# Design Principles: Part 1

## ENGI 5895: Software Design

Andrew Vardy

Faculty of Engineering & Applied Science
Memorial University of Newfoundland

January 24, 2018

# Outline

1. The Need for Design Principles
2. Refactoring
3. Design Principles
   - The Single-Responsibility Principle (SRP)
   - The Open-Closed Principle (OCP)

# Symptoms of Poor Design

The following are the symptoms of bad software designs, as defined in Ch. 7 of [Martin, 2003]:

# Symptoms of Poor Design

The following are the symptoms of bad software designs, as defined in Ch. 7 of [Martin, 2003]:

- Rigidity: The design is hard to change

# Symptoms of Poor Design

The following are the symptoms of bad software designs, as defined in Ch. 7 of [Martin, 2003]:

- Rigidity: The design is hard to change
- Fragility: The design is easy to break

# Symptoms of Poor Design

The following are the symptoms of bad software designs, as defined in Ch. 7 of [Martin, 2003]:

- Rigidity: The design is hard to change
- Fragility: The design is easy to break
- Immobility: The design is hard to reuse

# Symptoms of Poor Design

The following are the symptoms of bad software designs, as defined in Ch. 7 of [Martin, 2003]:

- Rigidity: The design is hard to change
- Fragility: The design is easy to break
- Immobility: The design is hard to reuse
- Viscosity: It is hard to do the right thing (i.e. forced into hacks)

# Symptoms of Poor Design

The following are the symptoms of bad software designs, as defined in Ch. 7 of [Martin, 2003]:

- Rigidity: The design is hard to change
- Fragility: The design is easy to break
- Immobility: The design is hard to reuse
- Viscosity: It is hard to do the right thing (i.e. forced into hacks)
- Needless Complexity: Overdesign

# Symptoms of Poor Design

The following are the symptoms of bad software designs, as defined in Ch. 7 of [Martin, 2003]:

- Rigidity: The design is hard to change
- Fragility: The design is easy to break
- Immobility: The design is hard to reuse
- Viscosity: It is hard to do the right thing (i.e. forced into hacks)
- Needless Complexity: Overdesign
- Needless Repetition: Mouse abuse

# Symptoms of Poor Design

The following are the symptoms of bad software designs, as defined in Ch. 7 of [Martin, 2003]:

- Rigidity: The design is hard to change
- Fragility: The design is easy to break
- Immobility: The design is hard to reuse
- Viscosity: It is hard to do the right thing (i.e. forced into hacks)
- Needless Complexity: Overdesign
- Needless Repetition: Mouse abuse
- Needless Repetition: (Just kidding)

# Symptoms of Poor Design

The following are the symptoms of bad software designs, as defined in Ch. 7 of [Martin, 2003]:

- Rigidity: The design is hard to change
- Fragility: The design is easy to break
- Immobility: The design is hard to reuse
- Viscosity: It is hard to do the right thing (i.e. forced into hacks)
- Needless Complexity: Overdesign
- Needless Repetition: Mouse abuse
- Needless Repetition: (Just kidding)
- Opacity: Disorganized expression

# Symptoms of Poor Design

The following are the symptoms of bad software designs, as defined in Ch. 7 of [Martin, 2003]:

- Rigidity: The design is hard to change
- Fragility: The design is easy to break
- Immobility: The design is hard to reuse
- Viscosity: It is hard to do the right thing (i.e. forced into hacks)
- Needless Complexity: Overdesign
- Needless Repetition: Mouse abuse
- Needless Repetition: (Just kidding)
- Opacity: Disorganized expression

# Symptoms of Poor Design

The following are the symptoms of bad software designs, as defined in Ch. 7 of [Martin, 2003]:

- Rigidity: The design is hard to change
- Fragility: The design is easy to break
- Immobility: The design is hard to reuse
- Viscosity: It is hard to do the right thing (i.e. forced into hacks)
- Needless Complexity: Overdesign
- Needless Repetition: Mouse abuse
- Needless Repetition: (Just kidding)
- Opacity: Disorganized expression

Also known as **design smells**.

# Software Rots

When the requirements change, the system must change—often in ways not anticipated by the initial design. Over time the software begins to acquire design smells... The software rots!

# Software Rots

When the requirements change, the system must change—often in ways not anticipated by the initial design. Over time the software begins to acquire design smells... The software rots!

- The design should be kept as clean, simple, and as expressive as possible

# Software Rots

When the requirements change, the system must change—often in ways not anticipated by the initial design. Over time the software begins to acquire design smells... The software rots!

- The design should be kept as clean, simple, and as expressive as possible
  - Never say, "we'll fix that later"

# Software Rots

When the requirements change, the system must change—often in ways not anticipated by the initial design. Over time the software begins to acquire design smells... The software rots!

- The design should be kept as clean, simple, and as expressive as possible
  - Never say, "we'll fix that later"
  - ...because you won't!

# Software Rots

When the requirements change, the system must change—often in ways not anticipated by the initial design. Over time the software begins to acquire design smells... The software rots!

- The design should be kept as clean, simple, and as expressive as possible
  - Never say, "we'll fix that later"
  - ...because you won't!
- When a requirement changes, the design should be updated to be resilient to that kind of change in the future

# Refactoring

How do we modify our designs and our code to prevent rot.
Refactoring...

# Refactoring

How do we modify our designs and our code to prevent rot.
Refactoring...

> Refactoring "...the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure" [Fowler, 1999]

# Refactoring

How do we modify our designs and our code to prevent rot.
Refactoring...

> Refactoring "...the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure" [Fowler, 1999]

# Refactoring

How do we modify our designs and our code to prevent rot.
Refactoring...

> Refactoring "...the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure" [Fowler, 1999]

- You can refactor code:

# Refactoring

How do we modify our designs and our code to prevent rot. Refactoring...

> Refactoring "...the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure" [Fowler, 1999]

- You can refactor code:
  - e.g. Read ch. 5 of [Martin, 2003]

# Refactoring

How do we modify our designs and our code to prevent rot. Refactoring...

> Refactoring "...the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure" [Fowler, 1999]

- You can refactor code:
  - e.g. Read ch. 5 of [Martin, 2003]
- You can refactor your design:

# Refactoring

How do we modify our designs and our code to prevent rot. Refactoring...

> Refactoring "...the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure" [Fowler, 1999]

- You can refactor code:
  - e.g. Read ch. 5 of [Martin, 2003]
- You can refactor your design:
  - We will see many examples

# The Single-Responsibility Principle (SRP)

*A class should have only one responsibility.*

# The Single-Responsibility Principle (SRP)

*A class should have only one responsibility.*

# The Single-Responsibility Principle (SRP)

*A class should have only one responsibility.*

*OR*
*A class should have only one reason to change.*

# The Single-Responsibility Principle (SRP)

*A class should have only one responsibility.*

*OR*
*A class should have only one reason to change.*

# The Single-Responsibility Principle (SRP)

> *A class should have only one responsibility.*
>
> *OR*
> *A class should have only one reason to change.*

A class with several responsibilities creates unnecessary couplings between those responsibilities.

# e.g. Rectangle Class

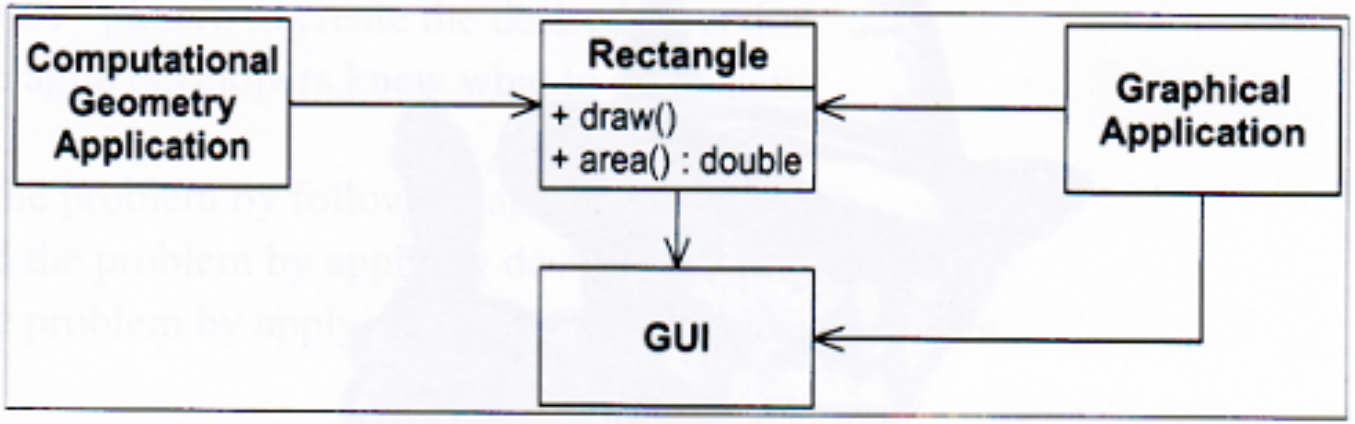


**Figure 8-1** More than one responsibility

# e.g. Rectangle Class

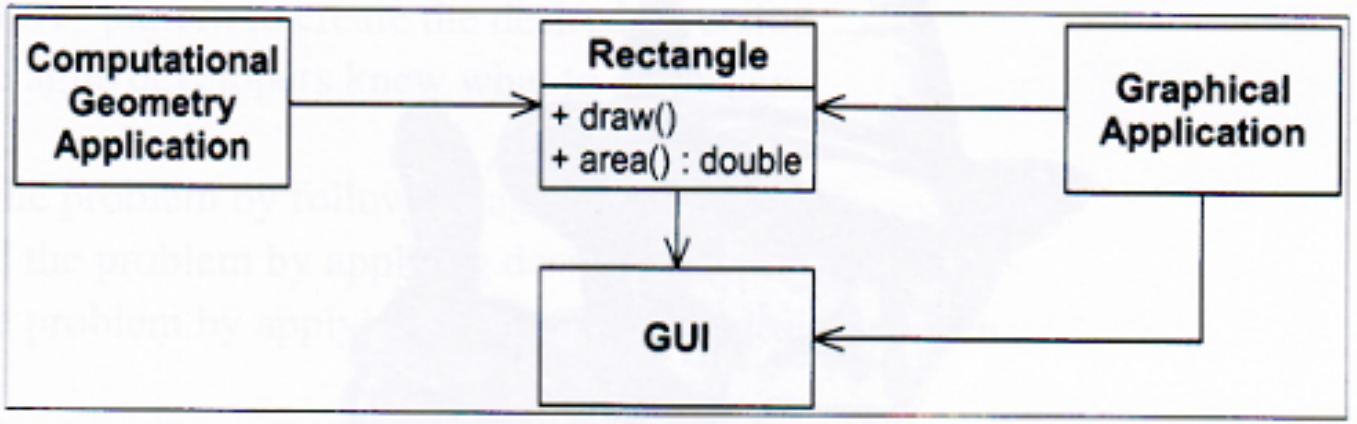


**Figure 8-1** More than one responsibility

- The Geometry Application is concerned with the mathematics of geometric shapes

# e.g. Rectangle Class

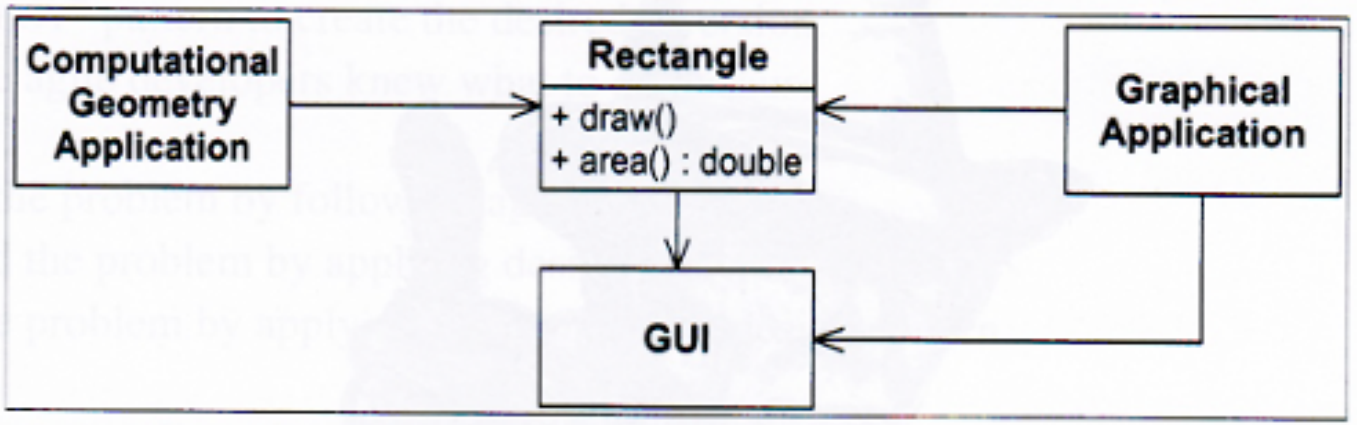


**Figure 8-1** More than one responsibility

- The Geometry Application is concerned with the mathematics of geometric shapes
- The Graphical Application may also involve some geometry, but it also needs to draw geometric shapes

# e.g. Rectangle Class

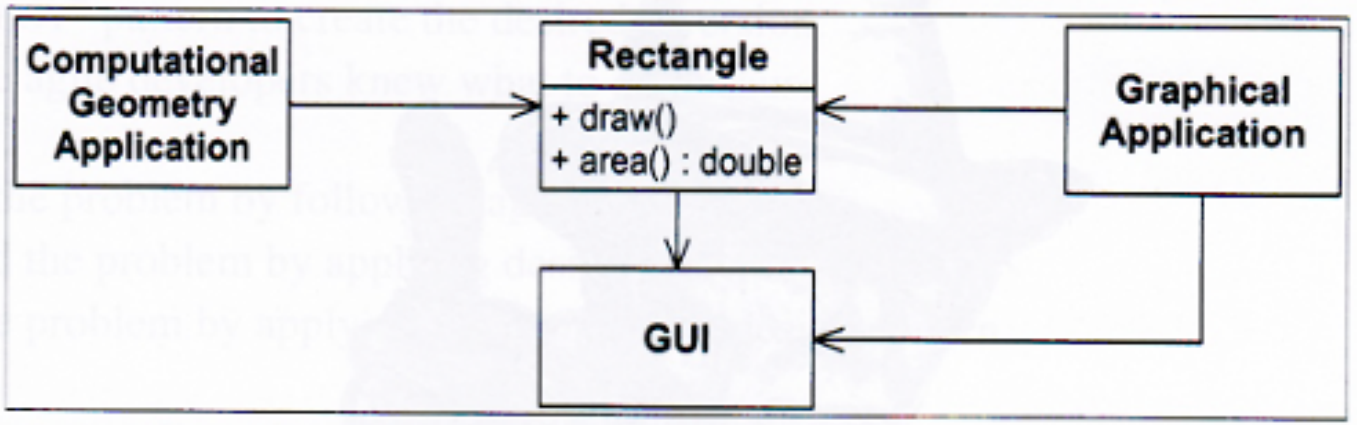


**Figure 8-1** More than one responsibility

- The Geometry Application is concerned with the mathematics of geometric shapes
- The Graphical Application may also involve some geometry, but it also needs to draw geometric shapes
- The Rectangle class has two responsibilities:

# e.g. Rectangle Class

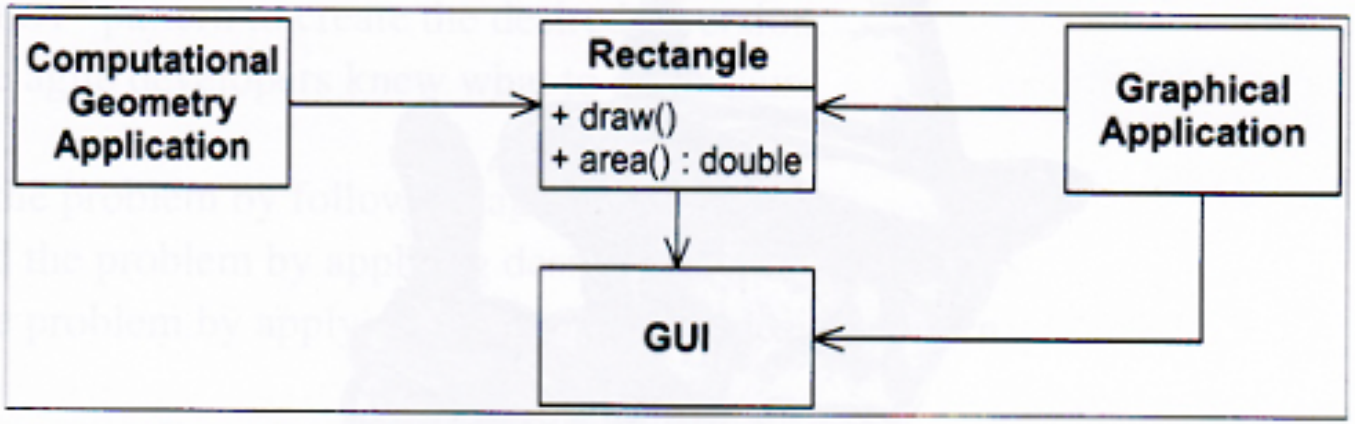


**Figure 8-1** More than one responsibility

- The Geometry Application is concerned with the mathematics of geometric shapes
- The Graphical Application may also involve some geometry, but it also needs to draw geometric shapes
- The Rectangle class has two responsibilities:
  - Provide a mathematical model of a rectangle

# e.g. Rectangle Class

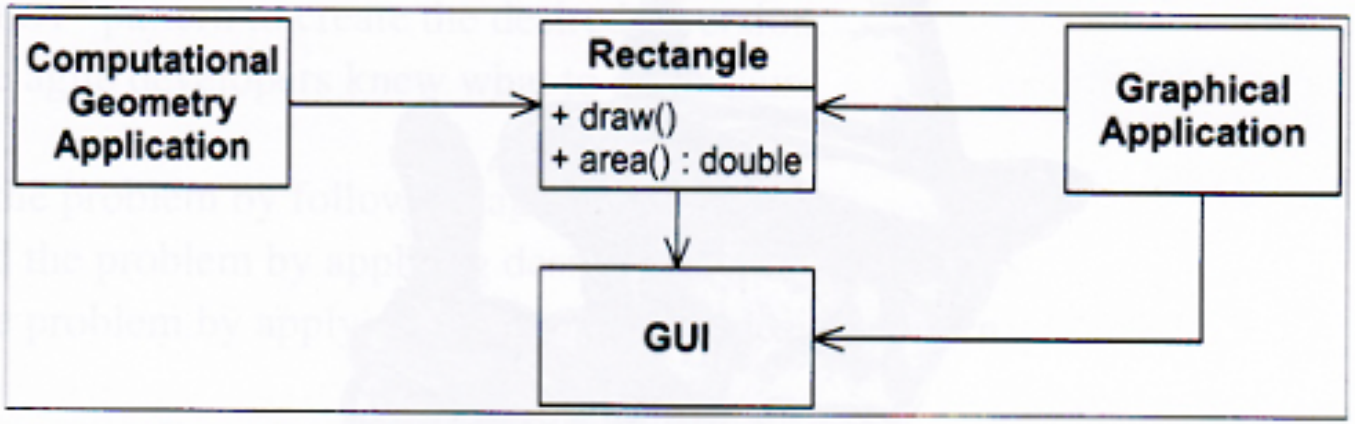


**Figure 8-1** More than one responsibility

- The Geometry Application is concerned with the mathematics of geometric shapes
- The Graphical Application may also involve some geometry, but it also needs to draw geometric shapes
- The Rectangle class has two responsibilities:
  - Provide a mathematical model of a rectangle
  - Render a rectangle

[Figure repeated]

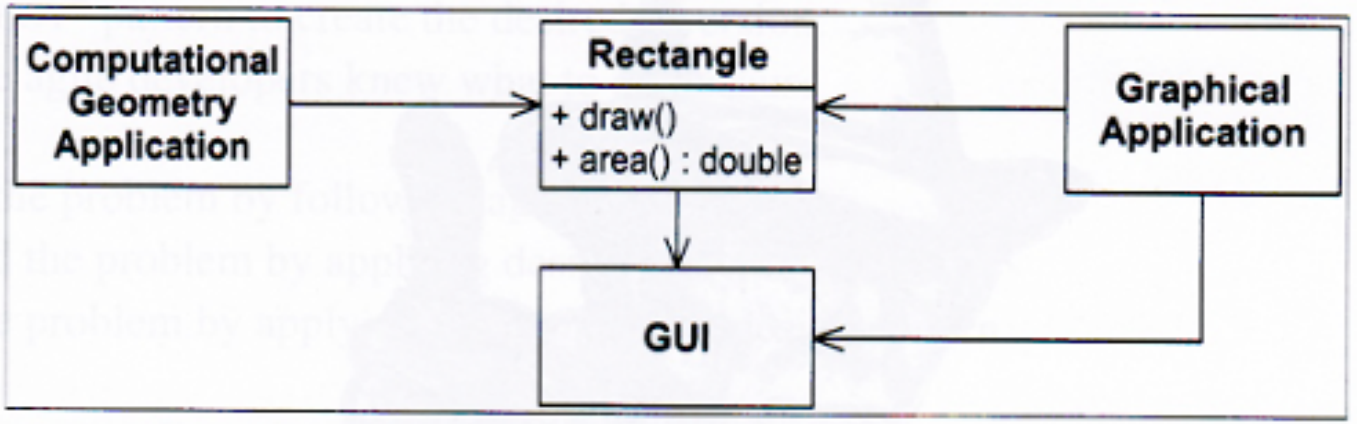


**Figure 8-1** More than one responsibility

[Figure repeated]

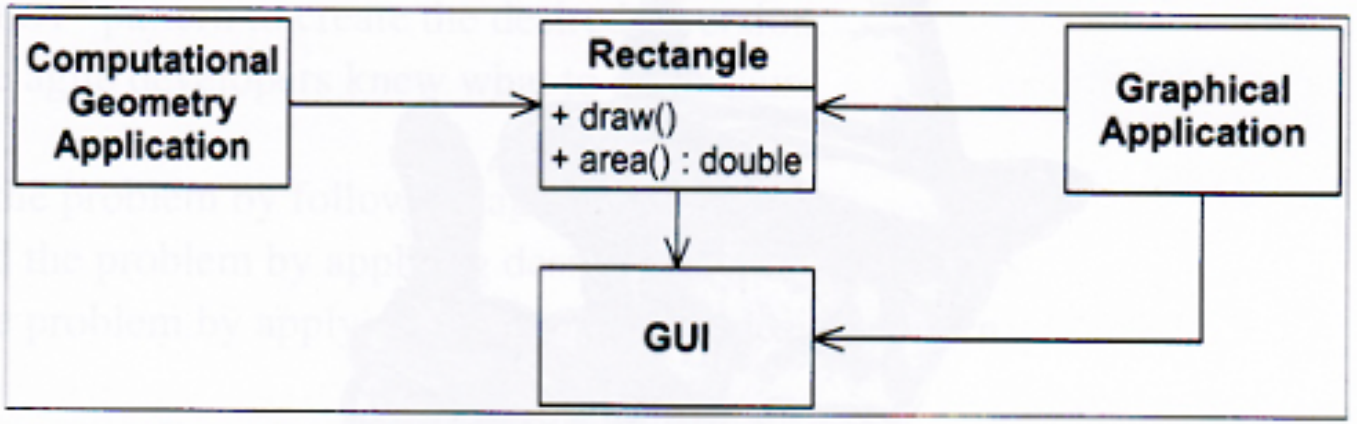


**Figure 8-1** More than one responsibility

- Problems created:

[Figure repeated]

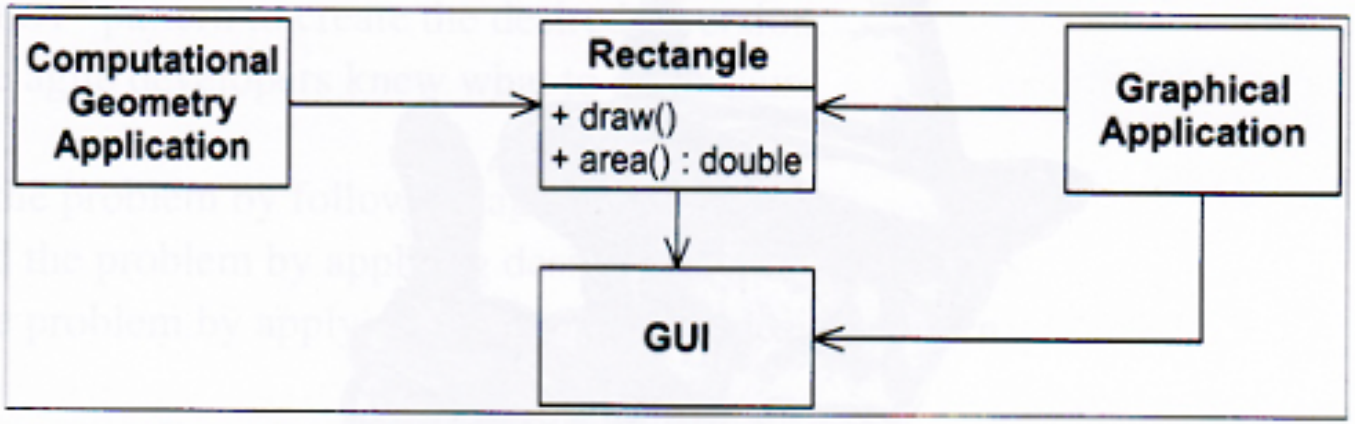


**Figure 8-1** More than one responsibility

- Problems created:
  - Inclusion: The GUI must be included in the Geometry Application (C++: linked into executable, Java: GUI.class file included in JAR file)

[Figure repeated]

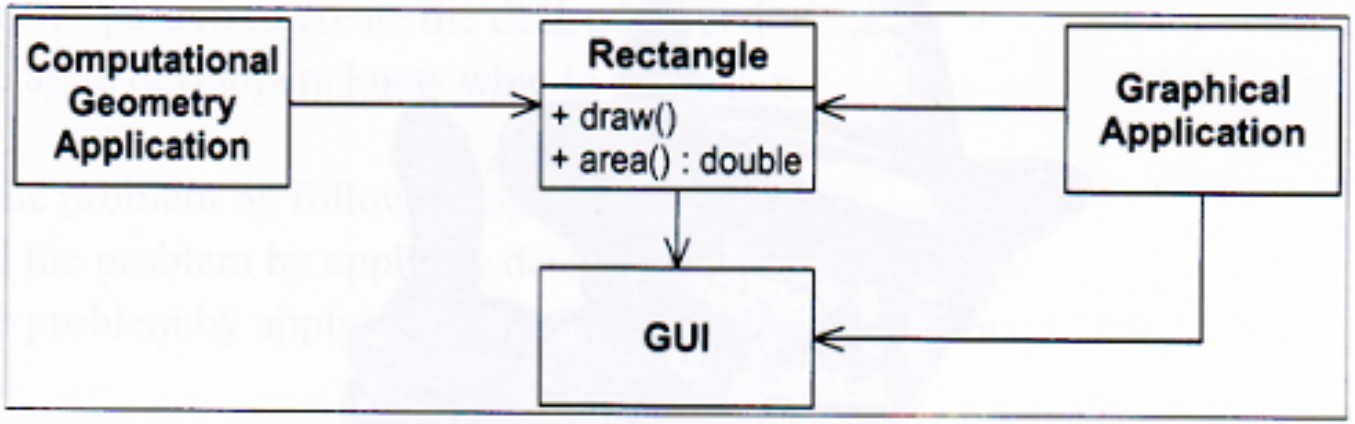


**Figure 8-1** More than one responsibility

- Problems created:
  - Inclusion: The GUI must be included in the Geometry Application (C++: linked into executable, Java: GUI.class file included in JAR file)
  - A change required for one application may affect the other (e.g. adding a colour attribute)

Solution:

- Separate the two responsibilities (math rep. + drawing) into two separate classes

Solution:

- Separate the two responsibilities (math rep. + drawing) into two separate classes

Solution:

- Separate the two responsibilities (math rep. + drawing) into two separate classes

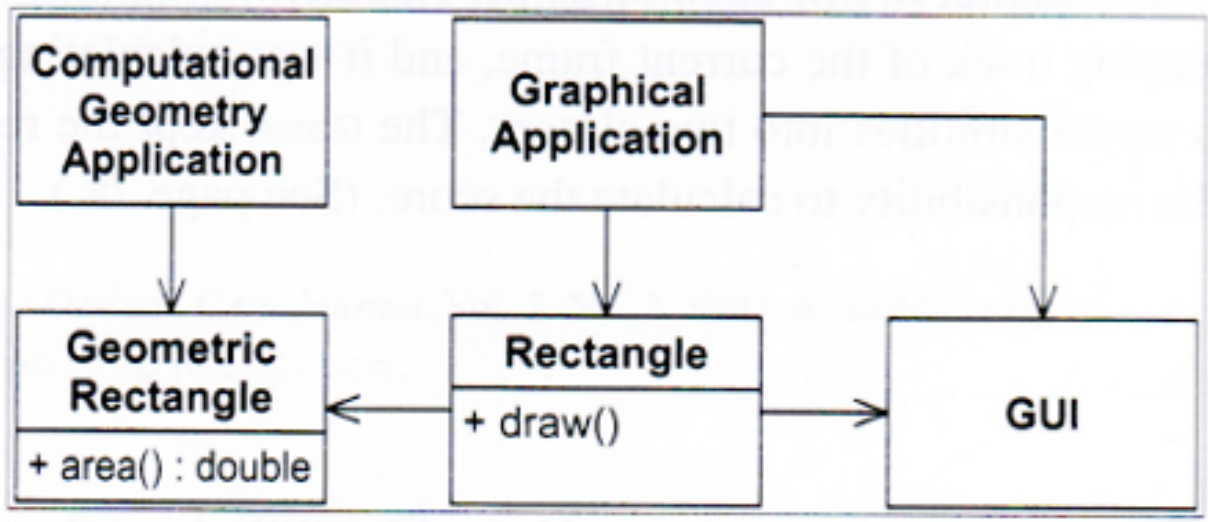


**Figure 8-2** Separated Responsibilities

# e.g. Modem Interface

```
interface Modem {
    void dial(String pno);
    void hangup();
    void send(char c);
    char recv();
}
```

# e.g. Modem Interface

```
interface Modem {
    void dial(String pno);
    void hangup();
    void send(char c);
    char recv();
}
```

Multiple responsibilities? You could say there are two:

# e.g. Modem Interface

```
interface Modem {
    void dial(String pno);
    void hangup();
    void send(char c);
    char recv();
}
```

Multiple responsibilities? You could say there are two:

- Connection management (dial, hangup)

# e.g. Modem Interface

```
interface Modem {
    void dial(String pno);
    void hangup();
    void send(char c);
    char recv();
}
```

Multiple responsibilities? You could say there are two:

- Connection management (dial, hangup)
- Data transfer (send, recv)

If connection management and data transfer are considered separate responsibilities then we can provide the following solution:

If connection management and data transfer are considered separate responsibilities then we can provide the following solution:

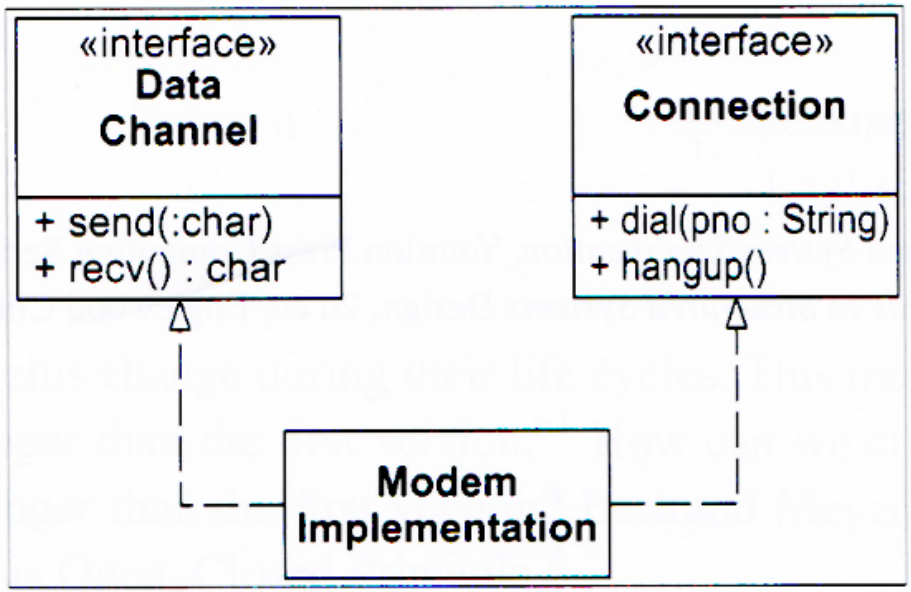


**Figure 8-3** Separated Modem Interface

If connection management and data transfer are considered separate responsibilities then we can provide the following solution:

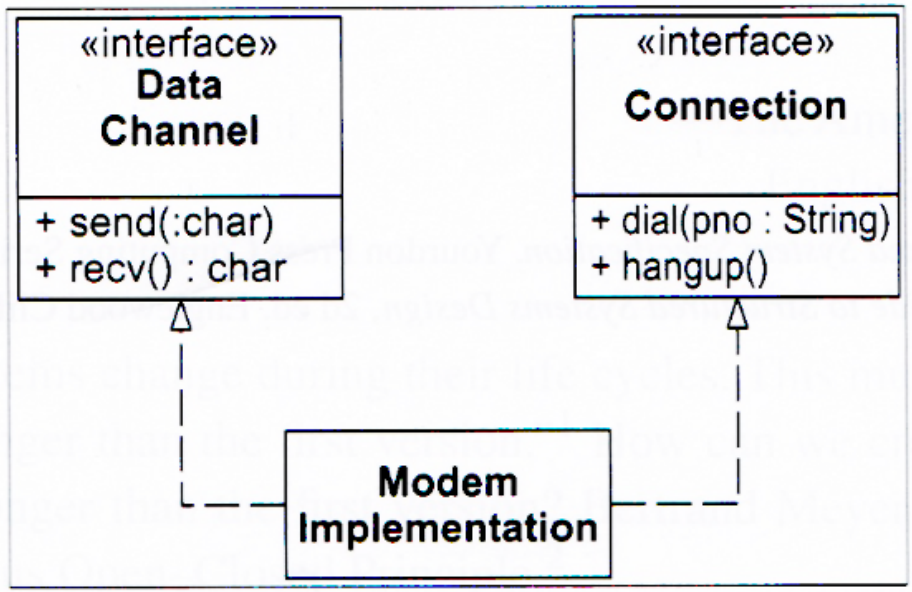


**Figure 8-3** Separated Modem Interface

However, what if connection management and data transfer always change together? Then Modem has only one reason for change and can be left as is. To modify Modem in this case would smell of **needless complexity**.

[Figure repeated]

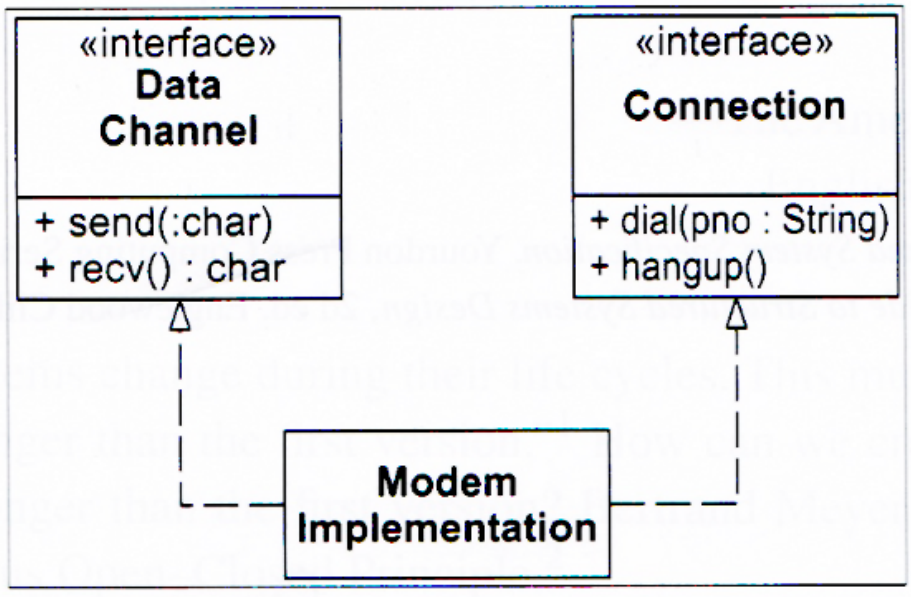


**Figure 8-3** Separated Modem Interface

[Figure repeated]

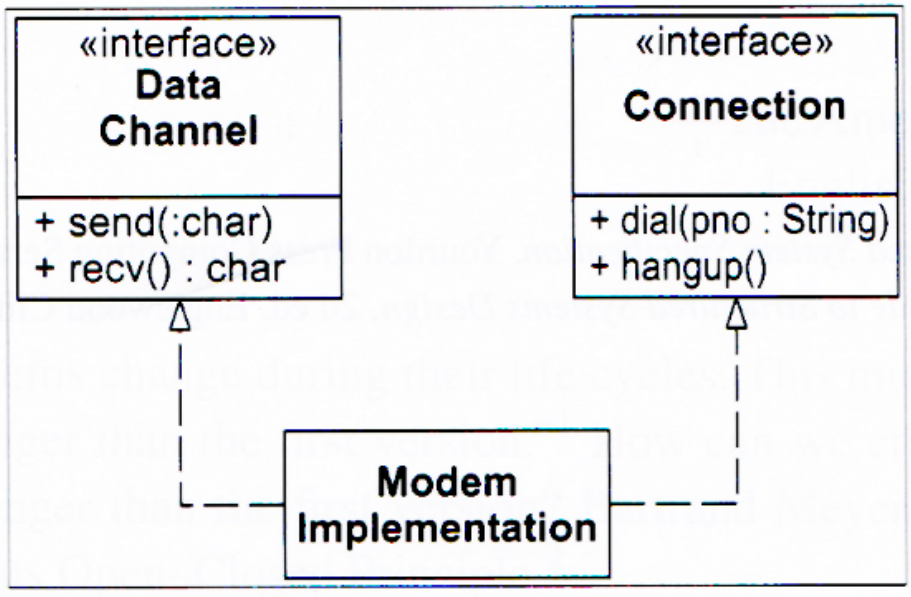


**Figure 8-3** Separated Modem Interface

Consider our solution again. Modem Implementation has two responsibilities! Isn't this bad? Yes but...

[Figure repeated]

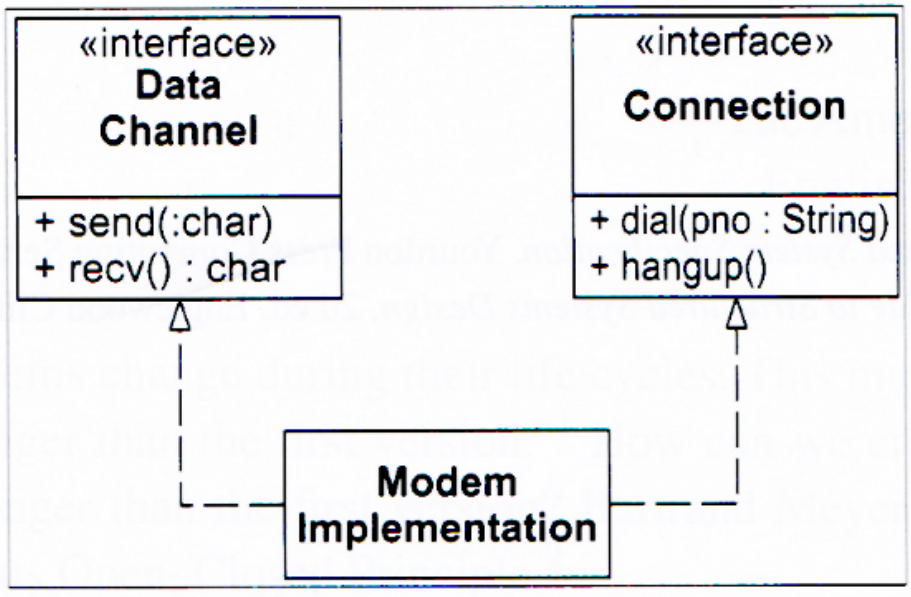


**Figure 8-3** Separated Modem Interface

Consider our solution again. Modem Implementation has two responsibilities! Isn't this bad? Yes but...

- This may be unavoidable due to h/w or OS constraints
- Even if Modem Implementation changes, other classes in the system should remain unaffected

# The Open-Closed Principle (OCP)

*Software entities (classes, modules, functions, etc.)*
*should be open for extension, but closed for modification.*

# The Open-Closed Principle (OCP)

*Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.*

*OR*
*To change behaviour, add new code rather than changing existing code.*

# The Open-Closed Principle (OCP)

*Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.*

*OR*
*To change behaviour, add new code rather than changing existing code.*

How? Abstraction.

# e.g. Client Server

With regards to the Client, the following design does not conform to the OCP.

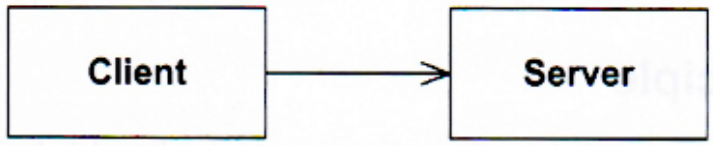


**Figure 9-1** Client is not open and closed

# e.g. Client Server

With regards to the Client, the following design does not conform to the OCP.

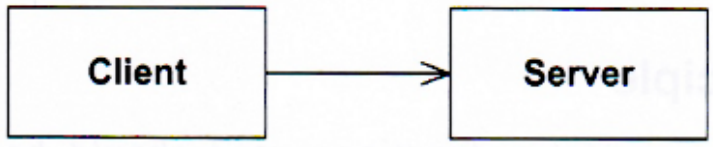


**Figure 9-1** Client is not open and closed

If we want the Client to use a different Server, we must change the Client. However, the following design resolves this problem:

# e.g. Client Server

With regards to the Client, the following design does not conform to the OCP.

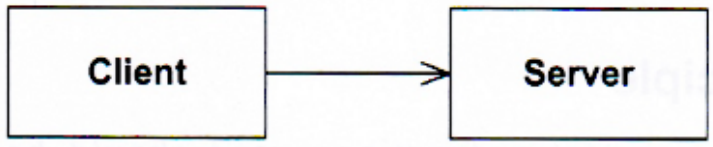


**Figure 9-1** Client is not open and closed

If we want the Client to use a different Server, we must change the Client. However, the following design resolves this problem:

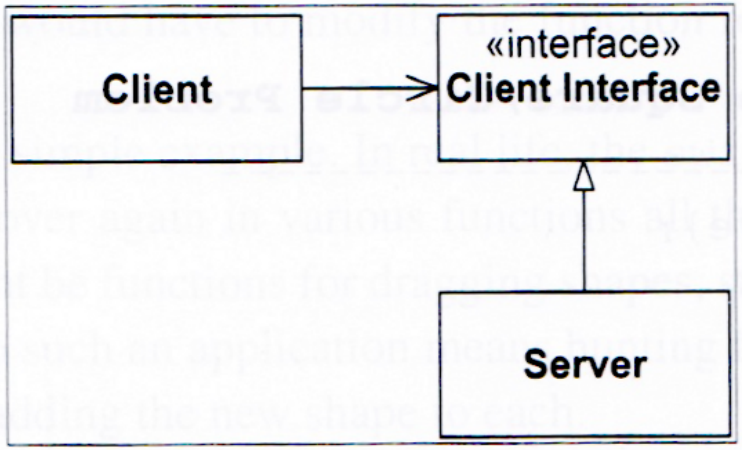


**Figure 9-2** STRATEGY pattern: Client is both open and closed

The DrawShape function violates the OCP:

The DrawShape function violates the OCP:

```
class Shape {
    enum ShapeType {SQUARE, CIRCLE} itsType;
    Shape(ShapeType t) : itsType(t) {}
};

class Circle : public Shape {
    Circle() : Shape(CIRCLE) {};
    void Draw();
    //...
};

class Square : public Shape {
    Square() : Shape(SQUARE) {};
    void Draw();
    //...
};

void DrawShape(const Shape& s) {
    if (s.itsType == Shape::SQUARE)
        static_cast<const Square&>(s).Draw();
    else if (s.itsType == Shape::CIRCLE)
        static_cast<const Circle&>(s).Draw();
}
```

The DrawShape function violates the OCP:

```
class Shape {
    enum ShapeType {SQUARE, CIRCLE} itsType;
    Shape(ShapeType t) : itsType(t) {}
};

class Circle : public Shape {
    Circle() : Shape(CIRCLE) {};
    void Draw();
    //...
};

class Square : public Shape {
    Square() : Shape(SQUARE) {};
    void Draw();
    //...
};

void DrawShape(const Shape& s) {
    if (s.itsType == Shape::SQUARE)
        static_cast<const Square&>(s).Draw();
    else if (s.itsType == Shape::CIRCLE)
        static_cast<const Circle&>(s).Draw();
}
```

New derivatives of Shape require changes to DrawShape.

The use of virtual methods solves this problem:

The use of virtual methods solves this problem:

```
class Shape {
public:
    virtual void Draw() const = 0;
};

class Square : public Shape {
public:
    virtual void Draw() const;
    //...
};

class Circle : public Shape {
public:
    virtual void Draw() const;
    //...
};

void DrawShape(const Shape& s) {
    s.Draw();
}
```

# References

Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003.