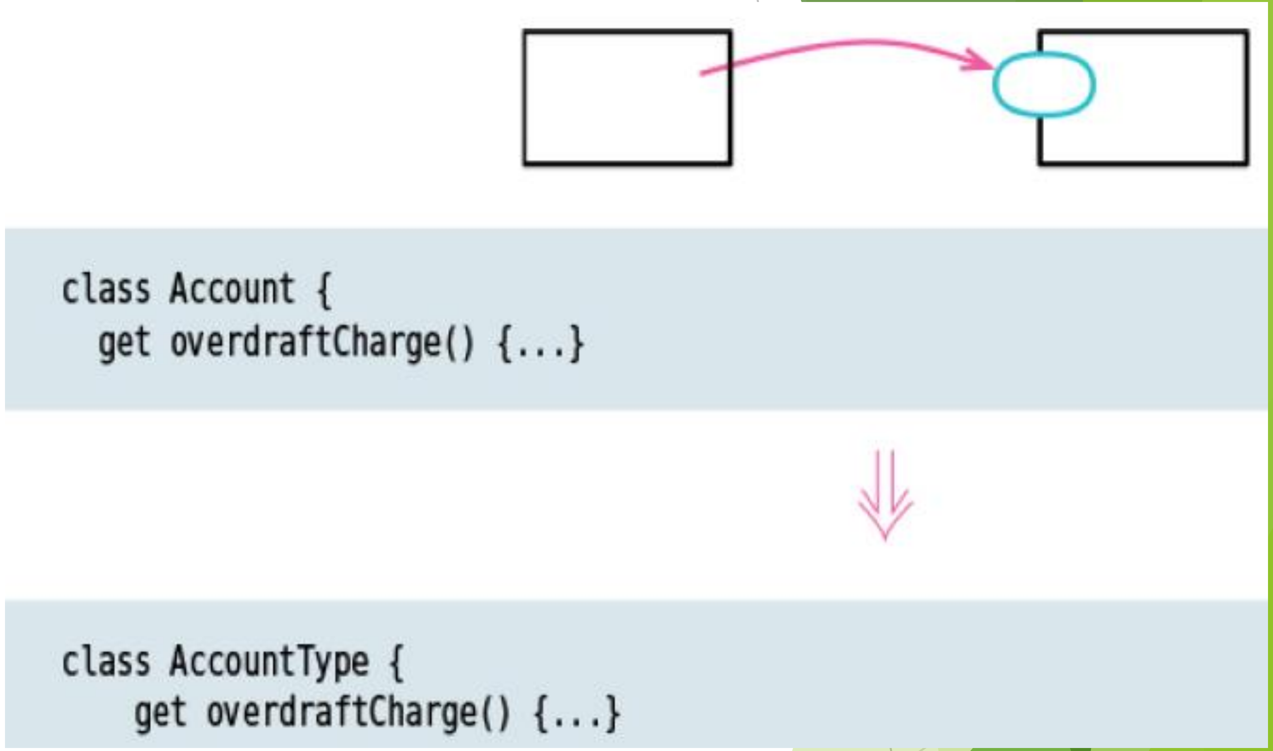
Design Defects & Restructuring

Week 12: 19/21 Nov 2022

Rahim Hasnani

Moving Features: Move Function

- Examine all the program elements used by the chosen function in its current context. Consider whether they should move too.
- Check if the chosen function is a polymorphic method.
- Copy the function to the target context. Adjust it to fit in its new home.
- Perform static analysis.
- Figure out how to reference the target function from the source context.
- Turn the source function into a delegating function.
- Test.
- Consider *Inline Function* on the source function.

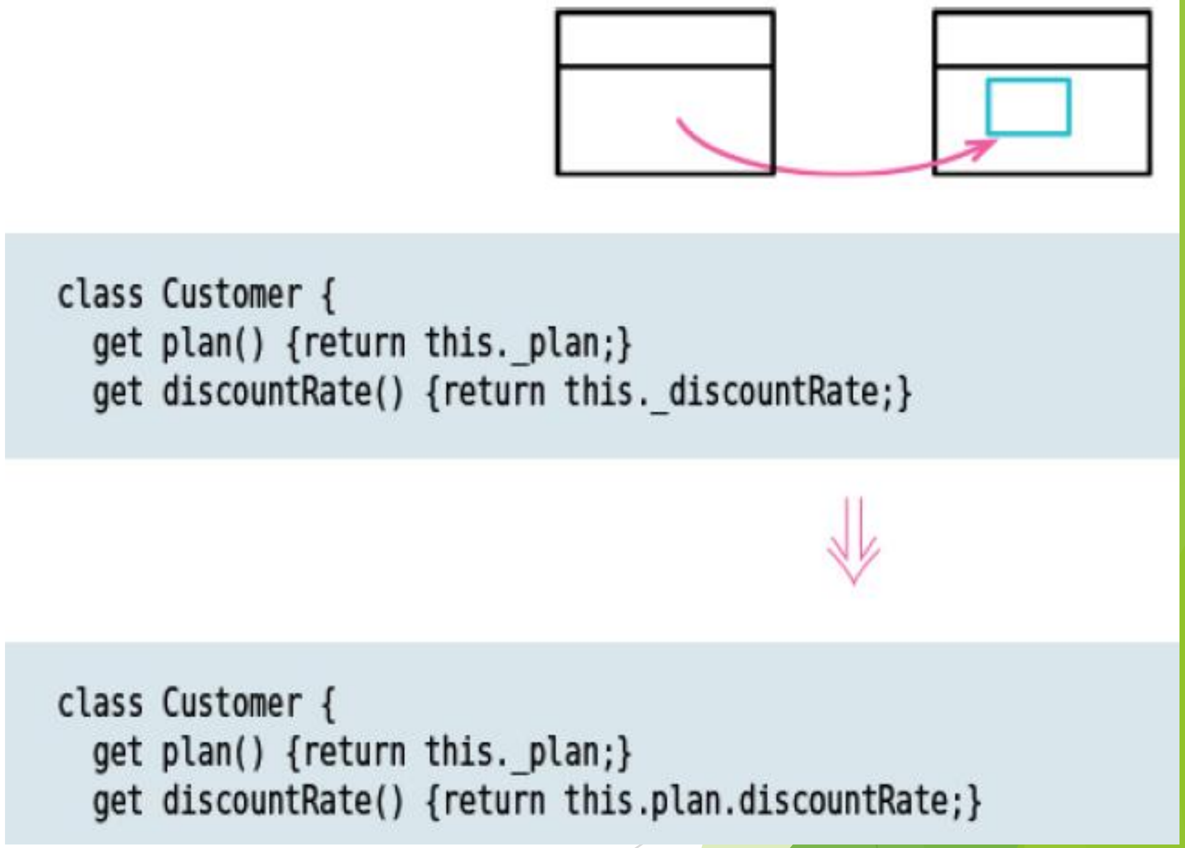


```
class Account {
  get overdraftCharge() {...}
```

```
class AccountType {
    get overdraftCharge() {...}
```

Moving Features: Move Field

- Ensure the source field is encapsulated.
- Test.
- Create a field (and accessors) in the target.
- Run static checks.
- Ensure there is a reference from the source object to the target object.
- Adjust accessors to use the target field.
- Test.
- Remove the source field.
- Test.

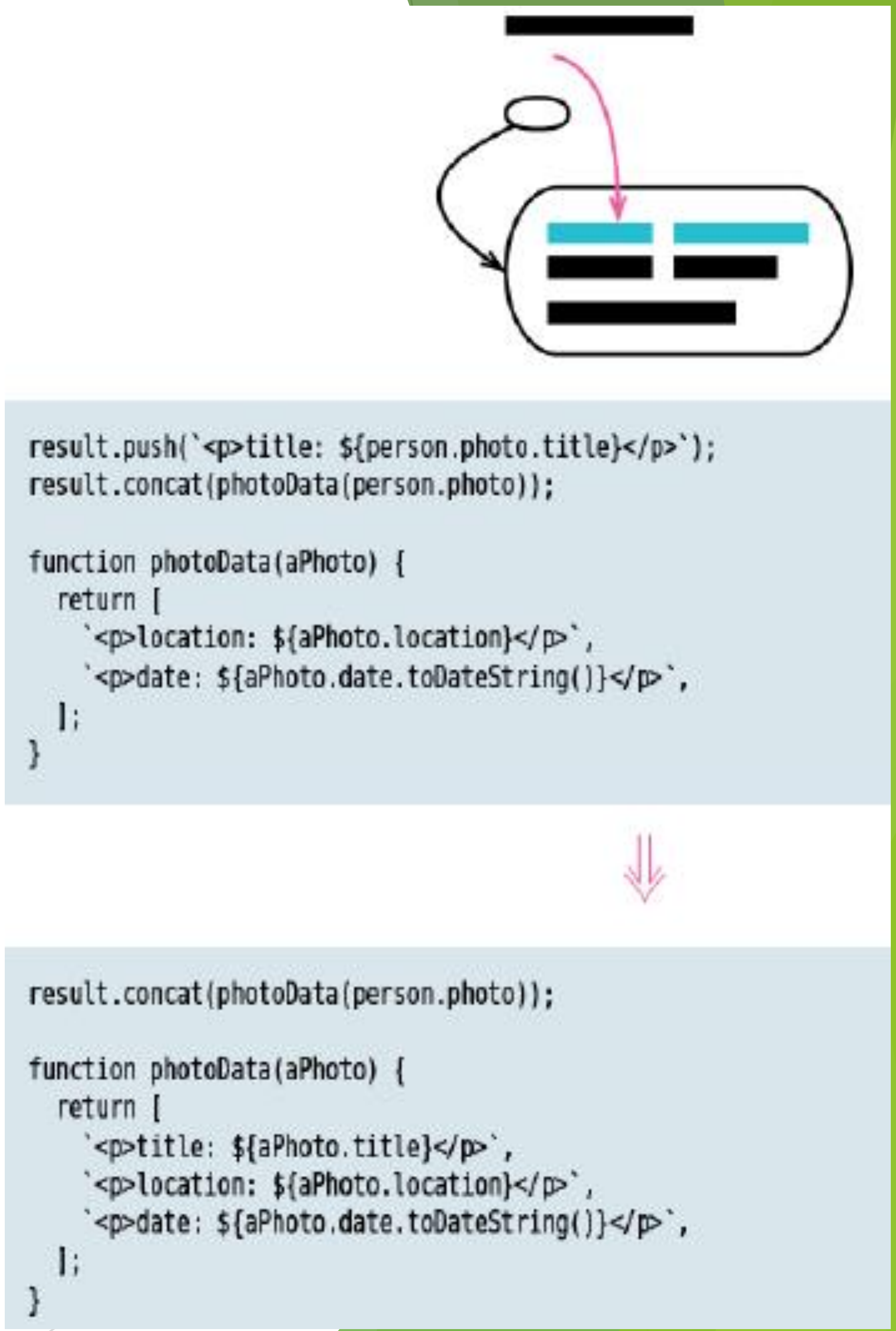


```
class Customer {
  get plan() {return this._plan;}
  get discountRate() {return this._discountRate;}
```

```
class Customer {
  get plan() {return this._plan;}
  get discountRate() {return this.plan.discountRate;}
```

Moving Features: Move Statements Into Functions

- If the repetitive code isn’t adjacent to the call of the target function, use *Slide Statements* to get it adjacent.
- If the target function is only called by the source function, just cut the code from the source, paste it into the target, test, and ignore the rest of these mechanics.
- If you have more callers, use *Extract Function* on one of the call sites to extract both the call to the target function and the statements you wish to move into it. Give it a name that’s transient, but easy to grep.
- Convert every other call to use the new function. Test after each conversion.
- When all the original calls use the new function, use *Inline Function (* to inline the original function completely into the new function, removing the original function.
- *Rename Function* to change the name of the new function to the same name as the original function. Or to a better name, if there is one.



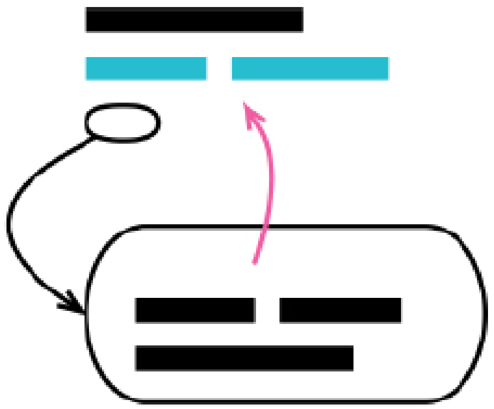
```
result.push(`<p>title: ${person.photo.title}</p>`);
result.concat(photoData(person.photo));

function photoData(aPhoto) {
  return [
    `<p>location: ${aPhoto.location}</p>`,
    `<p>date: ${aPhoto.date.toDateString()}</p>`,
  ];
}
```

```
result.concat(photoData(person.photo));

function photoData(aPhoto) {
  return [
    `<p>title: ${aPhoto.title}</p>`,
    `<p>location: ${aPhoto.location}</p>`,
    `<p>date: ${aPhoto.date.toDateString()}</p>`,
  ];
}
```

Moving Features: Move Statements to Callers

- In simple circumstances, where you have only one or two callers and a simple function to call from, just cut the first line from the called function and paste (and perhaps fit) it into the callers. Test and you're done.
- Otherwise, apply *Extract Function* to all the statements that you *don't* wish to move; give it a temporary but easily searchable name.

  If the function is a method that is overridden by subclasses, do the extraction on all of them so that the remaining method is identical in all classes. Then remove the subclass methods.
- Use *Inline Function* on the original function.
- Apply *Change Function Declaration* on the extracted function to rename it to the original name. Or to a better name, if you can think of one.

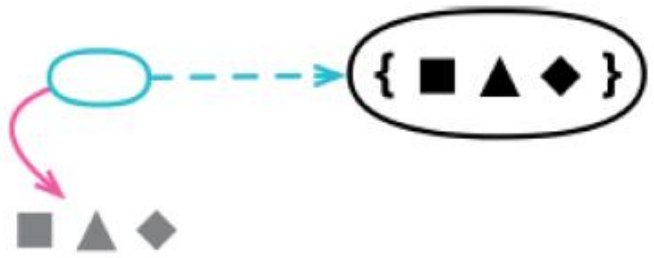


```
emitPhotoData(outStream, person.photo);

function emitPhotoData(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}
```

⇓

```
emitPhotoData(outStream, person.photo);
outStream.write(`<p>location: ${person.photo.location}</p>\n`);

function emitPhotoData(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
}
```

Moving Features: Replace Inline Code with Function Call



```
let appliesToMass = false;
for(const s of states) {
  if (s === "MA") appliesToMass = true;
}
```

⇓

```
appliesToMass = states.includes("MA");
```

- Replace the inline code with a call to the existing function.
- Test.

Moving Features: Slide Statements

- Identify the target position to move the fragment to. Examine statements between source and target to see if there is interference for the candidate fragment. Abandon action if there is any interference.

  A fragment cannot slide backwards earlier than any element it references is declared.

  A fragment cannot slide forwards beyond any element that references it.

  A fragment cannot slide over any statement that modifies an element it references.

  A fragment that modifies an element cannot slide over any other element that references the modified element.

- Cut the fragment from the source and paste into the target position.
- Test.

```
const pricingPlan = retrievePricingPlan();
const order = retreiveOrder();
let charge;
const chargePerUnit = pricingPlan.unit;
```

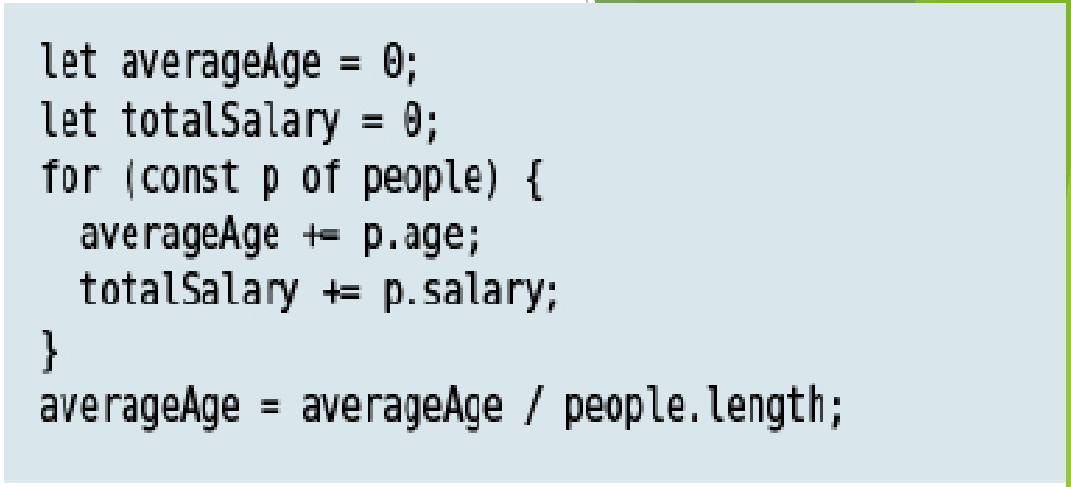
$\Downarrow$

```
const pricingPlan = retrievePricingPlan();
const chargePerUnit = pricingPlan.unit;
const order = retreiveOrder();
let charge;
```

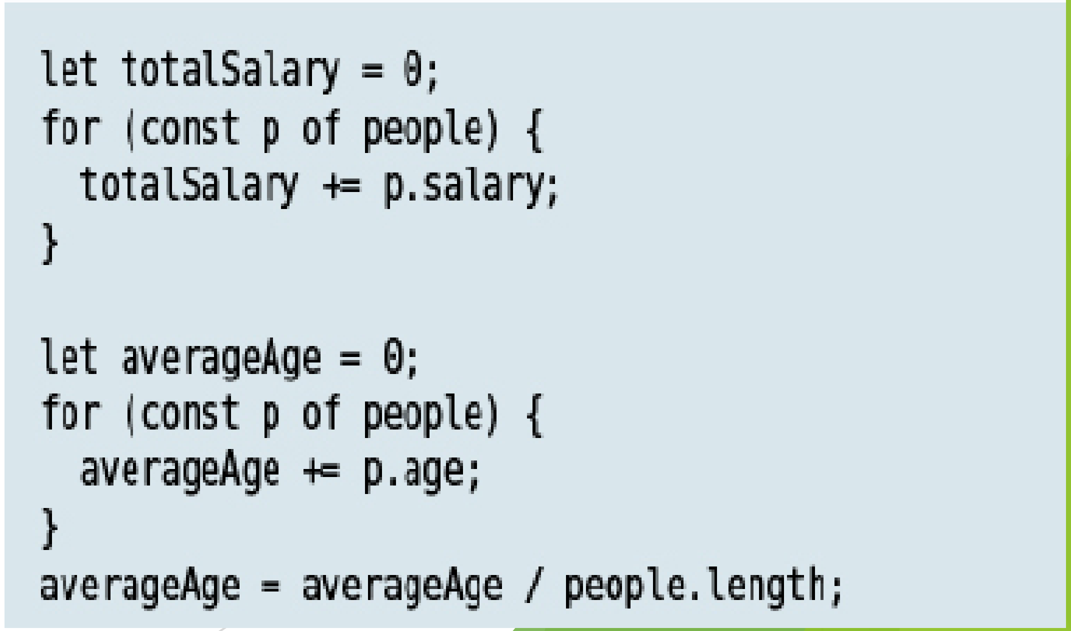
Moving Features: Split Loop

- Copy the loop.
- Identify and eliminate duplicate side effects.
- Test.

When done, consider *Extract Function* on each loop.



```
let averageAge = 0;
let totalSalary = 0;
for (const p of people) {
  averageAge += p.age;
  totalSalary += p.salary;
}
averageAge = averageAge / people.length;
```



```
let totalSalary = 0;
for (const p of people) {
  totalSalary += p.salary;
}

let averageAge = 0;
for (const p of people) {
  averageAge += p.age;
}
averageAge = averageAge / people.length;
```

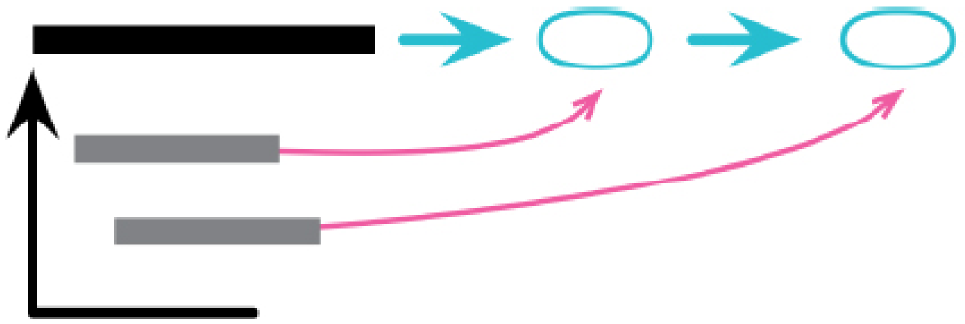
Moving Features: Replace Loop with Pipeline

- Create a new variable for the loop's collection.

  This may be a simple copy of an existing variable.

- Starting at the top, take each bit of behavior in the loop and replace it with a collection pipeline operation in the derivation of the loop collection variable. Test after each change.
- Once all behavior is removed from the loop, remove it.

  If it assigns to an accumulator, assign the pipeline result to the accumulator.



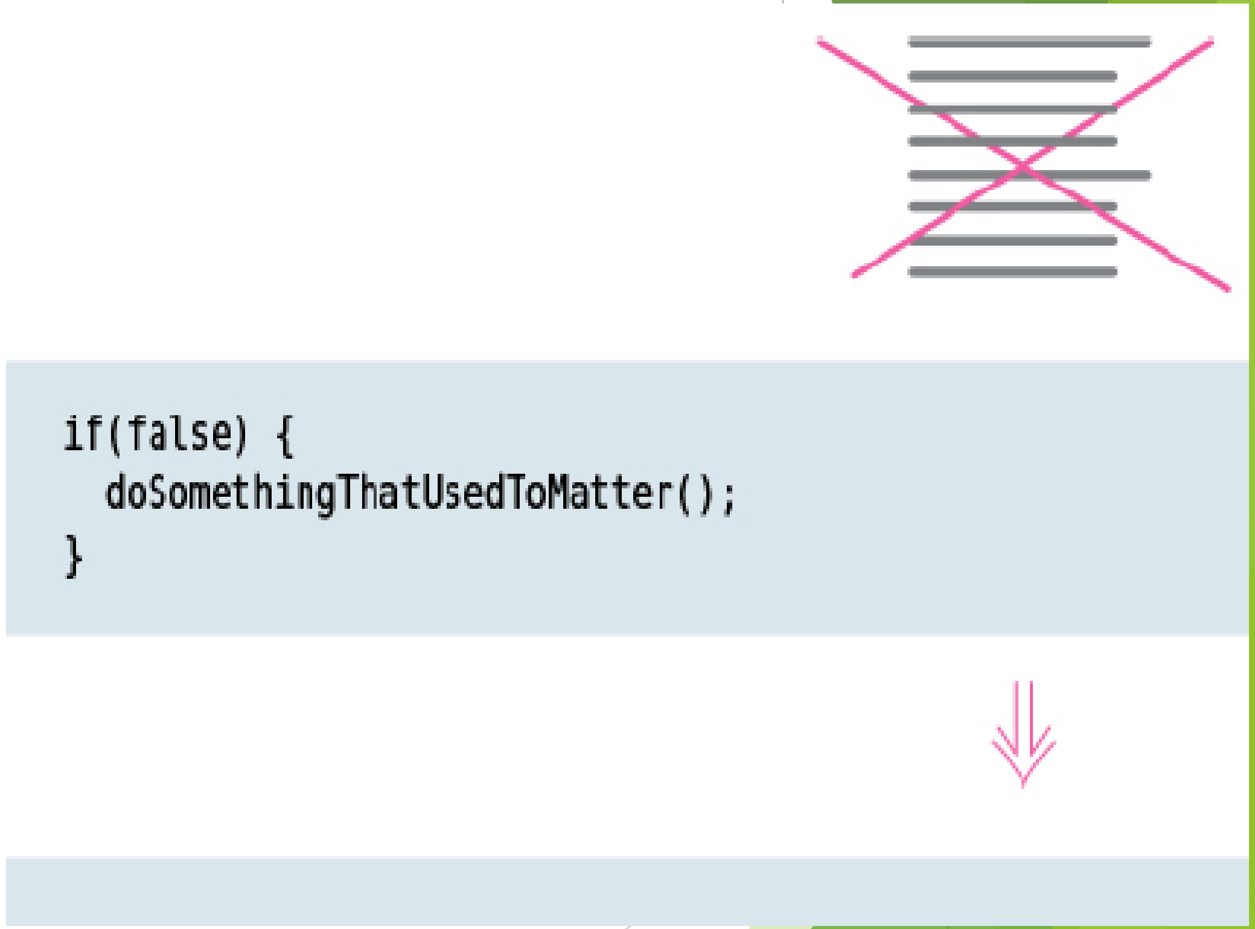
```
const names = [];
for (const i of input) {
  if (i.job === "programmer")
    names.push(i.name);
}
```

⇓

```
const names = input
  .filter(i => i.job === "programmer")
  .map(i => i.name)
;
```

Moving Features: Remove Dead Code

- If the dead code can be referenced from outside, e.g., when it’s a full function, do a search to check for callers.
- Remove the dead code.
- Test.

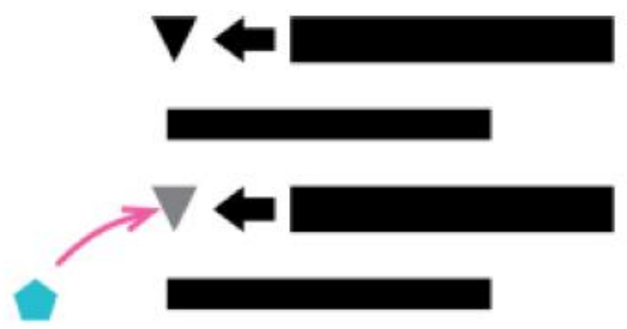


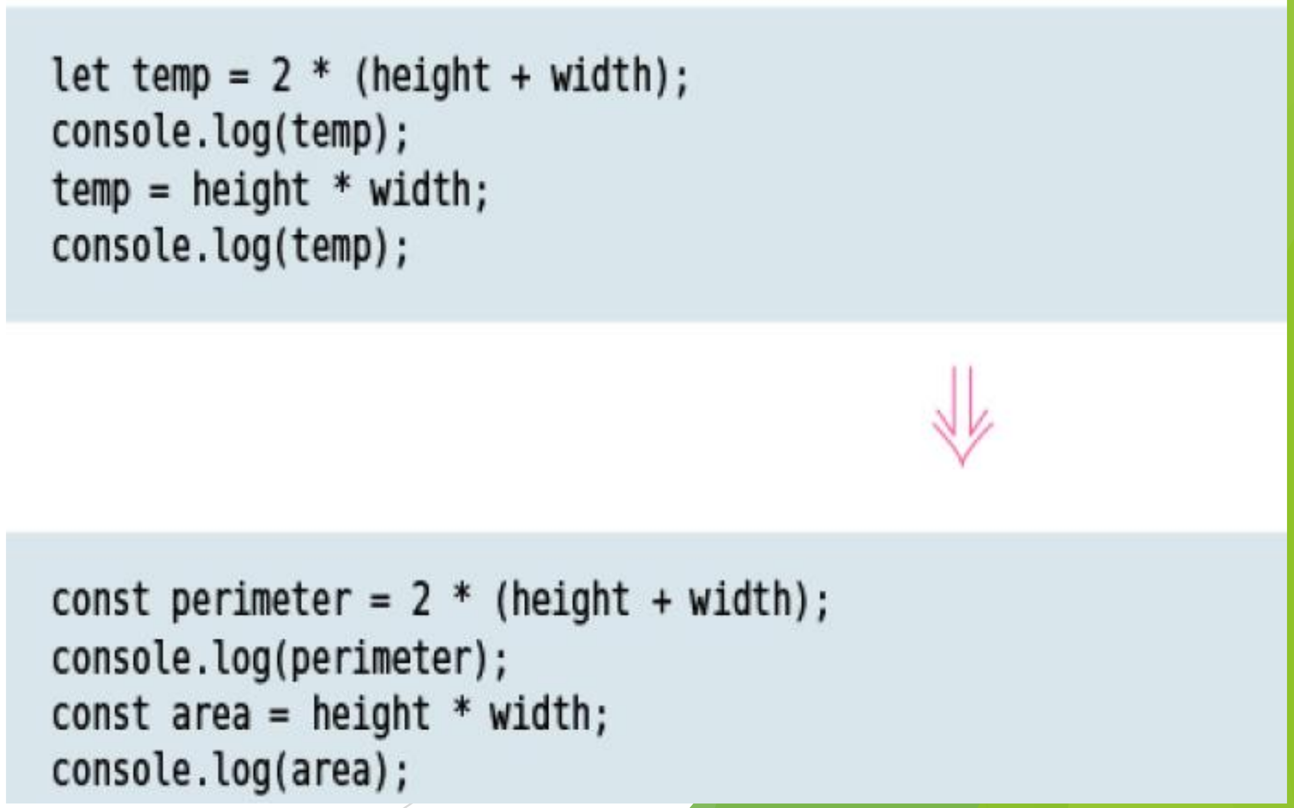
```
if(false) {
  doSomethingThatUsedToMatter();
}
```

Organizing Data: Split Variable

- Change the name of the variable at its declaration and first assignment.

  If the later assignments are of the form `i = i + something`, that is a collecting variable, so don’t split it. A collecting variable is often used for calculating sums, string concatenation, writing to a stream, or adding to a collection.
- If possible, declare the new variable as immutable.
- Change all references of the variable up to its second assignment.
- Test.
- Repeat in stages, at each stage renaming the variable at the declaration and changing references until the next assignment, until you reach the final assignment.





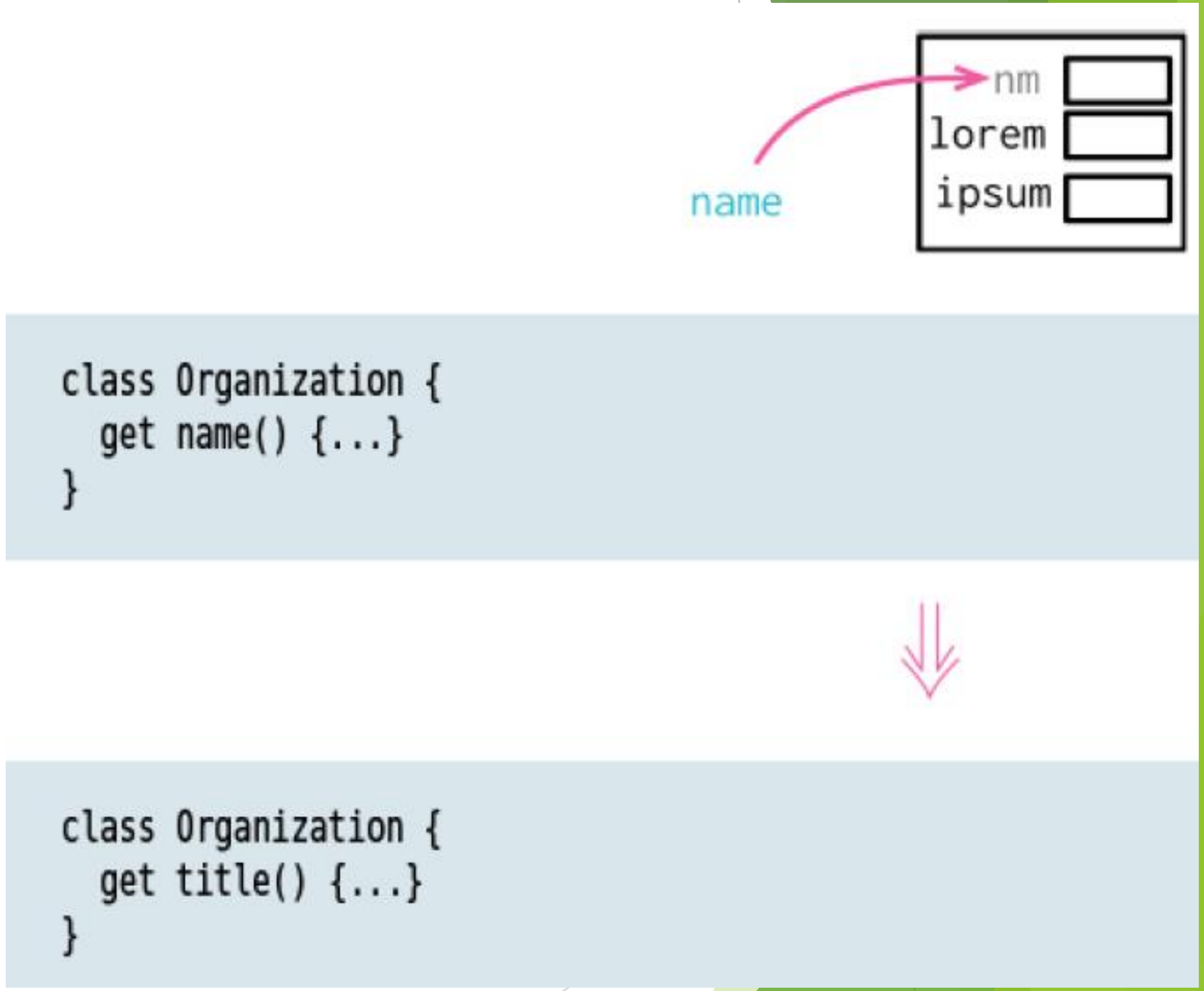
```
let temp = 2 * (height + width);
console.log(temp);
temp = height * width;
console.log(temp);
```

⇓

```
const perimeter = 2 * (height + width);
console.log(perimeter);
const area = height * width;
console.log(area);
```

Organizing Data: Rename Field

- If the record has limited scope, rename all accesses to the field and test; no need to do the rest of the mechanics.
- If the record isn't already encapsulated, apply *Encapsulate Record*.
- Rename the private field inside the object, adjust internal methods to fit.
- Test.
- If the constructor uses the name, apply *Change Function Declaration (124)* to rename it.
- Apply *Rename Function (124)* to the accessors.



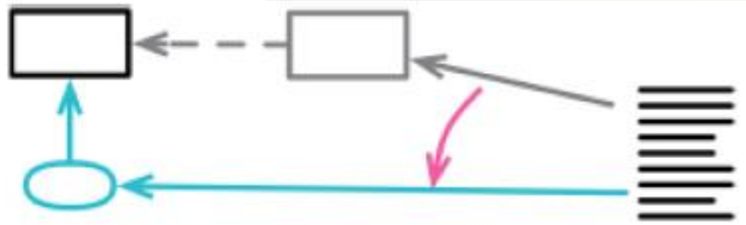
```
class Organization {
  get name() {...}
}
```

```
class Organization {
  get title() {...}
}
```

Organizing Data: Replace Derived Variable with Query

- Identify all points of update for the variable. If necessary, use *Split* to separate each point of update.
- Create a function that calculates the value of the variable.
- Use *Introduce Assertion* to assert that the variable and the calculation give the same result whenever the variable is used.

  If necessary, use *Encapsulate Variable* to provide a home for the assertion.
- Test.
- Replace any reader of the variable with a call to the new function.
- Test.
- Apply *Remove Dead Code* to the declaration and updates to the variable.



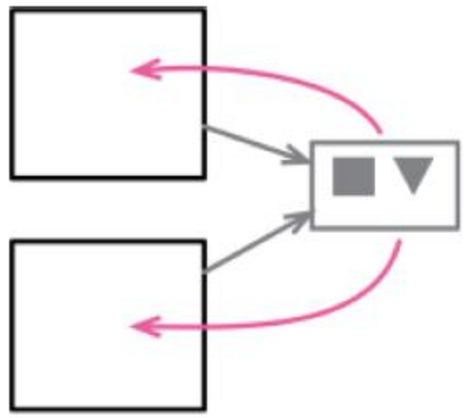
```
get discountedTotal() {return this._discountedTotal;}
set discount(aNumber) {
  const old = this._discount;
  this._discount = aNumber;
  this._discountedTotal += old - aNumber;
}
```

⇓

```
get discountedTotal() {return this._baseTotal - this._discount;}
set discount(aNumber) {this._discount = aNumber;}
```

Organizing Data: Change Reference to Value

- Check that the candidate class is immutable or can become immutable.
- For each setter, apply *Remove Setting Method* .
- Provide a value-based equality method that uses the fields of the value object.
- Most language environments provide an overridable equality function for this purpose. Usually you must override a hash-code generator method as well.



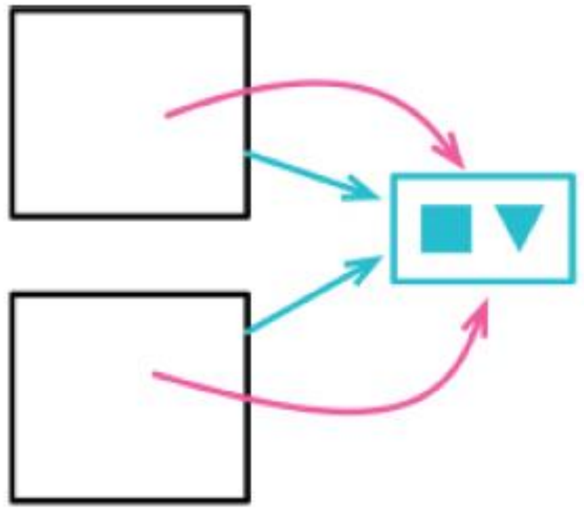
```
class Product {
  applyDiscount(arg) {this._price.amount -= arg;}
```

⇓

```
class Product {
  applyDiscount(arg) {
    this._price = new Money(this._price.amount - arg, this._price.currency);
  }
```

Organizing Data: Change Value to Reference

- Create a repository for instances of the related object (if one isn't already present).
- Ensure the constructor has a way of looking up the correct instance of the related object.
- Change the constructors for the host object to use the repository to obtain the related object. Test after each change



```
let customer = new Customer(customerData);
```

⇓

```
let customer = customerRepository.get(customerData.id);
```

More Methods…

- Covered directly from the book