Code Refactoring

What is refactoring?

- What is really the code refactoring and why we need to do that?

- After freezing the application features, we need to go over the code and modify it so that it runs the same designed features correctly, but **it is better understandable and easier to extend** in future.

- Refactoring makes the code understandable, easier to extend, and performant (sometimes runs with better performance).
- Refactoring makes it easier to extend it to more complex and complicated system.
- **Code refactoring is NOT debugging and removing bugs.**

What is refactoring?

" Refactoring is the process of changing a software system in such a way that it **does not alter the external behavior of the code yet improves its internal structure**.

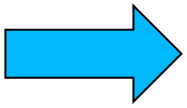
It is a disciplined way **to clean up code** that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written. …

With refactoring you can take a bad design, chaos even, and rework it into well-designed code."

Fowler et al. 1999

Book: Refactoring Improving the Design of Existing Code. by Martin Fowler

Fowler's Refactoring Taxonomy



Why refactoring in design?

Refactoring tooling

Big Refactorings

Composing Methods

Moving Features Between Objects

Organizing Data

Dealing With Generalization

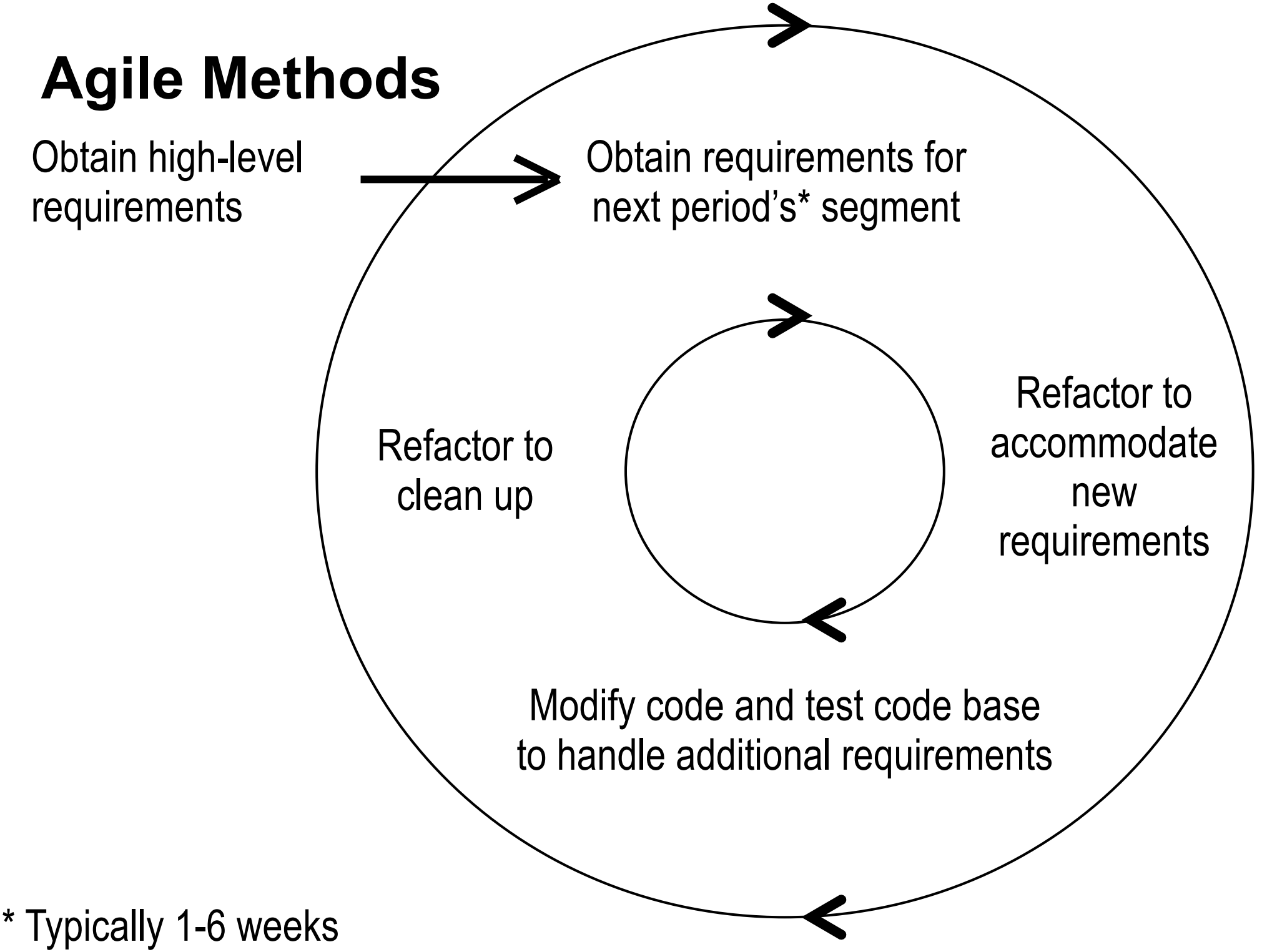
Making Method Calls Simpler

Book: Refactoring Improving the Design of Existing Code. by Martin Fowler

Refactoring Workflow

- When should we do the refactoring?
- What is the workflow for doing the refactoring?
  - When we see that our code has implemented some good features correctly and can pass acceptance functional tests.
  - We need to start over, read and check the written code and in the case that we see some specific issues (**code has some smells**), we need to apply refactoring techniques based on the specific issue.
- Be careful that **you might not agree in some of the situations** with some of refactoring changes because it might have some negative effects on the behavior of your application.
- This is **all about programmer experiences and solutions**.
- There is not only one good solution.

Agile Methods



* Typically 1-6 weeks

Practical Refactoring Tip

Build new components alongside existing ones. Plan for substitutions where possible.

Test the new ones thoroughly.

Avoid altering existing components until the end—when you may have to make alterations.

Code Smells

- **Bloaters**
  - Large code parts that is hard to work with like methods and classes that have increased to large proportions.
- **Object-Orientation Abusers**
  - Incorrect application of object-oriented programming principles like large inheritance hierarchies
- **Change Preventers**
  - In the case that you need to change the code in one place, you have to make many changes in other places too.
- **Dispensables**
  - Something unnecessary in the code whose absence would make your code cleaner, more efficient and easier to understand.
- **Couplers**
  - Excessive coupling between classes

Bloaters

It is all about large code parts

- **Long Method**
- **Large Classes**
- **Long Parameter List**
- **Primitive Obsession**
  - Usage of primitive types instead of small objects (not always)
  - Usage of constants for coding information
  - Usage of an array that contains various types of data
- **Data Clumps**
  - You have different parts of the code contain identical groups of variables (you would then extract a class for it)

Object-Orientation Abusers

- **Switch Statements**
- **Temporary Field**
  - Temporary fields are used inside methods for some algorithm calculation. You see a large number of parameters in the method.

    Extract a class for them and include a method for the calculation.
- **Refused Bequest**
  - You have a subclass uses only some of the methods and properties inherited from its parents but two classes are different from each other.
- **Alternative Classes with Different Interfaces**
  - Two classes have identical functions but have different method names.
  - The programmer who create the second one has no info about the first one.

Change Preventers

- **Shotgun Surgery**
  - You need to make a single change to multiple classes simultaneously
- **Divergent Change**
  - When you want to add a new feature, you find yourself to need to change many unrelated methods and classes.
  - Many changes are made to a single class
- **Parallel Inheritance Hierarchies**
  - You find yourself needing to create a new subclass to the hierarchy
  - With each new classes added, making changes will be harder and harder.

Dispensables

- **Comments**
- **Duplicate Code**
- **Dead Code**
- **Lazy Class**
  - Understanding and maintaining classes is costly. We do not need very small class because we then need to maintain them.
- **Data Class**
  - A class that contains only fields and crude methods for accessing them (getters and setters)
  - True power of objects is that they can contain behavior types or operations on their data
- **Speculative Generality**
  - Unused class, method, field or parameter.

Couplers

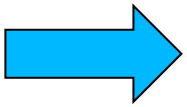
- **Feature Envy**
  - A method accesses the data of another object more than its own data.
- **Inappropriate Intimacy**
  - One class uses the internal fields and methods of another class.
- **Message Chains**
  - A series of calls resembling "a→b()→c()→d()"
  - A client requests another object, that object requests yet another one, and so on
- **Middle Man (Facade)**
  - A class performs only one action, delegating work to another class
  - You would ask why there is the middle man and what is its functionality.
- **Incomplete Library Class**
  - Use less external libraries, and create your own library from scratch.

Refactoring Techniques

In short - Use design patterns that you have learned in this class.

Fowler's Refactoring Taxonomy

Why refactoring in design?



Refactoring tooling

Big Refactoring

Composing Methods

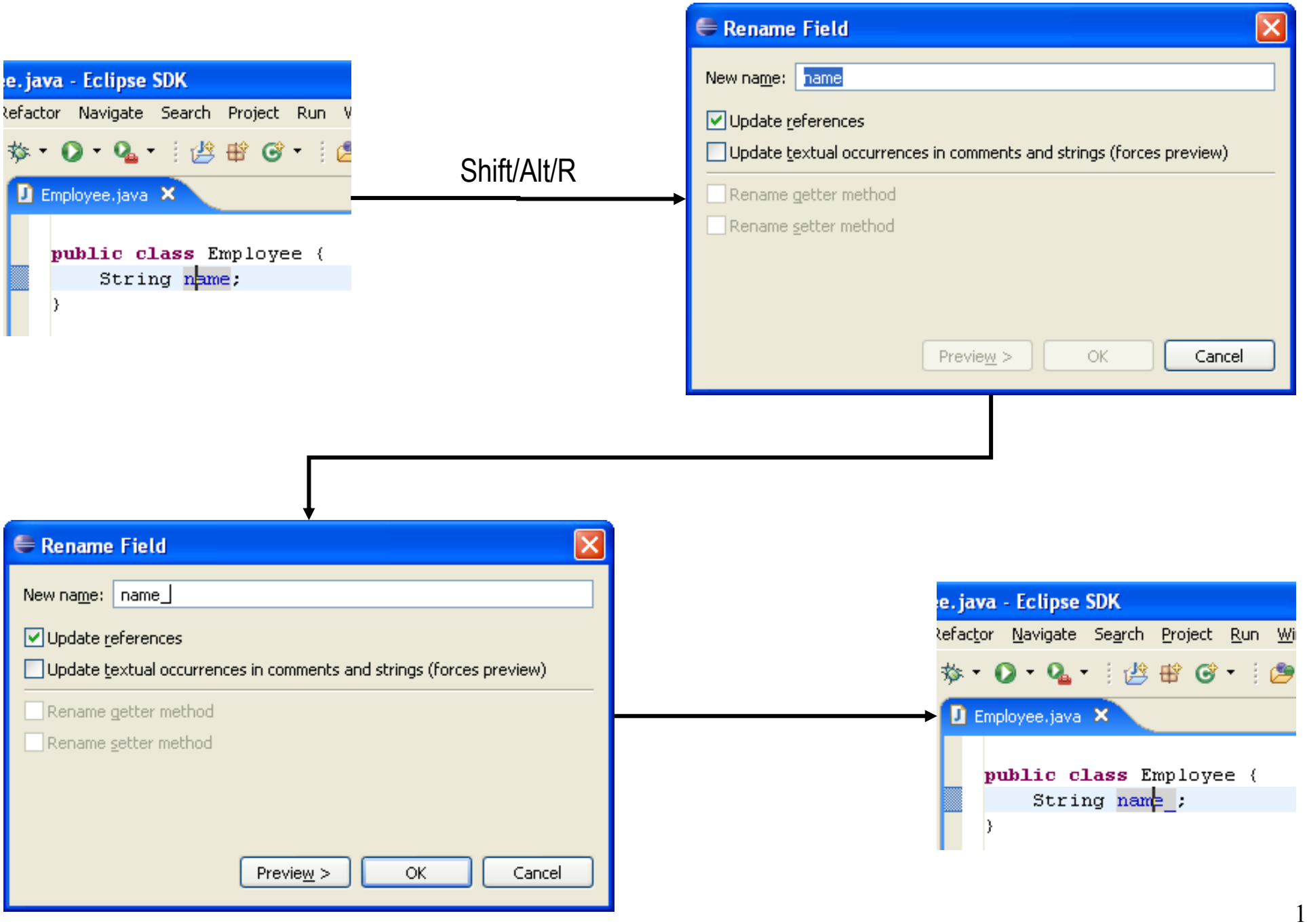
Moving Features Between Objects

Organizing Data

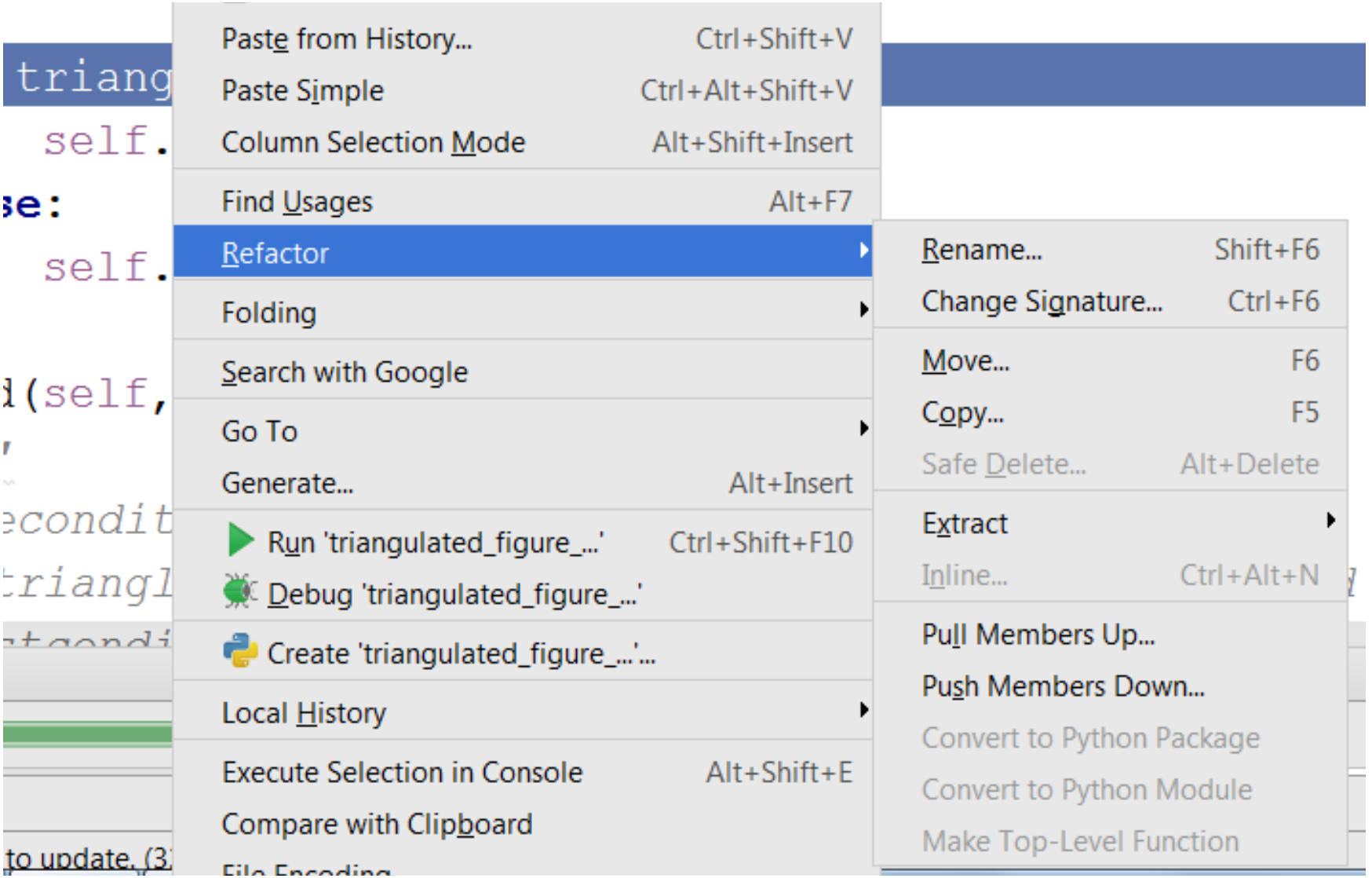
Dealing With Generalization

Making Method Calls Simpler

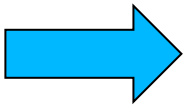
Using a Refactoring Wizard



Refactoring Menu in *PyCharm*



Fowler's Refactoring Taxonomy



Why refactoring in design?

Refactoring tooling

Big Refactorings

Composing Methods

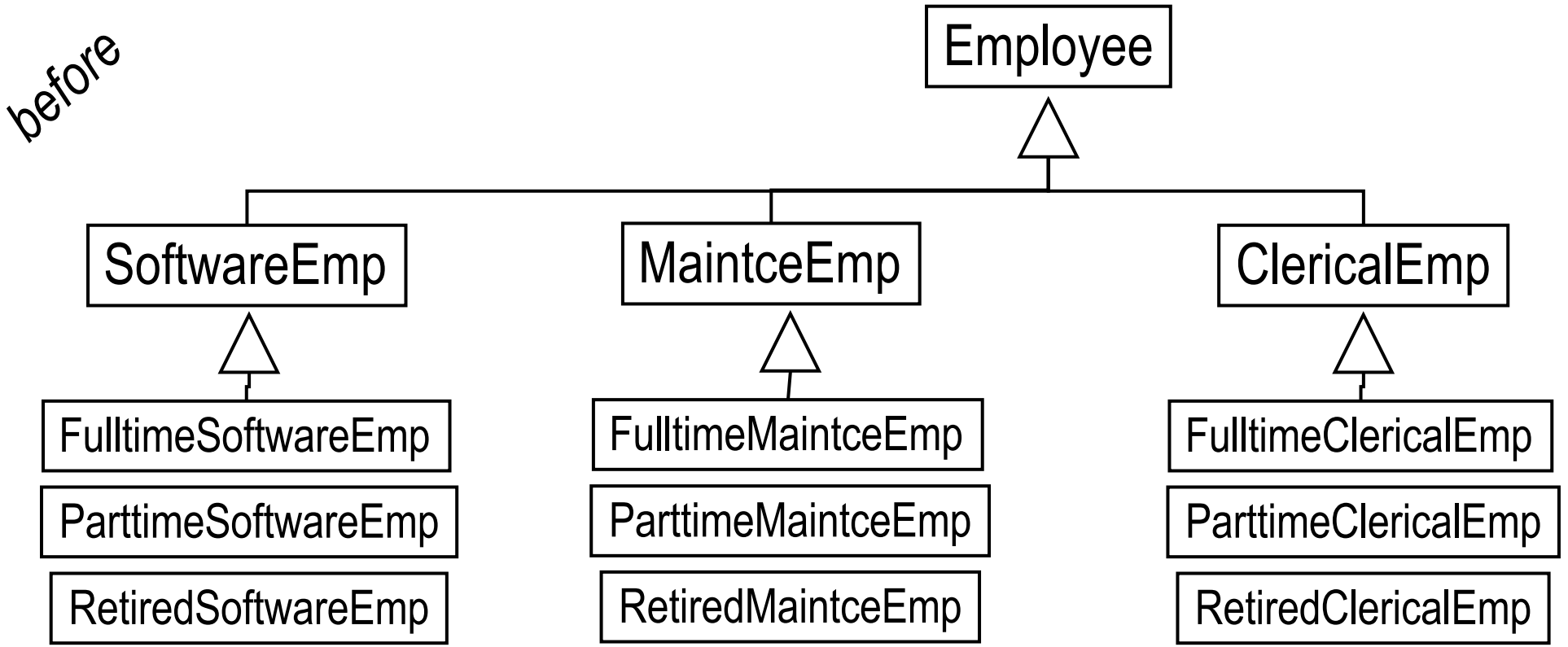
Moving Features Between Objects

Organizing Data

Dealing With Generalization

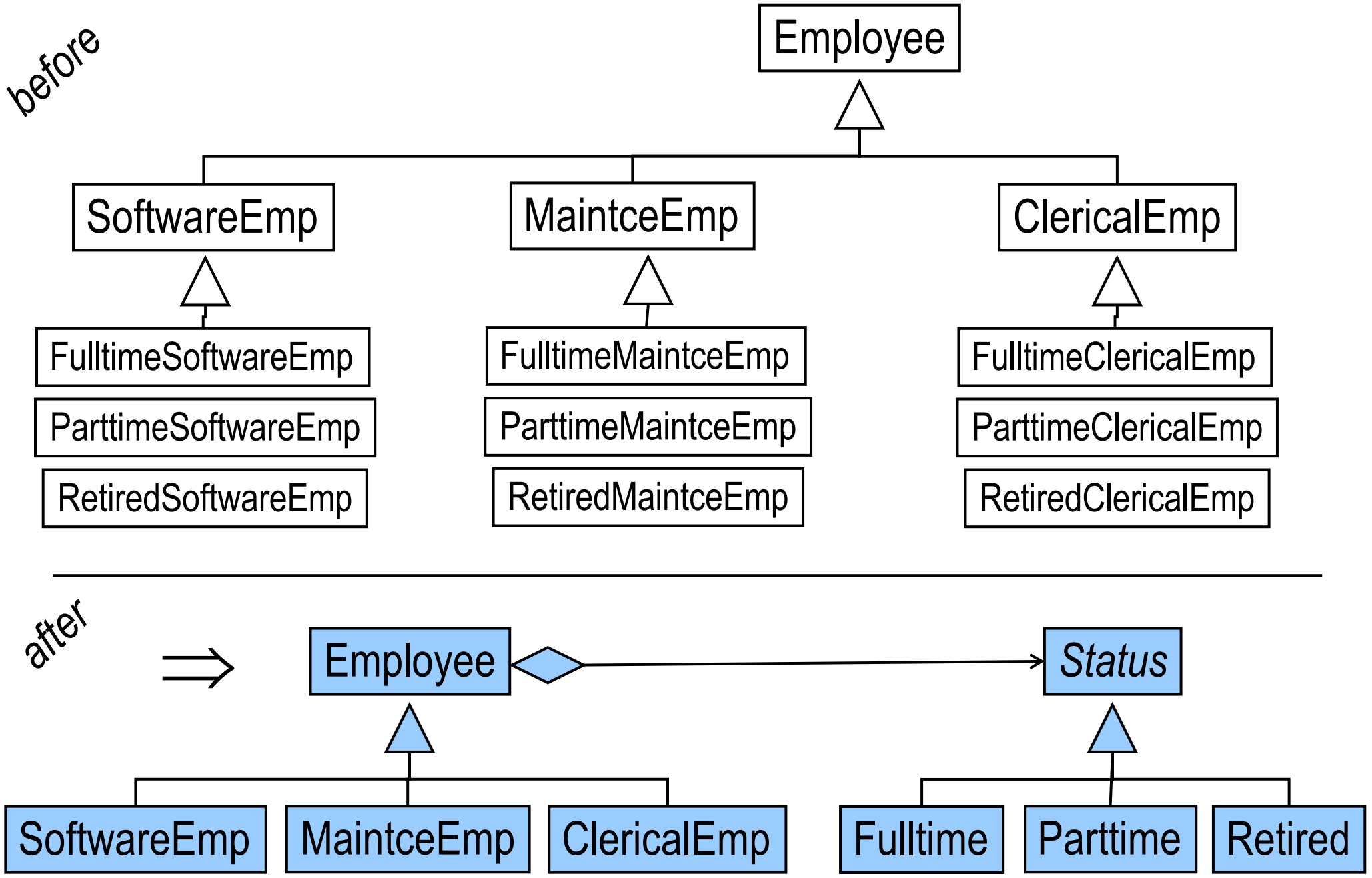
Making Method Calls Simpler

Big Refactorings 1: Tease Apart Inheritance *



*Fowlers' taxonomy

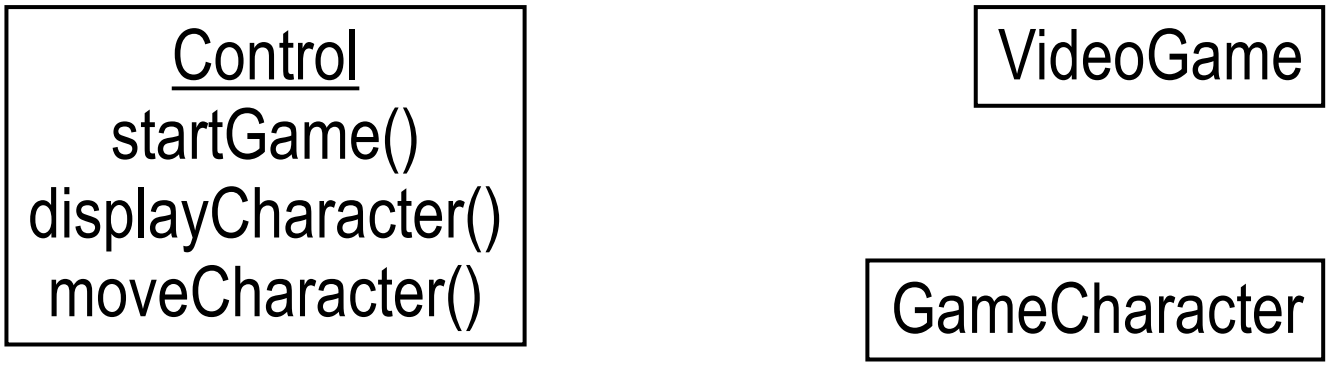
Big Refactorings 1: *Tease Apart Inheritance* *



*Fowlers' taxonomy

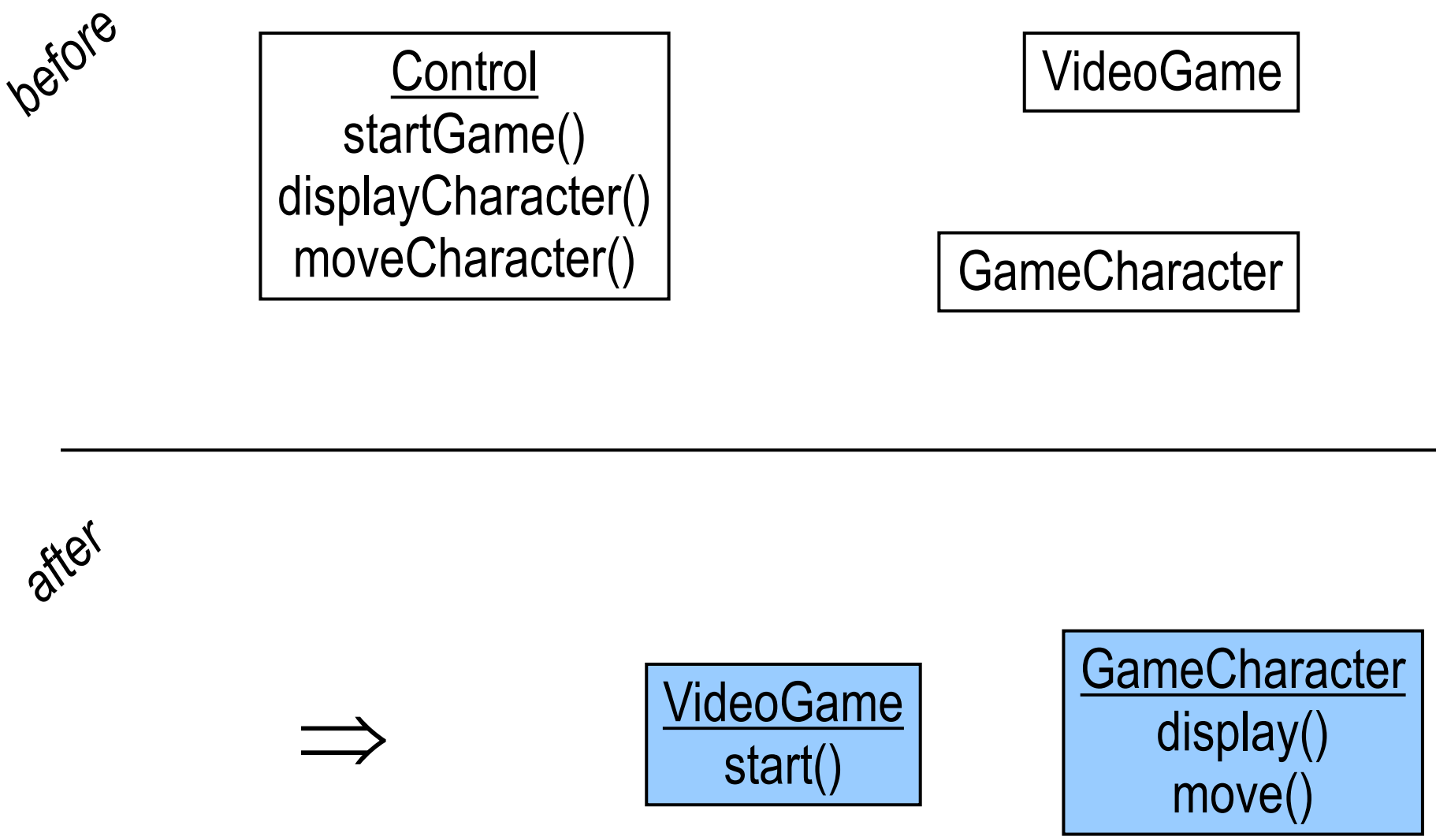
Big Refactorings 2*:
Convert Procedural Design to Objects

before



*Fowlers' taxonomy

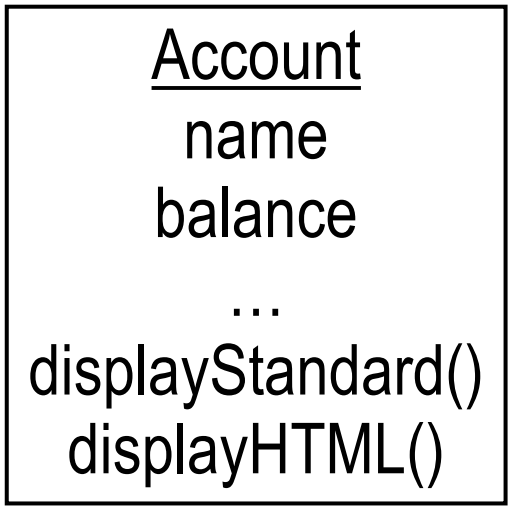
Big Refactorings 2*:
Convert Procedural Design to Objects



*Fowlers' taxonomy

Big Refactorings 3*: *Separate Domain from Presentation*

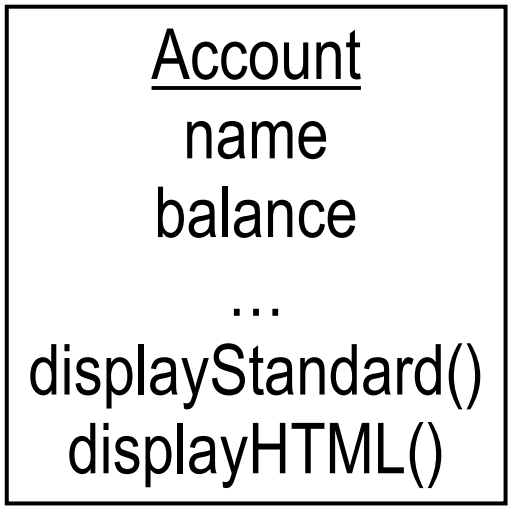
before



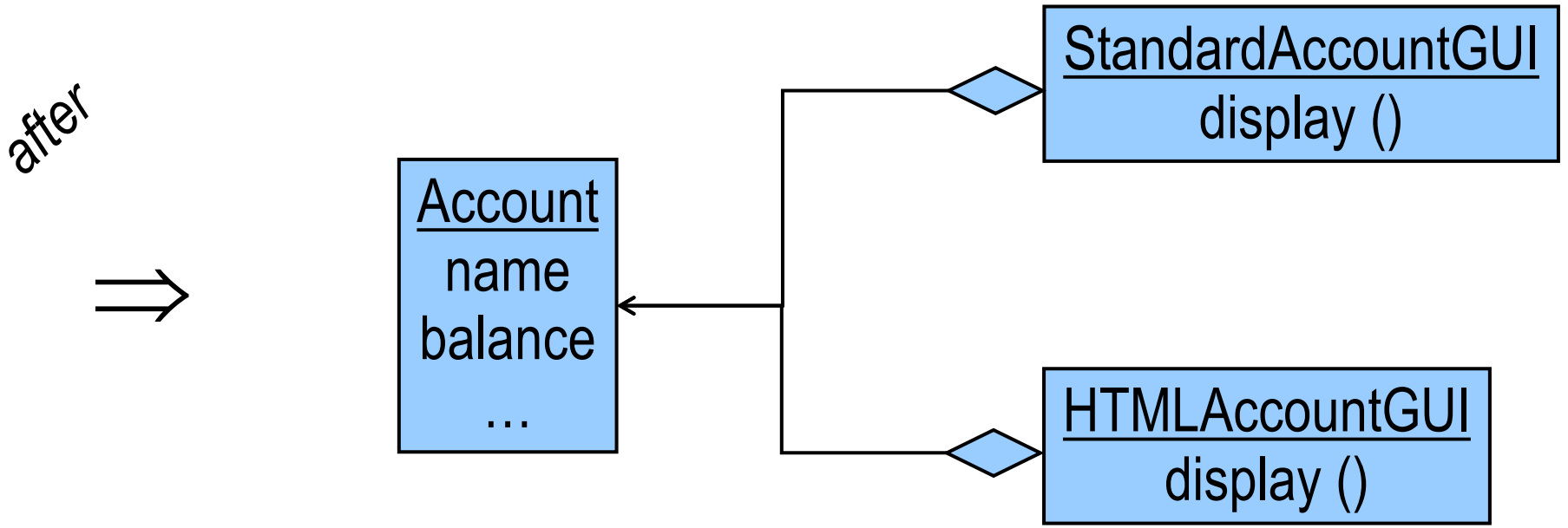
*Fowlers' taxonomy

Big Refactorings 3: Separate Domain from Presentation*

before



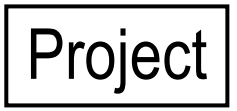
after



*Fowlers' taxonomy

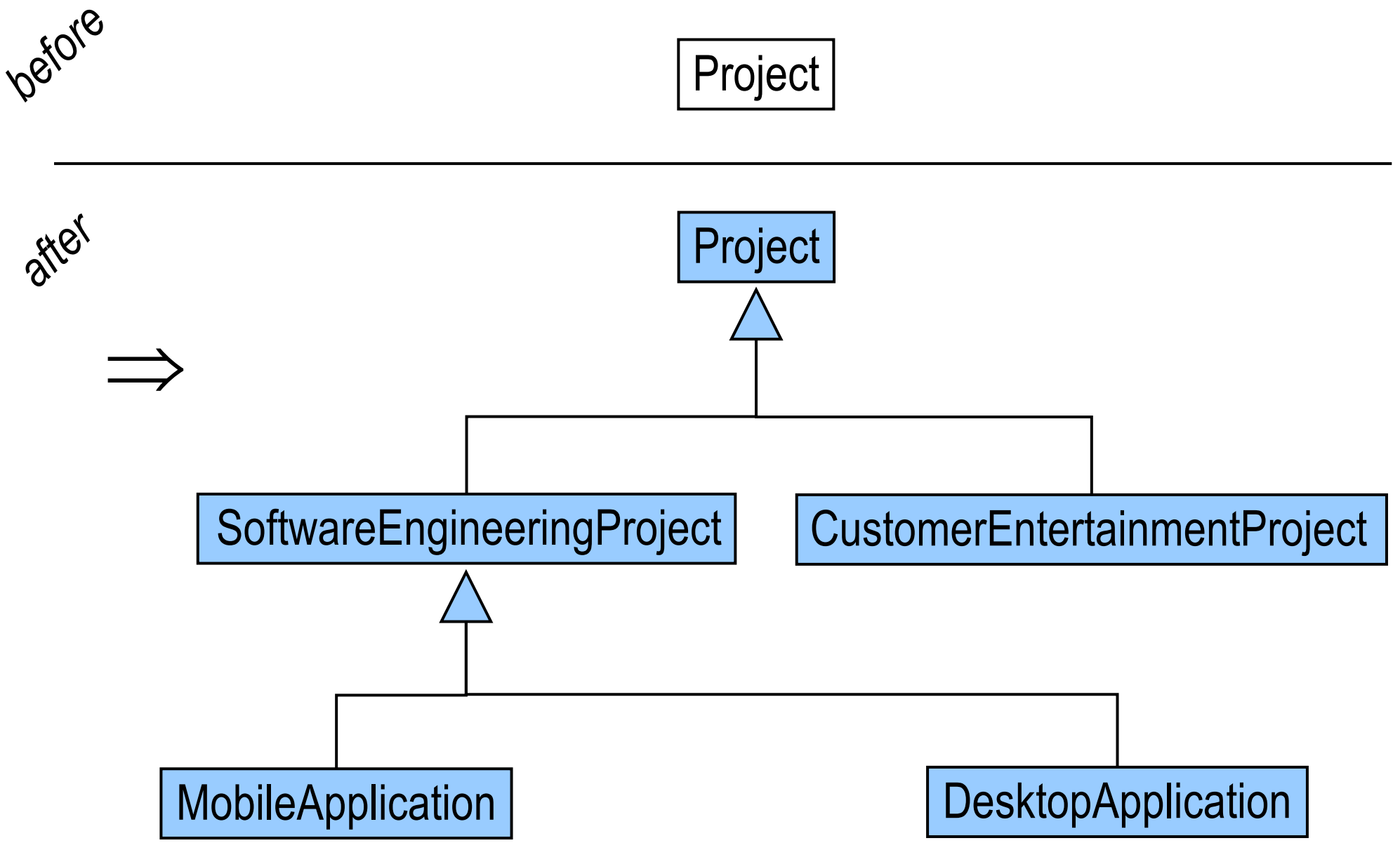
Big Refactorings 4: Extract Hierarchy*

before



*Fowlers' taxonomy

Big Refactorings 4: Extract Hierarchy*



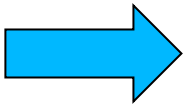
*Fowlers' taxonomy

Fowler's Refactoring Taxonomy

Why refactoring in design?

Refactoring tooling

Big Refactoring



Composing Methods

Moving Features Between Objects

Organizing Data

Dealing With Generalization

Making Method Calls Simpler

*Composing Methods**

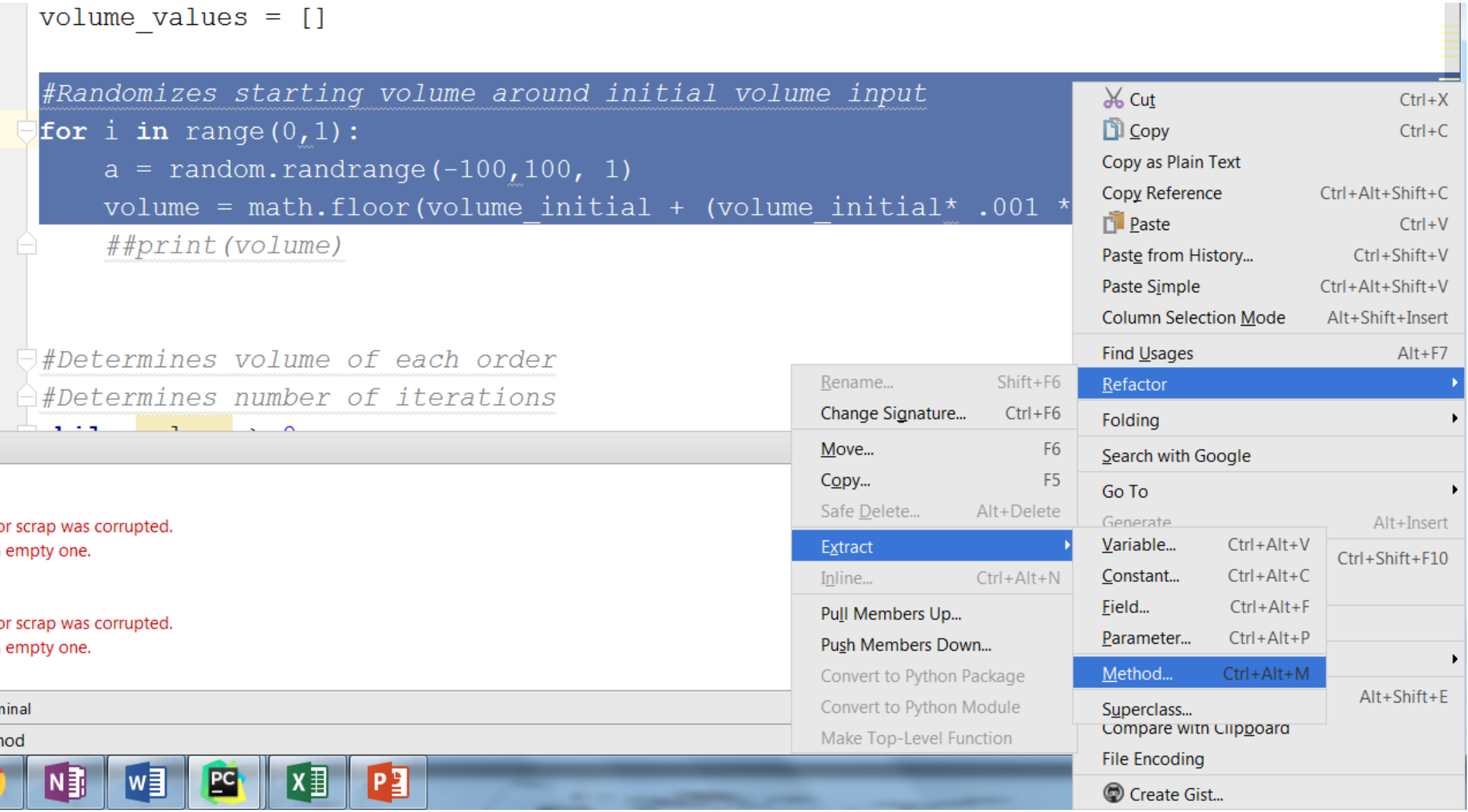
Extract method

Inline method

…

*Fowlers' taxonomy

Extract Method



*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)

…

*Fowlers' taxonomy

*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 (command pattern)

*Fowlers' taxonomy

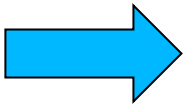
Fowler's Refactoring Taxonomy

Why refactoring in design?

Refactoring tooling

Big Refactoring

Composing Methods



Moving Features Between Objects

Organizing Data

Dealing With Generalization

Making Method Calls Simpler

Moving Features Between Objects *1**

Move Method

Trades off method holding vs. usage

Move Field

Trades off holding vs. usage

…

*Fowlers' taxonomy

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*

*Fowlers' taxonomy

Moving Features Between Objects *2**

Hide Delegate

Hide class dependencies from client classes**

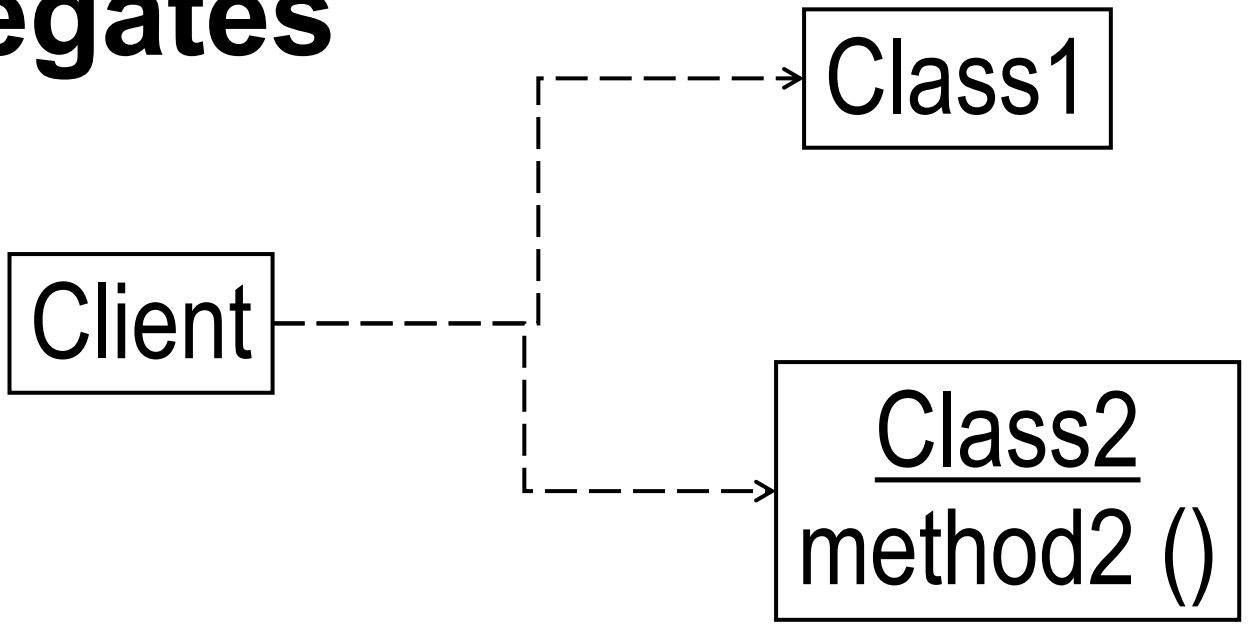
Remove Middle Man

Opposite of *Hide Delegate*

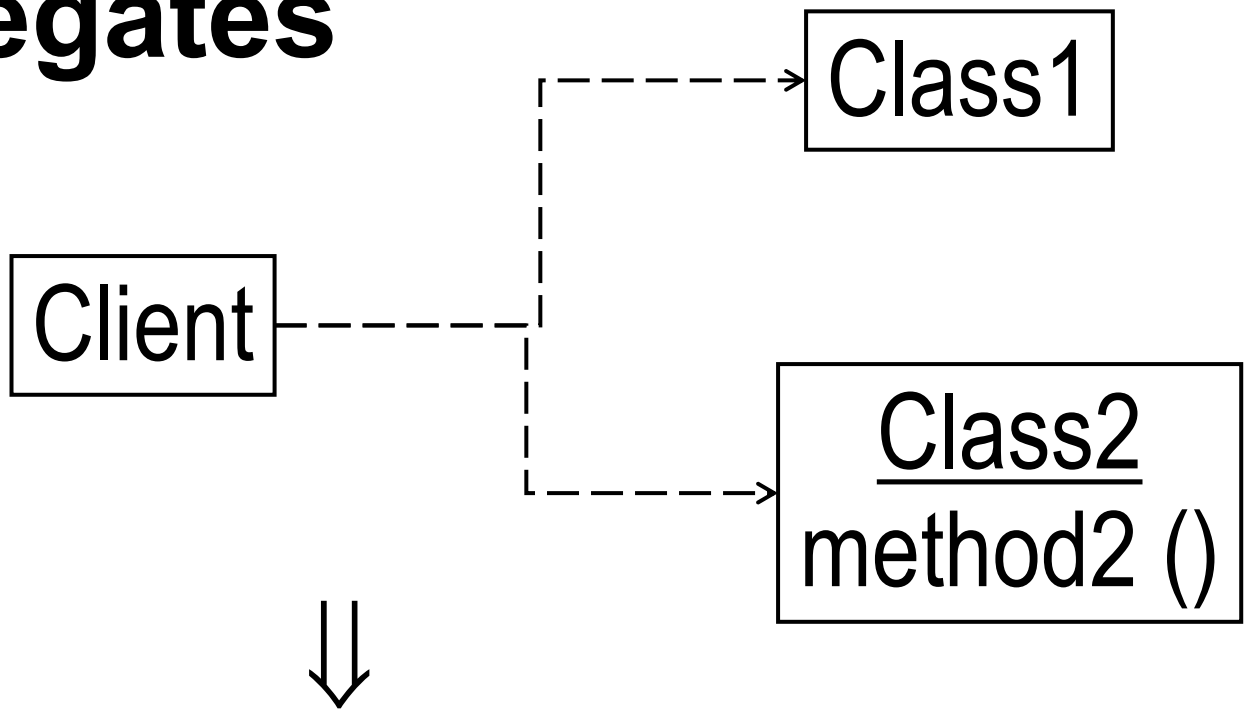
*Fowlers' taxonomy

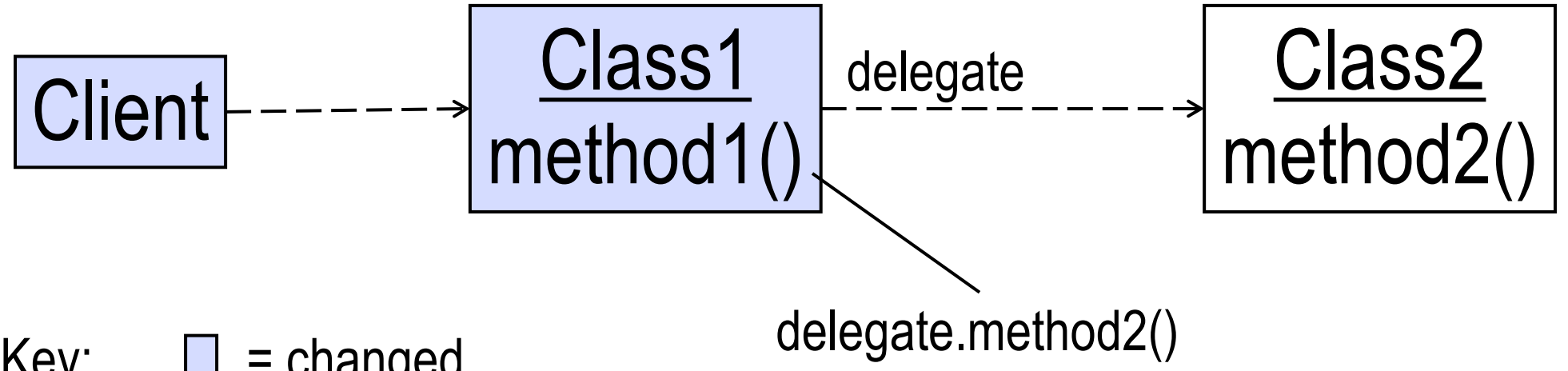
** See below

Hide Delegates



Hide Delegates





Fowler's Refactoring Taxonomy

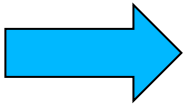
Why refactoring in design?

Refactoring tooling

Big Refactoring

Composing Methods

Moving Features Between Objects



Organizing Data

Dealing With Generalization

Making Method Calls Simpler

Organizing Data *1**

Self Encapsulate Field

Change direct access of an attribute to accessor use

Replace Data Value with Object

Change Value to Reference

class Order { Customer customer;….

class Order { private Customer getCustomer(String ….)

…

*Fowlers' taxonomy

Organizing Data *1**

Self Encapsulate Field

Change direct access of an attribute to accessor use

Replace Data Value with Object

Change Value to Reference

class Order { Customer customer;….

class Order { private Customer getCustomer(String ….)

Change Reference to Value

Replace Array with Object

*Fowlers' taxonomy

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

…

*Fowlers' taxonomy

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

Replace “Magic Number” with Constant

Encapsulate Field

public* attribute to *private/accessor

*Fowlers’ taxonomy

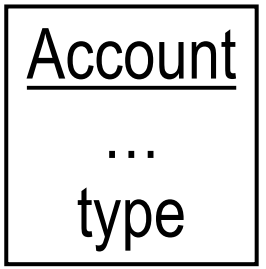
Organizing Data *3**

Replace Record with Data Class

Simplest object with private data field, accessor

Replace Type Code with Class

before



*Fowlers' taxonomy

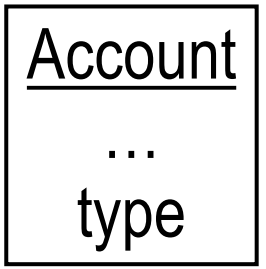
Organizing Data 3*

Replace Record with Data Class

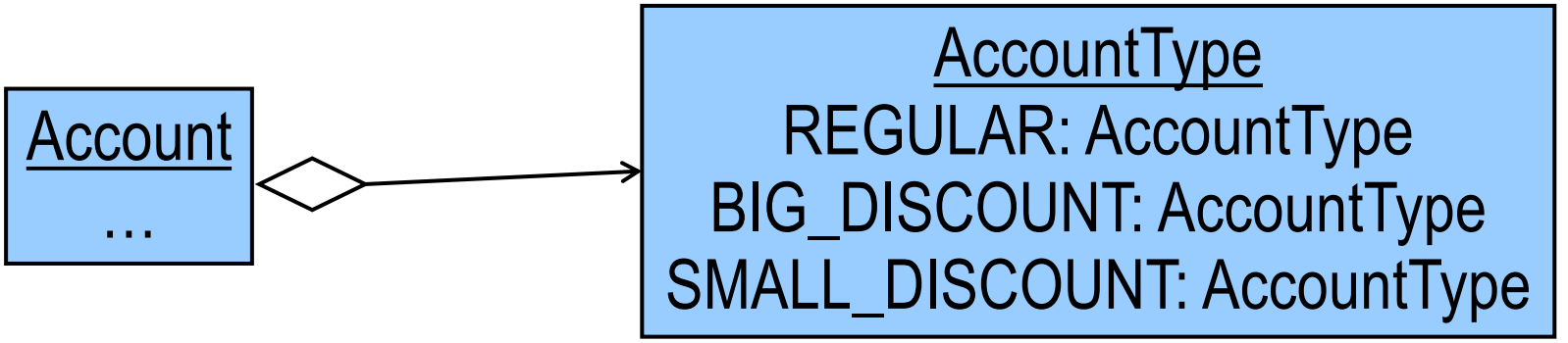
Simplest object with private data field, accessor

Replace Type Code with Class

before



after



*Fowlers' taxonomy

Fowler's Refactoring Taxonomy

Why refactoring in design?

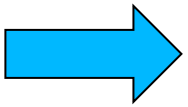
Refactoring tooling

Big Refactoring

Composing Methods

Moving Features Between Objects

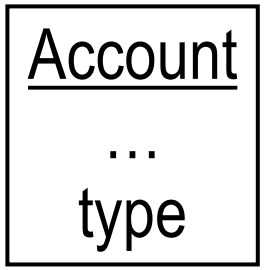
Organizing Data



Dealing With Generalization

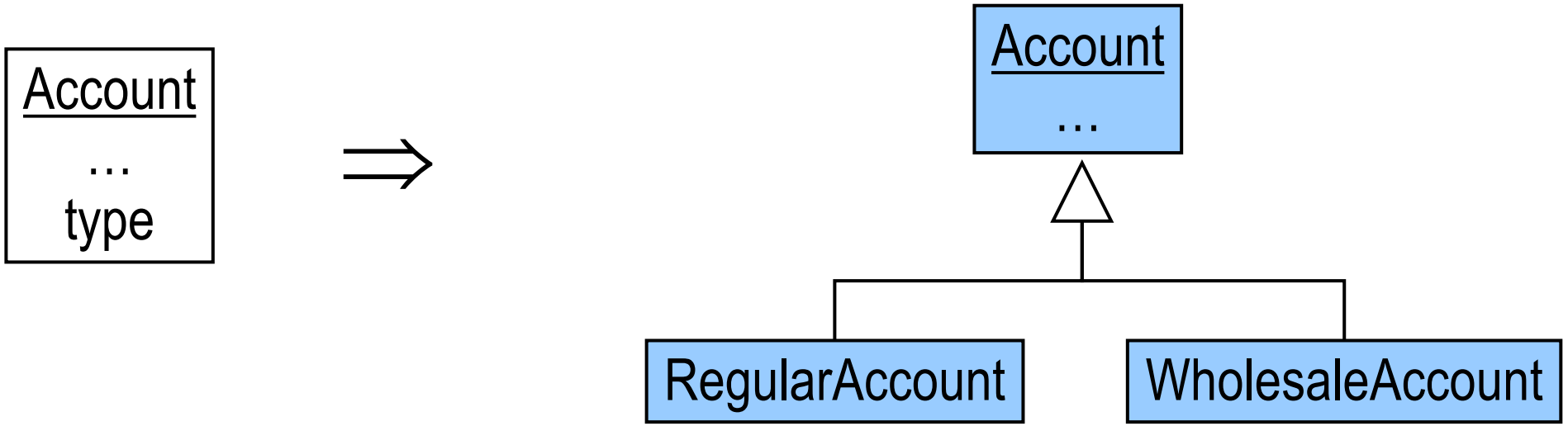
Making Method Calls Simpler

Replace Type Code with Subclass



*Fowlers' taxonomy

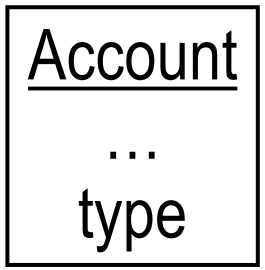
Replace Type Code with Subclass



*Fowlers' taxonomy

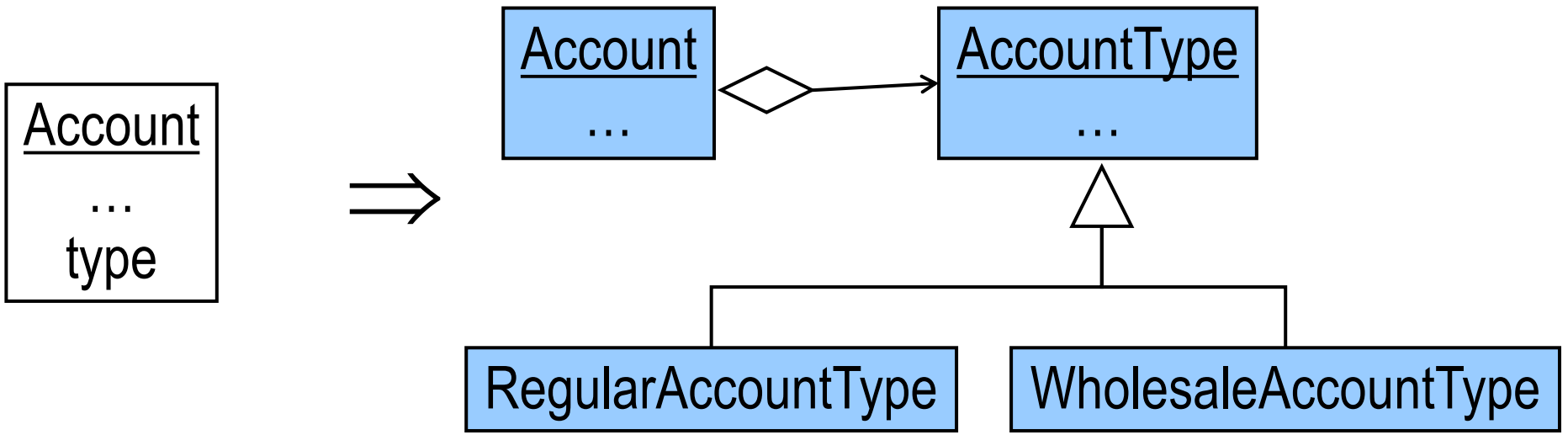
Organizing Data *4**

Replace Type Code with State/Strategy



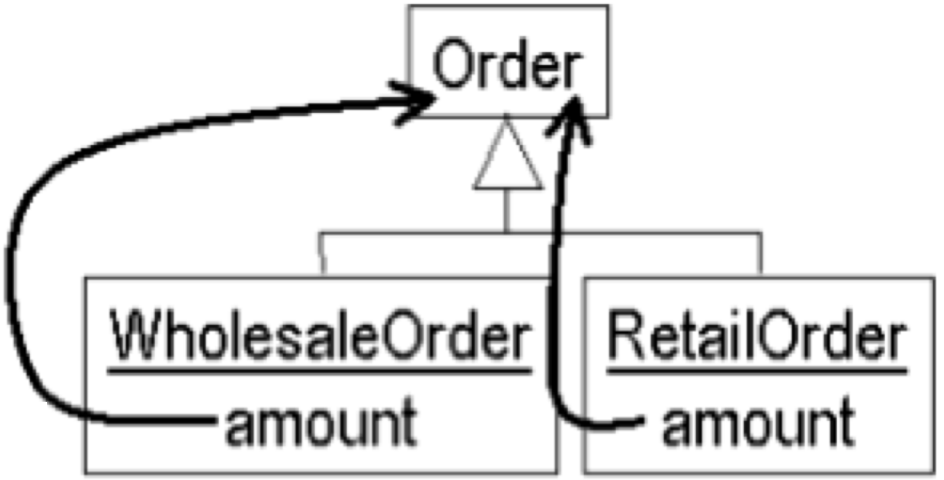
*Fowlers' taxonomy

Replace Type Code with State/Strategy



*Fowlers' taxonomy

Dealing with Generalization *1**



- **Pull up field**
- **Pull up method**
- **Pull up constructor body**
  - Replace by *super(…)*

*Fowlers' taxonomy

Dealing with Generalization *1**

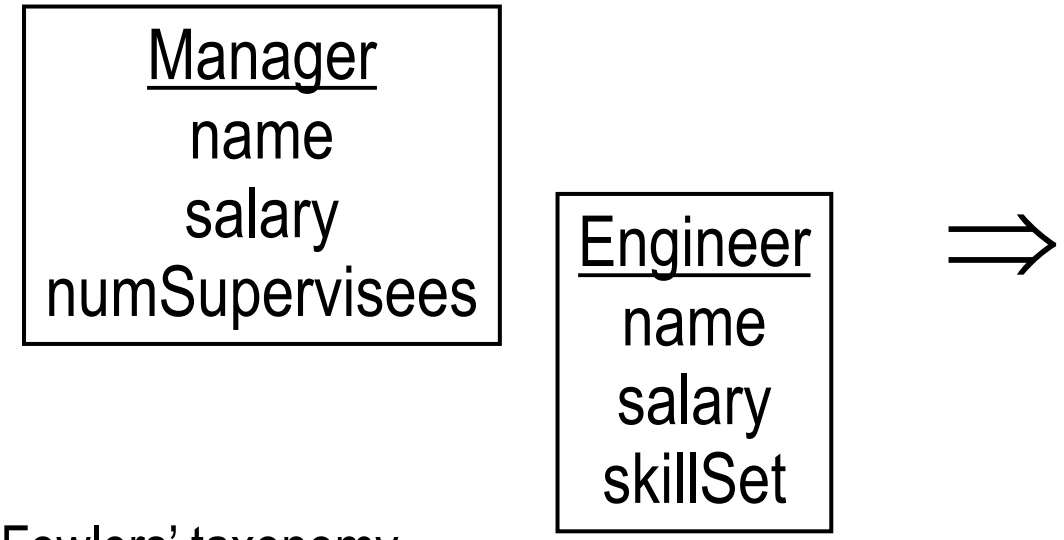
- **Push Down Method**
  - When base class method not used by most subclasses
- **Push Down Field** (clarification)

*Fowlers' taxonomy

*Dealing with Generalization 2**

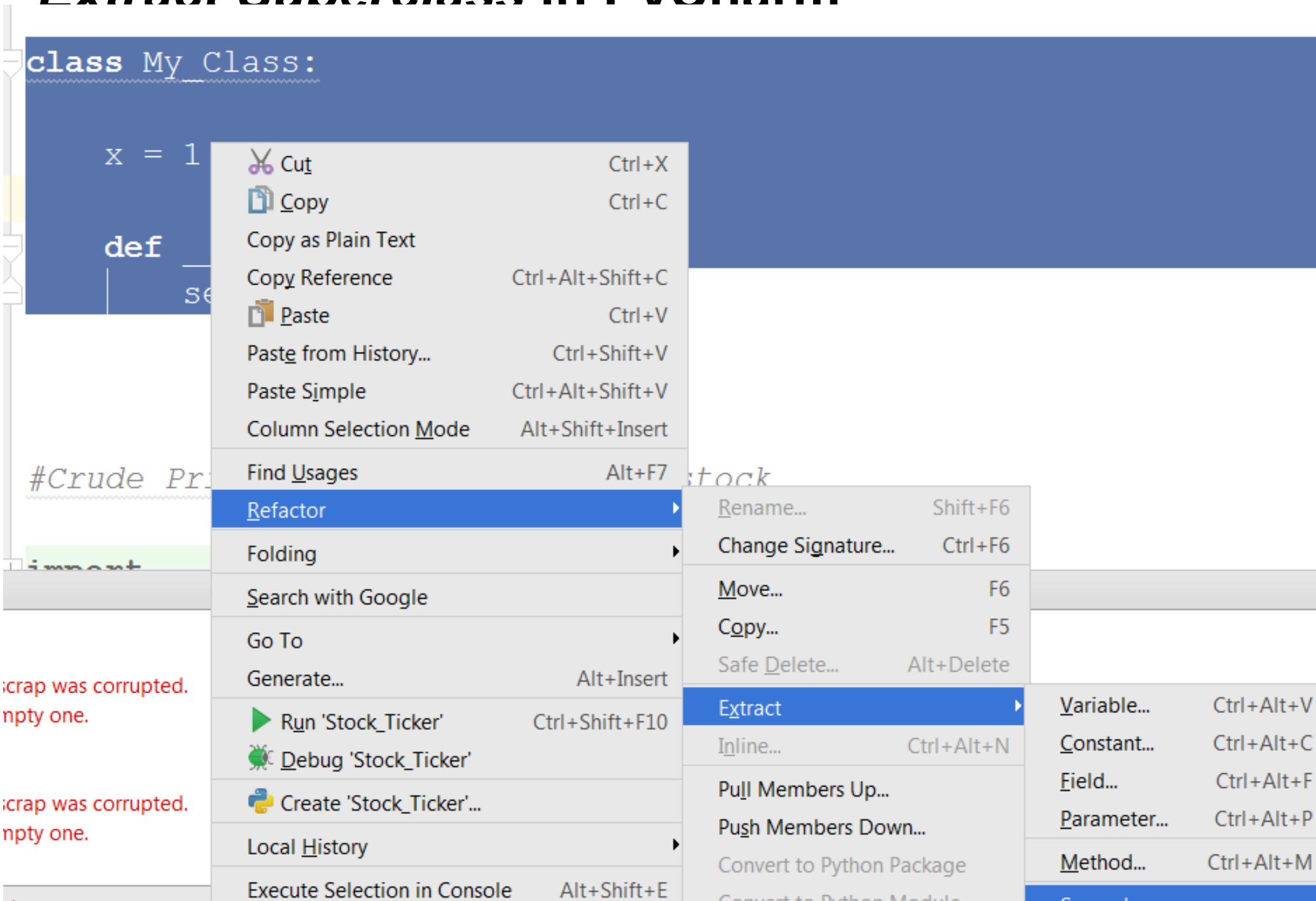
Extract Subclass

Extract Superclass



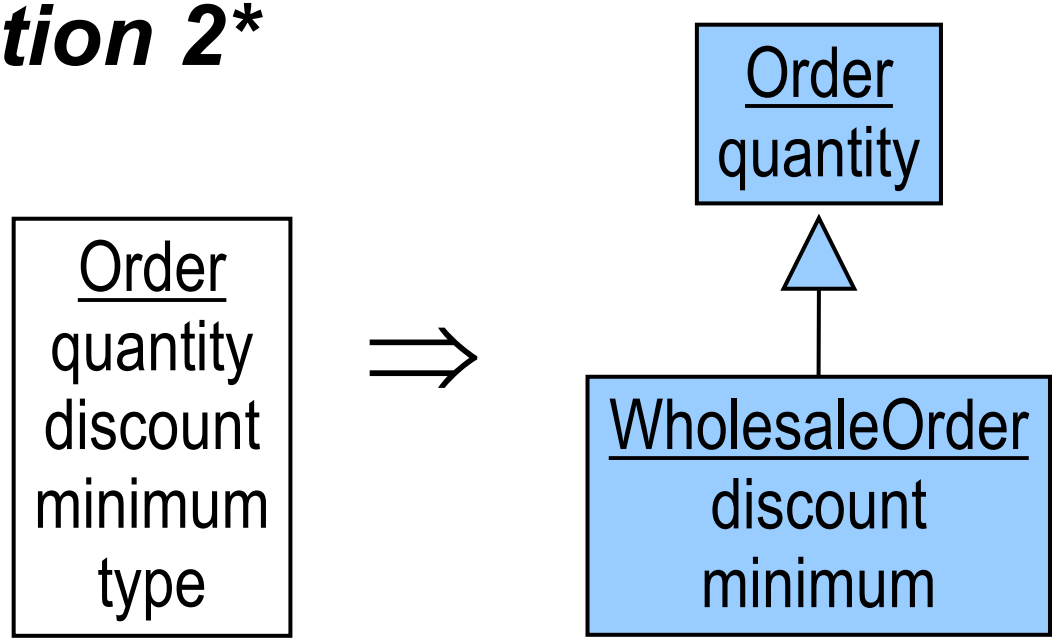
*Fowlers' taxonomy

Extract Superclass in PyCharm

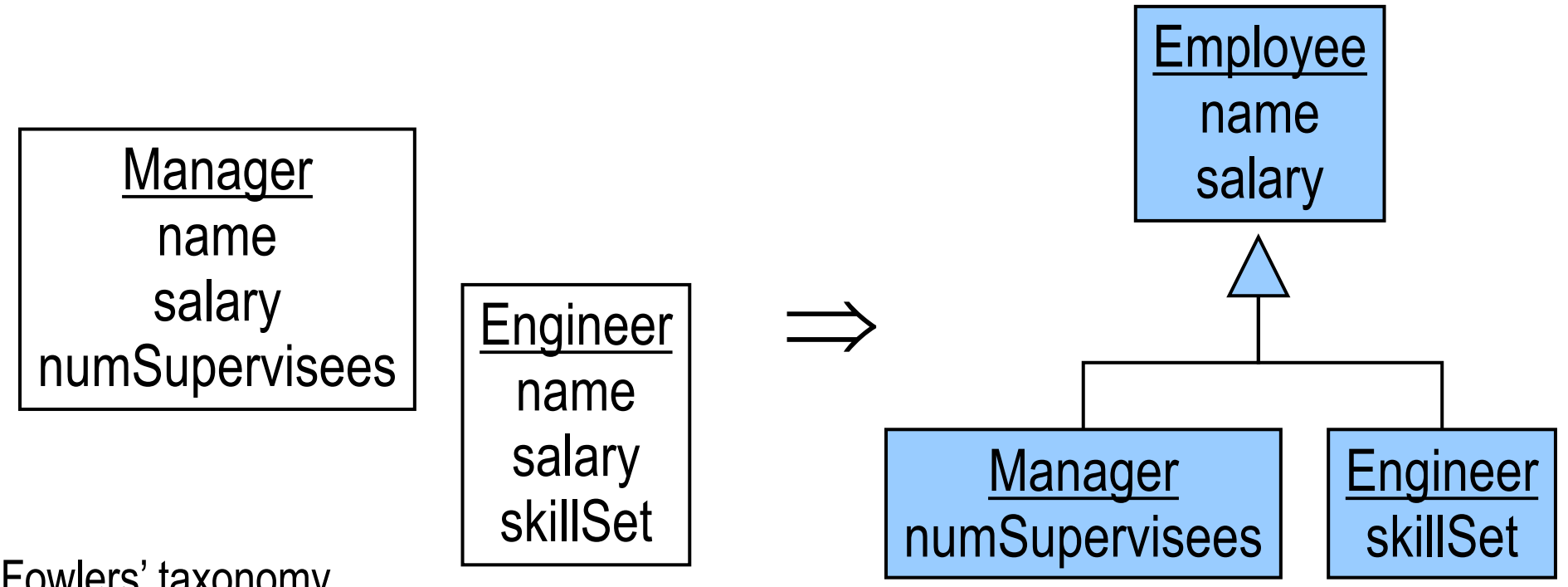


*Dealing with Generalization 2**

Extract Subclass



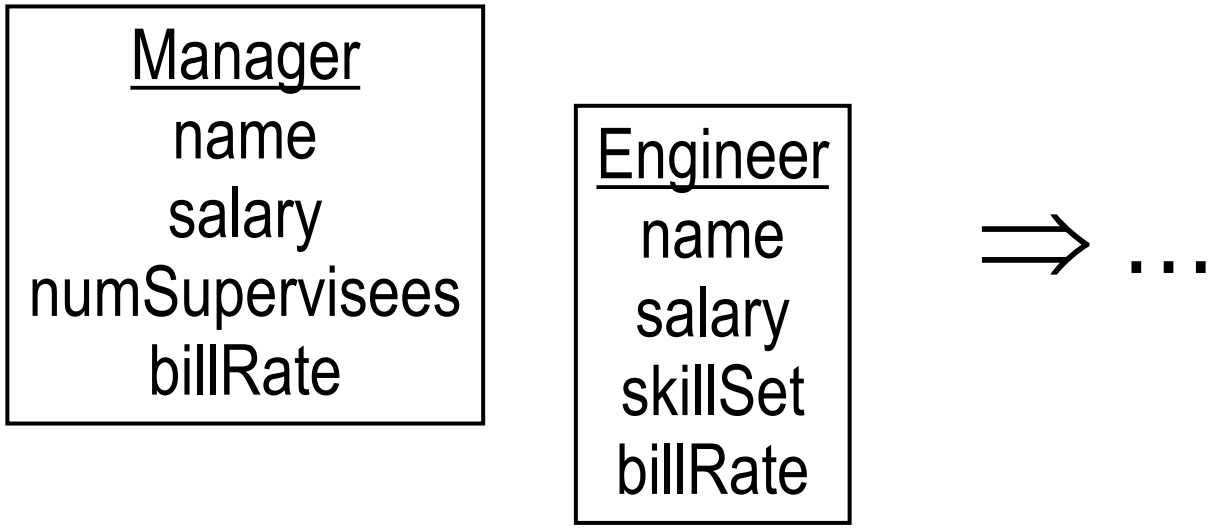
Extract Superclass



*Fowlers' taxonomy

*Dealing with Generalization 3**

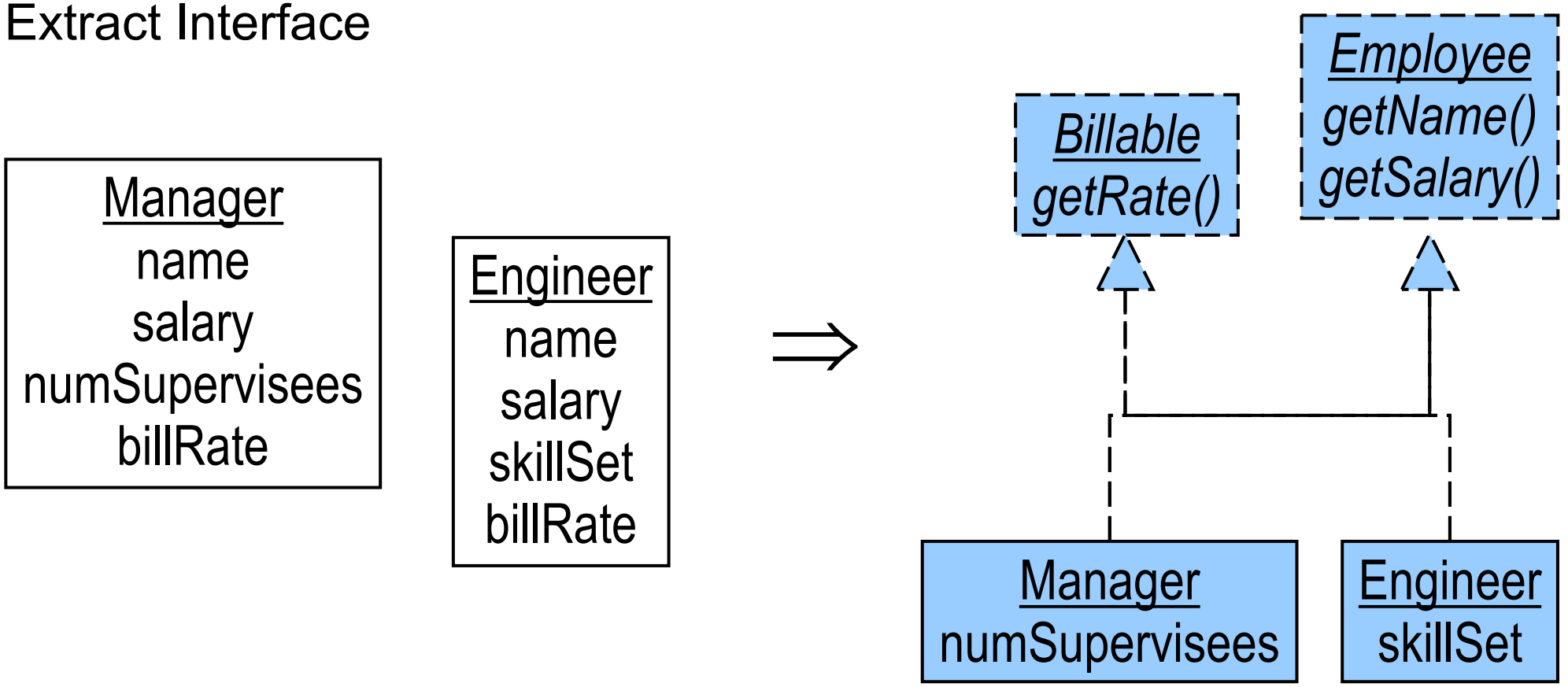
Extract Interface



*Fowlers' taxonomy

*Dealing with Generalization 3**

Extract Interface

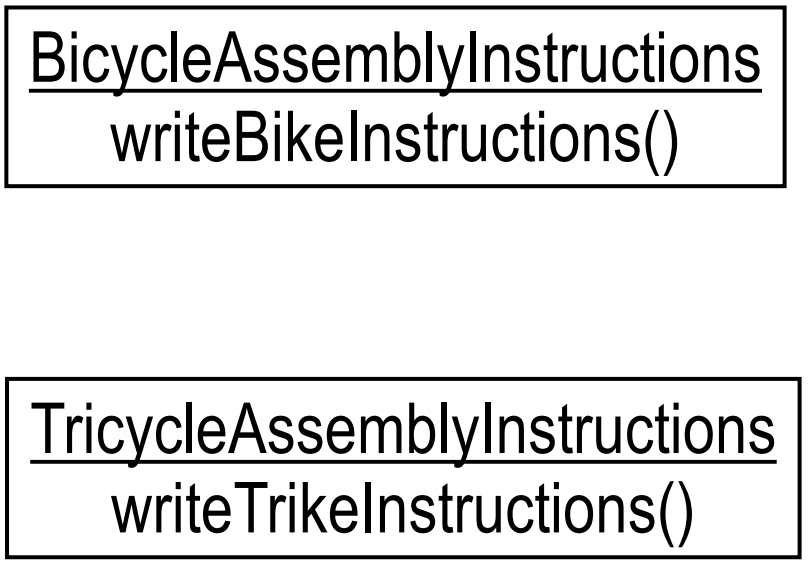


Collapse Hierarchy

Inherited class not special enough

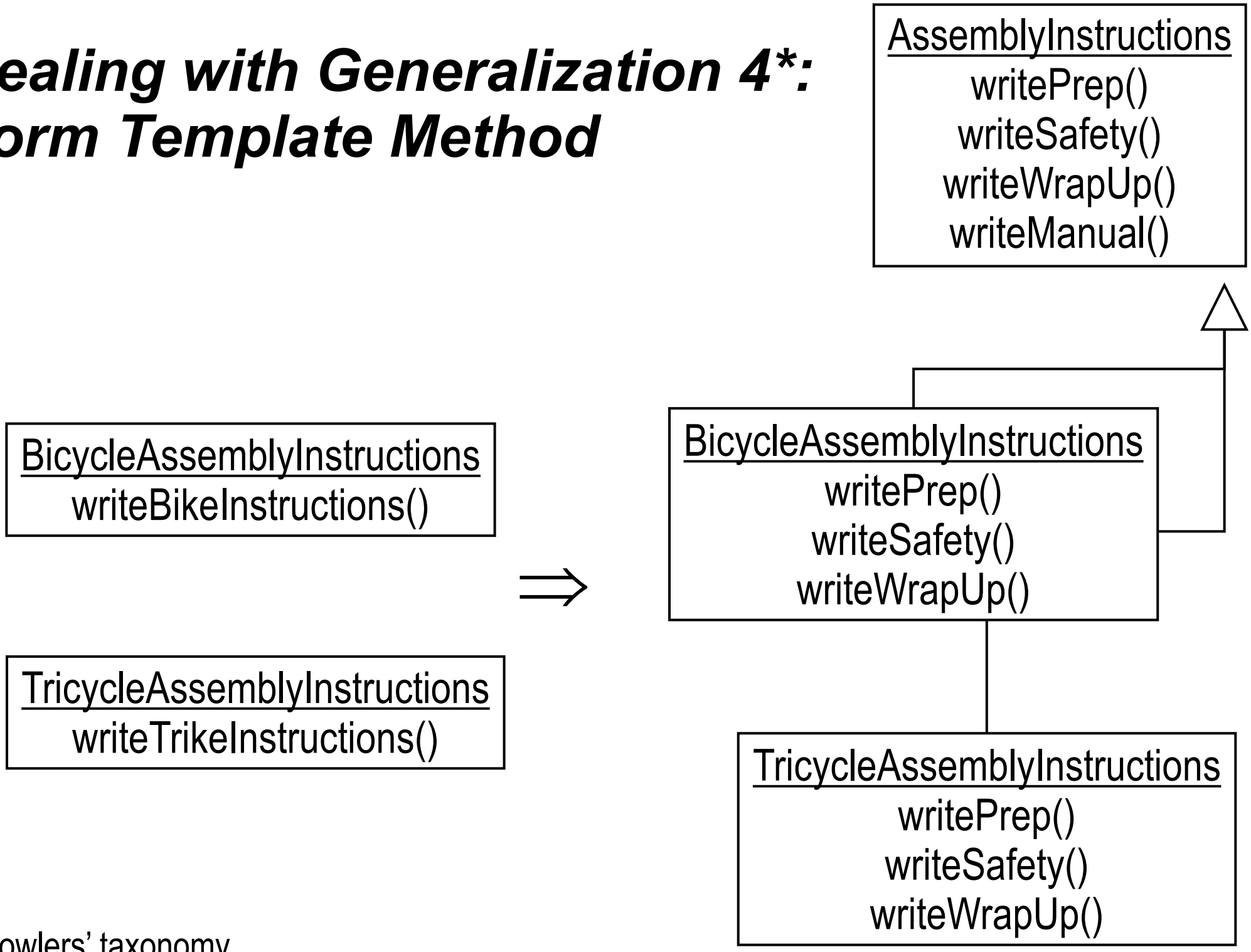
*Fowlers' taxonomy

Dealing with Generalization 4: Form Template Method*



*Fowlers' taxonomy

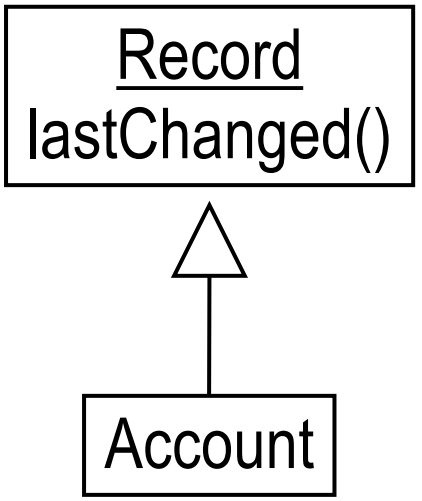
Dealing with Generalization 4: Form Template Method*



*Fowlers' taxonomy

Fowler: *Dealing with Generalization**

Replace Inheritance with Delegation

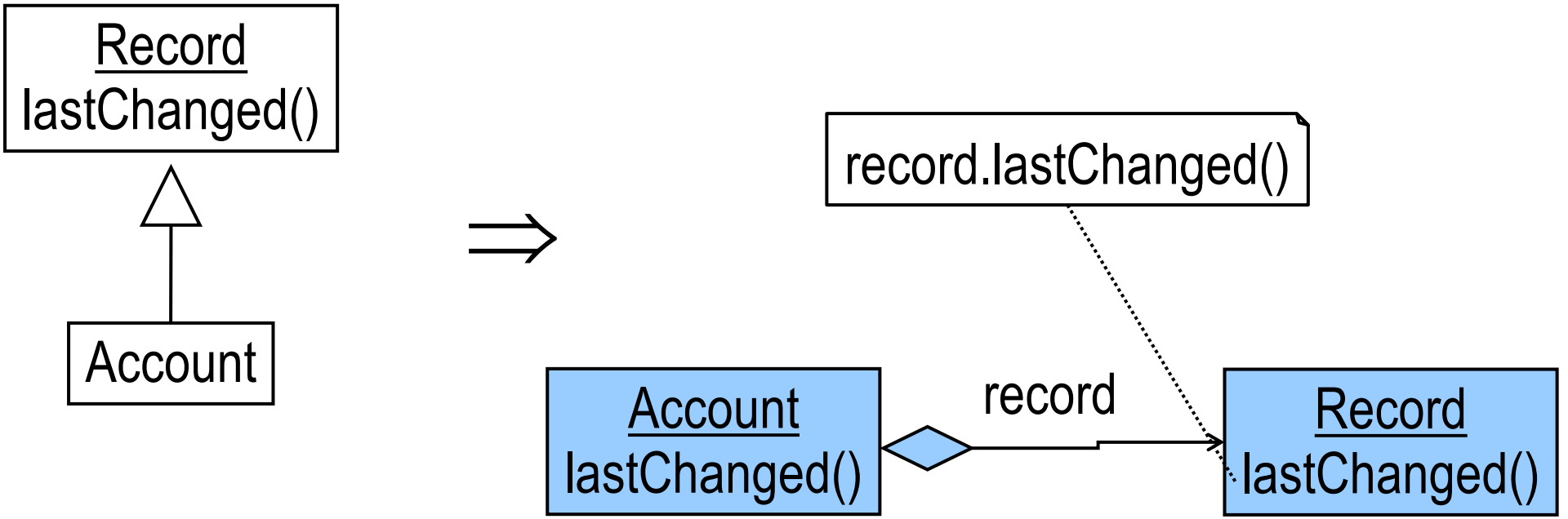


Replace Delegation with Inheritance

*Fowlers' taxonomy

Fowler: *Dealing with Generalization**

Replace Inheritance with Delegation



*Fowlers' taxonomy

Fowler's Refactoring Taxonomy

Why refactoring in design?

Refactoring tooling

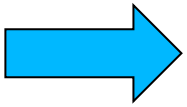
Big Refactoring

Composing Methods

Moving Features Between Objects

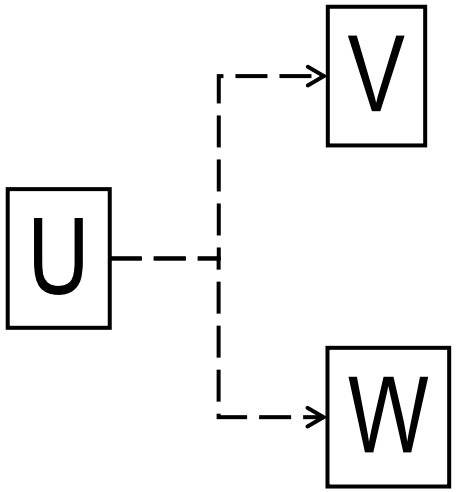
Organizing Data

Dealing With Generalization

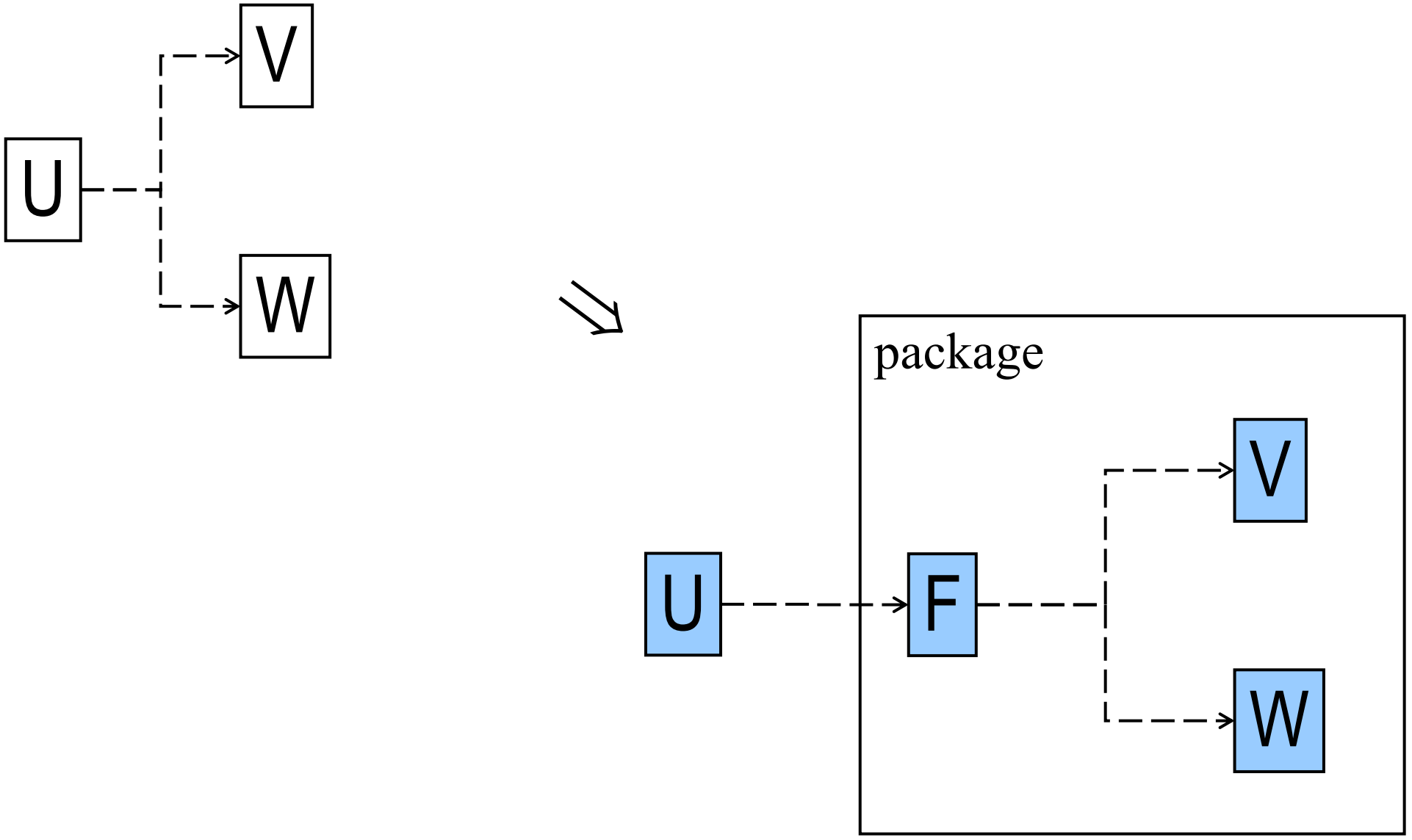


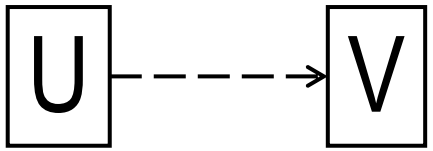
Making Method Calls Simpler

Introducing Façade



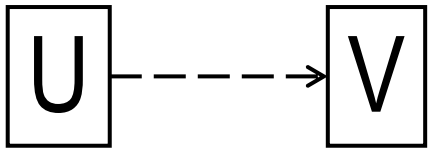
Introducing Façade

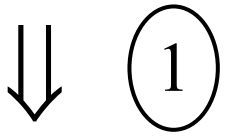


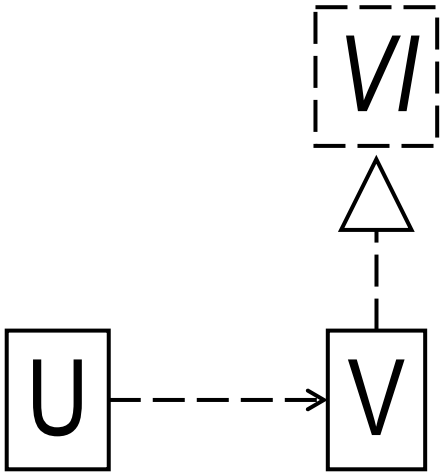


Introduce Module Refactoring

Introduce Module Refactoring





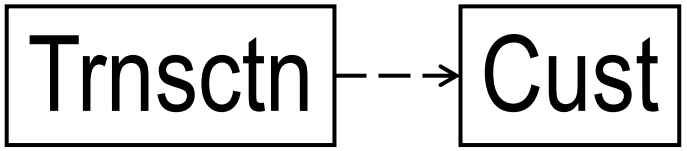


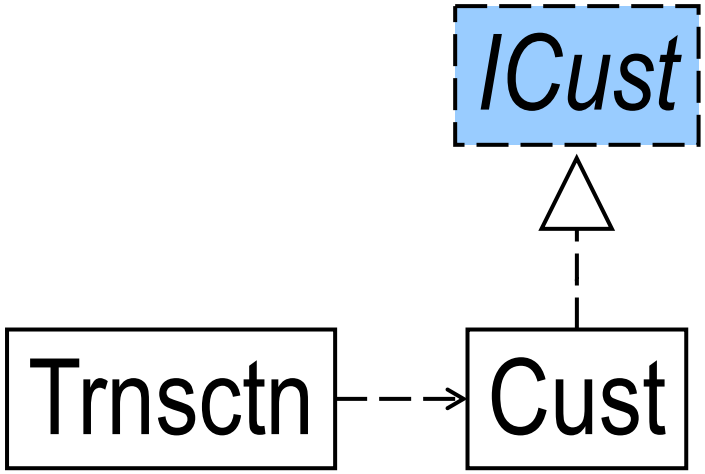
The challenge is that client code needs some degree of knowledge of the module's classes. This is like calling a Dell Computer (module) operator (façade) on the telephone to order a computer.

You need to know general concepts (abstract classes) such as Computer and Monitor. These are outside the "Dell" module.

Thus, abstract base classes have to be introduced (Computer and Monitor are, after all, abstract concepts that all buyers are expected to be familiar with).

Modularize Refactoring Applied *2*: Transaction/Customer





Now, client code such as Transaction can communicate (call methods of) the façade object, and is prepared to get objects of classes that implement the ICust interface.

References

Book:

Martin Fowler et al. Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., 1999 Boston, MA, USA.