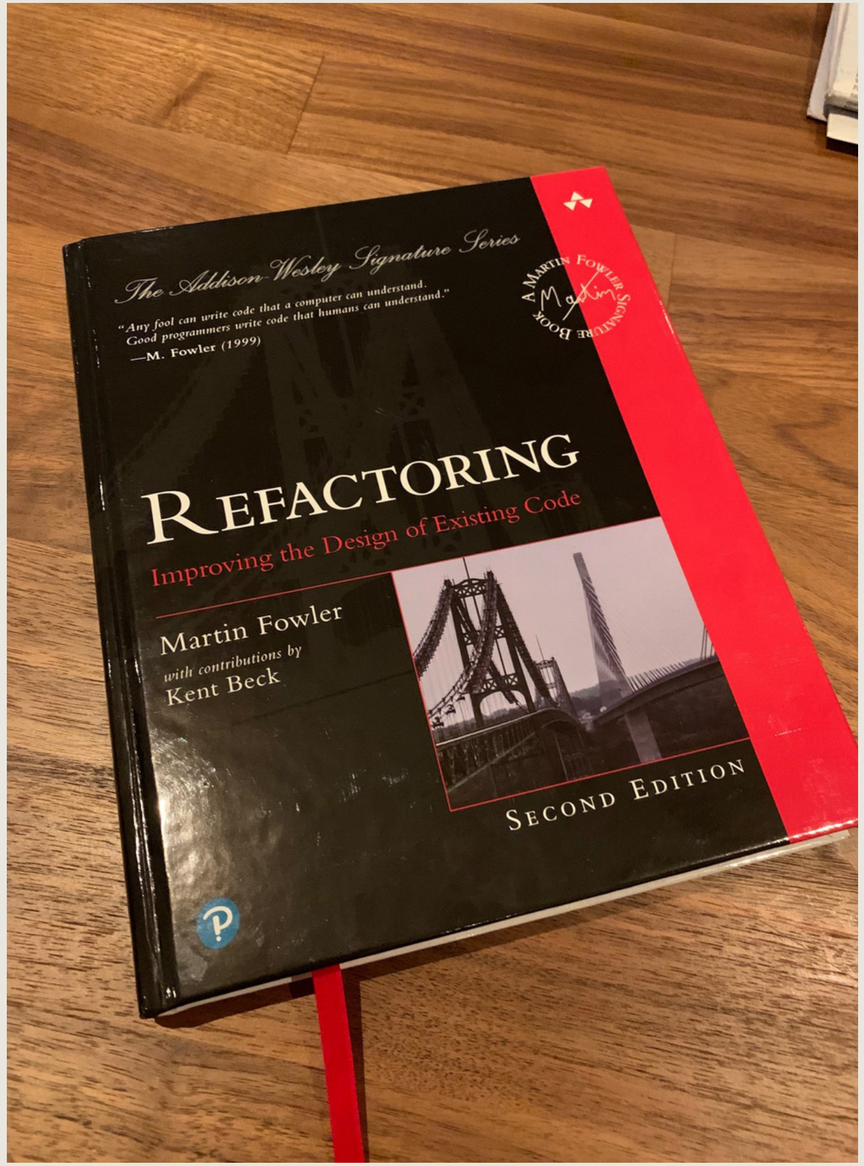
Refactoring:Improving the Design of Existing Code

Chepter 9,10

Ahmet Turhan

13.08.2021

i2i Systems



Titles

Chapter 9 Organizing Data

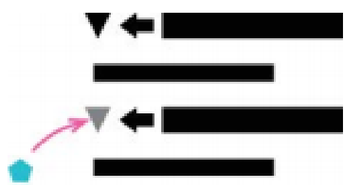
- Split Variable
- Rename Field
- Replace Derived Variable with Quary
- Change Reference to Value
- Change Value to Reference

Chapter 10 Simplifiying Conditional Logic

- Decompose Conditional
- Consolidate Conditional Expression
- Replace Nested Conditional with Guard Clauses
- Replace Conditional with Polimorphism
- Introduce Special Case
- Introduce Assertion

CHAPTER 9
ORGANIZING DATA

Split Variable



```
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);
```

Variables have various uses. Some of these uses naturally lead to the variable being assigned to several times. Many other variables are used to hold the result of a long-winded bit of code for easy reference later. These kinds of variables should be set only once. Any variable with more than one responsibility should be replaced with multiple variables, one for each responsibility. Using a variable for two different things is very confusing for the reader.

Example

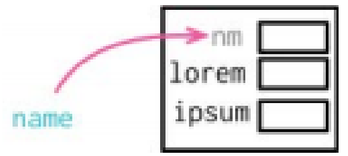
A nice awkward little function. The interesting thing for our example is the way the variable acc is set twice. It has two responsibilities: one to hold the initial acceleration from the first force and another later to hold the acceleration from both forces. I want to split this variable.

```
function distanceTravelled (scenario, time) {
  let result;
  const primaryAcceleration = scenario.primaryForce / scenario.mass;
  let primaryTime = Math.min(time, scenario.delay);
  result = 0.5 * primaryAcceleration * primaryTime * primaryTime;
  let secondaryTime = time - scenario.delay;
  if (secondaryTime > 0) {
    let primaryVelocity = primaryAcceleration * scenario.delay;
    let acc = (scenario.primaryForce + scenario.secondaryForce) / scenario.mass;
    result += primaryVelocity * secondaryTime + 0.5 * acc * secondaryTime * secondaryTime;
  }
  return result;
}
```

```
function distanceTravelled (scenario, time) {
  let result;
  let acc = scenario.primaryForce / scenario.mass;
  let primaryTime = Math.min(time, scenario.delay);
  result = 0.5 * acc * primaryTime * primaryTime;
  let secondaryTime = time - scenario.delay;
  if (secondaryTime > 0) {
    let primaryVelocity = acc * scenario.delay;
    acc = (scenario.primaryForce + scenario.secondaryForce) / scenario.mass;
    result += primaryVelocity * secondaryTime + 0.5 * acc * secondaryTime * secondaryTime;
  }
  return result;
}
```

```
function distanceTravelled (scenario, time) {
  let result;
  const primaryAcceleration = scenario.primaryForce / scenario.mass;
  let primaryTime = Math.min(time, scenario.delay);
  result = 0.5 * primaryAcceleration * primaryTime * primaryTime;
  let secondaryTime = time - scenario.delay;
  if (secondaryTime > 0) {
    let primaryVelocity = primaryAcceleration * scenario.delay;
    const secondaryAcceleration = (scenario.primaryForce + scenario.secondaryForce) / scenario.mass;
    result += primaryVelocity * secondaryTime +
      0.5 * secondaryAcceleration * secondaryTime * secondaryTime;
  }
  return result;
}
```

Rename Field




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

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

Names are important, and field names in record structures can be especially important when those record structures are widely used across a program. Data structures play a particularly important role in understanding.You may want to rename a field in a record structure, but the idea also applies to classes. Getter and setter methods form an effective field for users of the class. Renaming them is just as important as with bare record structures.

Example

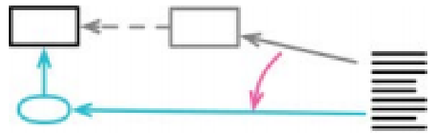
```
class Organization {
  constructor(data) {
    this._name = data.name;
    this._country = data.country;
  }
  get name()    {return this._name;}
  set name(aString) {this._name = aString;}
  get country()    {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}

const organization = new Organization({name: "Acme Gooseberries", country: "GB"});
```

```
class Organization {
  constructor(data) {
    this._title = data.name;
    this._country = data.country;
  }
  get name()    {return this._title;}
  set name(aString) {this._title = aString;}
  get country()    {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

```
class Organization {
  constructor(data) {
    this._title = data.title;
    this._country = data.country;
  }
  get name()    {return this._title;}
  set name(aString) {this._ title = aString;}
  get country()    {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

```
class Organization {
  constructor(data) {
    this._title = data.title;
    this._country = data.country;
  }
  get title()    {return this._title;}
  set title(aString) {this._title = aString;}
  get country()    {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

Replace Derived Veriable With Query



```
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;}
```

One of the biggest sources of problems in software is mutable data. Data changes can often couple together parts of code in awkward ways, with changes in one part leading to knock-on effects that are hard to spot. In many situations it’s not realistic to entirely remove mutable data but I do advocate minimizing the scope of mutable data at much as possible. A reasonable exception to this is when the source data for the calculation is immutable and we can force the result to being immutable too.

Example

class ProductionPlan...

```
get production() {return this._production;}
applyAdjustment(anAdjustment) {
  this._adjustments.push(anAdjustment);
  this._production += anAdjustment.amount;
}
```

```
get production() {
  assert(this._production === this.calculatedProduction);
  return this._production;
}

get calculatedProduction() {
  return this._adjustments
    .reduce((sum, a) => sum + a.amount, 0);
}
```

class ProductionPlan...

```
get production() {
  assert(this._production === this.calculatedProduction);
  return this.calculatedProduction;
}
```

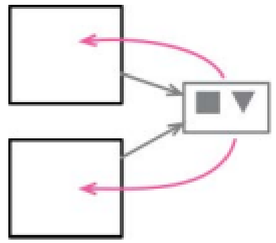
class ProductionPlan...

```
get production() {
  return this._adjustments
    .reduce((sum, a) => sum + a.amount, 0);
}
```

class ProductionPlan...

```
applyAdjustment(anAdjustment) {
  this._adjustments.push(anAdjustment);
  this._production += anAdjustment.amount;
}
```

Change Referance To Value



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

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

When I nest an object, or data structure, within another I can treat the inner object as a reference or as a value. The difference is most obviously visible in how I handle updates of the inner object's properties. If I treat it as a reference, I'll update the inner object's property keeping the same inner object. If I treat it as a value, I will replace the entire inner object with a new one that has the desired property.If I want to share an object between several objects so that any change to the shared object is visible to all its collaborators, then I need the shared object to be a reference.

Example

class Person...

```
constructor() {
  this._telephoneNumber = new TelephoneNumber();
}

get officeAreaCode()    {return this._telephoneNumber.areaCode;}
set officeAreaCode(arg) {this._telephoneNumber.areaCode = arg;}
get officeNumber()    {return this._telephoneNumber.number;}
set officeNumber(arg) {this._telephoneNumber.number = arg;}
```

class TelephoneNumber...

```
get areaCode()    {return this._areaCode;}
set areaCode(arg) {this._areaCode = arg;}

get number()    {return this._number;}
set number(arg) {this._number = arg;}
```

class TelephoneNumber...

```
equals(other) {
  if (!(other instanceof TelephoneNumber)) return false;
  return this.areaCode === other.areaCode &&
    this.number === other.number;
}
```

class TelephoneNumber...

```
constructor(areaCode, number) {
  this._areaCode = areaCode;
  this._number = number;
}
```

class Person...

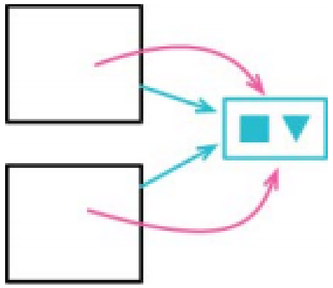
```
get officeAreaCode()    {return this._telephoneNumber.areaCode;}
set officeAreaCode(arg) {
  this._telephoneNumber = new TelephoneNumber(arg, this.officeNumber);
}
get officeNumber()    {return this._telephoneNumber.number;}
set officeNumber(arg) {this._telephoneNumber.number = arg;}
```

I then repeat that step with the remaining field.

class Person...

```
get officeAreaCode()    {return this._telephoneNumber.areaCode;}
set officeAreaCode(arg) {
  this._telephoneNumber = new TelephoneNumber(arg, this.officeNumber);
}
get officeNumber()    {return this._telephoneNumber.number;}
set officeNumber(arg) {
  this._telephoneNumber = new TelephoneNumber(this.officeAreaCode, arg);
}
```

Change Value To Referance



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

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

A data structure may have several records linked to the same logical data structure. I might read in a list of orders, some of which are for the same customer. When I have sharing like this, I can represent it by treating the customer either as a value or as a reference. With a value, the customer data is copied into each order; with a reference, there is only one data structure that multiple orders link to. Changing a value to a reference results in only one object being present for an entity, and it usually means I need some kind of repository where I can access these objects.

Example

class Order...

```
constructor(data) {
  this._number = data.number;
  this._customer = new Customer(data.customer);
  // load other data
}
get customer() {return this._customer;}
```

class Customer...

```
constructor(id) {
  this._id = id;
}
get id() {return this._id;}
```

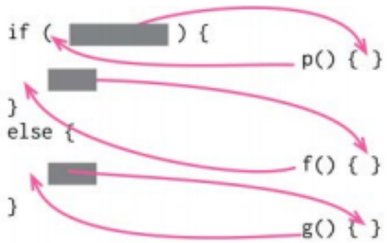
```
let _repositoryData;

export function initialize() {
  _repositoryData = {};
  _repositoryData.customers = new Map();
}

export function registerCustomer(id) {
  if (! _repositoryData.customers.has(id))
    _repositoryData.customers.set(id, new Customer(id));
  return findCustomer(id);
}

export function findCustomer(id) {
  return _repositoryData.customers.get(id);
}
```

class Order...

```
constructor(data) {
  this._number = data.number;
  this._customer = registerCustomer(data.customer);
  // load other data
}
get customer() {return this._customer;}
```

CHAPTER 10
SIMPLIFYING CONDITIONAL LOGIC

Decompose Conditional




```
if (!aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd))
  charge = quantity * plan.summerRate;
else
  charge = quantity * plan.regularRate + plan.regularServiceCharge;
```

```
if (summer())
  charge = summerCharge();
else
  charge = regularCharge();
```

One of the most common sources of complexity in a program is complex conditional logic. As I write code to do various things depending on various conditions, I can quickly end up with a pretty long functions with any large block of code, I can make my intention clearer by decomposing it and replacing each chunk of code with a function call named after the intention of that chunk. This way, I highlight the condition and make it clear what I'm branching on.

Example

```
if (!aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd))
  charge = quantity * plan.summerRate;
else
  charge = quantity * plan.regularRate + plan.regularServiceCharge;
```

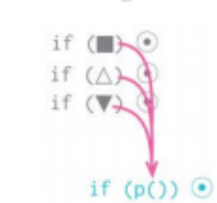
```
if (summer())
  charge = summerCharge();
else
  charge = quantity * plan.regularRate + plan.regularServiceCharge;

function summer() {
  return !aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd);
}
function summerCharge() {
  return quantity * plan.summerRate;
}
```

```
charge = summer() ? summerCharge() : regularCharge();

function summer() {
  return !aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd);
}
function summerCharge() {
  return quantity * plan.summerRate;
}
function regularCharge() {
  return quantity * plan.regularRate + plan.regularServiceCharge;
}
```

```
if (summer())
  charge = summerCharge();
else
  charge = regularCharge();

function summer() {
  return !aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd);
}
function summerCharge() {
  return quantity * plan.summerRate;
}
function regularCharge() {
  return quantity * plan.regularRate + plan.regularServiceCharge;
}
```

```
if (summer())
  charge = quantity * plan.summerRate;
else
  charge = quantity * plan.regularRate + plan.regularServiceCharge;

function summer() {
  return !aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd);
}
```

Consolidate Conditional Expression




```
if (anEmployee.seniority < 2) return 0;
if (anEmployee.monthsDisabled > 12) return 0;
if (anEmployee.isPartTime) return 0;
```

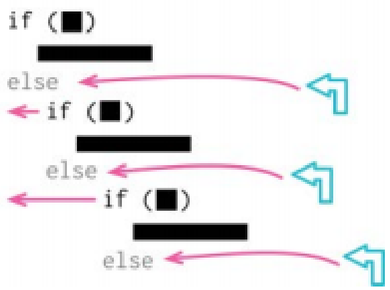
```
if (isNotEligableForDisability()) return 0;

function isNotEligableForDisability() {
  return ((anEmployee.seniority < 2)
          || (anEmployee.monthsDisabled > 12)
          || (anEmployee.isPartTime));
}
```

Sometimes, I run into a series of conditional checks where each check is different yet the resulting action is the same. When I see this, I use and and or operators to consolidate them into a single conditional check with a single result. Consolidating the conditional code is important because it makes it clearer by showing that I'm really making a single check that combines other checks.

Example

```
function disabilityAmount(anEmployee) {
  if (anEmployee.seniority < 2) return 0;
  if (anEmployee.monthsDisabled > 12) return 0;
  if (anEmployee.isPartTime) return 0;
  // compute the disability amount
```

```
function disabilityAmount(anEmployee) {
  if (isNotEligableForDisability()) return 0;
  // compute the disability amount

  function isNotEligableForDisability() {
    return ((anEmployee.seniority < 2)
            || (anEmployee.monthsDisabled > 12)
            || (anEmployee.isPartTime));
  }
```




```
function getPayAmount() {
  let result;
  if (isDead)
    result = deadAmount();
  else {
    if (isSeparated)
      result = separatedAmount();
    else {
      if (isRetired)
        result = retiredAmount();
      else
        result = normalPayAmount();
    }
  }
  return result;
}
```

```
function getPayAmount() {
  if (isDead) return deadAmount();
  if (isSeparated) return separatedAmount();
  if (isRetired) return retiredAmount();
  return normalPayAmount();
}
```


Replace Nested Conditional With Guard Clauses

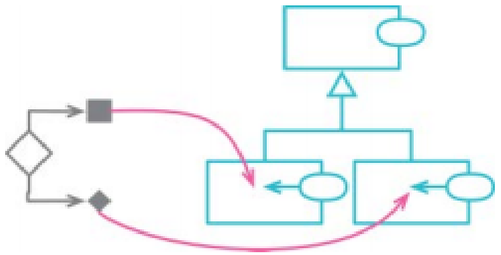
I often find that conditional expressions come in two styles. In the first style, both legs of the conditional are part of normal behavior, while in the second style, one leg is normal and the other indicates an unusual condition. If both are part of normal behavior, I use a condition with an if and an else leg. If the condition is an unusual condition, I check the condition and return if it’s true. This kind of check is often called a guard clause.

Example

```
function payAmount(employee) {
  let result;
  if(employee.isSeparated) {
    result = {amount: 0, reasonCode: "SEP"};
  }
  else {
    if (employee.isRetired) {
      result = {amount: 0, reasonCode: "RET"};
    }
    else {
      // logic to compute amount
      lorem.ipsum(dolor.sitAmet);
      consectetur(adipiscing).elit();
      sed.do.eiusmod = tempor.incididunt.ut(labore) && dolore(magna.aliqua);
      ut.enim.ad(minim.veniam);
      result = someFinalComputation();
    }
  }
  return result;
}
```

```
function payAmount(employee) {
  let result;
  if (employee.isSeparated) return {amount: 0, reasonCode: "SEP"};
  if (employee.isRetired)   return {amount: 0, reasonCode: "RET"};
  // logic to compute amount
  lorem.ipsum(dolor.sitAmet);
  consectetur(adipiscing).elit();
  sed.do.eiusmod = tempor.incididunt.ut(labore) && dolore(magna.aliqua);
  ut.enim.ad(minim.veniam);
  result = someFinalComputation();
  return result;
}
```

```
function payAmount(employee) {
  let result;
  if (employee.isSeparated) return {amount: 0, reasonCode: "SEP"};
  if (employee.isRetired)   return {amount: 0, reasonCode: "RET"};
  // logic to compute amount
  lorem.ipsum(dolor.sitAmet);
  consectetur(adipiscing).elit();
  sed.do.eiusmod = tempor.incididunt.ut(labore) && dolore(magna.aliqua);
  ut.enim.ad(minim.veniam);
  return someFinalComputation();
}
```



```
switch (bird.type) {
  case 'EuropeanSwallow':
    return "average";
  case 'AfricanSwallow':
    return (bird.numberOfCoconuts > 2) ? "tired" : "average";
  case 'NorwegianBlueParrot':
    return (bird.voltage > 100) ? "scorched" : "beautiful";
  default:
    return "unknown";
```

```
class EuropeanSwallow {
  get plumage() {
    return "average";
  }
class AfricanSwallow {
  get plumage() {
     return (this.numberOfCoconuts > 2) ? "tired" : "average";
  }
class NorwegianBlueParrot {
  get plumage() {
     return (this.voltage > 100) ? "scorched" : "beautiful";
  }
```


Replace Conditional With Polimorphism

Complex conditional logic is one of the hardest things to reason about in programming, so I always look for ways to add structure to conditional logic. Often, I find I can separate the logic into different circumstances—high-level cases—to divide the conditions. Most of my conditional logic uses basic conditional statements—if/else and switch/case. But when I see complex conditional logic that can be improved as discussed above, I find polymorphism a powerful tool.

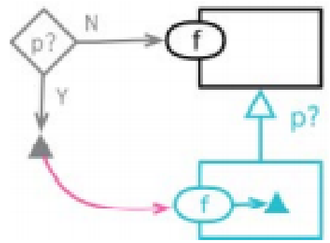
Example

```
function plumages(birds) {
  return new Map(birds.map(b => [b.name, plumage(b)]));
}
function speeds(birds) {
  return new Map(birds.map(b => [b.name, airSpeedVelocity(b)]));
}

function plumage(bird) {
  switch (bird.type) {
  case 'EuropeanSwallow':
    return "average";
  case 'AfricanSwallow':
    return (bird.numberOfCoconuts > 2) ? "tired" : "average";
  case 'NorwegianBlueParrot':
    return (bird.voltage > 100) ? "scorched" : "beautiful";
  default:
    return "unknown";
  }
}

function airSpeedVelocity(bird) {
  switch (bird.type) {
  case 'EuropeanSwallow':
    return 35;
  case 'AfricanSwallow':
    return 40 - 2 * bird.numberOfCoconuts;
  case 'NorwegianBlueParrot':
    return (bird.isNailed) ? 0 : 10 + bird.voltage / 10;
  default:
    return null;
  }
}
```

```
function plumages(birds) {
  return new Map(birds
                .map(b => createBird(b))
                .map(bird => [bird.name, bird.plumage]));
}
function speeds(birds) {
  return new Map(birds
                .map(b => createBird(b))
                .map(bird => [bird.name, bird.airSpeedVelocity]));
}

function createBird(bird) {
  switch (bird.type) {
  case 'EuropeanSwallow':
    return new EuropeanSwallow(bird);
  case 'AfricanSwallow':
    return new AfricanSwallow(bird);
  case 'NorwegianBlueParrot':
    return new NorwegianBlueParrot(bird);
  default:
    return new Bird(bird);
  }
}

class Bird {
  constructor(birdObject) {
    Object.assign(this, birdObject);
  }
  get plumage() {
    return "unknown";
  }
  get airSpeedVelocity() {
    return null;
  }
}
class EuropeanSwallow extends Bird {
  get plumage() {
    return "average";
  }
  get airSpeedVelocity() {
    return 35;
  }
}
```

```
lass AfricanSwallow extends Bird {
 get plumage() {
   return (this.numberOfCoconuts > 2) ? "tired" : "average";
 }
 get airSpeedVelocity() {
   return 40 - 2 * this.numberOfCoconuts;
 }

lass NorwegianBlueParrot extends Bird {
 get plumage() {
   return (this.voltage > 100) ? "scorched" : "beautiful";
 }
 get airSpeedVelocity() {
   return (this.isNailed) ? 0 : 10 + this.voltage / 10;
 }
```

Introduce Special Case




```
if (aCustomer === "unknown") customerName = "occupant";
```

```
class UnknownCustomer {
    get name() {return "occupant";}
```

A common case of duplicated code is when many users of a data structure check a specific value, and then most of them do the same thing. If I find many parts of the code base having the same reaction to a particular value, I want to bring that reaction into a single place. A good mechanism for this is the Special Case pattern where I create a specialcase element that captures all the common behavior.

Example

class Site...

```
get customer() {return this._customer;}
```

There are various properties of the customer class; I'll consider three of them.

class Customer...

```
get name()           {...}
get billingPlan()    {...}
set billingPlan(arg) {...}
get paymentHistory() {...}
```

class Customer...

```
get isUnknown() {return false;}
```

I then add an Unknown Customer class.

```
class UnknownCustomer {
  get isUnknown() {return true;}
}
```

client 1...

```
const aCustomer = site.customer;
// ... lots of intervening code ...
let customerName;
if (aCustomer === "unknown") customerName = "occupant";
else customerName = aCustomer.name;
```

client 2...

```
const plan = (aCustomer === "unknown") ?
      registry.billingPlans.basic
      : aCustomer.billingPlan;
```

client 3...

```
if (aCustomer !== "unknown") aCustomer.billingPlan = newPlan;
```

client 4...

```
const weeksDelinquent = (aCustomer === "unknown") ?
      0
      : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Example

```
function isUnknown(arg) {
  if (!((arg instanceof Customer) || (arg === "unknown")))
    throw new Error(`investigate bad value: <${arg}>`);
  return (arg === "unknown");
}
```

class Site...

```
get customer() {
  return (this._customer === "unknown") ? new UnknownCustomer() :  this._customer;
}
```

client 1...

```
let customerName;
if (isUnknown(aCustomer)) customerName = "occupant";
else customerName = aCustomer.name;
```

After a while, I have done them all.

client 2...

```
const plan = (isUnknown(aCustomer)) ?
      registry.billingPlans.basic
      : aCustomer.billingPlan;
```

client 3...

```
if (!isUnknown(aCustomer)) aCustomer.billingPlan = newPlan;
```

client 4...

```
const weeksDelinquent = isUnknown(aCustomer) ?
      0
      : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Example

class UnknownCustomer...

```
get name() {return "occupant";}
```

client 1...

```
const customerName = aCustomer.name;
```

client 2...

```
const plan = (isUnknown(aCustomer)) ?
      registry.billingPlans.basic
      : aCustomer.billingPlan;
```

client 3...

```
if (!isUnknown(aCustomer)) aCustomer.billingPlan = newPlan;
```

Introduce Assertion


```
if (this.discountRate)
  base = base - (this.discountRate * base);
```

```
assert(this.discountRate >= 0);
if (this.discountRate)
  base = base - (this.discountRate * base);
```

Often, sections of code work only if certain conditions are true. This may be as simple as a square root calculation only working on a positive input value. With an object, it may require that at least one of a group of fields has a value in it.Sometimes, the assumptions are stated with a comment. A better technique is to make the assumption explicit by writing an assertion.

Example

class Customer...

```
applyDiscount(aNumber) {
  if (!this.discountRate) return aNumber;
  else return aNumber - (this.discountRate * aNumber);
}
```

class Customer...

```
applyDiscount(aNumber) {
  if (!this.discountRate) return aNumber;
  else {
    assert(this.discountRate >= 0);
    return aNumber - (this.discountRate * aNumber);
  }
}
```

class Customer...

```
set discountRate(aNumber) {
  assert(null === aNumber || aNumber >= 0);
  this._discountRate = aNumber;
}
```

Thank you for listening to me.