All design evolves. Even the most talented architect doesn’t produce a design for a building without mental experimentation, repeated conceptualization, iteration, model-building, or trial-and-error. In any case, designers don’t work alone: They have customers to satisfy. Even the most aware customer has to evolve a conception of what he wants and needs, and has to flesh out the details of his requirements.

The agile approach embodies these evolution processes in the form of working code, together with helpful UML. Agile methods constitute a new approach to software design. In the past it has not been effective to modify code as requirements evolve: However, object-orientation, improved development environments, and, especially, refactoring, have made this increasingly effective. This chapter explains refactoring and describes how it affects software design. The learning goals of this chapter are shown in Figure 2.

**Goals of this Chapter**

- Learn Fowler’s refactoring categories
- Understand how refactoring influences the design process
- Learn high-level refactorings
- Anticipate refactoring to modules
This part of our investigation of software engineering concerns design; later parts concern code. However, one of the principles of agile methods is that coding is, in large measure, design. For this reason, the reader will find some of the discussion below to be at the code level.

21.1 Introduction to Refactoring

Engineering traditionally follows a waterfall-like process. For example, in building a bridge we first assess the requirements (e.g., traffic capacity, longevity), create a design (e.g., suspension bridge, concrete footings), and then implement it (e.g., pour concrete, build supports). It would be absurd for a construction team to arrive at a riverside without a design, and begin pouring concrete for a bridge without a design! Software projects, however, have a significant capacity for alteration, matched by the changeability of requirements. This is a central tenet of Agile Programming.

A substantial discipline for code alteration has been created. It is termed refactoring, introduced to a wide audience by Fowler in his classic book [Refactoring]. Refactoring is a process of changing the architecture, design and implementation of an application while retaining the same functionality. Motives for refactoring vary but a principal one is to better accommodate maintenance, especially enhancement. Here is an example that provides the flavor of refactoring.

“Extract Class” Refactoring

To accommodate increased scope, we often need to introduce a new class to replace an attribute. For example, suppose that we already have a class Automobile with integer variable mileage.

```java
class Automobile
{
    int mileage;
    . . .
}
```

We may decide later that “mileage” for used automobiles is substantially more involved than a single int variable. For example, the auto’s motor may be a replacement, resulting in a motor mileage different from a chassis mileage. In addition, if our application were required to account for fraud, the reported “mileage” would have to be modified by other attributes such as whether the car had ever been stolen. For these reasons, we would consider the “promote attribute to a class” refactoring. This would involve introducing a class like the following.

```java
class Mileage
{
    int nominalMileageValue = 0;  // shown on odometer
    int chassisMileageValue = 0;  // best estimate
}
```
int engineMileageValue = 0; // best estimate, accounting for replacement

    // to obtain estimate

public int computeEffectiveMileage(){ … }

class Automobile
{
    Mileage mileage;
    // . . .
}

The classic reference on refactoring is [Fowler]. In his book, Fowler includes detailed
steps to migrate an application from its original form to its refactored form. Fowler
organizes his refactoring as in Figure 3.

    Fowler’s Refactoring Taxonomy

1. Composing Methods
2. Moving Features Between Objects
3. Organizing Data
4. Simplifying Conditional Expressions
5. Making Method Calls Simpler
6. Dealing With Generalization
7. Big Refactorings

Figure 3

This chapter will introduce the refactorings listed in Fowler’s classical book
[Refactoring]. Those with impact at the highest levels of design will be discussed in
detail.

21.2 Refactoring by Composing Methods
Composing Methods*

- Extract method
- Inline method
- Inline temp (remove a temporary variable)
- Replace temp with query (i.e., a function)
- Introduce explaining variable (to replace complicated expression)
- Split temporary variable (i.e., used more than once)
- Remove assignment to parameters
- Replace method with method object
- Substitute algorithm

*Fowler’s taxonomy

Figure 4

21.3 Refactoring by Moving Features between Objects

These are summarized in Figure 5.

Moving Features Between Objects 1*

- Move Method
  - Trades off method holding vs. usage

- Move Field
  - Trades off holding vs. usage

- Extract Class
  - Encapsulate a set of attributes and methods of a class

- Inline Class
  - Opposite of Extract Class

*Fowler’s taxonomy

Figure 5
Moving Features Between Objects 2*

- **Hide Delegate** Hide class dependencies from client classes**

- **Remove Middle Man**
  - Opposite of **Hide Delegate**

- **Introduce Foreign Method**
  - Code belongs on a non-modifiable serving class

- **Introduce Local Extension**
  - When several methods as in **Introduce foreign method**
  - Create subclass of server

*Fowler’s taxonomy    ** See below

Figure 6

Figure 7 shows **Hide Delegates** in detail.

21.4 Refactoring by **Organizing Data**

In the refactorings of this section, Fowler collected those having to do with the location of data in an OO design and implementation.
Organizing Data 1*

- **Self Encapsulate Field**
  - Change direct access of an attribute to accessor use
  - clarification

- **Replace Data Value with Object**
  - extensibility

- **Change Value to Reference**
  - class Order { Customer customer; ...}
  - class Order { private Customer getCustomer() String .... } ...
  - extensibility

- **Replace Array with Object**
  - extensibility

*Fowler's taxonomy

Figure 8

Organizing Data 2*

- **Change Unidirectional Association to Bidirectional**
  - (Only if necessary.) Install backpointer.

- **Change Bidirectional Association to Unidirectional**
  - Find a way to drop; consider third party (repair)

- **Replace “Magic Number” with Constant** (extensibility)

- **Encapsulate Field** (repair)
  - public attribute to private/accessor

- **Encapsulate Collection** (extensibility)
  - FROM: Accessing a collection, as in store getCustomers()
  - TO: member accessors, as in get(Customer),
    - add(Customer), remove(Customer)

*Fowler's taxonomy

Figure 9
21.5 Refactoring with Generalization

The refactorings in this section improve the uses of inheritance.
Dealing with Generalization 1*

- Pull up field (extensibility)
- Pull up method (extensibility)
- Pull up constructor body (extensibility)
  o Replace by super(...)
- Push Down Method (clarification)
  o When base class method not used by most subclasses
- Push Down Field (clarification)

*Fowler's taxonomy

Figure 12

Dealing with Generalization 2*

- Extract Subclass (extensibility)
- Extract Superclass (extensibility)

*Fowler's taxonomy

Figure 13

Dealing with Generalization 3*

- Extract Interface (extensibility)
- Collapse Hierarchy (repair)
  o Inherited class not special enough

*Fowler's taxonomy

Figure 14
21.6 Big Refactorings

The refactorings in this section are mainly at the class level.
Big Refactorings 1*

- Tease Apart Inheritance

  Responsibility
  Employee type
  Employee status

  Employee

  TechEmployee
  NonTechEmployee

  RetiredTechEmployee
  RetiredNonTechEmployee

  ActiveTechEmployee
  ActiveNonTechEmployee

  Status

  Active
  Retired

⇒

*Fowlers' taxonomy

Figure 17

Big Refactorings 2*

- Convert Procedural Design to Objects

  Control
  startGame()
  displayCharacter()
  moveCharacter()

  VideoGame

  GameCharacter

⇒

  VideoGame
  start()

  GameCharacter
  display()
  move()

*Fowlers' taxonomy

Figure 18
21.7 Refactoring by Introducing Modules

Good design requires modularization; the process of separating the elements. Whenever feasible, this should be performed in advance. For agile methods are to succeed for complex applications, however, it is necessary to modularize after the fact as well: In other words, to recognize useful modularization as the application grows. A principal way to handle modularity is by means of the Façade design pattern. Simplifying matters to begin with, the problem can be reduced to that shown in Figure 21. The class \( U \) must be modified because it may no longer reference classes \( V \) and \( W \).
This refactoring is simple if $U$ merely uses functionality provided by $V$ and $W$. The situation may not be so simple, however. We can reduce the problem of modularization to that of a dependency between two classes, $U$ and $V$. If $U$ depends on $V$ in an initial design and we want to protect $V$ within a package, then $U$ has to depend on a façade interface $F$ and no longer on $V$ directly. Figure 22 shows how this can be dealt with, generally but in theory. First, $V$ is provided with an abstract interface $VI$ (step 1) that abstracts its public methods – if it does not already have one. The Extract Interface refactoring can be used for this. Next, in step 2, $V$ is enclosed in a module with façade $F$. The class $U$ must now be modified so as to reference $VI$ and $F$ only. We discuss the mechanics of this below.

Let’s apply this to an example, shown in Figure 23, where a transaction object of class $Trnsctn$ depends on a customer class $Cust$. 
Figure 23

There is no problem introducing \textit{ICust}, and \textit{Cust} is unchanged; but we do need to deal with the changes needed to \textit{Trnsctn}. Figure 24 lists the possibilities involved, and introduces remedies.

\begin{itemize}
\item \textit{Trnsctn} has attribute of type \textit{Cust}
  \begin{itemize}
  \item Replacing \textit{Cust cust}, with \textit{ICust cust}; \hspace{1cm} \text{OK if}
    \begin{itemize}
    \item \textit{Cust} has only private attributes (i.e., use via methods only)
    \item \textit{Trnsctn} does not instantiate \textit{Cust} instances
    \end{itemize}
  \end{itemize}
\item \textit{Trnsctn} has method with parameter of type \textit{Cust}
  \begin{itemize}
  \item Replacing \textit{myMethod(Cust)} with \textit{myMethod(ICust)} \hspace{1cm} \text{OK if}
    \begin{itemize}
    \item \textit{Cust} has only private attributes (i.e., use via methods only)
    \end{itemize}
  \end{itemize}
\item \textit{Trnsctn} has method returning type \textit{Cust}
  \begin{itemize}
  \item Replacing \textit{Cust myMethod()}, with \textit{ICust myMethod()}; \hspace{1cm} \text{OK}
  \end{itemize}
\item \textit{Trnsctn} has method with variable of type \textit{Cust}
  \begin{itemize}
  \item Replacing \textit{Cust cust}, with \textit{ICust cust}; \hspace{1cm} \text{Unresolved}
  \end{itemize}
\end{itemize}

Figure 24

These remedies suggest techniques for coding classes in the first place that allow for subsequent modularization. Some of these techniques have already been mentioned. The first is to code without public variables. In this way, access is always through methods and can thus be abstracted to interfaces. The second is to avoid instantiation except from setup classes: This is called \textit{dependency injection} and is commonly used in frameworks such as \textit{Spring}. Its form is frequently a line such as the following, in class \textit{Setup}.

\begin{verbatim}
Trnsctn trnsctn = new Trnsctn( new Cust( ... ) );
or trnsctn setCust( new Cust( ... ) );
\end{verbatim}
These techniques are summarized in Figure 25 where Setup handles the dependency injection. Setup therefore required to have unrestricted access to classes within packages.

### Preparing for Modularization

- **Make variables private or protected**
  - Eliminate possibility of direct reference from outside
- **Avoid instantiation**
  - Use dependency injection

![Diagram](image)

**Figure 25**

### 21.8 More Design Principles

Designing in an agile fashion requires one to be flexible, using refactoring as a tool. We have already seen that designing to interfaces rather than to implementations makes it easier to refactor to modules. Martin cites eleven object-oriented principles that facilitate agile design, as listed in Figure 26, Figure 27, and Figure 28. These are good principles for design in general.

**Object-Oriented Design Principles for Agility**

- **Single Responsibility Principle**
  - Each class should have only one responsibility
- **Open-Closed Principle**
  - Entities should be open for extension but closed for modification
- **Liskov Substitution Principle**
  - Subtypes should be substitutable for their base types
- **Dependency Inversion Principle**
  - Details should depend on abstractions; and not vice versa

Object-Oriented Design Principles for Agility* 2

- **Interface Segregation Principle**
  - Clients should not be forced to depend on methods they don’t use

- **Release-Reuse Equivalency Principle**
  - Release increments are the viable granularity of reuse

- **Common Release Principle**
  - Reuse should be at the package level

- **Common Closure Principle**
  - Package so that each change affects only one


Object-Oriented Design Principles for Agility* 3

- **Acyclic Dependency Principle**
  - There should be no cycles in package dependency

- **Stable Dependency Principle**
  - Dependency should be towards stability
    (*more stable* means *less likely to change*)

- **Stable Abstractions Principle**
  - Each package should be as abstract as it is stable
    (We often abstract whatever is most stable)

21.9 Summary

Figure 29 summarizes this chapter.

**Summary**

- Refactoring allows **design / code / redesign**
- Low- and **high-level** refactorings are possible
- **Refactoring to modules** can be anticipated

Figure 29