**DESIGNING THE PRODUCT（A）**

Most hard-core project courses put emphasis on the design aspect. Here are some things to keep in mind when designing your system.

1 ABSTRACTION IS YOUR FRIEND

A software design is a complex thing, especially, if you do not use abstraction to good effect. The design should be done at various levels of abstraction. At higher levels, you visualize the system as a small number of big components while abstracting away the details of what is inside each component. Once you have a good definition of what each of those components represent and how they interact with each other, you can move to the next lower level of abstraction. That is, you take each one of those components and design it using a small number of smaller components. This can go on until you reach a lower level of abstraction that is concrete enough to transform to an implementation.

2 SEPARATE CONCERNS

We should try to keep different 'concerns' of the system as separated from each other as possible. For example, parsing (and things related to the 'parsing' concern) should be done by the *parser* component, and everything that has to do with sorting should be done by the *sorter* component.

3 DON'T TALK TO STRANGERS

Otherwise known as the *Law of Demeter*, the 'don't talk to strangers' principle advocates keeping unrelated things independent of each other. For example, if the *parser* component can function without any knowledge of the *sorter* component, then the *sorter* is a stranger to the *parser*. That means the *parser* should not have any reference to the *sorter* (e.g. can you compile the *parser* without compiling the *sorter*?)

A classic example where this principle applies is when choosing between a *central controller* model and *chain of controllers* model. The former lets you keep strangers as strangers, while the latter forces strangers to talk to each other. Notice how one design below lets *B* and *C* remain strangers while the other forces them to know each other.



4 ENCAPSULATE

A component should reveal as little as possible about itself. This is also known as *information hiding*. For example, other components that interact with your component should not know in what format your component stores data and they should not be allowed to manipulate those data directly.

**DESIGNING THE PRODUCT（B）**

Most hard-core project courses put emphasis on the design aspect. Here are some things to keep in mind when designing your system.

5 PRESERVE CONCEPTUAL INTEGRITY

A student team is a team of peers that usually does not have a designated architect to dictate a design for others to follow. Everyone may want to have their say in the design. However, after discussing all alternative designs proposed, you should still choose one of them to follow, rather than devise a design that combines everyone's ideas.

Combining ideas into one design has its merits, but do not do it just for the sake of achieving a compromise between competing ideas. If in doubt, get your supervisor's opinion as well.

6 STANDARDISE SOLUTIONS

Similar problems should be solved in a similar fashion. Sometimes, it pays to solve even slightly different yet largely similar problems in exactly the same way. It makes programs easier to understand. In other words, do not go out of your way to customize a solution to fit a problem *precisely*. It may be better to preserve the similarity of the solution instead.

7 USE PATTERNS

Patterns, as well as other types of patterns such as analysis patterns, testing patterns, etc.) embody tried-and-tested solutions to common problems. Learn patterns and use them where applicable.

8 DO NOT OVERUSE PATTERNS

Most patterns come with extra baggage. Do not use patterns for the sake of using them. For example, there is no need to apply the Singleton pattern to *every* class that will have only one instance in the system; use it when there is a danger of someone creating multiple instances of such a class.

9 VALUE SIMPLICITY

Try to make the design as simple as possible (but no simpler!). Simple yet elegant designs are much better than complex solutions: The former is much harder to achieve however, but that is what you should strive for. Given any design, try to see whether you can simplify it. Resist any change that makes it more complex.

10 INCREASE *FAN-IN*, DECREASE *FAN-OUT*

When many other components use a given component (i.e. the component has high *fan-in*), this is a good thing because it increases reuse. When a given component uses many other components (i.e. it has high *fan-out*), this is not a good thing because it increases coupling.

11 GIVE BRUTE FORCE A CHANCE

Some problems have an obvious and simple brute force solution. Do not dismiss this solution too quickly in your haste to look for a smarter solution. If you can afford it, give this brute force solution a chance; it may be all you need.

**DESIGNING THE PRODUCT（C）**

Most hard-core project courses put emphasis on the design aspect. Here are some things to keep in mind when designing your system.

12 LESS IS MORE

Trim all parts of the design that are not immediately useful to the system. It does not matter how elegant they are, how proud you are of dreaming them up, and how hard you worked at building them. The same applies to code.

13 DO NOT FORGET NON-FUNCTIONAL QUALITIES

Some non-functional qualities need to be incorporated from the design stage. One non-functional quality rarely mentioned in the specification and often forgotten in the design is the testability. Improving testability improves many other qualities of the design.

14 BEWARE OF PREMATURE OPTIMISATION

Hoare wasn't kidding when he said "Premature optimization is the root of all evil in programming". Opt for a simple design. If it is fast enough, stick with it. If it is not, find the real bottlenecks and optimize accordingly.

Caveat: This does not mean that you should start with a stupid design. Some designs are obviously inefficient and should be discarded immediately. Start with a design that is, in Einstein's words "as simple as possible, but no simpler".

15 AVOID *ANALYSIS PARALYSIS*

During analysis and design, consider all the known facts but do not fret too much about unknowns. If you are given a concrete and stable specification (e.g. writing a parser for a given language) it would be stupid to start with a design that does not take the entire specification into account. Such short-sighted designs will eventually require change, causing rework that could have been avoided. On the other hand, if you are defining a first-of-a-kind exploratory system for an unspecified user base, go for a design with a reasonable degree of flexibility; do not worry about all the nitty-gritty issues that it might or might not have to face later.

16 POLISH DOCUMENTS LATER

Documenting design often requires wrestling with UML editors and other diagramming tools. Therefore, it is very frustrating when we have to modify those documents as the design changes over time.

While designs should be documented as they are done, there is no need to start creating well-polished design documents right away. You can keep the documentation as low-maintenance rough sketches (with none of the important points missing) until the design is sufficiently stable. For example, you can take a photo of the whiteboard on which you drew the initial design, print it out, do your (minor) modifications on the hard copy, and convert it to a digitized UML diagram much later.

17 REMEMBER THE REASON

As students, you are not expected to choose the best design in the first try. Designs often evolve over time. Be sure to document such changes so that you get credit for the effort spent in the process. Make sure your design has a rationale. Often evaluators ask "why did you choose this design (over another)?" No design is perfect. You may be able to score extra credit by critically evaluating your final design, comparing it with an alternative design, and discussing ways of improving it further.