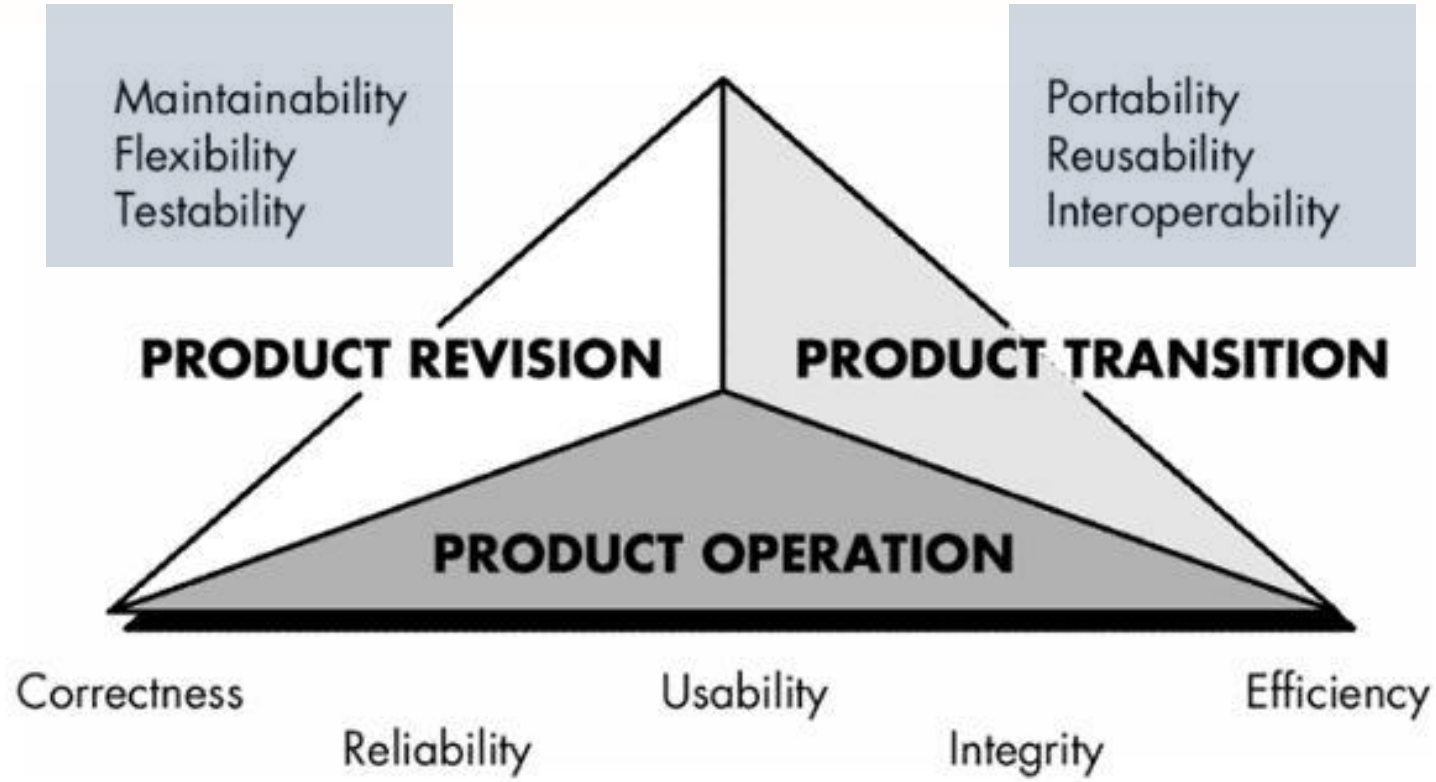
PEMROGRAMAN LANJUT

SOLID: Single Responsibility Principle (SRP)

Oleh
Tri Hadiah Muliawati
Politeknik Elektronika Negeri Surabaya
2021

Politeknik Elektronika Negeri Surabaya
Departemen Teknik Informatika dan Komputer

Review



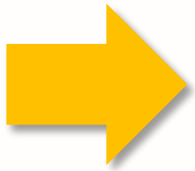
Mc Call Software Quality Metric

Design by Contract (DbC)

- This principle views the relationship between a server and its clients as a formal agreement, expressing each party's rights and obligations
- Methods should specify their pre- and post-conditions: what must be true before and what must be true after their execution, respectively.
- The client should do whatever is necessary to ensure it will meet the pre-conditions.
- Java: iContract, AssertMate, JASS, C4J, Cofoja, Annotated-contracts

Design by Contract: iContract

```
/**
 *  @inv !isEmpty() implies top() != null
 */
public interface Stack
{
    /**
     *  @pre o != null
     *  @post !isEmpty()
     *  @post top() == o
     */
    void push(Object o);
    /**
     *  @pre !isEmpty()
     *  @post @return == top()@pre
     */
    Object pop();
    /**
     *  @pre !isEmpty()
     */
    Object top();
    boolean isEmpty();
}
```



```
import java.util.*;
/**
 *  @inv isEmpty() implies elements.size() == 0
 */
public class StackImpl implements Stack
{
    private final LinkedList elements = new LinkedList();
    public void push(Object o)
    {
        elements.add(o);
    }
    public Object pop()
    {
        final Object popped = top();
        elements.removeLast();
        return popped;
    }
    public Object top()
    {
        return elements.getLast();
    }
    public boolean isEmpty()
    {
        return elements.size() == 0;
    }
}
```

Design by Contract: iContract

```
public  class StackTest
{
    public static void main(String[] args)
    {
        final Stack s = new StackImpl();
        s.push("one");
        s.pop();
        s.push("two");
        s.push("three");
        s.pop();
        s.pop();
        s.pop();    //  causes an assertion to fail
    }
}
```

```
$ java -cp ./src StackTest
Exception in thread "main" java.util.NoSuchElementException
        at java.util.LinkedList.getLast(LinkedList.java:107)
        at StackImpl.top(StackImpl.java:24)
        at StackImpl.pop(StackImpl.java:17)
        at StackTest.main(StackTest.java:14)
```

Executed without using iContract

```
$ java -cp ./instr StackTest
Exception in thread "main" java.lang.RuntimeException:
java.lang.RuntimeException: src/StackImpl.java:22: error:
precondition violated (StackImpl::top()): (/*declared in Stack::top()*/ (!isEmpty()))
        at StackImpl.top(StackImpl.java:210)
        at StackImpl.pop(StackImpl.java:124)
        at StackTest.main(StackTest.java:15)
```

Executed using iContract

Defensive Programming

- Defensive Programming is based on the idea that every program module is solely responsible for itself.
- Defensive programming encourages each procedure to defend itself against errors.
- Assume that your program will be called with incorrect inputs, i.e.: files that are supposed to be open may be closed, that files that are supposed to be closed may be open, and so forth.

Defensive Programming

```
1  function calculateInsurance(userID: number){
2      const user = myDB.findOne(userID);
3      if(!user){
4        throw new UserNotFoundException('User NotFound!');
5      }
6      if(!isValidInsurante(user)){
7         throw new UserInsuranceNotFoundException(user);
8      }
9      if(!isSpanish(user)){
10        throw new UserIsNotSpanishException(user);
11     }
12
13     const value = /***
14           Complex Algorithm
15         */
16     return value;
17 }
```

Checking pre-condition

Common Closure Principle

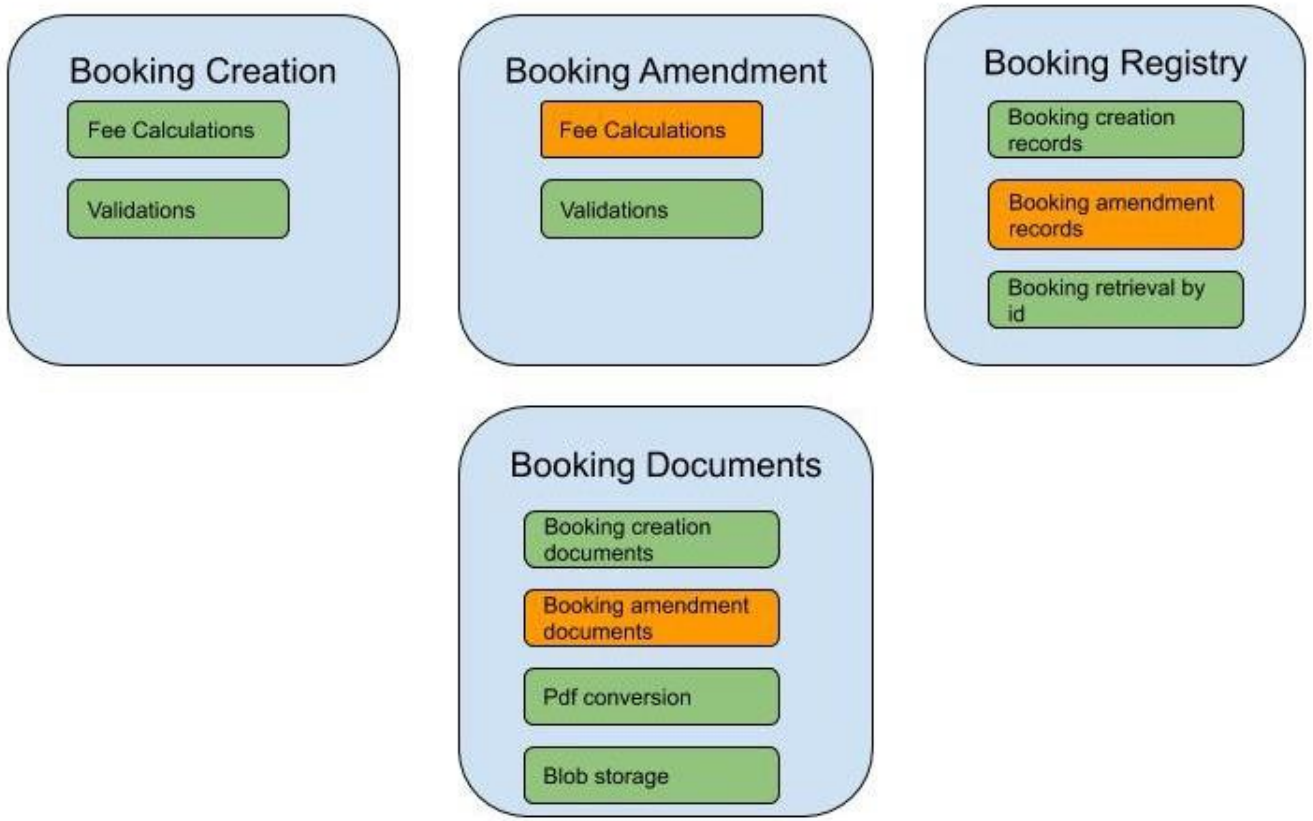
Common Closure Principle

- If the code in an application must change, you would rather that all of the changes occur in one component, rather than being distributed across many components
- If two classes are so tightly bound, that they always change together, then they belong in the same component.
- By following this principle each time we need to change our software the minimum number of components will be affected.

Common Closure Principle

First Scenario:
There is a fee each time a booking amendment happens

- Booking Amendment needs to have the logic which calculates the total fee based on the details of each amendment.
- Booking Amendment Documents must reflect the incurred fee.
- Booking registry needs to store the calculated fee in its records.

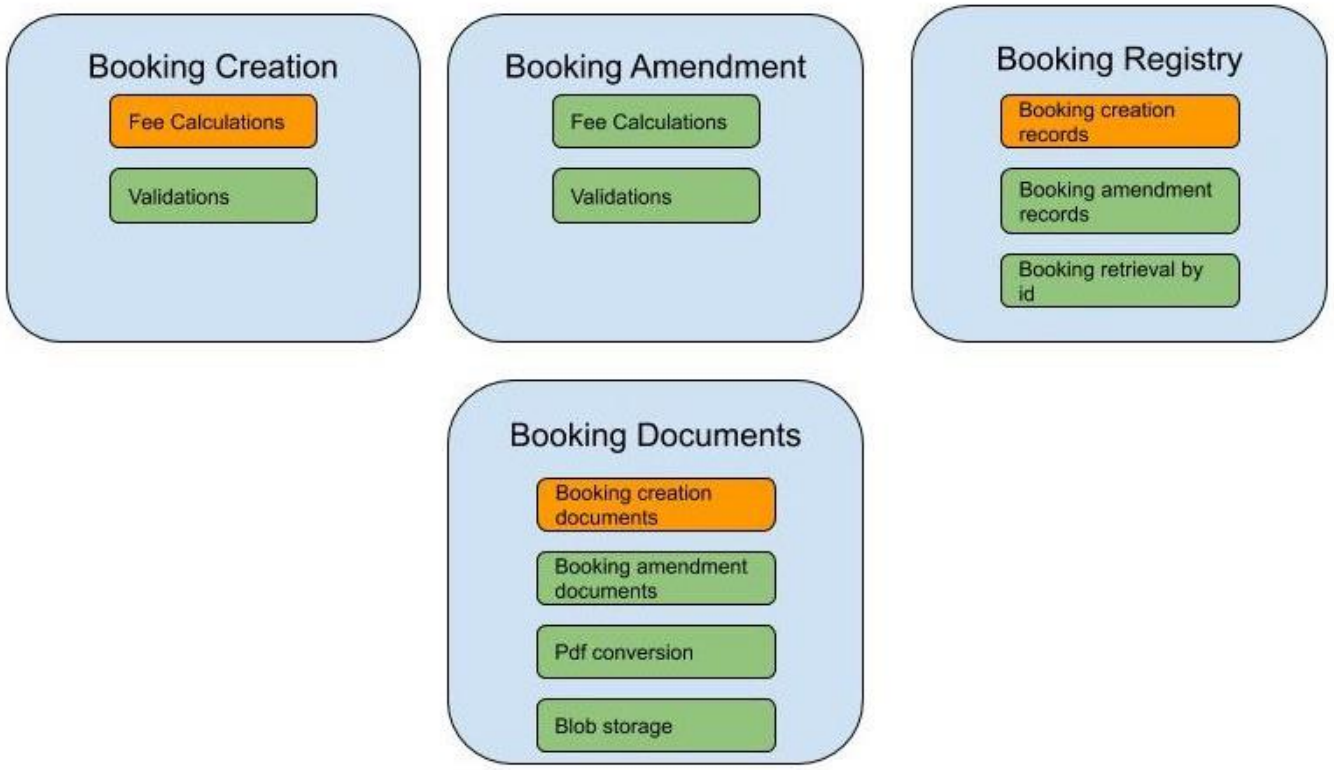


First Scenario

Common Closure Principle

Second Scenario:
There is a promotion code for booking.

- Booking Creation: Needs to be able calculate the discounted fee if there is a promotion code.
- Booking Creation Documents must show the discount when we are creating a booking.
- Booking registry needs to store the promotion code whenever the operation is booking creation and includes a promotion code.

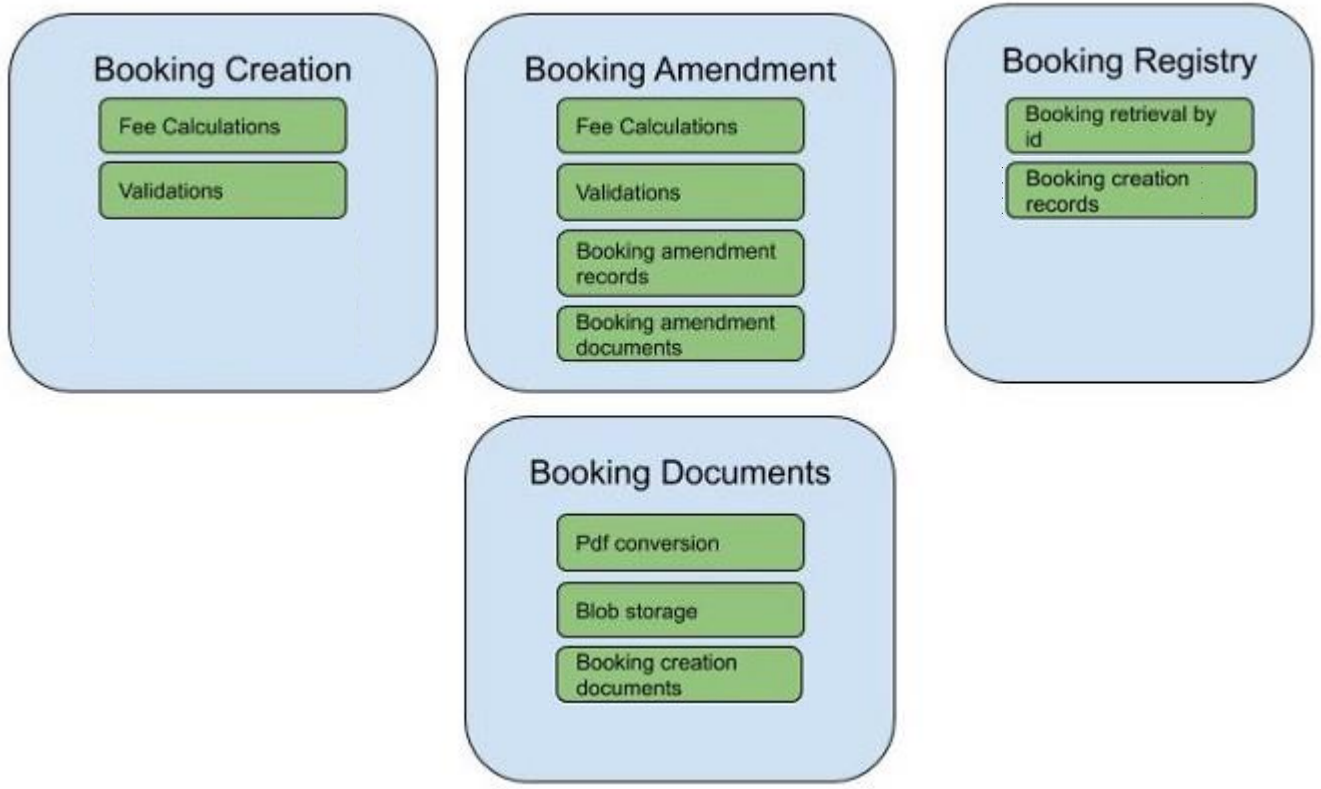


Second Scenario

Common Closure Principle

Revised Design (after applying Common Closure Principle)

- Move the classes which change at the same time and with the same reason to the same component.
- Move the Booking Amendment Document and Booking Amendment Record classes into the Booking Amendment to accommodate first scenario.

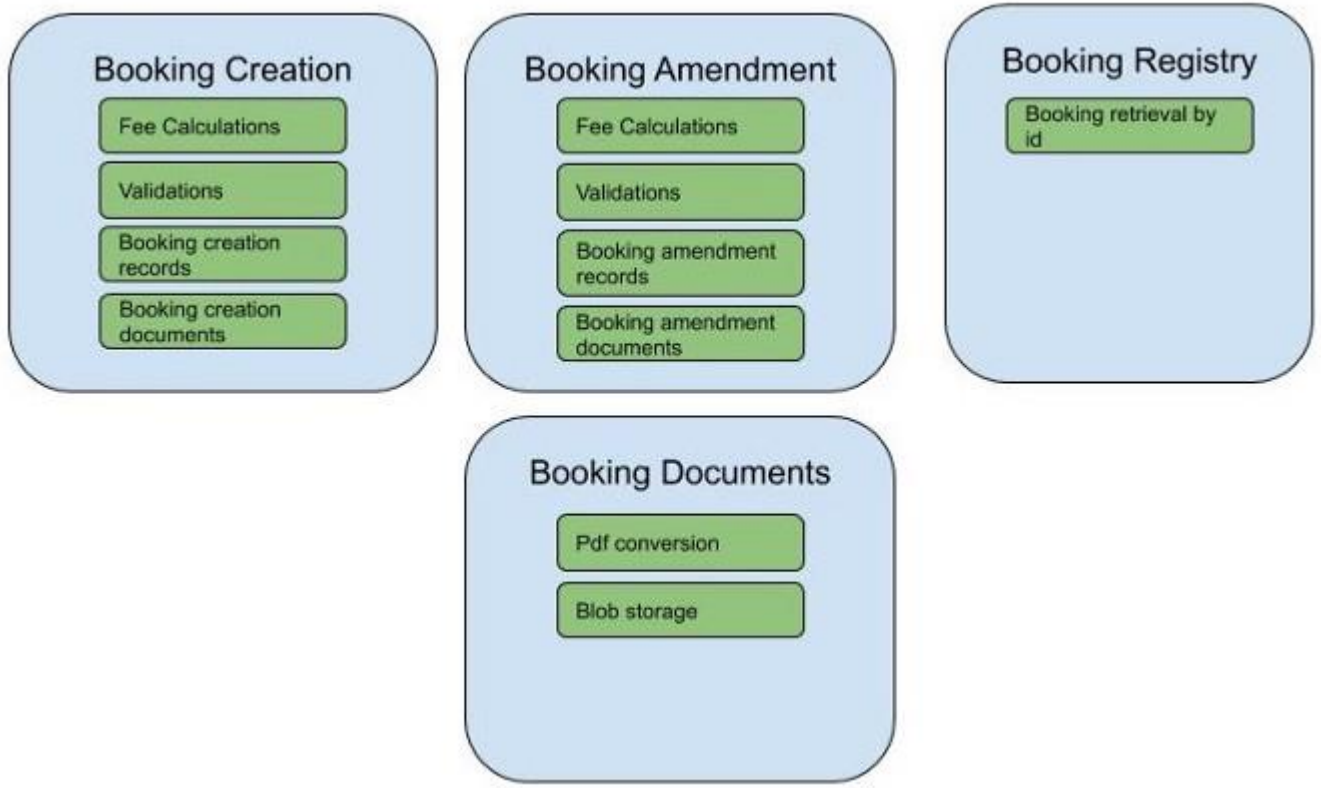


Revised Design

Common Closure Principle

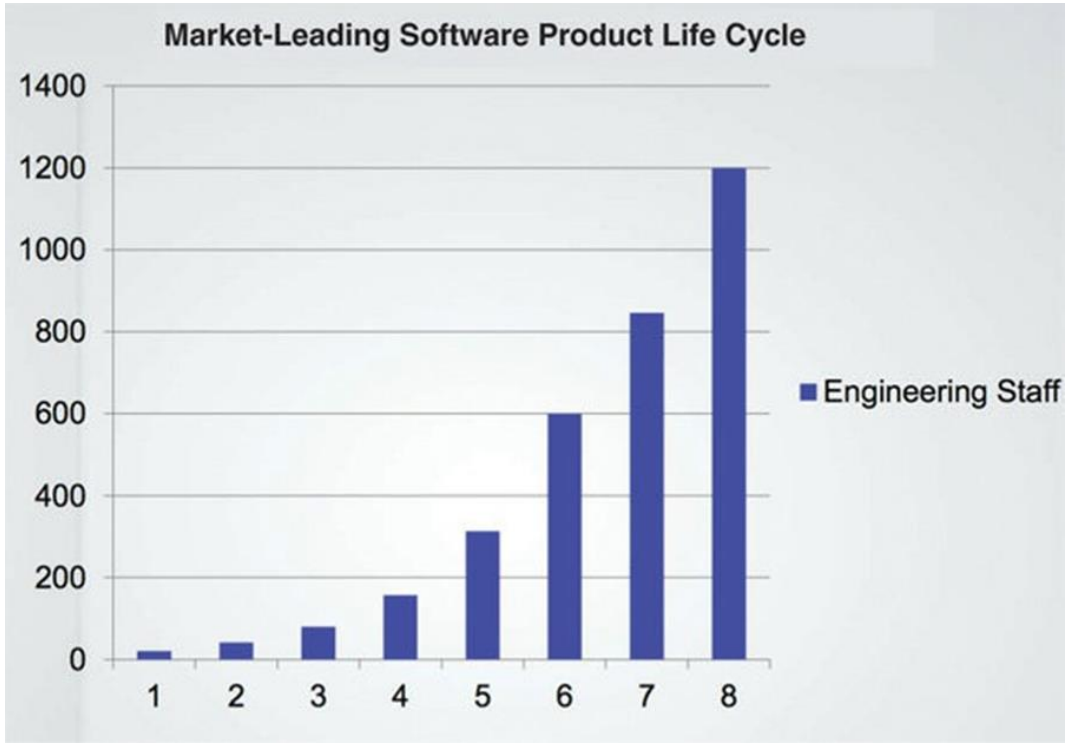
Final Design (after applying Common Closure Principle)

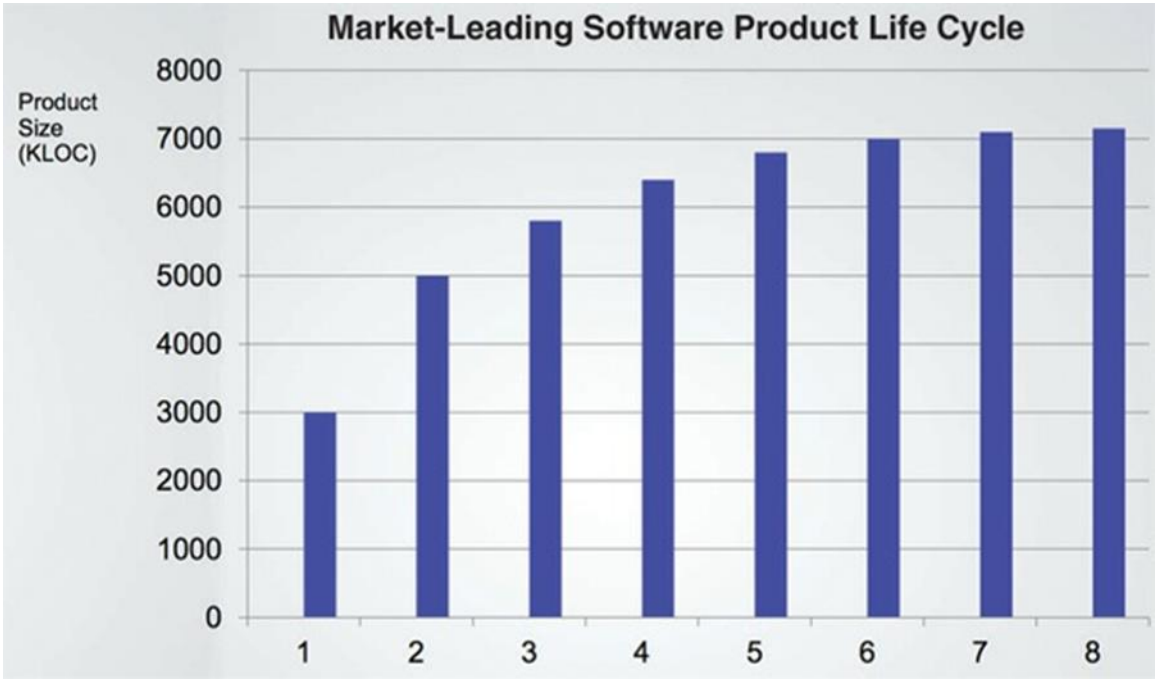
- Move the classes which change at the same time and with the same reason to the same component.
- Move the Booking Creation Document and Booking Creation Record classes into the Booking Creation to accommodate second scenario.



Final Design

End of Review





Cost/LOC is increasing per iteration

What stimulates software to rot

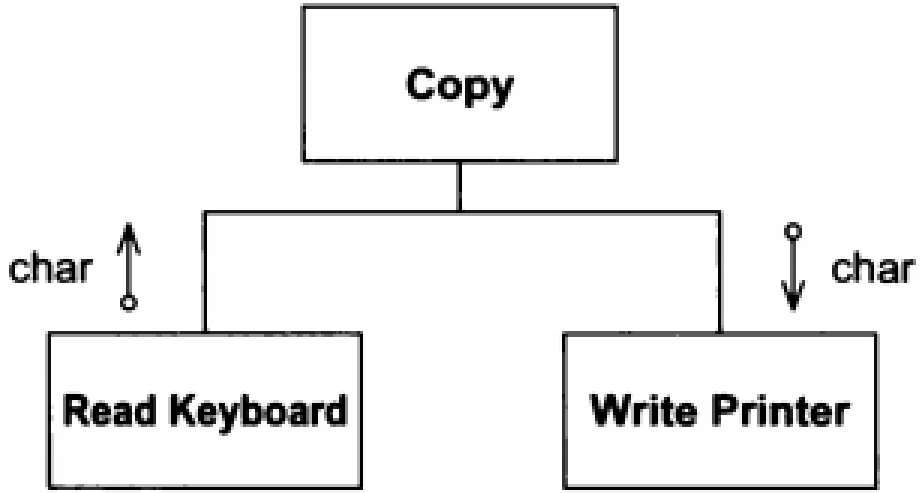
- There are changes of requirement over time which may not be anticipated in the initial design
- Often, these changes need to be made quickly, or may be made by developers who are not familiar with the original design philosophy.
- Even though, the change to design works, it somehow violates the original design.

Symptoms of Poor Design – Designs Smells

1. Rigidity – hard to change
   The system is hard to change because every change forces many other changes to other parts of the system
2. Fragility – easy to break
   Changes cause the system to break in places that have no conceptual relationship to the part that was changed
3. Immobility – hard to reuse
   It is hard to disentangle the system into components that can be reused in other systems
4. Viscosity – hard to do the right thing
   Doing the things right is harder than doing the things wrong
5. Needless Complexity – overdesign
   The design contains infrastructure that adds no direct benefit
6. Needless Repetition – mouse abuse
   The design contains repeating structures that could be unified under a single abstraction
7. Opacity – disorganized expression
   It is hard to read and understand. It does not express its intent well

Case Study

- Aim:
  Program to copy characters from keyboard to printer



Initial Code

```
void Copy()
{
  int c;
  while ((c=RdKbd()) != EOF)
    WrtPrt(c);
}
```

Case Study

- 1st change of requirements:
  Copy program should be able to read from printer and paper tape reader

```
void Copy()
{
  int c;
  while ((c=RdKbd()) != EOF)
    WrtPrt(c);
}
```

```
bool ptFlag = false;
// remember to reset this flag
void Copy()
{
  int c;
  while ((c=(ptflag ? RdPt() : RdKbd())) != EOF)
    WrtPrt(c);
}
```

RdPt() ➔ read from paper tape reader
RdKbd() ➔ read from keyboard
ptFlag ➔ flag to check whether input is paper tape reader

Case Study

WrtPrt() ➔ output is sent to printer
WrtPunch() ➔ output is sent to paper tape punch
punchFlag ➔ flag to check whether output is sent to paper tape punch

- 2st change of requirements:
  Copy program should be able to output to paper tape punch

```
bool ptFlag = false;
// remember to reset this flag
void Copy()

{
  int c;
  while ((c=(ptflag ? RdPt() : RdKbd())) != EOF)
    WrtPrt(c);
}
```

```
bool ptFlag = false;
bool punchFlag = false;
// remember to reset these flags
void Copy()
{
  int c;
  while ((c=(ptflag ? RdPt() : RdKbd())) != EOF)
    punchFlag ? WrtPunch(c) : WrtPrt(c);
}
```

The structure of program is beginning to topple.
Any more changes to the input device will certainly force developer to completely restructure `while`-loop conditional.

Requirements always change ☺

We (developers) live in the world of changing requirements, and our job is to make sure that our software can survive those changes.

-Robert C. Martin-

Case Study

```
bool ptFlag = false;
// remember to reset this flag
void Copy()
{
  int c;
  while ((c=(ptflag ? RdPt() : RdKbd())) != EOF)
    WrtPrt(c);
}
```

```
class Reader
{
  public:
    virtual int read() = 0;
};

class KeyboardReader : public Reader
{
  public:
    virtual int read() {return RdKbd();}
};
```

```
KeyboardReader GdefaultReader;

void Copy(Reader& reader = GdefaultReader)
{
  int c;
  while ((c=reader.read()) != EOF)
    WrtPrt(c);
}
```

Instead of simply changing the code to accommodate the first change of requirements (**Copy program should be able to read from printer and paper tape reader**), developers can improve the design as well.

Therefore, the design can be more resilient towards similar changes in the future (additional type of reader used).

Principles of OO Design

- Single Responsibility Principle
- Open Close Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Principles of OO Design

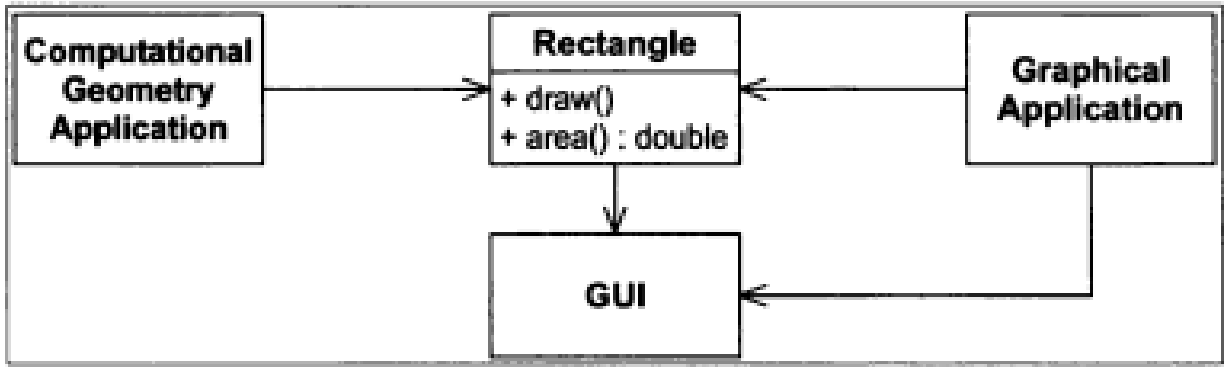
- Those principles are not applied directly in up-front design. Rather, they are applied from iteration to iteration in an attempt to keep the code, and the design it embodies, clean.
- They don’t apply principles when there are no smells. It is a mistake to unconditionally conform to a principle just because it’s a principle.
- Principles are not perfume to be liberally scattered all over the system. Over conformance to the principles leads to the design smell of needless complexity.

Single Responsibility Principle

Single Responsibility Principle (SRP)

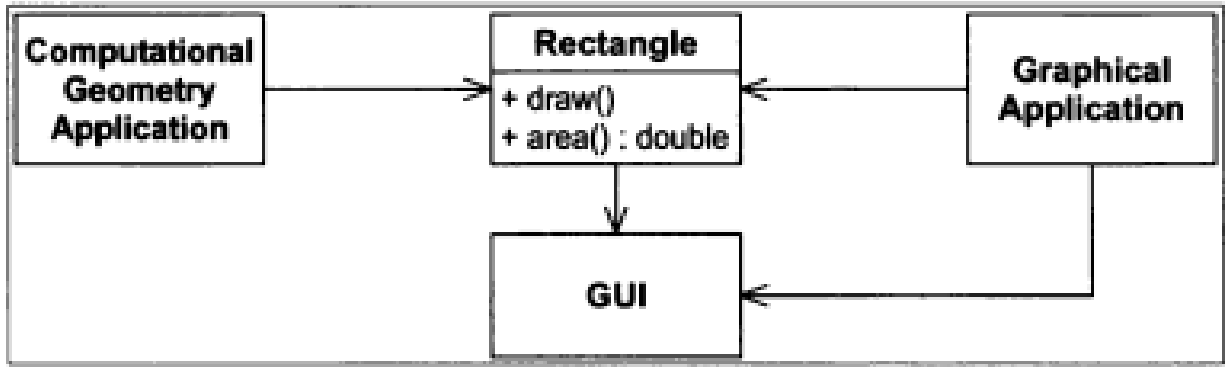
- A module should have only one reason to change. It doesn't mean that every module should do just one thing.
- If a module has more than on responsibility, then the responsibilities become coupled. Changes to one responsibility may impair or inhibit the ability of the module to meet the others.

Case Study: Rectangle



- ComputationalGeometryApplication uses Rectangle to help it with mathematics of geometric shapes. It doesn't need to draw Rectangle on screen.
- GraphicalApplication definitely draws Rectangle on screen, but doesn't always do computational geometry.

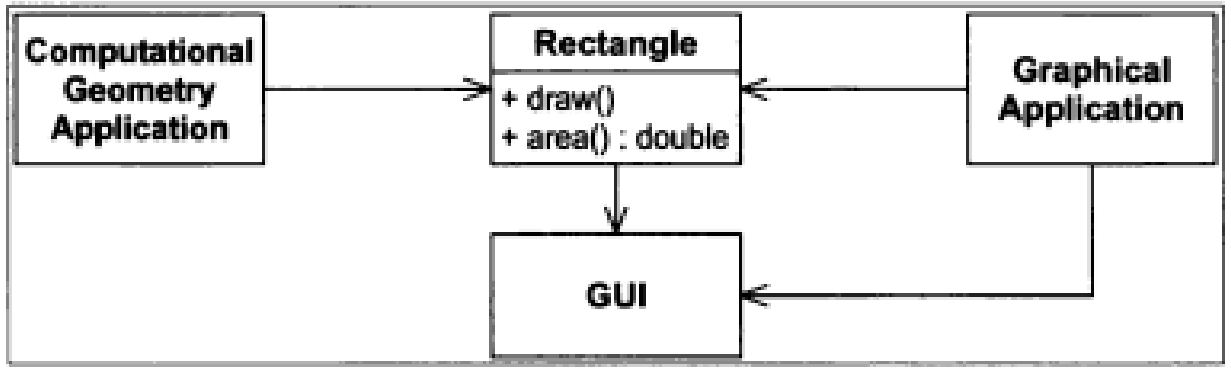
Case Study: Rectangle



- Rectangle has two responsibilities, i.e.:

1. Provide mathematical model of the geometry of rectangle
2. Render rectangle on a GUI

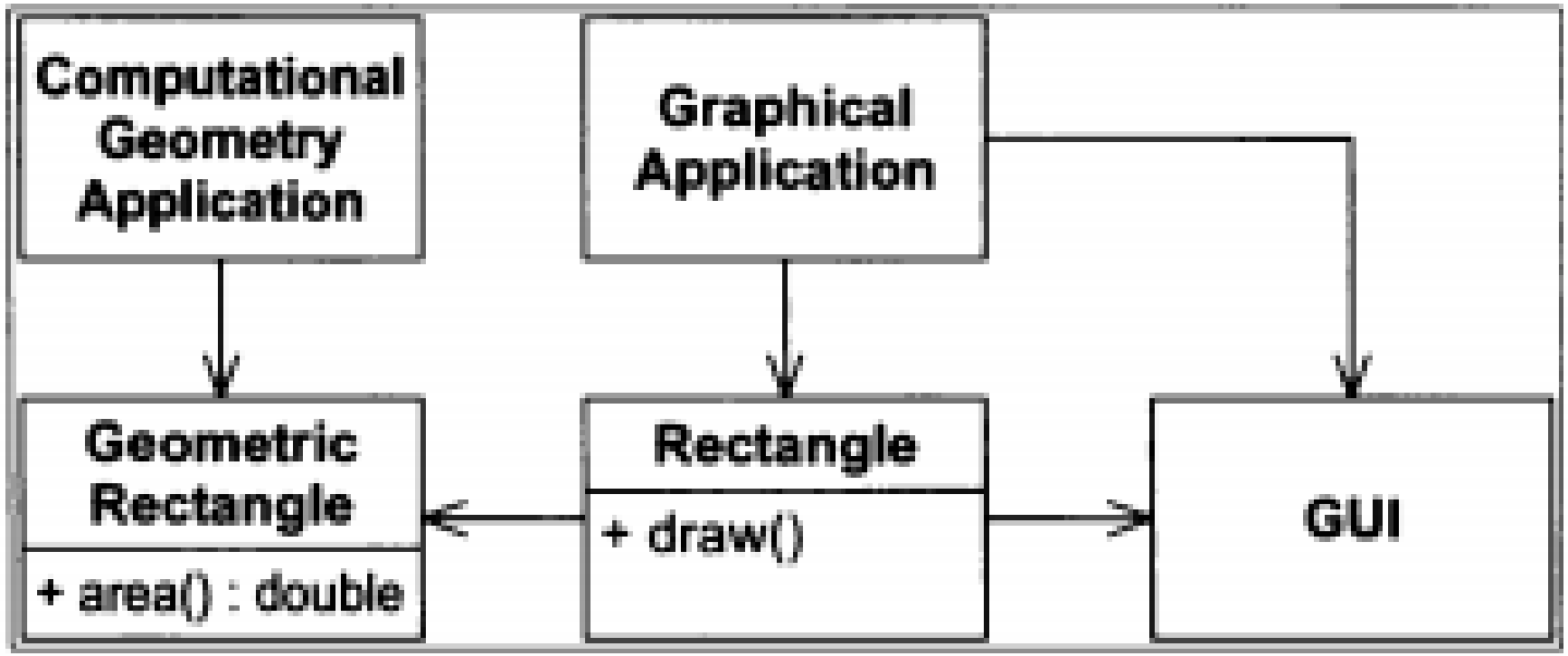
Case Study: Rectangle



- Problems:

1. GUI should be included in ComputationGeometryApplication, because Rectangle uses GUI. It will consume link and compile time as well as memory footprint.
2. If GraphicalApplication causes Rectangle to change for a reason, that change may force us to rebuild, retest, and redeploy the ComputationalGeometryApplication. If not, the application may break in unpredictable ways.

Case Study: Rectangle



Case Study: Email

```
// single responsibility principle - bad example

interface IEmail {
        public void setSender(String sender);
        public void setReceiver(String receiver);
        public void setContent(String content);
}

class Email implements IEmail {
        public void setSender(String sender) {// set sender; }
        public void setReceiver(String receiver) {// set receiver; }
        public void setContent(String content) {// set content; }
}
```

- IEmail interface and Email class have 2 responsibilities (reasons to change):

1. The use of the class in some email protocols such as pop3 or imap. If other protocols must be supported the objects should be serialized in another manner and code should be added to support new protocols.
2. Even if content is a string maybe we want in the future to support HTML or other formats.

Case Study: Email

```
// single responsibility principle - bad example

interface IEmail {
        public void setSender(String sender);
        public void setReceiver(String receiver);
        public void setContent(String content);
}

class Email implements IEmail {
        public void setSender(String sender) {// set sender; }
        public void setReceiver(String receiver) {// set receiver; }
        public void setContent(String content) {// set content; }
}
```

- If we keep only one class, each change for a responsibility might affect the other one:

1. Adding a new protocol will create the need to add code for parsing and serializing the content for each type of field.
2. Adding a new content type (like html) make us to add code for each protocol implemented.

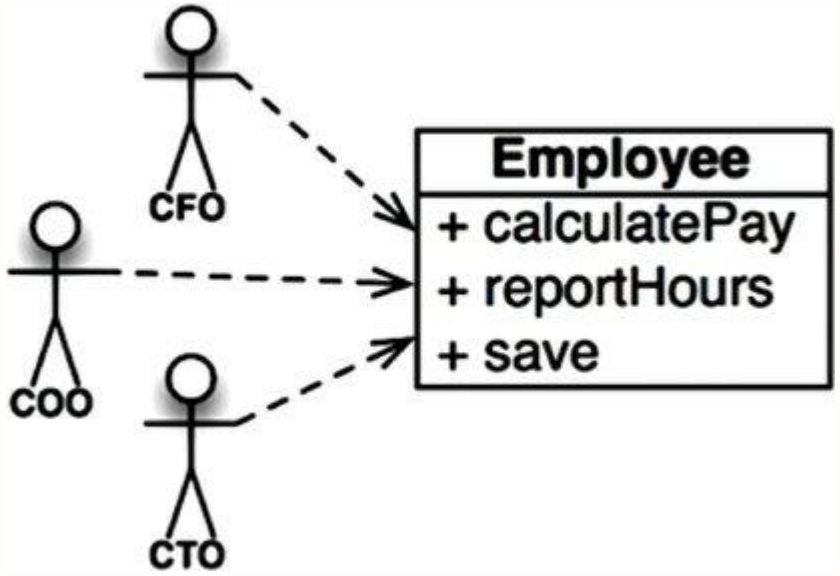
Case Study: Email

```
interface IEmail {
        public void setSender(String sender);
        public void setReceiver(String receiver);
        public void setContent(IContent content);
}

interface Content {
        public String getAsString(); // used for serialization
}

class Email implements IEmail {
        public void setSender(String sender) {// set sender; }
        public void setReceiver(String receiver) {// set receiver; }
        public void setContent(IContent content) {// set content; }
}
```

- Having only one responsibility for each class give us a more flexible design:

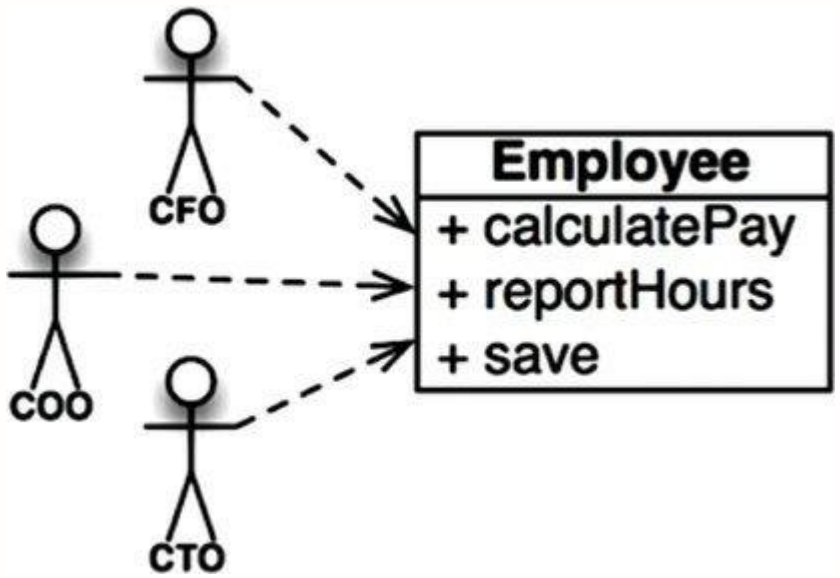
1. adding a new protocol causes changes only in the Email class.
2. adding a new type of content supported causes changes only in Content class.

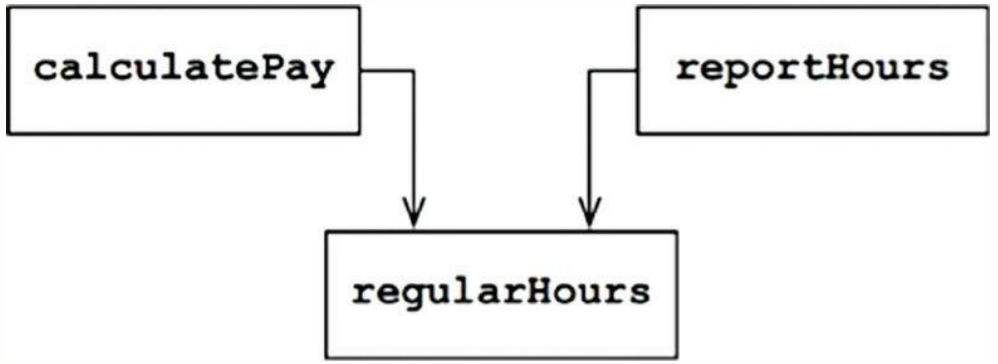
Case Study: Employee



- This class violates the SRP because those three methods are responsible to three very different actors.
- The `calculatePay()` method is specified by the accounting department, which reports to the CFO.
- The `reportHours()` method is specified and used by the human resources department, which reports to the COO.
- The `save()` method is specified by the database administrators (DBAs), who report to the CTO.

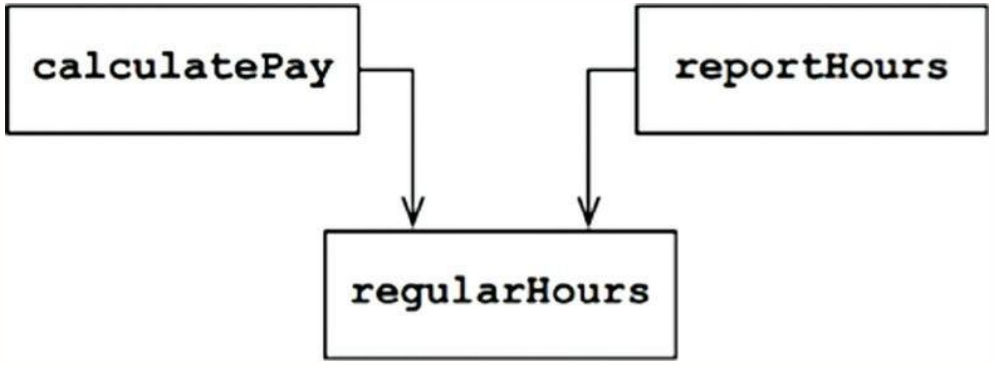
Case Study: Employee





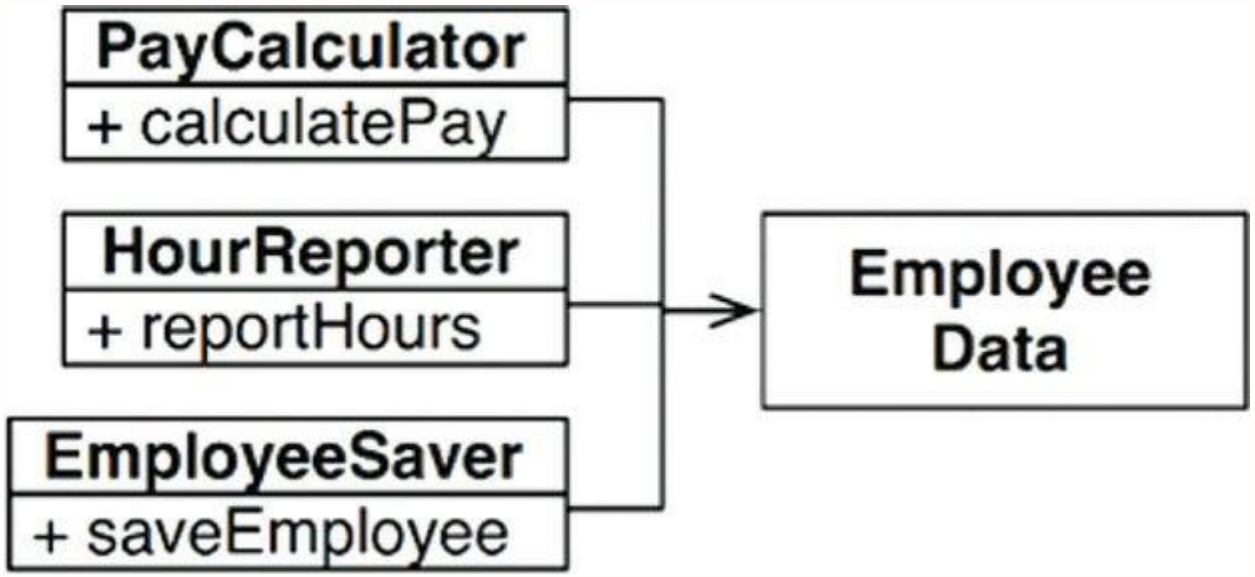
- By putting the source code for these three methods into a single Employee class, the developers have coupled each of these actors to the others.
- This coupling can cause the actions of the CFO's team to affect something that the COO's team depends on.

Case Study: Employee



- Suppose that the `calculatePay()` function and the `reportHours()` function share a common algorithm for calculating non-overtime hours. Suppose that algorithm is put into a function named `regularHours()`
- CFO's team decides that the way non-overtime hours are calculated needs to be tweaked. In contrast, the COO's team in HR does not want that particular tweak because they use non-overtime hours for a different purpose.

Case Study: Employee



The downside of this solution is that the developers now have three classes that they have to instantiate and track.

- The most obvious way to solve the problem is to separate the data from the functions.
- The three classes share access to `EmployeeData`, which is a simple data structure with no methods.
- Each class holds only the source code necessary for its particular function.
- The three classes are not allowed to know about each other. Thus any accidental duplication is avoided.

Case Study: Employee

The `EmployeeFacade` contains very little code. It is responsible for instantiating and delegating to the classes with the functions

References

- Martin, Robert C. Agile Software Development, Principles, Patterns, and Practices. Pearson. Pearson. 2002
- Martin, Robert C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Pearson. 2017.
- https://www.oodesign.com/single-responsibility-principle.html

bridge to the future

http://www.eepis-its.edu