized routines, which require still more design.

The act of fully defining the class's routines often results in a better understanding of the class's interface, and that causes corresponding changes to the interface—that is, changes back at Level 3.

This level of decomposition and design is often left up to the individual programmer, and it's needed on any project that takes more than a few hours. It doesn't need to be done formally, but it at least needs to be done mentally.

Level 5: Internal Routine Design

Design at the routine level consists of laying out the detailed functionality of the individual routines. Internal routine design is typically left to the individual programmer working on an individual routine. The design consists of activities such as writing pseudocode, looking up algorithms in reference books, deciding how to organize the paragraphs of code in a routine, and writing programming-language code. This level of design is always done, though sometimes it's done unconsciously and poorly rather than consciously and well. In Figure 5-2, design at this level is marked with a 5.



Design the Routine

Cross-Reference For details on other aspects of design, see Chapters 5 through 8.

Once you've identified a class's routines, the first step in constructing any of the class's more complicated routines is to design it. Suppose that you want to write a routine to

output an error message depending on an error code, and suppose that you call the routine *ReportErrorMessage()*. Here's an informal spec for *ReportErrorMessage()*:

> ReportErrorMessage() *takes an error code as an input argument and outputs an error message corresponding to the code. It's responsible for handling invalid codes. If the program is operating interactively,* ReportErrorMessage() *displays the message to the user. If it's operating in command-line mode,* ReportErrorMessage() *logs the message to a message file. After outputting the message,* ReportErrorMessage() *returns a status value, indicating whether it succeeded or failed.*

The rest of the chapter uses this routine as a running example. The rest of this section describes how to design the routine.

Steps in Building a Routine

Many of a class's routines will be simple and straightforward to implement: accessor routines, pass-throughs to other objects' routines, and the like. Implementation of other routines will be more complicated, and creation of those routines benefits from a systematic approach. The major activities involved in creating a routine—designing the routine, checking the design, coding the routine, and checking the code—are typically performed in the order shown in Figure 9-2.

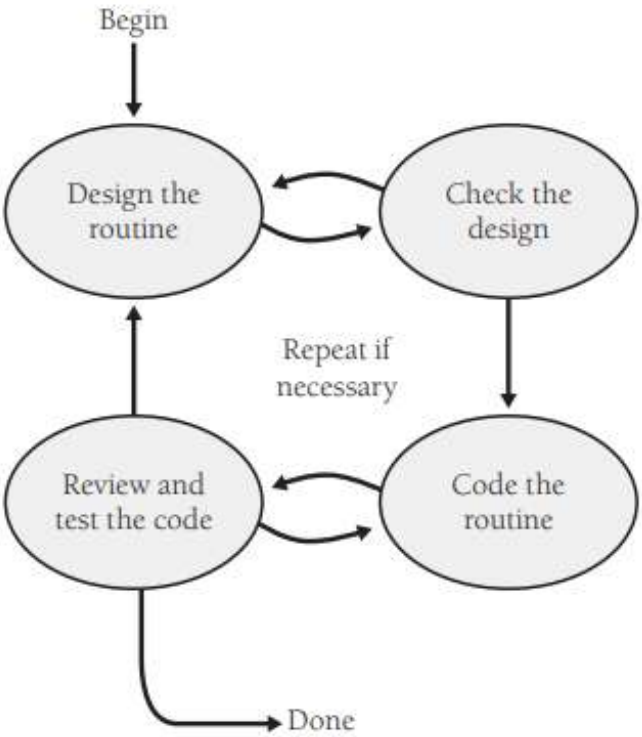


Figure 9-2 These are the major activities that go into constructing a routine. They're usually performed in the order shown.

Experts have developed numerous approaches to creating routines, and my favorite approach is the Pseudocode Programming Process, described in the next section.

9.2 Pseudocode for Pros

The term "pseudocode" refers to an informal, English-like notation for describing how an algorithm, a routine, a class, or a program will work. The Pseudocode Programming Process defines a specific approach to using pseudocode to streamline the creation of code within routines.

Because pseudocode resembles English, it's natural to assume that any English-like description that collects your thoughts will have roughly the same effect as any other. In practice, you'll find that some styles of pseudocode are more useful than others. Here are guidelines for using pseudocode effectively:

- Use English-like statements that precisely describe specific operations.
- Avoid syntactic elements from the target programming language. Pseudocode allows you to design at a slightly higher level than the code itself. When you use programming-language constructs, you sink to a lower level, eliminating the main benefit of design at a higher level, and you saddle yourself with unnecessary syntactic restrictions.
- Write pseudocode at the level of intent. Describe the meaning of the approach rather than how the approach will be implemented in the target language.
- Write pseudocode at a low enough level that generating code from it will be nearly automatic. If the pseudocode is at too high a level, it can gloss over problematic details in the code. Refine the pseudocode in more and more detail until it seems as if it would be easier to simply write the code.

Cross-Reference For details on commenting at the level of intent, see "Kinds of Comments" in Section 32.4.

Once the pseudocode is written, you build the code around it and the pseudocode turns into programming-language comments. This eliminates most commenting effort. If the pseudocode follows the guidelines, the comments will be complete and meaningful.

Here's an example of a design in pseudocode that violates virtually all the principles just described:

Example of Bad Pseudocode

```
increment resource number by 1
allocate a dlg struct using malloc
if malloc() returns NULL then return 1
invoke OSrsrc_init to initialize a resource for the operating system
*hRsrcPtr = resource number
return 0
```

Example of Good Pseudocode

```
Keep track of current number of resources in use
If another resource is available
   Allocate a dialog box structure
   If a dialog box structure could be allocated
      Note that one more resource is in use
      Initialize the resource
      Store the resource number at the location provided by the caller
   Endif
Endif
Return true if a new resource was created; else return false
```

Design the Routine

Cross-Reference For details on other aspects of design, see Chapters 5 through 8.

Once you've identified a class's routines, the first step in constructing any of the class's more complicated routines is to design it. Suppose that you want to write a routine to

output an error message depending on an error code, and suppose that you call the routine *ReportErrorMessage()*. Here's an informal spec for *ReportErrorMessage()*:

> ReportErrorMessage() *takes an error code as an input argument and outputs an error message corresponding to the code. It's responsible for handling invalid codes. If the program is operating interactively,* ReportErrorMessage() *displays the message to the user. If it's operating in command-line mode,* ReportErrorMessage() *logs the message to a message file. After outputting the message,* ReportErrorMessage() *returns a status value, indicating whether it succeeded or failed.*

The rest of the chapter uses this routine as a running example. The rest of this section describes how to design the routine.

Check the prerequisites Before doing any work on the routine itself, check to see that the job of the routine is well defined and fits cleanly into the overall design. Check to be sure that the routine is actually called for, at the very least indirectly, by the project's requirements.

Define the problem the routine will solve State the problem the routine will solve in enough detail to allow creation of the routine. If the high-level design is sufficiently detailed, the job might already be done. The high-level design should at least indicate the following:

- The information the routine will hide
- Inputs to the routine
- Outputs from the routine
- Preconditions that are guaranteed to be true before the routine is called (input values within certain ranges, streams initialized, files opened or closed, buffers filled or flushed, etc.)
- Postconditions that the routine guarantees will be true before it passes control back to the caller (output values within specified ranges, streams initialized, files opened or closed, buffers filled or flushed, etc.)

Here's how these concerns are addressed in the *ReportErrorMessage()* example:

- The routine hides two facts: the error message text and the current processing method (interactive or command line).
- There are no preconditions guaranteed to the routine.
- The input to the routine is an error code.
- Two kinds of output are called for: the first is the error message, and the second is the status that *ReportErrorMessage()* returns to the calling routine.
- The routine guarantees that the status value will have a value of either *Success* or *Failure*.

Name the routine Naming the routine might seem trivial, but good routine names are one sign of a superior program and they're not easy to come up with. In general, a routine should have a clear, unambiguous name. If you have trouble creating a good name, that usually indicates that the purpose of the routine isn't clear. A vague, wishy-washy name is like a politician on the campaign trail. It sounds as if it's saying something, but when you take a hard look, you can't figure out what it means. If you can make the name clearer, do so. If the wishy-washy name results from a wishy-washy design, pay attention to the warning sign. Back up and improve the design.

In the example, *ReportErrorMessage()* is unambiguous. It is a good name.

Decide how to test the routine As you're writing the routine, think about how you can test it. This is useful for you when you do unit testing and for the tester who tests your routine independently.

In the example, the input is simple, so you might plan to test *ReportErrorMessage()* with all valid error codes and a variety of invalid codes.

Research functionality available in the standard libraries The single biggest way to improve both the quality of your code and your productivity is to reuse good code. If you find yourself grappling to design a routine that seems overly complicated, ask whether some or all of the routine's functionality might already be available in the library code of the language, platform, or tools you're using. Ask whether the code might be available in library code maintained by your company. Many algorithms have already been invented, tested, discussed in the trade literature, reviewed, and improved. Rather than spending your time inventing something when someone has already written a Ph.D. dissertation on it, take a few minutes to look through the code that's already been written and make sure you're not doing more work than necessary.

Think about error handling Think about all the things that could possibly go wrong in the routine. Think about bad input values, invalid values returned from other routines, and so on.

Routines can handle errors numerous ways, and you should choose consciously how to handle errors. If the program's architecture defines the program's error-handling strategy, you can simply plan to follow that strategy. In other cases, you have to decide what approach will work best for the specific routine.

Think about efficiency Depending on your situation, you can address efficiency in one of two ways. In the first situation, in the vast majority of systems, efficiency isn't critical. In such a case, see that the routine's interface is well abstracted and its code is readable so that you can improve it later if you need to. If you have good encapsulation, you can replace a slow, resource-hogging, high-level language implementation with a better algorithm or a fast, lean, low-level language implementation, and you won't affect any other routines.

Cross-Reference For details on efficiency, see Chapter 25, "Code-Tuning Strategies," and Chapter 26, "Code-Tuning Techniques."

In the second situation—in the minority of systems—performance is critical. The performance issue might be related to scarce database connections, limited memory, few available handles, ambitious timing constraints, or some other scarce resource. The architecture should indicate how many resources each routine (or class) is allowed to use and how fast it should perform its operations.

Design your routine so that it will meet its resource and speed goals. If either resources or speed seems more critical, design so that you trade resources for speed or vice versa. It's acceptable during initial construction of the routine to tune it enough to meet its resource and speed budgets.

Aside from taking the approaches suggested for these two general situations, it's usually a waste of effort to work on efficiency at the level of individual routines. The big optimizations come from refining the high-level design, not the individual routines. You generally use micro-optimizations only when the high-level design turns out not to support the system's performance goals, and you won't know that until the whole program is done. Don't waste time scraping for incremental improvements until you know they're needed.

Research the algorithms and data types If functionality isn't available in the available libraries, it might still be described in an algorithms book. Before you launch into writing complicated code from scratch, check an algorithms book to see what's already available. If you use a predefined algorithm, be sure to adapt it correctly to your programming language.

Write the pseudocode You might not have much in writing after you finish the preceding steps. The main purpose of the steps is to establish a mental orientation that's useful when you actually write the routine.

With the preliminary steps completed, you can begin to write the routine as high-level pseudocode. Go ahead and use your programming editor or your integrated environment to write the pseudocode—the pseudocode will be used shortly as the basis for programming-language code.

Start with the general and work toward something more specific. The most general part of a routine is a header comment describing what the routine is supposed to do, so first write a concise statement of the purpose of the routine. Writing the statement will help you clarify your understanding of the routine. Trouble in writing the general comment is a warning that you need to understand the routine's role in the program better. In general, if it's hard to summarize the routine's role, you should probably assume that something is wrong. Here's an example of a concise header comment describing a routine:

Example of a Header Comment for a Routine

```
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.
```

After you've written the general comment, fill in high-level pseudocode for the routine. Here's the pseudocode for this example:

Example of Pseudocode for a Routine

```
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.

set the default status to "fail"
look up the message based on the error code

if the error code is valid
   if doing interactive processing, display the error message
   interactively and declare success

   if doing command line processing, log the error message to the
   command line and declare success

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

return status information
```

Again, note that the pseudocode is written at a fairly high level. It certainly isn't written in a programming language. Instead, it expresses in precise English what the routine needs to do.

Again, note that the pseudocode is written at a fairly high level. It certainly isn't written in a programming language. Instead, it expresses in precise English what the routine needs to do.

Cross-Reference For details on effective use of variables, see Chapters 10 through 13.

Think about the data You can design the routine's data at several different points in the process. In this example, the data is simple and data manipulation isn't a prominent part of the routine. If data manipulation is a prominent part of the routine, it's worthwhile to think about the major pieces of data before you think about the routine's logic. Definitions of key data types are useful to have when you design the logic of a routine.

Cross-Reference For details on review techniques, see Chapter 21, "Collaborative Construction."

Check the pseudocode Once you've written the pseudocode and designed the data, take a minute to review the pseudocode you've written. Back away from it, and think about how you would explain it to someone else.

Ask someone else to look at it or listen to you explain it. You might think that it's silly to have someone look at 11 lines of pseudocode, but you'll be surprised. Pseudocode can make your assumptions and high-level mistakes more obvious than programming-language code does. People are also more willing to review a few lines of pseudocode than they are to review 35 lines of C++ or Java.

Make sure you have an easy and comfortable understanding of what the routine does and how it does it. If you don't understand it conceptually, at the pseudocode level, what chance do you have of understanding it at the programming-language level? And if you don't understand it, who else will?

Try a few ideas in pseudocode, and keep the best (iterate) Try as many ideas as you can in pseudocode before you start coding. Once you start coding, you get emotionally involved with your code and it becomes harder to throw away a bad design and start over.

The general idea is to iterate the routine in pseudocode until the pseudocode statements become simple enough that you can fill in code below each statement and leave the original pseudocode as documentation. Some of the pseudocode from your first attempt might be high-level enough that you need to decompose it further. Be sure you do decompose it further. If you're not sure how to code something, keep working with the pseudocode until you are sure. Keep refining and decomposing the pseudocode until it seems like a waste of time to write it instead of the actual code.

Code the Routine

Once you've designed the routine, construct it. You can perform construction steps in a nearly standard order, but feel free to vary them as you need to. Figure 9-3 shows the steps in constructing a routine.

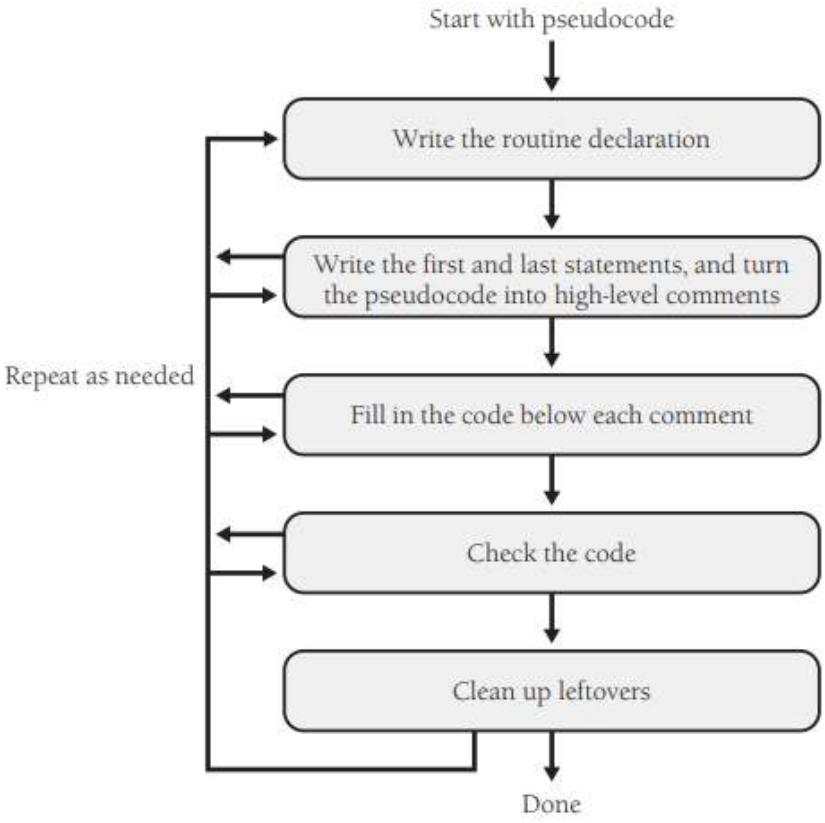


Figure 9-3 You'll perform all of these steps as you design a routine but not necessarily in any particular order.

Write the routine declaration Write the routine interface statement—the function declaration in C++, method declaration in Java, function or sub procedure declaration in Microsoft Visual Basic, or whatever your language calls for. Turn the original header comment into a programming-language comment. Leave it in position above the pseudocode you've already written. Here are the example routine's interface statement and header in C++:

C++ Example of a Routine Interface and Header Added to Pseudocode

Here's the header comment that's been turned into a C++-style comment.

Here's the interface statement.

```
/* 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"
look up the message based on the error code

if the error code is valid
   if doing interactive processing, display the error message
   interactively and declare success

   if doing command line processing, log the error message to the
   command line and declare success

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

return status information
```

This is a good time to make notes about any interface assumptions. In this case, the interface variable *errorToReport* is straightforward and typed for its specific purpose, so it doesn't need to be documented.

Turn the pseudocode into high-level comments Keep the ball rolling by writing the first and last statements: *{* and *}* in C++. Then turn the pseudocode into comments. Here's how it would look in the example:

C++ Example of Writing the First and Last Statements Around Pseudocode

```
/* 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
   ) {
```


The pseudocode statements from here down have been turned into C++ comments.

```
   // set the default status to "fail"
   // look up the message based on the error code
   // if the error code is valid
      // if doing interactive processing, display the error message
      // interactively and declare success

      // if doing command line processing, log the error message to the
      // command line and declare success

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

   // return status information
}
```