Reasons to Refactor

Adding new features. A lot of software development involves enhancing an existing system rather than creating a new system from scratch. When you find you have to add a feature to a program, and the program's code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature.

Refactoring to Patterns. Using design patterns in a project’s initial design is appropriate when the patterns provide capabilities that you know you will need. However, using patterns just for the sake of using patterns can introduce unnecessary complexity (i.e., overkill). However, as a system is enhanced over time, oftentimes the use of patterns becomes more necessary as more advanced capabilities are needed. A good approach is to introduce patterns as they are actually needed through a careful refactoring process.

Writing new code. Software is often hurriedly prototyped as a "proof of concept" or in an effort to refine end-user requirements. Once the prototype has served its purpose, we should either throw it away, or refactor it to improve its structure and design.

Combatting design decay. Software designs decay over time. Why? 1) People who don’t understand the system’s design are required to make changes (e.g., new team members), 2) Changes are made under schedule pressure; rather than doing the “right” thing, we do the “quick and dirty” thing. 3) Some developers are less skillful and/or disciplined than others, and make poor design choices. Refactoring is an effective way of combating design decay.

Clarify the code

- Understanding a new code base
- Finding & fixing bugs
- Code that is just confusing

When your design stinks (i.e., when you detect one of the design smells listed in chapter 3 of the Refactoring book)
Example: Code Duplication

> “The first time you do something, you just do it. The second time you do something similar, you wince at the duplication, but you do the duplication thing anyway. The third time you do something similar, you refactor.”

Evolutionary design. The traditional view of software development says that a program’s design should be nailed down in good detail before much code has been written. This is based on the assumption that later design changes require major rework, and, therefore, are expensive to make. This approach puts a lot of pressure on designers up front to accurately predict the future concerning what features will be added down the road, and to build in support for them now.

A different approach that is typified by agile development processes says that a program should always use the simplest design that supports the currently-implemented feature set. Less effort is made to predict the future, or to build in mechanisms to support future features. As new features are implemented, the design is refactored to support them. The assumption is that disciplined refactoring makes changing the design sufficiently inexpensive, and our predictions of the future are almost always wrong, thus wasting the work we might do to support future features. <u>Up-front design is still important, but not as important.</u>