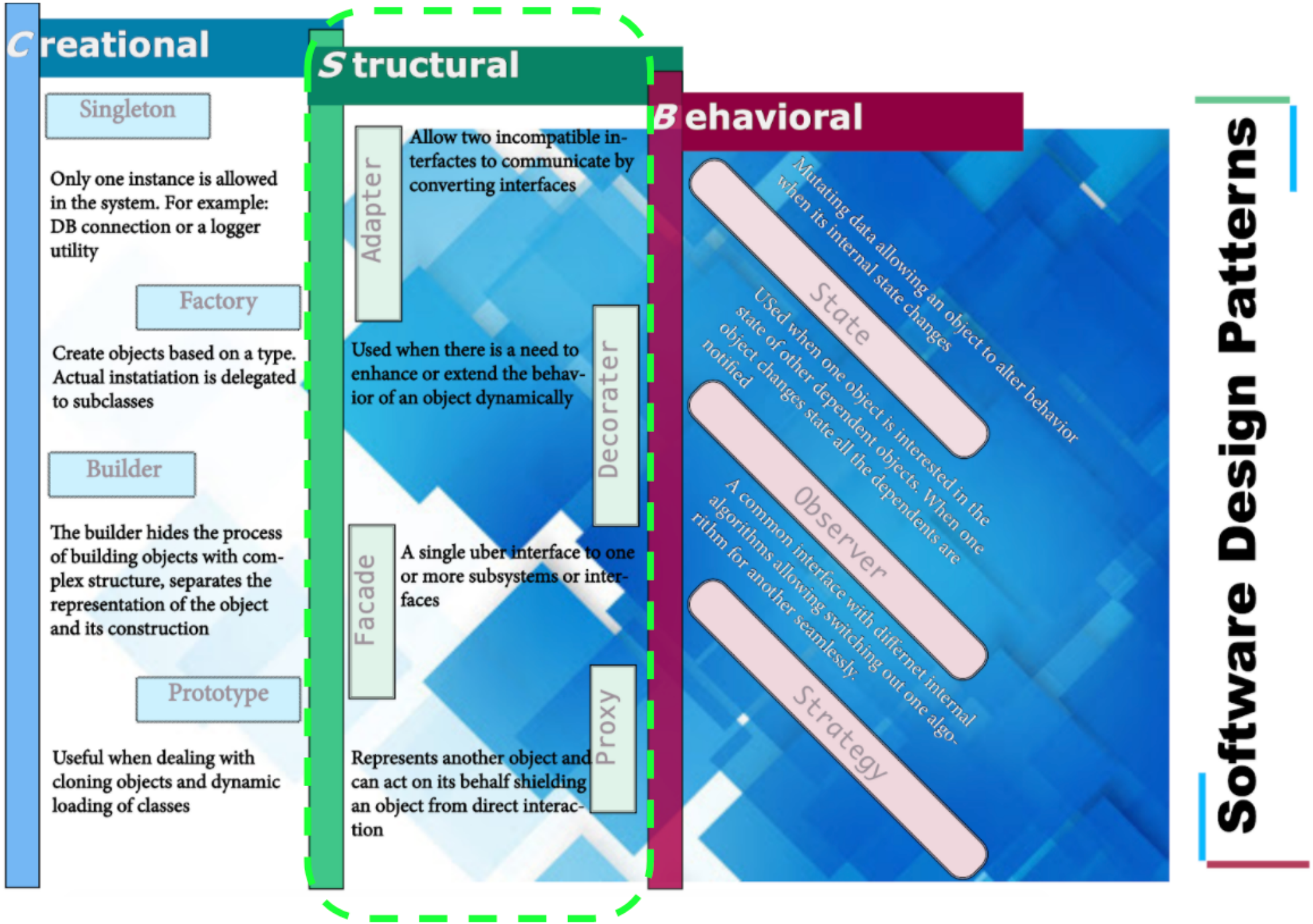
# Software Design Patterns: Session Four

Sahar Mostafa
UC Berkeley Extension
Summer 2020

# Week In Review

1. Creational Design Patterns
2. Singleton Pattern
3. Factory Pattern
4. Builder Pattern
5. Prototype Pattern
6. Exercises


Creational
Singleton
Only one instance is allowed in the system. For example: DB connection or a logger utility
Factory
Create objects based on a type. Actual instatiation is delegated to subclasses
Builder
The builder hides the process of building objects with complex structure, separates the representation of the object and its construction
Prototype
Useful when dealing with cloning objects and dynamic loading of classes
Structural
Adapter
Allow two incompatible interfactes to communicate by converting interfaces
Decorater
Used when there is a need to enhance or extend the behavior of an object dynamically
Facade
A single uber interface to one or more subsystems or interfaces
Proxy
Represents another object and can act on its behalf shielding an object from direct interaction
Behavioral
State
Mutating data allowing an object to alter behavior when its internal state changes
Observer
Used when one object is interested in the state of other dependent objects. When one object changes state all the dependents are notified
Strategy
A common interface with differnet internal algorithms allowing switching out one algorithm for another seamlessly.
Software Design Patterns

# Structural Patterns

# Description

**Intent** - Different interfaces work together

System is composed of different classes as part of a larger structure

Allows independently developed class libraries to interact together

**Control over what gets created who creates and how it gets created**

Structure

# Adaptor Pattern

# Description

## Intent

**One interface (adaptee) conform to  the client interface**

## Motivation

**Same construction can be used for different representations Creating a complex object independently**

Reusable class cooperates with unrelated or unforseen classes

## Consequences

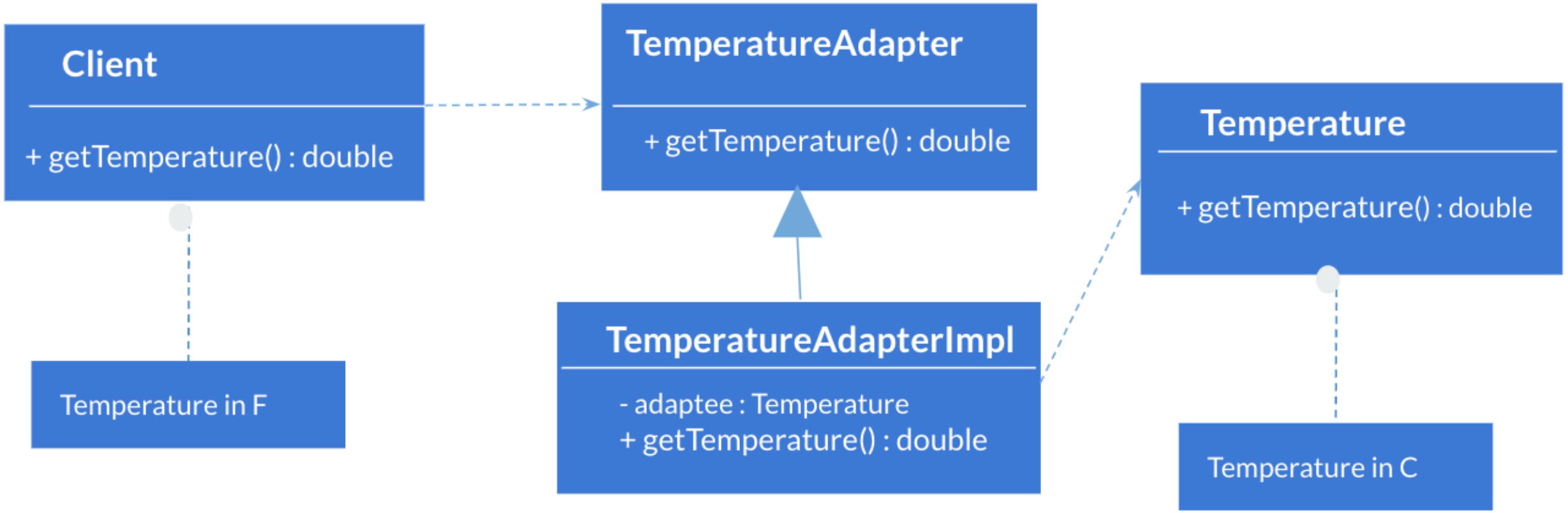
**Class to override some of the adaptee(original interface) behavior**

Consider class adapter(inheritance)  vs object adapter(composition)

## Use cases

**Formatting JSON strings from an incoming request to an object expected by a service**

# UML Diagram


Client
+ getTemperature() : double
TemperatureAdapter
+ getTemperature() : double
Temperature
+ getTemperature() : double
Temperature in F
TemperatureAdapterImpl
- adaptee : Temperature
+ getTemperature() : double
Temperature in C

# Implementation

```
interface TemperatureAdapter {
  double getTemperature();
};

class TemperatureAdapterImpl implements TemperatureAdapter {
  private Temperature temperature;

  @Override
  public double getTemperature() {
    return convertToFahrenheit(temperature.getTemperature());
  }

  private convertToFahrenheit(double val) {
    return (9/5) * val + 32;
 }
}
```

Structure

# Decorator Pattern

# Description

Intent

**Attach additional responsibilities dynamically**

Motivation

**Extend functionality without subclassing**

Change a class behavior from the outside

Consequences

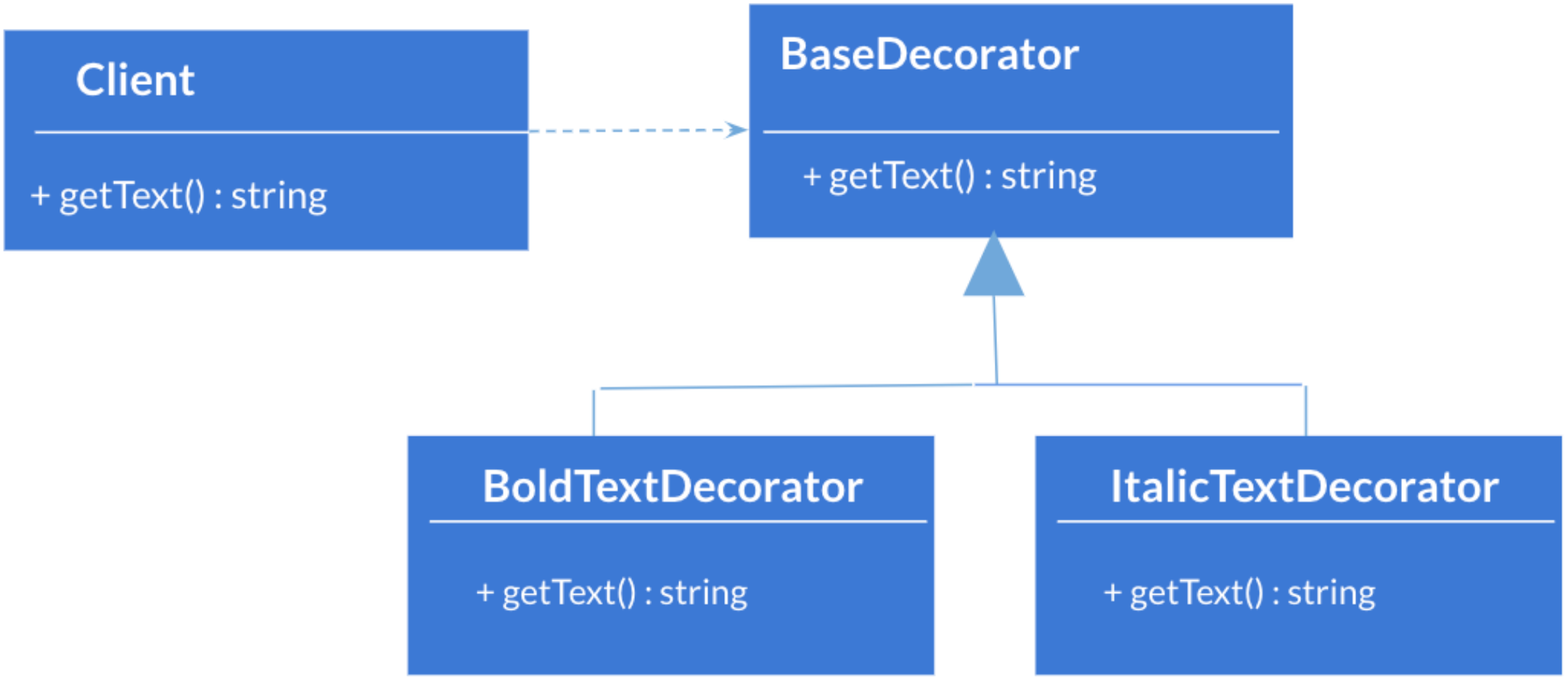
**Decorator must conform with the current component it is adding new responsibility to**

Components are isolated from decorators

Use cases

**From a base text, apply different formats: Bold, Italic, Underline, ... etc.**

# UML Diagram


Client
+ getText() : string
BaseDecorator
+ getText() : string
BoldTextDecorator
+ getText() : string
ItalicTextDecorator
+ getText() : string

# Implementation

```
public interface IText {
  string getText();
};

public abstract class TextBase : IText {
  private string _text;
  public TextBase(string text) {
    _text = text;
  }
  public virtual string getText() {
    return _text;
 }
}
```

```
public abstract class TextDecorator {
  private IText _itext;
  public TextDecorator(IText itext) { _itext =
text; }
  string getText() { return itext.getText(); }
};

public class BoldTextDecorator : TextDecorator
{
public BoldTextDecorator(IText itext) { }
public override string getText() {
    return "<b>" + itext.getText() + "</b>";
 }
}
```


15 mins Break

Structure

# Facade Pattern

# Description

Intent

Higher level interface providing a unified front to a client

Motivation

Unified simple interface hiding the complexity of internal subsystems

Consequences

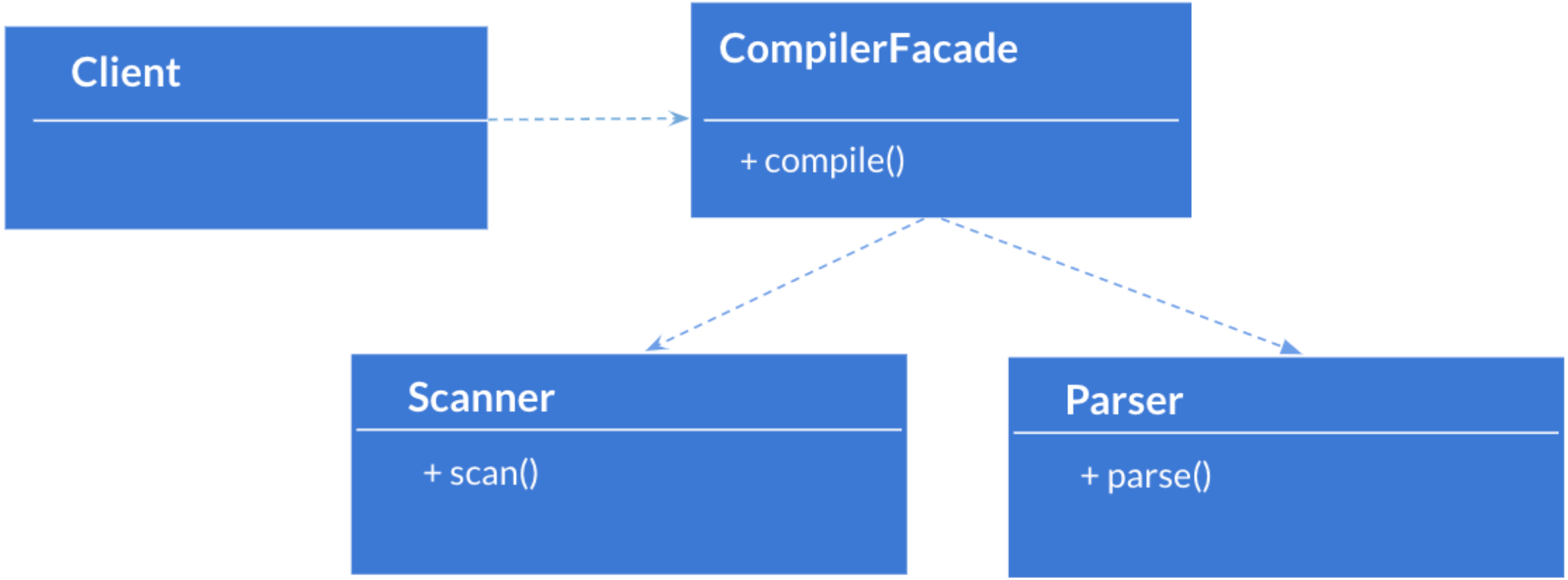
Shields client from subsystem components weak coupling

Subsystems can communicate via the facade

Use cases

Different internal storages, loggers or utilities

# UML Diagram



# Implementation

```
class Compiler {
public:
  compiler();
  void compile(istream& code);
};

void Compiler::compile(istream& code) {
 Scanner scanner;
 Parser parser;
 CodeGenerator codeGenerator;
 BytecodeStream output = parser.parse(scanner, code);
 codeGenerator.generate(output);
}
```

Structure

# Proxy Pattern

# Description

## Intent

A surrogate for another object to control access to

## Motivation

Remote proxy: local representation for an object in a different space
Protection proxy: controls access
Virtual proxy: creates expensive objects on demand

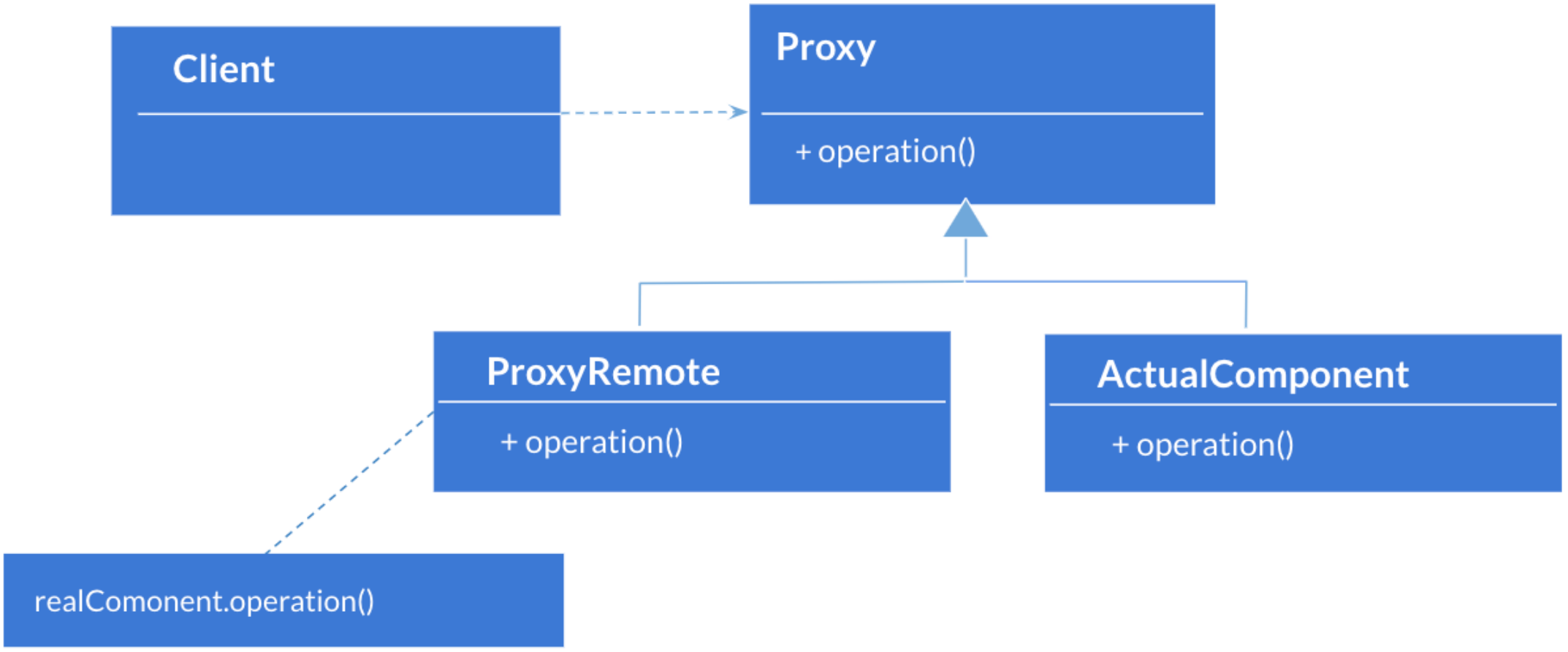
## Consequences

Perform optimization copy on write, hide different address space

## Use cases

Service proxy ensuring security

# UML Diagram

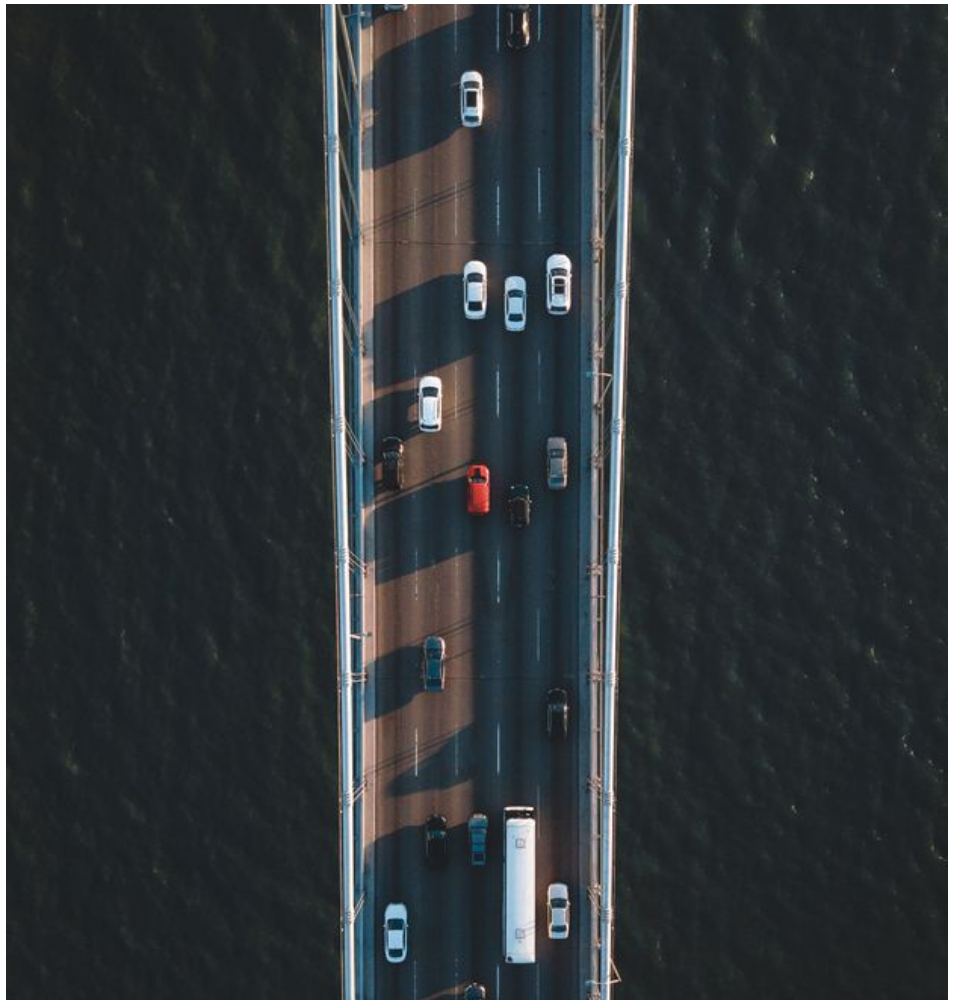

Client
Proxy
+ operation()
ProxyRemote
+ operation()
ActualComponent
+ operation()
realComonent.operation()

# Implementation

```
class Image:
   def __init__( self, filename ):
      self._filename = filename

   def load_image_from_disk( self ):
      print("loading " + self._filename )

class Proxy:
   def __init__( self, subject ):
      self._subject = subject
      self._proxystate = None

class ProxyImage( Proxy ):
   def display_image( self ):
      if self._proxystate == None:
         self._subject.load_image_from_disk()
         self._proxystate = 1
```

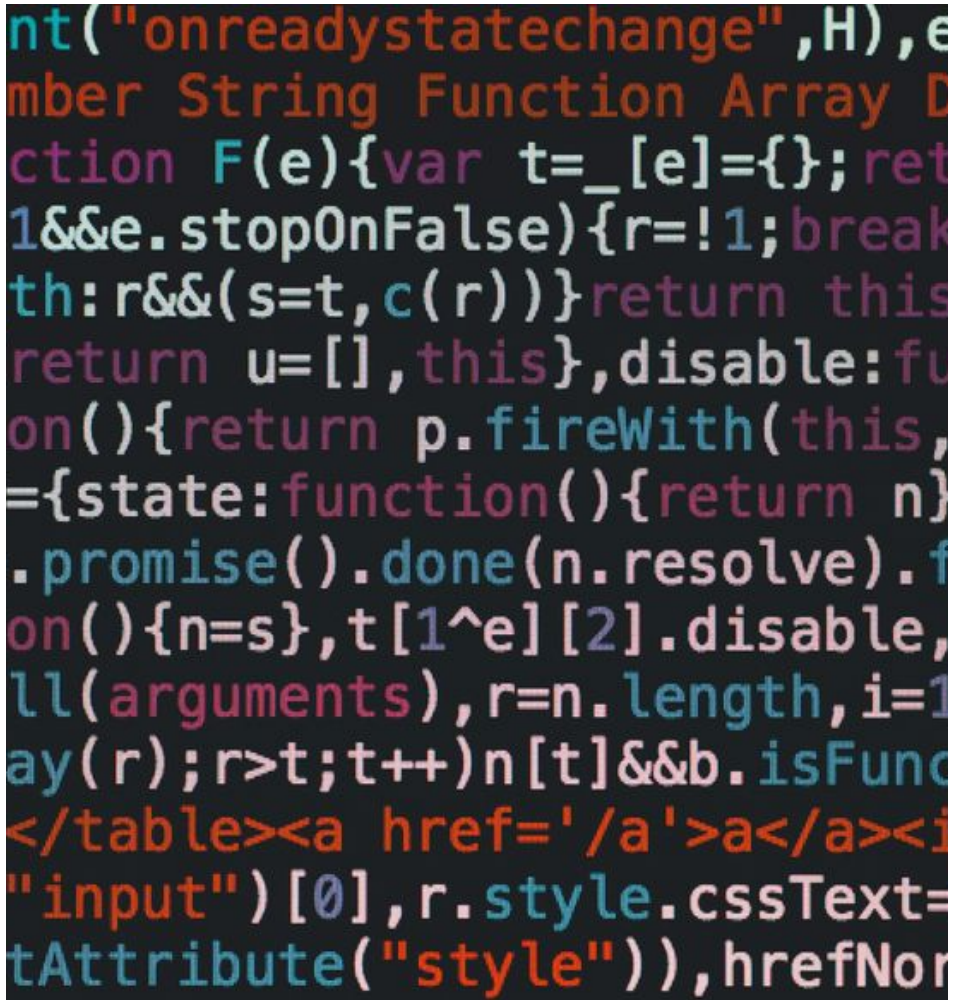
# Exercises

# Ordering System



- Design a car assembly ordering  system that creates cars with different trims, finishes and added on features. Luxury vs Sports car
- Which structural design pattern to use?
- Provide UML class diagram and test cases highlighting how the classes can be called

# Buffered Reader




- **Design a buffered reader that may read date data from different sources, for example from a file or from in memory buffer then format the data accordingly**

  If source is a file, format date data using ISO 8601 format: YY-MM-DD, else formate date data YY/MM/DD

- **Which structural design pattern to use?**

- **Provide UML class diagram and test cases highlighting how the classes can be called**

# Code Review

```
class Stack<E> extends Vector<E> {
 public void push(E item) {}
 public E pop() {}
}


public void main(int[] args) {
 Stack<Integer> stack = new Stack<Integer>();
 stack.push(1);
 stack.push(2);
 stack.push(3);
 stack.push(4);
 system.out.println(stack);
 Collections.shuffle(stack)
 system.out.println(stack);
 stack.remove(2);
}
```

# Next Week

# Session Five

1. Review Session Four
2. Behavioral Design Patterns
3. Strategy Pattern
4. Observer Pattern
5. State Pattern
6. Exercises