Designing for Change

Jeff Offutt
George Mason University

January 2018

The first level of engineering software for maintainability is coding. As I said before, tidy programs are easier to understand and modify. The second level is during design and integration. Well organized, simple, and clean interfaces among components make future changes easier.

Integrating Software Components

In the 20th century, many programs were single stand-alone systems. 21st century software, however, is often collections of integrated software systems that live in an eco-system, not in isolation. They interact with other applications, they use shared libraries, they communicate with related applications on the internet and the cloud, and they share data with multiple computing devices. This increase in couplings among software systems is very powerful but makes careful design more important. In effect, distributed computing has become the norm, not the exception.

Many factors make this kind of integration-heavy software difficult. Networks are notoriously unreliable and much slower than the devices that use them. Programs are diverse in terms of language, operating system, data formats, and many other characteristics. And of course, change is inevitable and continuous. The eco-system changes every time hardware or software is updated and when new applications are brought on board.

Back "in the day," software components were coupled through function calls and shared, nonlocal, variables. Now software components are coupled through networks, messages, the cloud, databases, and other convenient mechanisms. The notion of coupling has been extended:

- Tight Coupling: Dependencies among the methods are encoded in their logic. Changes in method **A** may require changing the logic in a coupled module **B**. This was common in the 1980s and often indicated a functional design of software.
- Loose Coupling: Dependencies among modules are encoded in the structure and data flows. Changes in module **A** may require changing data uses in module **B**. This was the original goal of data abstraction and object-oriented concepts [1], which are now embedded firmly in early programming courses such as data structures and in object-oriented programming languages.
- Extremely Loose Coupling (ELC): Dependencies are encoded only in the data contents. Changes in **A** only affect the **contents** of **B**'s data. This is the primary motivating goal for most advances in distributed software and web applications over the last 20 years.

eXtensible Markup Language (XML)

ELC leads directly to XML as a simple but powerful way to support it. Passing data from one software component to another has always been difficult. The two components must agree on format, types, and organization. In method calls, this agreement is syntactically hard-coded into method signatures. But we don't have the same level of type checking in distributed software (such as web applications), and we have additional requirements:

The software must support extremely loose coupling Software components may be integrated dynamically (during run time) The software must adapt to frequent changes

I examine this through a common form of coupling: One program, **P2**, needs to use data created by another program, **P1**. In the 1970s, before the emphasis on data abstraction, **P1** would write the data into a file. To save space, files were saved in very compact forms (binary, not text), and with rigidly structured records that were often not documented. If **P2** is written **after** the source of P1 is no longer available, the structure of the file would have to be deduced by a slow trial-and-error process of reading bytes into memory, and printing them in different formats to see if they legible.

By the 1980s, the concept of data abstraction led to the file being controlled by a "wrapper module" that could read and write the file. The wrapper was shared by both P1 and P2. This was still slow, and since the wrapper module was shared among multiple programs, it was very difficult to change it or the structure of the file. Adding a single field could disrupt dozens of programs.

The modern solution is to use file formats that are free-form, textual, and self-documenting. That is, XML. XML files take a lot more space:

```
<book>
  <title>Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability</
     title>
  <author>Steve Krug</author>
  <year>2014</year>
</book>
```

but make programs easier to change.

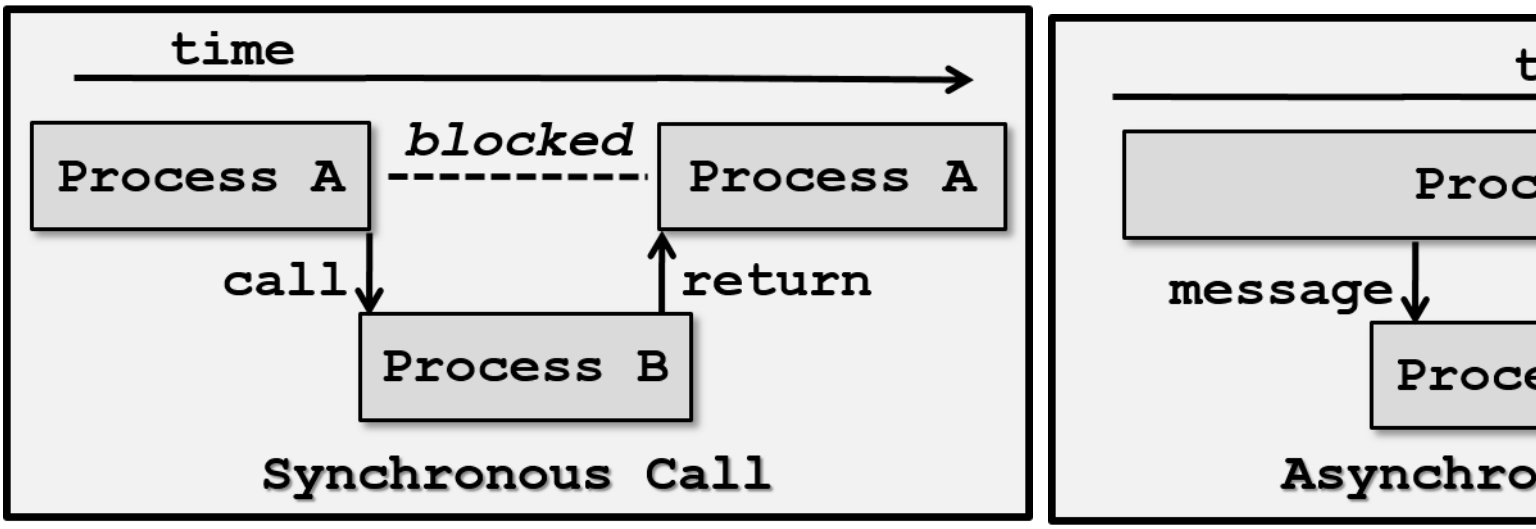
Sharing Data and Message Passing

A major factor in the maintainability of software is how data is shared and information is passed among software components. Four major styles are:

1. File storage: This is a traditional method where one program writes to a file that another later reads. Both programs need to agree on several things, including the file name and location, the format of the file, when the file can be read and written to, and who will delete the file.
2. Shared database: A more robust method is to replace a file with a database. This allows most decisions to be encapsulated in the table design, and offloads much of the effort onto a database package.
3. Remote Method Invocations (RMI): This is similar to an old-fashioned method call, except the method operates in a different memory space, runs inside a different owning process, and may be on a different computer. Communication is real-time and synchronous, and needs to be managed by robust and sophisticated software. RMIs typically expect the caller to pause execution until the callee completes, and returns some result.
4. Message passing: This is more asynchronous than RMIs. One module sends a message to a common message channel, and other modules read the messages. The sender does not necessarily wait for the receiver to respond, and the receiver does not necessarily read the message

immediately. Both modules must agree on the channel and message format, usually without built-in type checking. XML is commonly used to encode messages because of its flexibility and self-documenting nature.

The distinction between synchronous and asynchronous messages is very important. A telephone call is synchronous, because both parties need to be on the call at the same time. Synchronous communication has greater bandwidth, but introduces two restrictions. First, both parties have to be available at the same time, and second, communication must be in real-time. Voice mail is asynchronous. We leave messages for later retrieval, so the real-time aspects are less important. In web applications, the traditional request-response cycle imposes a synchronous model, but Ajax introduces asynchronous calls.



(a) A synchronous call

(b) An asyn

Asynchronous messaging architectures are very powerful, but we have to change how we design and develop software. We have decades of deep knowledge about using files, shared databases, and remote method invocations, but we teach college students relatively little about asynchronous software engineering.

Advantages and Disadvantages of Message Passing

Data encapsulation is stronger than with databases and file storage. RMIs have reliability problems because any glitch in the network will disrupt the communication, whereas messages simply wait until the network recovers. Message passing reduces dependencies, making it less likely that changes will cause problems elsewere in the system. That is, fewer ripple effects. This improves maintainability, as well as reliability, security, and scalability.

On the other hand, the lack of deep knowledge for how to write asynchronous software makes them less likely to be reliable, and harder to understand (a negative for maintainability).

The programming model is different and complex. Logic is distributed across several software components, which does not match how we teach topics like algorithms. Many universities do not teach event-driven software at all.

The sequencing of software tasks is harder. Message systems do not guarantee when the message will arrive, so messages sent in one sequence may arrive in a different sequence. In fact, many applications that could use asynchronous events intentionally do not because of the engineering challenges.

Using Design Patterns to Integrate

Enterprise systems contain hundreds, sometimes thousands, of separate applications. They are a mix of custom-built components, third party vendors, and legacy systems. They are often designed with multiple tiers that run on different computers and different operating systems. Many companies depend on large enterprise systems that encapsulate the operation of many aspects of the business. Patriot Web is an example at my university. That and Blackboard, which is used widely to support teaching, are both unreliable, hard to modify, and have extreme usability problems. Although universities suffer more than many companies, problems with enterprise systems are quite common. Many actually grew from multiple smaller software components, just like small towns grow together, slowly integrating to form cities. This type of organic growth invariably creates maintenance debt and confusion—just think of the last time you saw a street change its name without warning.

Integrating diverse applications into a coherent enterprise application will be an important task for years to come. It's not easy, but understanding important goals like maintainability and usability help. Lots of frameworks and integration platforms are available. One of the most important thing to understand is their set of basic assumptions. Some assume that the data never changes, but new functions will be continually added. Thus, APIs should be strong and clear, although the central database may be very hard to change. Others assume that the functions will remain constant, but they will be adapted to new hardware platforms and to new users. Thus, new features will be hard to add, but the UI should be easy to change.

These are about tradeoffs. When systems are integrated, we usually can't support maintainability in all aspects, so a crucial early decision is which types of changes should be planned for. If an organization's software architect gets that wrong, the entire organization will suffer for years. One of the hardest part about making these decisions is that the planning team must be able to look 5 or 10 years into the future. This is a rare ability.

To summarize, reducing coupling is a key goal to ensuring maintainability at any level. Software engineers have known about the importance of coupling since the 1970s, although the specifics change with each generation of language, hardware, and software engineering technologies. The primary goal of coupling is to reduce the assumptions that two software components have to make when exchanging data. Loose coupling means fewer assumptions. Local method calls have very tight coupling, as do remote method invocations. Worse, RMIs come with the complexity of distributed processing. Message passing, however, has extremely loose coupling and is thus a strong way to increase maintainability.

References

[1] Dave Parnas, On the Criteria To Be Used in Decomposing Systems into Modules, Communications of the ACM, 15(12):1053-1058, December 1972