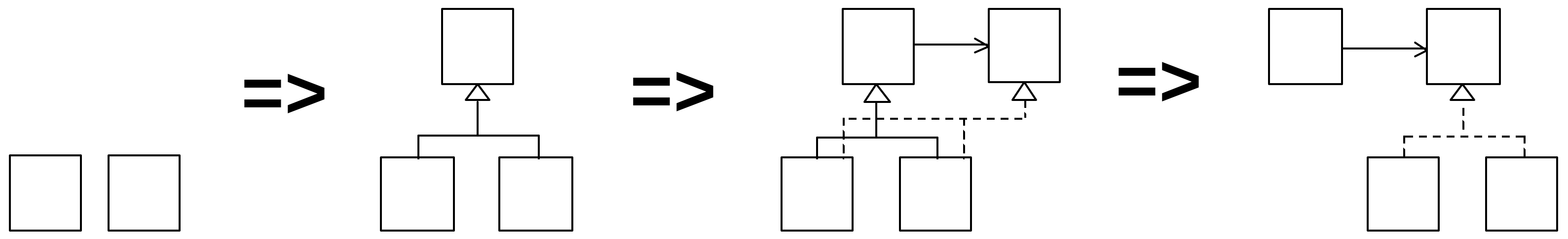
Refactoring Thumbnails

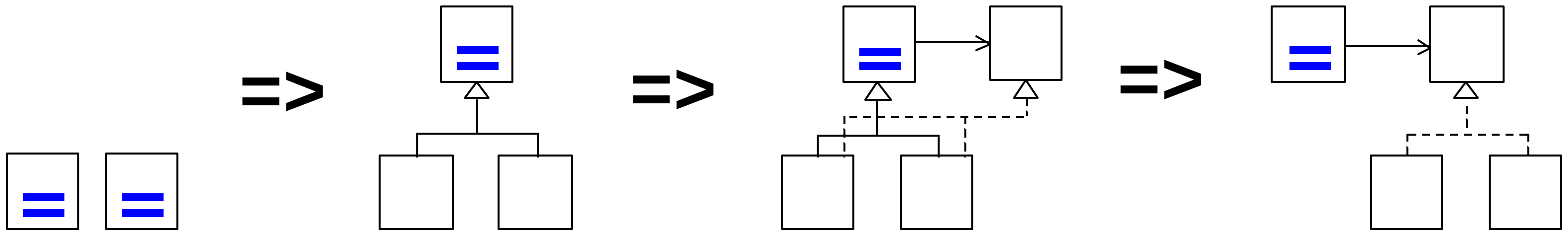
Sven.Gorts@refactoring.be

Refactoring Thumbnails



- express the evolution of a design
- sequence of high-level refactorings
- present intermediate stages

Different Stages



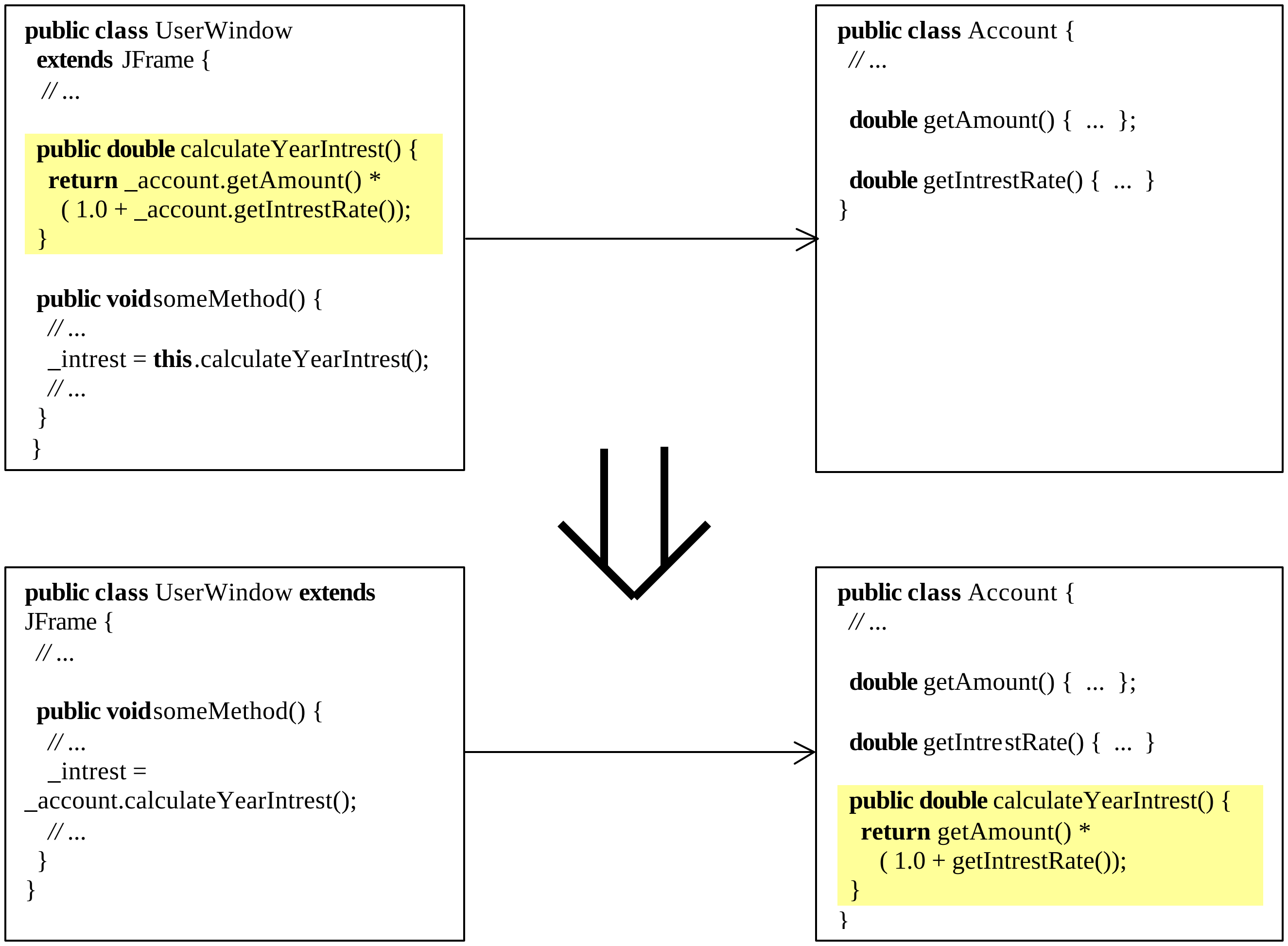
we start with	: copy-paste code
and evolve to	: inheritance-based reuse
passing an	: intermediate stage
and end with	: interface-based solution

Communication

If we want a coherent design to emerge …

- team members need a common vision
  - where evolving design is going
- need express refactoring ideas
  - as high level refactorings
  - without detailed mechanics

Moving a Feature



```
public class UserWindow
  extends JFrame {
   // ...

  public double calculateYearIntrest() {
    return _account.getAmount() *
      ( 1.0 + _account.getIntrestRate());
  }

  public void someMethod() {
    // ...
    _intrest = this.calculateYearIntrest();
    // ...
  }
 }
```

```
public class Account {
  // ...

  double getAmount() {  ...  };

  double getIntrestRate() {  ...  }
}
```

```
public class UserWindow extends
JFrame {
  // ...

  public void someMethod() {
    // ...
    _intrest =
_account.calculateYearIntrest();
    // ...
  }
}
```

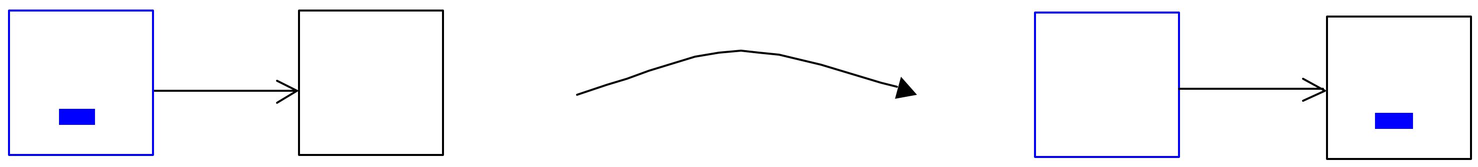
```
public class Account {
  // ...

  double getAmount() {  ...  };

  double getIntrestRate() {  ...  }

  public double calculateYearIntrest() {
    return getAmount() *
      ( 1.0 + getIntrestRate());
  }
}
```

Move Feature



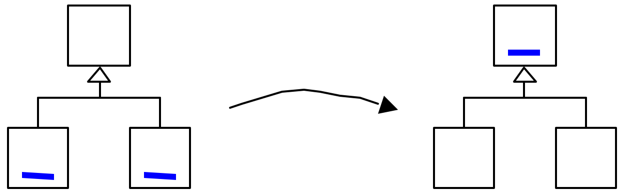
- want to express that the feature is moving *independent of the actual mechanics*
- feature is some behaviour we want to move *a field, a method, an inner class, ...*

Refactoring Thumbnails

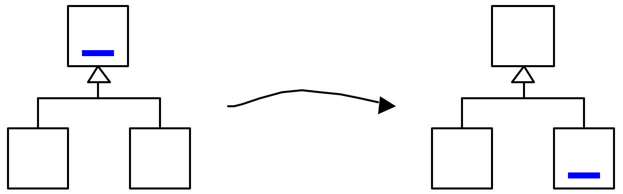
- look like small class diagrams
- use an informal notation
  - easy and fast to sketch
  - avoid UML overhead
- are focused
  - zoom in at the design problem
  - leave out irrelevant details

Thumbnail Primitives

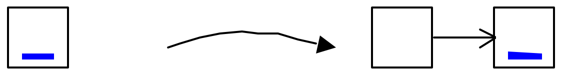
- Pull Up Feature



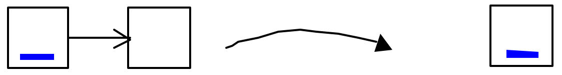
- Push Down Feature



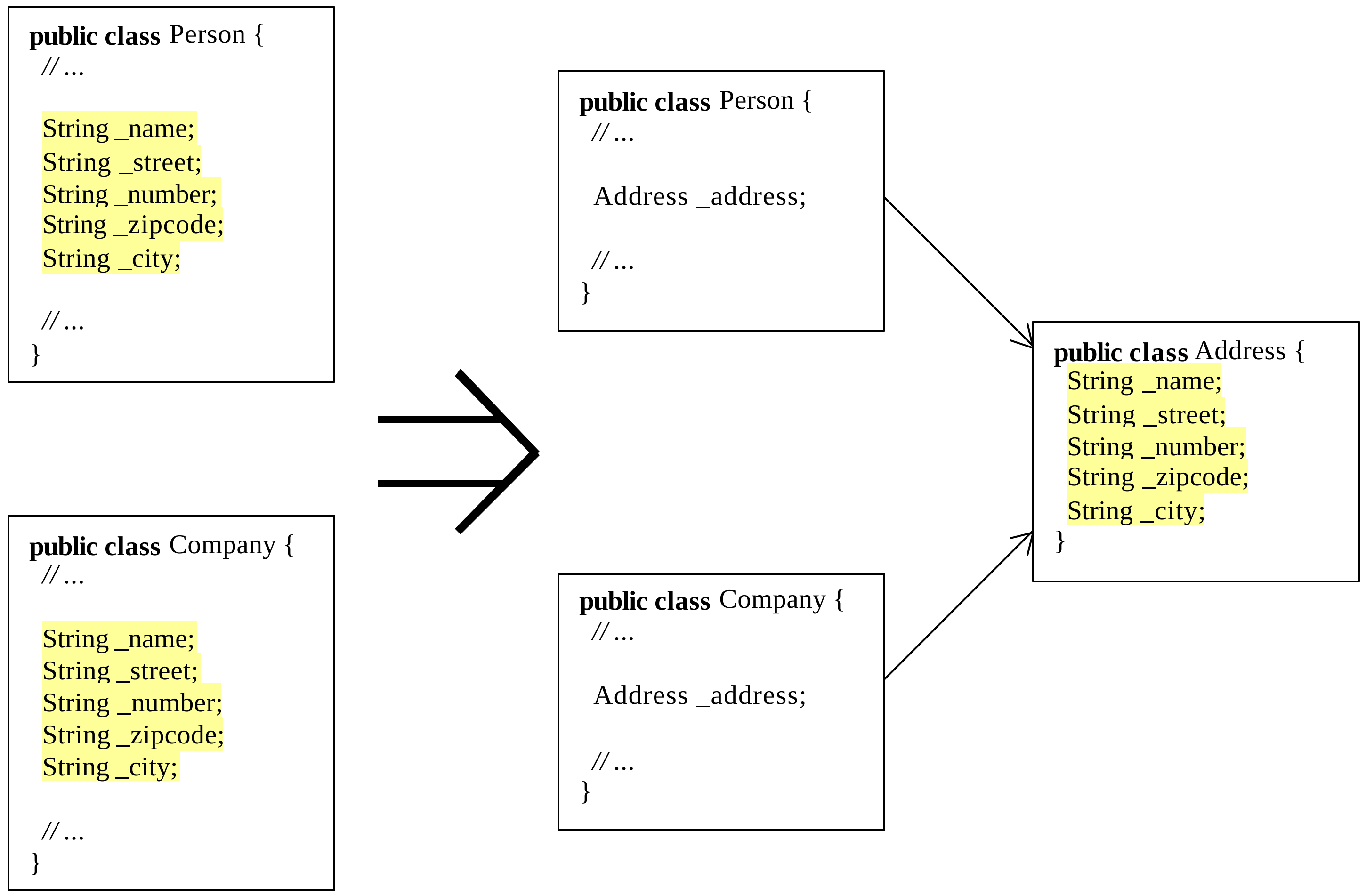
- Extract Feature



- Inline Feature



Eliminating Duplication By Composition



```
public class Person {
 // ...

 String _name;
 String _street;
 String _number;
 String _zipcode;
 String _city;

 // ...
}
```

```
public class Company {
 // ...

 String _name;
 String _street;
 String _number;
 String _zipcode;
 String _city;

 // ...
}
```

```
public class Person {
 // ...

  Address _address;

 // ...
}
```

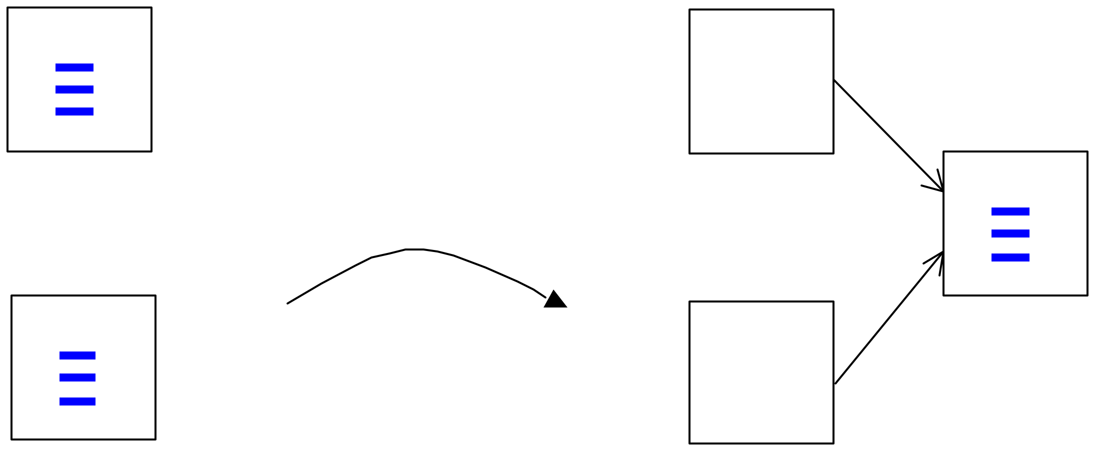
```
public class Company {
 // ...

  Address _address;

 // ...
}
```

```
public class Address {
 String _name;
 String _street;
 String _number;
 String _zipcode;
 String _city;
}
```

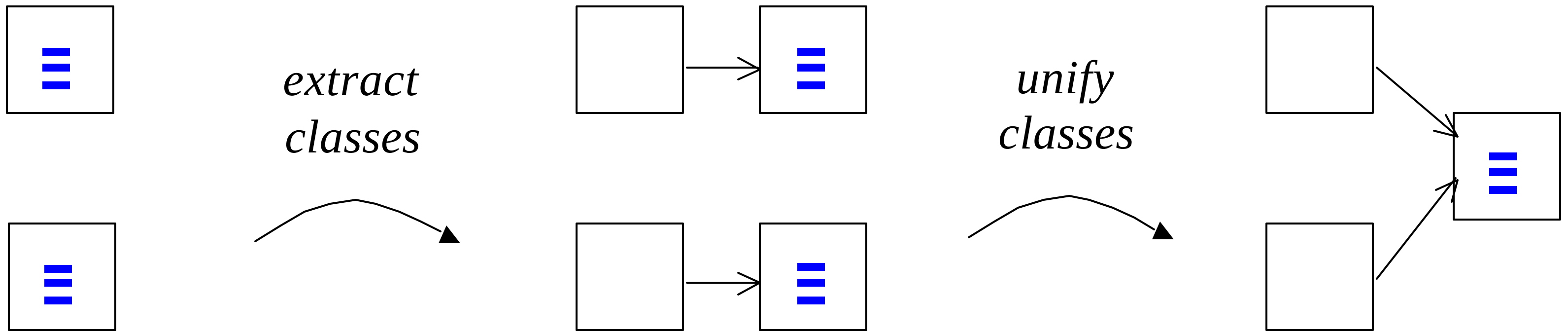
Eliminate Duplication By Composition



Q: How can we perform this refactoring ?

- different routes to choose from

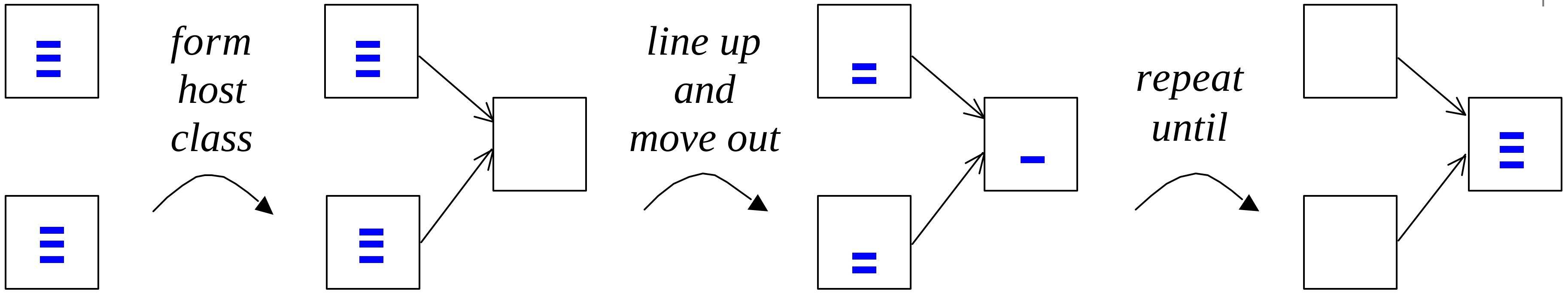
1. Extract and Unify



When: Similar Behaviour but Different Code

- Extract Similarities into Different Classes
- Make Duplication Explicit
- Eliminate The Duplication

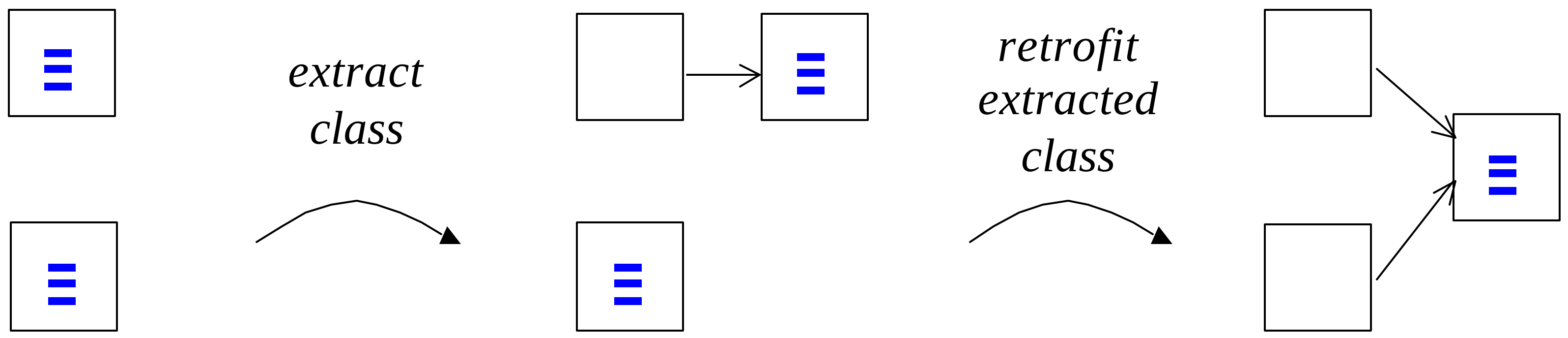
2. Gradual Extract and Unify



When: Unsure About Missing Concept

- Create Empty Class To Host Commonalities
- Feature by Feature -> Line Up and Unify
- Repeat - Until Duplication is Eliminated

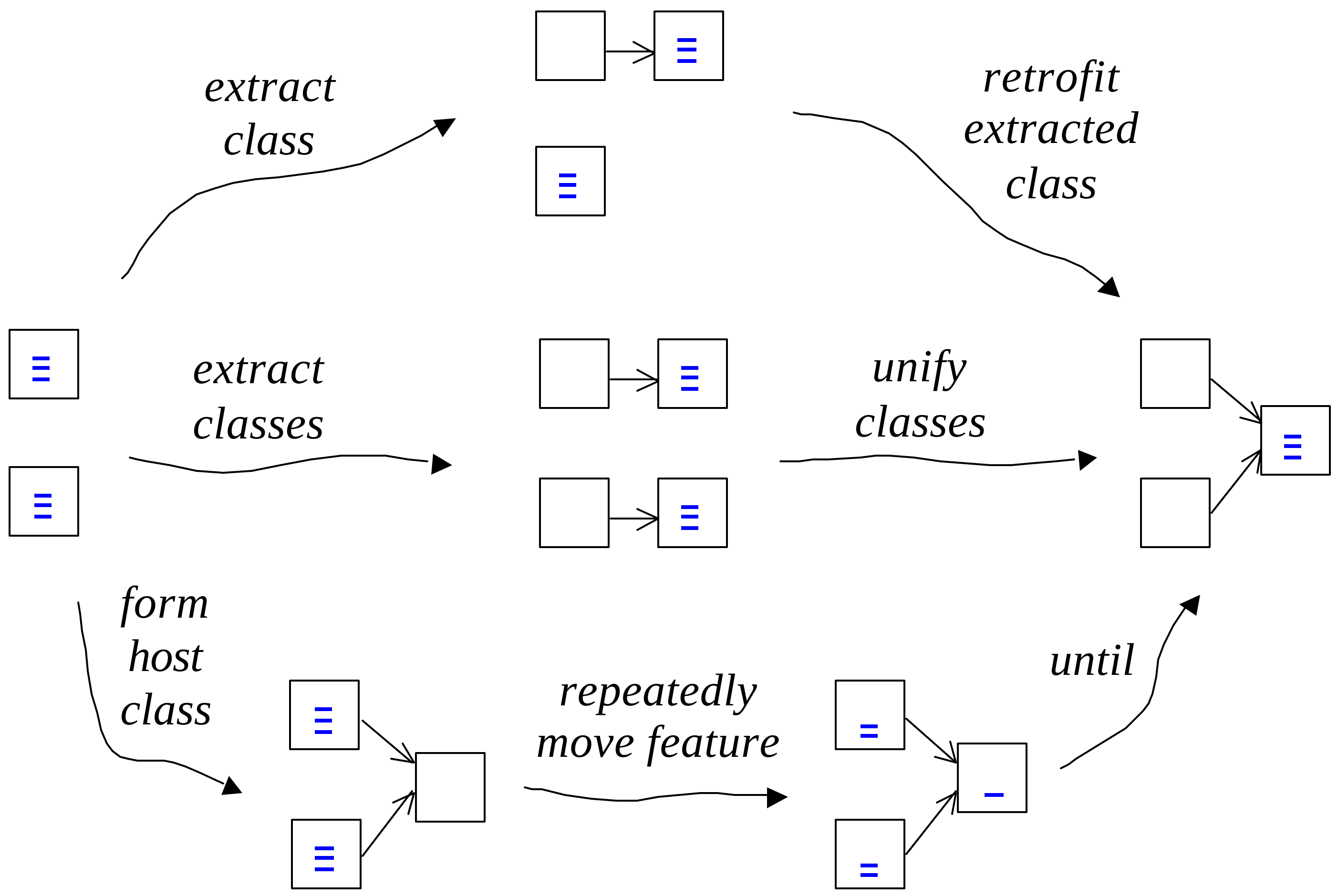
3. Extract and Retrofit



When: Widespread Code Duplication

- Extract Best Fit For Missing Abstraction
- Polish To A Really Neat Abstraction
- Retrofit New Abstraction To Other Classes

Evolution Chart



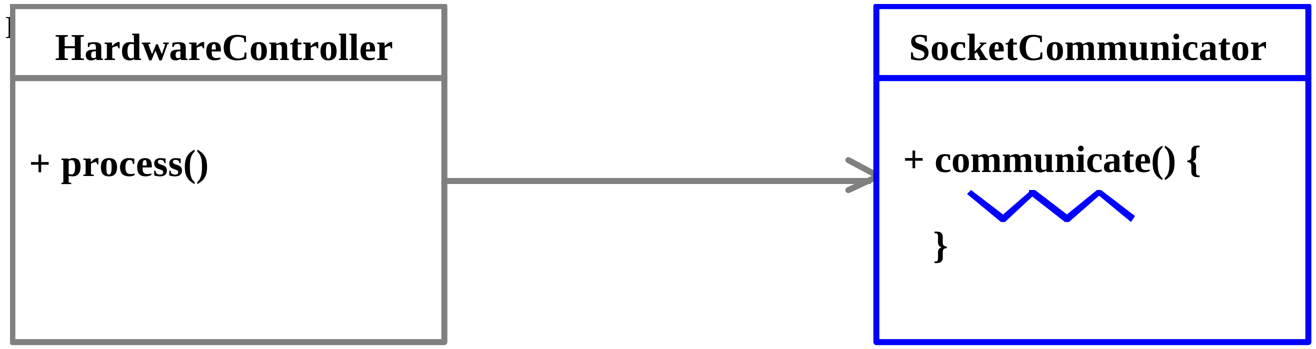
Refactoring Knowledge

- Sketching refactoring thumbnails
  - similar sequences of drawings recur
  - we may use thumbnails to capture refactoring knowledge
- Towards a *pattern language*
  - build a catalogue of thumbnail refactorings
  - document strategies for large refactorings

Evolving to Decorator

- Discuss Toy Example
  - Represents Initial Design
- Choose Our Target
  - Decorator Pattern
- Discuss Some Alternative Routes

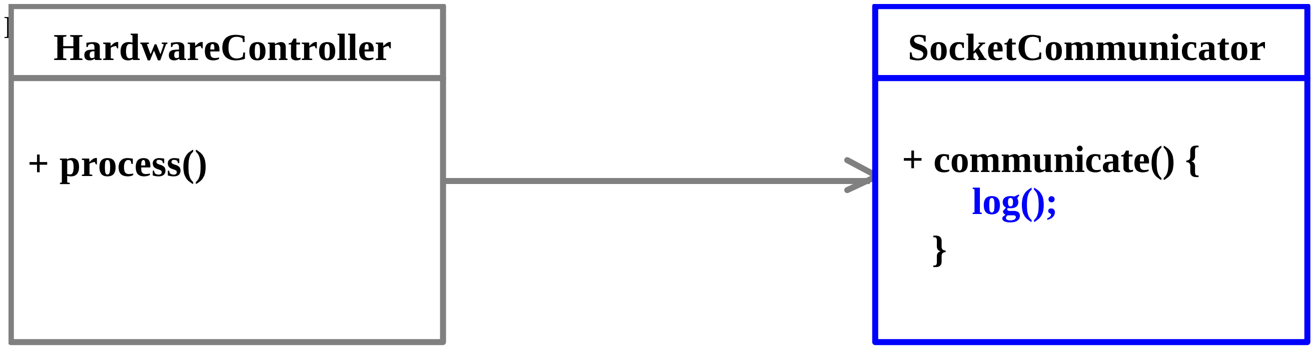
Client: HardwareController



```
public void process() {
   ...
   response = _communicator.communicate("incr cntr2;");
   if (response.equals("cntr2 overflow")) {
      ...
   }

   ...
}
```

Server: SocketCommunicator



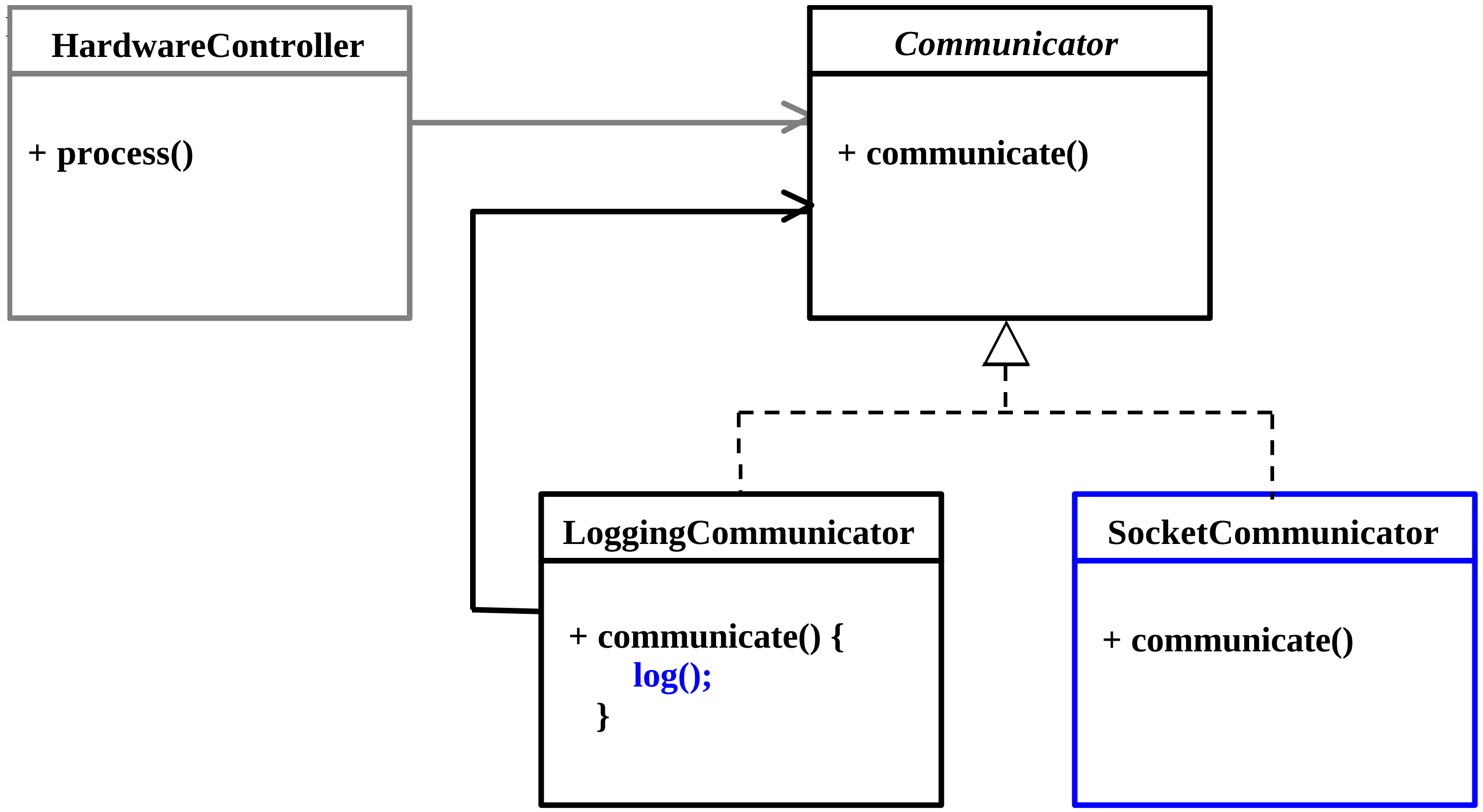
```
public String communicate(String command) {

   log("command [" + command  + "]");
   sendMessage(command);

   // block till response received
   String response = receiveMessage();
   log("response [" + response + "]");

   return response;
 }
```

Target: Decorator Pattern



Participants

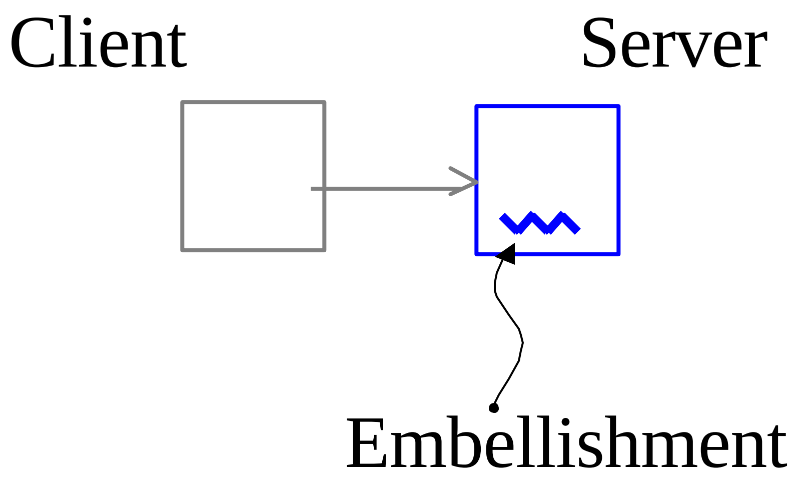
Client

Server

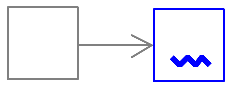
- Primary responsibility

Embellishment

- Secondary responsibility

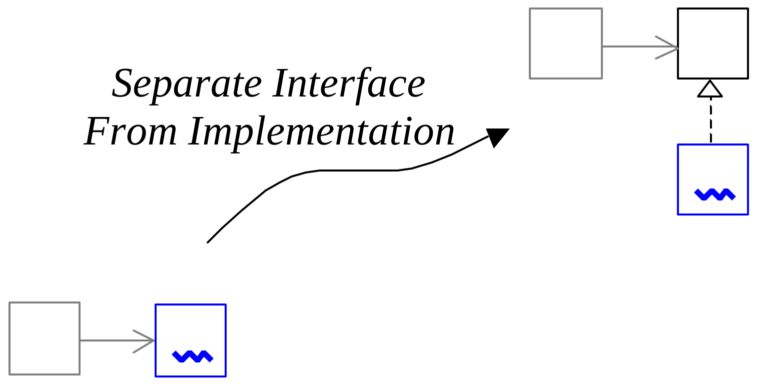


version 0.1.1



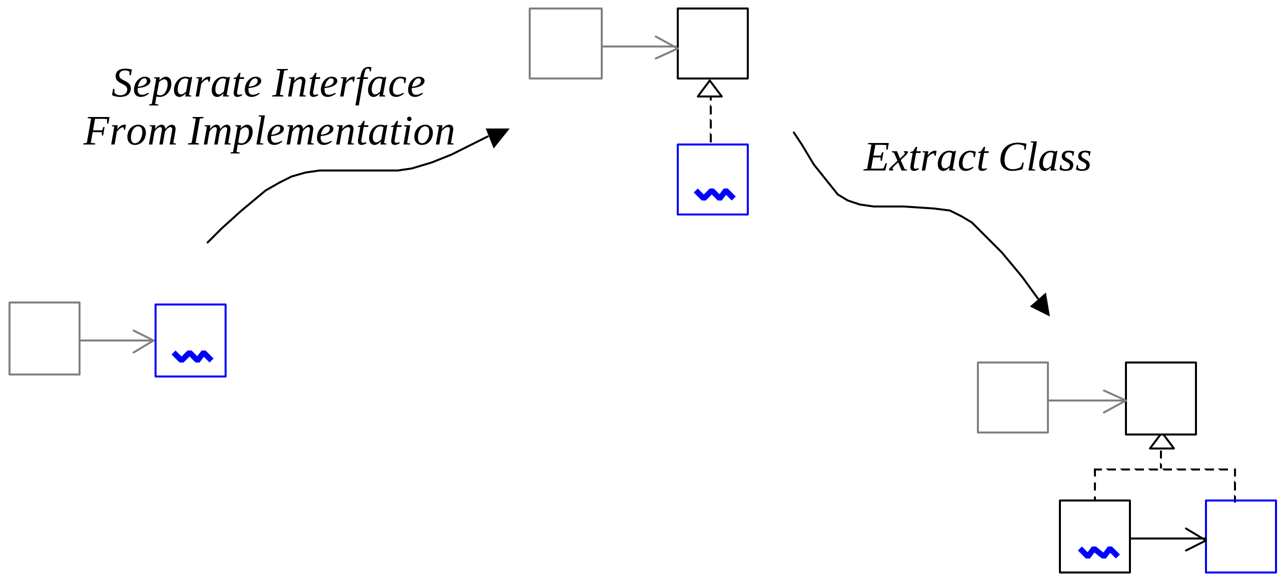
Q: Who would start by introducing an interface ?

version 0.1.2



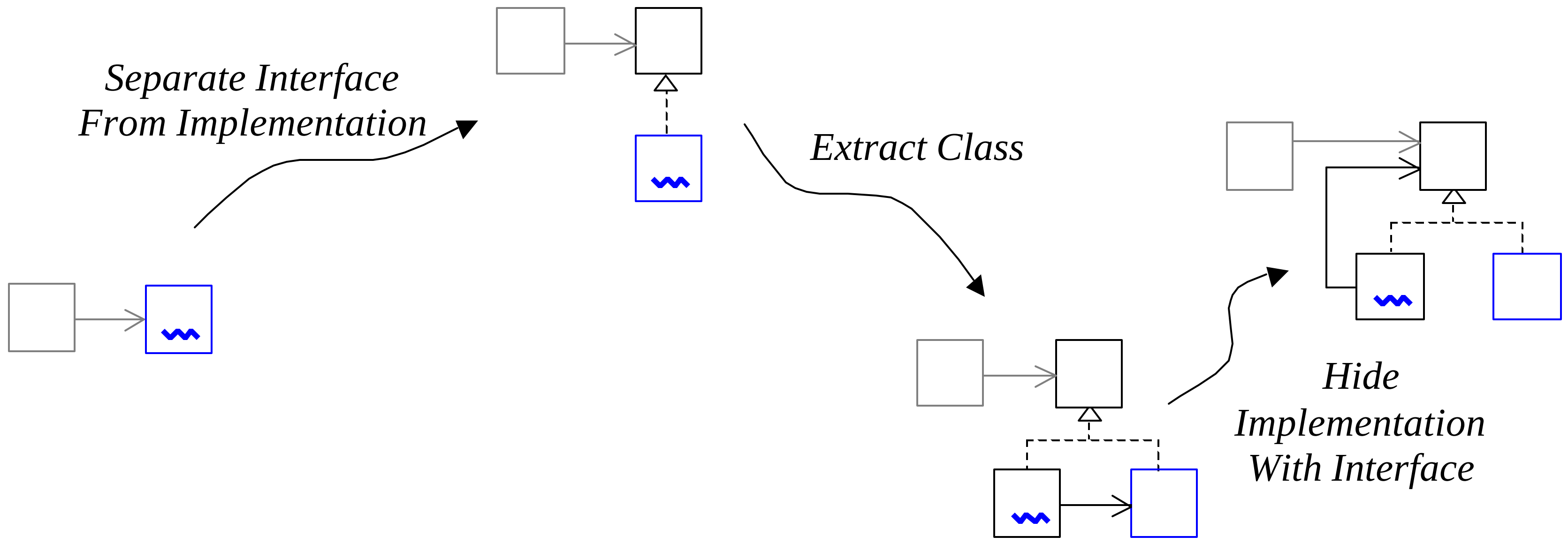
Client now separated from implementation

version 0.1.3



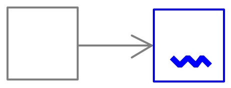
Decoupling Embellishment from Server

version 0.1.4



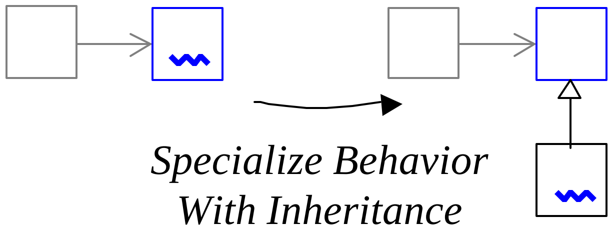
Generalize Decorating Class

version 0.2.1



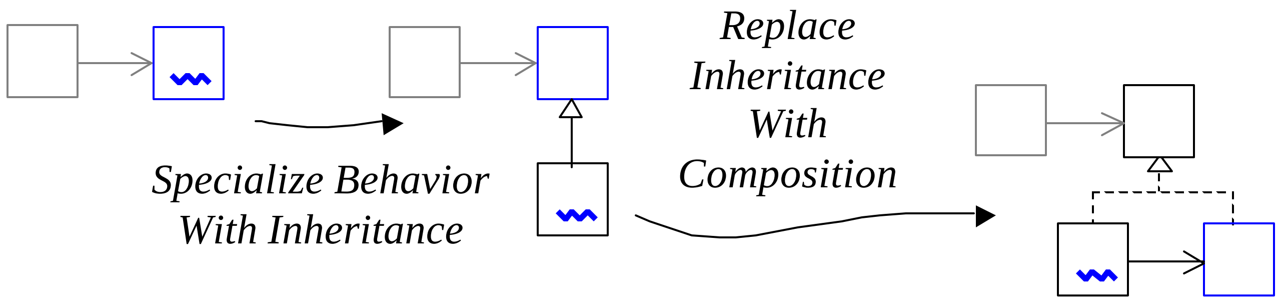
Q: Who would start by introducing a subclass ?

version 0.2.2



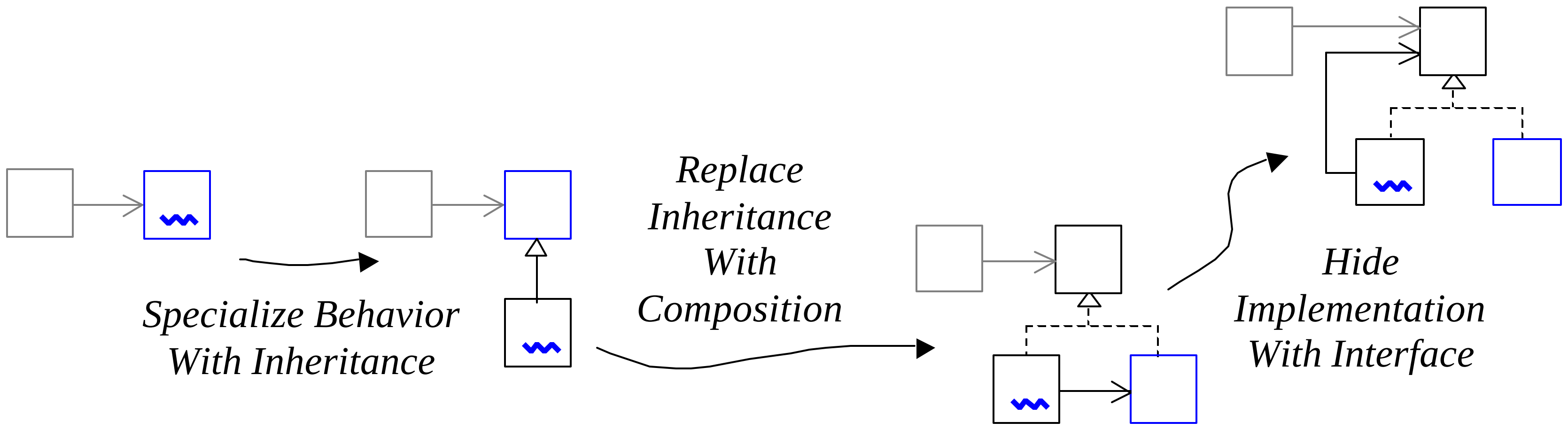
Separated Embellishment from Server

version 0.2.3


Specialize Behavior With Inheritance
Replace Inheritance With Composition

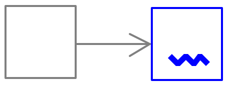
With the implicit interfacing role of the Server class made explicit the Embellishment class no longer needs to subclass the Server class

version 0.2.4


Specialize Behavior With Inheritance
Replace Inheritance With Composition
Hide Implementation With Interface

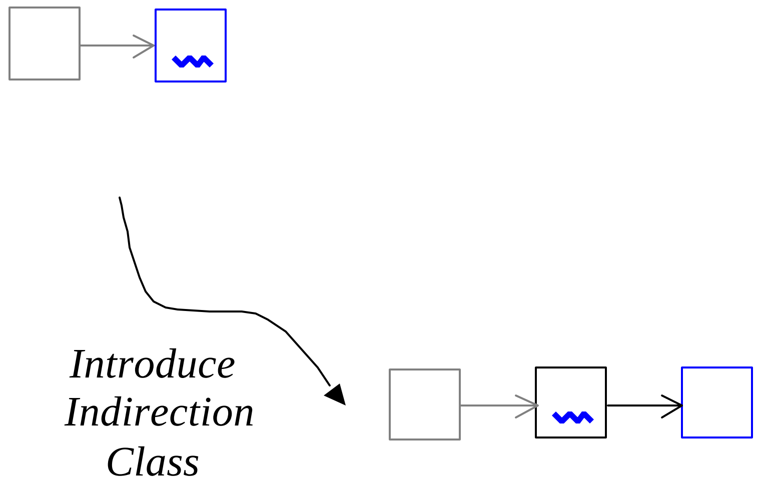
And again we can generalize

version 0.3.1



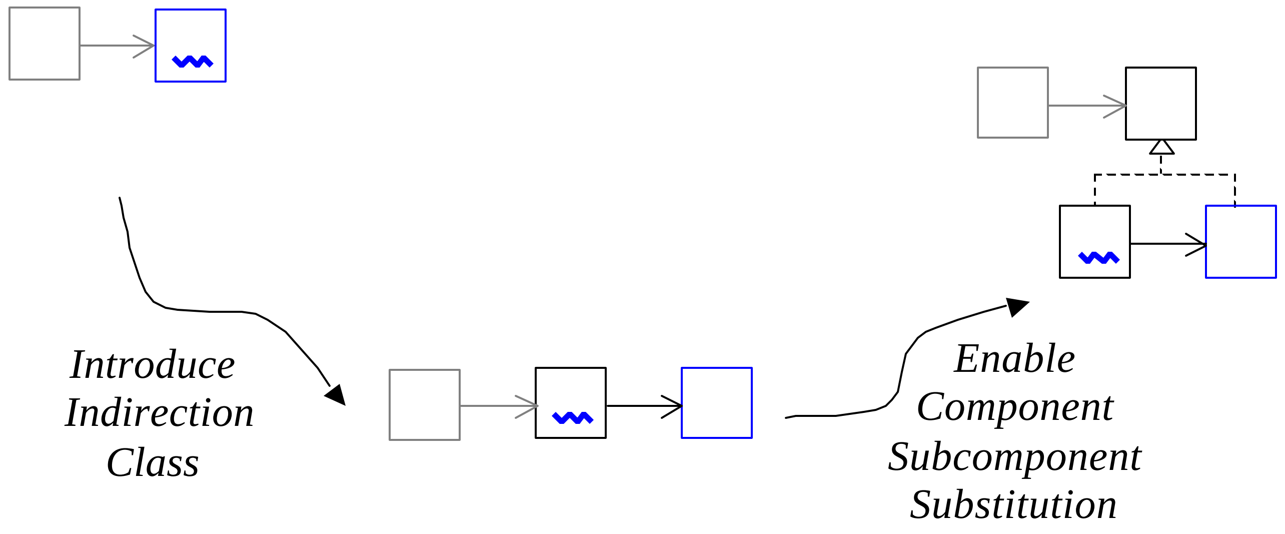
Q: Who would start by extracting a class ?

version 0.3.2



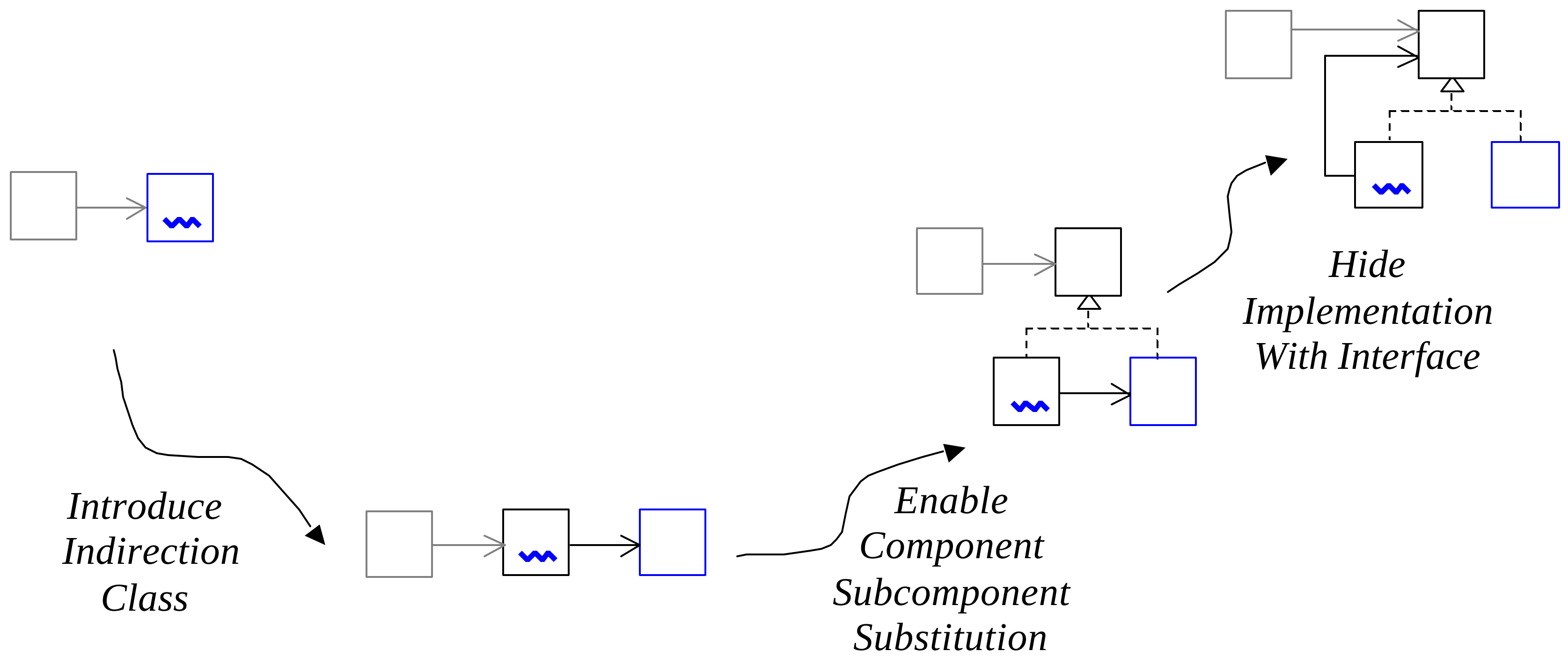
Separated Embellishment class from Server class

version 0.3.3


Introduce Indirection Class
Enable Component Subcomponent Substitution

Client can now talk directly to both

version 0.3.4



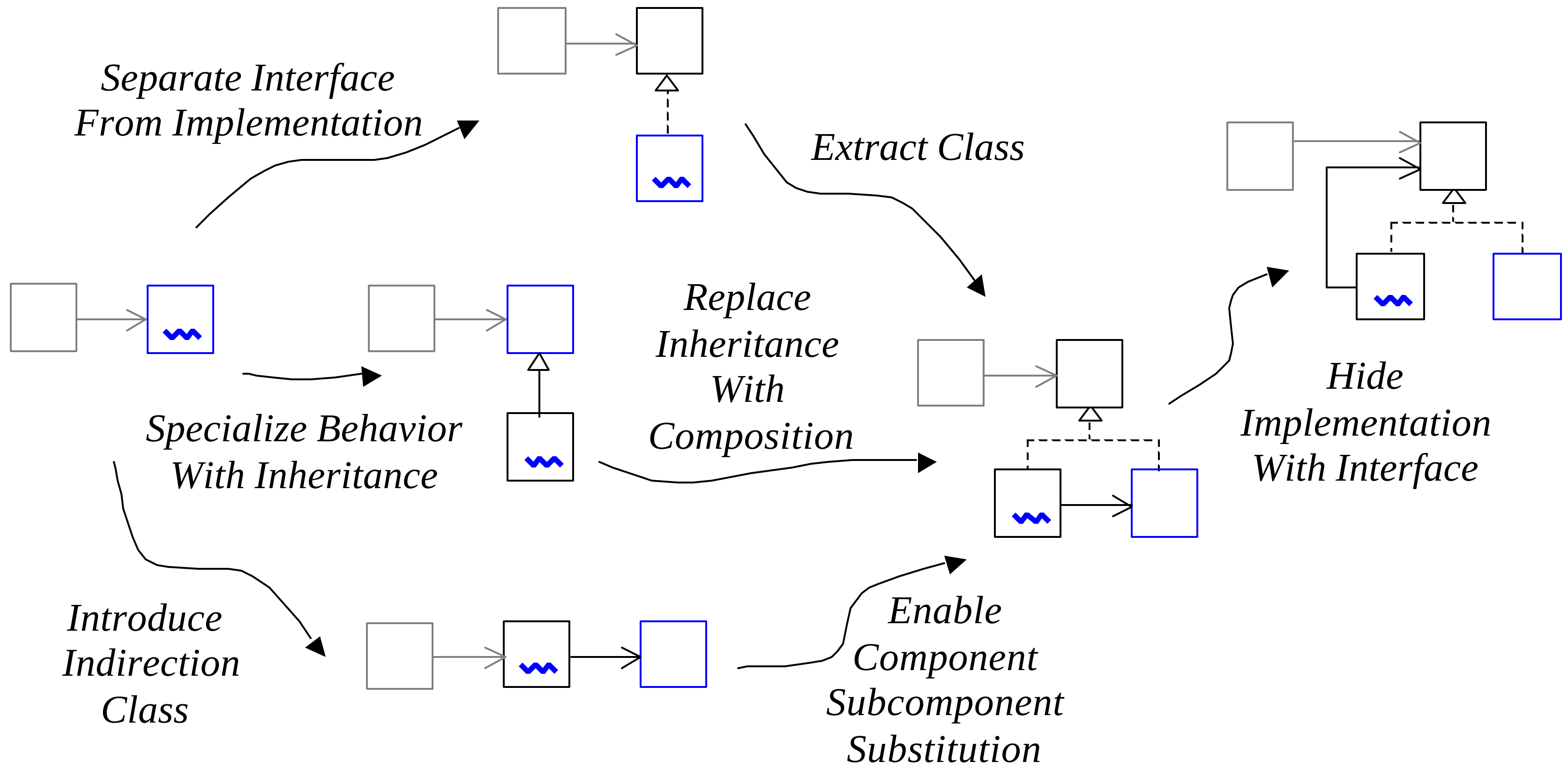
Decorator can now decorate any implementation

Conclusion

Refactoring Thumbnails:

- Describe High Level Refactoring Ideas
- Using Simple Graphical Notation
- Present Alternative Routes Visually
  - Informed Decisions

Discussion



Separate Interface From Implementation
Extract Class
Replace Inheritance With Composition
Hide Implementation With Interface
Specialize Behavior With Inheritance
Introduce Indirection Class
Enable Component Subcomponent Substitution

For More About Thumbnails

At the website:

<www.refactoring.be>

Contact me:

Sven.Gorts@refactoring.be

References

- Refactoring Workbook
  (William C.Wake)
- Refactoring To Patterns
  (Joshua Kerievsky)
- Working Effectively With Legacy Code
  (Michael Feathers)