SOLID

Principles of Software Design

Course Learning Objectives

- Learn:
  - Clean Code Basics
  - SOLID Principles of Software Design
  - Dependency Management
  - Professional Practices

SOFTWARE ROT

Messy Code

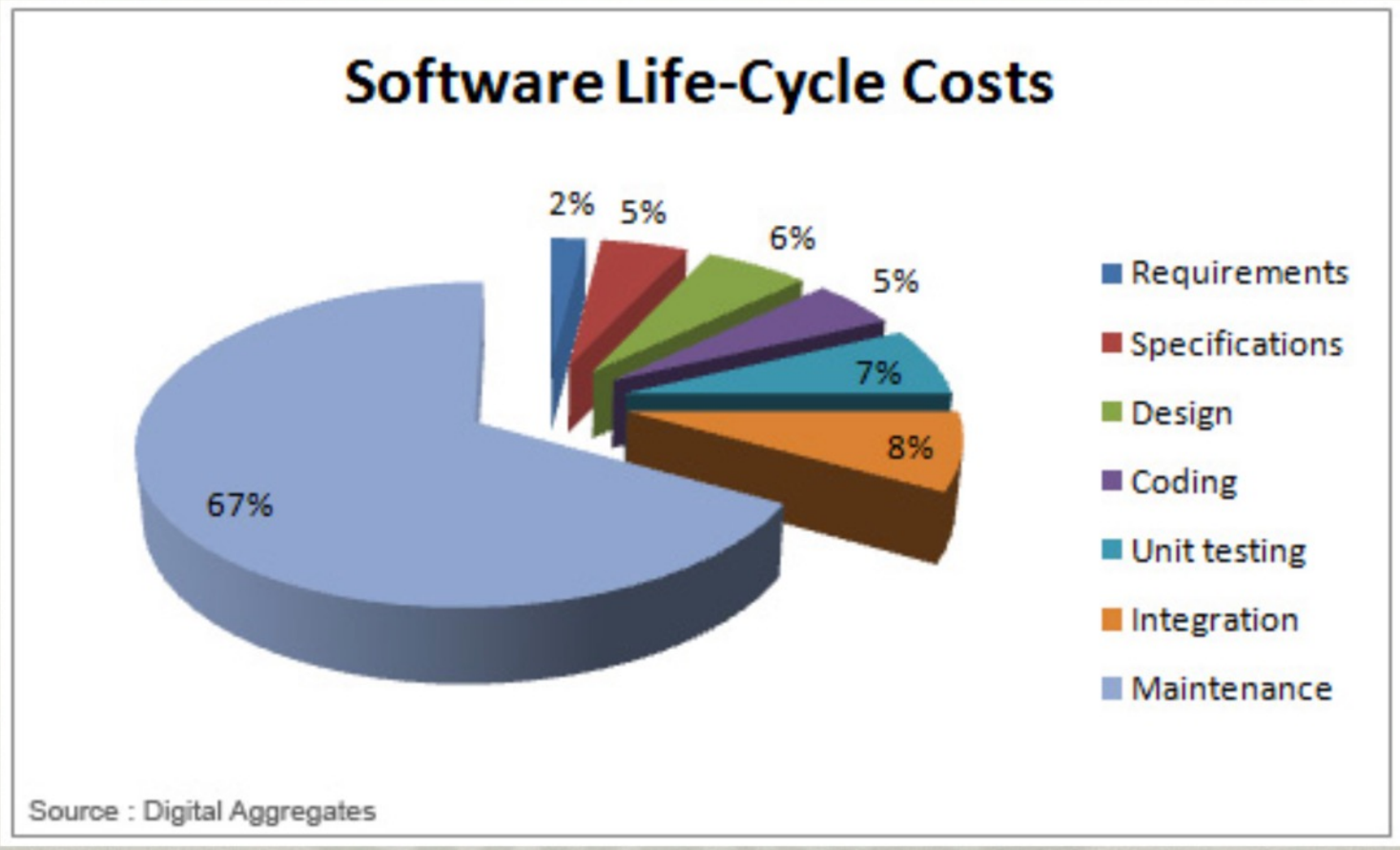
- Have you been significantly impeded by messy code?

With permission from: Mrs D. http://www.belly-timber.com/2005/10/03/embracing-our-inner-web-stat/

Messy Code

- Why did you write it?

Software Maintenance Cost



Maintenance Costs Are High Because...

- The code is messy.
- And messy code makes it hard to:
  - Understand existing code.
  - Find and fix bugs.
  - Add new features.
  - Adapt to new technology.

Designs Naturally Degrade

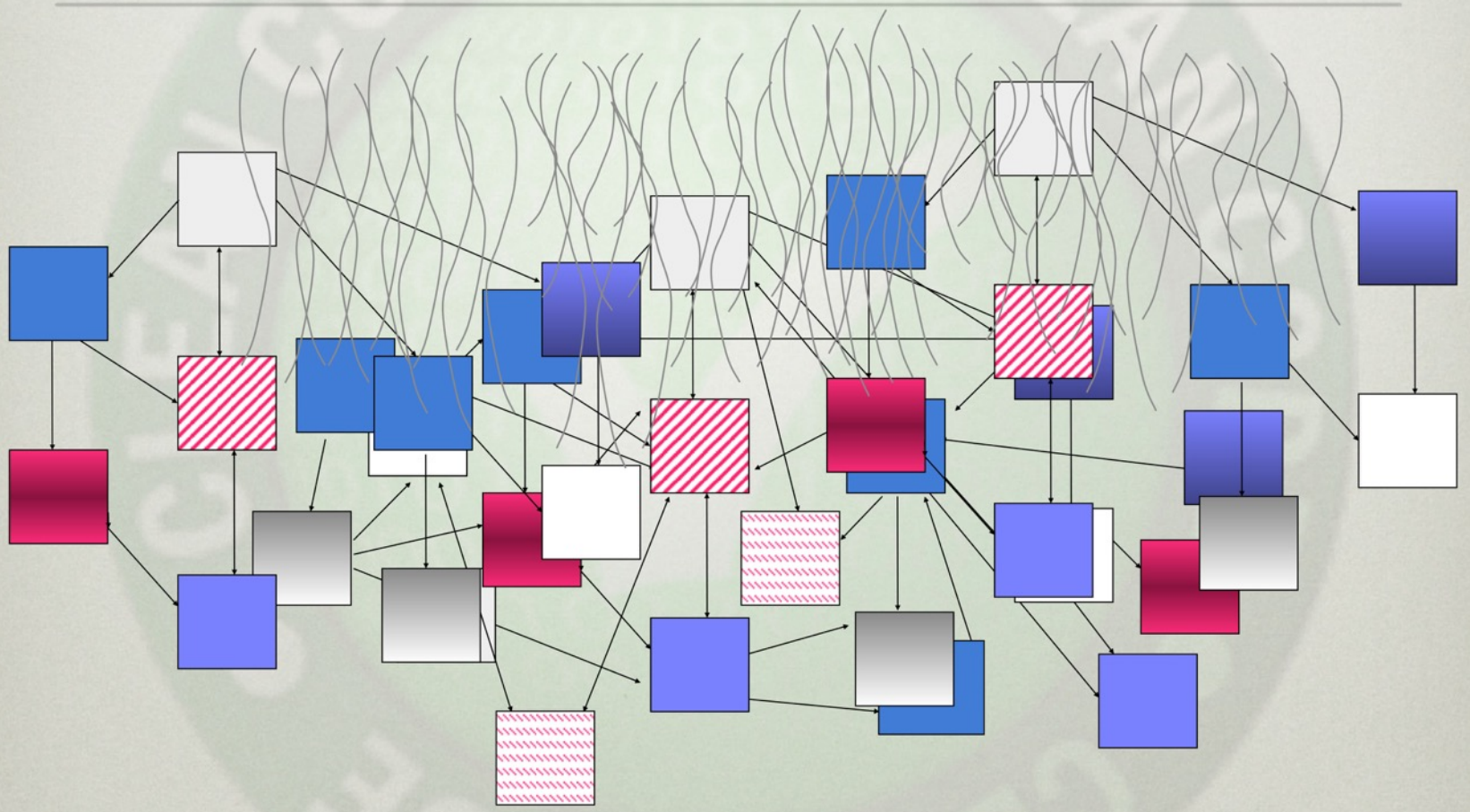
```
public class Pair
{
    private Object first;
    private Object second;
    private Object third;

    public Pair() { }
    public Pair( Object first, Object second, Object third )
    {
        this.first  = first;
        this.second  = second;
        this.third  = third;
    }

    public Object getFirst()  { return first; }
    public Object getSecond() { return second; }
    public Object getThird()  { return third; }

    public void setFirst( Object first  )  { this.first  = first; }
    public void setSecond( Object second ) { this.second  = second; }
    public void setThird( Object third  )  { this.third  = third; }
}
```

Neglect Is Contagious

- Disorder increases and software rots over time.
- Don't tolerate a broken windows.

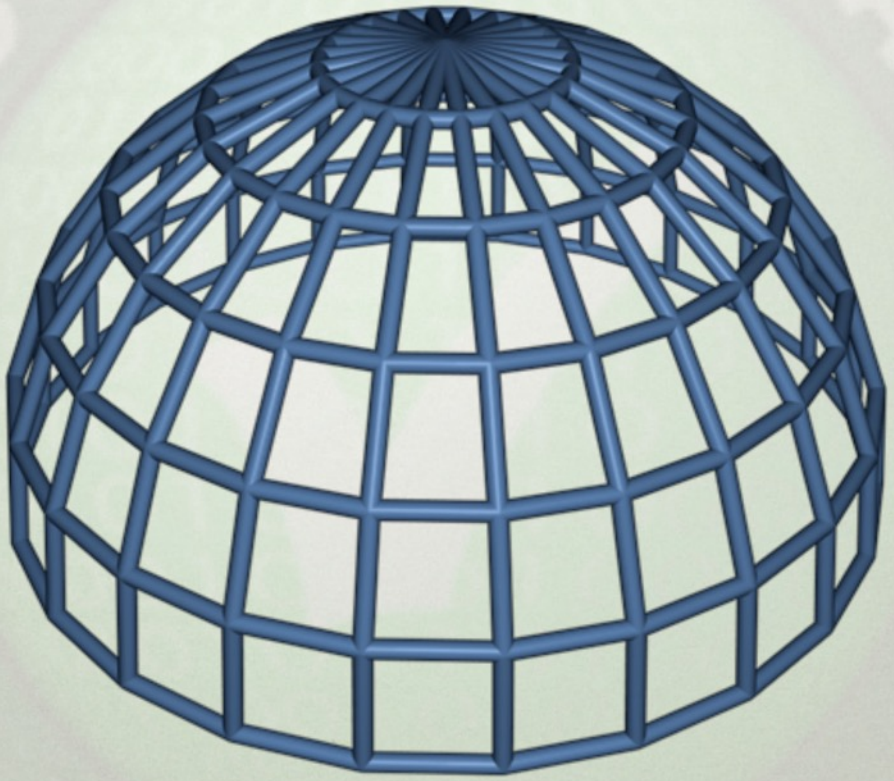
http://www.pragmaticprogrammer.com/ppbook/extracts/no_broken_windows.html

What Happens To a Design Over Time?



Designs rot when not carefully maintained.

Rigidity



- *The system is hard to change because every change forces many other changes to other parts of the system.*

Fragility

- Hidden dependencies make systems easy to break.
- The consequences of change are not localized.

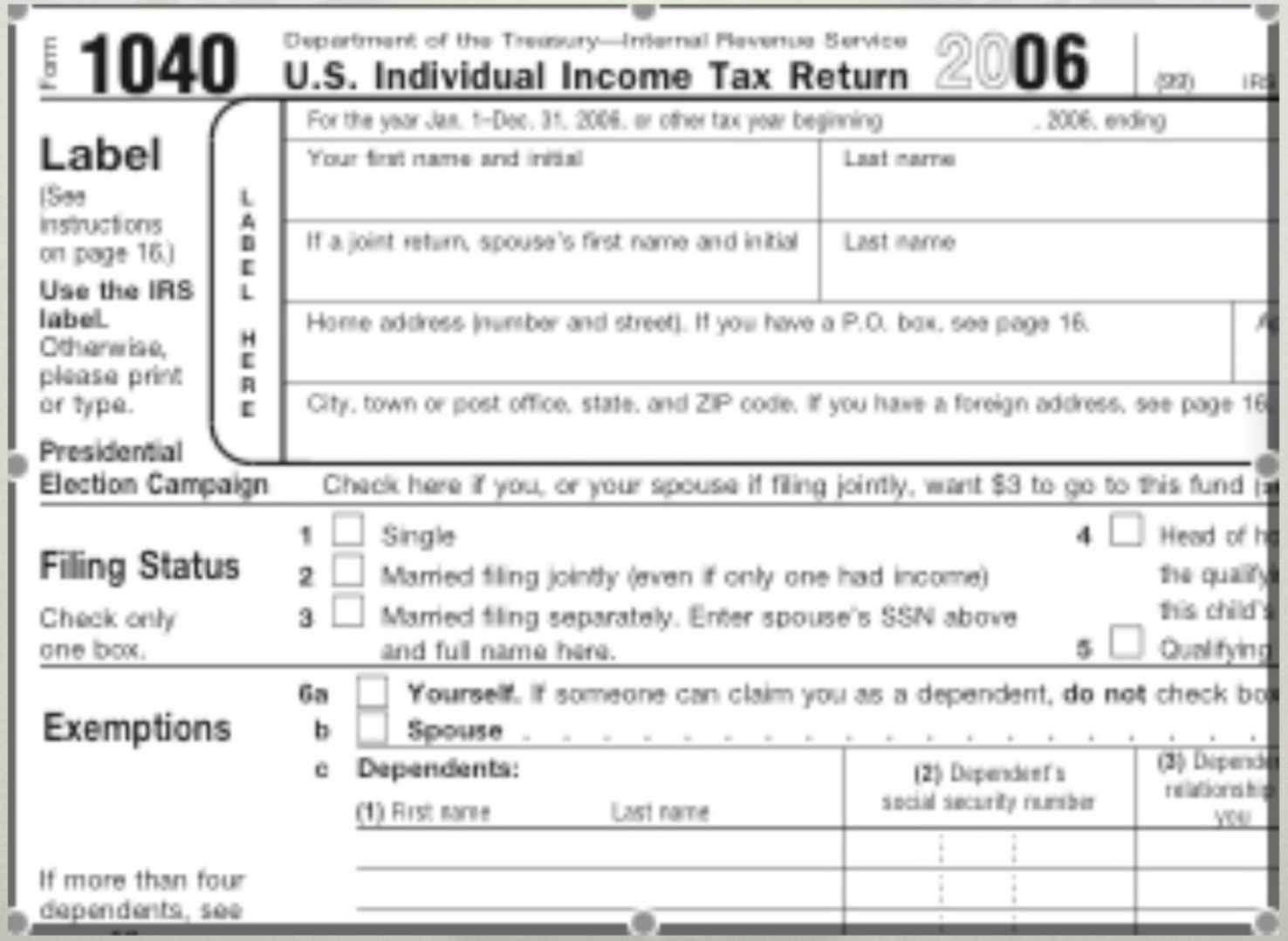
Immobility

- It is hard to disentangle the system into components that can be reused in other systems.

Viscosity

- Tools, environment, and design impede progress.
- It's slower to do the *right* thing than the *wrong* thing.

Needless Complexity



Form **1040** Department of the Treasury—Internal Revenue Service **U.S. Individual Income Tax Return** **2006**

Label
(See instructions on page 16.)
Use the IRS label. Otherwise, please print or type.

LABEL HERE

For the year Jan. 1–Dec. 31, 2006, or other tax year beginning . 2006, ending

Your first name and initial | Last name

If a joint return, spouse's first name and initial | Last name

Home address (number and street). If you have a P.O. box, see page 16.

City, town or post office, state, and ZIP code. If you have a foreign address, see page 16.

Presidential Election Campaign Check here if you, or your spouse if filing jointly, want $3 to go to this fund.

Filing Status
Check only one box.

1 ☐ Single
2 ☐ Married filing jointly (even if only one had income)
3 ☐ Married filing separately. Enter spouse's SSN above and full name here.
4 ☐ Head of h... the qualify... this child's...
5 ☐ Qualifying...

Exemptions

6a ☐ **Yourself.** If someone can claim you as a dependent, **do not** check box...
b ☐ **Spouse** . . . . . .
c **Dependents:**

(1) First name	Last name	(2) Dependent's social security number	(3) Dependent relationship to you

If more than four dependents, see

Needless Repetition

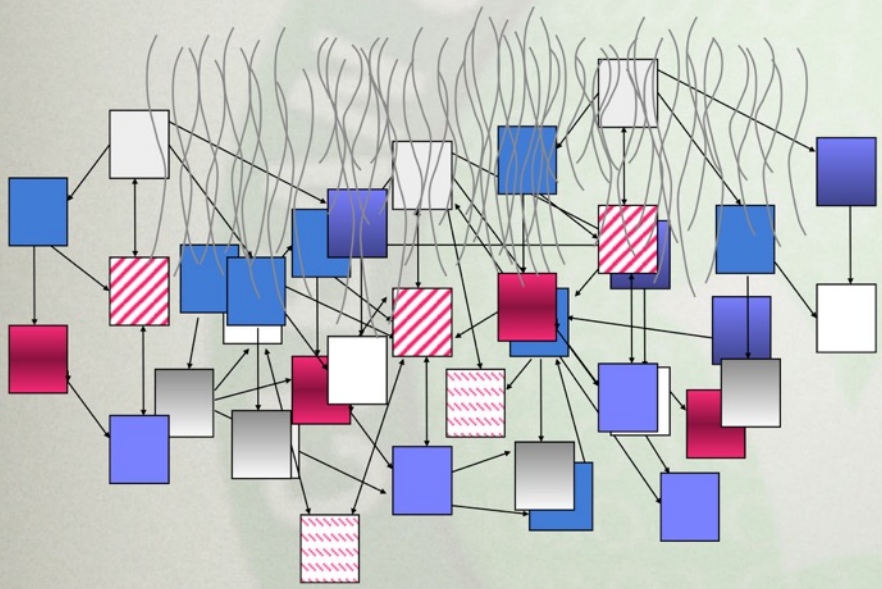
Opacity: Hard to understand

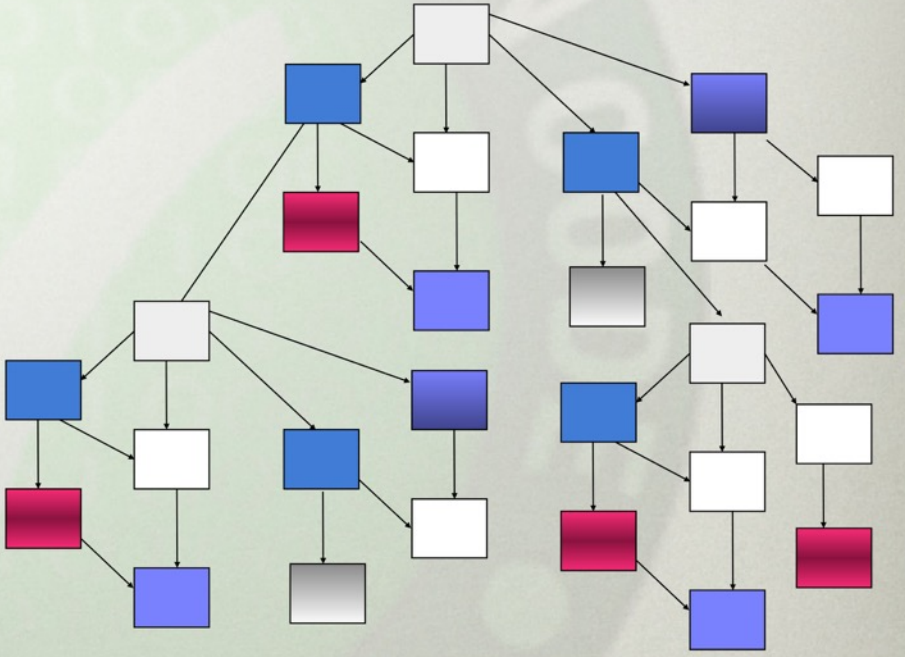
Code Smells

- Duplication (DRY)
- Long method
- Big class
- Feature envy
- Useless comment
- Poor name
- Inappropriate Intimacy
- Shotgun Surgery
- Switch
- God Class
- Primitive Obsession
- Refused Bequest

Remedies

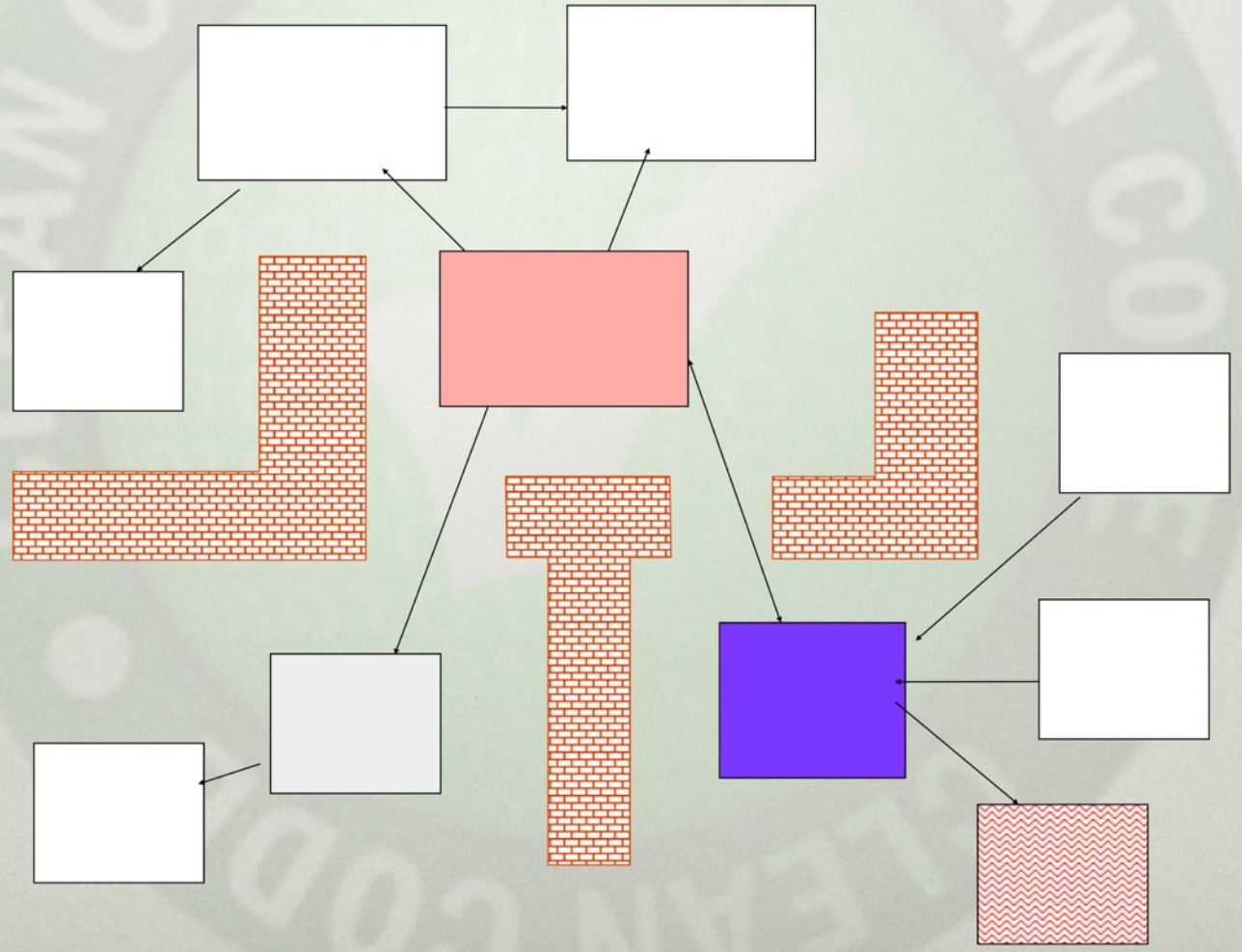
Manage Dependencies





Which design will more easily accept new features?

Separate Concerns



Apply Principles of Class Design

The **S**ingle Responsibility Principle

The **O**pen/Closed Principle

The **L**iskov Substitution Principle

The **I**nterface Segregation Principle

The **D**ependency Inversion Principle

Attributes of a Good Design

- Reveals Intent
  - Provides insights into requirements analysis.
- Adaptable
  - Tolerates evolution of business need.
  - Accepts new technologies without undue cost.

Clean Code

Clean Code Separates Levels of Detail

```
public int deduct(Money amount) {
  if (amount == null ||
       (balance.getAmount() <
        amount.getAmount()))
    return -1;
  else {
    int bal2 = balance.getAmount()*100+.5;
    int amt2 = amount.getAmount()*100+.5;
    balance.setAmount((bal2-amt2)/100.0);
    if (balance.getAmount() == 0.0)
      return 0; // is zero.
  }
  return 1;
}
```

Clean Code Tells a Story

```
public void deduct(Money amount)
    throws InsufficientFundsException {
  if (balance.isLessThan(amount))
    throw new InsufficientFundsException();
  else
    balance = balance.minus(amount);
}
```

Clean Code Needs Few Comments

- Don't allow comments as an excuse to preserve bad code.

```
// The following is messy, so here's what it does...
```

- Prefer small well-named methods to commented sections.

```
// Now deduct taxes
          vs.
deductTaxes();
```

- Don't use comments to explain names.

```
Date d; // birth-date
```

- Don't use comments brainlessly.

```
int i; // i is an integer
i++; // increment i;
```

- Comments become lies.

```
/**
 * @param amt double with precision .01
 * @param int currency code
 * @return -1 if fails
 */
public void deduct(Money amount)
  throws InsufficientFundsException {
   …
}
```

Clean Code Has Intention-Revealing Names

- Has intention-revealing names:
  - `findById(id)`
    - vs. `binarySearch(id)`
    - vs. `find1(id)`
  - `class BookCatalog`
    - vs. `class BkCat`

Clean Code Has Small Methods

- A method should do one thing.
- Its name describes everything it does.
- Not this:

```
public double mean(double[] l) {
    if (l.length == 0)
      return 0.0;
    double sum = 0;
    for (int i = 0; i < l.length; i++) {
      sum += l[i];
    }
    return sum/l.length;
  }
```

Clean Code Has Small Methods

```
public double mean(double[] numbers) {
    if (numbers.length == 0)
      return 0.0;
    return sum(numbers) / numbers.length;
}

private double sum(double[] numbers) {
    double sum = 0;
    for (int i = 0; i < numbers.length; i++) {
      sum += numbers[i];
    }
    return sum;
}
```

Extract Till You Drop

Clean Code has Command/Query Separation

- Functions should either return a value or change state.
- Functions that do both lead to confusion and error.

```
if (setArgument(arg))
      validArgs.add(arg);
```

- It should be:

```
if (argumentValid(arg)) {
      setArgument(arg);
      validArgs.add(arg);
}
```

And Much More...

Professionalism

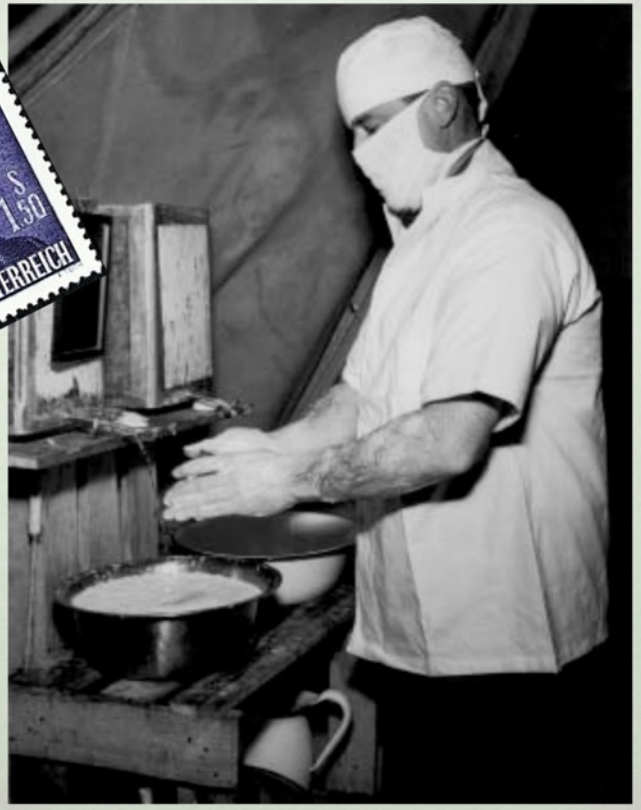
Understandable Code Doesn't Just Happen

- You have to craft it.
- You have to clean it.
- You have to make a professional commitment.

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
– Martin Fowler

But We Don't Have Time!

Professional Responsibility

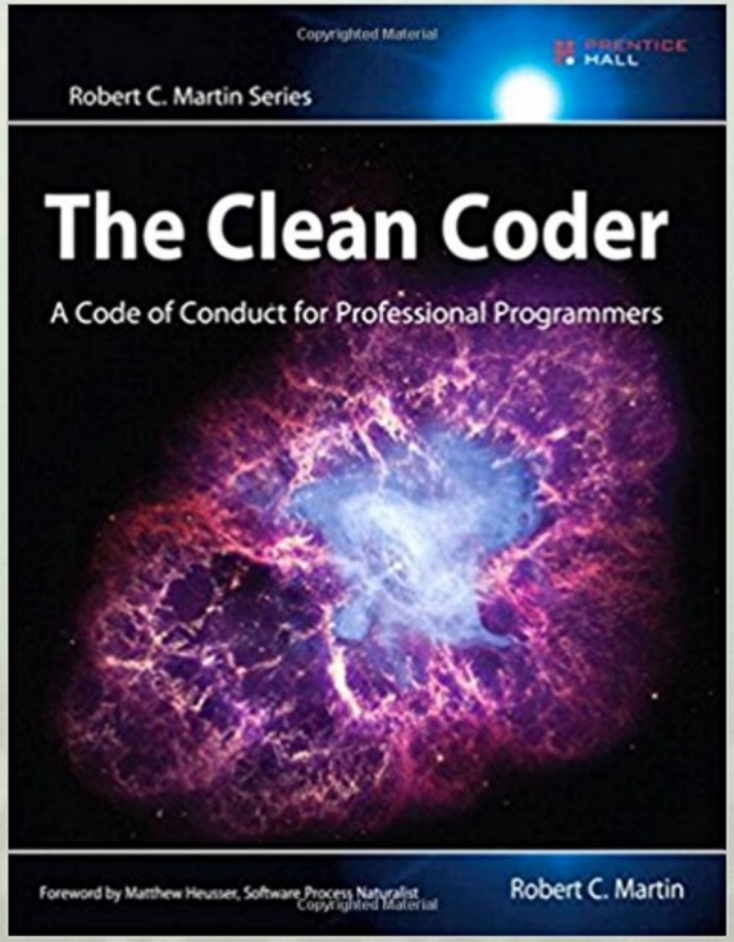


There's no time to wash hands, get to the next patient!

Professionalism

- Make it your responsibility to craft code that:
  - Delivers business value.
  - Is clean.
  - Is tested.
    - 80+% automated test coverage.
  - Is simple.

For more on professionalism:



Test Driven Development

The Three Laws of TDD

- Write no production code unless you have a failing unit test.
- Do not write more of a unit test than is sufficient to fail. (And not compiling is failing.)
- Do not write more production code than is sufficient to pass.

- Rinse and repeat every minute.

The Litany of Benefits

- When we work this way:
  - Debug time drops close to zero.
  - We have formal design documents.
  - Our code is decoupled.
  - We can refactor!
    - Tests eliminate fear.
    - Tests make code flexible.

Parachute

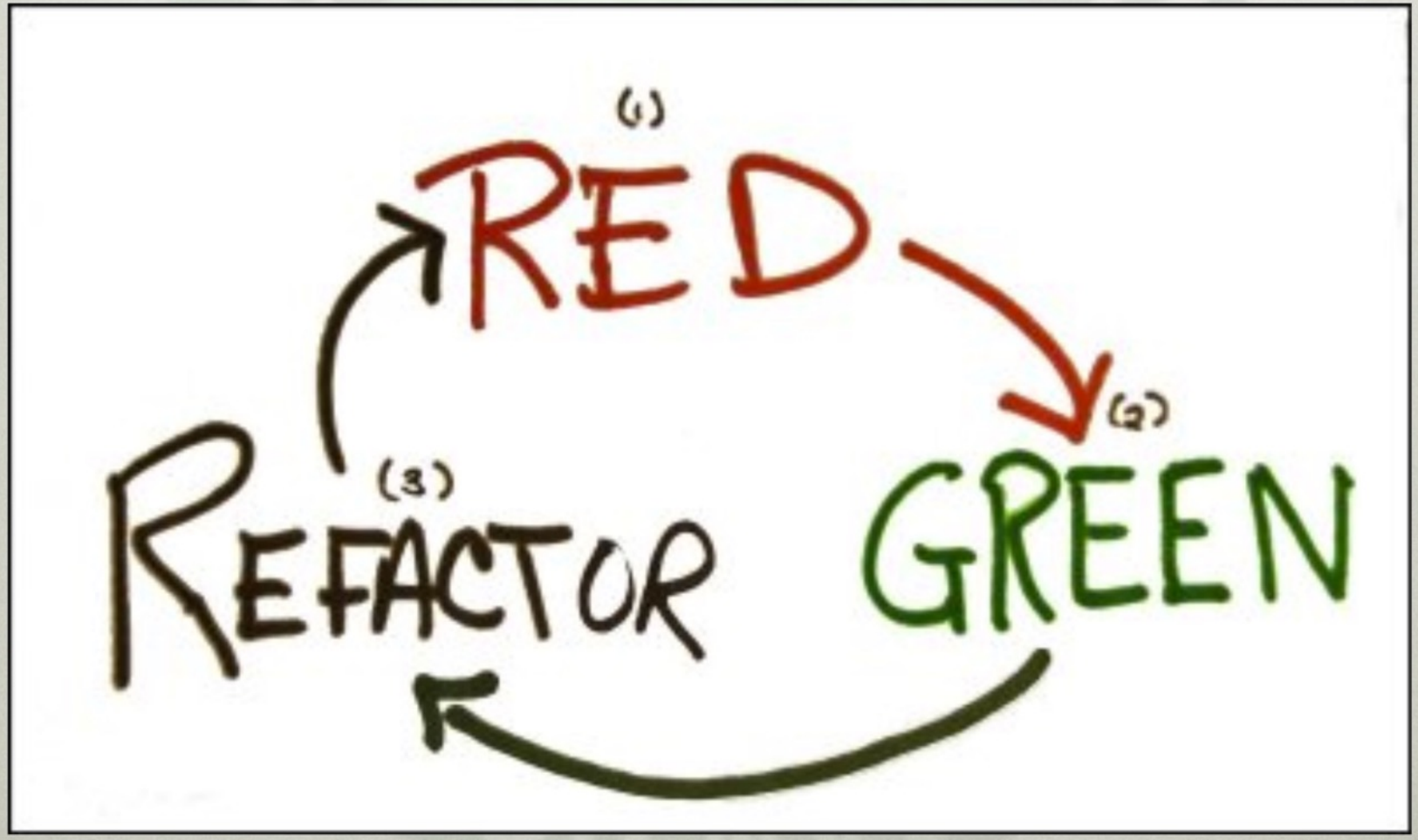
- Tests only work if you can trust your life to them.
- There can't be any holes in the parachute.
- Test-after leaves holes.

Objections Heard (but not believed)

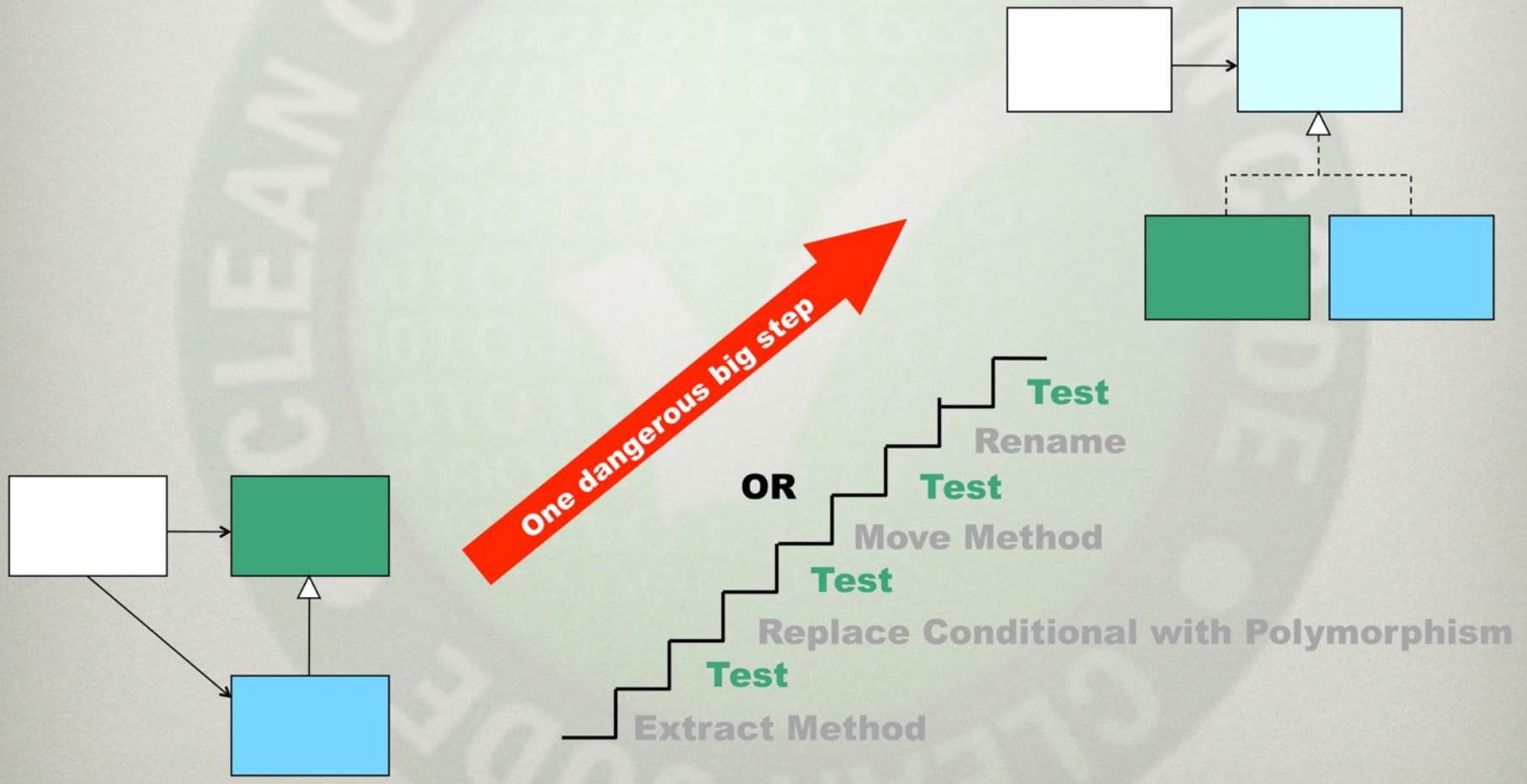
- "I know how to just write the class, but I don't know how to test it."
- "We have to write twice as much code and provide two people to do it."
- "I have to debug twice as much code."
- "We have a testing department."
- "I can test my code after I write it."

DUAL ENTRY BOOKKEEPING

Test Driven Development Cycle



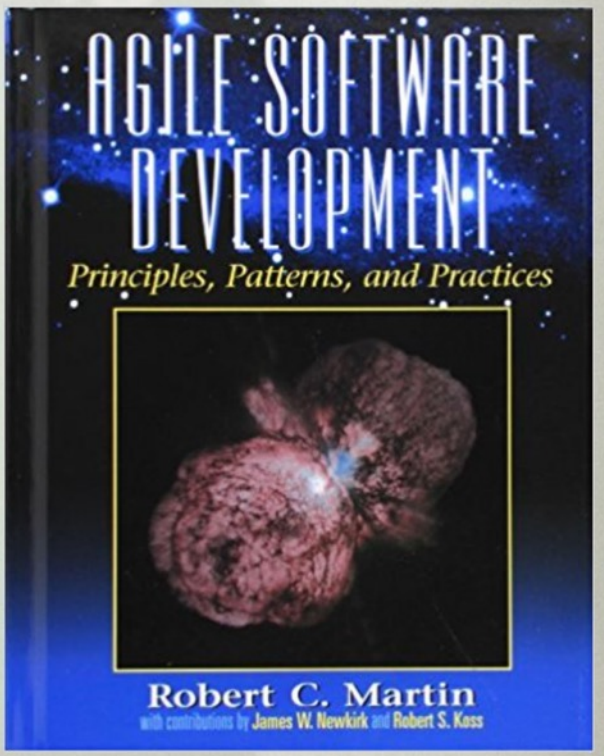
Transform Design with Small Steps



SOLID

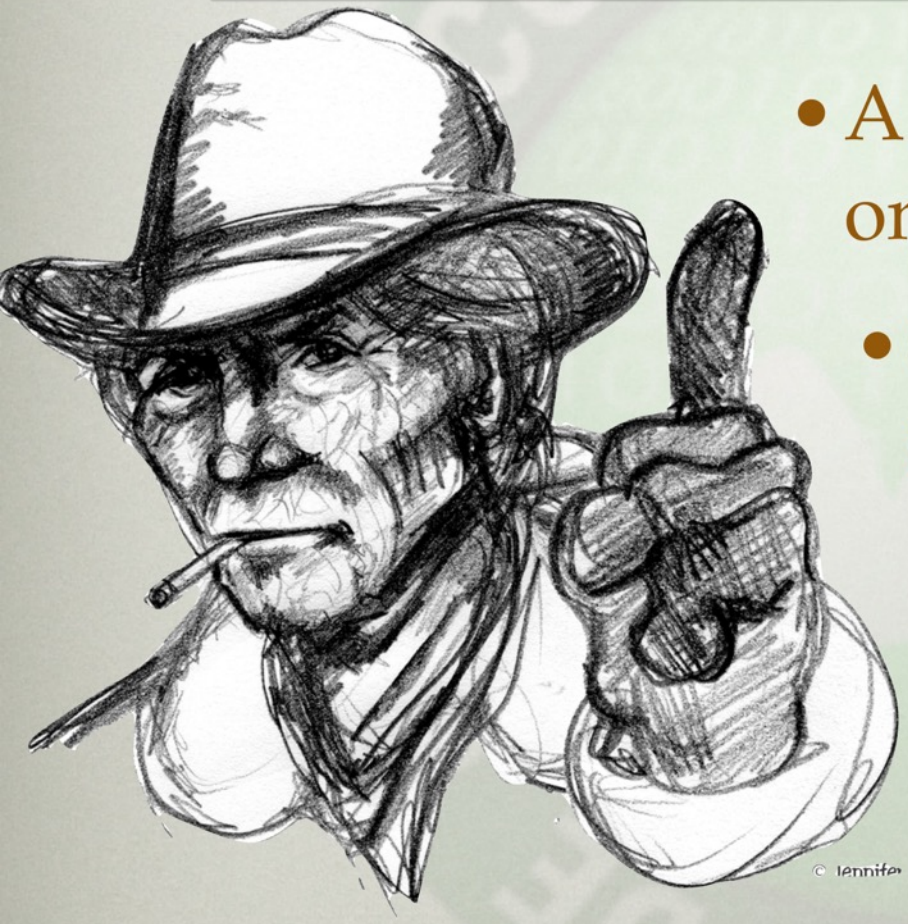
History

- comp.object debates
- James O. Coplien.
  Barbara Liskov.
  Bertrand Meyer.
- NCARB Project.
- Designing Object Oriented C++ Applications using the Booch Method
- Agile Software Development: Principles Patterns and Practices.



(SRP)
The Single Responsibility Principle

Single Responsibility Principle



- A class should have one and only one reason to change.
  - One actor that is the source of that change.

Single Responsibility Principle

- A class should have one and only one reason to change.

Employee
+ calculatePay() + save() + printReport()

- CFO
- CTO
- COO

- How many reasons does **`Employee`** have to change?

CM Collision

1. Tim checks out `Employee` in order to add a new business rule to `calculatePay()`
2. Dean checks out `Employee` in order to change the format of the `EmployeeReport`.
3. They check in at the same time and are forced to do a merge.

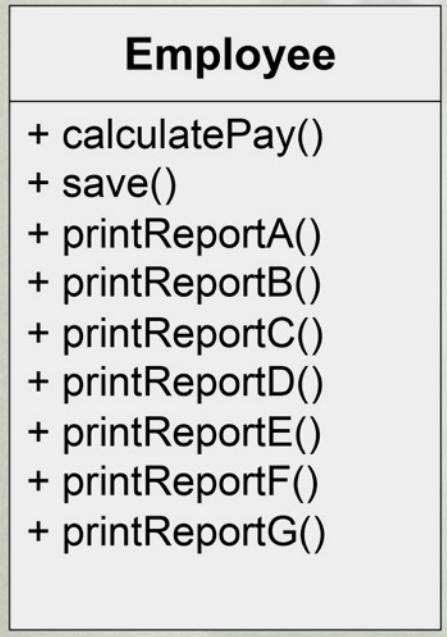
When responsibilities are combined in the same source file,
collisions become more frequent.

Collateral Damage

1. Tim adds a new `InsuranceReport` to `Employee`.
2. He checks out `Employee` and adds the code.
3. The report looks good. Business users are happy.
4. During the next payroll cycle, all paychecks have the wrong amount.

When responsibilities are co-located, inadvertent breakage becomes more likely.

Open Ended Responsibilities



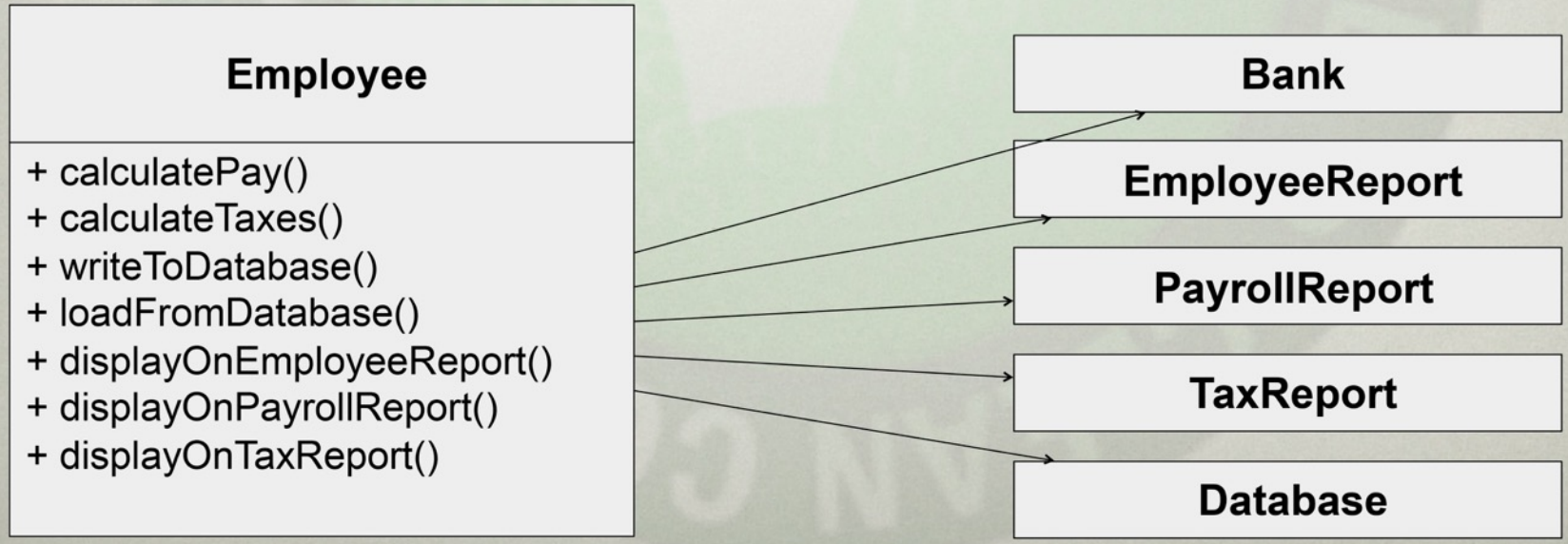
- More and more report types have been created.
- **`Employee`** has become polluted with dozens of methods dealing with reports.

Transitive Impact

- Bob checks out **`Employee`** to make all reports produce XML.
- Later there is a new release of the XML libraries.
- **`Employee`** is changed to match the new API.
- All users of **`Employee`** are impacted.
- Why should all users of employee be impacted by changes to the XML library?

Fan Out

- SRP violations often look like hub and spoke diagrams.
  - Dependencies radiate out from **`Employee`**.
- **`Employee`** is impacted by changes to many other modules.



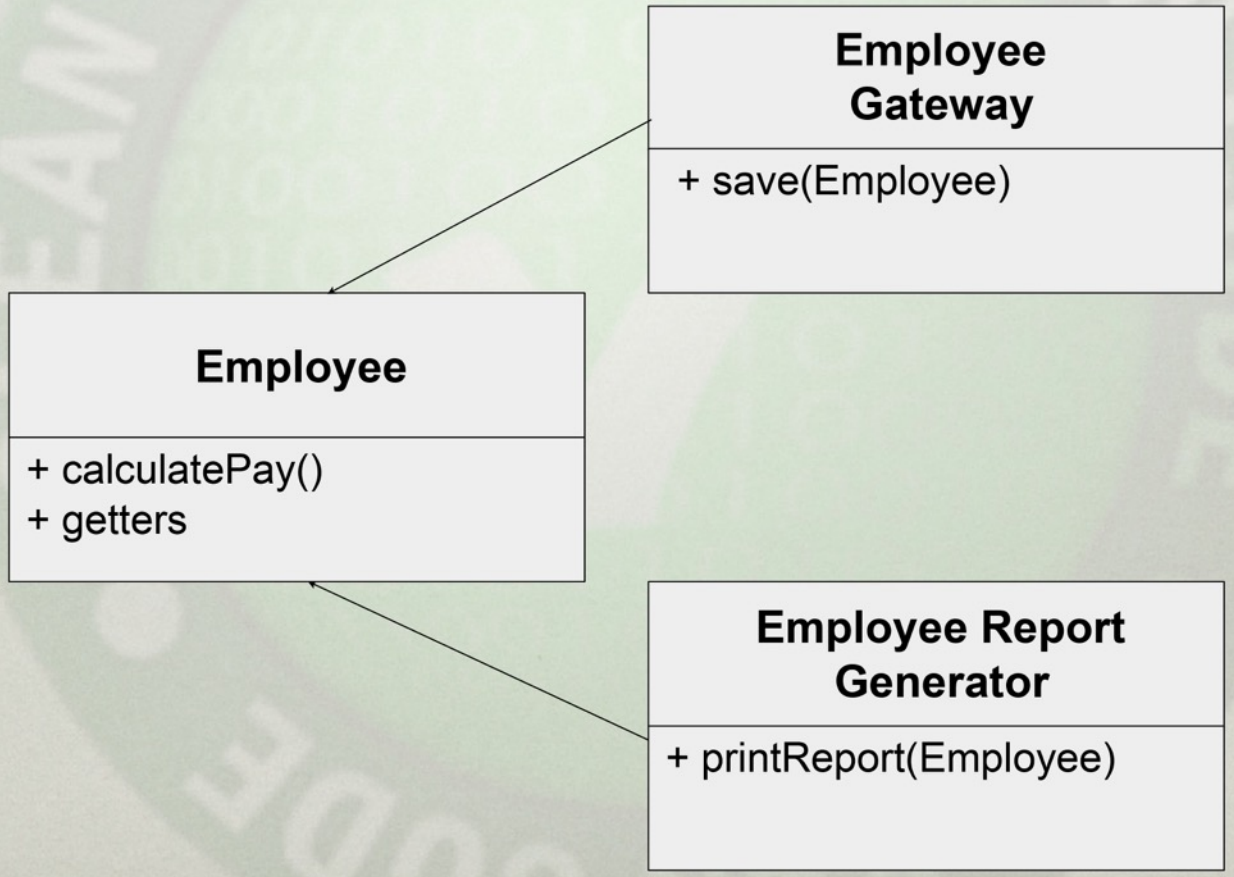
Deployment Thrashing

- James checks out **`Employee`** to change business rules.
  - **`Employee.jar`** must be rebuilt, retested, and redeployed.
- Dean checks out **`Employee`** to change report format.
  - **`Employee.jar`** must be rebuilt,

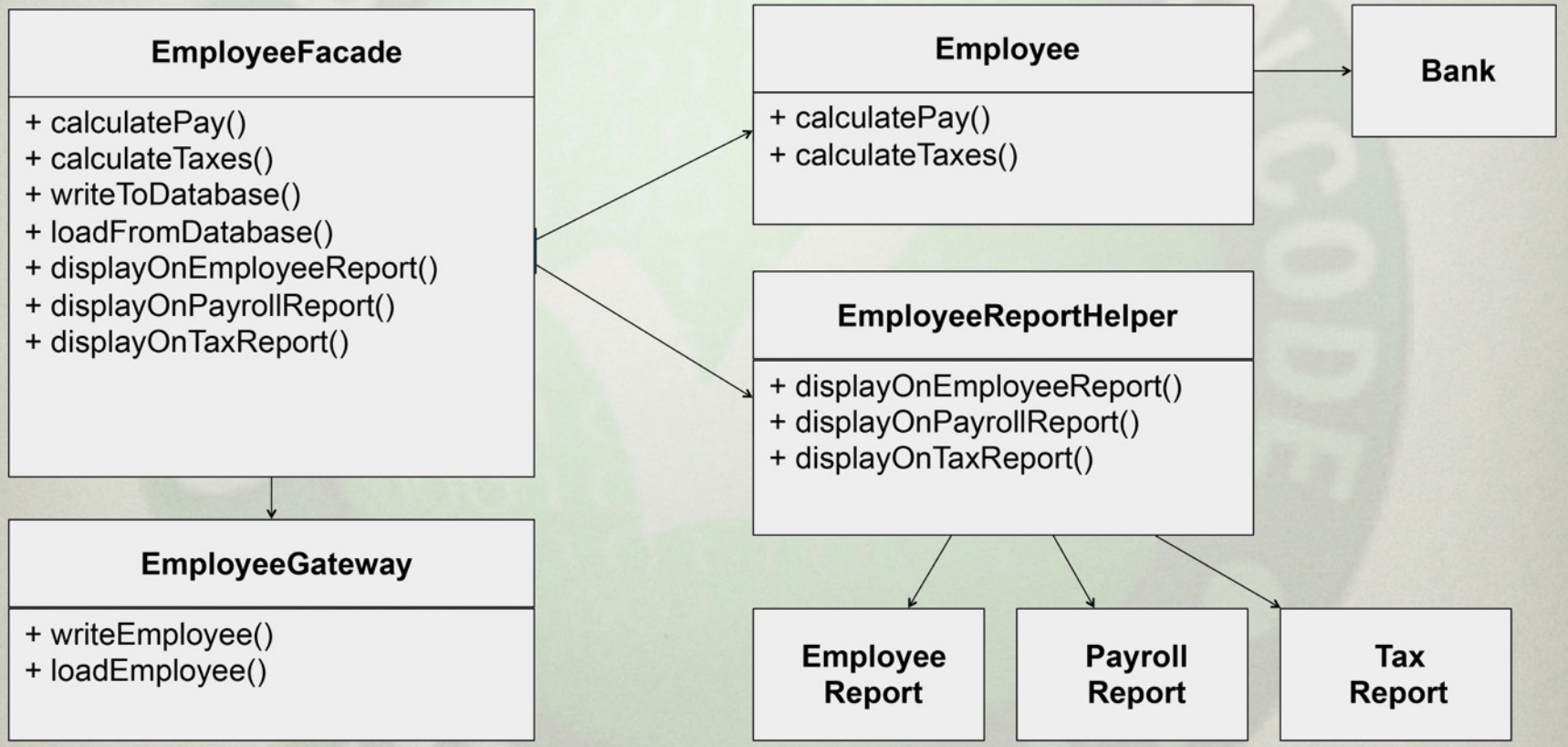
Summary: Symptoms of SRP Violation

- Classes that violate SRP:
  - Are difficult to understand.
  - Change for too many reasons.
  - Are subject to frequent CM collisions and merges.
  - Cause transitive impact.
  - Are subject to deployment thrashing.

Solution: Split Responsibilities



Façade Pattern: A Partial Solution



- Separates implementation
- Preserves interface

What issues remain?

Wrap-up

- The Single Responsibility Principle
  - A class should have one and only one reason to change.
    - Is this true for a method?
    - Is this true for a component?
    - Define responsibility.

(OCP)
The Open-Closed Principle

Open for Extension

- We are able to change or extend the behavior of a system...

Closed for Modification

- …without changing any existing code.

The Open-Closed Principle

- When we change old, working code, we often break it.
- When we must change a module, we must also re-test, re-release, and re-deploy that module.
  - We may not even be allowed to touch an old module!

Change should be Low-Cost and Low-Risk

The Intent is Hidden by the Details (1 of 2)

```
public class Payroll {
  EmployeeGateway employeeGateway;

  public void run(Date payDate) {
    for (Employee employee : employeeGateway.findAll()) {
      boolean payToday = false;
      switch (employee.getPaymentSchedule()) {
        case BIWEEKLY:
          payToday = DateUtils.isOddFriday(payDate);
          break;
        case MONTHLY:
          payToday = DateUtils.isLastDayOfMonth(payDate);
          break;
        case WEEKLY:
          payToday = DateUtils.isFriday(payDate);
          break;
      }
```

The Intent is Hidden by the Details (2 of 2)

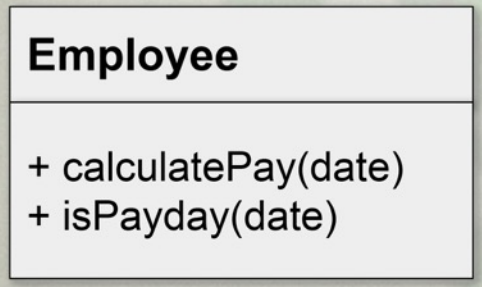
```
if (payToday) {
  Money pay;
  switch (employee.getPayClass()) {
    case SALARIED:
      pay = employee.getSalary();
      break;
    case HOURLY:
      List<TimeCard> timeCards = employee.getTimeCards();
      pay = PayCalculator.calcHourlyPay(payDate, timeCards,
                                        employee.getHourlyRate());
      break;
    case COMMISSIONED:
      List<SalesReceipt> receipts = employee.getSalesReceipts();
      pay = PayCalculator.calcCommissionedPay(
              payDate, receipts, employee.getSalary(),
              employee.getCommissionRate());
      break;
```

Obscuration of Intent

- A good module should read like a story.
- The story can be obscured by:
  - Little decisions.
  - Odd flags.
  - Strange function calls.

Solution: Nominal Flow

```
public class Payroll {
  EmployeeGateway employeeGateway;

  public void run(Date payDate) {
    for (Employee employee : employeeGateway.findAll()) {
      if (employee.isPayday(payDate)) {
        Money pay = employee.calculatePay(payDate);
        ...
      }
    }
  }
}
```



- The Pure Truth
- The intent is exposed.

The Code Tells the Story

- This code tells it's own story.
- It doesn't need comments.
- There are few obscuring details.
- The `Employee` object has hidden most details behind an abstract interface.
- This is what Object Oriented Design is all about.

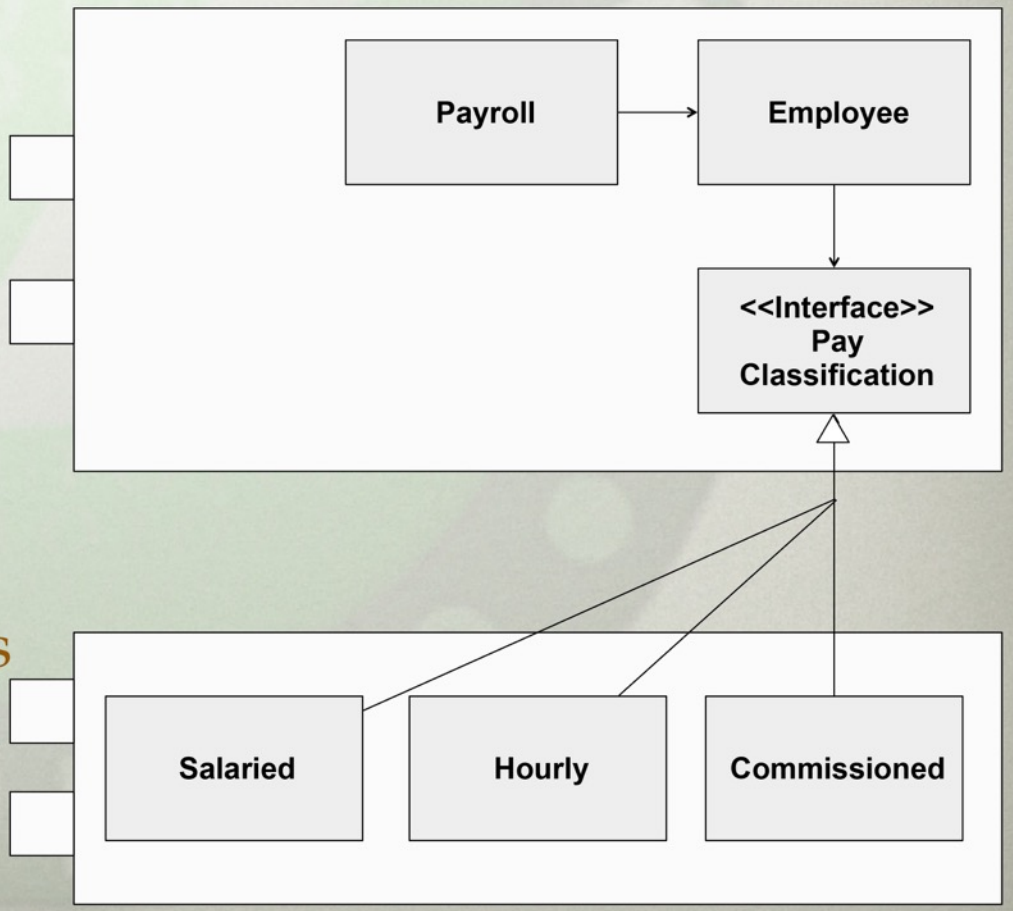
Symptom: Switching on Type

```
switch (employee.getEmployeeType()) {
  case COMMISSIONED:
    payToday = DateUtils.isOddFriday(payDate);
    break;
  case SALARIED:
    payToday = DateUtils.isLastDayOfMonth(payDate);
    break;
  case HOURLY:
    payToday = DateUtils.isFriday(payDate);
    break;
}
```

- Switch statements are repeated across many clients of **`Employee`**.
  - **Rigidity**
- Each will be slightly different and those differences must be located and understood.
  - **Fragility**
- Every module that has a type case depends on all the cases.
  - **Immobility**

Solution: Abstraction and Polymorphism

- New **`Pay-Classifications`** can be added without modifying any existing code.
  - **Not Rigid**
- Don't have to search through code looking for switch statements.
  - **Not Fragile**
- **`Payroll`** can be deployed with as many or as few **`Pay-Classifications`** as desired.
  - **Mobile**



Misuse: Endless Abstraction

Notes on Endless Abstraction

- OCP is not an excuse for doing more than is necessary.
- Abstraction is expensive.
  - Use it when there is an imminent risk of pain.
  - In general prove (or at least demonstrate) to yourself that it will be worth the cost.

What should you abstract?

- Things that are going to change.
- How do you know what will change?
- You could think real hard.
  - To predict the future…

Previous Change Predicts Future Change

- Create abstractions when changes occur.
  - JITA - Just In Time Abstraction
  - “The second guy pays for abstraction”.
- Release early and often.
  - This subjects your systems to forces of change.
    - Users.

Wrap-up

- We want to reduce the cost of change.
- By creating appropriate abstractions, we can produce modules that are open for extension but closed for modification.
- This should not be overdone because abstractions are expensive.
- Working in short cycles with lots of tests and frequent releases helps us decide the best abstractions.

(LSP)
The Liskov Substitution Principle

Liskov Substitution Principle

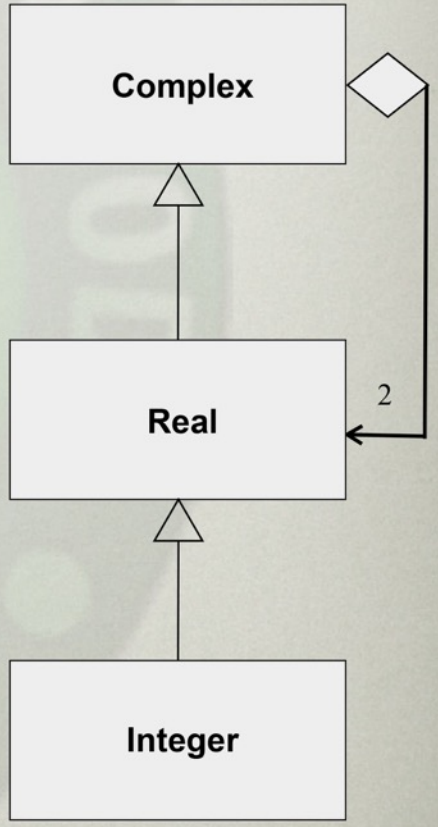
- Instances of a derived class must be usable through the interface of its base class without clients of the base class being able to tell the difference.
- Subtypes must be substitutable for their base types.

Liskov Substitution Principle

- Inheritance is a relationship between:
  - A base class and its derived classes.
  - Each derived class and all clients of its base class.
- A derived class must respond to the same messages that its base class responds to.
  - The response can be different, but it must be consistent with the expectations of clients of the base class.
- The Liskov Substitution Principle
  - Instances of a derived class must be usable through the interface of its base class without clients of the base class being able to tell the difference.
  - Subtypes must be substitutable for their base types.

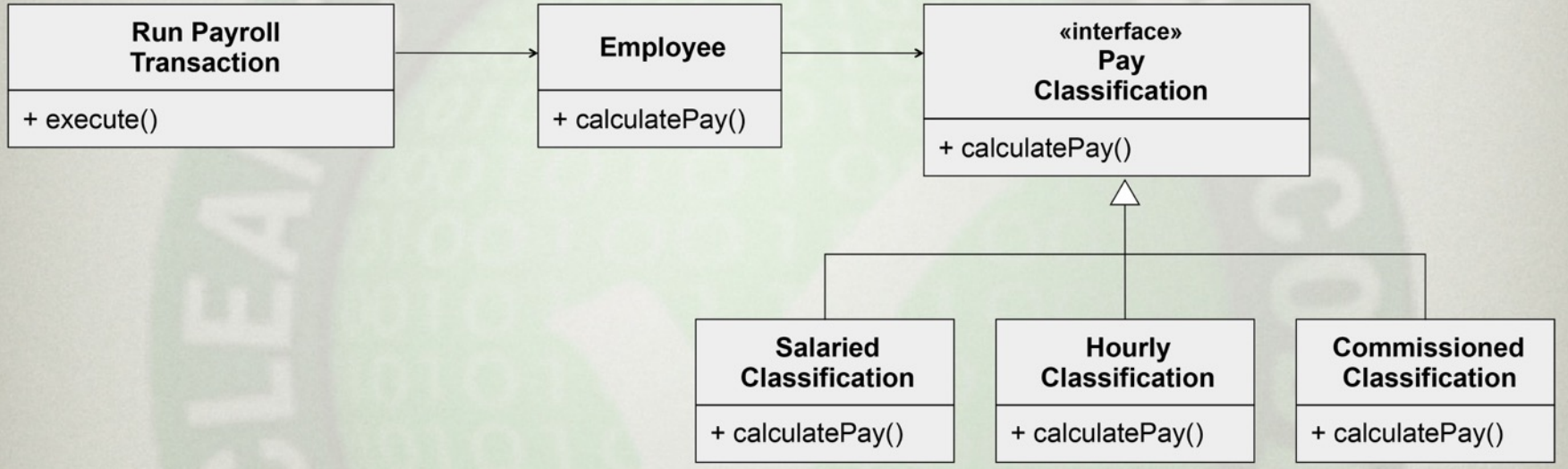
Inheritance != IS-A

- **Real** Number IS-A **Complex** Number.
- **Integer** IS-A **Real** Number.
- **Complex** Number HAS-A **Real** Number.
- Is a **Real** a type of **Complex**?
  - It is mathematically.
- Does a **Real** BEHAVE like a **Complex**?
  - real.setImaginary(n)???
- Substitutability in software is about **how** something behaves, not about **what**



Inheritance == BEHAVES-AS-A

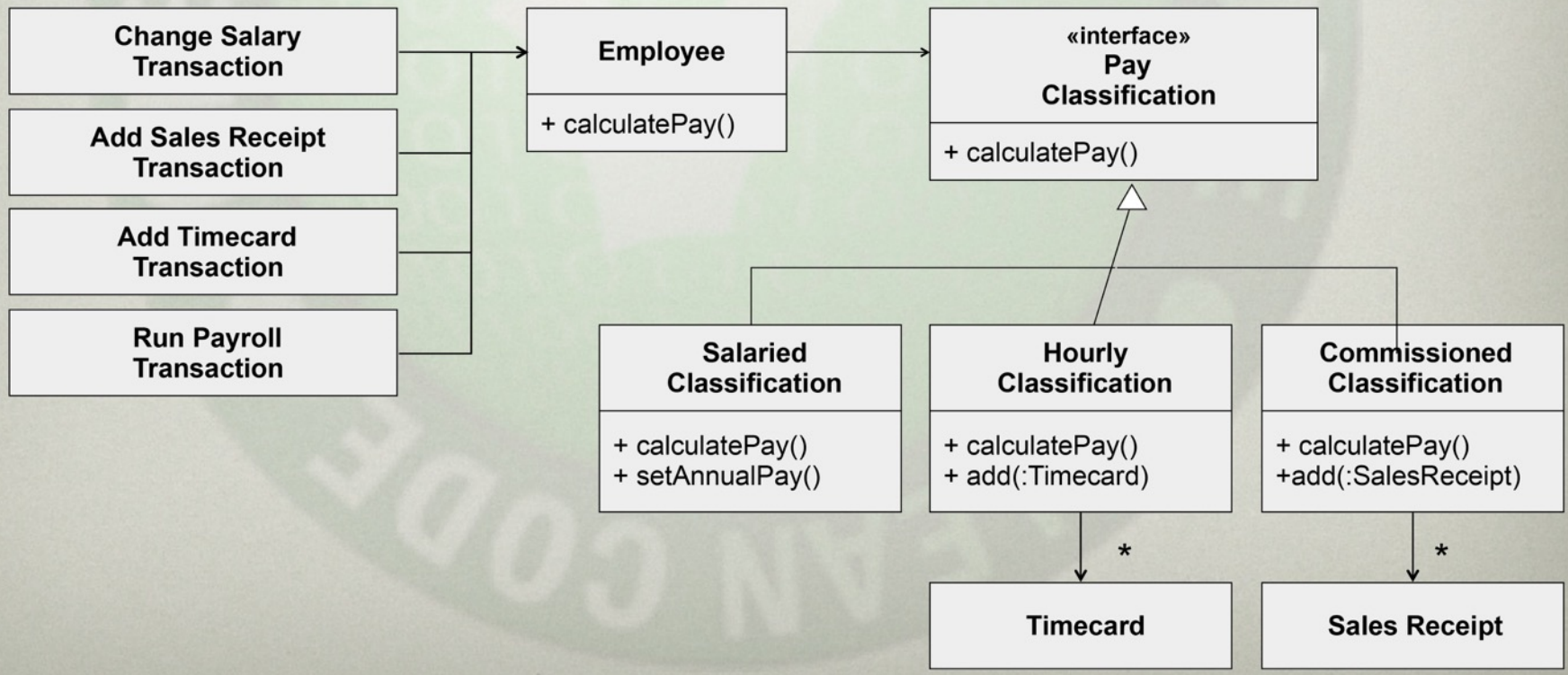
Example



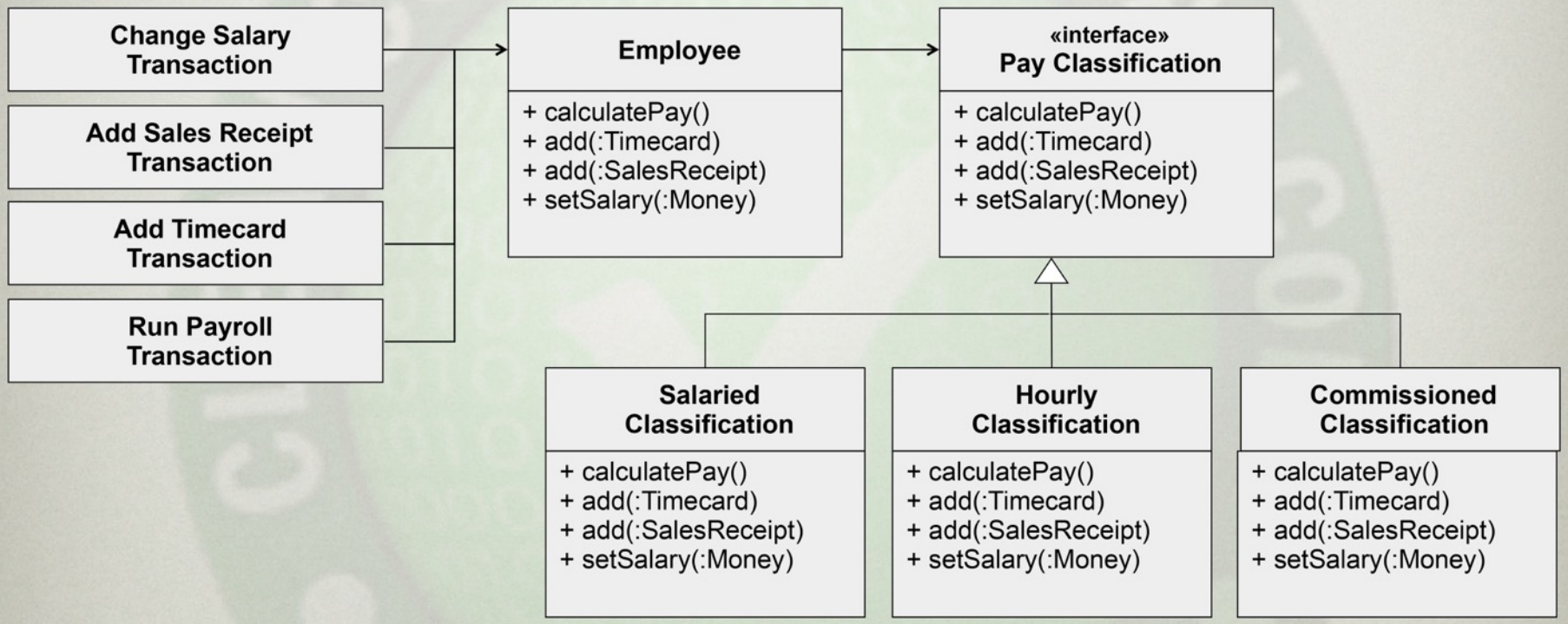
No apparent OCP or LSP violations.

Problem: No Access to Detail

How does the `AddSalesReceiptTransaction` add `SalesReceipts` to the `CommissionedClassification`?



Violates Substitutability



- *Refused Bequest [Fowler]*
  - **`SalariedClassification`** can not accept a **`TimeCard`**.

LSP Violations Cause OCP Violations

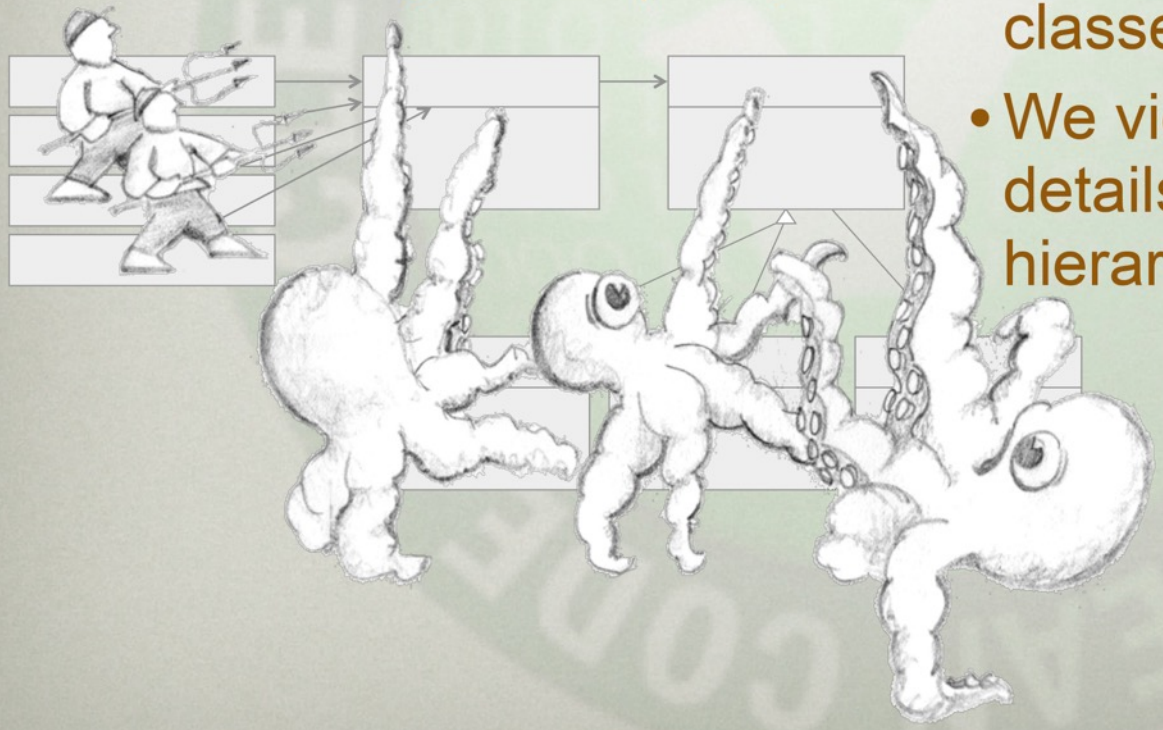
- **`Employee`** is not closed to modification.
- In order to add a new **`PayClassification`**:
  - **`PayClassification`** interface must change.
  - All **`PayClassification`** derivatives must change.
  - **`Employee`** must change.
  - Clients of **`Employee`** are affected. Client jars must be:
    - Rebuilt.
    - Re-tested.
    - Re-released.
    - Redeployed.

LSP Violations Cause Latent OCP Violations

- New reports about **`TimeCards`** and **`SalesReceipts`**:
  - Require getters on **`Employee`** and **`PayClassification`**.
    - **`getTimeCards();`**
    - **`getSalesReceipts();`**
  - Require switching on type.

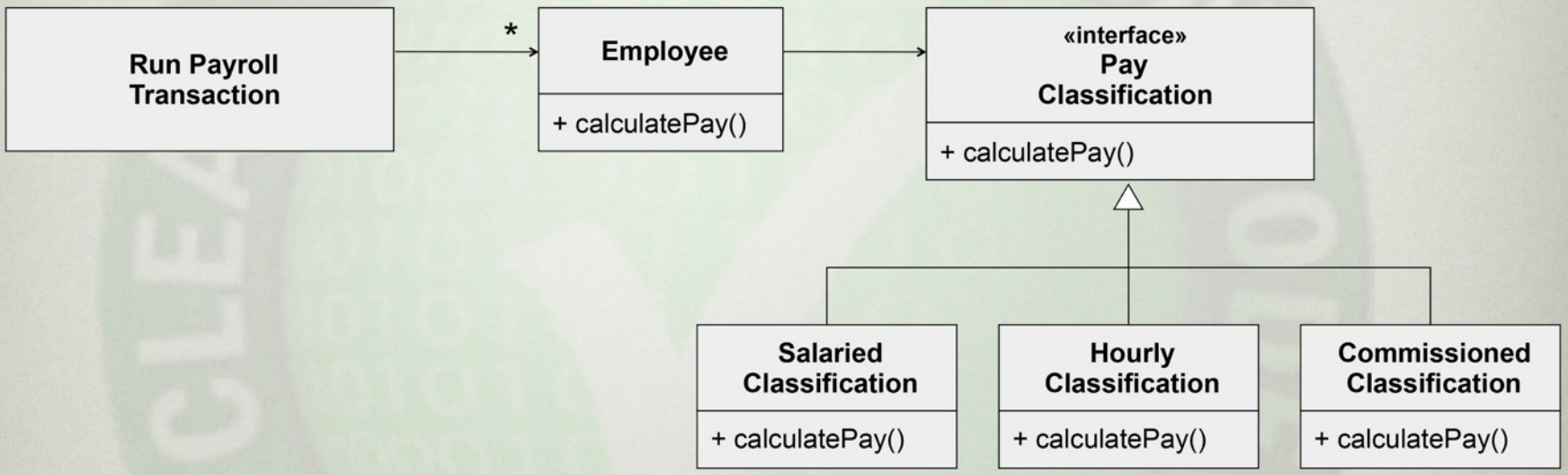
Symptoms of LSP Violations

- Abstract methods do not apply to all implementations.
- Lower level details become visible in high level policy.



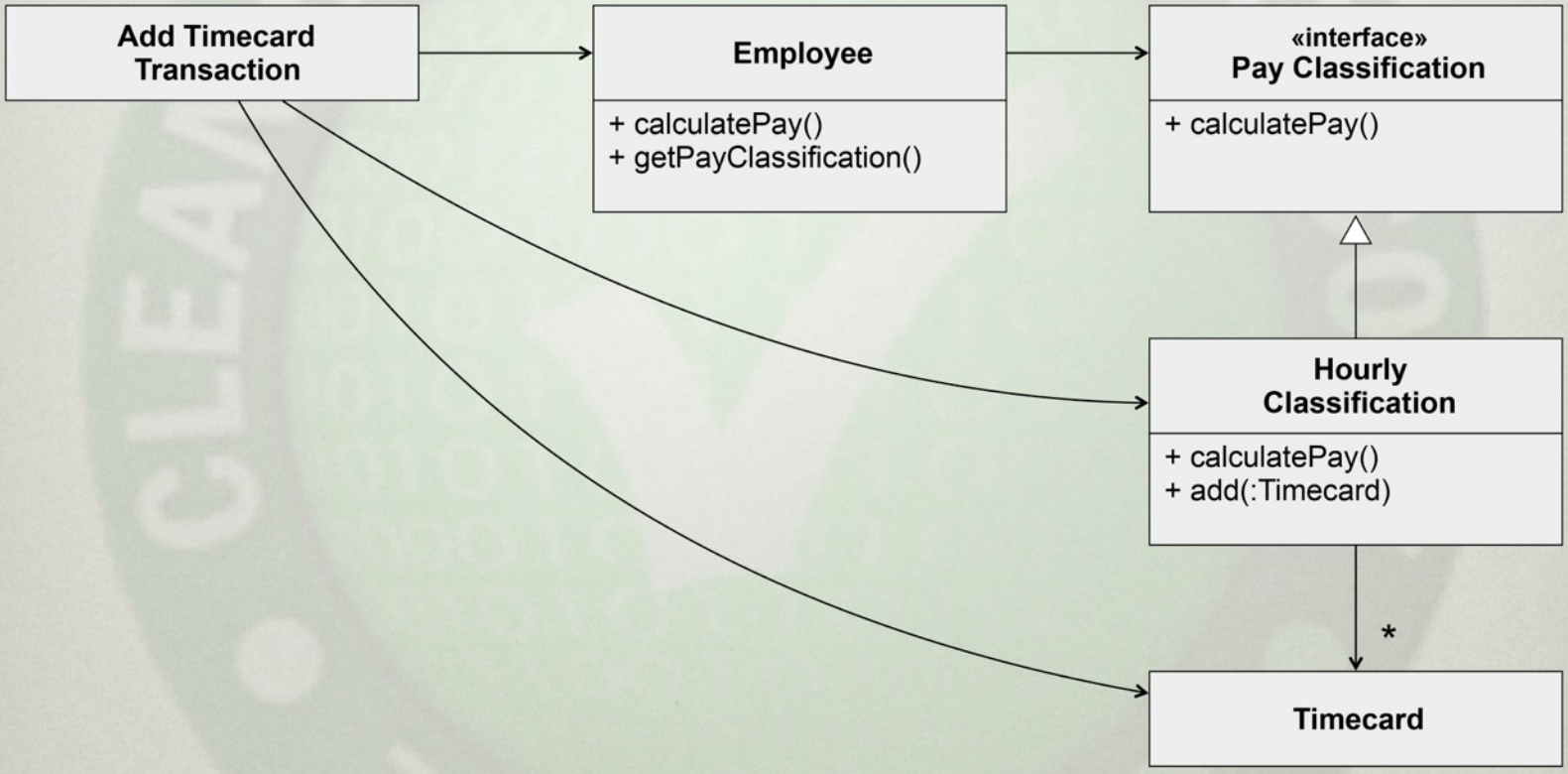
- Base classes need to know about their sub classes.
- Clients of base classes need to know about the sub classes.
- We violate OCP when we let details creep up the hierarchy.

Solution: Separate Needs of the Many...



- `Payroll` can ask any `Employee` to `calculatePay()` regardless of its `PayClassification`.

... From the Needs of the Few



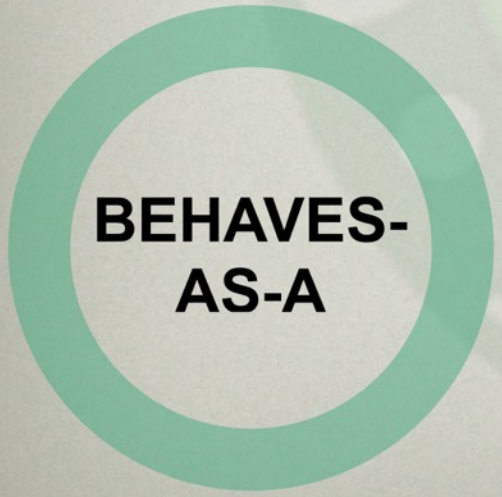
- `AddTimecardTransaction` will only add `TimeCards` to `Employees` with `HourlyClassification`.

Implementation

```
public class AddTimeCardTransaction implements Transaction {
  Employee employee;
  Timecard timecard;

  public AddTimeCardTransaction(Employee employee, Timecard timecard) {
    this.employee = employee;
    this.timecard = timecard;
  }

  public void execute() {
    PayClassification payClass = employee.getPayClassification();

    try {
      ((HourlyPayClassification)payClass).addTimecard(timecard);
    } catch (ClassCastException e) {
      throw new NotHourlyEmployeeException(employee);
    }
  }
}
```

- Is this an LSP violation?
- How about OCP?

Wrap Up



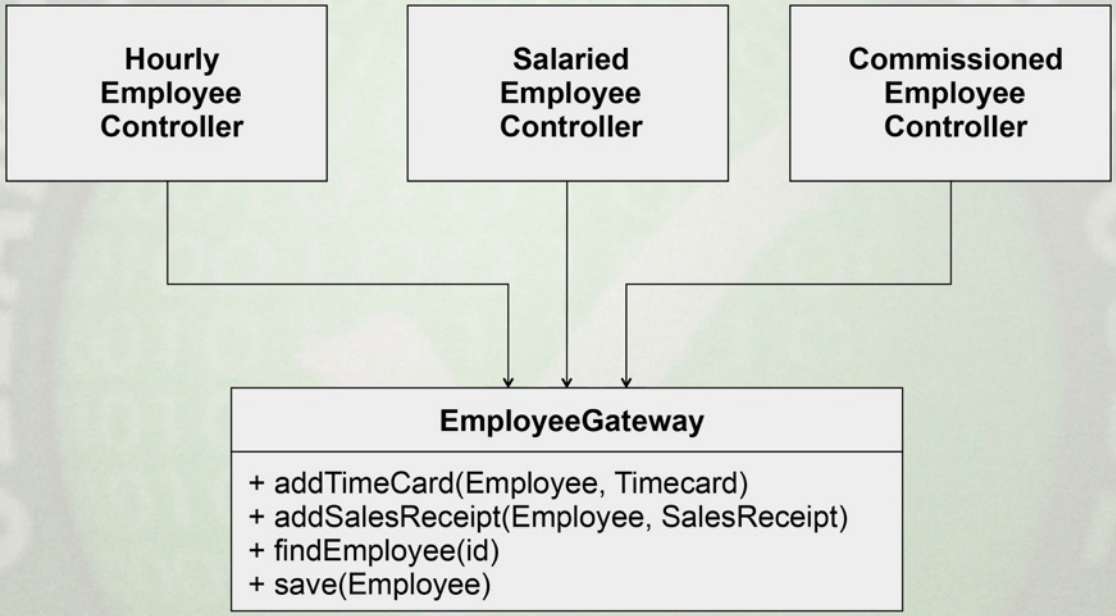
- LSP is a prime enabler of OCP.
  - Substitutability supports extensibility without modification.
- Polymorphism demands substitutability.
- Derived classes must uphold the contract between their base class and its clients!

(ISP)
The Interface Segregation Principle

The Interface Segregation Principle

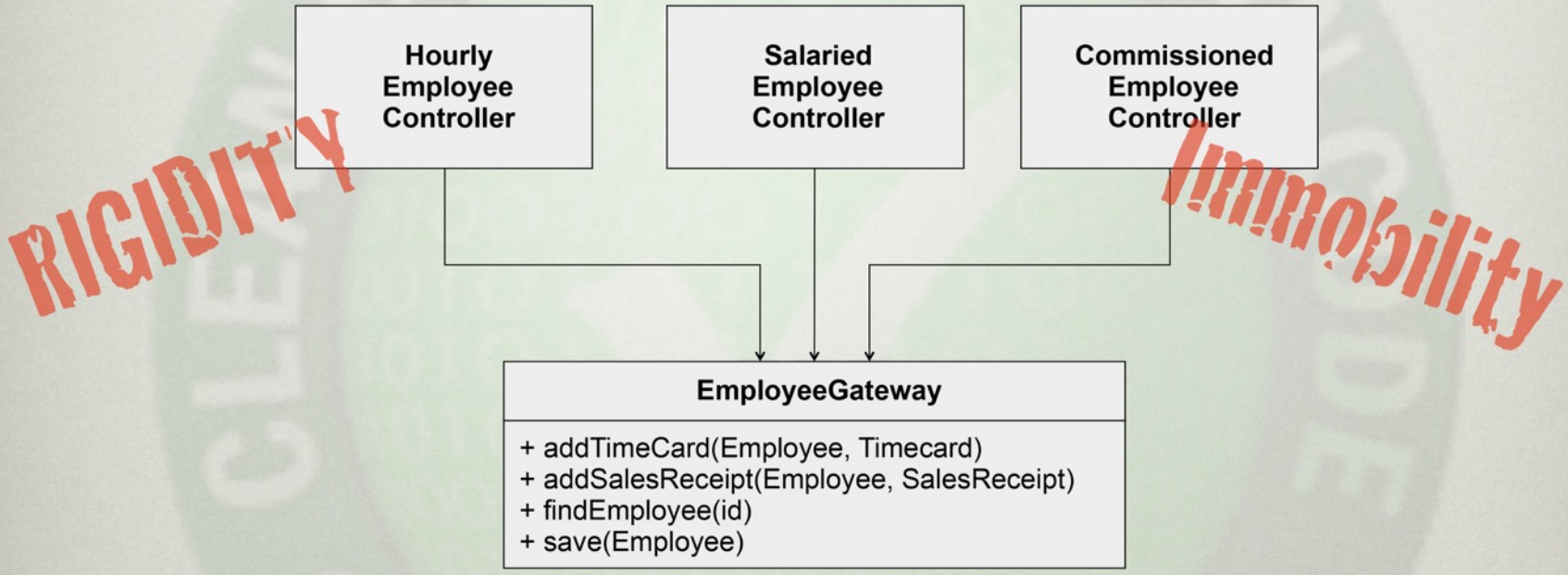
- Clients should depend only on methods they call.

Fat Interfaces



- Methods are grouped for ease of implementation.
- There are more methods than any one client needs.

Problem: Phantom Dependency

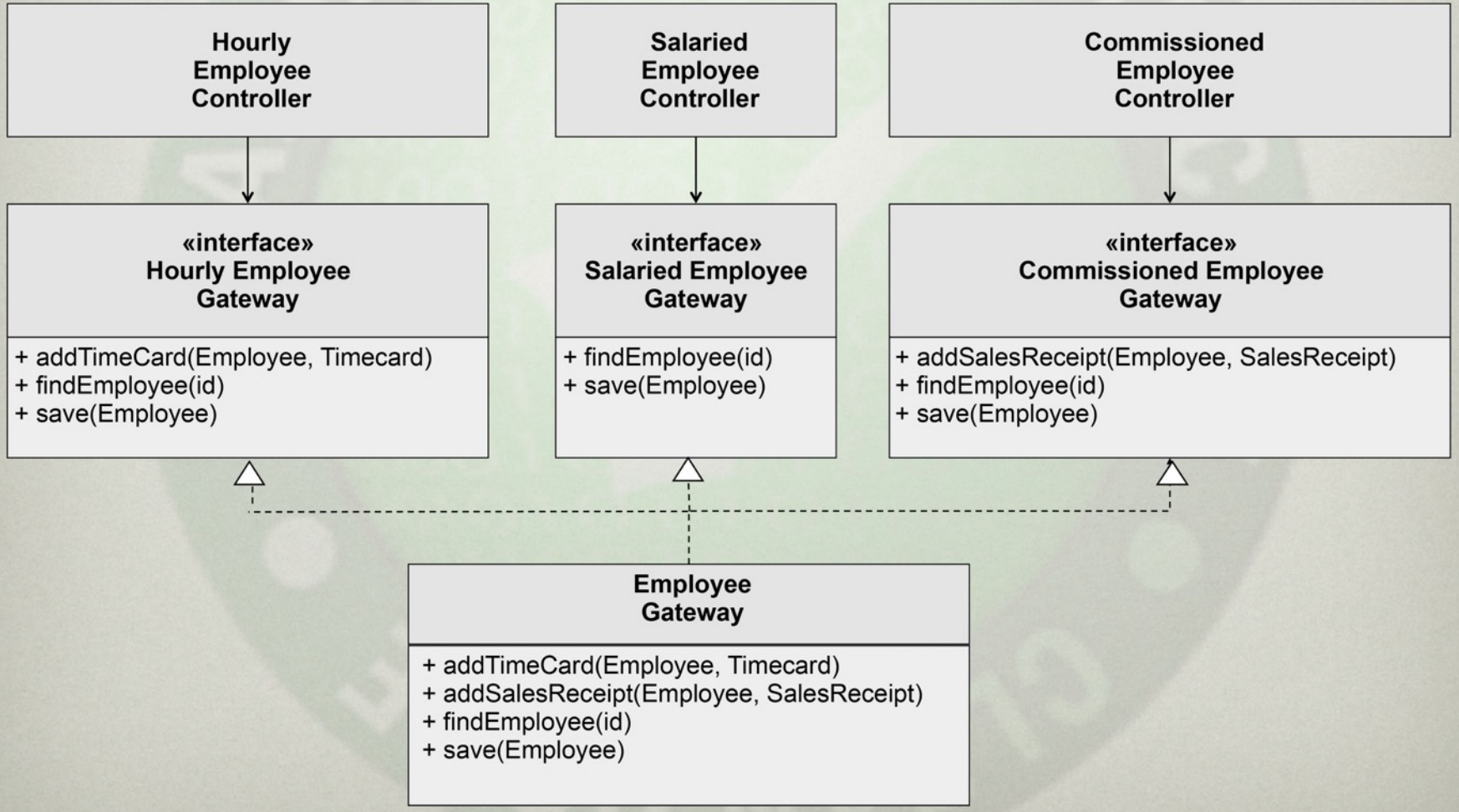


- A change on behalf of one client impacts all clients.
- A new client impacts all existing clients.

Splitting Fat Interfaces

- In general an interface should be small.
- Sometimes a class with a fat interface is difficult to split.
  - The good news is that you can fake it.

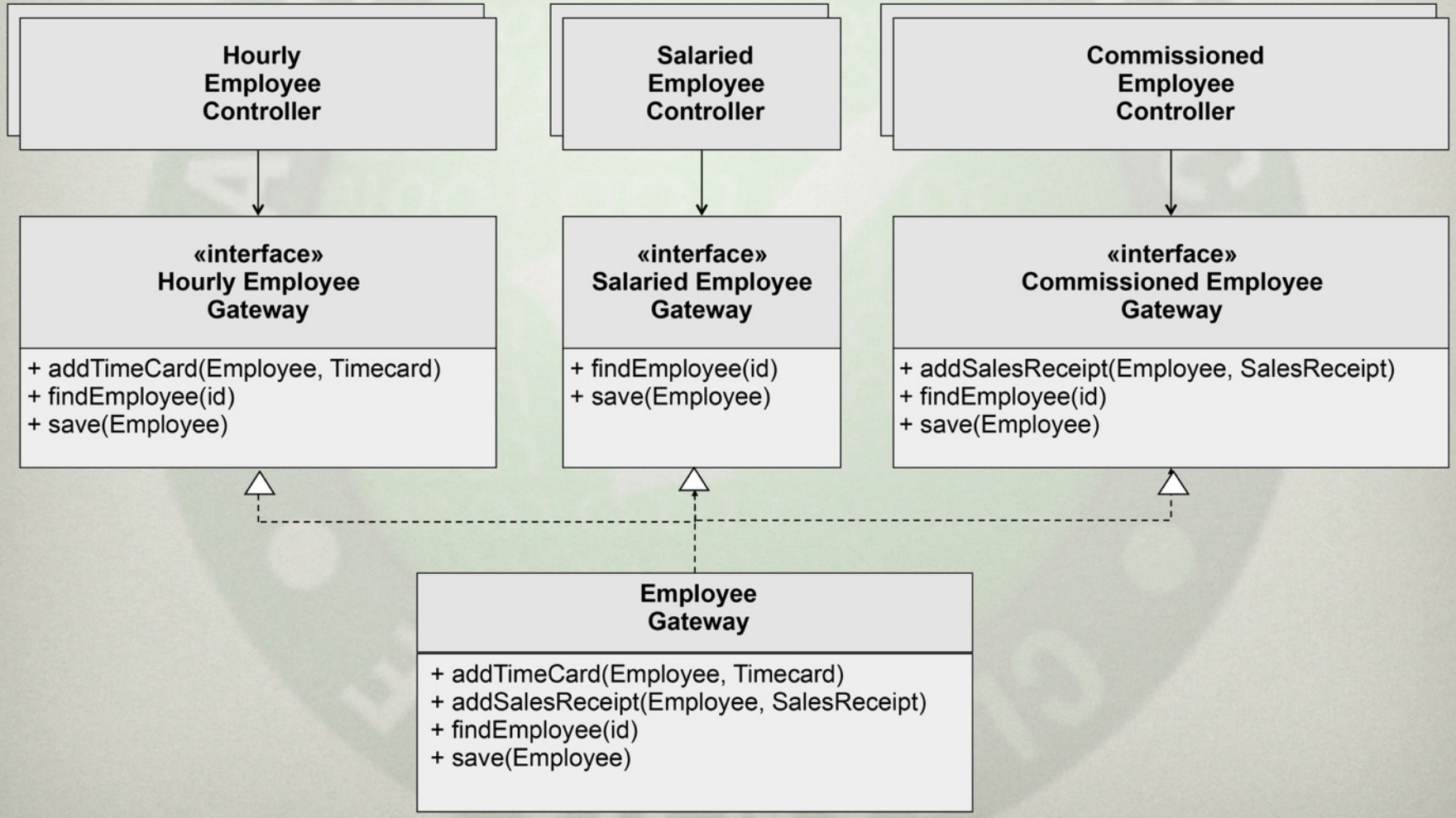
Interface Segregation Principle



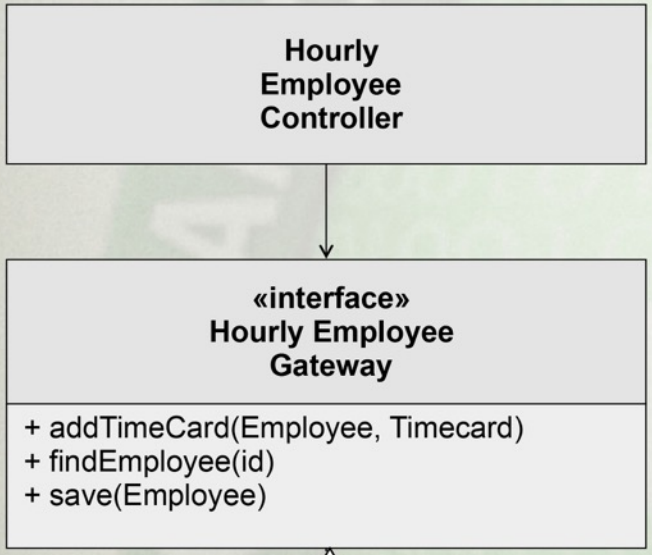
Notes on ISP

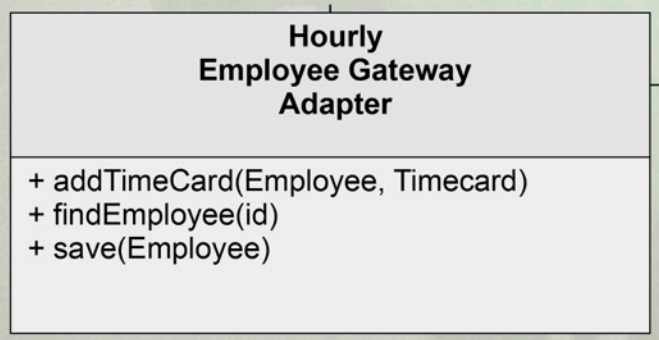
- **EmployeeGateway** still has a fat interface.
  - But clients don't know this because all dependencies point AWAY from it.
- Phantom dependency has been eliminated.
  - Changing existing interfaces or adding new interfaces do not impact other interfaces, and therefore do not impact other clients.

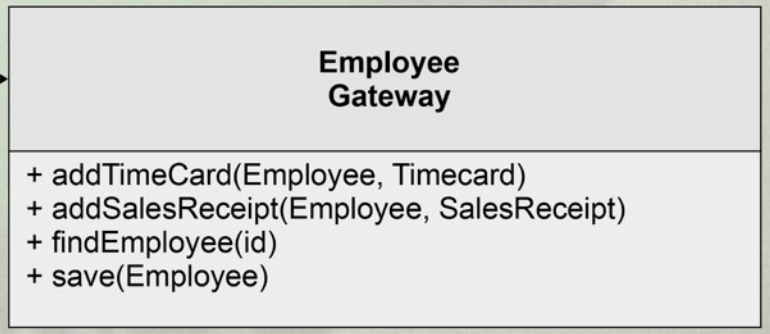
Client Groupings



When You Can't Change the Fat interface







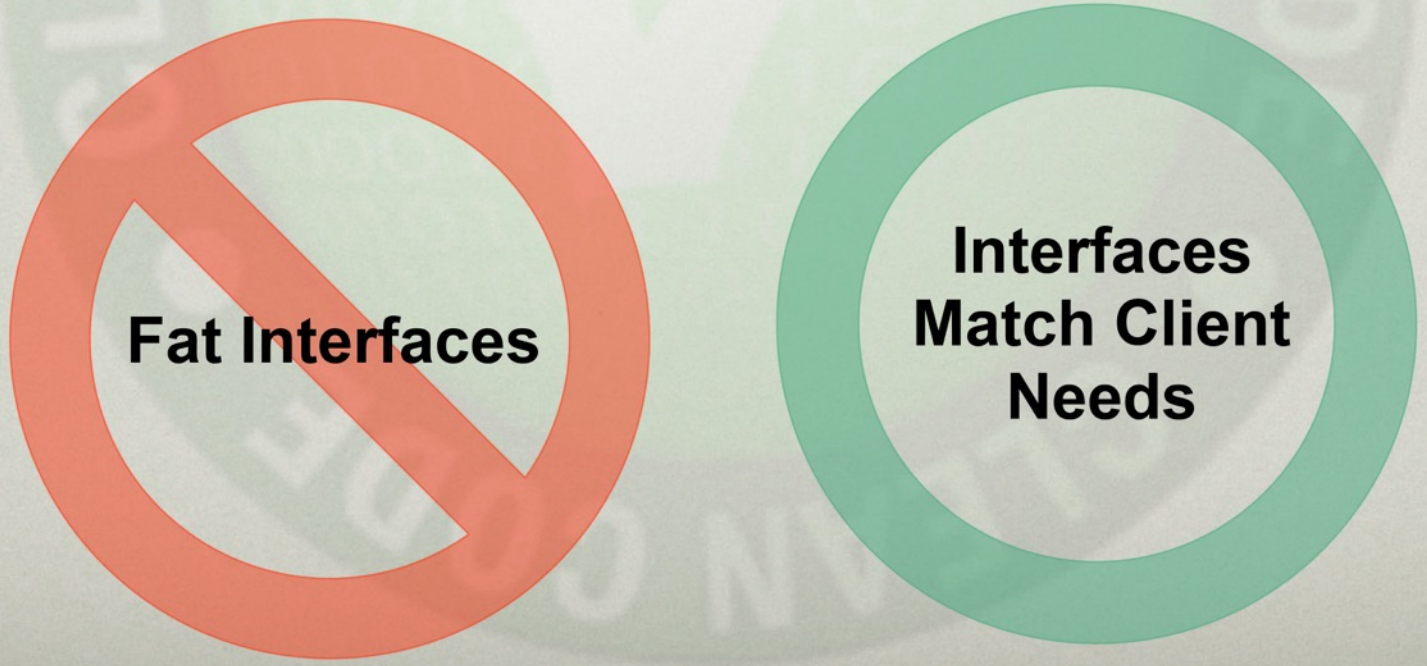
- Use the Adapter Pattern [GOF].

Resist Anticipatory Segregation

- Segregate when there is **Rigidity** and / or **Immobility** due to phantom dependencies.
  - Or at least a clear and present threat.
- Clients define interfaces!
  - Create interfaces that clients need.
  - Don't create interfaces because you think they are "right".

Wrap Up

- Fat Interfaces make clients depend on classes they do not care about.
- Segregate interfaces so that they satisfy client needs.



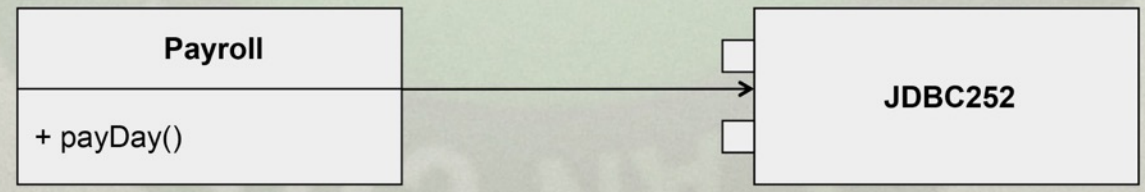
(DIP)
The Dependency Inversion Principle

The Dependency Inversion Principle

- Low level details should depend on high level policy.
- High level policy should be independent.

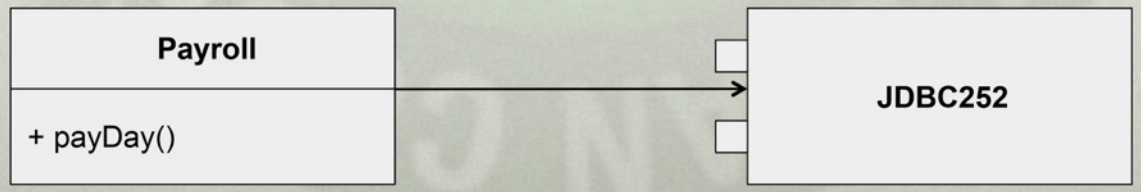
A Procedural Design

```
statement = connection.createStatement();
resultSet = statement.executeQuery("select * from employees");
while(resultSet.next()) {
 switch(resultSet.getInt(3)) {
    case 1: //employee is exempt
      calculatePayForSalaried(resultSet);
      break;
    case 2: //employee is non-exempt
      calculatePayForHourly(resultSet);
      break;
    case 3: //better get a cup of coffee...
  }
}
```



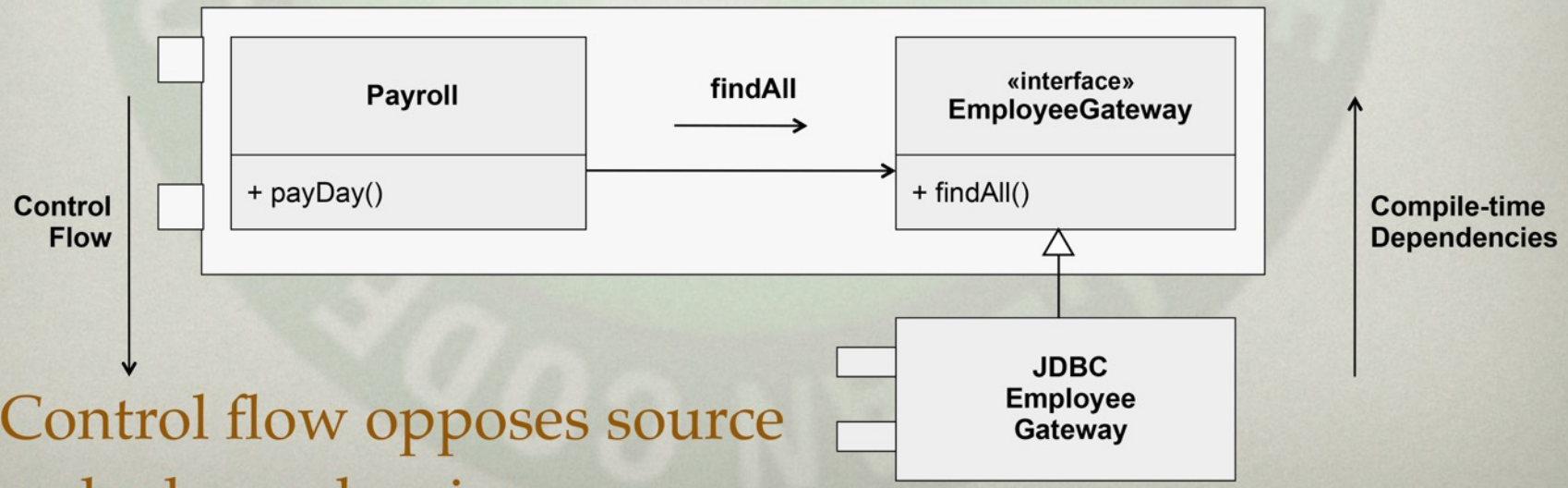
Problems with Procedural Designs

- How do we test **`Payroll`**?
- How do we test the clients of **`Payroll`**?
- How do we prevent **`Payroll`** from being redeployed when the SQL changes?
- How do we get this code to tell its story?



OO Design

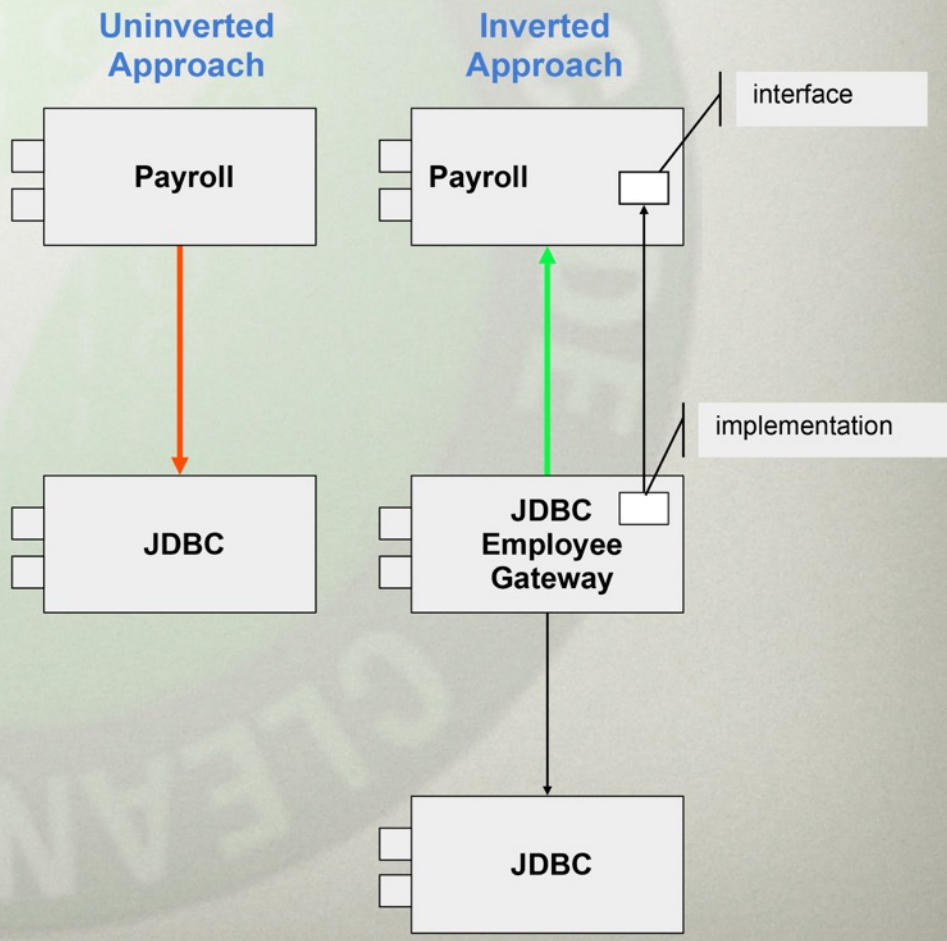
```
public class Payroll {
  private EmployeeGateway employeeGateway;

  public void payDay(Date payDate) {
    for (Employee employee : employeeGateway.findAll()) {
      employee.calculatePay(payDate);
    }
  }
}
```



- Control flow opposes source code dependencies.

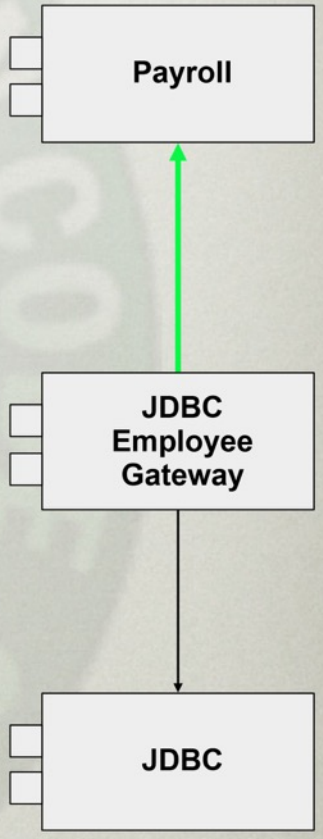
Inverting a Problematic Dependency

- The procedural case has a problematic dependency.
  - **`Payroll`** depends on **`JDBC`**
- The OO case fixes that dependency.
  - Making **`Payroll`** independent of **`JDBC`**.



Solutions Provided by DIP

- We can test **`Payroll`** and its clients by using a test version of the **`EmployeeGateway`**.
- SQL changes are confined to the **`JDBCEmployeeGateway`** which can be deployed independently of **`Payroll`** and its clients.
- This code tells its story unpolluted by implementation details.



Dependency Inversion Heuristics

- High level policies should not depend on details.
  - Details obscure and pollute policy.
- High level policies should depend on abstractions so that:
  - They can be tested.
  - They can be reused.
  - They tell a story.

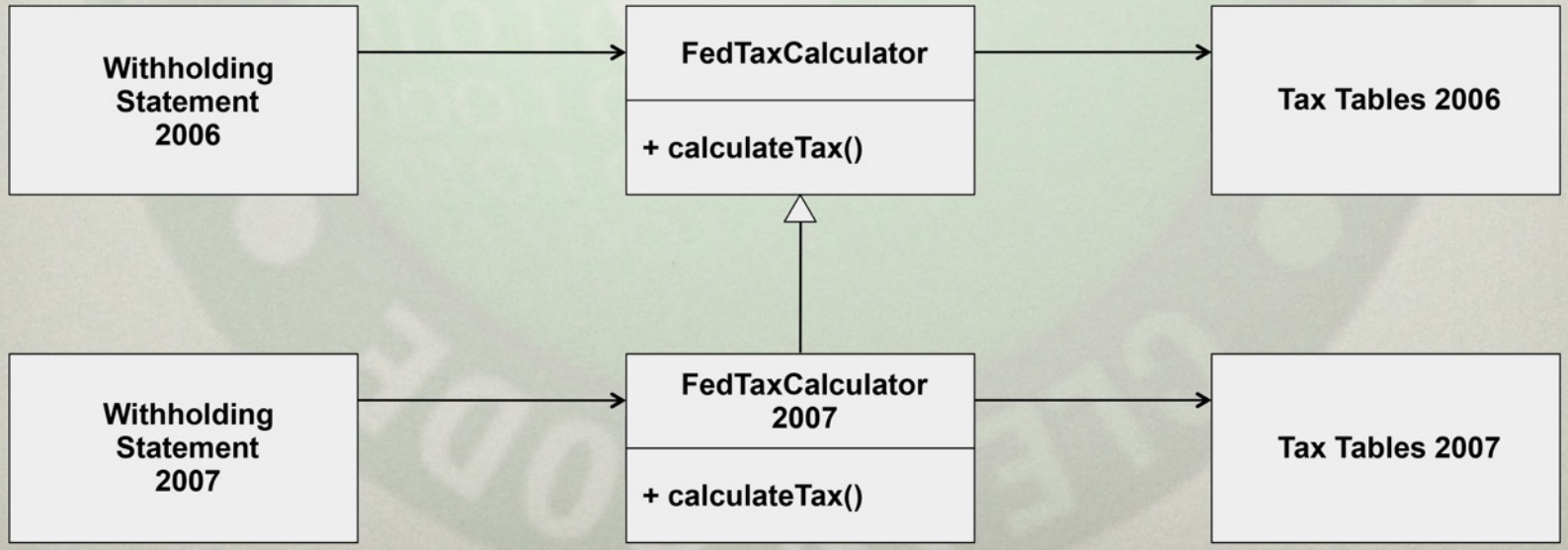
Depend on Abstractions

- Prefer deriving from abstract classes or interfaces.
  - Over deriving from concrete classes.

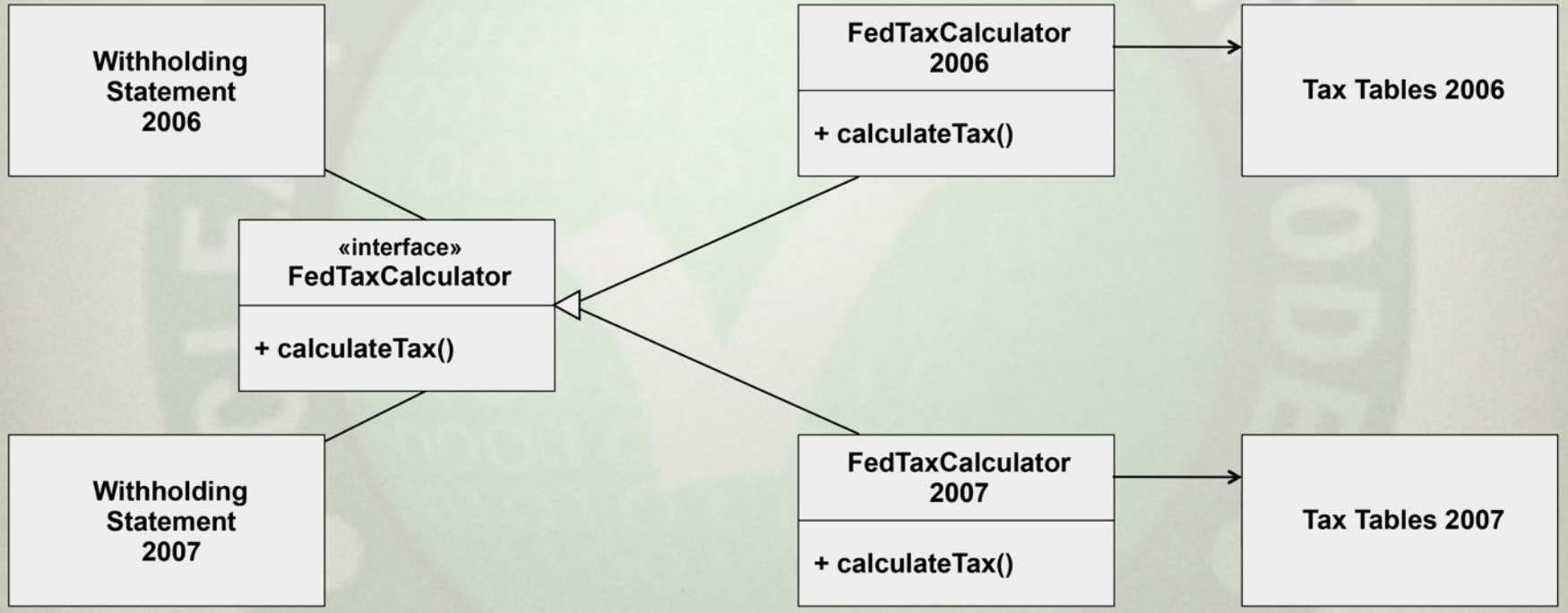
- Prefer overriding abstract methods or interface methods.
  - Over overriding implemented

Overriding Implemented Methods

- **`WithholdingStatement2007`** depends on **`TaxTables2006`** even though it does not:
  - Call it, need it, or want it.



Solution: Factor out an Interface

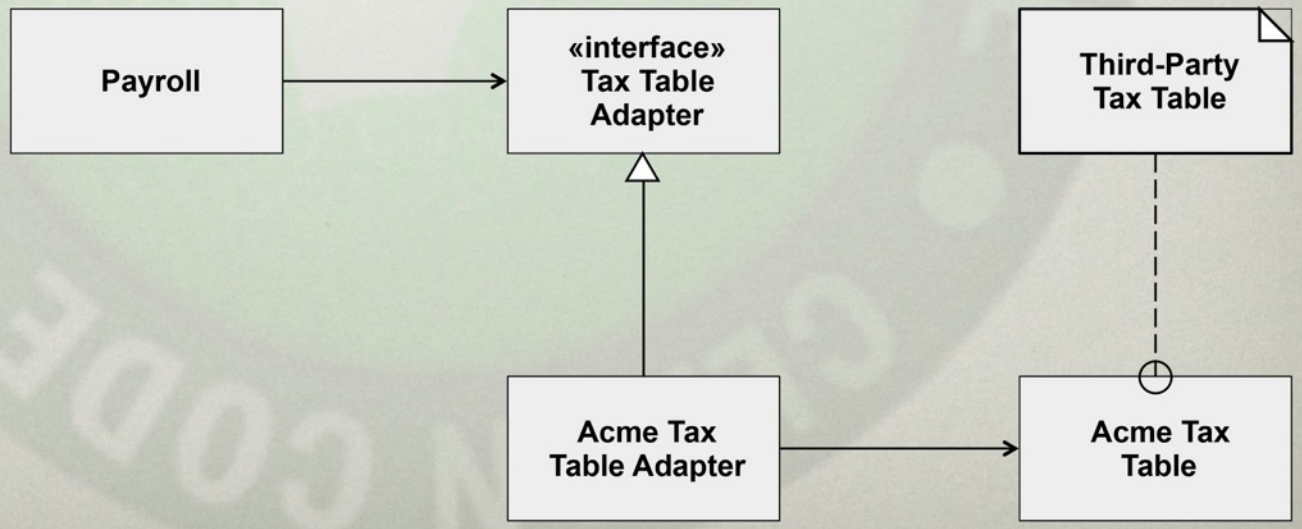


Depending On Concrete Classes

- Sometimes there is no problem.
  - String, HashMap, etc.
    - They are non-volatile.
    - They are dependency dead-ends.
- Sometimes there is no choice.
  - Third party packages
  - Concrete classes that are heavily used.
    - You can't change them to add interfaces.
  - Keyword: new.

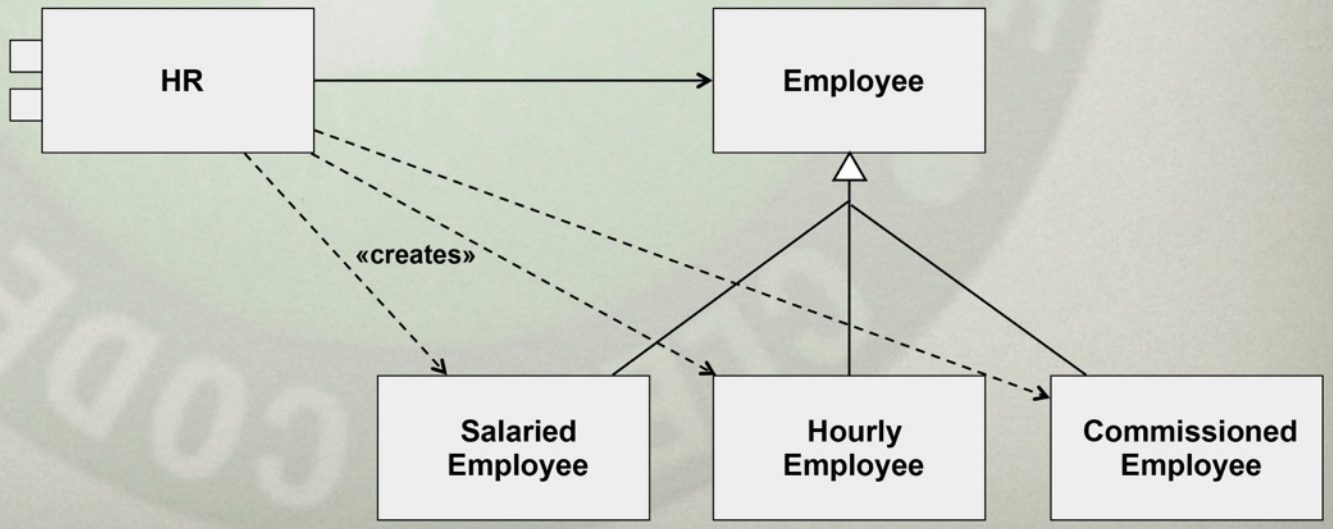
Solution: Adapter [GOF]

- Limits the dependency on classes that are hard to change.
- Lets you define the ideal interface for your needs.



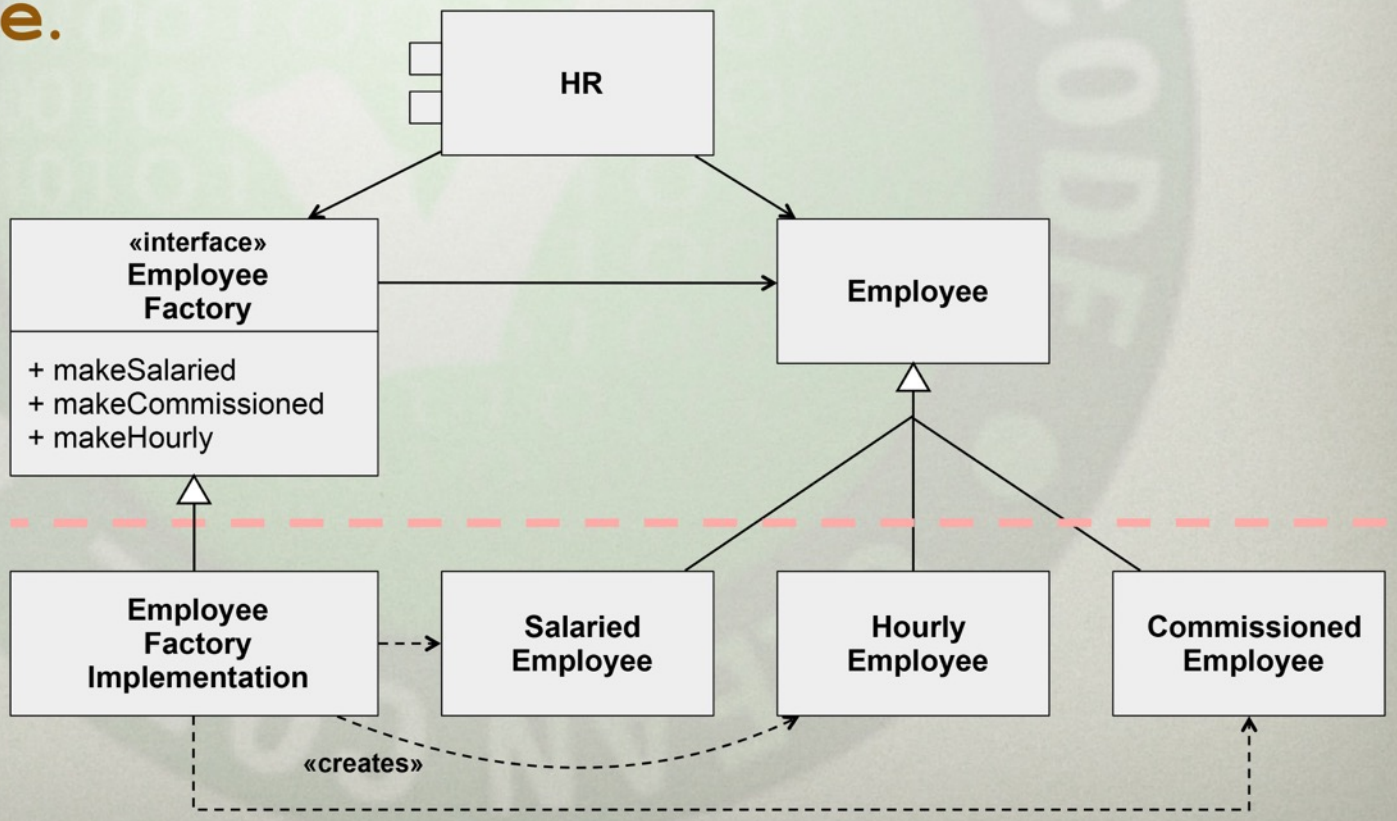
Keyword new: The Problem of Creation

- **`HR`** depends upon all derivatives of **`Employee`**.
  - But only because it creates them.
  - It only calls methods in **`Employee`**.



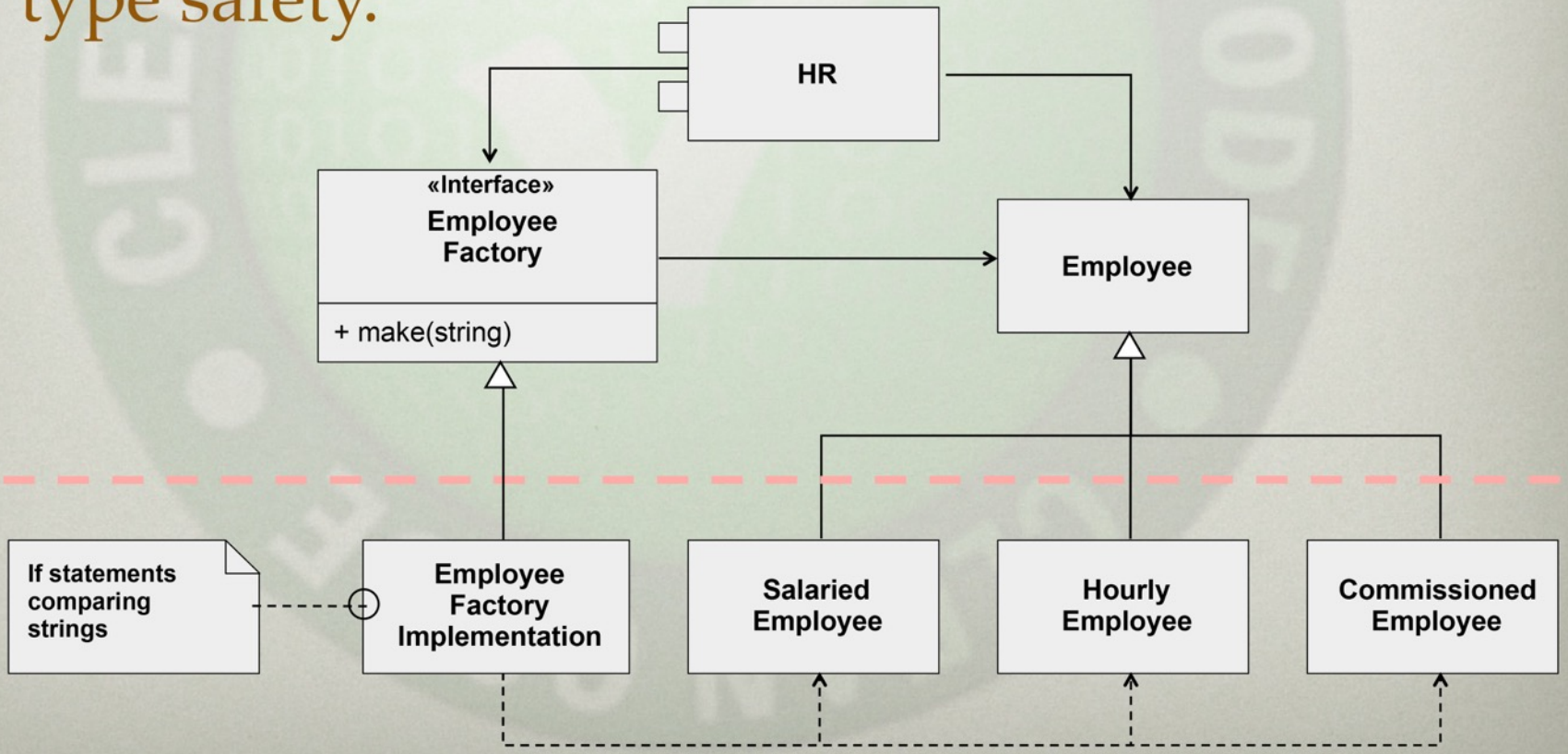
Solution: Abstract Factory [GOF]

- Dependencies are inverted.
- **`HR`** no longer depends on derivatives of **`Employee`**.



Solution: Dynamic Factory

- Solves the problem.
  - But only by removing *compile-time* type safety.

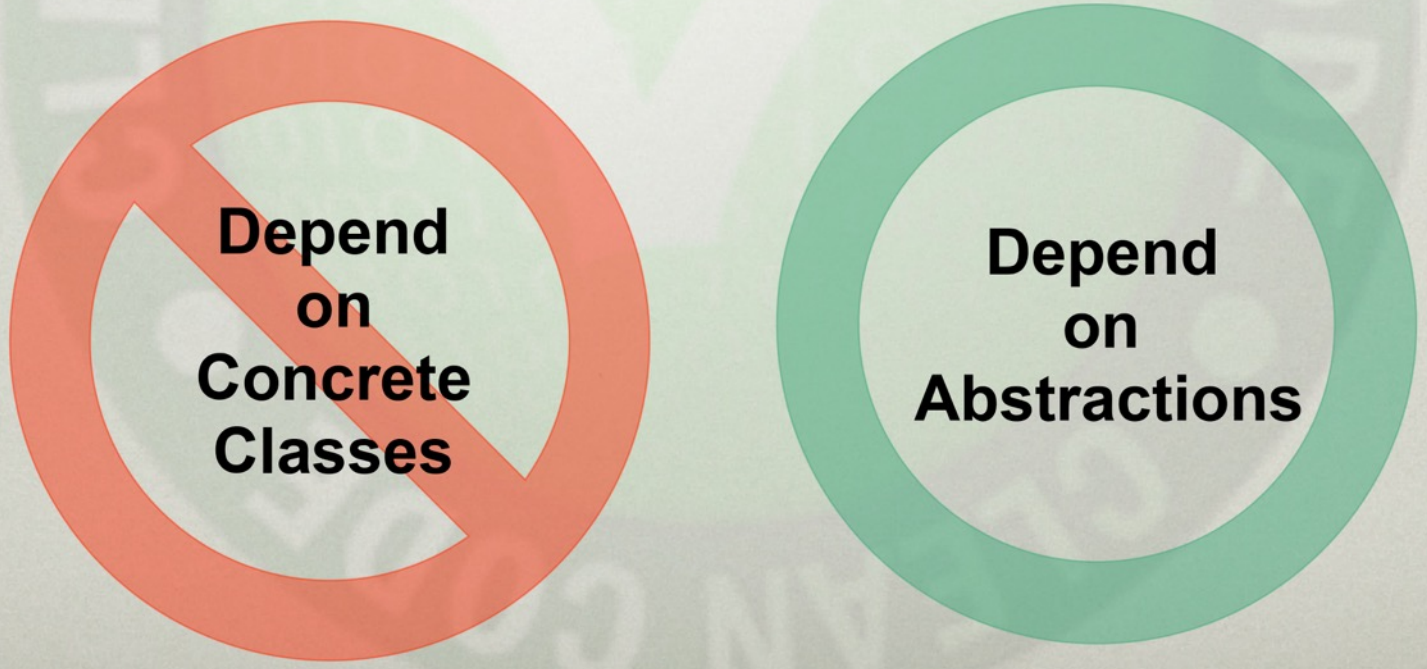


Review: Dependency Inversion

- In procedural designs source code dependencies flow *with* control.
  - This makes it easy to add functions to existing data structures without changing the data structures.
- In OO designs dependencies can be inverted so that source code dependencies flow *against* control.

Wrap Up

- High-Level policy should not depend on details.
- Details should implement abstractions.



Practices

Practices that Fight Rot

- Simple Design
- Automated Testing
- Test-Driven Development
- Refactoring
- Teamwork

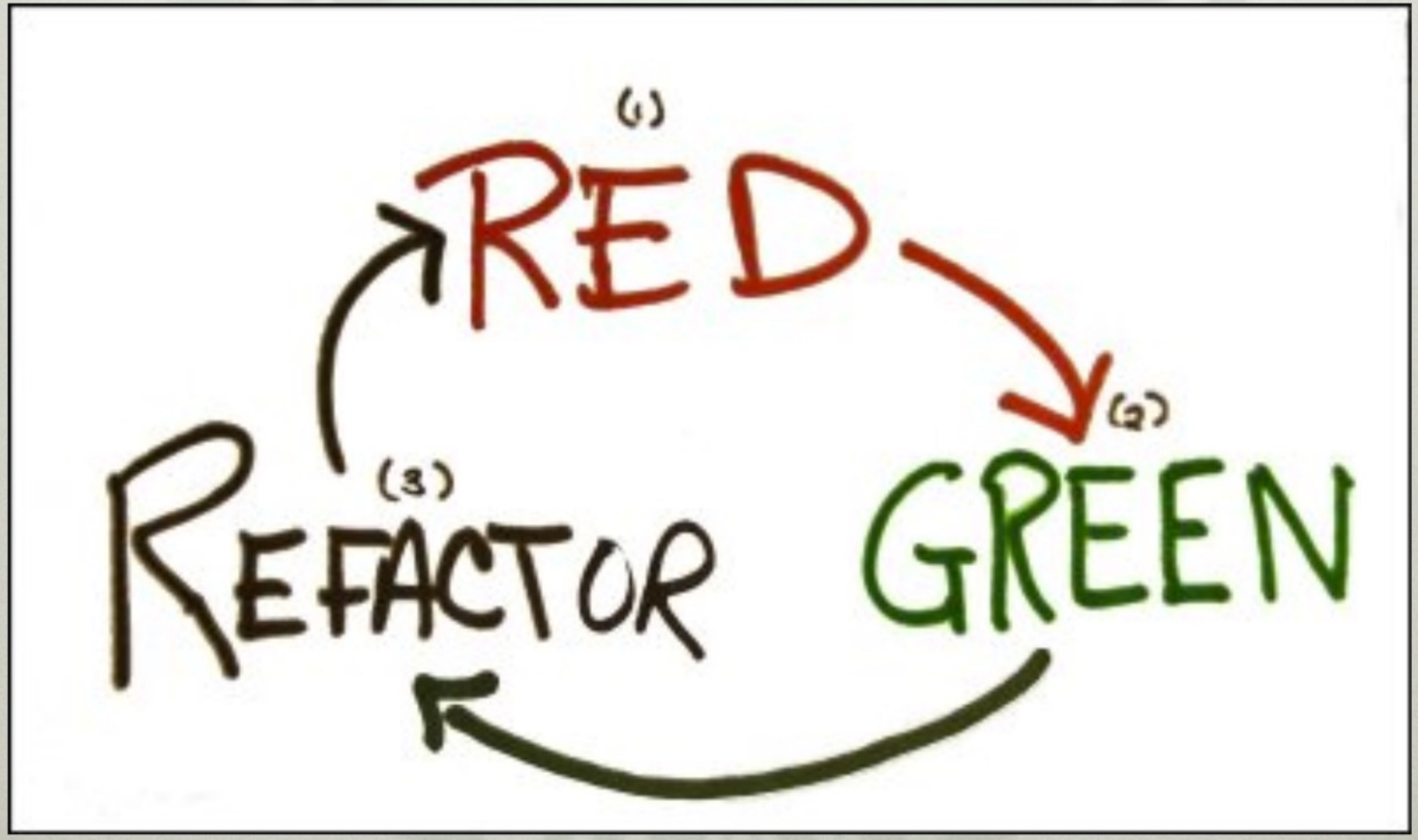
Rules of Simple Design

- In order of importance:
  - Keep all tests passing.
  - Eliminate duplication (DRY, once and only once).
  - Reveal developer's intent.
  - Minimize the number of classes and methods.

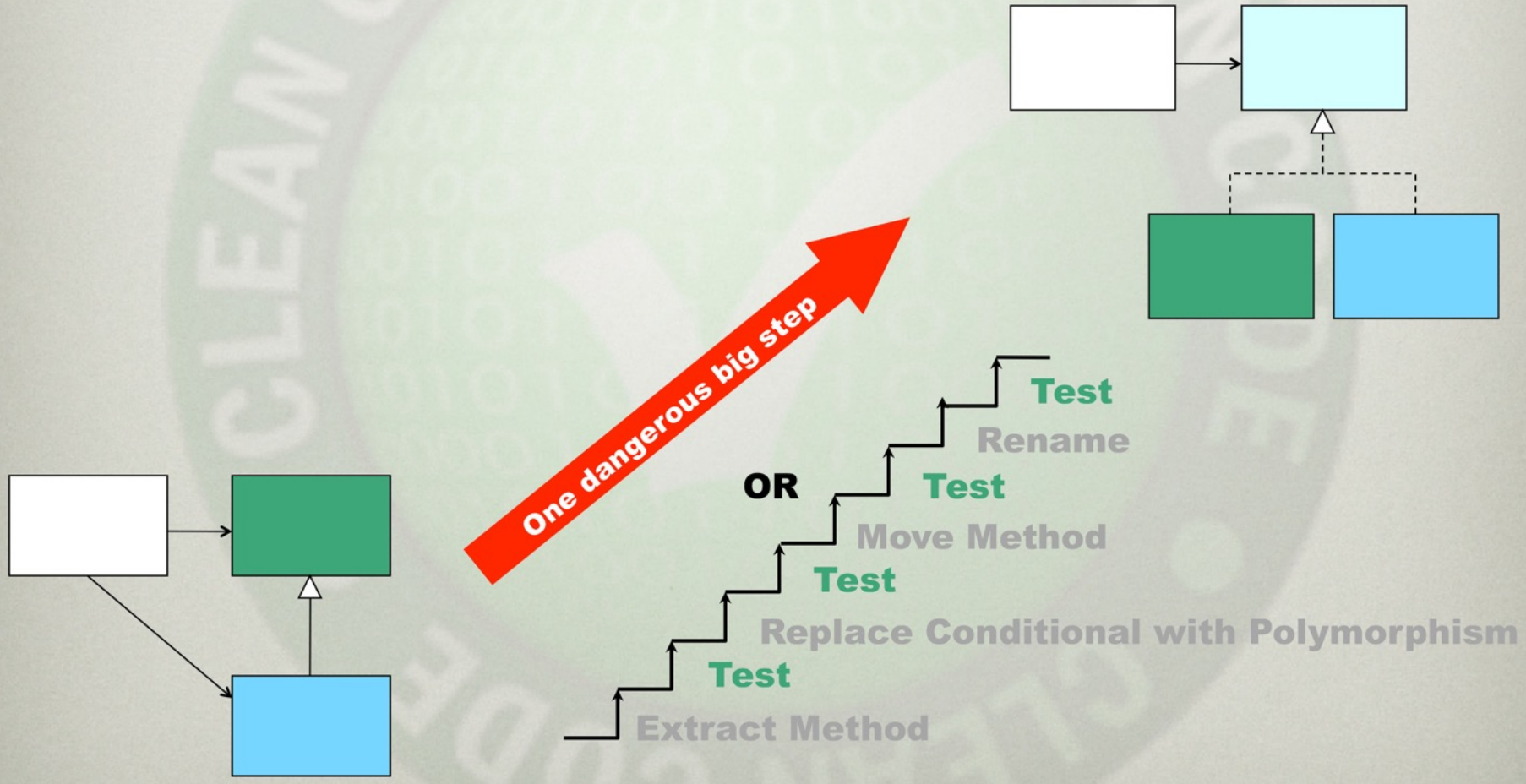
Automated Testing

- Tests should be run often.
- Manual tests are expensive to run:
  - So they aren't run very often.
- Automated tests cost little to run.

TDD & Refactoring

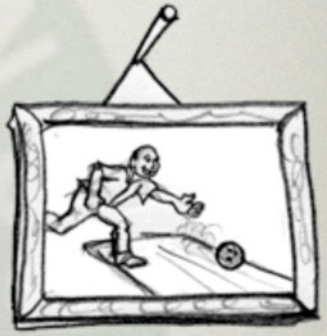


Transform Design with Small Steps



Pair Programming

- Two programmers, one workstation
- Share knowledge and skill
- Continuous code/design review
- Opportunistic
- Optional
- Encouraged



Collective Ownership:

- Team owns code.
- Reduces single-owner bottlenecks.
- Reduces knowledge silos.

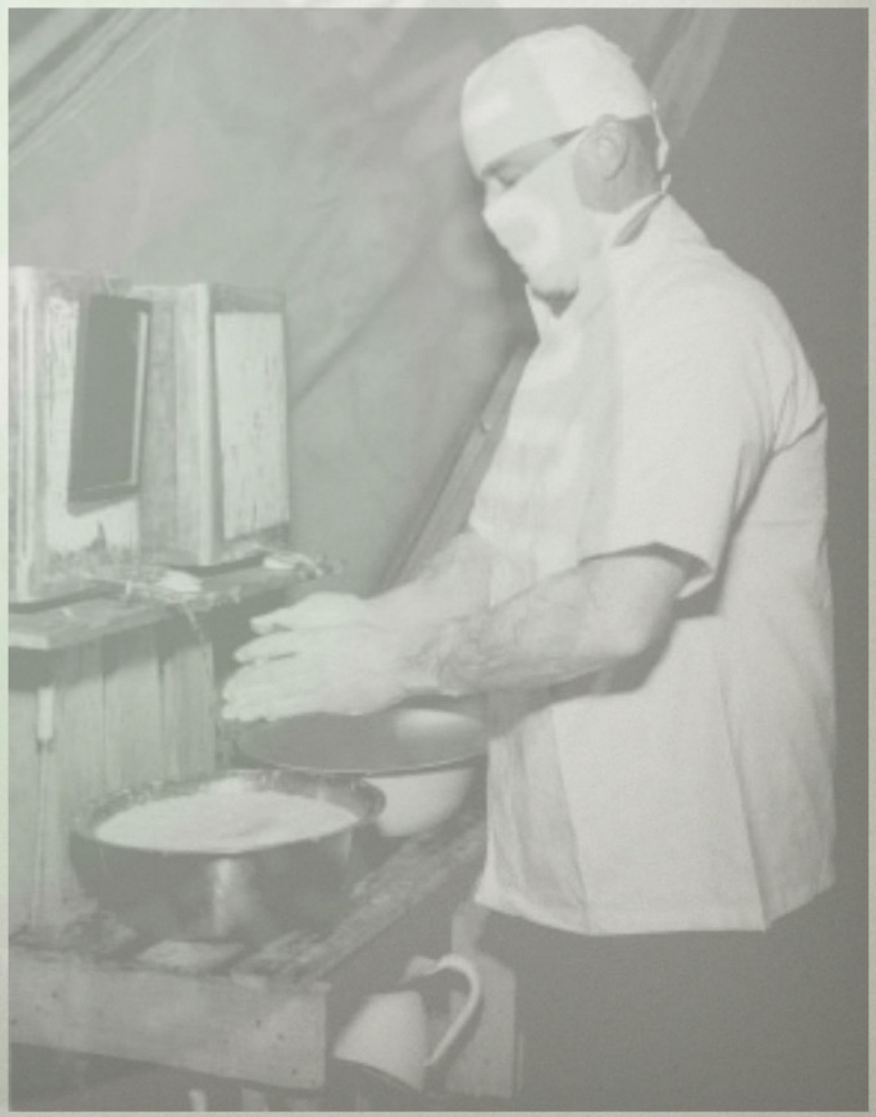
Continuous Integration

- Frequent check-ins
- Non blocking source code control
- All tests pass before commit.
- Automated build
- A failing build is a “Stop the Presses” event.
  - ALL tests must pass at every build.

Conclusion

Clean Code that Works

- Design Smells
- Professionalism
- Dependency Management
- Neglect is Contagious
- Fighting Code Rot
- Test Driven Development



SRP: Single Responsibility Principle

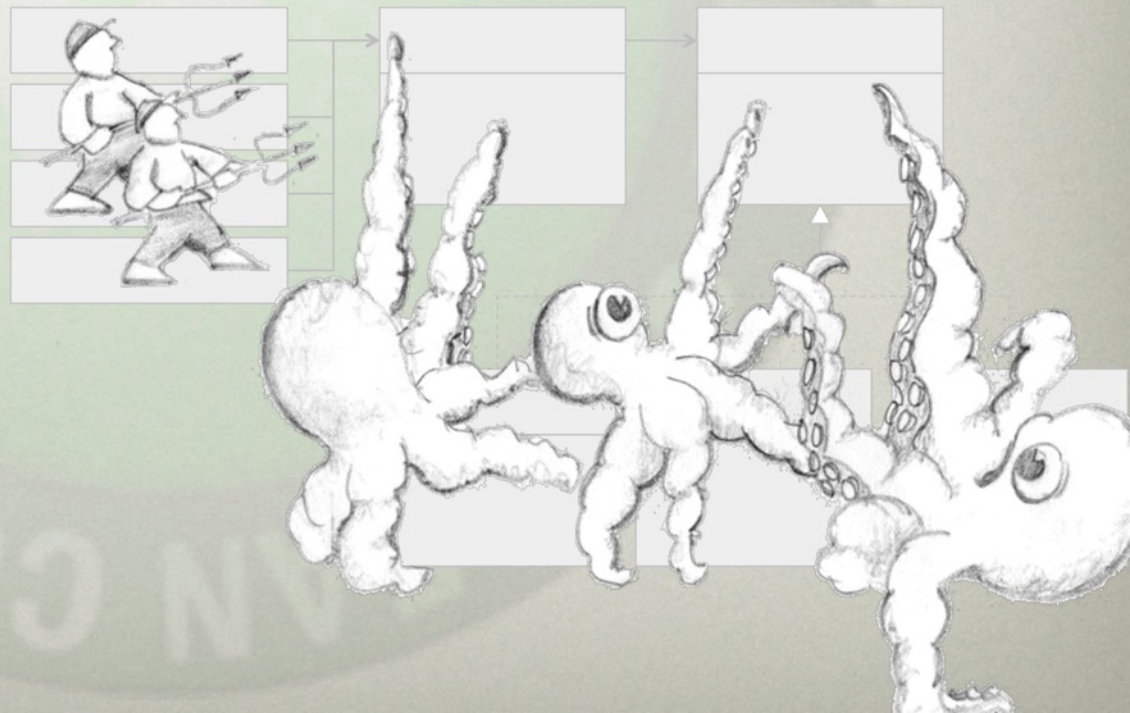
- A class should have one and only one reason to change.
  - Avoid Concurrent Modification.
  - Avoid Deployment Thrashing.

OCP: Open Closed Principle

- Modules should be open for extension but closed for modification.
  - Change should be low cost and low risk.
  - Intention-revealing code.
  - Avoid type-cases.
  - Employ polymorphism and

LSP: Liskov Substitution Principle

- Subtypes must be substitutable.
  - Inheritance != IS-A
  - LSP violations cause latent OCP violations.
  - The needs of the many vs. the needs of the few.

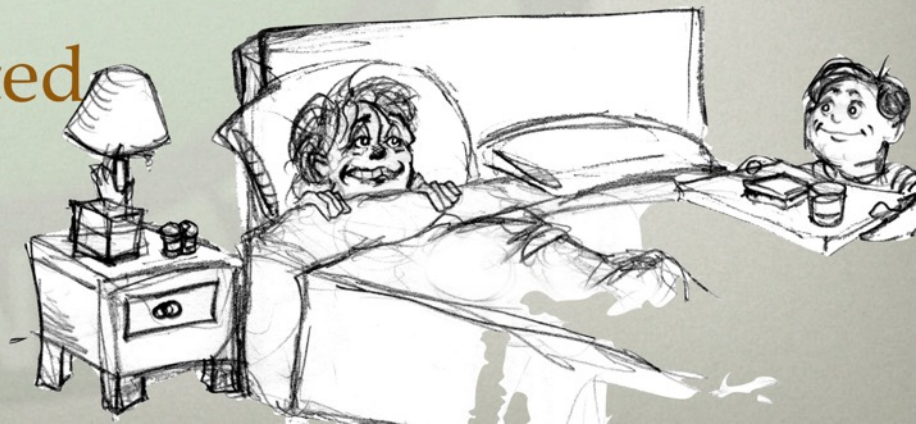


Module 5: Interface Segregation Principle

- Clients should not depend on methods they don't call.
  - Fat Classes
  - Phantom Dependencies
  - Multiple Interface Inheritance
  - Adapter Pattern
  - Clients drive segregation

Module 6: Dependency Inversion Principle

- Abstractions should not depend on details, details should depend on abstractions.
  - Source code dependencies vs. Flow of control:
    - =====> Procedural
    - <====== Object Oriented



Twitter: @unclebobmartin

email: unclebob@cleancoder.com

Website: cleancoder.com

Videos: cleancoders.com

Thank You!

CLEANCODERS

Code-casts for Software Professionals