Abstraction

Abstraction is the fundamental technique used by software engineers to be able to manage the complexity of their systems. Abstraction enables engineers to focus on the *key* information for a given task while eliding unnecessary detail. (e.g., a UX designer might focus on UI flows, but would not reason about backend cryptographic protocols). The 'right' abstraction will vary from task to task. The most high-level and common kinds of abstractions relate to control and data abstraction.

Data abstraction

Data abstraction is the process of explicitly separating the abstract properties of a data type and its concrete implementation. The primary benefit of data abstraction is that it enables client code to be oblivious of the underlying implementation of a data type allowing it to be upgraded and improved without impacting client code.

For example, a `Vector` in Java is implemented using an array, but the JDK authors could choose to change this implementation (say to a linked list) without impacting client programs. A Java `HashMap` is stored as an array of linked lists. In both cases, client code can be oblivious of the underlying implementation as long as they understand the behaviour of the methods provided by `Vector` and `HashMap`.

Here is a concrete example of data abstraction from CPSC 210:

```
// Creates a new team.
// Requires: A non-empty and unused name.
// Modifies: Team database.
// Effects:  Returns whether a team was created.
 public createTeam(name: string):boolean {...}
```

By explicitly describing the data the method requires, modifies, and its side effects, client code can be completely oblivious of the underlying implementation used by `createTeam(..)`. What we are really trying to do in this example is define the method's contract, that is its preconditions (expects), postconditions (provides), and invariants (must always be true). Correctly and adequately documenting these abstractions is important because state-based errors are a common source of difficult-to-diagnose faults in modern systems that type systems provide little defence against.

Control abstraction

One of the major benefits of modern programming languages is that they provide an abstraction layer between the developer expressing themselves and how a machine will interpret and execute their instructions. Abstracting away how the code will execute is called *control abstraction* and was one of the early advances in software engineering productivity (as described by Fred Books in No Silver Bullet). An easy way to think about control abstraction is that it allows an engineer to consider *what* they are doing without becoming overwhelmed with *how* they are going to do it.

It is easy to take control abstraction for granted, but the difference between the following Java and Assembler snippets demonstrates the benefit of this abstraction layer (example from Beyond Java):

Java:

```
a = b + 1;
b = c + 2;
c = a + 3;
```

Assembly:

```
mov     r11d,2h
mov     r10,7d56d18b0h
add     r11d,dword ptr [r10+78h]
mov     r9d,dword ptr [r10+74h]
mov     dword ptr [r10+74h],r11d
mov     r11d,r9d
inc     r11d
mov     dword ptr [r10+70h],r11d
add     r9d,4h
mov     dword ptr [r10+78h],r9d
```

Dijkstra coined the term 'structured programming' to describe language constructs that are designed to allow developers to express their programs using logical blocks of code that execute in sequence with control statements for choosing which blocks execute and subroutines so programs can be more meaningfully decomposed.

Language constructs are continually evolving to enable developers to better focus on their task intent instead of worrying about underlying execution context. JavaScript promises are one such example as it allows sequences of asynchronous operations to be developed without resorting to nested callbacks and written in a way that mimics synchronous method calls.

```
// Manages team creation flow
// Defers to other methods for operations
makeTeam(teamName: string, memberName: string) {
  var that = this;
  // create team
  that.createTeam(teamName).then(function (teamId) { // createTeam
  // add member
  return that.addMember(teamId, memberName); // addMember is async
}).then(function (success) {
  // ...
}).catch(function (err) {
  // ...
});
}
```

Decomposition

Decomposition is the process of taking a complex high-level entity and splitting it into more manageable smaller pieces. One of the main goals during decomposition is to make simple tasks simple while ensuring that exceptional tasks are still possible. There are many kinds of decomposition strategies (e.g., based around program units, algorithms, subsystems, etc.) but the most general (and common) is top-down decomposition.

In top-down composition a description is broken up into pieces starting at the top and working to greater level of detail. Initially this will mean that many important details will be represented by 'black boxes' (which can represent systems, modules, or any other relevant level of abstraction) that can be filled in in the future through either further decomposition of the box or by providing final details of what the piece does. Working top-down is a great way to have global awareness about the full system but can be challenging once terminal boxes develop constraints that necessitate revisiting prior decisions.

An alternative (also common) approach is working bottom-up. In this way decisions can be made about leaf nodes that can be composed into the final overall system. This approach is great when you have concrete details about the team implementing the system (e.g., because you can have them make the most relevant decisions for their team right away). A downside of bottom-up approaches is they often lack the global overview which can lead to inconsistencies and extraneous focusing on details early in the design stage.

Information hiding

Software engineers often talk about abstraction in terms of *information hiding*. Information hiding was first proposed by David Parnas in 1972 as a means for separating the parts of the program that are most likely to change from those parts that are more static. Information hiding is a high-level motivation for APIs: by describing the expected behaviour of the API you can "hide" the implementation behind it.

Information hiding is a specific, common, and important form of abstraction that intentionally seeks to identify 'that which varies' from 'that which stays the same'. This is important, because all abstractions in code come with a cost: trying to understand a system with unnecessary abstractions can add complexity and difficulty, while balancing this complexity against trying to evolve a system lacking necessary abstractions is a challenging task.

Encapsulation is related to information hiding and is practiced most concretely in object-oriented programming languages like Java, C#/C++, or TypeScript. Encapsulation is concerned with delineating the contractual interface with its interface. The most common language feature for supporting encapsulation is the interface whereby the interface describes the public contract and the concrete class describes the implementation (along with its supporting private methods and fields).

Constant change

As Jeff Dean noted in his WSDM 2009 Keynote it is important to recognize the parameters that lead to the abstractions you choose will change over time, and often by many orders of magnitude. The right design for one system will probably be different at 10X load or 100X load. While it can be tempting to try to design for the 'end game' there are high real costs associated with premature optimization. In his talk, Jeff advocates designing for 10x load with the expectation that a rewrite would be required at 100x load. While one could view this process as wasteful, an alternative viewpoint is that in the time between 10x and 100x you will learn things about your system you would not have known in advance (and would thus not solved your 100x problem from the start anyways).

Thinking about change is also related to encapsulation in that thinking concretely about what parts of the system are likely to change in the short and medium terms are more likely to lead to useful and valuable abstraction layers than taking an 'anything can change' view to design.

References

- Original information hiding paper.
