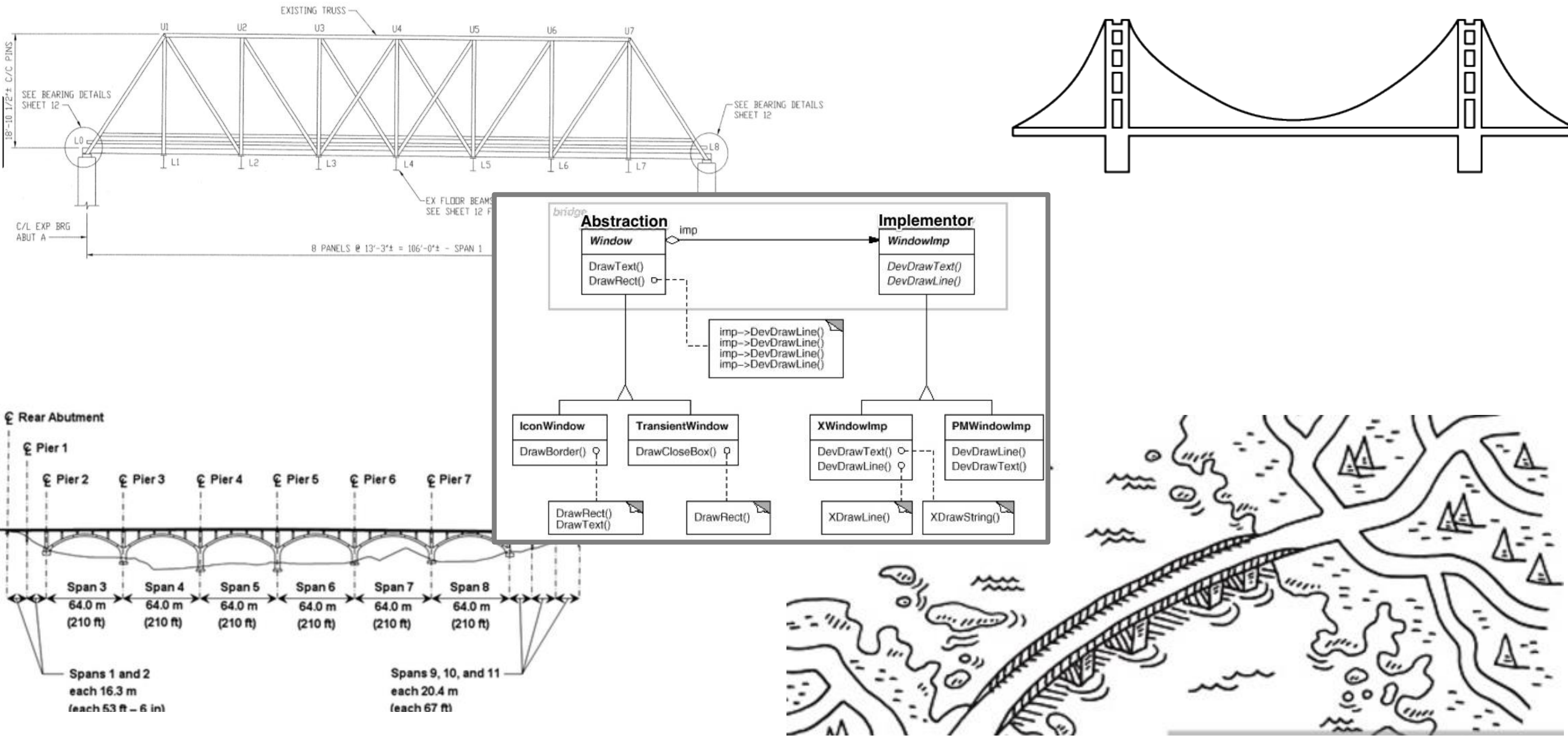
Samsung Electronics / DS

Solution Expert Training Program



Advanced
Design Patterns

August 2017

Soo Dong Kim, Ph.D.
Professor, School of Software
Soongsil University, Seoul, Korea
Office 02-820-0909 Mobile 010-7392-2220
sdkim777@gmail.com http://soft.ssu.ac.kr

Contents (1)

Part 1. Introduction

Part 2. Creation Patterns

Contents (3)

Part 3. Structural Patterns

Contents (4)

Part 4. Behavioral Patterns

Contents (5)

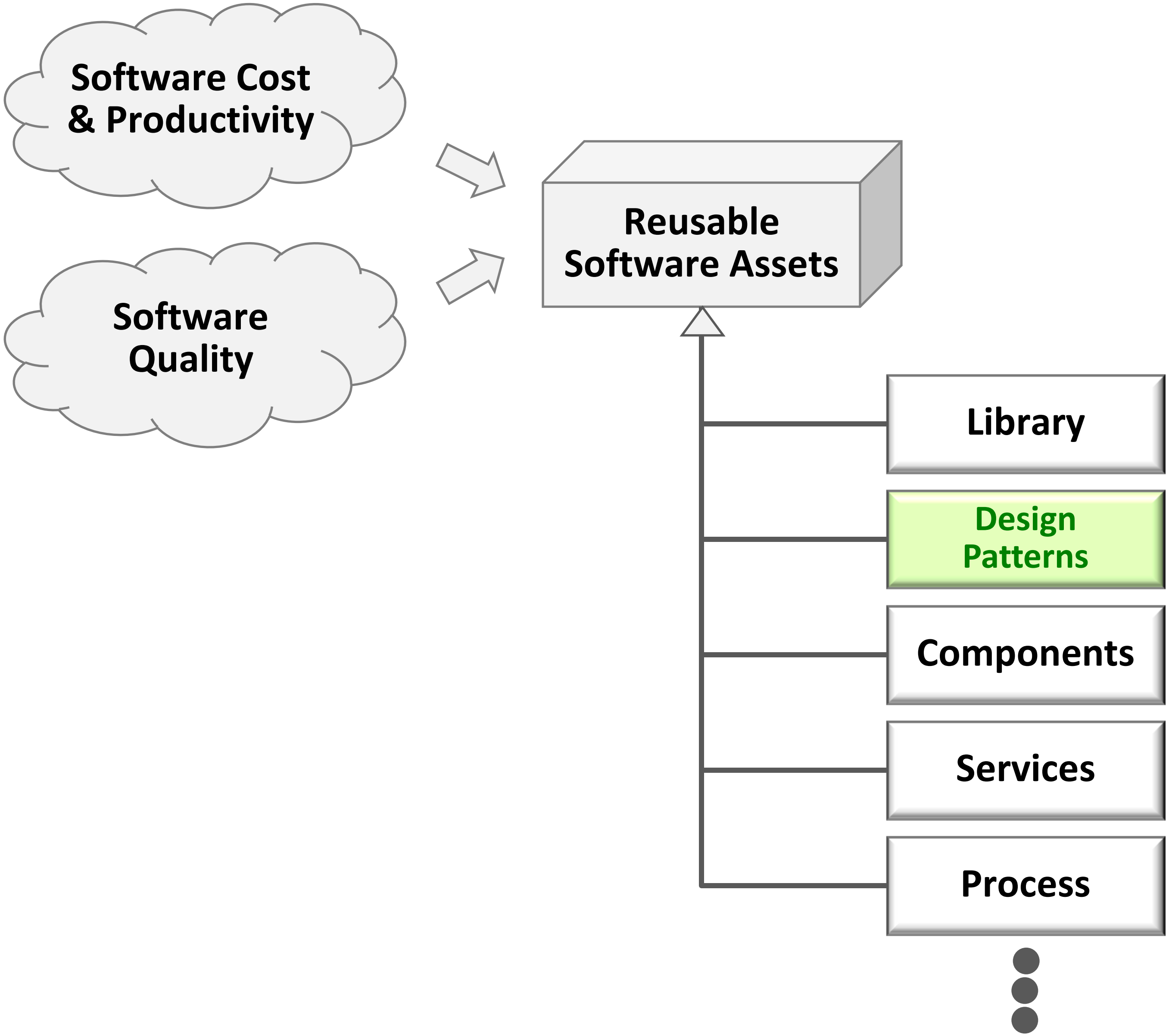
Part 5. Hardware Patterns

Part 6. Software Architecture

Principles of Design Patterns

Unit 1

Motivation



Software Cost & Productivity

Software Quality

Reusable Software Assets

Library

Design Patterns

Components

Services

Process

Reuse Assets

- **Code Reuse**
- **Design Reuse → Design Patterns**
- **Analysis Reuse**
- **Component Reuse → CBD**
- **Service Reuse**
  - Cloud Service
  - Micro Services
- **Framework Reuse**
- **Architecture Reuse**
- **Knowledge Reuse**
  - Ontology

Patterns

- **Patterns**
  - To capture the static and dynamic structures and collaborations of successful solutions to problems that arise when building applications in a particular domain
- **Architecture Patterns**
  - Architecture Styles
- **Design Patterns**

Design Patterns

- **To represent solutions to problems that arise when developing software within a particular context**
  - Pattern = Problem/Solution pair in a Context
- **To capture the static and dynamic structure and collaboration among key participants in software designs**
- **To facilitate reuse of successful software architectures and designs**

Design Pattern Description

- **Main parts**
  - Name and intent
  - Problem and context
  - Force(s) addressed
  - Abstract description of structure and collaborations in solution
  - Positive and negative consequence(s) of use
  - Implementation guidelines and sample code
  - Known uses and related patterns
- **Pattern descriptions are often independent of programming language or implementation details.**
  - Contrast with frameworks

Classifications of Design Patterns

- **Creational Patterns**
  - Deal with initializing and configuring classes and objects
- **Structural Patterns**
  - Deal with decoupling interface and implementation of classes and Objects
- **Behavioral Patterns**
  - Deal with dynamic interactions among societies of classes and objects

5 Creational Patterns

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

7 Structural Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

11 Behavioral Patterns

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

When to Use Patterns

- **Solutions to problems that recur with variations**
  - No need for reuse if the problem only arises in one context
- **Solutions that require several steps**
  - Patterns can be overkill if solution is simple linear set of instructions
- **Solutions where the solver is more interested in the existence of the solution than its complete derivation**
  - Patterns leave out too much to be useful to someone who really wants to understand.

Principles of Design Patterns

- **Design patterns are devised with 3 principles.**
- **Principle 1**
  - Separate interface from implementation
- **Principle 2**
  - Allow substitution of variable implementations via a common interface.
- **Principle 3**
  - Determine what is common and what is variable with an interface and an implementation
    - Common ⇔ Stable
    - Variable ⇔ Unstable, To be resolved
  - Open Closed Principle (OCP)

Open/Closed Principle

- **Determining Common vs. Variable Features**
  - Insufficient variation makes it hard for users to customize applications.
- **Components should be:**
  - The design of variable features should be open for customization and extension.
  - The design of common features should be closed for modification.
    - Cannot be modified.

Benefits of Design Patterns

- **Utilizing *expert knowledge* embedded on design patterns**
- **Promoting *effective communication* among developers**
- **Assisting better quality object-oriented design with**
  - High Modularity
  - High Readability
  - High Modifiability
  - High Extendibility

Abstract Factory

Unit 2

Intent

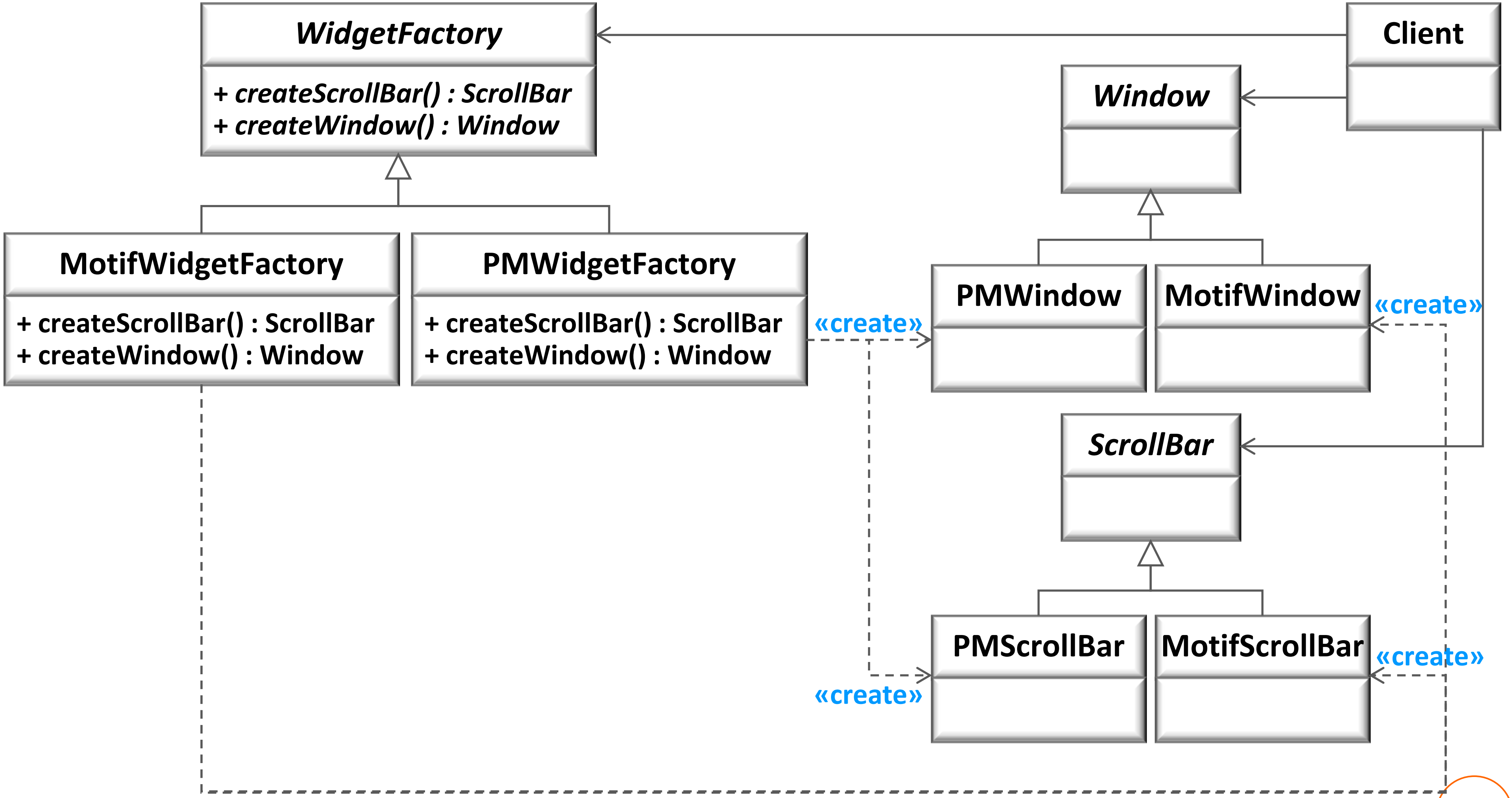
- **Provide an interface for creating families of related or dependent objects without specifying their concrete classes.**
- **Similar to the Factory Method pattern, but different**
  - With the Abstract Factory pattern, a class delegates the responsibility of object instantiation to another object via composition.
  - The Factory Method pattern uses inheritance and relies on a subclass to handle the desired object instantiation.
- **Actually, the delegated object frequently uses factory methods to perform the instantiation!**

Motivation (1)

- **Different look-and-feels define different appearances and behaviors for user interface widgets like scroll bars, windows, and buttons.**
  - To be portable across look-and-feel standards, an application should not hard-code its widgets for a particular look and feel.
  - Instantiating look-and-feel-specific classes of widgets makes the application <u>hard to change</u> the look and feel later.

Motivation (2)

- A GUI toolkit that supports multiple look-and-feels:

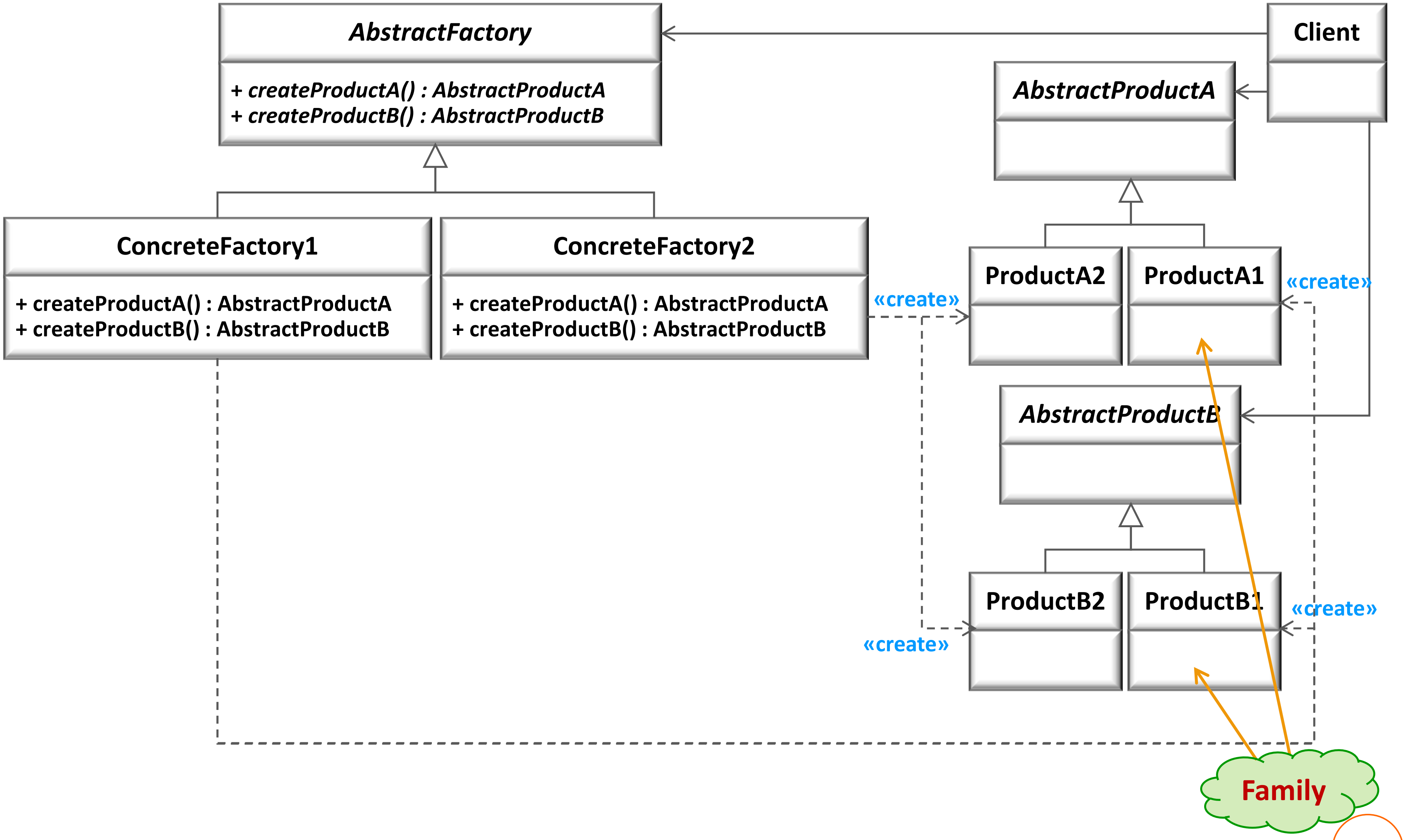


WidgetFactory
+ createScrollBar() : ScrollBar
+ createWindow() : Window
Client
Window
MotifWidgetFactory
+ createScrollBar() : ScrollBar
+ createWindow() : Window
PMWidgetFactory
+ createScrollBar() : ScrollBar
+ createWindow() : Window
«create»
PMWindow
MotifWindow
«create»
ScrollBar
PMScrollBar
MotifScrollBar
«create»
«create»

Applicability

- **Use the Abstract Factory pattern in any of the following situations:**
  - A system should be independent of how its products are created, composed, and represented.
  - A class can't anticipate the class of objects it must create.
  - A system must use just one of a set of products families.
  - A family of related product objects is designed to be used together, and you need to enforce this constraint.
    - i.e. PMWindow and PMScrollBar

Structure


AbstractFactory
+ createProductA() : AbstractProductA
+ createProductB() : AbstractProductB
Client
AbstractProductA
ConcreteFactory1
+ createProductA() : AbstractProductA
+ createProductB() : AbstractProductB
ConcreteFactory2
+ createProductA() : AbstractProductA
+ createProductB() : AbstractProductB
«create»
ProductA2
ProductA1
«create»
AbstractProductB
ProductB2
ProductB1
«create»
«create»
Family

Participants

- **AbstractFactory**
  - To declare an interface for operations that create abstract product objects
- **ConcreteFactory**
  - To implement the operations to create concrete product objects
- **AbstractProduct**
  - To declare an interface for a type of product objects
- **ConcreteProduct**
  - To define a product object to be created by the corresponding concrete factory
  - To implement the AbstractProduct interface
- **Client**
  - To use only interfaces declared by AbstractFactory and AbstractProduct classes

Collaborations

- **Normally a single instance of a ConcreteFactory class is created at runtime.**
    - Singleton Pattern is applied.
    - This concrete factory creates product objects having a particular implementation.
    - To create different product objects, clients should use a different concrete factory.
- **AbstractFactory defers creation of product objects to its ConcreteFactory.**

Consequences

- **Advantages**
  - To isolate clients from concrete implementation classes
  - To make exchanging product families easy
    - Since a particular concrete factory can support a complete family of products
  - To enforce the use of products only from one family
- **Disadvantages**
  - Supporting new kinds of products requires changing the AbstractFactory interface.

Implementation

- **Factories as Singleton**
  - An application typically needs only one instance of a ConcreteFactory per product family.
    - Use the Singleton pattern for this purpose
- **Creating the products**
  - The most common way to create products is to define a factory method for each product.
- **Defining extensible factories**
  - Adding a new kind of product requires changing the the AbstractFactory interface and all the classes that depend on it.
  - A more flexible but less safe design is to add a parameter to operations that create objects.

Sample Code (1)

```
class MazeFactory {
public:
    MazeFactory();
    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Wall* MakeWall() const
        { return new Wall; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
    virtual Door* MakeDoor(Room* r1, Room*
        r2) const
        { return new Door(r1, r2); }
};
```

```
Maze* MazeGame::CreateMaze
  (MazeFactory& factory) {
      Maze* aMaze = factory.MakeMaze();
      Room* r1 = factory.MakeRoom(1);
      Room* r2 = factory.MakeRoom(2);
      Door* aDoor = factory.MakeDoor(r1, r2);
      aMaze->AddRoom(r1);
      aMaze->AddRoom(r2);
      r1->SetSide(North, factory.MakeWall());
      …
      r2->SetSide(North, factory.MakeWall());
      …
      return aMaze;
}
```

Sample Code (2)

```
class EnchantedMazeFactory : public
    MazeFactory {
public:
    EnchantedMazeFactory();
    virtual Room* MakeRoom(int n) const
        { return new EnchantedRoom(n,
        CastSpell()); }
    virtual Door* MakeDoor(Room* r1, Room*
        r2) const
        { return new DoorNeedingSpell(r1, r2); }
protected:
    Spell* CastSpell() const;
};
```

```
Wall* BombedMazeFactory::MakeWall () const {
    return new BombedWall;
}
Room* BombedMazeFactory::MakeRoom(int n)
    const {
    return new RoomWithABomb(n);
}
```

Known Uses

- **InterViews**
  - To use the "Kit" suffix to denote AbstractFactory classes
    - WidgetKit
    - DialogKit
    - LayoutKit
- **ET++**
  - To use the Abstract Factory pattern to achieve portability across different window systems.

Related Patterns

- **Abstract Factory are often implemented with Factory Method or Prototype.**
- **Concrete Factory is often a Singleton.**

Builder

Unit 3

Intent

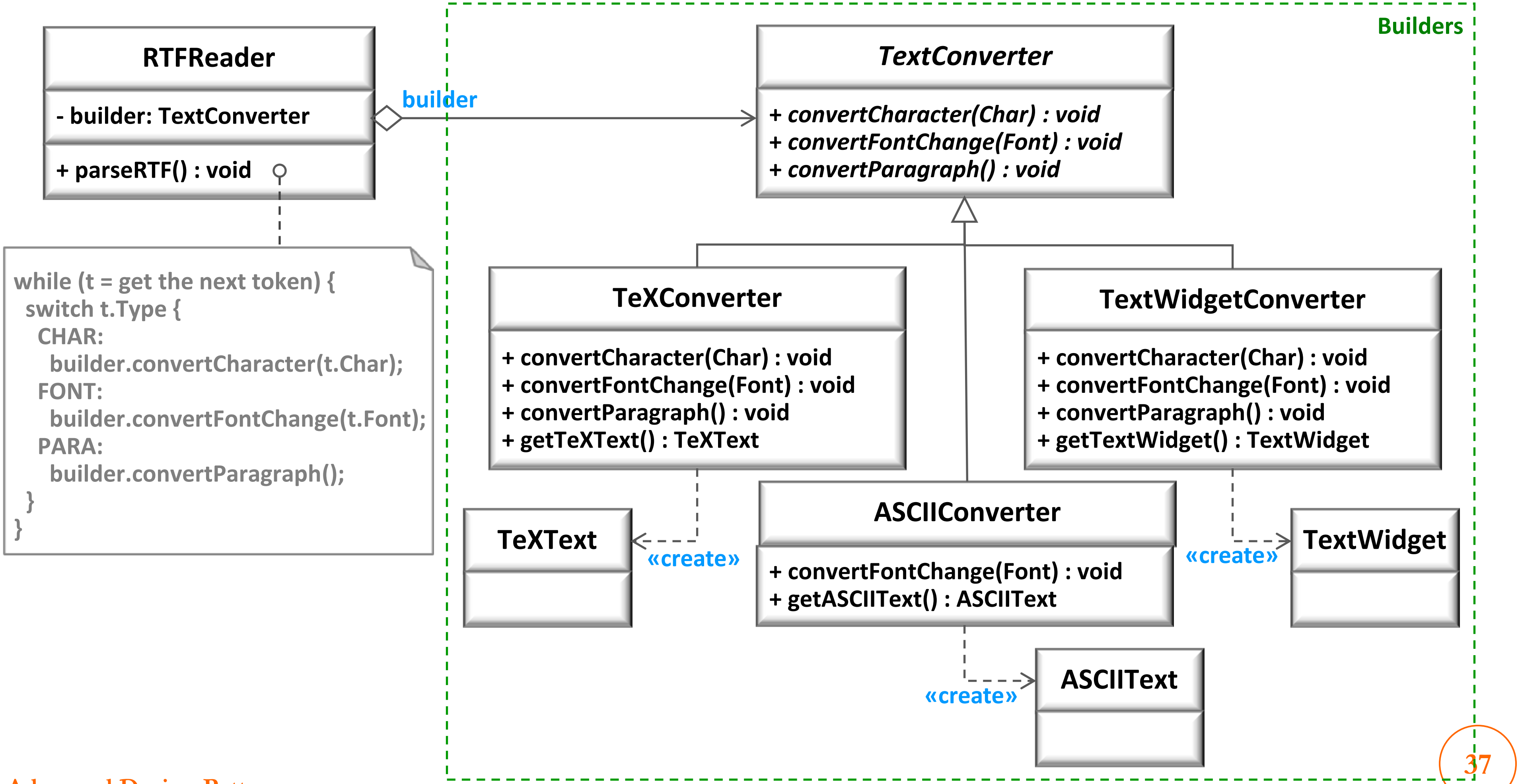
- Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Allow for the dynamic creation of objects based upon easily interchangeable algorithms.

Motivation (1)

- **A reader for the RTF (Rich Text Format) document exchange format should be able to convert RTF to many text formats such as plain ASCII text or text widget.**
  - The number of possible conversions is open-ended.
  - It should be easy to add a new conversion without modifying the reader.

Motivation (2)

- A solution to configuring the RTFReader class with a TextConverter object that converts RTF to another textual representation:



RTFReader
- builder: TextConverter
+ parseRTF() : void

```
while (t = get the next token) {
  switch t.Type {
    CHAR:
      builder.convertCharacter(t.Char);
    FONT:
      builder.convertFontChange(t.Font);
    PARA:
      builder.convertParagraph();
  }
}
```

Builders

TextConverter
+ convertCharacter(Char) : void
+ convertFontChange(Font) : void
+ convertParagraph() : void

TeXConverter
+ convertCharacter(Char) : void
+ convertFontChange(Font) : void
+ convertParagraph() : void
+ getTeXText() : TeXText

TextWidgetConverter
+ convertCharacter(Char) : void
+ convertFontChange(Font) : void
+ convertParagraph() : void
+ getTextWidget() : TextWidget

ASCIIConverter
+ convertFontChange(Font) : void
+ getASCIIText() : ASCIIText

TeXText

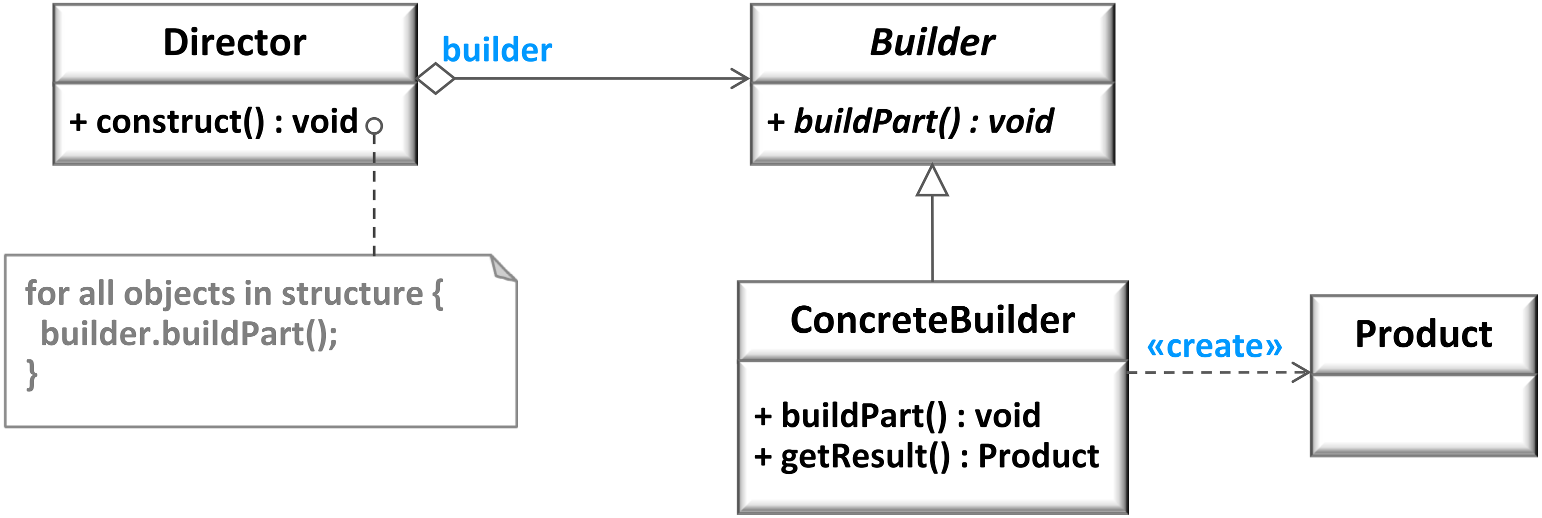
TextWidget

ASCIIText

Applicability

- **Use the Builder pattern when:**
  - The algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled.
  - The construction process must allow different representations for the object that's constructed.
  - The addition of new creation functionality without changing the core code is necessary.
  - Runtime control over the creation process is required.

Structure



```
for all objects in structure {
  builder.buildPart();
}
```

Participants (1)

- **Builder (TextConverter)**
  - To specify an abstract interface for creating parts of a Product object
- **ConcreteBuilder (ASCIIConverter, TeXConverter, TextWidgetConverter)**
  - To construct and assemble parts of the product by implementing the Builder interface
  - To define and keep track of the representation
  - To provide an interface for retrieving the product
    - e.g., GetASCIIText, GetTextWidget

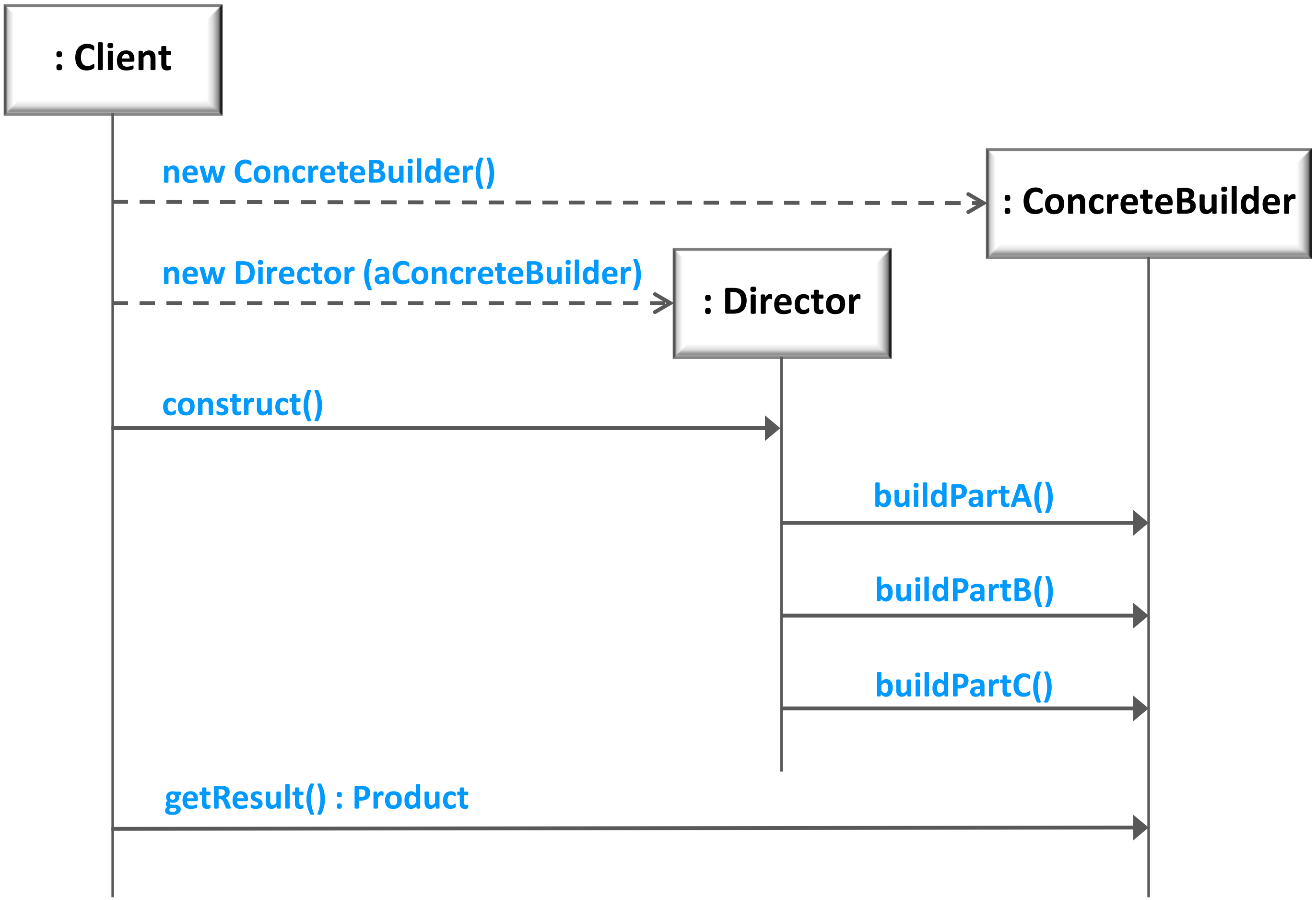
Participants (2)

- **Director (RTFReader)**
  - To construct an object using the Builder interface
- **Product (ASCIIText, TeXText, TextWidget)**
  - To represent the complex object under construction
  - ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.
  - To include classes that define the constituent parts, including interfaces for assembling the parts into the final result

Collaborations (1)

- The client creates the Director object and configures it with the desired Builder object.
- Director notifies the builder whenever a part of the product should be built.
- Builder handles requests from the director and adds parts to the product.
- The client retrieves the product from the builder.

Collaborations (2)



Consequences

- **To let you vary a product's internal representation**
  - Define a new kind of builder to change the product’s internal representation.
- **To isolate code for construction and representation**
  - Improves modularity by encapsulating the way a complex object is constructed and represented.
- **To give you finer control over the construction process**
  - Supports finer control over the construction process and consequently the internal structure of the resulting product.

Implementation

- **Assembly and construction interface**
  - Builder class interface must be general enough to allow the construction of products for all kinds of concrete builders.
- **Why no abstract class for products?**
  - Products produced by the concrete builders differ so greatly in their representation that there is little to gain from giving different products a common parent class.
- **Empty methods as default in Builder**
  - Let clients override only the operations they're interested in.

Sample Code (1)

```
class MazeBuilder {
  public:
     virtual void BuildMaze() { }
     virtual void BuildRoom(int room) { }
     virtual void BuildDoor(int roomFrom, int
   roomTo) { }

     virtual Maze* GetMaze() { return 0; }
  protected:
     MazeBuilder();
  };
```

```
Maze* MazeGame::CreateMaze (MazeBuilder&
    builder) {
     builder.BuildMaze();

     builder.BuildRoom(1);
     builder.BuildRoom(2);
     builder.BuildDoor(1, 2);

     return builder.GetMaze();
  }
```

Sample Code (2)

```
Maze* MazeGame::CreateComplexMaze
     (MazeBuilder& builder) {
       builder.BuildRoom(1);
       // ...
       builder.BuildRoom(1001);

       return builder.GetMaze();
   }
```

```
class StandardMazeBuilder : public MazeBuilder {
   public:
      StandardMazeBuilder();

      virtual void BuildMaze();
      virtual void BuildRoom(int);
      virtual void BuildDoor(int, int);

      virtual Maze* GetMaze();
   private:
      Direction CommonWall(Room*, Room*);
      Maze* _currentMaze;
   };
```

Known Uses

- **The RTF converter application is from ET++.**
  - Its text building block uses a builder to process text stored in the RTF format.
- **Smalltalk-80**
  - Parser class
  - ClassBuilder
  - ByteCodeStream
- **Adaptive Communications Environment**
  - Service Configurator framework uses a builder to construct network service components that are linked into a server at run-time.

Related Patterns

- **Abstract Factory is similar to Builder in that it too may construct complex objects.**
  - Builder pattern focuses on constructing a complex object step by step.
  - Abstract Factory's emphasis is on families of product objects (either simple or complex).
- **Composite is what the builder often builds.**

Factory Method

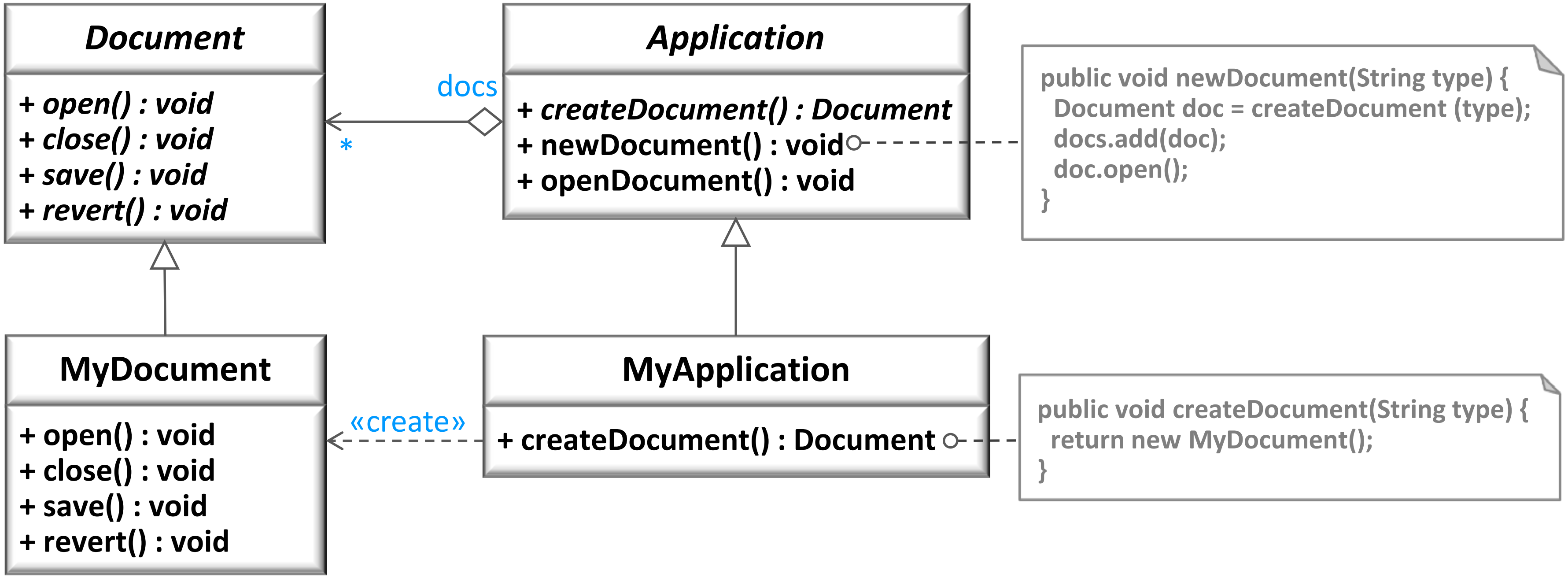
Unit 4

Intent

- Define an interface for creating an object, but let subclasses decide which class to instantiate.
- Factory Method lets a class defer instantiation to subclasses.

Motivation

- **Consider the following framework:**



```
public void newDocument(String type) {
  Document doc = createDocument (type);
  docs.add(doc);
  doc.open();
}
```

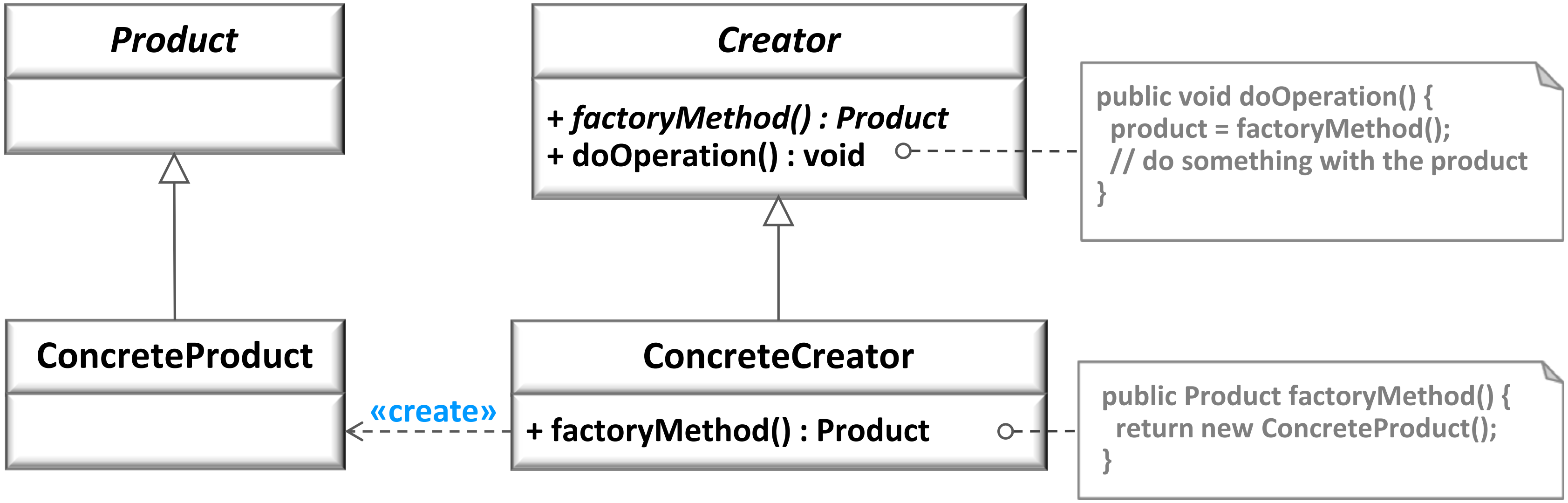
```
public void createDocument(String type) {
  return new MyDocument();
}
```

- **The createDocument( ) method is a factory method.**

Applicability

- **Use the Factory Method pattern in any of the following situations:**
  - A class can't anticipate the class of objects it must create
  - A class wants its subclasses to specify the objects it creates

Structure



```
public void doOperation() {
  product = factoryMethod();
  // do something with the product
}
```

```
public Product factoryMethod() {
  return new ConcreteProduct();
}
```

Participants

- **Product**
  - To define the interface for the type of objects the factory method creates
- **ConcreteProduct**
  - To implement the Product interface
- **Creator**
  - To declare the factory method, which returns an object of type Product
- **ConcreteCreator**
  - To override the factory method to return an instance of a ConcreteProduct

Collaborations

- Creator relies on its subclasses to implement the factory method so that it returns an instance of the appropriate ConcreteProduct.

Consequences

- **Advantages**
  - Code is made more flexible and reusable by the elimination of instantiation of application-specific classes.
  - Code deals only with the interface of the Product class and can work with any ConcreteProduct class that supports this interface.
- **Disadvantages**
  - Clients might have to subclass the Creator class just to instantiate a particular ConcreteProduct.

Implementation

- Creator can be abstract or concrete.
- Should the factory method be able to create multiple kinds of products? If so, then the factory method has a parameter (possibly used in an if-else!) to decide what object to create.

Sample Code (1)

```
class MazeGame {
  public:
     Maze* CreateMaze();

  // factory methods:

     virtual Maze* MakeMaze() const
        { return new Maze; }
     virtual Room* MakeRoom(int n) const
        { return new Room(n); }
     virtual Wall* MakeWall() const
        { return new Wall; }
     virtual Door* MakeDoor(Room* r1, Room*
   r2) const
        { return new Door(r1, r2); }
  };
```

```
Maze* MazeGame::CreateMaze () {
      Maze* aMaze = MakeMaze();
      Room* r1 = MakeRoom(1);
      Room* r2 = MakeRoom(2);
      Door* theDoor = MakeDoor(r1, r2);
      aMaze->AddRoom(r1);
      aMaze->AddRoom(r2);

      r1->SetSide(North, MakeWall());
      …

      r2->SetSide(North, MakeWall());
      …
      return aMaze;
   }
```

Sample Code (2)

```
class BombedMazeGame : public MazeGame {
  public:
    BombedMazeGame();

    virtual Wall* MakeWall() const
      { return new BombedWall; }

    virtual Room* MakeRoom(int n) const
      { return new RoomWithABomb(n); }
  };
```

```
class EnchantedMazeGame : public MazeGame {
  public:
    EnchantedMazeGame();

    virtual Room* MakeRoom(int n) const
      { return new EnchantedRoom
        (n, CastSpell()); }

    virtual Door* MakeDoor(Room* r1,
     Room* r2) const
      { return new DoorNeedingSpell(r1, r2); }
  protected:
    Spell* CastSpell() const;
  };
```

Known Uses

- **Factory methods pervade toolkits and frameworks.**
  - MacApp, ET++, Unidraw, etc.
- **Orbix ORB system from IONA Technologies**
  - Uses Factory Method to generate an appropriate type of proxy.

Related Patterns

- Abstract Factory is often implemented with factory methods.
- Factory methods are usually called within Template Methods.
- Prototypes often require an Initialize operation on the Product class.

Prototype

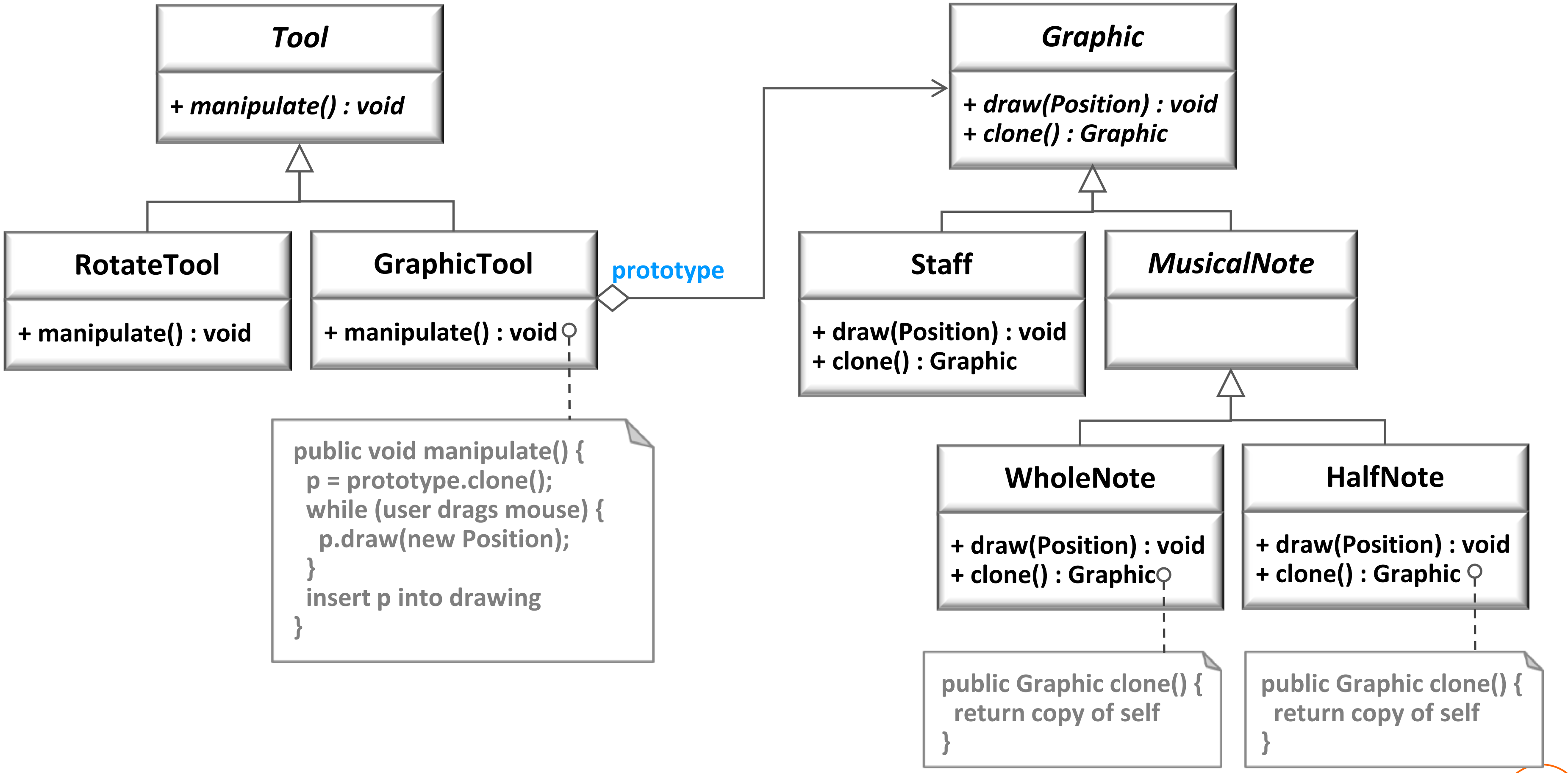
Unit 5

Intent

- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Motivation

- **Building an editor for music scores by customizing a general framework for graphical editors and adding new objects that represent notes, rests, and staves.**



```
public void manipulate() {
  p = prototype.clone();
  while (user drags mouse) {
    p.draw(new Position);
  }
  insert p into drawing
}
```

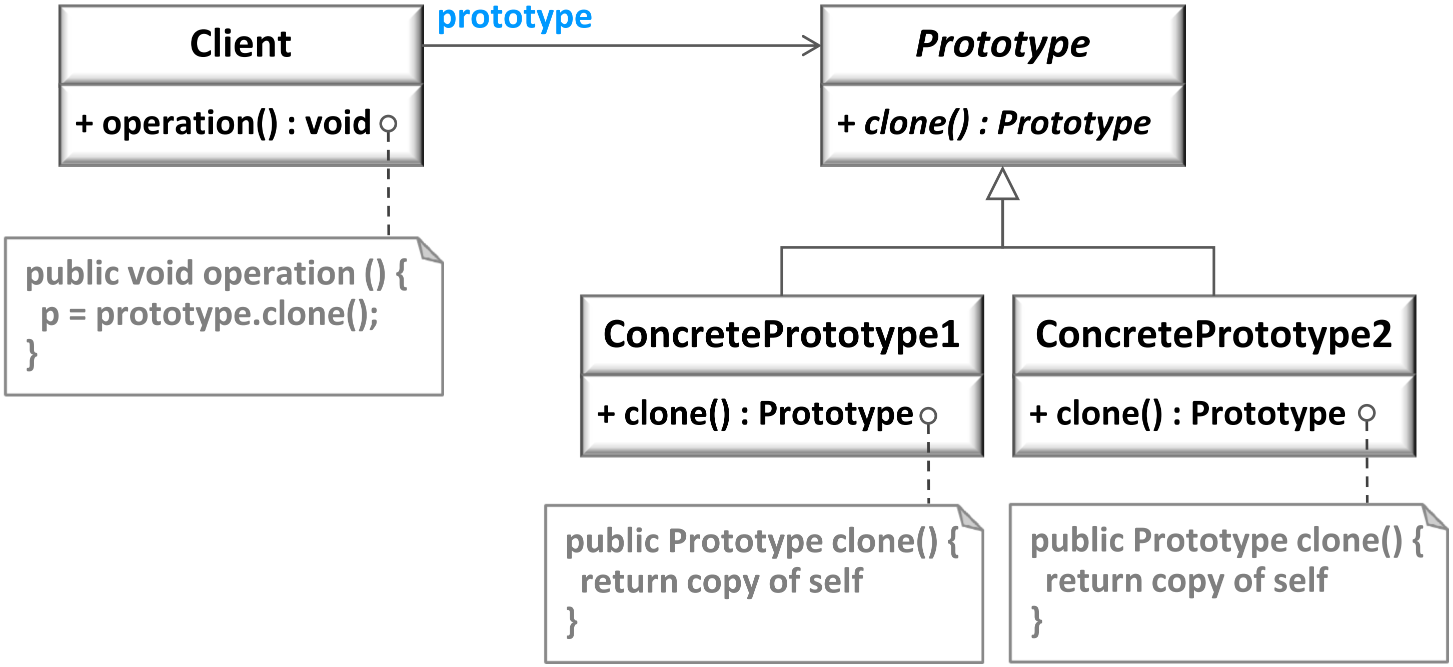
```
public Graphic clone() {
  return copy of self
}
```

```
public Graphic clone() {
  return copy of self
}
```

Applicability

- **Use the Prototype pattern when:**
  - A system should be independent of how its products are created, composed, and represented; *and*
  - The classes to instantiate are specified at run-time, for example, by dynamic loading, or to avoid building a class hierarchy of factories that parallels the class hierarchy of products
  - Instances of a class can have one of only a few different combinations of state.
    - It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

Structure



Client

+ operation() : void

prototype

Prototype

+ clone() : Prototype

ConcretePrototype1

+ clone() : Prototype

ConcretePrototype2

+ clone() : Prototype

```
public void operation () {
  p = prototype.clone();
}
```

```
public Prototype clone() {
  return copy of self
}
```

```
public Prototype clone() {
  return copy of self
}
```

Participants

- **Prototype (Graphic)**
  - To declare an interface for cloning itself
- **ConcretePrototype (Staff, WholeNote, HalfNote)**
  - To implement an operation for cloning itself
- **Client (GraphicTool)**
  - To create a new object by asking a prototype to clone itself

Collaborations

- A client asks a prototype to clone itself.

Consequences

- **Same consequences as Abstract Factory and Builder**
  - Hides the concrete product classes from the client, thereby reducing the number of names clients know about.
  - Let a client work with application-specific classes without modification.
- **Adding and removing products at run-time**
- **Specifying new objects by varying values**
- **Specifying new objects by varying structure**
- **Reduced subclassing**
- **Configuring an application with classes dynamically**

Implementation

- **Using a prototype manager.**
  - When the number of prototypes in a system isn't fixed (that is, they can be created and destroyed dynamically), keep a registry of available prototypes.
- **Implementing the Clone operation.**
  - Smalltalk; copy
  - C++; copy constructor
- **Initializing clones.**
  - We will want to initialize some or all of its internal state to values of their choosing.
  - Prototype classes already define operations for (re)setting key pieces of state.

Sample Code (1)

```
class MazePrototypeFactory : public MazeFactory
   {
  public:
    MazePrototypeFactory(Maze*, Wall*,
   Room*, Door*);
    virtual Maze* MakeMaze() const;
    virtual Room* MakeRoom(int) const;
    virtual Wall* MakeWall() const;
    virtual Door* MakeDoor(Room*, Room*)
   const;

  private:
    Maze* _prototypeMaze;
    Room* _prototypeRoom;
    Wall* _prototypeWall;
    Door* _prototypeDoor;
  };
```

```
MazePrototypeFactory::MazePrototypeFactory (
    Maze* m, Wall* w, Room* r, Door* d)  {
    _prototypeMaze = m;
    _prototypeWall = w;
    _prototypeRoom = r;
    _prototypeDoor = d;
  }
```

Sample Code (2)

```
Wall* MazePrototypeFactory::MakeWall ()
 const {
   return _prototypeWall->Clone();
}


Door* MazePrototypeFactory::MakeDoor
 (Room* r1, Room *r2) const {
   Door* door = _prototypezDoor->Clone();
   door->Initialize(r1, r2);
   return door;
}
```

```
MazeGame game;
MazePrototypeFactory simpleMazeFactory(
   new Maze, new Wall, new Room, new Door
);


Maze* maze =
 game.CreateMaze(simpleMazeFactory);



MazePrototypeFactory bombedMazeFactory(
   new Maze, new BombedWall,
   new RoomWithABomb, new Door
);
```

Known Uses

- **Ivan Sutherland's**
  - Sketchpad system
- **Mode Composer**
  - “Interaction technique library" stores prototypes of objects that support various interaction techniques

Related Patterns

- Abstract Factory
- Composite
- Decorator

Singleton

Unit 6

Intent

- **Ensure a class only has one instance, and provide a global point of access to it.**

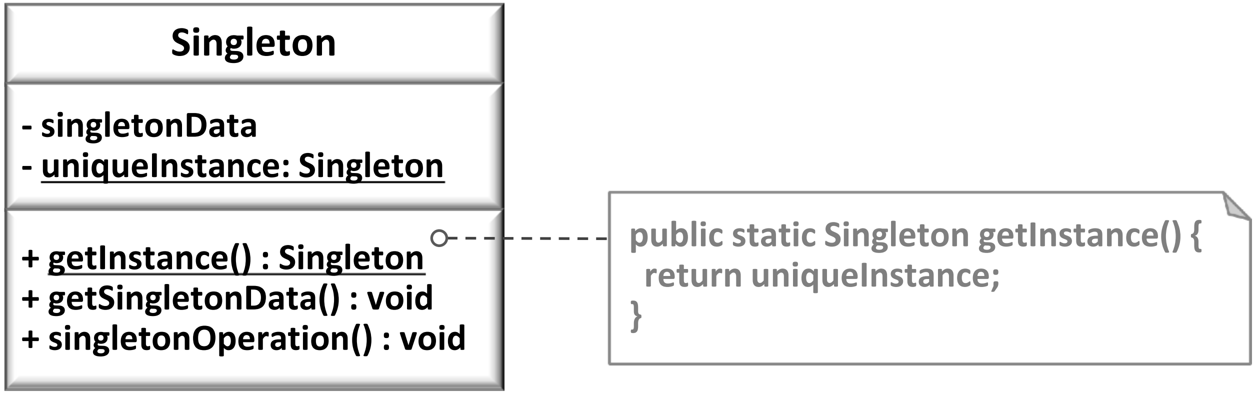
Motivation

- **Sometimes we want just a single instance of a class to exist in the system.**
    - For example, we want just one window manager. Or just one factory for a family of products.
- **We need to have that one instance easily accessible.**
- **And we want to ensure that additional instances of the class can not be created.**

Applicability

- **Use the Singleton pattern in any of the following situations:**
  - When there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
  - When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Structure



Singleton

- singletonData
- uniqueInstance: Singleton

+ getInstance() : Singleton
+ getSingletonData() : void
+ singletonOperation() : void

```
public static Singleton getInstance() {
  return uniqueInstance;
}
```

Participants

- **Singleton**
  - To define an Instance operation that lets clients access its unique instance
    - Instance is a class operation.
      - A class method in Smalltalk
      - A static member function in C++
  - May be responsible for creating its own unique instance.

Collaborations

- Clients access a Singleton instance solely through Singleton's Instance operation.

Consequences

- **Advantages**
  - Controlled access to sole instance
  - Permits a variable number of instances

Implementation

- **Ensuring a Unique Instance**
  - Use a static method to allow clients to get a reference to the single instance and use a private constructor.
  - Note that the singleton instance is only created when needed. This is called lazy instantiation.
- **Subclassing the Singleton Class**
  - Method 1: Have the Superclass instance() method determine the subclass to instantiate
  - Method 2: Have each subclass provide a static instance method()

Sample Code (1)

```
class MazeFactory {
        public:
                static MazeFactory* Instance();
                // existing interface goes here
        protected:
                MazeFactory();
        private:
                static MazeFactory* _instance;
};
```

```
MazeFactory* MazeFactory::_instance = 0;
MazeFactory* MazeFactory::Instance ( ) {
        if (_instance == 0) {
                _instance = new MazeFactory;
        }
        return _instance;
}
```

Sample Code (2)

```
MazeFactory* MazeFactory::Instance ( ) {
    if (_instance == 0) {
        const char* mazeStyle = getenv("MAZESTYLE");
        if (strcmp(mazeStyle, "bombed") == 0) {
          _instance = new BombedMazeFactory;
        } else if (strcmp(mazeStyle, "enchanted") == 0) {
          _instance = new EnchantedMazeFactory;
        // ... other possible subclasses
        } else { // default
          _instance = new MazeFactory;
        }
    }
    return _instance;
}
```

Known Uses

- **Smalltalk-80**
  - The set of changes to the code, which is ChangeSet current.
- **InterViews user interface toolkit**
  - Uses the Singleton pattern to access the unique instance of its Session and WidgetKit classes, among others.

Related Patterns

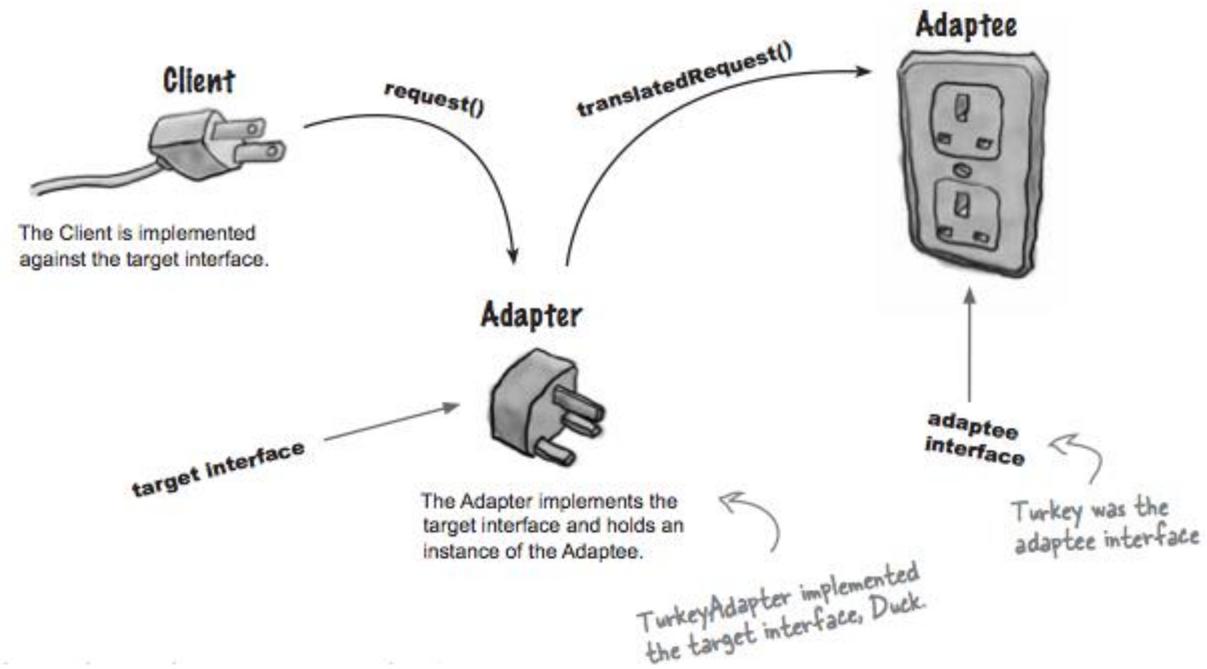
- **Many patterns can be implemented using the Singleton pattern.**
  - Abstract Factory
  - Builder
  - Prototype

Adapter

Unit 7

Intent

- **Convert the interface of a class into another interface clients expect.**
  - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



The Client is implemented against the target interface.

The Adapter implements the target interface and holds an instance of the Adaptee.

TurkeyAdapter implemented the target interface, Duck.

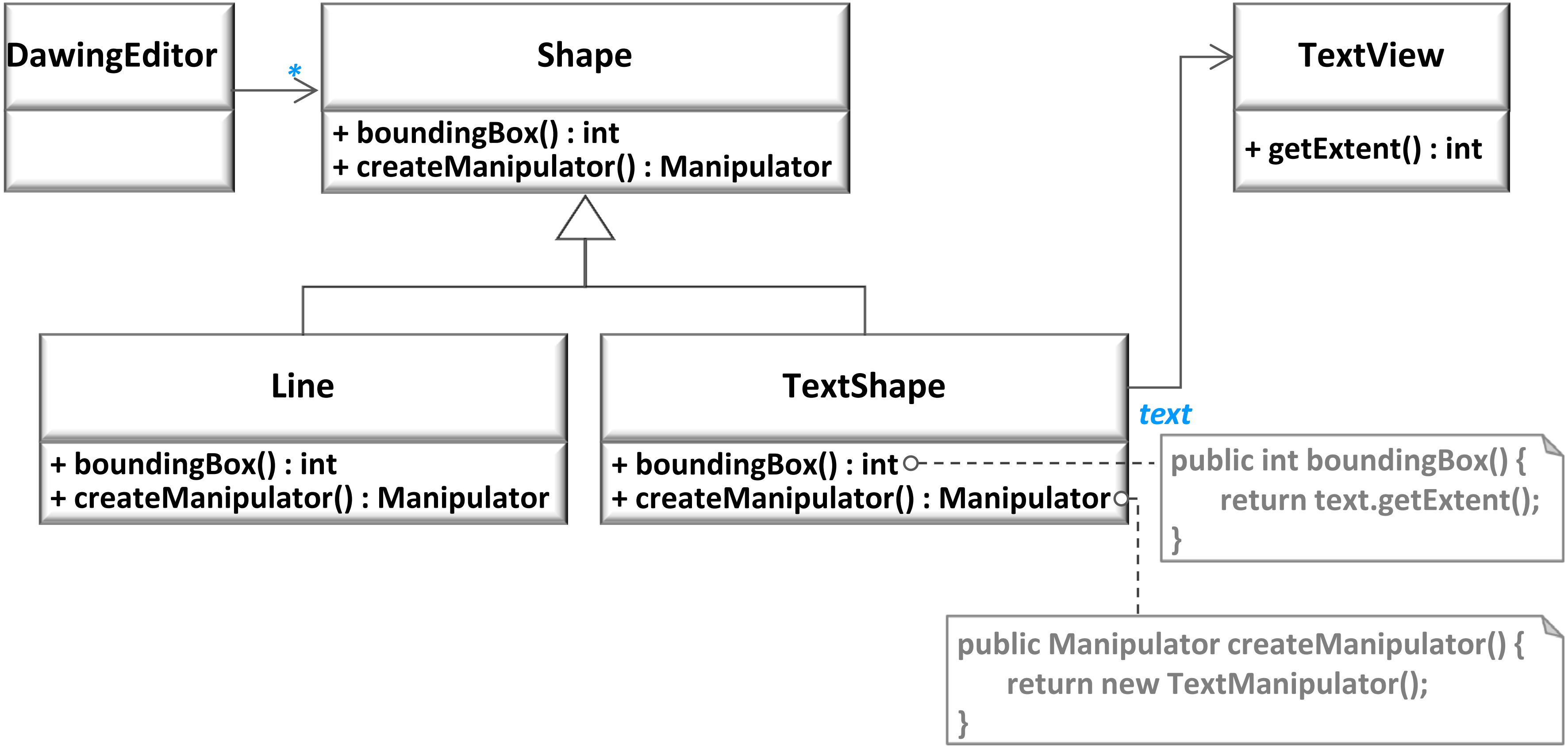
Turkey was the adaptee interface

Motivation (1)

- **Sometimes a toolkit or class library can not be used because its interface is incompatible with the interface required by an application.**
  - We can not change the library interface, since we may not have its source code.
  - Even if we did have the source code, we probably should not change the library for each domain-specific application.
- **Two Approaches**
  - Class Adapter
    - Inherit an adapter and an adaptee.
  - Object Adapter
    - Compose an adaptee instance within an dapter and implement the adapter in terms of the adaptee's interface.

Motivation (2)

- A solution using an object adapter:



```
public int boundingBox() {
     return text.getExtent();
}
```

```
public Manipulator createManipulator() {
     return new TextManipulator();
}
```

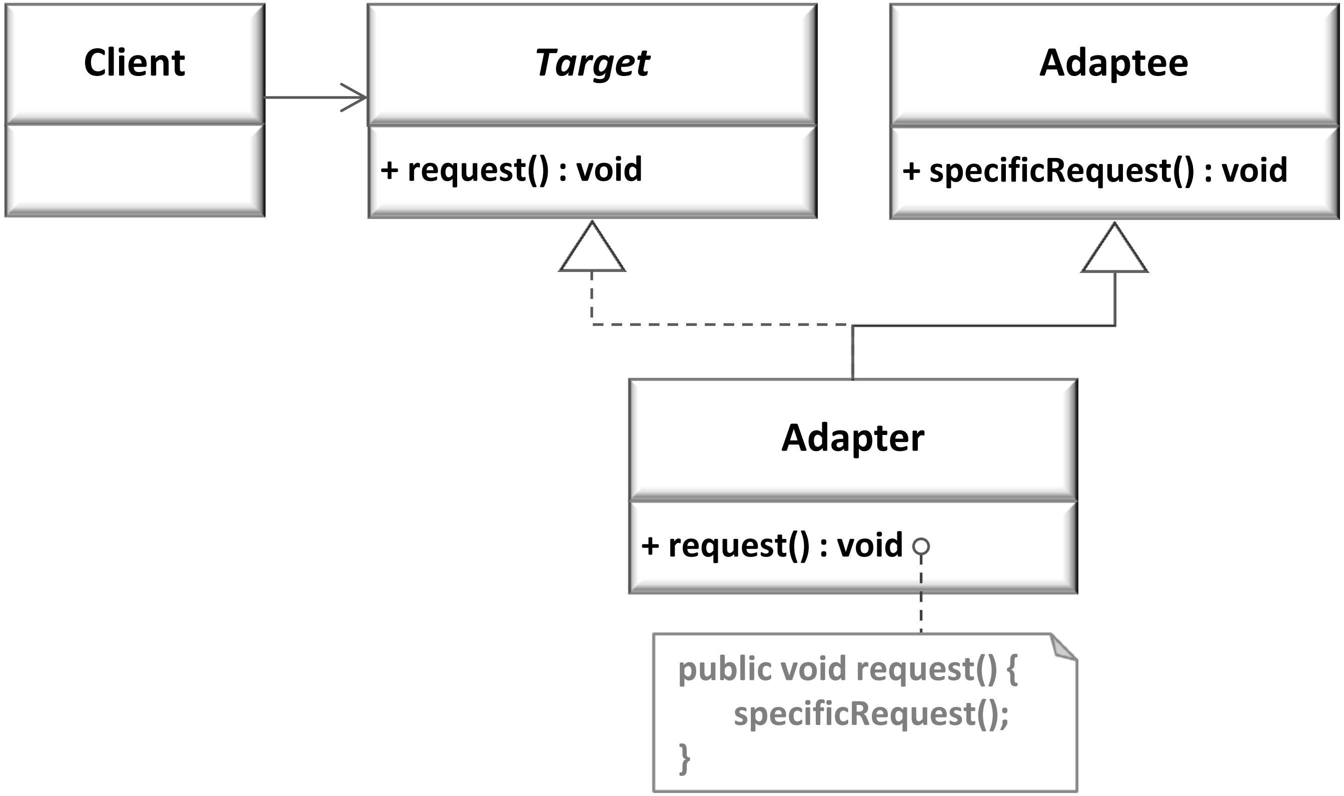
boundingBox() in TextShape are incompatible with getExtent() in TextView.

Applicability

- **Use the Adapter pattern when:**
  - To use an existing class, and its interface does not match the one you need
  - To create a reusable class that cooperates with unrelated classes with incompatible interfaces
  - To use several existing subclasses, but it's impractical to adapt interface by subclassing every one
    - Only for object adapters

Structure: Class Adapter

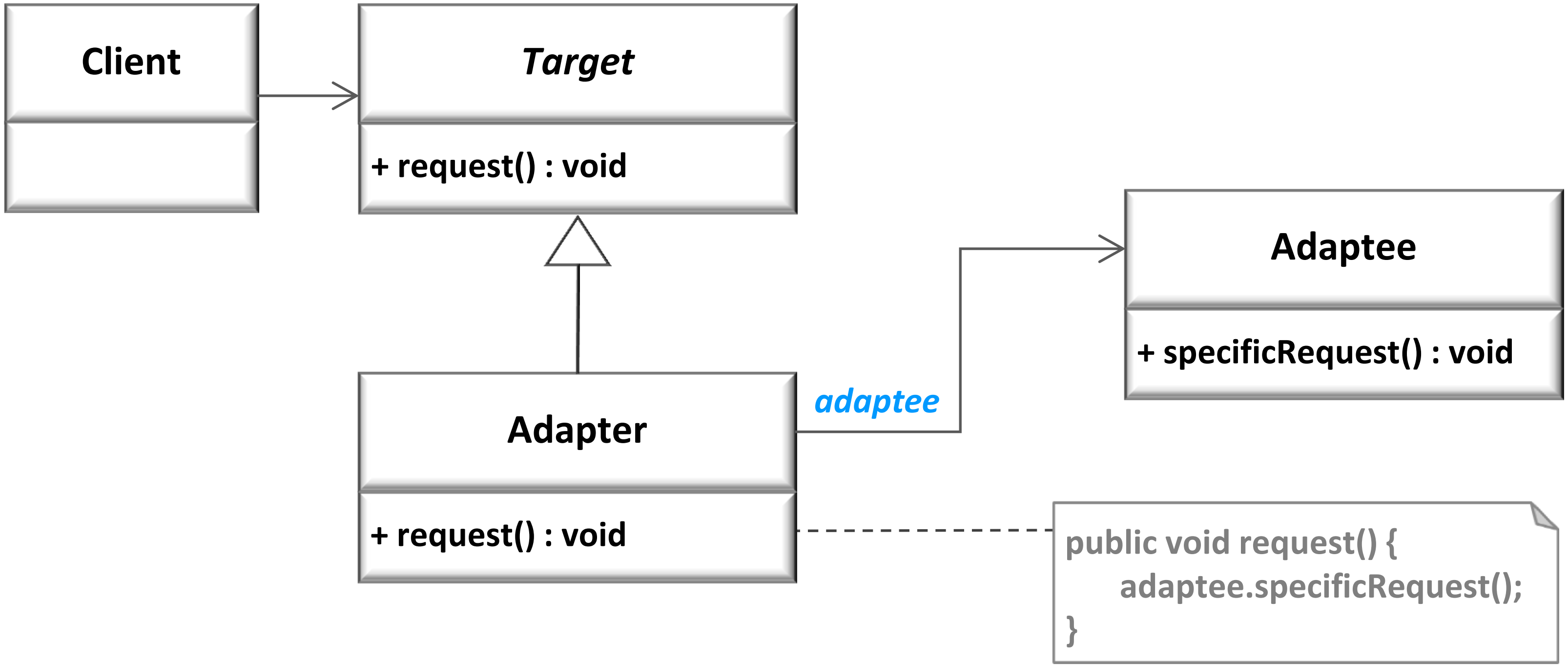
- A class adapter uses multiple inheritance to adapt one interface to another:



```
public void request() {
      specificRequest();
}
```

Structure: Object Adapter

- An object adapter relies on object composition:



```
public void request() {
    adaptee.specificRequest();
}
```

Participants

- **Target (Shape)**
  - To define the domain-specific interface that Client uses
- **Client (DrawingEditor)**
  - To collaborate with objects conforming to the Target interface
- **Adaptee (TextView)**
  - To define an existing interface that needs adapting
- **Adapter (TextShape)**
  - To adapt the interface of Adaptee to the Target interface

Collaborations

- **Clients call operations on an Adapter instance.**
- **The adapter calls Adaptee operations that carry out the request.**

Consequences: Class Adapter

- **Advantages**
  - Lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
  - Introduces only one object, and no additional pointer indirection is needed to get to the adaptee.
- **Disadvantages**
  - Will not work to adapt a class and all its subclasses.

Consequences: Object Adapter

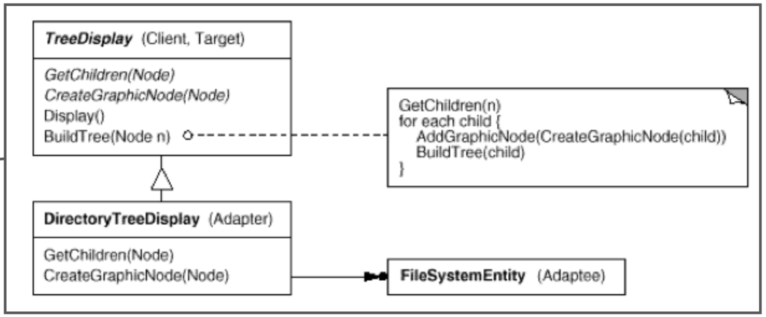
- **Advantages**
  - Lets a single Adapter work with many Adaptees.
    - Can also add functionality to all Adaptees at once.
- **Disadvantages**
  - Makes it harder to override Adaptee behavior.
    - Requires subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

Implementation (1)

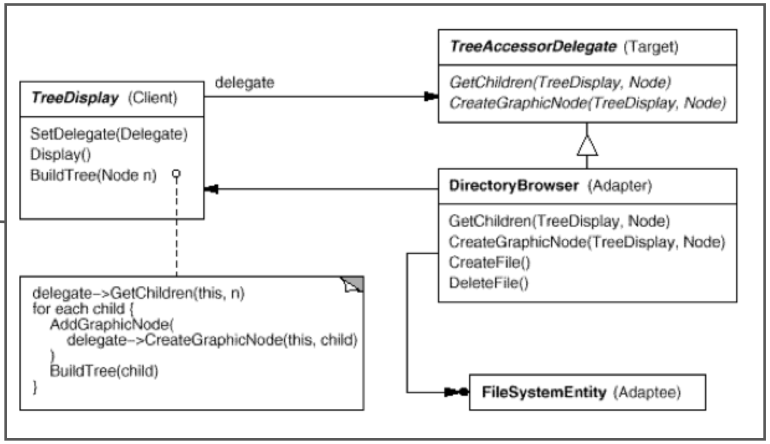
- **How much adapting does Adapter do?**
  - The amount of work Adapter does depends on how similar the Target interface is to Adaptee's.
- **Pluggable adapters**
  - A class is more reusable when the assumptions other classes must make to use it are minimized.
- **Using two-way adapter to provide transparency**
  - A two-way adapter supports both the Target and the Adaptee interface.
  - It allows an adapted object (Adapter) to appear as an Adaptee object or a Target object.
  - It is useful when two different clients need to view an object differently.

Implementation (2)

- **Implementing class adapters**
  - Adapter would inherit publicly from Target and privately from Adaptee.
- **Pluggable adapters**
  - Using abstract operations
  - Using delegate objects
  - Parameterized adapters



```
GetChildren(n)
for each child {
    AddGraphicNode(CreateGraphicNode(child))
    BuildTree(child)
}
```


TreeDisplay (Client)
SetDelegate(Delegate)
Display()
BuildTree(Node n)
delegate
TreeAccessorDelegate (Target)
GetChildren(TreeDisplay, Node)
CreateGraphicNode(TreeDisplay, Node)
DirectoryBrowser (Adapter)
GetChildren(TreeDisplay, Node)
CreateGraphicNode(TreeDisplay, Node)
CreateFile()
DeleteFile()
FileSystemEntity (Adaptee)

```
delegate->GetChildren(this, n)
for each child {
    AddGraphicNode(
        delegate->CreateGraphicNode(this, child)
    )
    BuildTree(child)
}
```

Sample Code (1)

```
class Shape {
public:
    Shape();
    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual Manipulator* CreateManipulator()
        const;
};

class TextView {
public:
    TextView();
    void GetOrigin(Coord& x, Coord& y) const;
    void GetExtent(Coord& width, Coord& height)
        const;
    virtual bool IsEmpty() const;
};
```

```
class TextShape : public Shape, private TextView {
public:
        TextShape();
        virtual void BoundingBox(
                Point& bottomLeft, Point& topRight
        ) const;
        virtual bool IsEmpty() const;
        virtual Manipulator* CreateManipulator()
        const;
};
```

Sample Code (2)

```
void TextShape::BoundingBox (Point&
    bottomLeft, Point& topRight ) const {
    Coord bottom, left, width, height;
    GetOrigin(bottom, left);
    GetExtent(width, height);
    bottomLeft = Point(bottom, left);
    topRight = Point
        (bottom + height, left + width);
}
```

```
bool TextShape::IsEmpty () const {
            return TextView::IsEmpty();
}
Manipulator* TextShape::CreateManipulator ()
const {
            return new TextManipulator(this);
}
```

Known Uses

- **InterViews 2.6**
  - Interactor abstract class for user interface elements such as scroll bars, buttons, and menus.
  - Object adapter called GraphicBlock, a subclass of Interactor that contains a Graphic instance.
- **ObjectWorks\Smalltalk**
  - Pluggable adapters
  - A subclass of ValueModel called PluggableAdaptor
  - TableAdaptor class
- **NeXT's AppKit**
  - NXBrowser
- **Meyer's "Marriage of Convenience"**

Related Patterns

- **Bridge has a structure similar to an object adapter, but has a different intent.**
- **Decorator enhances another object without changing its interface.**
  - A decorator is thus more transparent to the application than an adapter is.
- **Proxy defines a representative or surrogate for another object and does not change its interface.**

Bridge

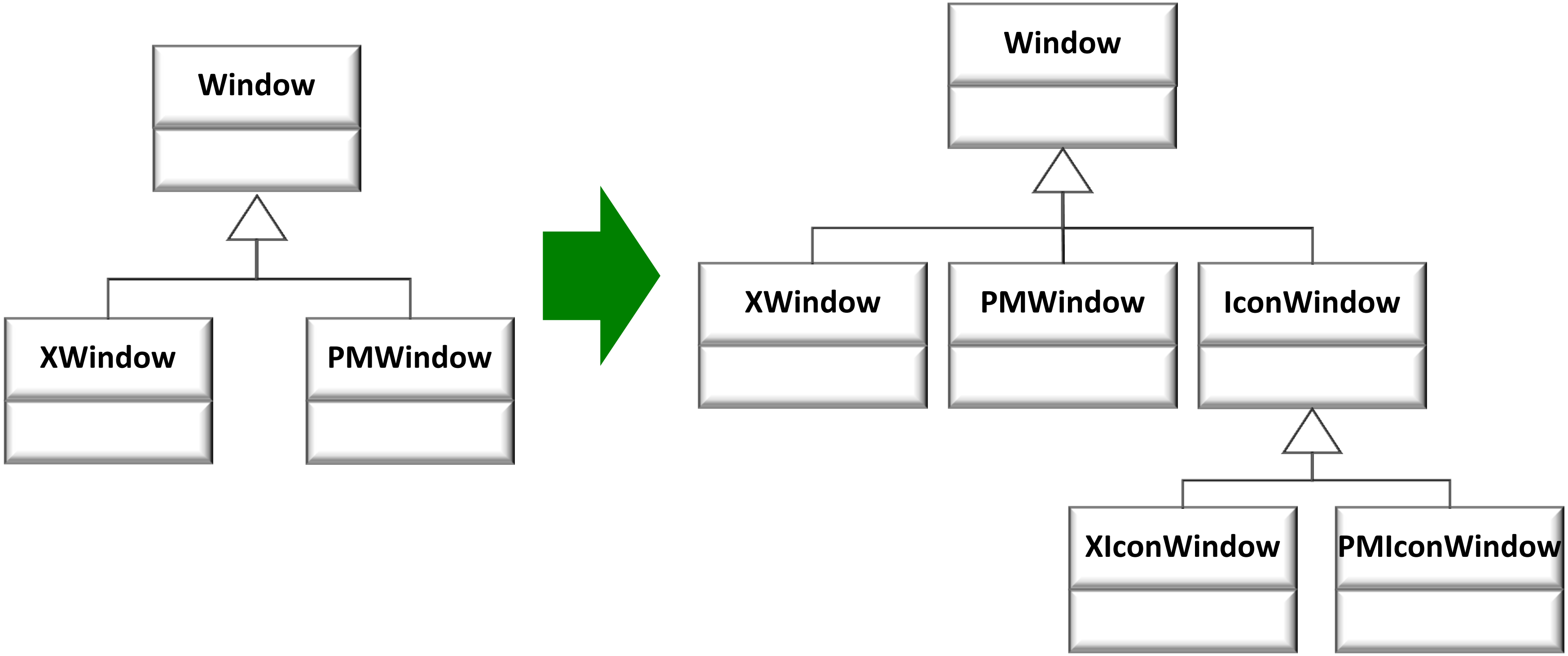
Unit 8

Intent

- **Decouple an abstraction from its implementation so that the two can vary independently.**

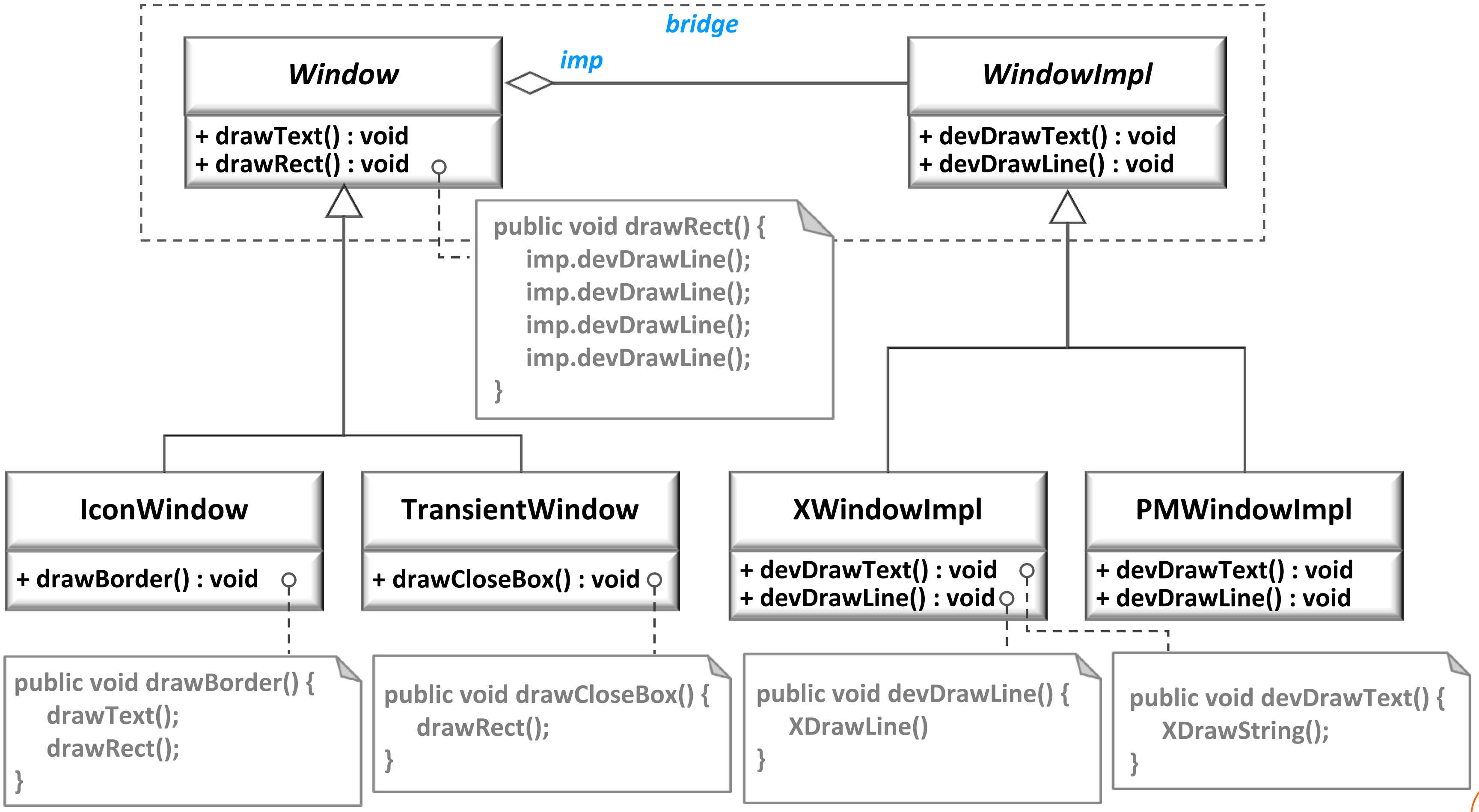
Motivation (1)

- **Consider the implementation of a portable Window abstraction in a user interface toolkit.**
  - It's inconvenient to extend the Window abstraction to cover different kinds of windows or new platforms.
  - It makes client code platform-dependent.



Motivation (2)

- **Bridge pattern addresses these problems by putting the Window abstraction and its implementation in separate class hierarchies.**



```
public void drawRect() {
    imp.devDrawLine();
    imp.devDrawLine();
    imp.devDrawLine();
    imp.devDrawLine();
}
```

```
public void drawBorder() {
    drawText();
    drawRect();
}
```

```
public void drawCloseBox() {
    drawRect();
}
```

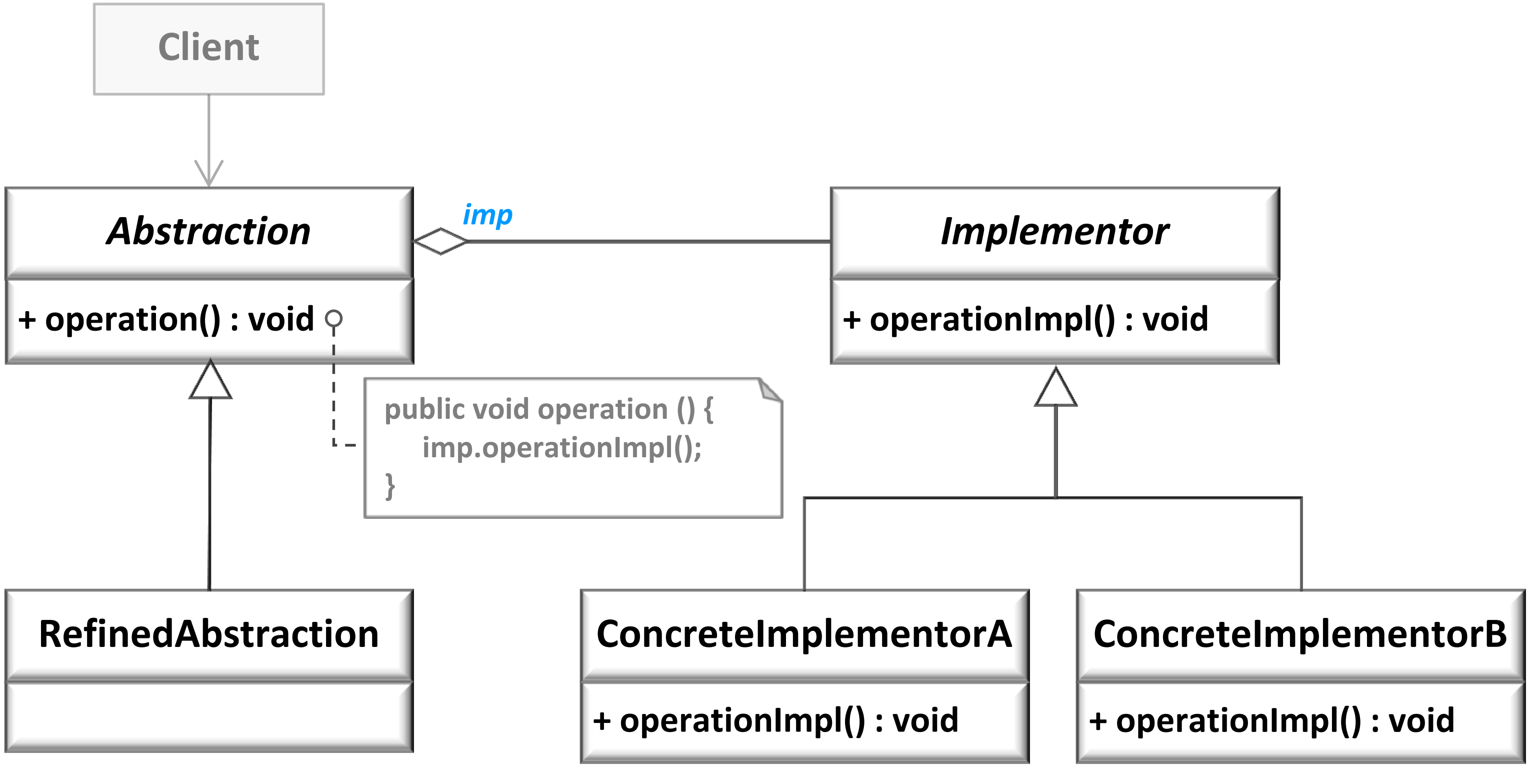
```
public void devDrawLine() {
    XDrawLine()
}
```

```
public void devDrawText() {
    XDrawString();
}
```

Applicability

- **Use the Bridge pattern when:**
  - You want to avoid a permanent binding between an abstraction and its implementation.
  - Both the abstractions and their implementations should be extensible by subclassing.
  - Changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.
  - You want to hide the implementation of an abstraction completely from clients.
  - You have a proliferation of classes as shown earlier in the first Motivation diagram.
  - You want to share an implementation among multiple objects (perhaps using reference counting), and this fact should be hidden from the client.

Structure



```
public void operation () {
    imp.operationImpl();
}
```

Participants

- **Abstraction (Window)**
  - To define the abstraction's interface
  - To maintain a reference to an object of type Implementor
- **RefinedAbstraction (IconWindow)**
  - To extend the interface defined by Abstraction
- **Implementor (WindowImp)**
  - To define the interface for implementation classes
    - This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different.
    - Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
- **ConcreteImplementor (XWindowImp, PMWindowImp)**
  - To implement the Implementor interface
  - To define its concrete implementation

Collaborations

- **Abstraction forwards client requests to its Implementor object.**

Consequences

- Decoupling interface and implementation
- Improved extensibility
- Hiding implementation details from clients

Implementation

- **Only one Implementor**
  - In situations where there's only one implementation, creating an abstract Implementor class isn't necessary.
- **Creating the right Implementor object**
  - If Abstraction knows about all ConcreteImplementor classes, then it can instantiate one of them in its constructor; it can decide between them based on parameters passed to its constructor.
- **Sharing implementors**
  - The Body stores a reference count that the Handle class increments and decrements.
- **Using multiple inheritance**
  - se multiple inheritance in C++ to combine an interface with its implementation.

Sample Code (1)

```
class Window {
public:
    Window(View* contents);
    // requests handled by window
    …
    // requests forwarded to implementation
    …
    virtual void DrawLine(const Point&, const
        Point&);
    virtual void DrawRect(const Point&, const
        Point&);
    virtual void DrawPolygon(const Point[], int n);
    virtual void DrawText(const char*, const
        Point&);
```

```
protected:
        WindowImp* GetWindowImp();
        View* GetView();
private:
        WindowImp* _imp;
        View* _contents; // the window's contents
};
```

Sample Code (2)

```
class WindowImp {
public:
    virtual void ImpTop() = 0;
    virtual void ImpBottom() = 0;
    virtual void ImpSetExtent(const Point&) = 0;
    virtual void ImpSetOrigin(const Point&) = 0;
    virtual void DeviceRect(Coord, Coord, Coord, Coord) = 0;
    virtual void DeviceText(const char*, Coord, Coord) = 0;
    virtual void DeviceBitmap(const char*, Coord, Coord) = 0;
    // lots more functions for drawing on windows...
protected:
    WindowImp();
};
```

Sample Code (3)

```
class ApplicationWindow : public Window {
public:
    // ...
    virtual void DrawContents();
};
void ApplicationWindow::DrawContents () {
    GetView()->DrawOn(this);
}
```

```
class IconWindow : public Window {
public:
        // ...
        virtual void DrawContents();
private:
    const char* _bitmapName;
};
void IconWindow::DrawContents() {
        WindowImp* imp = GetWindowImp();
        if (imp != 0) {
            imp->DeviceBitmap(_bitmapName, 0.0, 0.0);
        }
}
```

Sample Code (4)

```
class XWindowImp : public WindowImp {
public:
    XWindowImp();
    virtual void DeviceRect(Coord, Coord, Coord,
        Coord);
    // remainder of public interface...
private:
    // lots of X window system-specific state,
        including:
    Display* _dpy;
    Drawable _winid; // window id
    GC _gc; // window graphic context
};
```

```
class PMWindowImp : public WindowImp {
public:
        PMWindowImp();
        virtual void DeviceRect(Coord, Coord,
            Coord, Coord);
        // remainder of public interface...
private:
        // lots of PM window system-specific state,
            including:
        HPS _hps;
};
```

Known Uses

- **ET++, WindowImp**
  - WindowPort
- **Both Coplien and Stroustrup mention Handle classes and give some examples.**
- **libg++ defines classes that implement common data structures, such as Set, LinkedSet, HashSet, LinkedList, and HashTable.**
- **NeXT's AppKit uses the Bridge pattern in the implementation and display of graphical images.**

Related Patterns

- Abstract Factory can create and configure a particular Bridge.
- Adapter pattern is geared toward making unrelated classes work together.

Composite

Unit 9

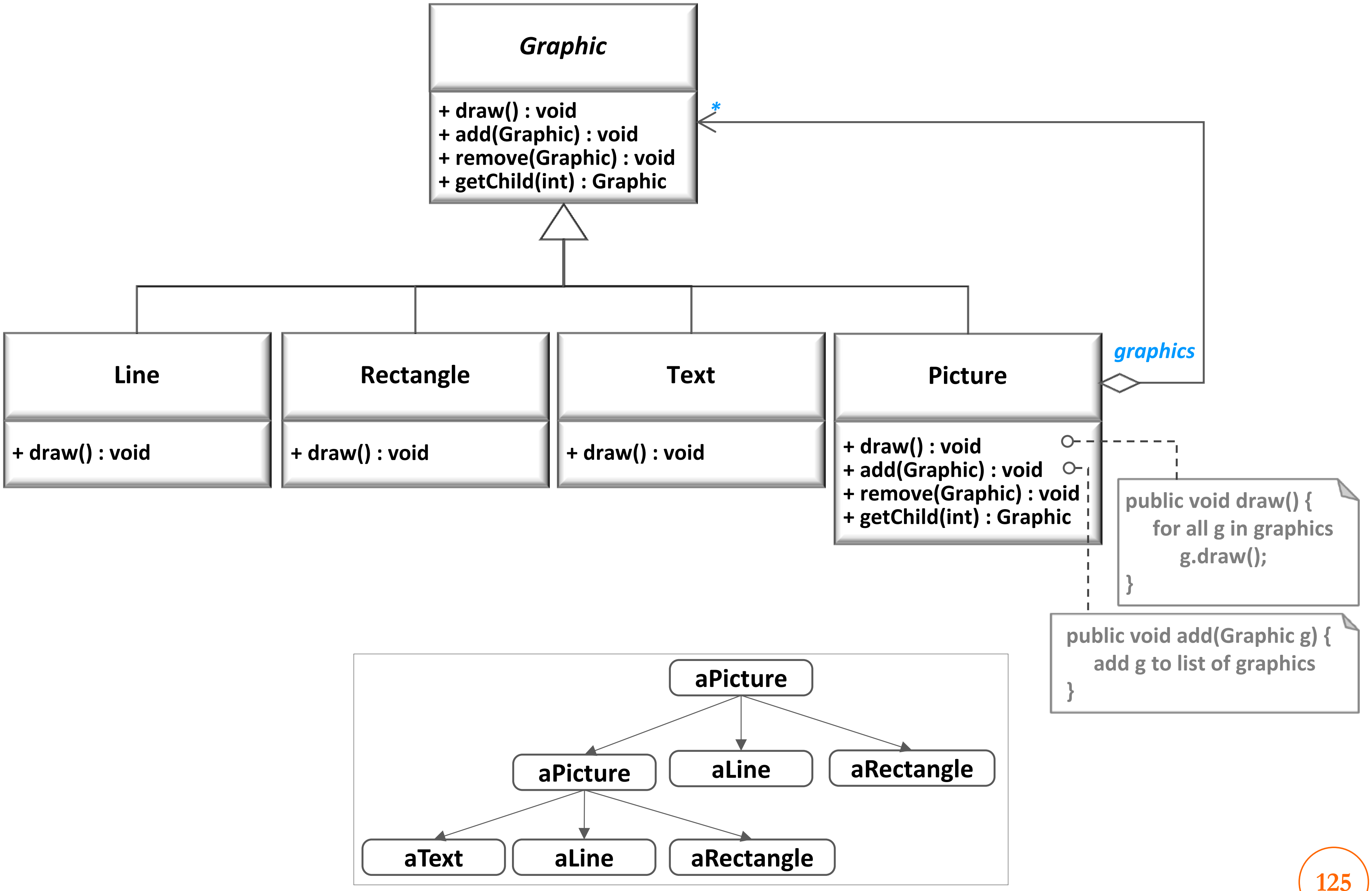
Intent

- **Compose objects into tree structures to represent part-whole hierarchies.**
  - Composite lets clients treat individual objects and compositions of objects uniformly.
  - This is called recursive composition.

Motivation (1)

- **Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components.**
  - The user can group components to form larger components, which in turn can be grouped to form still larger components.
- **Problem**
  - Code that uses these classes must treat primitive and container objects differently.
    - Even if most of the time the user treats them identically.
    - Having to distinguish these objects make the application more complex.

Motivation (2)



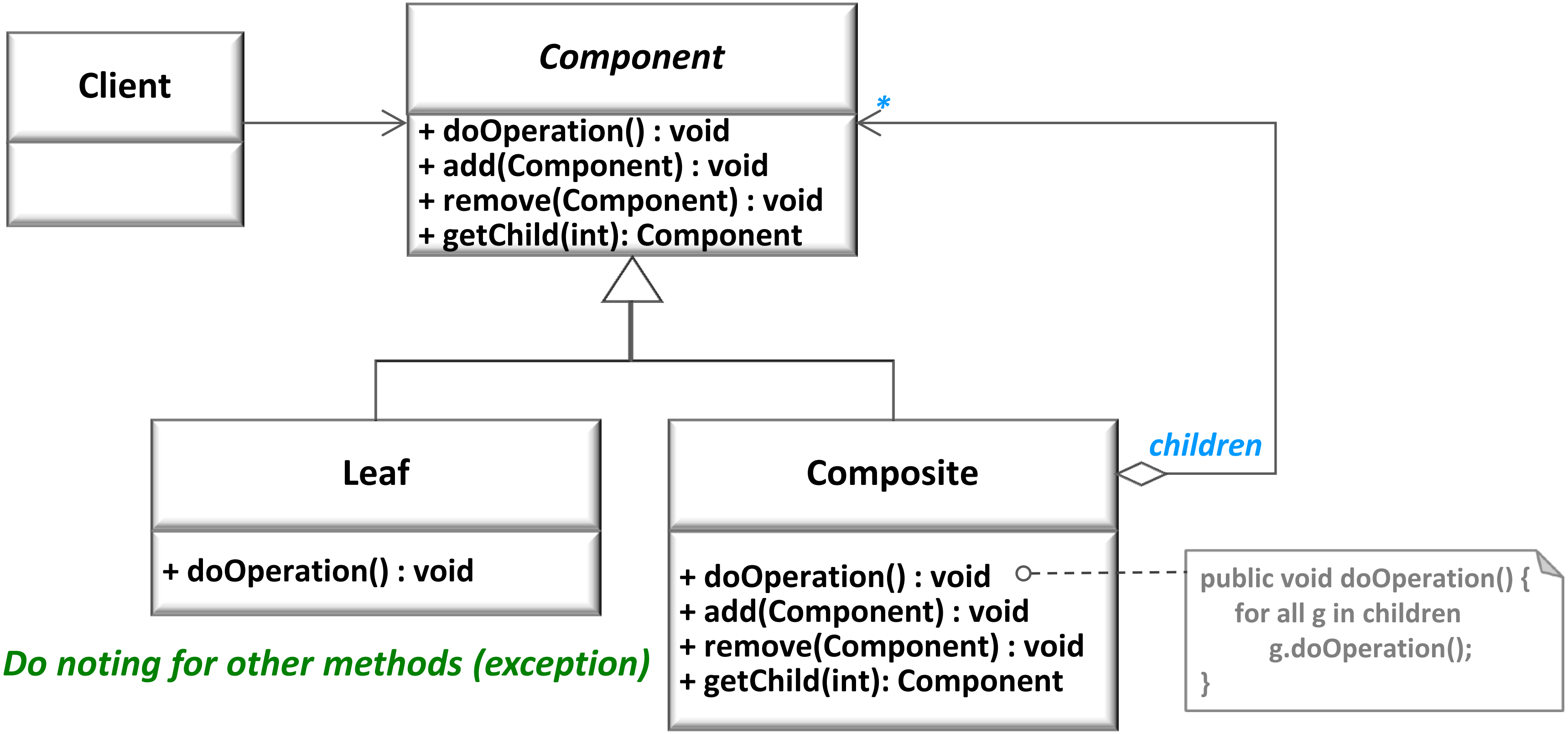
```
public void draw() {
    for all g in graphics
        g.draw();
}
```

```
public void add(Graphic g) {
    add g to list of graphics
}
```

Applicability

- **Use the Composite pattern when:**
  - You want to represent part-whole hierarchies of objects.
  - You want clients to be able to ignore the difference between compositions of objects and individual objects.
    - Clients will treat all objects in the composite structure uniformly.

Structure



Client

Component

+ doOperation() : void
+ add(Component) : void
+ remove(Component) : void
+ getChild(int): Component

Leaf

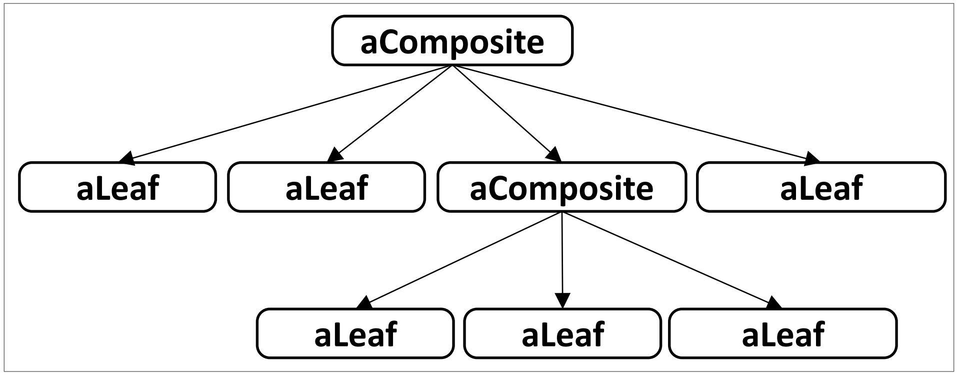
+ doOperation() : void

Composite

+ doOperation() : void
+ add(Component) : void
+ remove(Component) : void
+ getChild(int): Component

```
public void doOperation() {
    for all g in children
        g.doOperation();
}
```

Do noting for other methods (exception)



Participants (1)

- **Component (Graphic)**
  - To declare the interface for objects in the composition
  - To implement default behavior for the interface common to all classes, as appropriate
  - To declare an interface for accessing and managing its child components
  - (optional) To defines an interface for accessing a component's parent in the recursive structure, and implement it if that's appropriate
- **Leaf (Rectangle, Line, Text, etc.)**
  - To represent leaf objects in the composition
    - A leaf has no children.
  - To define behavior for primitive objects in the composition

Participants (2)

- **Composite (Picture)**
  - To define behavior for components having children
  - To store child components
  - To implement child-related operations in the Component interface
- **Client**
  - To manipulate objects in the composition through the Component interface

Collaborations

- Clients use the Component class interface to interact with objects in the composite structure.
- If the recipient is a Leaf, then the request is handled directly.
- If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

Consequences

- **Advantages**
  - To be able to define class hierarchies consisting of primitive objects and composite objects
  - To make clients simpler, since they do not have to know if they are dealing with a leaf or a composite component
  - To make it easy to add new kinds of components
- **Disadvantages**
  - To make it harder to restrict the type of components of a composite

Implementation (1)

- **A composite object knows its contained components, that is, its children. Should components maintain a reference to their parent component?**
  - Depends on application, but having these references supports the Chain of Responsibility pattern.
- **Where should the child management methods (add(), remove(), getChild()) be declared?**
  - In the Component class: Gives transparency, since all components can be treated the same.
    - But it's not safe, since clients can try to do meaningless things to leaf components at run-time.
  - In the Composite class: Gives safety, since any attempt to perform a child operation on a leaf component will be caught at compile-time.
    - But, we lose transparency, since now leaf and composite components have different interfaces.

Implementation (2)

- **Should Component maintain the list of components that will be used by a composite object? Should this list be an instance variable of Component rather than Composite?**
  - Better to keep this part of Composite and avoid wasting the space in every leaf object
- **Is child ordering important?**
  - Depends on application
- **Who should delete components?**
  - Not a problem in Java! The garbage collector will come to the rescue!
- **What's the best data structure to store components?**
  - Depends on application

Sample Code (1)

```
class Equipment {
  public:
    virtual ~Equipment();
    const char* Name() { return _name; }
    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
    virtual void Add(Equipment*);
  protected:
    Equipment(const char*);
  private:
    const char* _name;
  };
```

```
class FloppyDisk : public Equipment {
  public:
    FloppyDisk(const char*);
    virtual ~FloppyDisk();
    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
  };
```

Sample Code (2)

```
class CompositeEquipment : public Equipment {
  public:
    virtual ~CompositeEquipment();
    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
    virtual void Add(Equipment*);
  protected:
    CompositeEquipment(const char*);
  private:
    List _equipment;
  };
```

```
class Chassis : public CompositeEquipment {
  public:
    Chassis(const char*);
    virtual ~Chassis();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
  };
```

Known Uses

- **Examples can be found in almost all object-oriented systems.**
- **Every user interface toolkit or framework that followed the View class of Smalltalk Model/View/Controller**
  - ET++, InterViews, Graphics, and Glyphs.
- **RTL Smalltalk compiler framework**

Related Patterns

- **Often the component-parent link is used for a Chain of Responsibility.**
- **Decorator is often used with Composite.**
  - They will usually have a common parent class.
- **Flyweight lets you share components, but they can no longer refer to their parents..**
- **Iterator can be used to traverse composites.**
- **Visitor localizes operations and behavior that would otherwise be distributed across Composite and Leaf classes.**

Decorator

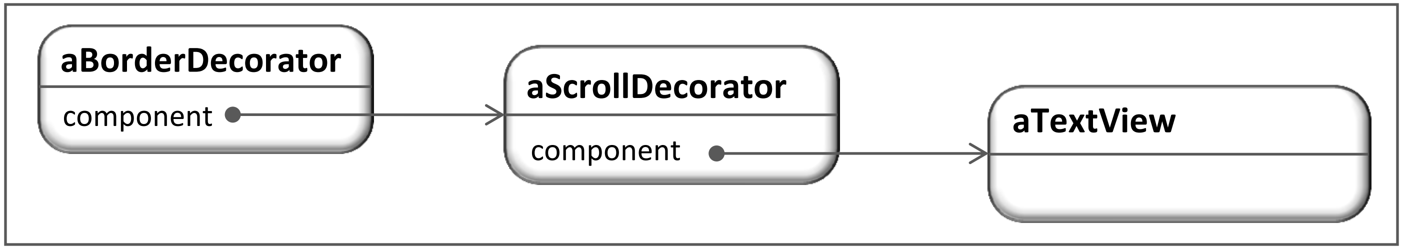
Unit 10

Intent

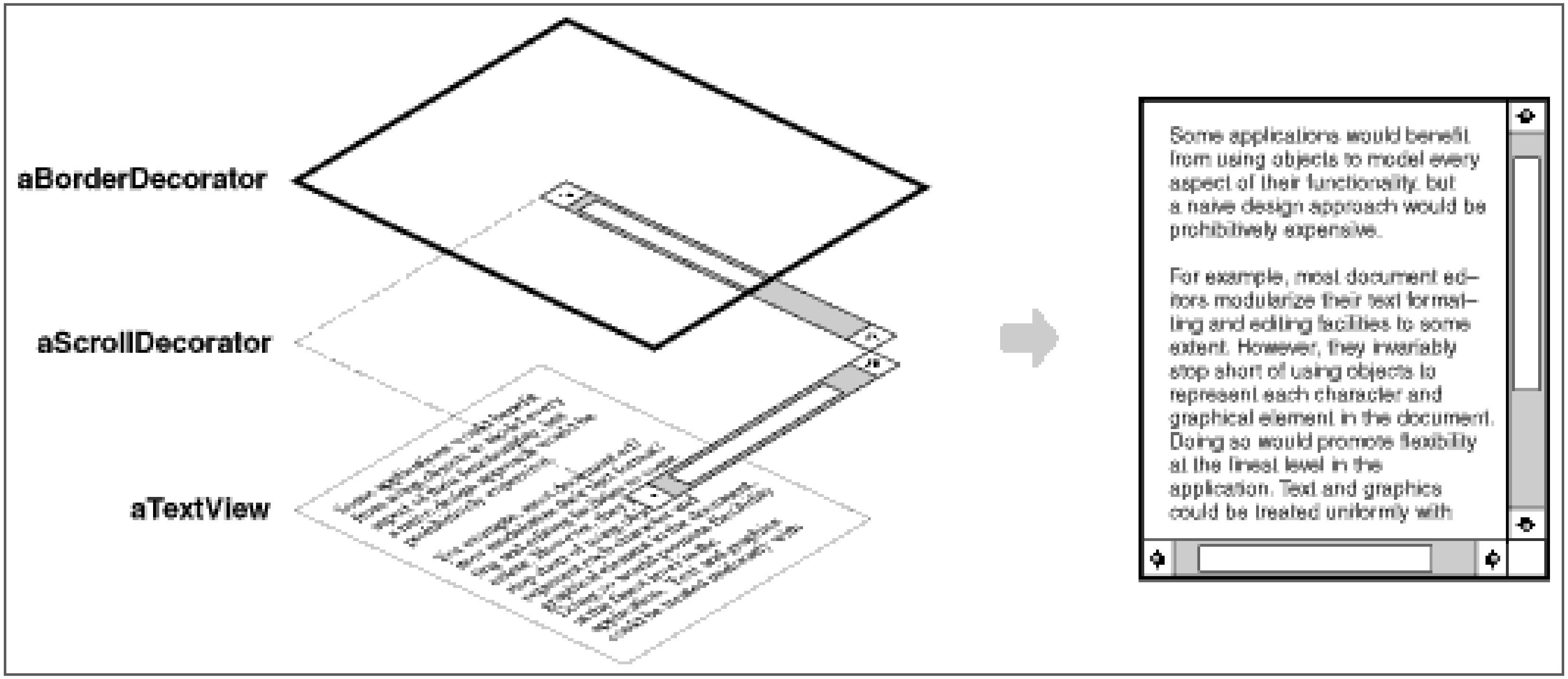
- Attach additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to subclassing for extending functionality.

Motivation (1)

- **GUI Components**
  - To add properties, such as borders or scrollbars to a GUI component.
  - We can do this with inheritance (subclassing), but this limits our flexibility.
  - A better way is to use composition!



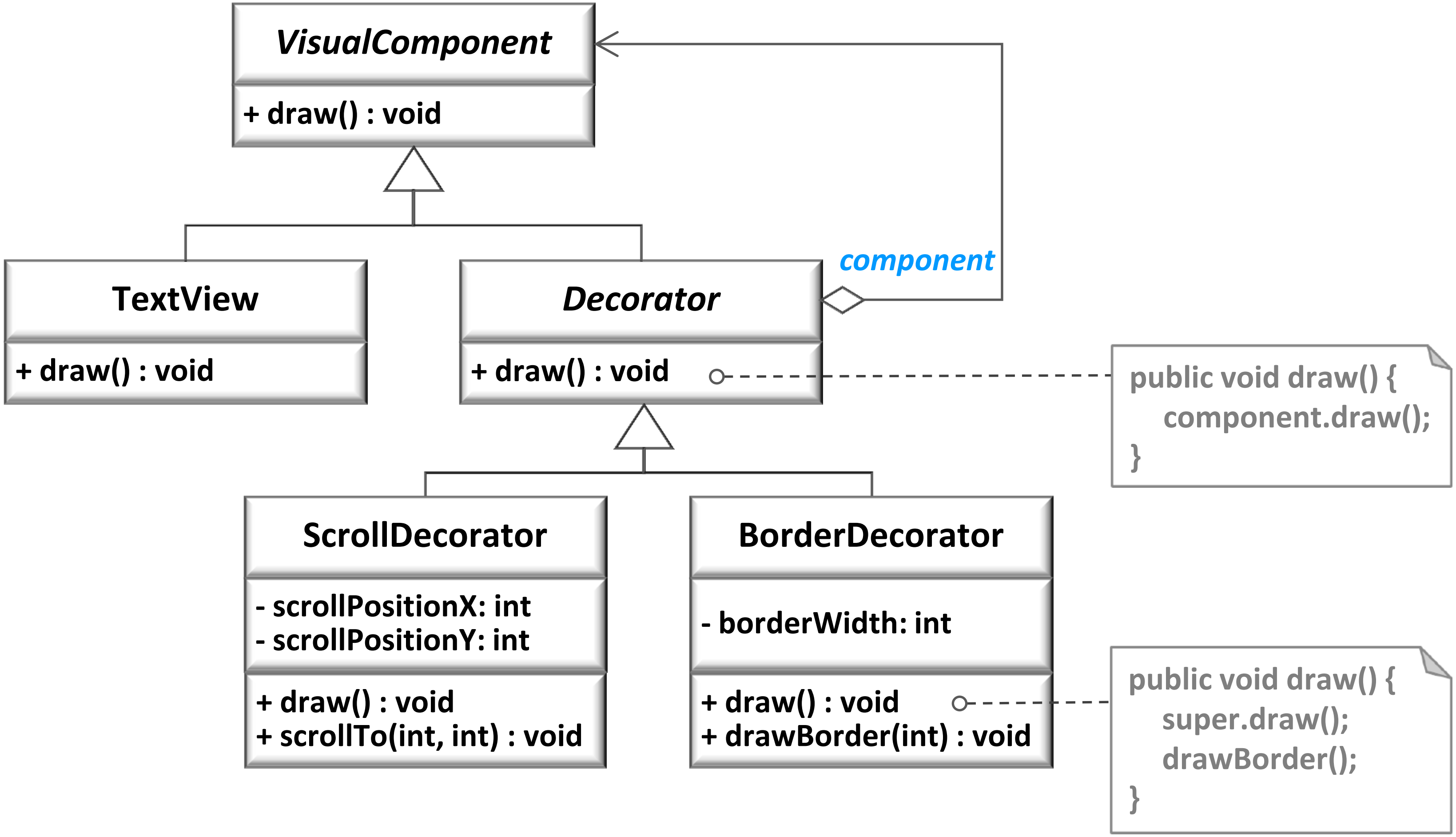
Motivation (2)



Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.

For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with

Motivation (3)



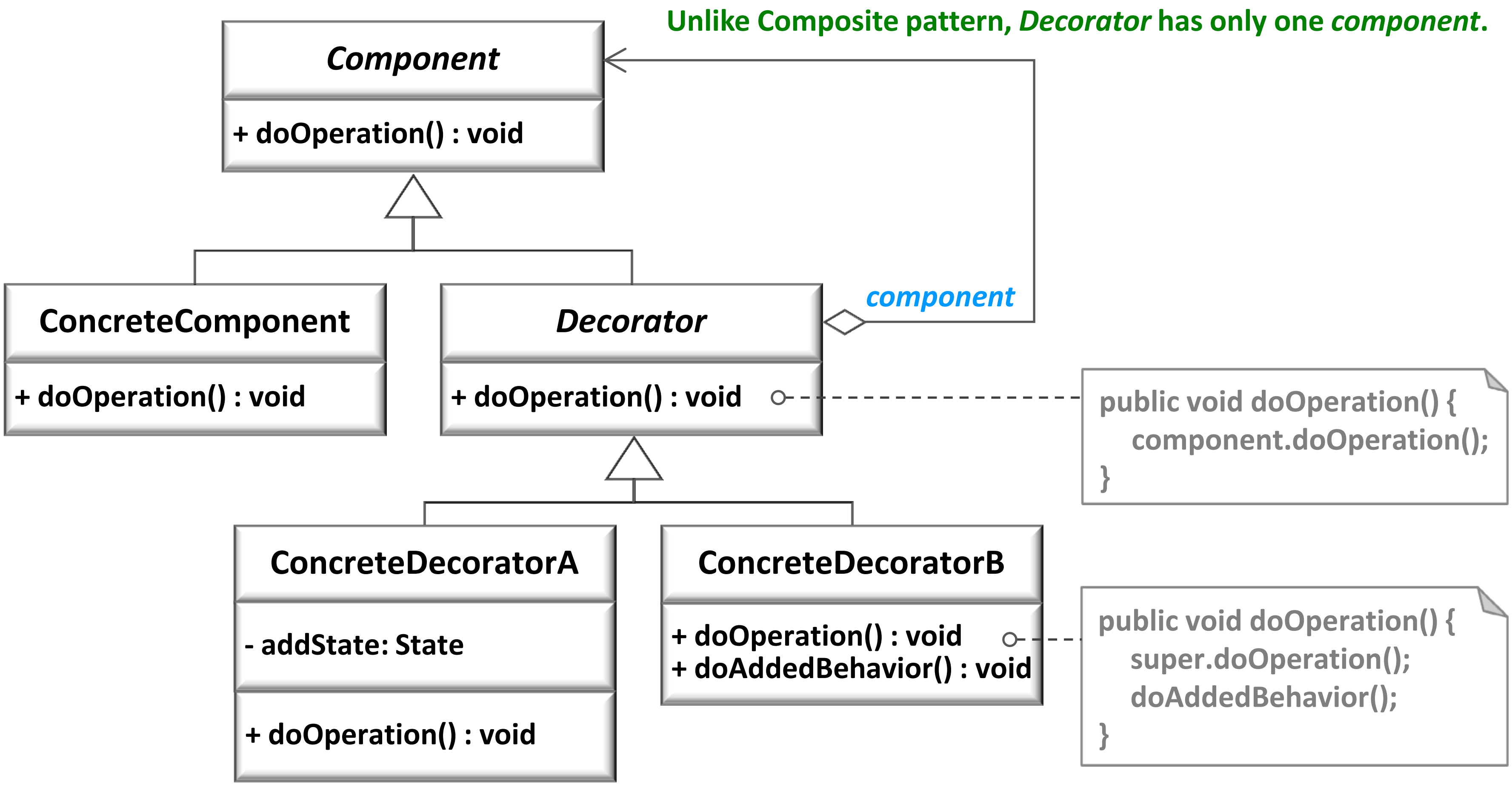
```
public void draw() {
    component.draw();
}
```

```
public void draw() {
    super.draw();
    drawBorder();
}
```

Applicability

- **Use Decorator:**
  - To add responsibilities to individual objects dynamically without affecting other objects.
  - When extension by subclassing is impractical.
    - Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination.
    - A class definition may be hidden or otherwise unavailable for subclassing.
- **Difference with Subsclassing?**
  - Subsclassing is more strict than Decorator pattern.
    - Impossible to flexibly modify the order of decorating objects

Structure



```
public void doOperation() {
    component.doOperation();
}
```

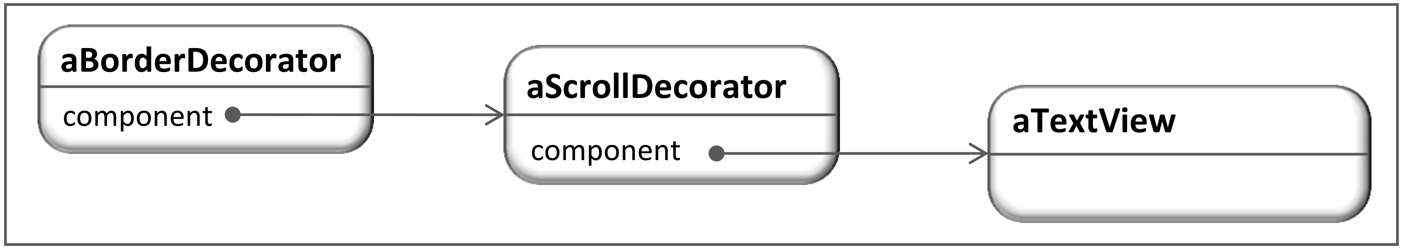
```
public void doOperation() {
    super.doOperation();
    doAddedBehavior();
}
```

Participants

- **Component (VisualComponent)**
  - To define the interface for objects that can have responsibilities added to them dynamically
- **ConcreteComponent (TextView)**
  - To define an object to which additional responsibilities can be attached
- **Decorator**
  - To maintain a reference to a Component object and defines an interface that conforms to Component's interface
- **ConcreteDecorator (BorderDecorator, ScrollDecorator)**
  - To add responsibilities to the component

Collaborations

- **Decorator forwards requests to its Component object.**
  - It may optionally perform additional operations before and after forwarding the request.
- **Example)**



  - 1. A client program invokes an operation of aBorderDecorator (the outmost decorator).
  - 2. aBorderDecorator invokes an operation of aScrollerDecorator via component.
  - 3. aScrollDecorator invokes an operation of aTextView via component.
  - 4. aTextView returns its result to aScrollDecorator, aScrollerDecorator returns its result to aBorderDecorator, and aBorderDecorator returns its result to the client program.

Consequences

- **Advantages**
  - More flexibility than static inheritance
  - To avoid feature-laden classes high up in the hierarchy
- **Disadvantages**
  - A decorator and its component aren't identical.
  - Lots of little objects

Implementation (1)

- **Interface conformance**
  - A decorator object's interface must conform to the interface of the component it decorates.
- **Omitting the abstract Decorator class**
  - There's no need to define an abstract Decorator class when you only need to add one responsibility.
    - That's often the case when you're dealing with an existing class hierarchy rather than designing a new one.
  - Merge Decorator's responsibility for forwarding requests to the component into the ConcreteDecorator.
- **Keeping Component classes lightweight**
  - To ensure a conforming interface, focus on defining an interface not on storing data.
  - The definition of the data representation should be deferred to subclasses.

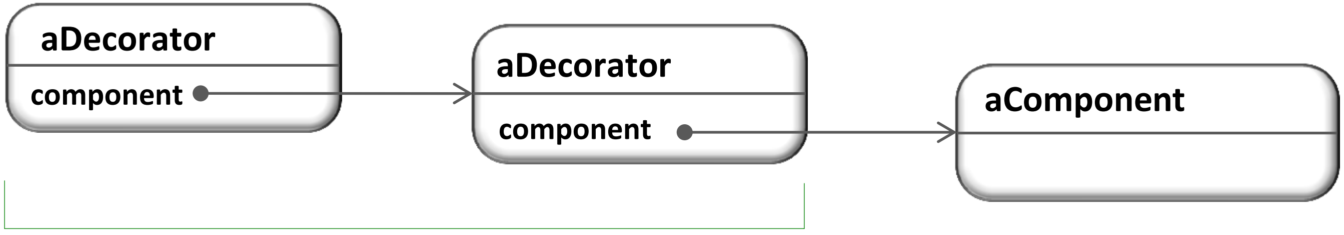
Implementation (2)

- **Changing the skin of an object versus changing its guts**
  - We can think of a decorator as a skin over an object that changes its behavior.
  - An alternative is to change the object's guts which is supported by Strategy Pattern.
    - Strategies are a better choice in situations where the Component class is intrinsically heavyweight, thereby making the Decorator pattern too costly to apply.
    - The Strategy pattern lets us alter or extend the component's functionality by replacing the strategy object.

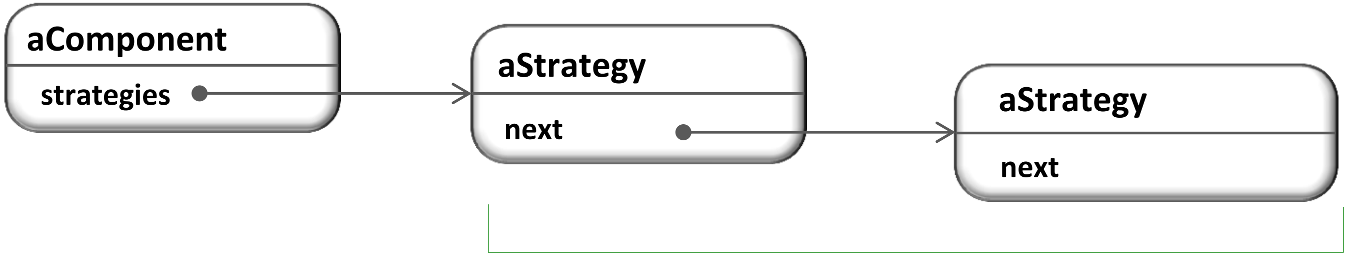
Implementation (3)

- Decorator vs. Strategy

	Decorator	Strategy
Modifying Component	No	Yes
Interface Conformance to Component	Yes (A decorator's interface must conform to the component's.)	No (A strategy can have its own specialized interface.)



Decorator-extended Functionality



Strategy-extended Functionality

Sample Code (1)

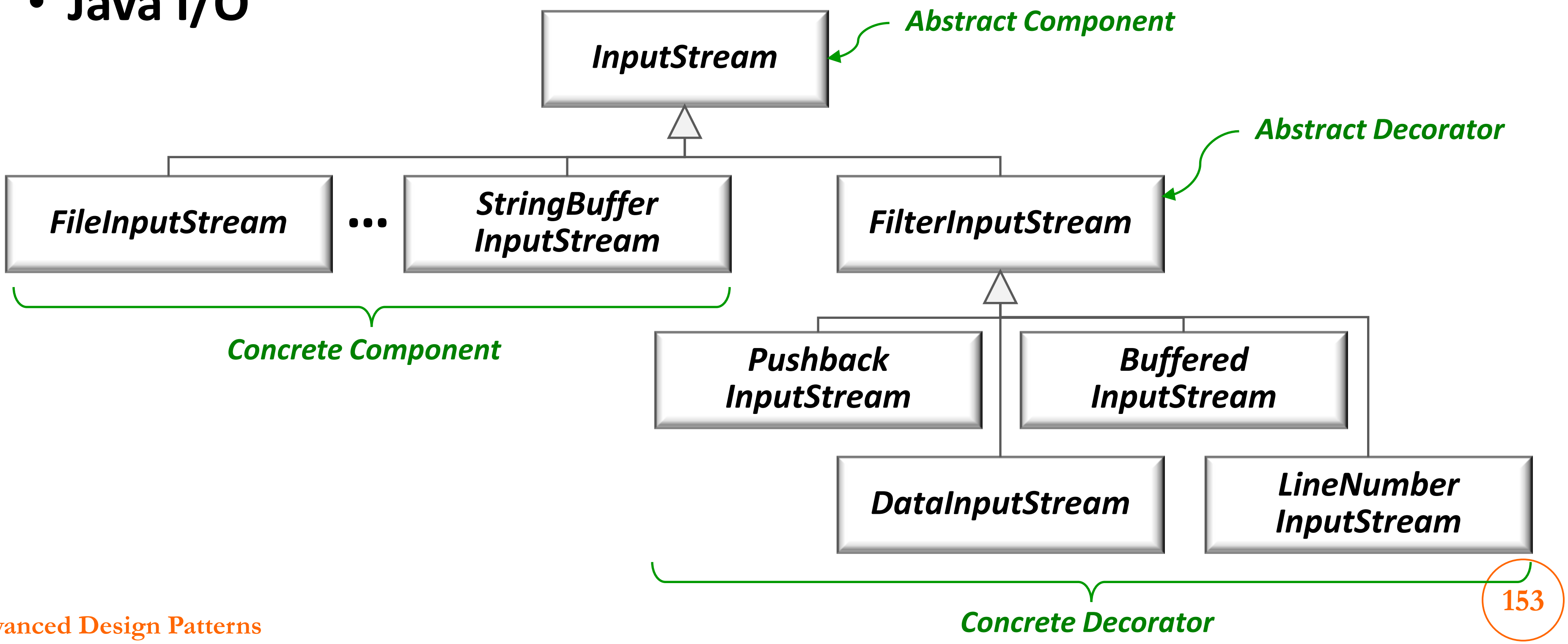
```
class VisualComponent {
  public:
    VisualComponent();
    virtual void Draw();
  };

  class Decorator : public VisualComponent {
  public:
    Decorator(VisualComponent*);
    virtual void Draw();
  private:
    VisualComponent* _component;
  };
```

Sample Code (2)

```
class BorderDecorator : public Decorator {
  public:
      BorderDecorator(VisualComponent*, int borderWidth);
      virtual void Draw();
  private:
      void DrawBorder(int);
  private:
      int _width;
  };
   void BorderDecorator::Draw () {
      Decorator::Draw();
      DrawBorder(_width);
  }
```

Known Uses

- **Many object-oriented user interface toolkits use decorators to add graphical embellishments to widgets.**
  - InterViews, ET++, ObjectWorks\Smalltalk class library
  - DebuggingGlyph from InterViews
  - PassivityWrapper from ParcPlace Smalltalk
- **Java I/O**



Related Patterns

- **Adapter**
  - A decorator is different from an adapter in that a decorator only changes an object's responsibilities, not its interface; an adapter will give an object a completely new interface.
- **Composite**
  - A decorator can be viewed as a degenerate composite with only one component.
  - A decorator adds additional responsibilities and isn't intended for object aggregation.
- **Strategy**
  - A decorator lets you change the skin of an object; a strategy lets you change the guts.
  - These are two alternative ways of changing an object.

Facade

Unit 11

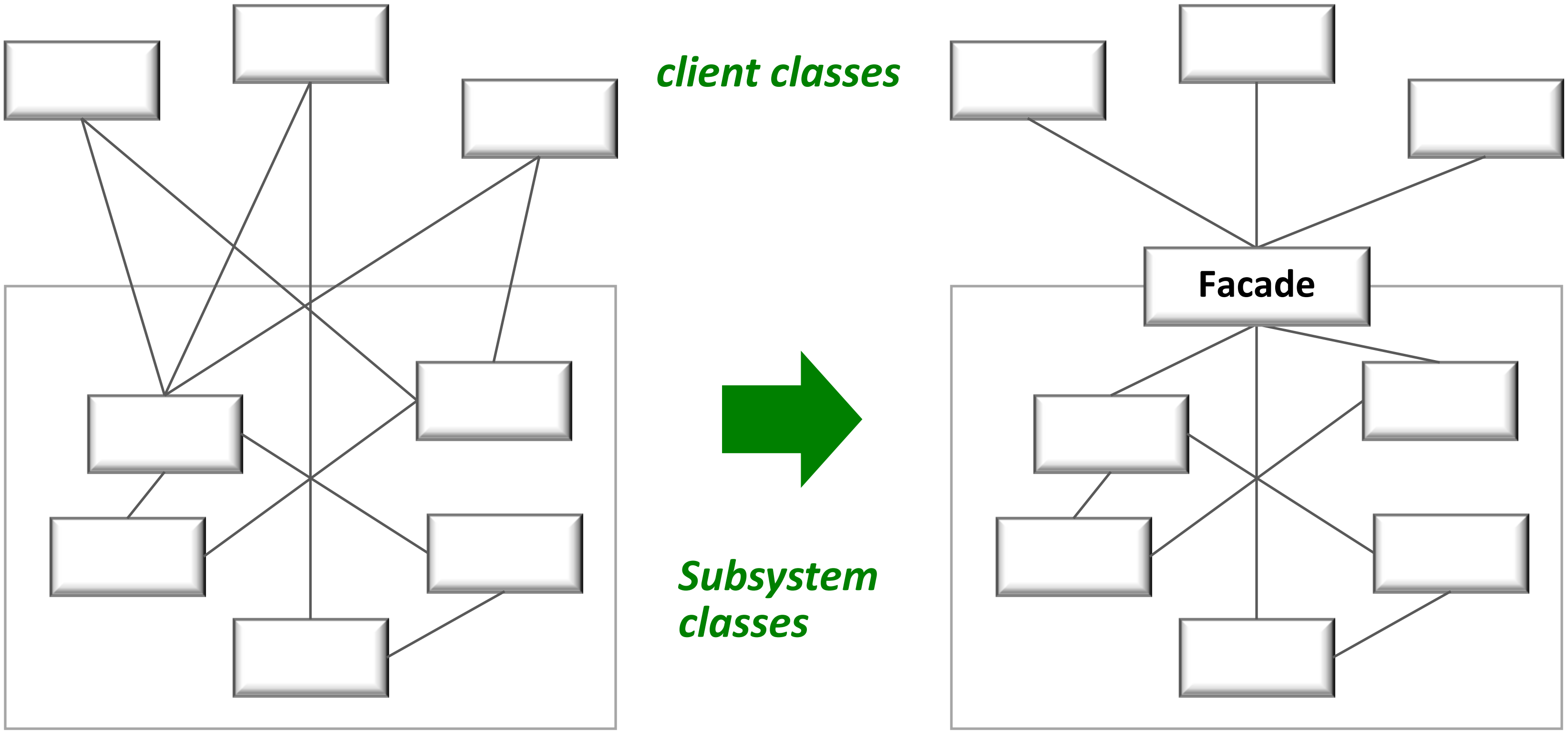
Intent

- To provide a unified interface for a set of interfaces in a subsystem
- To define a higher-level interface that makes the subsystem easier to use

Motivation (1)

- **Structuring a system into subsystems helps reduce complexity.**
  - To minimize the communication and dependencies between subsystems
- **One way to reduce complexity**
  - To introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem

Motivation (2)



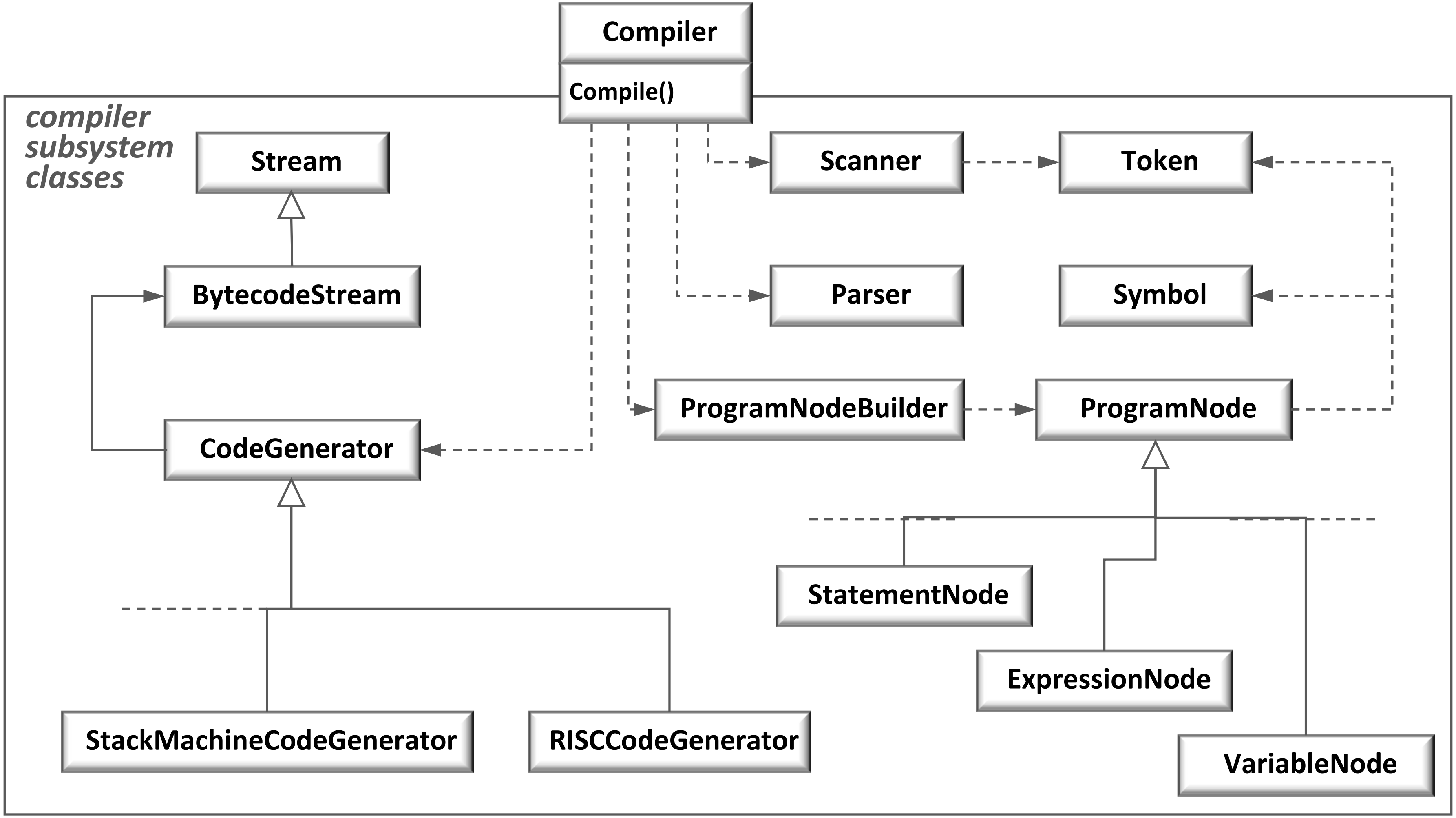
client classes

Subsystem classes

Facade

Motivation (3)

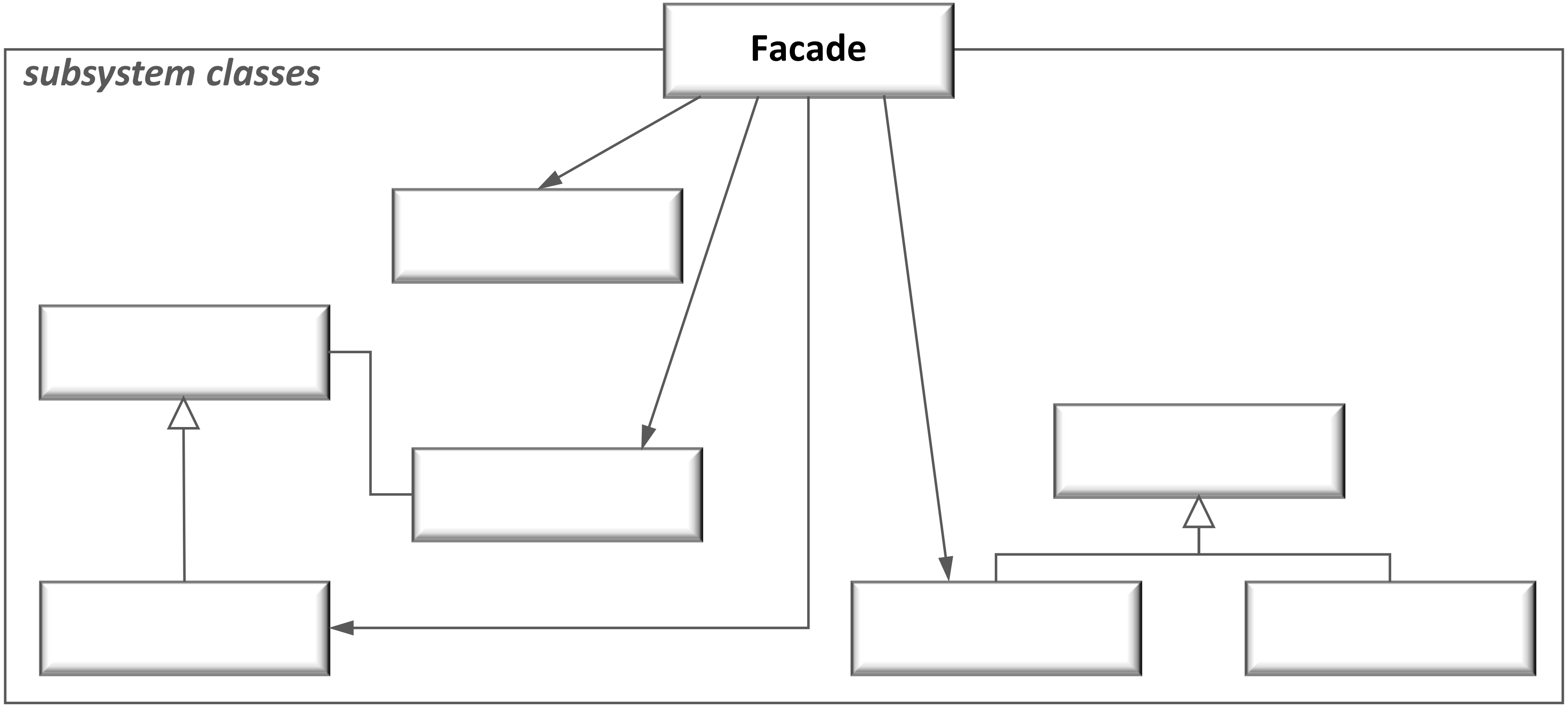
- **Example of Compiler Subsystem**



Applicability

- **Applicable Situations:**
  - To provide a simple interface for a complex subsystem
    - Providing a simple default view of the subsystem that is good enough for most clients
    - Only clients needing more customizability will need to look beyond the façade.
  - To decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability
  - To layer the subsystems
    - Using a façade to define an entry point to each subsystem level
    - If subsystems are dependent, then the dependencies between subsystems can be simplified by making them communicate with each other solely through their façades.

Structure



Participants

- **Facade (Compiler)**
  - To know which subsystem classes are responsible for a request
  - To delegate client requests to appropriate subsystem objects
- **Subsystem classes (Scanner, Parser, ProgramNode, etc.)**
  - To Implement subsystem functionality
  - To handle work assigned by the Façade object
  - To have no knowledge of the façade;
    - Subsystem classes keep no references to the façade

Collaborations

- **Clients communicate with the subsystem by sending requests to Façade, which forwards them to the appropriate subsystem object(s).**
  - Although the subsystem objects perform the actual work, the facade may have to do work of its own to translate its interface to subsystem interfaces.
- **Clients that use the facade don't have to access its subsystem objects directly.**

Consequences

- **Advantages**
  - To hide implementations of a subsystem from its clients
    - Reducing the number of objects that clients deal with
    - Making the subsystem easier to use
  - To promote weak coupling between the subsystem and its clients
    - To allow changing the subsystem classes without affecting its clients.
  - To layer a system and the dependencies between objects
    - Eliminating complex or circular dependencies
  - To reduce compilation dependencies in large software systems
  - To simplify porting systems to other platforms
  - Not to prevent sophisticated clients from accessing subsystem classes

Implementation

- **Reducing client-subsystem coupling**
  - Making Façade an abstract class with concrete subclasses for different implementations of a subsystem, then clients can communicate with the subsystem through the interface of the abstract Facade class.
  - To configure a Façade object with different subsystem objects
- **Public versus private subsystem classes**
  - The public interface to a subsystem consists of classes that all clients can access. The private interface is just for subsystem extenders.
  - The Façade class is part of the public interface, but it's not the only part.
    - Other subsystem classes are usually public as well.
  - Making subsystem classes private would be useful, but few object-oriented languages support it.

Sample Code – Compiler Subsystem (1)

```
class Scanner {
    public:
        Scanner(istream&);
        virtual ~Scanner();
        virtual Token& Scan();
    private:
        istream& _inputStream;
};

class Parser {
    public:
        Parser();
        virtual ~Parser();
        virtual void Parse(Scanner&,
        ProgramNodeBuilder&);
};
```

```
class ProgramNodeBuilder {
    public:
        ProgramNodeBuilder();
        virtual ProgramNode* NewVariable(
            const char* variableName
        ) const;
        virtual ProgramNode* New Assignment(
            ProgramNode* variable,
            ProgramNode* expression
        ) const;
        virtual ProgramNode* NewReturnStatement(
            ProgramNode* Value
        ) const;
        virtual ProgramNode* NewCondition(
            ProgramNode* condition,
            ProgramNode* truePart,
            ProgramNode* falsePart
        ) const;
        // …

        ProgramNode* GetRootNode();

    private:
        ProgramNode* _node;
};
```

Sample Code – Compiler Subsystem (2)

```
class ProgramNode {
   public:
      // program node manipulation
      virtual void GetSourcePosition(
         int& line, int& index);
      // …

      // child manipulcation
      virtual void Add(ProgramNode*);
      virtual void Remove(ProgramNode*);
      // …

      virtual void Traverse(CodeGenerator&);

   protected:
      ProgramNode();
};
```

```
class CodeGenerator {
   public:
      virtual void Visit(StatementNode*);
      virtual void Visit(ExpressionNode*);
      // …

   protected:
      CodeGenerator(BytecodeStream&);

   protected:
      BytecodeStream& _output;
};
```

Sample Code – Compiler Subsystem (3)

```
class Compiler {
    public:
        Compiler();
        virtual void Compile(istream&, BytecodeStream&);
};

Void Compiler::Compile(
    istream& input, BytecodeStream& output
)   {
        Scanner scanner(input);
        ProgramNodeBuilder builder;
        Parser parser;

        parser.Parse(scanner, builder);

        RISCCodeGenerator generator(output);
        ProgramNode* parseTree = builder.GetRootNode();
        parseTree -> Traverse(generator);
    }
```

Known Uses

- **ET++ application framework**
  - Built-in browsing tools for inspecting its objects at run-time
    - These browsing tools are implemented in a separate subsystem that includes a Façade class called "ProgrammingEnvironment."
    - To abstract coupling between the application and the browsing subsystem
- **Choices operating system**
  - To compose many frameworks into one
  - Three key abstractions
    - Processes, storage, and address spaces
  - To support porting *Choices* to a variety of different hardware platforms

Related Patterns

- **Abstract Factory**
  - Abstract Factory can be used with Façade to provide an interface for creating subsystem objects in a subsystem-independent way.
- **Mediator**
  - Mediator is similar to Façade in that it abstracts functionality of existing classes.
  - Mediator abstracts arbitrary communication between colleague objects.
- **Singleton**
  - Usually only one Façade object is required. Thus Facade objects are often Singletons.

Flyweight

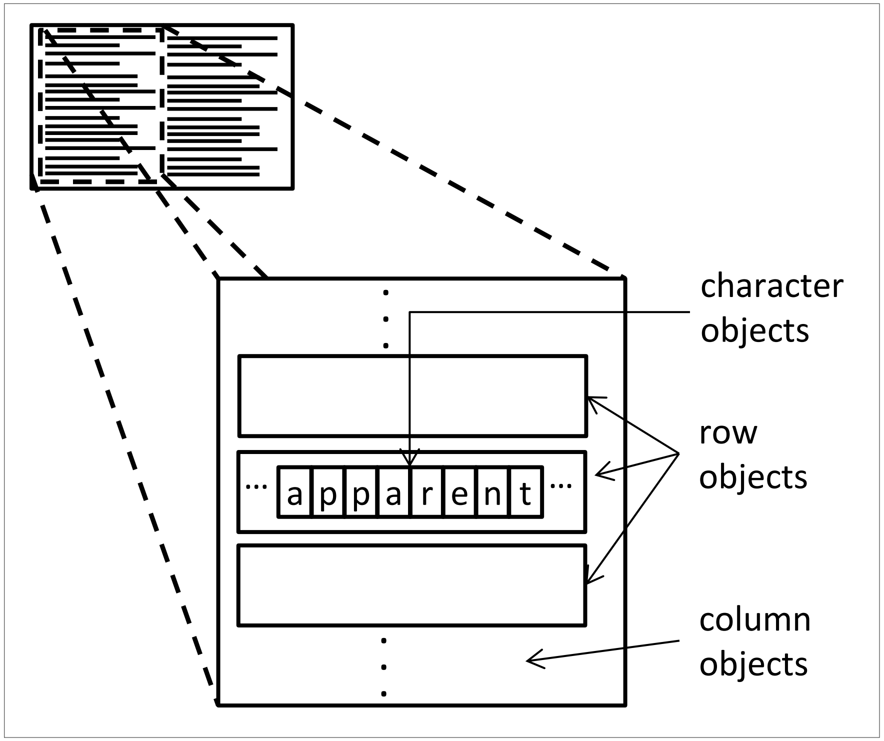
Unit 12

Intent

- **Use sharing to support large numbers of fine-grained objects efficiently.**

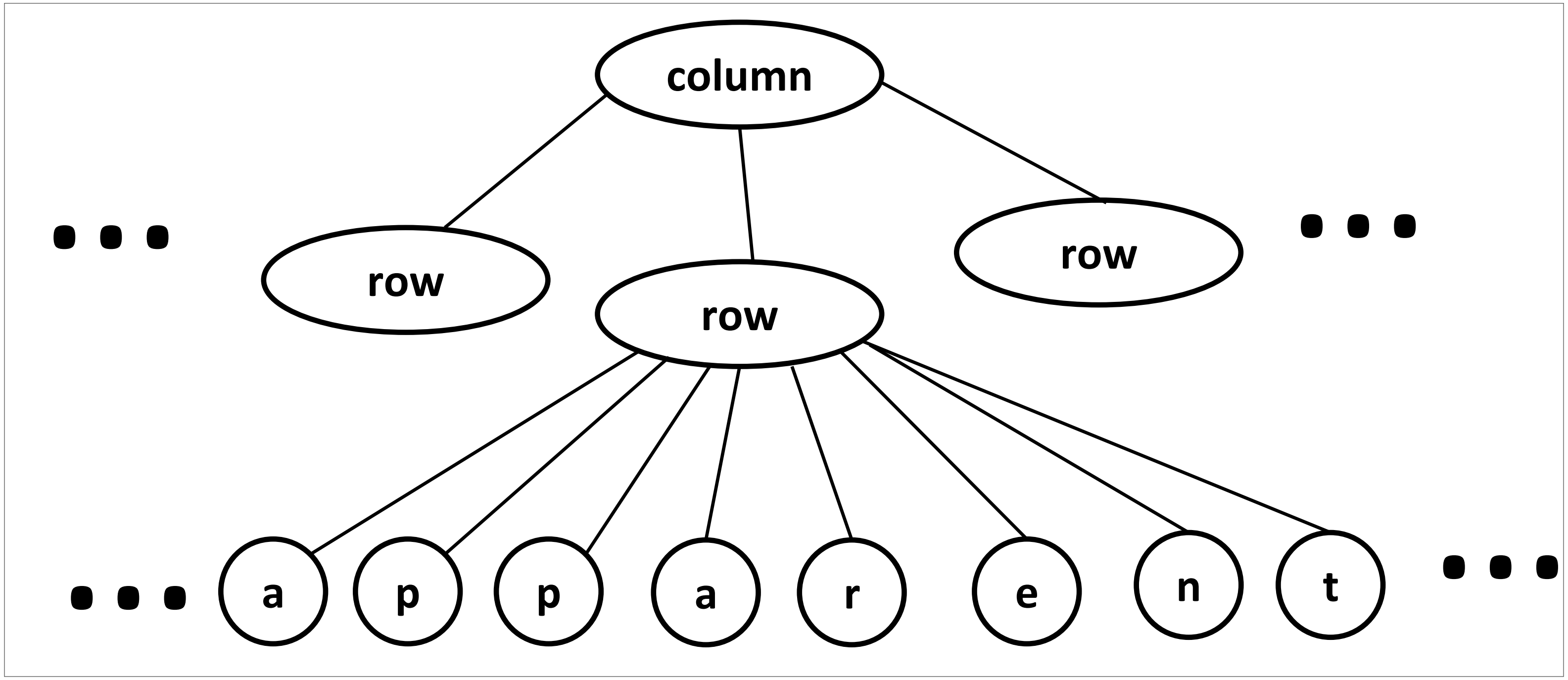
Motivation (1)

- **Applications could benefit from using objects throughout their design, but a naive implementation would be prohibitively expensive.**
  - Most document editor implementations have text formatting and editing facilities that are modularized to some extent.

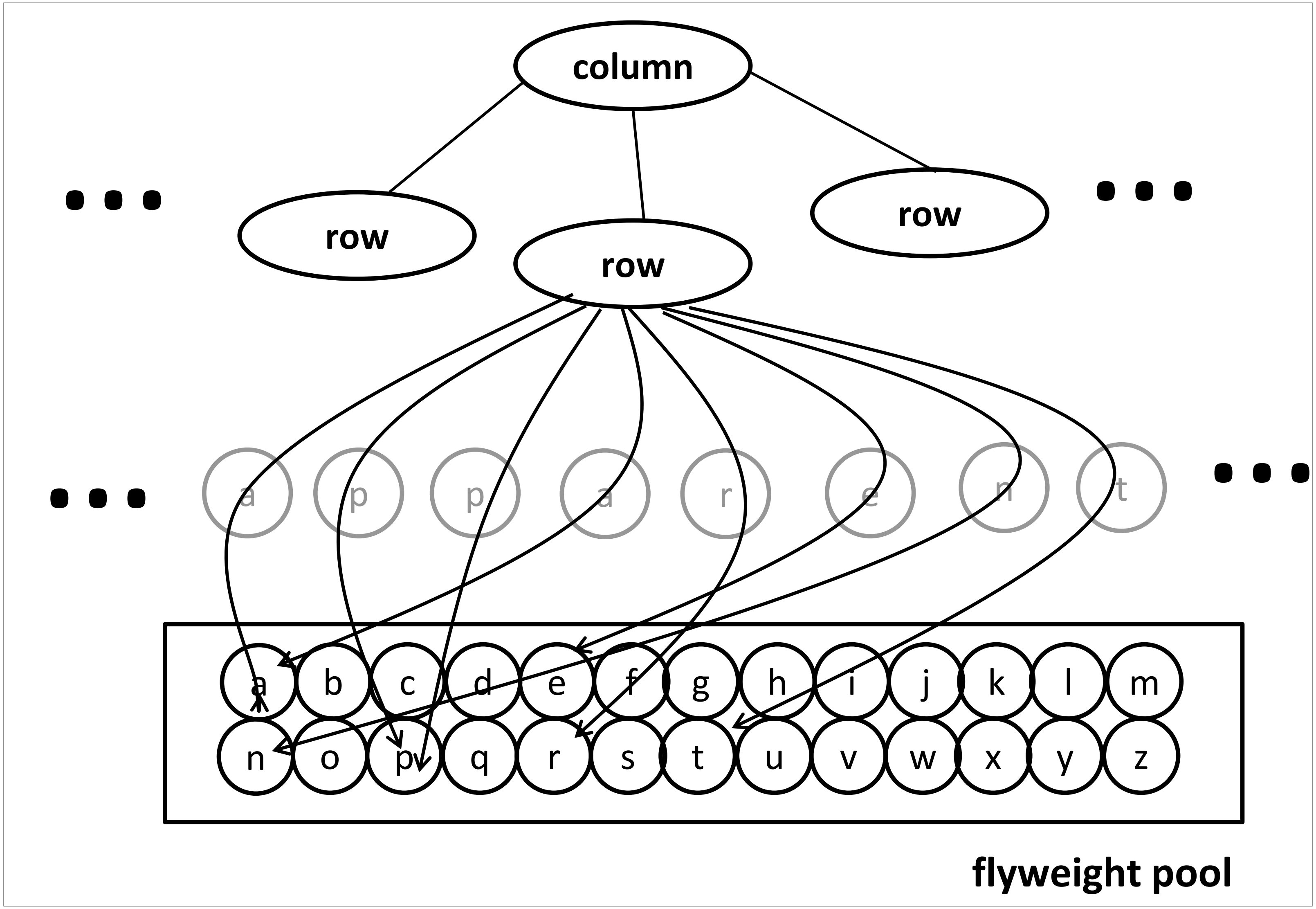


Motivation (2)

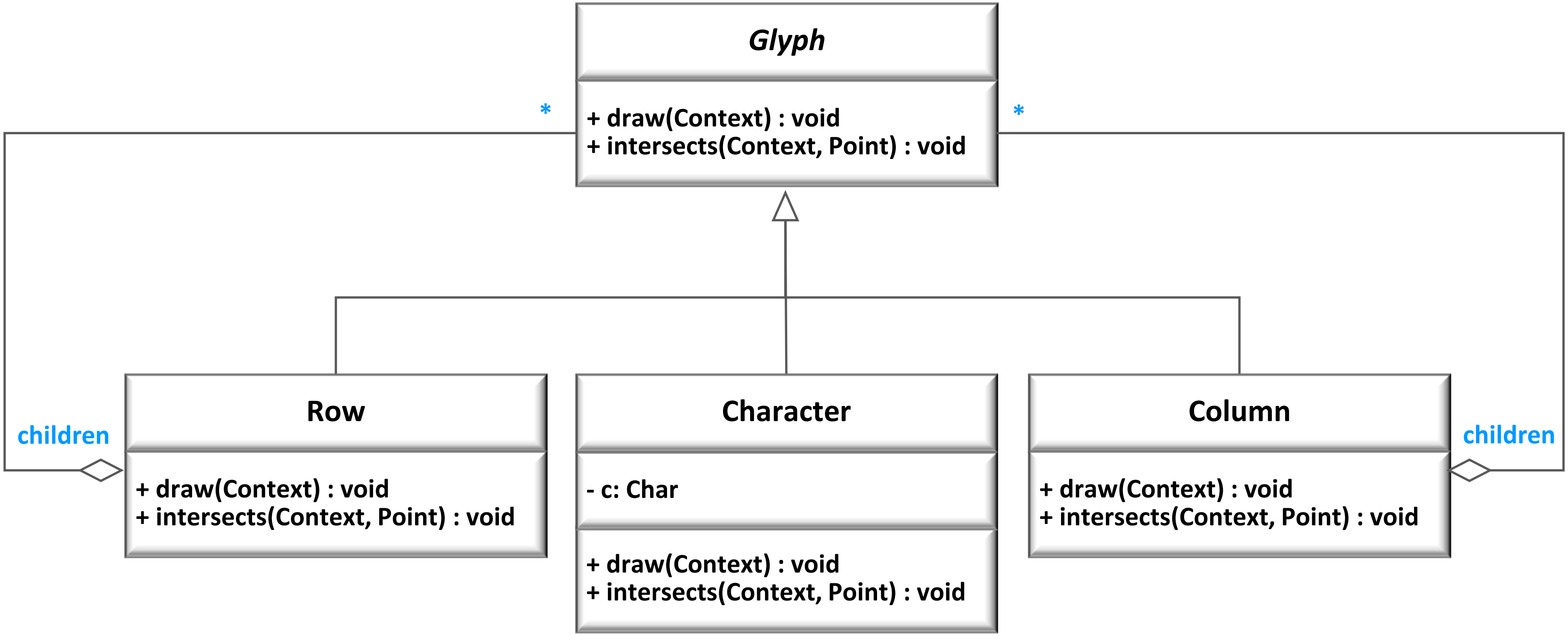
- **Solution**



Motivation (3)



Motivation (4)

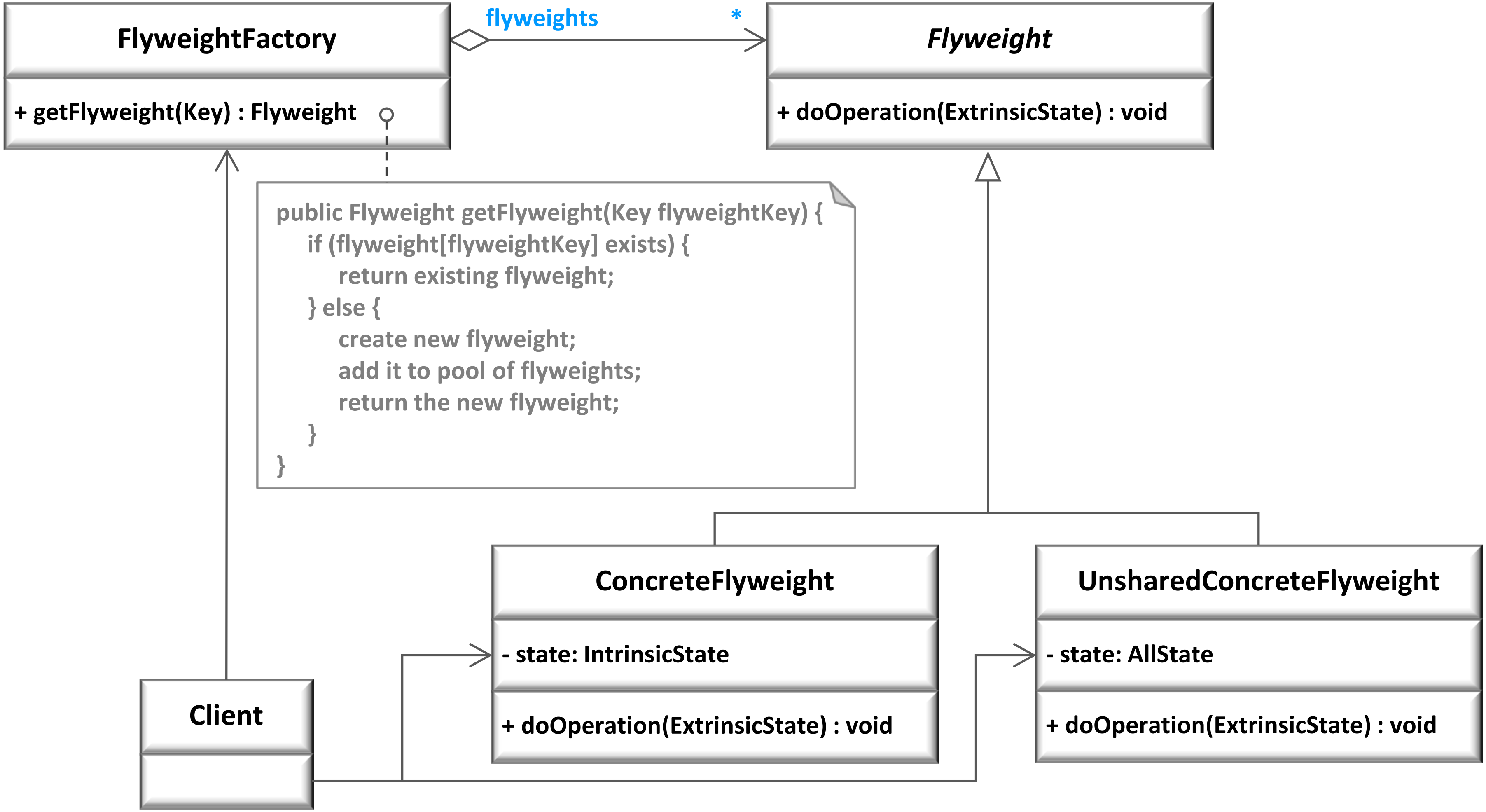


Glyph
+ draw(Context) : void
+ intersects(Context, Point) : void
*
*
Row
+ draw(Context) : void
+ intersects(Context, Point) : void
children
Character
- c: Char
+ draw(Context) : void
+ intersects(Context, Point) : void
Column
+ draw(Context) : void
+ intersects(Context, Point) : void
children

Applicability

- **Apply the Flyweight pattern when all of the following are true:**
  - An application uses a large number of objects.
  - Storage costs are high because of the sheer quantity of objects.
  - Most object state can be made extrinsic.
  - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
  - The application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.

Structure (1)



FlyweightFactory

+ getFlyweight(Key) : Flyweight

flyweights

*

Flyweight

+ doOperation(ExtrinsicState) : void

```
public Flyweight getFlyweight(Key flyweightKey) {
    if (flyweight[flyweightKey] exists) {
        return existing flyweight;
    } else {
        create new flyweight;
        add it to pool of flyweights;
        return the new flyweight;
    }
}
```

ConcreteFlyweight

- state: IntrinsicState

+ doOperation(ExtrinsicState) : void

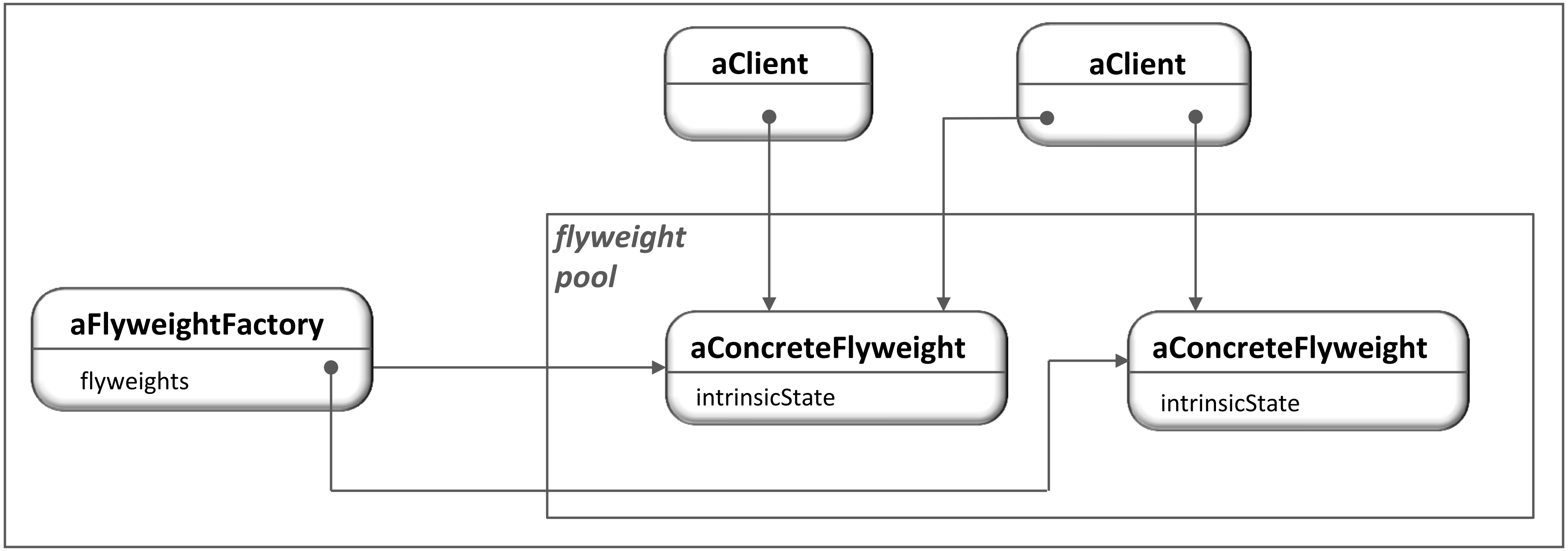
UnsharedConcreteFlyweight

- state: AllState

+ doOperation(ExtrinsicState) : void

Client

Structure (2)


aClient
aClient
flyweight pool
aFlyweightFactory
flyweights
aConcreteFlyweight
intrinsicState
aConcreteFlyweight
intrinsicState

Participants (1)

- **Flyweight**
  - To declare an interface through which flyweights can receive and act on extrinsic state
- **ConcreteFlyweight (Character)**
  - To implement the Flyweight interface and adds storage for intrinsic state, if any
  - A ConcreteFlyweight object must be sharable.
  - Any state it stores must be intrinsic; that is, it must be independent of the ConcreteFlyweight object's context.
- **UnsharedConcreteFlyweight (Row, Column)**
  - Not all Flyweight subclasses need to be shared.
  - The Flyweight interface enables sharing; it doesn't enforce it.
  - It's common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).

Participants (2)

- **FlyweightFactory**
  - To create and manages flyweight objects
  - To ensure that flyweights are shared properly
    - When a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists.
- **Client**
  - To maintain a reference to flyweight(s)
  - To compute or store the extrinsic state of flyweight(s)

Collaborations

- **State that a flyweight needs to function must be characterized as either intrinsic or extrinsic.**
  - Intrinsic state is stored in the ConcreteFlyweight object.
  - Extrinsic state is stored or computed by Client objects.
  - Clients pass this state to the flyweight when they invoke its operations.
- **Clients should not instantiate ConcreteFlyweights directly.**
  - Clients must obtain ConcreteFlyweight objects exclusively from the FlyweightFactory object to ensure they are shared properly.

Consequences

- Flyweights may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state.
- Storage savings are a function of several factors
- The more flyweights are shared, the greater the storage savings. The savings increase with the amount of shared state.
- The Flyweight pattern is often combined with the Composite pattern to represent a hierarchical structure as a graph with shared leaf nodes.

Implementation

- **Removing extrinsic state.**
  - Extrinsic state can be computed from a separate object structure, one with far smaller storage requirements.
- **Managing shared objects.**
  - Because objects are shared, clients shouldn't instantiate them directly.

Sample Code

```
class Glyph {
  public:
    virtual ~Glyph();
    virtual void Draw(Window*,
   GlyphContext&);
   protected:
    Glyph();
  };

  class Character : public Glyph {
  public:
    Character(char);
    virtual void Draw(Window*,
   GlyphContext&);
  private:
    char _charcode;
  };
```

```
const int NCHARCODES = 128;

  class GlyphFactory {
  public:
    GlyphFactory();
    virtual ~GlyphFactory();

    virtual Character* CreateCharacter(char);
    virtual Row* CreateRow();
    virtual Column* CreateColumn();
    // ...
  private:
    Character* _character[NCHARCODES];
  };
```

Known Uses

- **The concept of flyweight objects was first described and explored as a design technique in InterViews 3.0.**
  - Its developers built a powerful document editor called Doc as a proof of concept.
- **ET++ uses flyweights to support look-and-feel independence.**

Related Patterns

- **The Flyweight pattern is often combined with the Composite pattern.**
  - To implement a logically hierarchical structure in terms of a directed-acyclic graph with shared leaf nodes
- **It's often best to implement State and Strategy objects as flyweights.**

Proxy

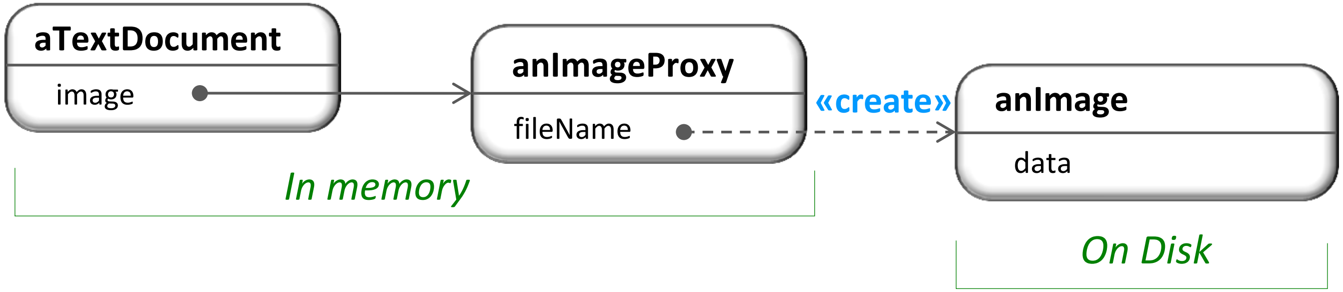
Unit 13

Intent

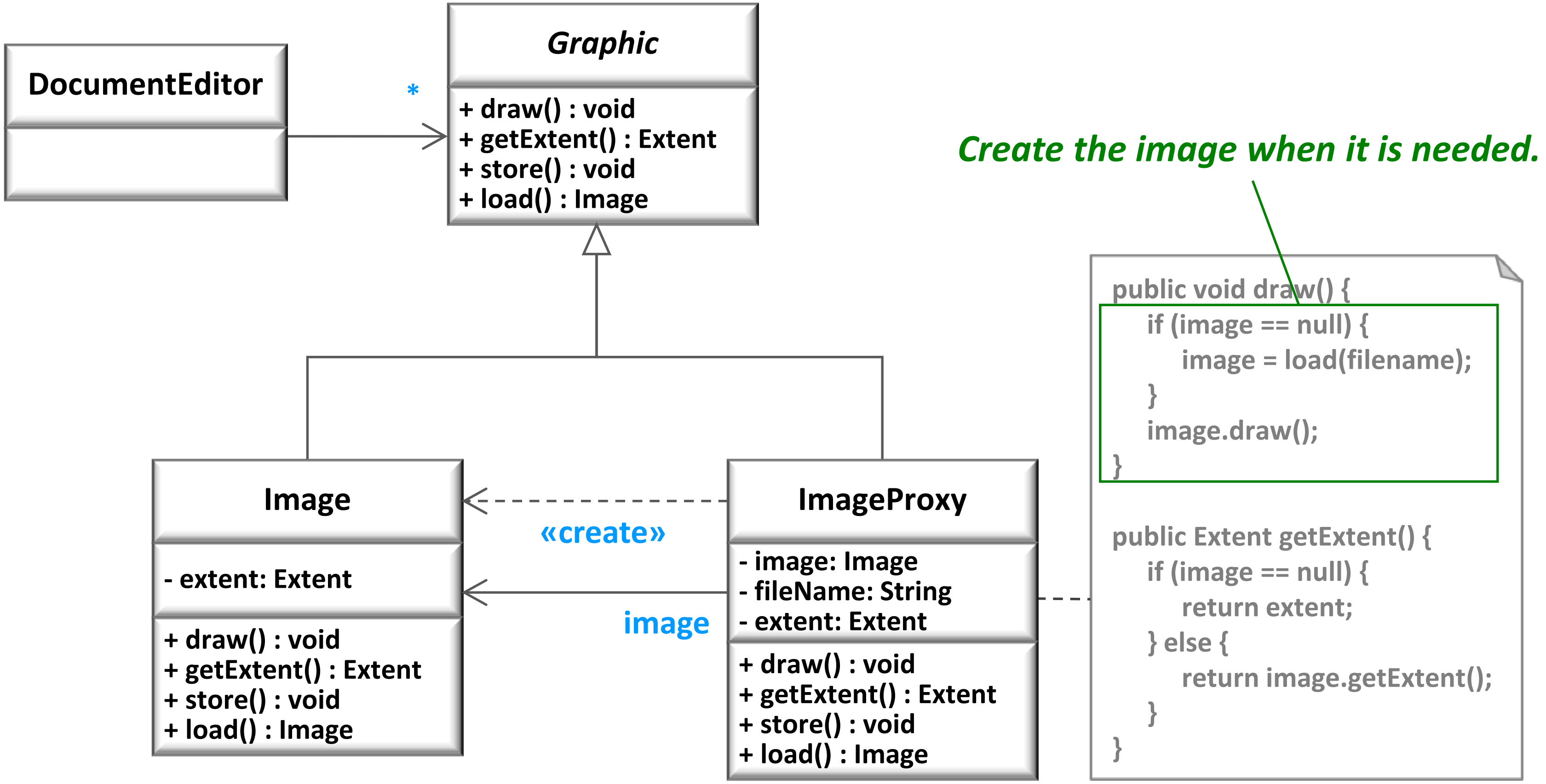
- **Provide a surrogate or placeholder for another object to control access to it.**
- **A proxy is**
  - a person authorized to act for another person
  - an agent or substitute
  - the authority to act for another
- **There are situations in which a client does not or can not reference an object directly, but wants to still interact with the object.**
- **A proxy object can act as the intermediary between the client and the target object.**
  - The proxy object has the same interface as the target object.
  - The proxy holds a reference to the target object and can forward requests to the target as required (delegation!).
  - In effect, the proxy object has the authority the act on behalf of the client to interact with the target object.

Motivation (1)

- **Consider a document editor that can embed graphical objects in a document.**
  - Creating larger graphic objects can be expensive to create, but opening a document should be fast.
  - Need to defer the full cost of its creation and initialization until we actually need to use it
  - A solution for this is to use proxy acting as a stand-in for the real image.
    - The image proxy creates the real image only when the document editor asks it to display itself by invoking its Draw operation.



Motivation (2)



DocumentEditor

*

Graphic

+ draw() : void
+ getExtent() : Extent
+ store() : void
+ load() : Image

Image

- extent: Extent

+ draw() : void
+ getExtent() : Extent
+ store() : void
+ load() : Image

«create»

image

ImageProxy

- image: Image
- fileName: String
- extent: Extent

+ draw() : void
+ getExtent() : Extent
+ store() : void
+ load() : Image

Create the image when it is needed.

```
public void draw() {
    if (image == null) {
        image = load(filename);
    }
    image.draw();
}

public Extent getExtent() {
    if (image == null) {
        return extent;
    } else {
        return image.getExtent();
    }
}
```

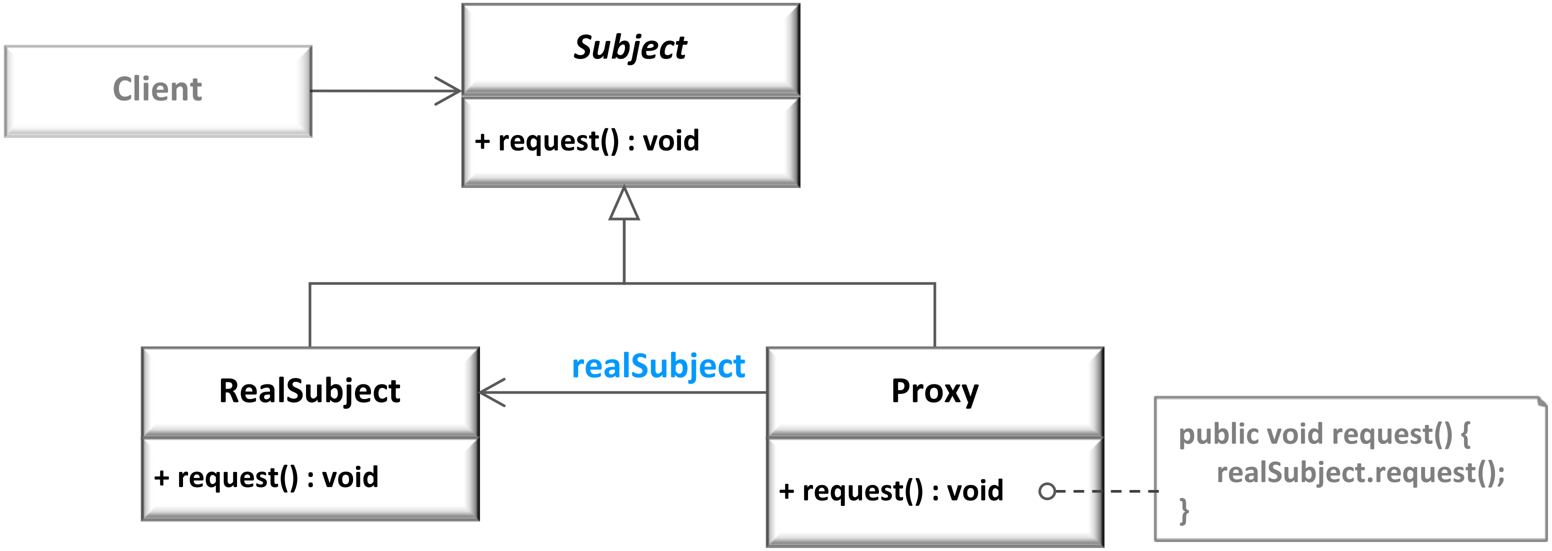
Applicability (1)

- **Proxies are useful wherever there is a need for a more sophisticated reference to a object than a simple pointer or simple reference can provide.**

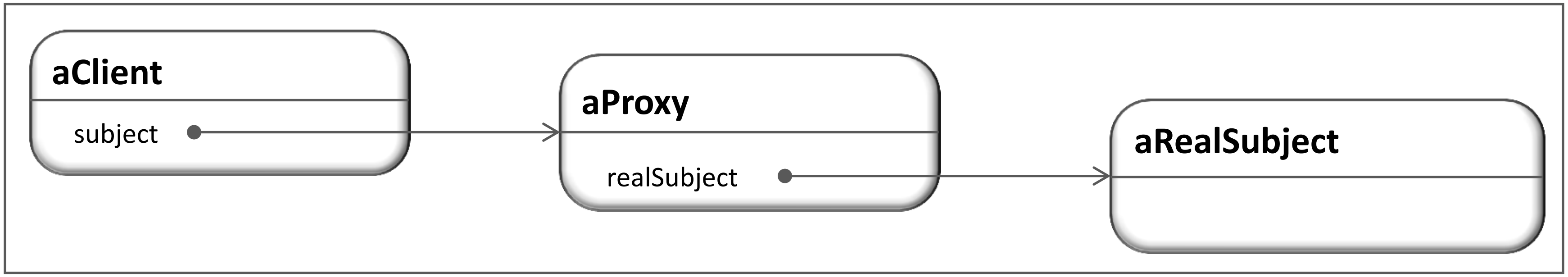
Applicability (2)

- **Common situations in which the Proxy pattern is applicable**
  - Remote Proxy
    - To provide a reference to an object located in a different address space on the same or different machine
  - Virtual Proxy
    - To allow creation of a memory intensive object on demand
    - The object will not be created until it is really needed.
  - Protection (Access) Proxy
    - To provide different clients with different levels of access to a target object
    - Useful when objects should have different access rights.
  - Smart Reference Proxy
    - To provide additional actions whenever a target object is referenced such as;
      - Counting the number of references to the object
      - Loading a persistent object into memory when it's first referenced
      - Checking that the real object is locked before it's accessed to ensure that no other object can change it.

Structure



```
public void request() {
    realSubject.request();
}
```



Participants (1)

- **Proxy (ImageProxy)**
  - To maintain a reference that lets the proxy access the real subject
    - Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
  - To provide an interface identical to Subject's so that a proxy can by substituted for the real subject
  - To control access to the real subject and may be responsible for creating and deleting it
  - Other responsibilities depending on the kind of proxy
    - *Remote Proxies* – To encode a request and its arguments and send the encoded request to the real subject in a different address space
    - *Virtual Proxies* – To cache additional information about the real subject so that they can postpone accessing it
    - *Protection Proxies* – To check that the caller has the access permissions required to perform a request

Participants (2)

- **Subject (Graphic)**
  - To define the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected
- **RealSubject (Image)**
  - To define the real object that the proxy represent

Collaborations

- **Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy.**

Consequences

- **To introduce a level of indirection when accessing an object**
  - A remote proxy can hide the fact that an object resides in a different address space.
  - A virtual proxy can perform optimizations such as creating an object on demand.
  - Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.

Implementation

- **Overloading the member access operator in C++.**
  - C++ supports overloading operator->, the member access operator.
  - Overloading this operator lets you perform additional work whenever an object is dereferenced.
  - This can be helpful for implementing some kinds of proxy; the proxy behaves just like a pointer.
- **Proxy doesn't always have to know the type of real subject.**
  - If a Proxy class can deal with its subject solely through an abstract interface, then there's no need to make a Proxy class for each RealSubject class; the proxy can deal with all RealSubject classes uniformly.

Sample Code

```
class Graphic {
  public:
    virtual ~Graphic();
    virtual void Draw(const Point& at) = 0;
  protected:
    Graphic();
  };

  class Image : public Graphic {
  public:
    Image(const char* file);  // loads image from a file
    virtual ~Image();
    virtual void Draw(const Point& at);
  };
```

```
class ImageProxy : public Graphic {
  public:
    ImageProxy(const char* imageFile);
    virtual ~ImageProxy();
    virtual void Draw(const Point& at);
  protected:
    Image* GetImage();
  private:
    Image* _image;
    Point _extent;
  };
```

Known Uses

- **NEXTSTEP**
  - Uses proxies as local representatives for objects that may be distributed.
- **McCullough discusses using proxies in Smalltalk to access remote objects.**

Related Patterns

- **Adapter**
  - To provide a different interface to the object it adapts, while a proxy provides the same interface as its subject
  - A proxy used for access protection might refuse to perform an operation that the subject will perform, so its interface may be effectively a subset of the subject's.
- **Decorators**
  - To have similar implementations as proxies, but have a different purpose
    - A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object.

Chain of Responsibility

Unit 14

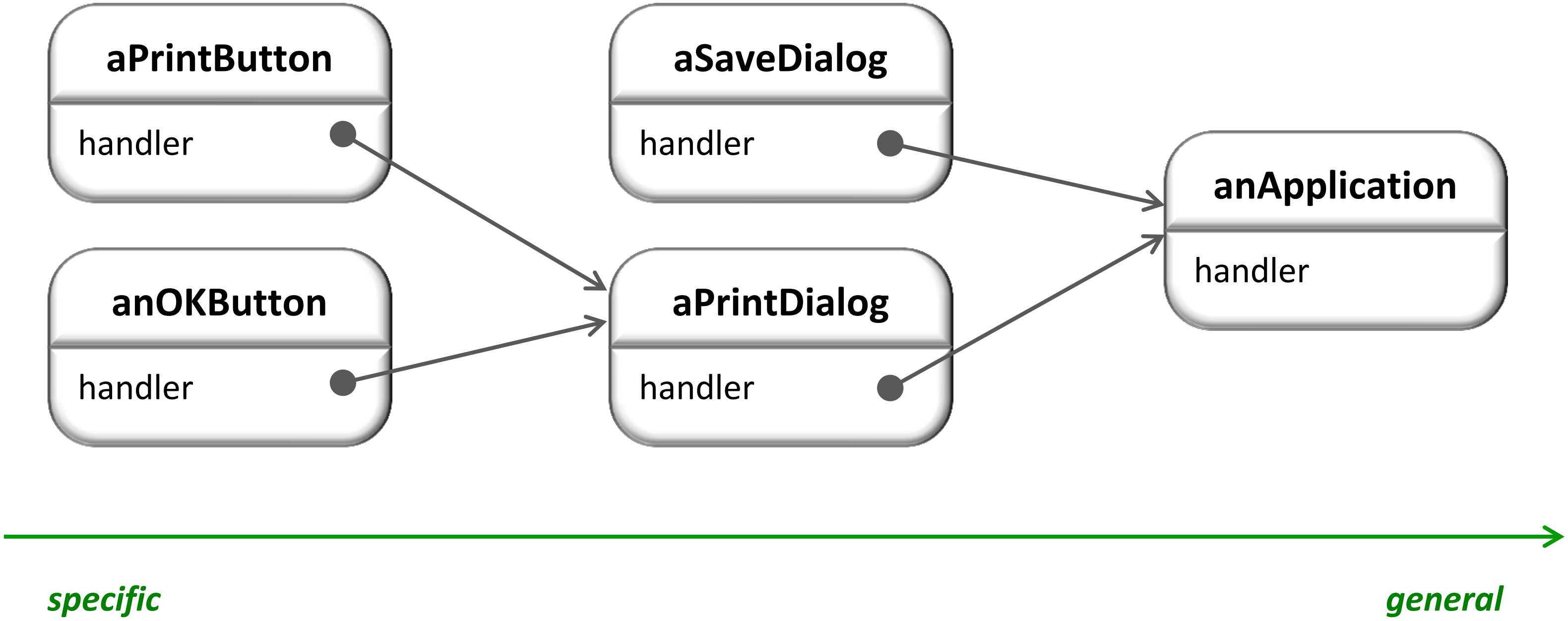
Intent

- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
- Chain the receiving objects and pass the request along the chain until an object handles it.

Motivation (1)

- **Consider a context-sensitive help system for a GUI**
  - The object that ultimately provides the help isn't known explicitly to the object (e.g., a button) that initiates the help request.
- **So use a chain of objects to decouple the senders from the receivers. The request gets passed along the chain until one of the objects handles it.**
- **Each object on the chain shares a common interface for handling requests and for accessing its successor on the chain.**

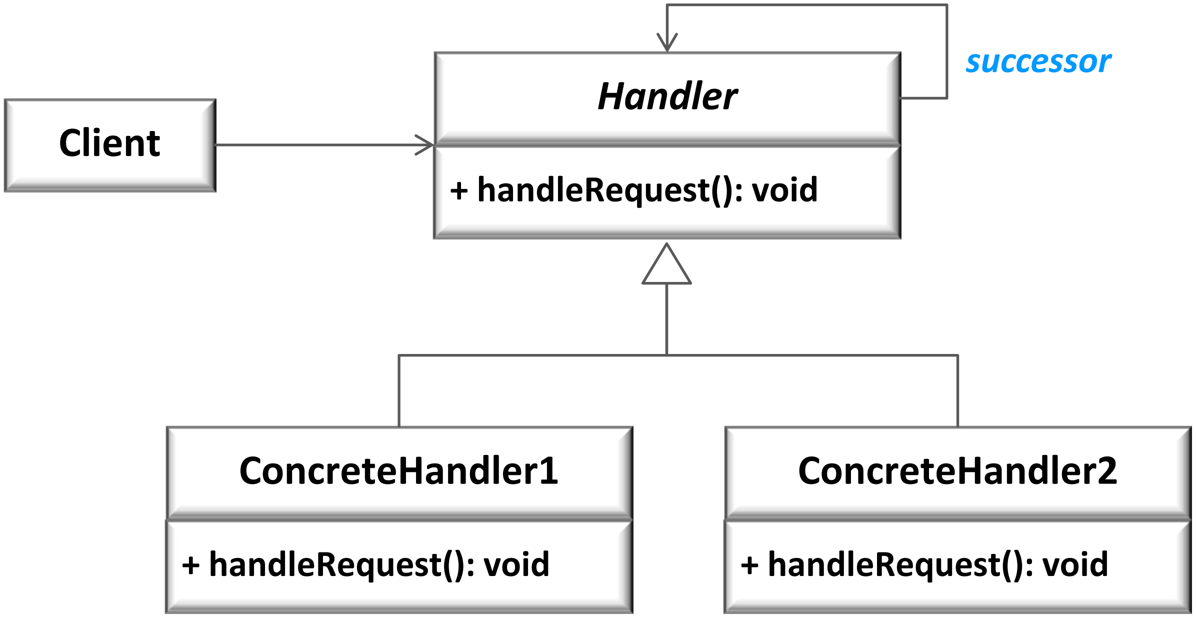
Motivation (2)

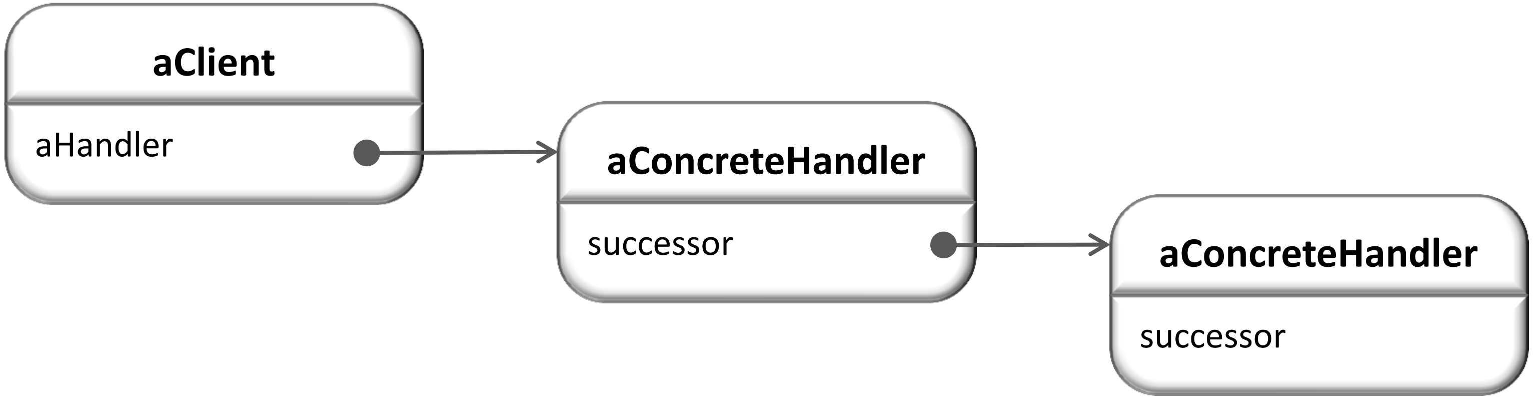


Applicability

- **Use Chain of Responsibility when:**
  - More than one object may handle a request and the actual handler is not know in advance.
  - Requests follow a “handle or forward” model - that is, some requests can be handled where they are generated while others must be forwarded to another object to be handled.

Structure


successor
Handler
+ handleRequest(): void
Client
ConcreteHandler1
+ handleRequest(): void
ConcreteHandler2
+ handleRequest(): void


aClient
aHandler
aConcreteHandler
successor
aConcreteHandler
successor

Participants

- **Handler (HelpHandler)**
  - To define an interface for handling requests
  - (optional) To implement the successor link
- **ConcreteHandler (PrintButton, PrintDialog)**
  - To handle requests it is responsible for
  - To be able to access its successor
  - If the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor.
- **Client**
  - To initiate the request to a ConcreteHandler object on the chain

Collaborations

- **When a client issues a request, the request propagates along the chain until a ConcreteHandler object takes responsibility for handling it.**

Consequences

- Reduced coupling between the sender of a request and the receiver – the sender and receiver have no explicit knowledge of each other
- Receipt is not guaranteed - a request could fall off the end of the chain without being handled
- The chain of handlers can be modified dynamically

Implementation

- **Implementation issues to consider in Chain of Responsibility**
  - Implementing the successor chain
  - Connecting successors
  - Representing requests
  - Automatic forwarding in Smalltalk

Sample Code (1)

```
typedef int Topic;
  const Topic NO_HELP_TOPIC = -1;
    class HelpHandler {
  public:
    HelpHandler(HelpHandler* = 0, Topic =
  NO_HELP_TOPIC);
    virtual bool HasHelp();
    virtual void SetHandler(HelpHandler*,
  Topic);
    virtual void HandleHelp();
  private:
    HelpHandler* _successor;
    Topic _topic;
  };
```

```
class Widget : public HelpHandler {
  protected:
    Widget(Widget* parent, Topic t =
   NO_HELP_TOPIC);
  private:
    Widget* _parent;
  };

  Widget::Widget (Widget* w, Topic t) :
   HelpHandler(w, t) {
    _parent = w;
  }
```

Sample Code (2)

```
class Button : public Widget {
  public:
    Button(Widget* d, Topic t =
   NO_HELP_TOPIC);


    virtual void HandleHelp();
    // Widget operations that Button overrides...
  };
```

```
Button::Button (Widget* h, Topic t) :
   Widget(h, t) { }


 void Button::HandleHelp () {
    if (HasHelp()) {
       // offer help on the button
    } else {
       HelpHandler::HandleHelp();
    }
 }
```

Known Uses

- **MacApp and ET++**
  - "EventHandler,"
- **Symantec's TCL library**
  - "Bureaucrat,"
- **NeXT's AppKit**
  - "Responder."

Related Patterns

- **Applied in conjunction with Composite**
  - There, a component's parent can act as its successor.

Command

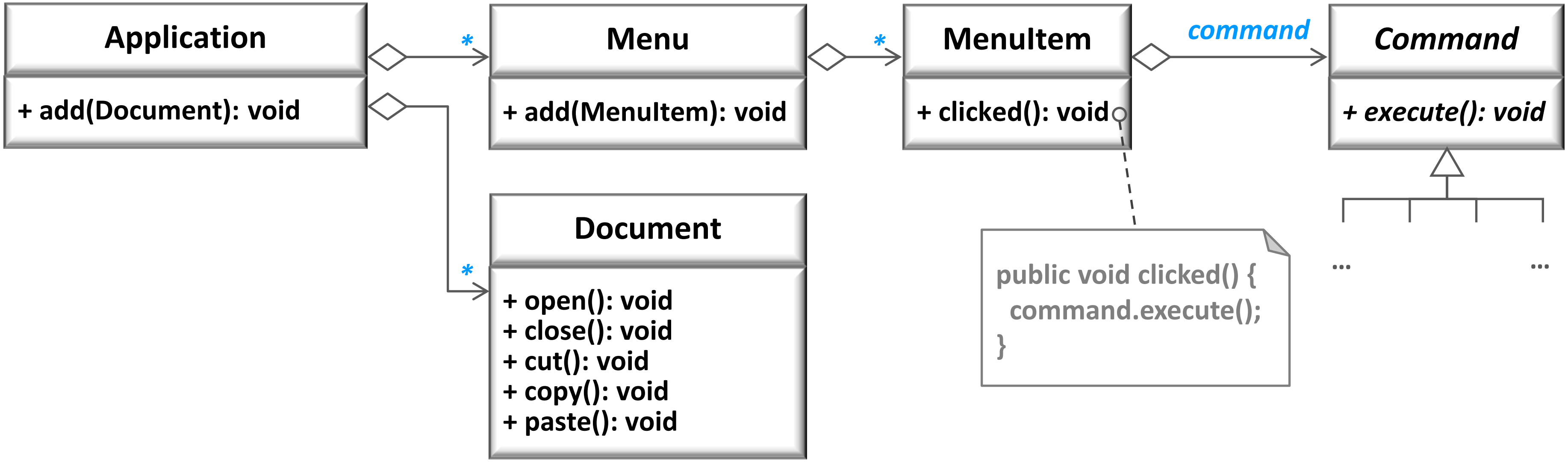
Unit 15

Intent

- Encapsulate requests for service from an object inside other objects, thereby letting you manipulate the requests in various ways

Motivation (1)

- It's necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

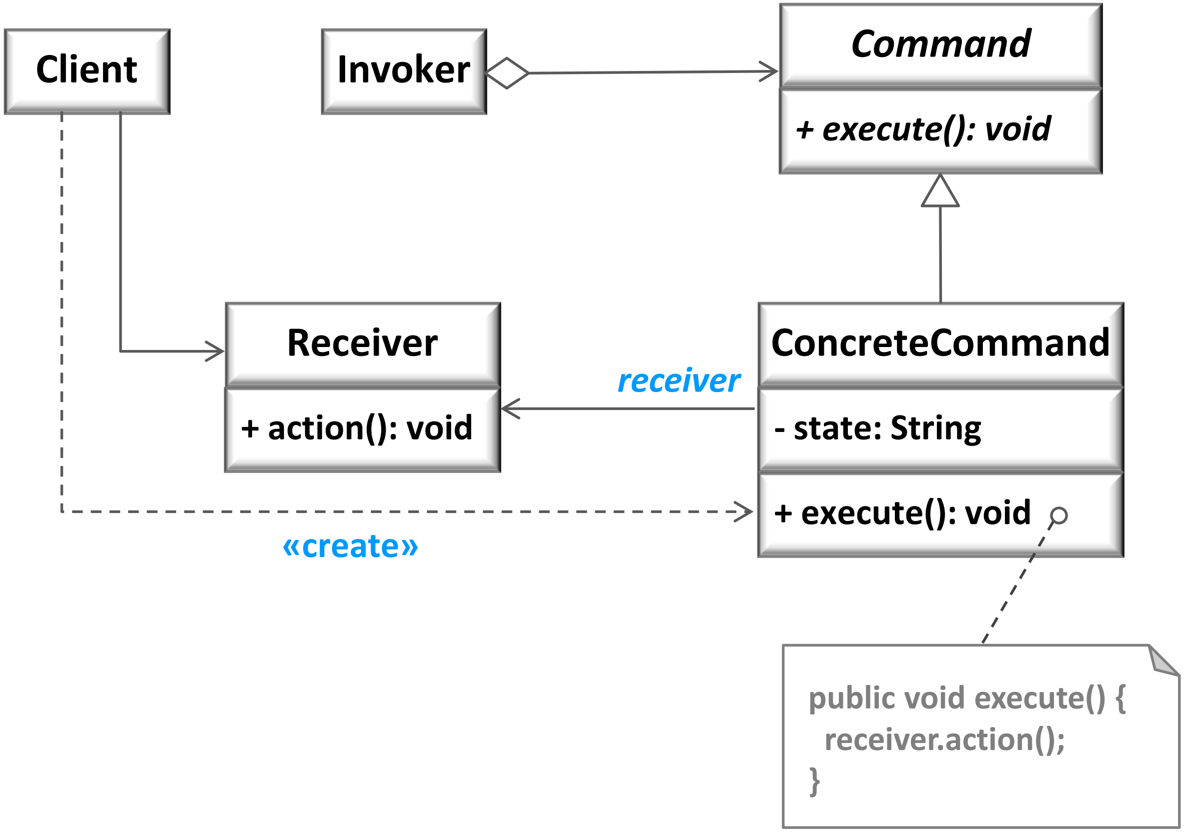


```
public void clicked() {
  command.execute();
}
```

Applicability

- **Use the Command pattern when:**
  - You want to implement a callback function capability.
  - You want to specify, queue, and execute requests at different times.
  - You need to support undo and change log operations.

Structure

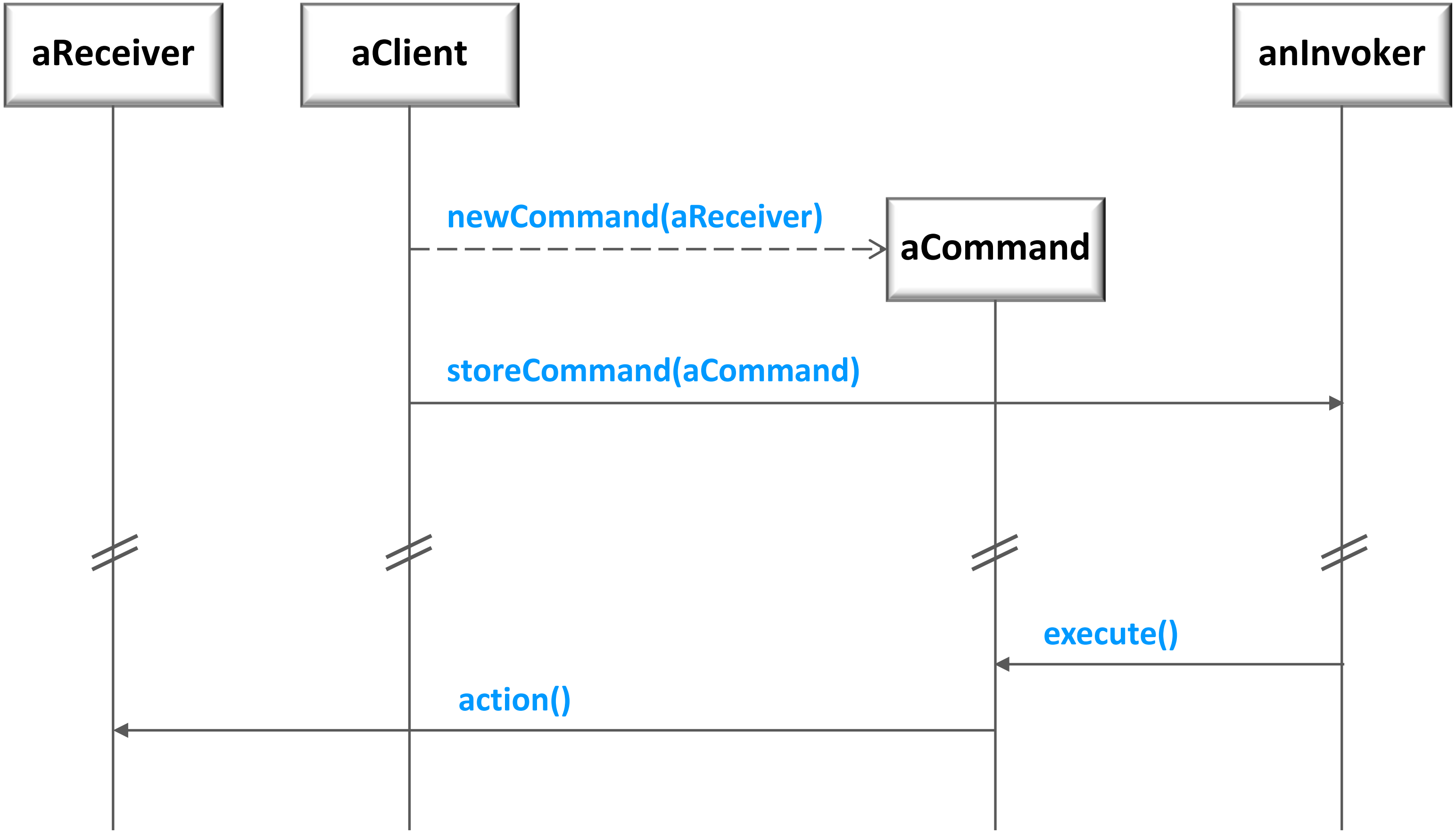


```
public void execute() {
  receiver.action();
}
```

Participants

- **Command**
  - To declare an interface for executing an operation
- **ConcreteCommand (PasteCommand, OpenCommand)**
  - To define a binding between a Receiver object and an action
  - To implement Execute by invoking the corresponding operation(s) on Receiver
- **Client (Application)**
  - To create a ConcreteCommand object and sets its receiver
- **Invoker (MenuItem)**
  - To ask the command to carry out the request
- **Receiver (Document, Application)**
  - To know how to perform the operations associated with carrying out a request
  - Any class may serve as a Receiver.

Collaborations



Consequences

- **Command decouples the object that invokes the operation from the one that knows how to perform it.**
- **Commands are first-class objects.**
    - They can be manipulated and extended like any other object.
- **Commands can be made into a composite command.**

Implementation

- **Consider the following issues when implementing the Command pattern**
  - How intelligent should a command be?
  - Supporting undo and redo.
  - Avoiding error accumulation in the undo process.
  - Using C++ templates.

Sample Code (1)

```
class Command {
  public:
     virtual ~Command();

     virtual void Execute() = 0;
  protected:
     Command();
  };
```

Sample Code (2)

```
class OpenCommand : public Command {
  public:
     OpenCommand(Application*);


     virtual void Execute();
  protected:
     virtual const char* AskUser();
  private:
     Application* _application;
     char* _response;
  };
```

```
OpenCommand::OpenCommand
   (Application* a) {
     _application = a;
  }


 void OpenCommand::Execute () {
     const char* name = AskUser();


     if (name != 0) {
        Document* document = new
   Document(name);
        _application->Add(document);
        document->Open();
     }
  }
```

Known Uses

- **ET++, InterViews, and Unidraw**
  - Define classes that follow the Command pattern
- **InterViews**
  - Define an Action abstract class that provides command functionality.

Related Patterns

- A Composite can be used to implement MacroCommands.
- A Memento can keep state the command requires to undo its effect.
- A command that must be copied before being placed on the history list acts as a Prototype.

Interpreter

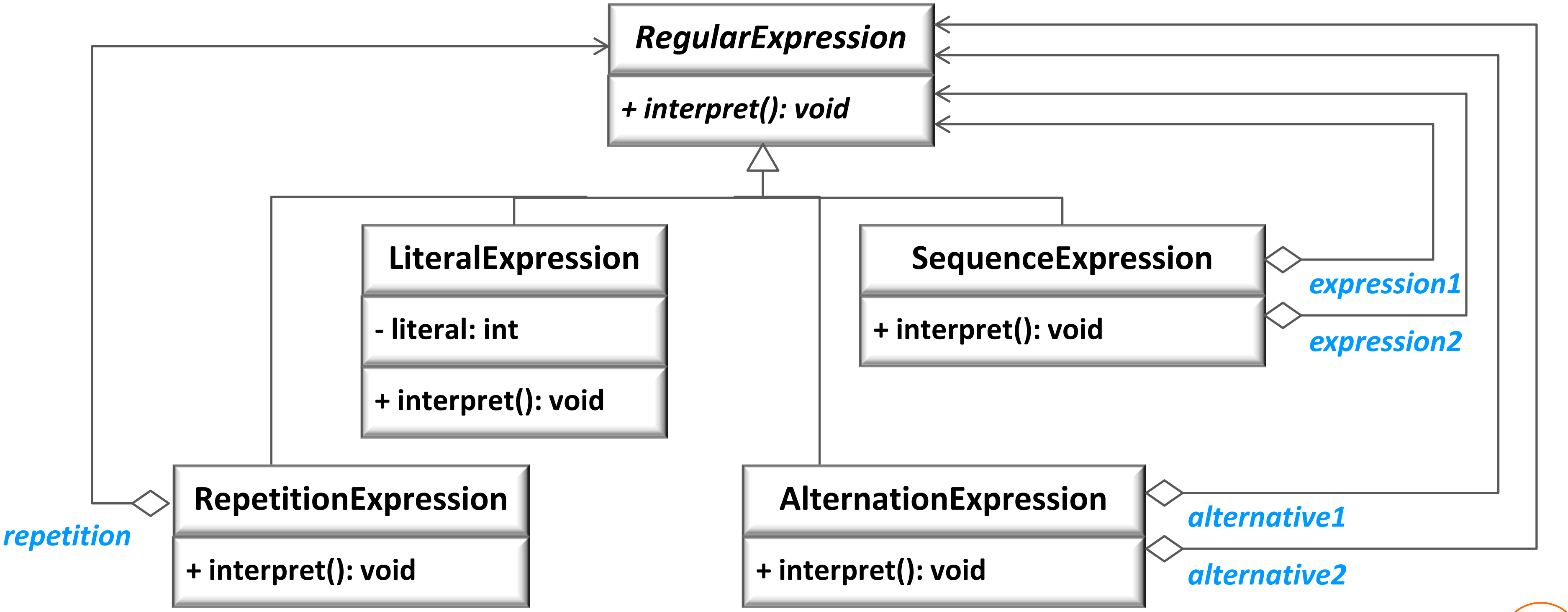
Unit 16

Intent

- Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

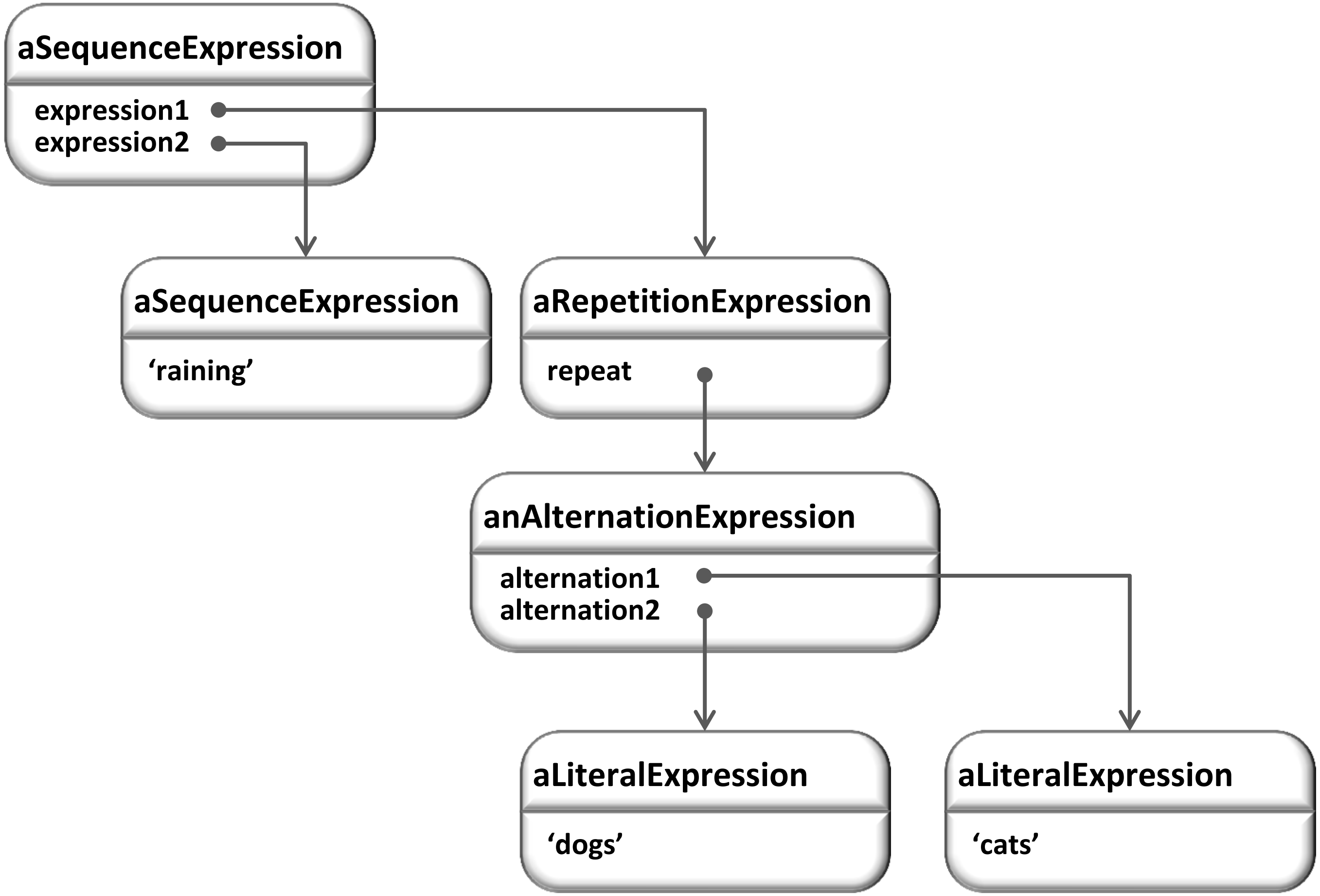
Motivation (1)

- If a particular kind of problem occurs often enough, then it might be worthwhile to express instances of the problem as sentences in a simple language. Then you can build an interpreter that solves the problem by interpreting these sentences.



Motivation (2)

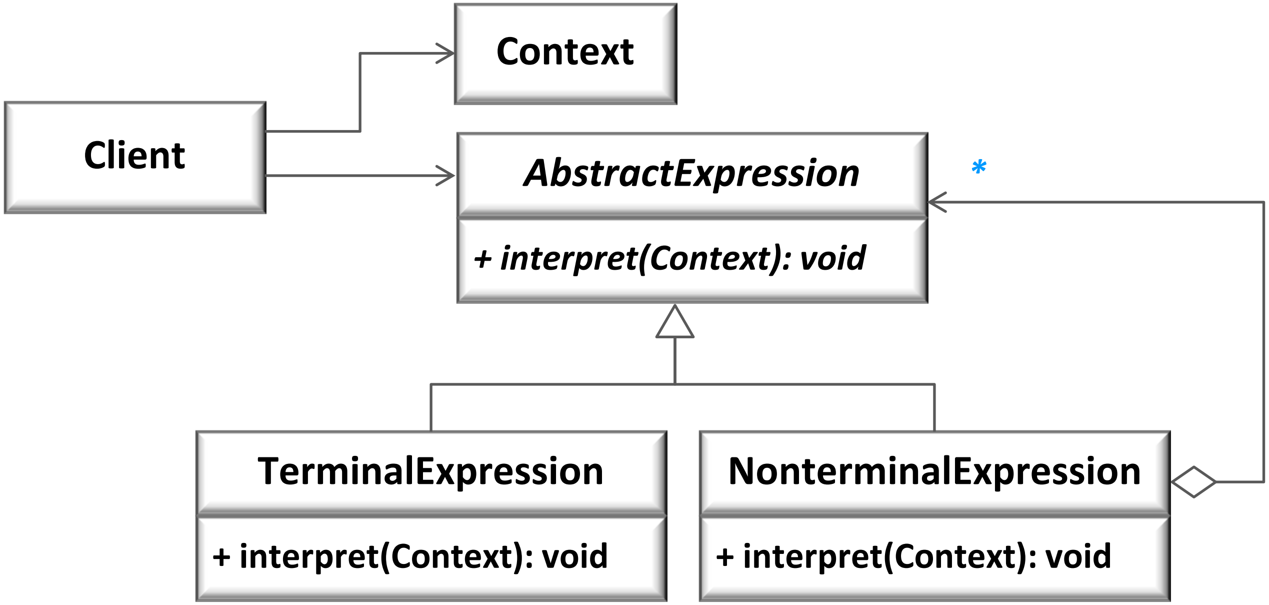
- An abstract syntax tree made up of instances of these classes



Applicability

- **When there is a language to interpret, and you can represent statements in the language as abstract syntax trees.**
- **The Interpreter pattern works best**
  - The grammar is simple.
  - Efficiency is not a critical concern.

Structure


Context
Client
AbstractExpression
+ interpret(Context): void
*
TerminalExpression
+ interpret(Context): void
NonterminalExpression
+ interpret(Context): void

Participants (1)

- **AbstractExpression (RegularExpression)**
  - To declare an abstract Interpret operation that is common to all nodes in the abstract syntax tree
- **TerminalExpression (LiteralExpression)**
  - To implement an Interpret operation associated with terminal symbols in the grammar
  - An instance is required for every terminal symbol in a sentence.
- **NonterminalExpression (AlternationExpression, RepetitionExpression, SequenceExpressions)**
  - One such class is required for every rule R ::= R_1 R_2 ... R_n in the grammar.
  - To maintain instance variables of type AbstractExpression for each of the symbols R_1 through R_n
  - To implement an Interpret operation for nonterminal symbols in the grammar.
    - Interpret typically calls itself recursively on the variables representing R_1 through n.

Participants (2)

- **Context**
  - To contain information that's global to the interpreter
- **Client**
  - To build (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines.
    - The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes.
  - To invoke the Interpret operation

Collaborations

- The client builds (or is given) the sentence as an abstract syntax tree of NonterminalExpression and TerminalExpression instances.
- Then the client initializes the context and invokes the Interpret operation.
- Each NonterminalExpression node defines Interpret in terms of Interpret on each subexpression.
- The Interpret operation of each TerminalExpression defines the base case in the recursion.
- The Interpret operations at each node use the context to store and access the state of the interpreter.

Consequences

- **Advantages**
  - Easy to change and extend the ways to interpret expressions
  - Easy to implement the ways to interpret expressions
- **Disadvantages**
  - Hard to maintain complex ways to interpret expressions

Implementation

- **Following issues are specific to Interpreter**
  - Creating the abstract syntax tree.
  - Defining the Interpret operation.
  - Sharing terminal symbols with the Flyweight pattern.

Sample Code (1)

```
class AndExp : public BooleanExp {
 public:
     AndExp(BooleanExp*, BooleanExp*);
     virtual ~ AndExp();

     virtual bool Evaluate(Context&);
     virtual BooleanExp* Replace(const char*, BooleanExp&);
     virtual BooleanExp* Copy() const;
 private:
     BooleanExp* _operand1;
     BooleanExp* _operand2;
 };

 AndExp::AndExp (BooleanExp* op1, BooleanExp* op2) {
     _operand1 = op1;
     _operand2 = op2;
 }
```

Sample Code (2)

```
class AndExp : public BooleanExp {
 public:
    AndExp(BooleanExp*, BooleanExp*);
    virtual ~ AndExp();


    virtual bool Evaluate(Context&);
    virtual BooleanExp* Replace(const char*,
  BooleanExp&);
    virtual BooleanExp* Copy() const;
 private:
    BooleanExp* _operand1;
    BooleanExp* _operand2;
 };
```

```
AndExp::AndExp (BooleanExp* op1,
  BooleanExp* op2) {
    _operand1 = op1;
    _operand2 = op2;
 }


bool AndExp::Evaluate (Context& aContext) {
    return
       _operand1->Evaluate(aContext) &&
       _operand2->Evaluate(aContext);
 }
```

Known Uses

- **The Smalltalk compilers**
- **SPECTalk**
  - To interpret descriptions of input file formats
- **The QOCA constraint-solving toolkit**
  - To evaluate constraints

Related Patterns

- **Composite**
  - The abstract syntax tree is an instance of the Composite pattern.
- **Flyweight**
  - How to share terminal symbols within the abstract syntax tree
- **Iterator**
  - Use an Iterator to traverse the structure.
- **Visitor**
  - To maintain the behavior in each node in the abstract syntax tree in one class

Iterator

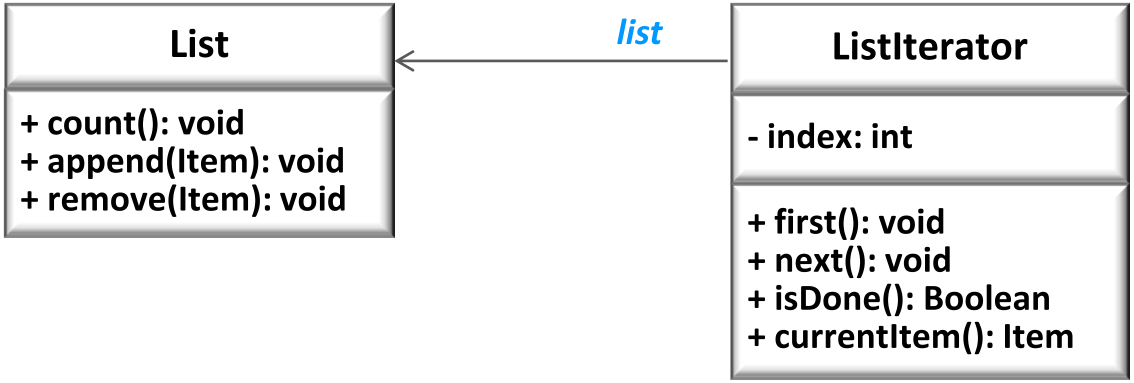
Unit 17

Intent

- **To provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation**
- **An aggregate object is an object that contains other objects for the purpose of grouping those objects as a unit.**
    - It is also called a container or a collection.
    - Examples are a linked list and a hash table.

Motivation (1)

- An aggregate object such as a list should allow a way to traverse its elements without exposing its internal structure.
- To allow different traversal methods without bloating the List interface with operations for different traversals
- To allow multiple traversals pending on the same list.

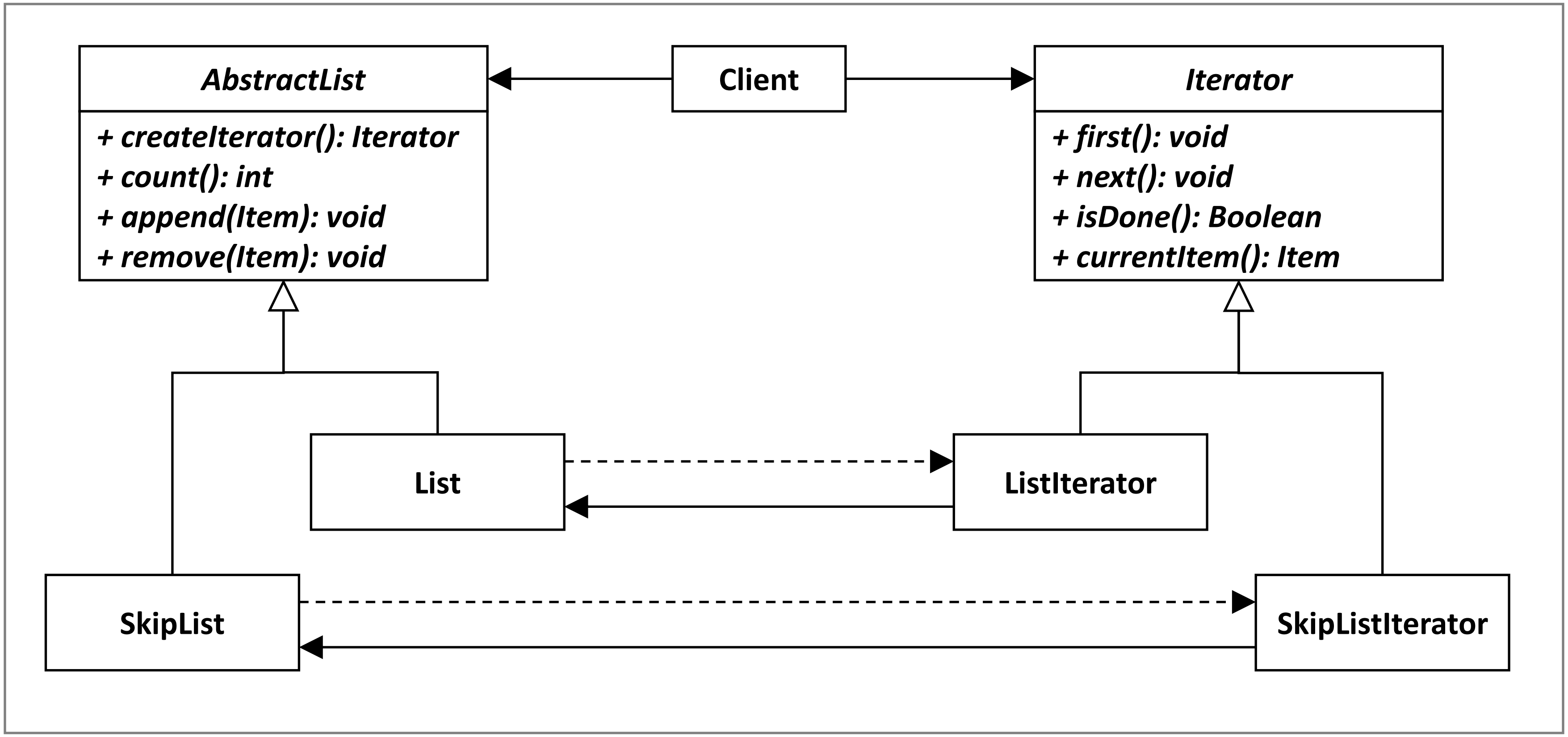


Motivation (2)

- To define iterators for different traversal policies without enumerating them in a List interface
- To allow changing the aggregate class without changing implementations of a client by generalizing the iterator concept to support *polymorphic iteration*

Example

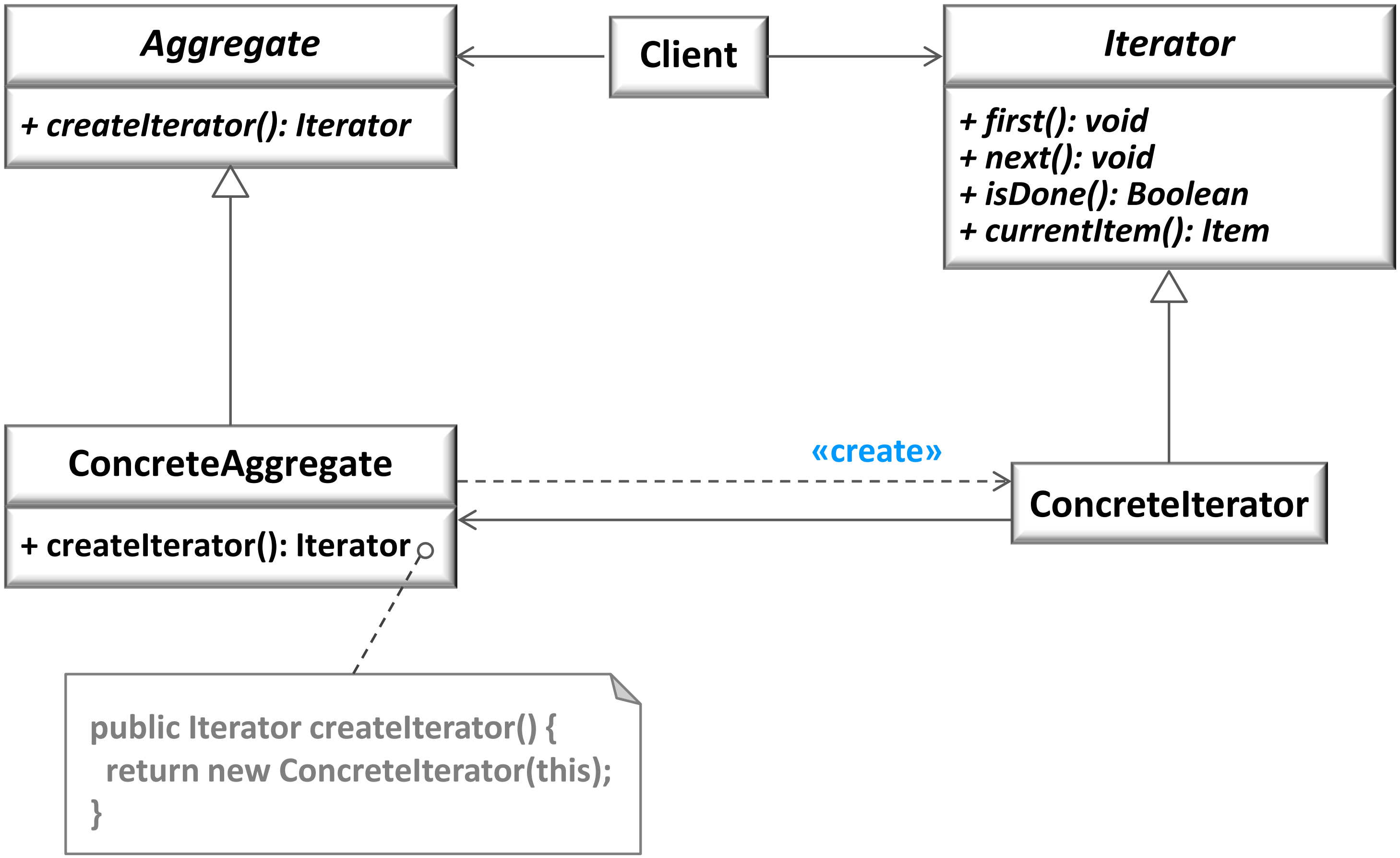
- **SkipList**
  - A probabilistic data structure with characteristics similar to balanced trees



Applicability

- **Use Iterator Patter when:**
  - To access an aggregate object’s contents without exposing its internal representation
  - To support multiple traversals of aggregate objects
  - To provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration)

Structure



Aggregate

+ createIterator(): Iterator

Client

Iterator

+ first(): void
+ next(): void
+ isDone(): Boolean
+ currentItem(): Item

ConcreteAggregate

+ createIterator(): Iterator

«create»

ConcreteIterator

```
public Iterator createIterator() {
  return new ConcreteIterator(this);
}
```

Participants

- **Iterator**
  - To define an interface for accessing and traversing elements
- **ConcreteIterator**
  - To implement the Iterator interface
  - To keep track of the current position in the traversal
- **Aggregate**
  - To define an interface for creating an Iterator object (a factory method!)
- **ConcreteAggregate**
  - To implement the Iterator creation interface to return an instance of the proper ConcreteIterator

Collaborations

- A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.

Consequences

- **To support variations in the traversal of an aggregate**
- **To simplify the aggregate interface**
  - Iterator's traversal interface obviates the need for a similar interface in Aggregate.
- **To support multiple, concurrent traversals**

Implementation

- **Important *Iterator* implementation variants and alternatives**
  - Who controls the iteration?
  - Who defines the traversal algorithm?
  - How robust is the iterator?
  - Additional Iterator operations
  - Using polymorphic iterators in a target programming language
  - Iterators may have privileged access.
  - Iterators for composites
  - Null iterators

Sample Code – List and Iterator Interface (1)

- 1. List and Iterator Interfaces

```
template <class Item>
Class List {
    public:
        List(long size = DEFAULT_LIST_CAPACITY);
        long count() const;
        Item& get(long index) const;
        // …
};

template <class Item>
class Iterator {
    public:
        virtual void first() = 0;
        virtual void next() = 0;
        virtual bool isDone() const = 0;
        virtual Item currentItem() const = 0;

    protected:
        Iterator();
};
```

- 2. Iterator subclass impl.

```
template <class Item>
class ListIterator : public Iterator<Item> {
    public:
        ListIterator(const List<Item>* aList);
        virtual void first();
        virtual void next();
        virtual bool isDone() const;
        virtual Item currentItem() const;

    private:
        const List<Item>* _list;
        long _current;
};

Template <class Item>
ListIterator<Item>::ListIterator(const List<Item>* aList)
    : _list(aList), _current(0) {}

Template <class Item>
void ListIterator<Item>::fist ()
{   _current = 0; }
```

Sample Code – List and Iterator Interface (2)

- **2. Iterator subclass impl. (Continue)**

```
Template <class Item>
void ListIterator<Item>::next ()
{   _current++; }

Template <class Item>
bool ListIterator<Item>::isDone () const
{   return _current >= _list -> Count(); }

Template <class Item>
Item ListIterator<Item>::currentItem () const {
    if (isDone()) {
        throw IteratorOutOfBounds;
    }

    return _list -> Get(_current);
}
```

- **3. Using the iterators**

```
void PrintEmployees (Iterator<Employee*>& i) {
    for (i.first(); !i.isDone(); i.next()) {
        i.currentItem() -> Print();
    }
}

// Since we have iterators for both back-to-front and
front-to-back traversals, we can reuse this operation to
print the employees in both orders.

List<Employee*>* employees;
// …
ListIterator<Employee*> forward(employees);
ReverseListIterator<Employee*>
    backward(employees);
PrintEmployees (forward);
PrintEmployees (backward);
```

Sample Code – List and Iterator Interface (3)

- **4. Avoiding commitment to a specific list impl.**

```
SkipList<Employee*>* employees;
// …
SkipListIterator<Employee*> iterator(employees);
PrintEmployees (iterator);

// To enable polymorphic iteration, AbstractList defines
a factory method createIterator
template <class Item>
class AbstractList {
    public:
        virtual Iterator<Item>* createIterator() const = 0;
        // …
};

// List overrides createIterator to return a ListIterator
object
template <class Item>
Iterator<Item>* List<Item>::createIterator() const {
    return new ListIterator<Item>(this);
}
```

- **5. Making user iterators get deleted**

```
template <class Item>
class IteratorPtr {
    public:
        IteratorPtr(Iterator<Item>* i): _i(i) { }
        ~IteratorPtr() { delete _i; }
        Iterator<Item>* operator->() { return _i; }
        Iterator<Item>& operator*() { return *_i; }
    private:
        // disallow copy and assignment to avoid
        // multiple deletions of _i:
        IteratorPtr(const IteratorPtr&);
        IteratorPtr& operator=(const IteratorPtr&);
    private:
        Iterator<Item>* _i;
};

// IteratorPtr lets us simplify our printing code:
AbstractList<Employee*>* employees;
// ...
IteratorPtr<Employee*> iterator(employees
        ->CreateIterator());
PrintEmployees(*iterator);
```

Sample Code – List and Iterator Interface (4)

- **6. An Internal ListIterator**

```
// rely on subclassing
template <class Item>
class ListTraverser {
    public:
        ListTraverser(List<Item>* aList);
        bool Traverse();

    protected:
        virtual bool ProcessItem(const Item&) = 0;

    private:
        ListIterator<Item> _iterator;
};
```

```
// ListTraverser takes a List instance as a parameter.
template <class Item>
ListTraverser<Item>::ListTraverser ( List<Item>* aList )
: _iterator(aList) { }

// Traverse starts the traversal and calls ProcessItem for
each item.
// terminating a traversal by returning false from
ProcessItem
template <class Item>
bool ListTraverser<Item>::Traverse () {
    bool result = false;
    for ( _iterator.First();
            !_iterator.IsDone();_iterator.Next() ) {

        result = ProcessItem(_iterator.CurrentItem());

        if (result == false) {
            break;
        }
    }
    return result;
}
```

Sample Code – List and Iterator Interface (5)

```
// Using a ListTraverser to print the first 10 employees
from the employee list.

class PrintNEmployees : public
      ListTraverser<Employee*> {
  public:
    PrintNEmployees(List<Employee*>* aList, int n) :
    ListTraverser<Employee*>(aList),
    _total(n), _count(0) { }

  protected:
    bool ProcessItem(Employee* const&);

  private:
    int _total;
    int _count;
};

bool PrintNEmployees::ProcessItem (
      Employee* const& e) {
  _count++;
  e->Print();
  return _count < _total;
}
```

```
// Here's how PrintNEmployees prints the first 10
employees on the list.

List<Employee*>* employees;
// ...
PrintNEmployees pa(employees, 10);
pa.Traverse();

// iteration loop using an external iterator
ListIterator<Employee*> i (employees);
int count = 0;
for (i.first(); !i.isDone(); i.next()) {
  count++;
  i.currentItem()->Print();
  if (count >= 10) {
    break;
  }
}
```

Sample Code – List and Iterator Interface (6)

```
// Internal iterators can encapsulate different kinds of
iteration.
// FilteringListTraverser encapsulates an iteration that
processes only items that satisfy a test.

template <class Item>
class FilteringListTraverser {
    public:
        FilteringListTraverser(List<Item>* aList);
        bool Traverse();

    protected:
        virtual bool ProcessItem(const Item&) = 0;
        virtual bool TestItem(const Item&) = 0;

    private:
        ListIterator<Item> _iterator;
};
```

```
// Traverse decides to continue the traversal based on
theoutcome of the test.

template <class Item>
void FilteringListTraverser<Item>::Traverse () {
    bool result = false;

    for (_iterator.First();!_iterator.IsDone();
            _iterator.Next() ) {

        if (TestItem(_iterator.CurrentItem())) {
            result = ProcessItem(_iterator.CurrentItem());

            If (result == false) {
                    break;
            }
        }
    }
    return result;
}

// A variant of this class could define Traverse to return
if at least one item satisfies the test.
```

Known Uses

- **Most collection class libraries in Object-oriented systems**
- **Booch components**
  - A popular collection class library
  - To provide bounded and unbounded implementation of a queue
- **Standard collection classes in Smalltalk**
  - Bag, Set, Dictionary, OrderedCollection, String, and etc.
- **ET++ container classes**
  - Polymorphic iterators and the cleanup Proxy
- **ObjectWindows 2.0**
  - To provide a class hierarchy of iterators for containers

Related Patterns

- **Composite**
  - Iterators are often applied to recursive structures such as composites.
- **Factory Method**
  - Polymorphic iterators rely on factory methods to instantiate the appropriate iterator subclass.
- **Memento**
  - An iterator can use a memento to capture the state of an iteration.

Mediator

Unit 18

Intent

- To define an object that encapsulates how a set of objects interact
- To promote loose coupling by keeping objects from referring to each other explicitly
- To allow multiple objects interaction independently

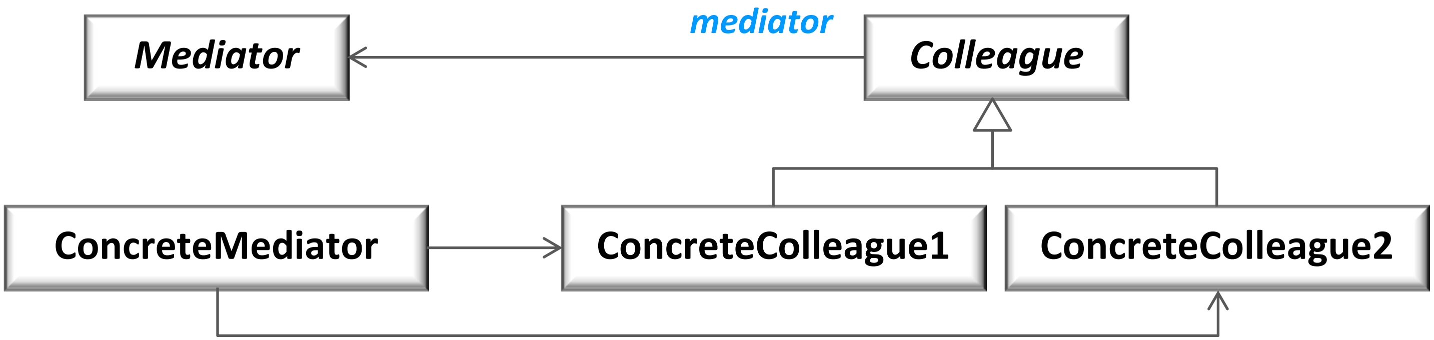
Motivation

- **Object-oriented design encourages the distribution of behavior among objects.**
  - Such distribution can result in an object structure with many connections between objects.
- **Partitioning a system into many objects generally enhances reusability, proliferating interconnections tend to reduce it again.**

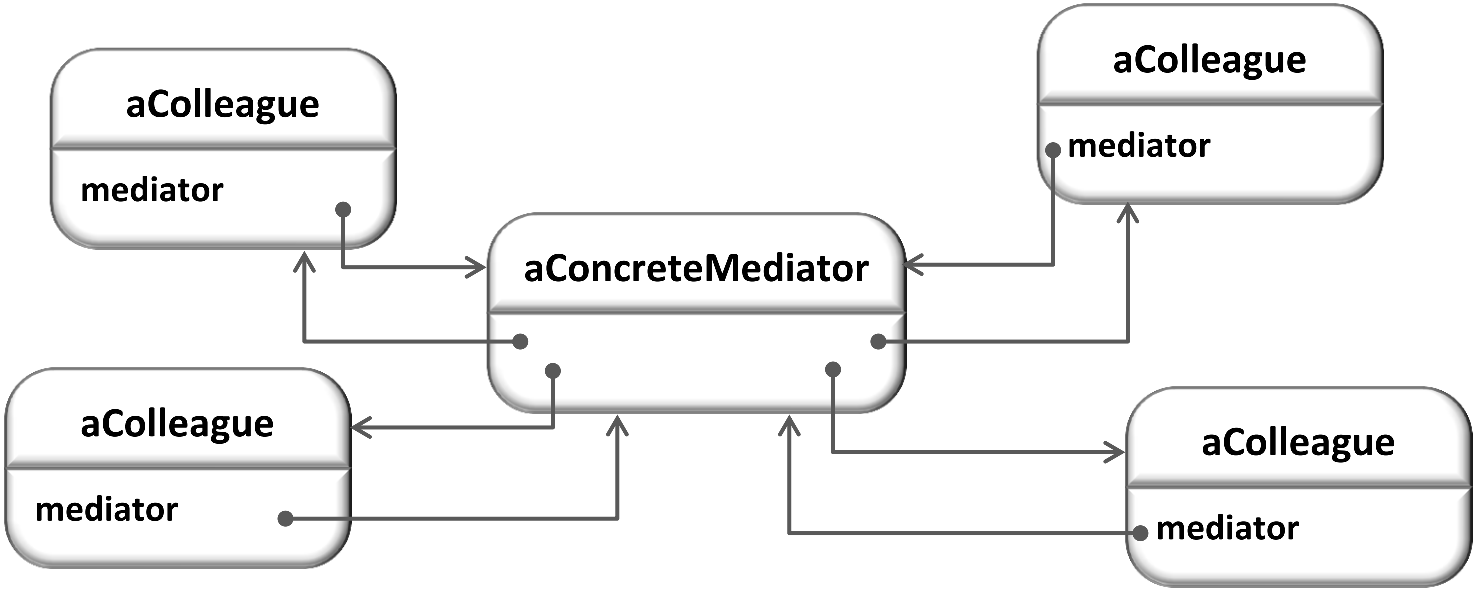
Applicability

- **Applicable Situations:**
  - A set of objects communicate in well-defined but complex ways.
    - The resulting interdependencies are unstructured and difficult to understand.
  - Reusing an object is difficult because it refers to and communicates with many other objects.
  - A behavior that's distributed between several classes should be customizable without a lot of subclassing.

Structure



- A typical object structure



Participants

- **Mediator (DialogDirector)**
  - To define an interface for communicating with Colleague objects
- **ConcreteMediator (FontDialogDirector)**
  - To implement cooperative behavior by coordinating Colleague objects
  - To know and maintain its colleagues
- **Colleague classes (ListBox, EntryField)**
  - Each Colleague class knows its Mediator object.
  - Each colleague communicates with its mediator whenever it would have communicated with another colleague.

Collaborations

- Colleagues send and receive requests from a Mediator object.
- The mediator implements the cooperative behavior by routing requests between the appropriate colleague(s).

Consequences

- To limit subclassing
- To decouple colleagues
- To simplify object protocols
- To abstract how objects cooperate
- To centralize controls

Implementation

- **Omitting the abstract Mediator class**
  - There's no need to define an abstract Mediator class when colleagues work with only one mediator.
  - The abstract coupling by the Mediator class lets colleagues work with different Mediator subclasses, and vice versa.
- **Colleague-Mediator communication**
  - Colleagues have to communicate with their mediator when an event occurs.

Sample Code – DialogDirector (1)

```
// The abstract class DialogDirector defines the interface
for directors.

class DialogDirector {
    public:
        virtual ~DialogDirector();
        virtual void ShowDialog();
        virtual void WidgetChanged(Widget*) = 0;

protected:
        DialogDirector();
        virtual void CreateWidgets() = 0;
};
```

```
// Widget is the abstract base class for widgets.
// A widget knows its director.

class Widget {
    public:
        Widget(DialogDirector*);
        virtual void Changed();
        virtual void HandleMouse(MouseEvent& event);
        // ...

    private:
        DialogDirector* _director;
};

// Changed calls the director's WidgetChanged
operation.
// Widgets call WidgetChanged on their director to
inform it of a significant event.

void Widget::Changed () {
    _director->WidgetChanged(this);
}
```

Sample Code – DialogDirector (2)

```
// The ListBox, EntryField, and Button are subclasses of
Widget for specialized user interface elements.

class ListBox : public Widget {
    public:
        ListBox(DialogDirector*);
        virtual const char* GetSelection();
        virtual void SetList(List<char*>* listItems);
        virtual void HandleMouse(MouseEvent& event);
        // ...
};
class EntryField : public Widget {
    public:
        EntryField(DialogDirector*);
        virtual void SetText(const char* text);
        virtual const char* GetText();
        virtual void HandleMouse(MouseEvent& event);
        // ...
};
class Button : public Widget {
    public:
        Button(DialogDirector*);
        virtual void SetText(const char* text);
        virtual void HandleMouse(MouseEvent& event);
        // ...
};
```

```
// Button is a simple widget that calls Changed
whenever it’s pressed.

void Button::HandleMouse (MouseEvent& event) {
    // ...
    Changed();
}
```

Sample Code – DialogDirector (3)

```
// The FontDialogDirector class mediates between
widgets in the dialog box.

class FontDialogDirector : public DialogDirector {
    public:
        FontDialogDirector();
        virtual ~FontDialogDirector();
        virtual void WidgetChanged(Widget*);

    protected:
        virtual void CreateWidgets();

private:
        Button* _ok;
        Button* _cancel;
        ListBox* _fontList;
        EntryField* _fontName;
};
```

```
// FontDialogDirector keeps track of the widgets it
displays.

void FontDialogDirector::CreateWidgets () {
    _ok = new Button(this);
    _cancel = new Button(this);
    _fontList = new ListBox(this);
    _fontName = new EntryField(this);
    // fill the listBox with the available font names
    // assemble the widgets in the dialog
}

// WidgetChanged ensures that the widgets work
together properly.

void FontDialogDirector::WidgetChanged ( Widget*
            theChangedWidget ) {
    if (theChangedWidget == _fontList) {
        _fontName->SetText(_fontList->GetSelection());
    } else if (theChangedWidget == _ok) {
        // apply font change and dismiss dialog
        // ...
    } else if (theChangedWidget == _cancel) {
        // dismiss dialog
    }
}
```

Known Uses

- **ET++ and the THINK C class library**
  - Director-like objects in dialogs as mediators between widgets
- **Application Architecture of Smalltalk/V for Windows**
  - An application consists of a Window containing a set of panes.
  - The library contains several predefined Pane objects.
- **Coordinating Complex Updates**
  - The ChangeManager class in Observer pattern to avoid redundant updates

Related Patterns

- **Façade**
  - Façade pattern differs from Mediator in that it abstracts a subsystem of objects to provide a more convenient interface.
  - Protocol of Façade is unidirectional.
- **Observer**
  - Colleagues can communicate with the mediator using the Observer pattern.

Memento

Unit 19

Intent

- To capture and externalize an object's internal state so that the object can be restored to this state later without violating encapsulation

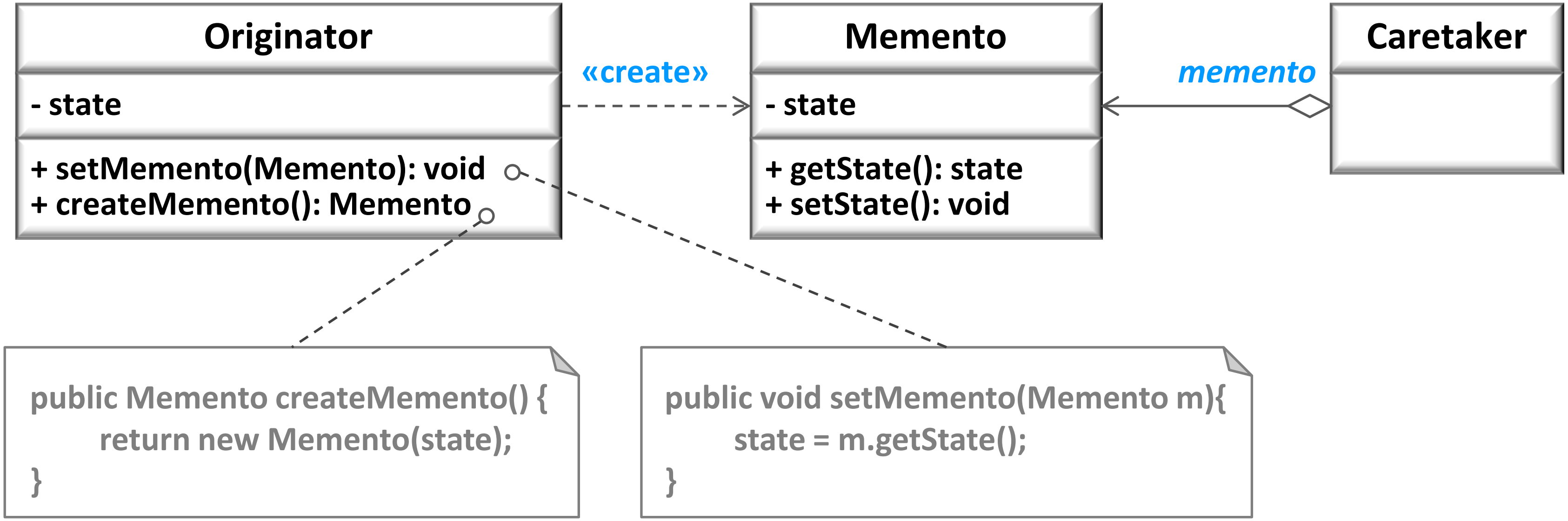
Motivation

- **To record the internal state of an object**
  - To save state information somewhere so that you can restore objects to their previous states
- **Objects normally encapsulate some or all of their state, making it inaccessible to other objects and impossible to save externally.**
- **Exposing this state would violate encapsulation, which can compromise the application's reliability and extensibility.**

Applicability

- **Applicable Situations:**
  - A snapshot of an object's state must be saved so that it can be restored to that state later.
  - A direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

Structure



Originator

- state

+ setMemento(Memento): void
+ createMemento(): Memento

«create»

Memento

- state

+ getState(): state
+ setState(): void

memento

Caretaker

```
public Memento createMemento() {
        return new Memento(state);
}
```

```
public void setMemento(Memento m){
        state = m.getState();
}
```

Participants (1)

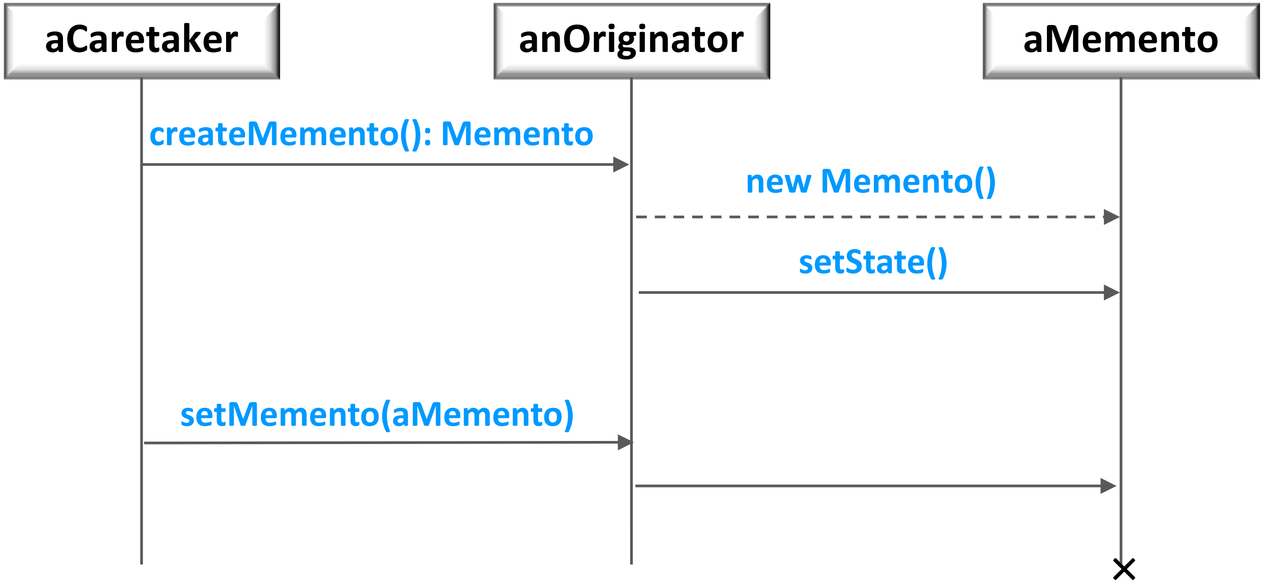
- **Memento (SolverState)**
  - To store internal state of the Originator object
    - The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
  - To protect against access by objects other than the originator. Mementos have effectively <u>two interfaces</u>;
    - Caretaker sees a *narrow* interface to the Memento.
      - it can only pass the memento to other objects.
    - Originator sees a *wide* interface, one that lets it access all the data necessary to restore itself to its previous state.
      - Ideally, only the originator that produced the memento would be permitted to access the memento's internal state.

Participants (2)

- **Originator (ConstraintSolver)**
  - To create a memento containing a snapshot of its current internal state
  - To use the memento to restore its internal state
- **Caretaker (undo mechanism)**
  - To be responsible for the memento's safekeeping
  - Never to operate on or examine the contents of a memento

Collaborations

- **A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator, as the following interaction diagram illustrates.**



- **Mementos are passive.**
  - Only the originator that created a memento will assign or retrieve its state.

Consequences

- To preserve encapsulation boundaries
- To simplify Originator
- Using mementos might be expensive.
- To define narrow and wide interfaces
- To hide costs in caring for mementos

Implementation

- **Two Implementation issues**
  - Language support
    - Mementos have two interfaces
      - A wide one for originators and a narrow one for caretakers.
    - Ideally the implementation language will support two levels of static protection.
  - Storing incremental changes
    - When mementos get created and passed back to their originator in a predictable sequence, then Memento can save just the *incremental change* to the originator's internal state.

Sample Code – ConstraintSolver (1)

```
class Graphic;
// base class for graphical objects in the graphical editor

class MoveCommand {
    public:
        MoveCommand(Graphic* target,
                    const Point& delta);
        void Execute();
        void Unexecute();
    private:
        ConstraintSolverMemento* _state;
        Point _delta;
        Graphic* _target;
};
```

```
class ConstraintSolver {
    public:
        static ConstraintSolver* Instance();
        void Solve();
        void AddConstraint(Graphic* startConnection,
                    Graphic* endConnection);
        void RemoveConstraint(Graphic* startConnection,
                    Graphic* endConnection);
        ConstraintSolverMemento* CreateMemento();
        void SetMemento(ConstraintSolverMemento*);

    private:
        // nontrivial state and operations for enforcing
        // connectivity semantics
};

class ConstraintSolverMemento {
    public:
        virtual ~ConstraintSolverMemento();

    private:
        friend class ConstraintSolver;
        ConstraintSolverMemento();
        // private constraint solver state
};
```

Sample Code – ConstraintSolver (2)

```
// Given these interfaces, we can implement MoveCommand membersExecute and Unexecute.

void MoveCommand::Execute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _state = solver->CreateMemento();
    // create a memento
    _target->Move(_delta);
    solver->Solve();
}

void MoveCommand::Unexecute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _target->Move(-_delta);
    solver->SetMemento(_state);
    // restore solver state
    solver->Solve();
}

// Execute acquires a ConstraintSolverMemento memento before it moves the graphic.
// Unexecute moves the graphic back, sets the constraint solver's state to the previous state, and finally tells the
constraint solver to solve the constraints.
```

Known Uses

- **CSolver class**
- **Collections in Dylan**
  - To provide an iteration interface that reflects the Memento pattern
  - “State” object, which is memento that represents the state of the iteration
  - Benefits of the memento-based iteration
    - More than one state can work on the same collection.
    - It doesn’t require breaking a collection’s encapsulation to support iteration.
- **QOCA constraint-solving toolkit**
  - To store incremental information in mementos

Related Patterns

- **Command**
  - Commands can use mementos to maintain state for undoable operations.
- **Iterator**
  - Mementos can be used for iteration as described earlier.

Observer

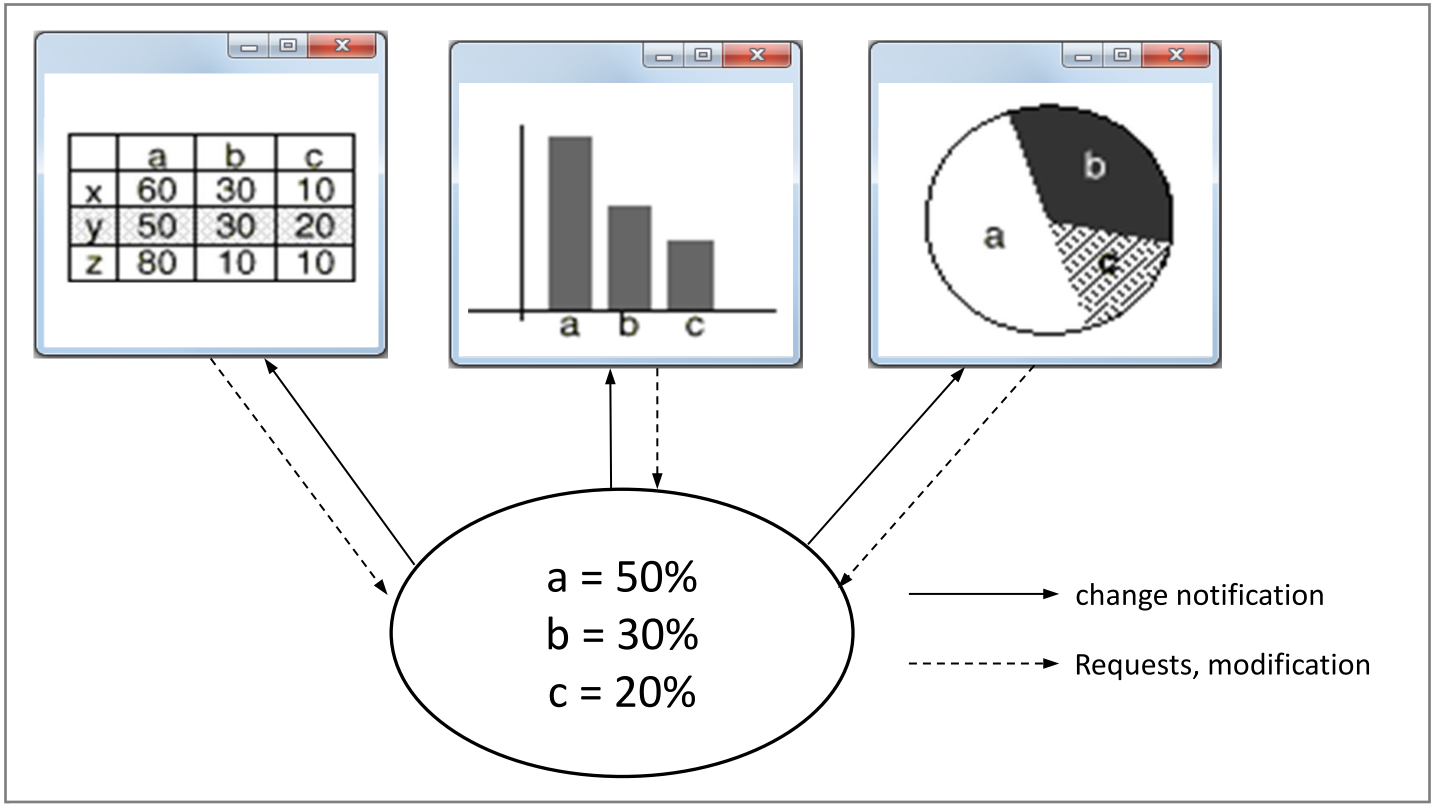
Unit 20

Intent

- To define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

Motivation

- **Consider graphical user interface toolkits which separates the presentational aspects of the user interface from the underlying application data**
  - The need to maintain consistency between related objects without making classes tightly coupled



	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10

a = 50%
b = 30%
c = 20%

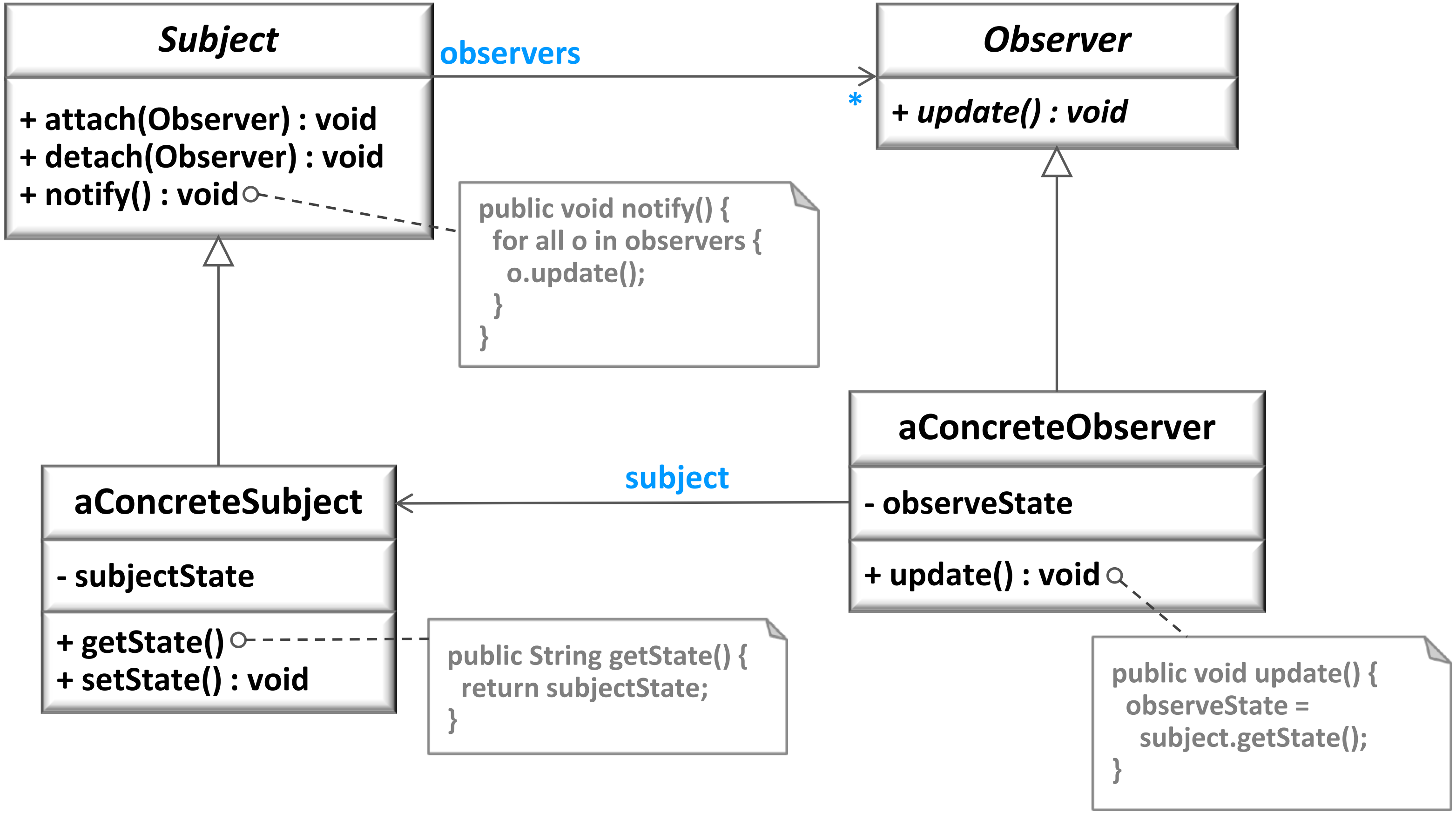
change notification

Requests, modification

Applicability

- **Use the Observer pattern in any of the following situations:**
  - When an abstraction has two aspects, one dependent on the other.
    - Encapsulating these aspects in separate objects lets you vary and reuse them independently.
  - When a change to one object requires changing others
  - When an object should be able to notify other objects without making assumptions about those objects

Structure



```
public void notify() {
  for all o in observers {
    o.update();
  }
}
```

```
public String getState() {
  return subjectState;
}
```

```
public void update() {
  observeState =
    subject.getState();
}
```

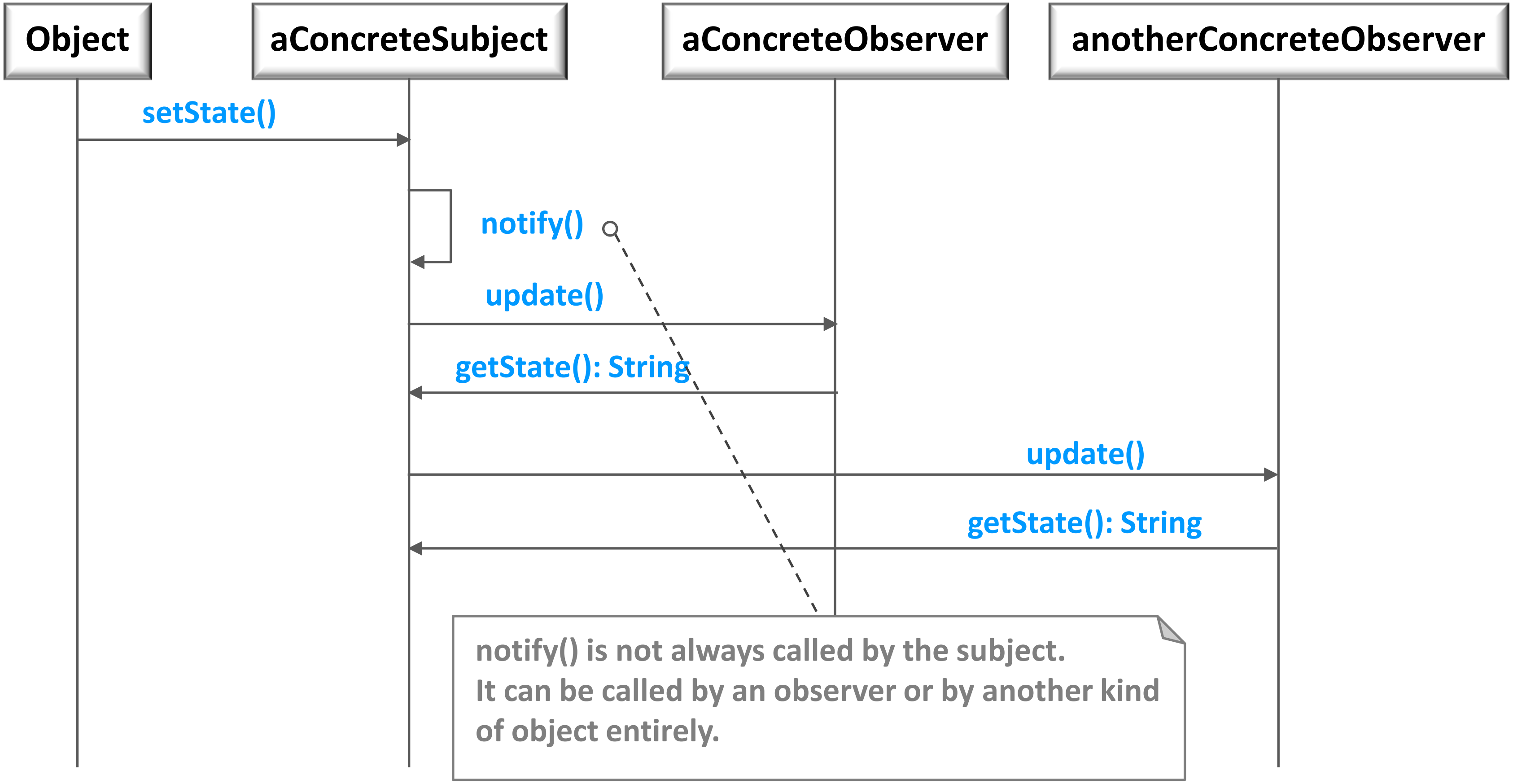
Participants

- **Subject**
  - To keep track of its observers
  - To provide an interface for attaching and detaching Observer objects
- **Observer**
  - To define an interface for update notification
- **ConcreteSubject**
  - The object being observed
  - To store state of interest to ConcreteObserver objects
  - To send a notification to its observers when its state changes
- **ConcreteObserver**
  - The observing object
  - To store state that should stay consistent with the subject's
  - To implement the Observer update interface to keep its state consistent with the subject's

Collaborations (1)

- **Notifying changes to observers**
  - ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
- **Reflecting the changes**
  - After being informed of a change in the concrete object, ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

Collaborations (2)



notify() is not always called by the subject.
It can be called by an observer or by another kind of object entirely.

Consequences

- **Minimal coupling between the Subject and the Observer**
  - Can reuse subjects without reusing their observers and vice versa
  - Observers can be added without modifying the subject
  - All subject knows is its list of observers
  - Subject does not need to know the concrete class of an observer, just that each observer implements the update interface
  - Subject and observer can belong to different abstraction layers
- **Support for event broadcasting**
  - Subject sends notification to all subscribed observers
  - Observers can be added/removed at any time

Consequences (2)

- **Disadvantages**
  - Possible cascading of notifications
    - Observers are not necessarily aware of each other and must be careful about triggering updates
  - Simple update interface requires observers to deduce changed item

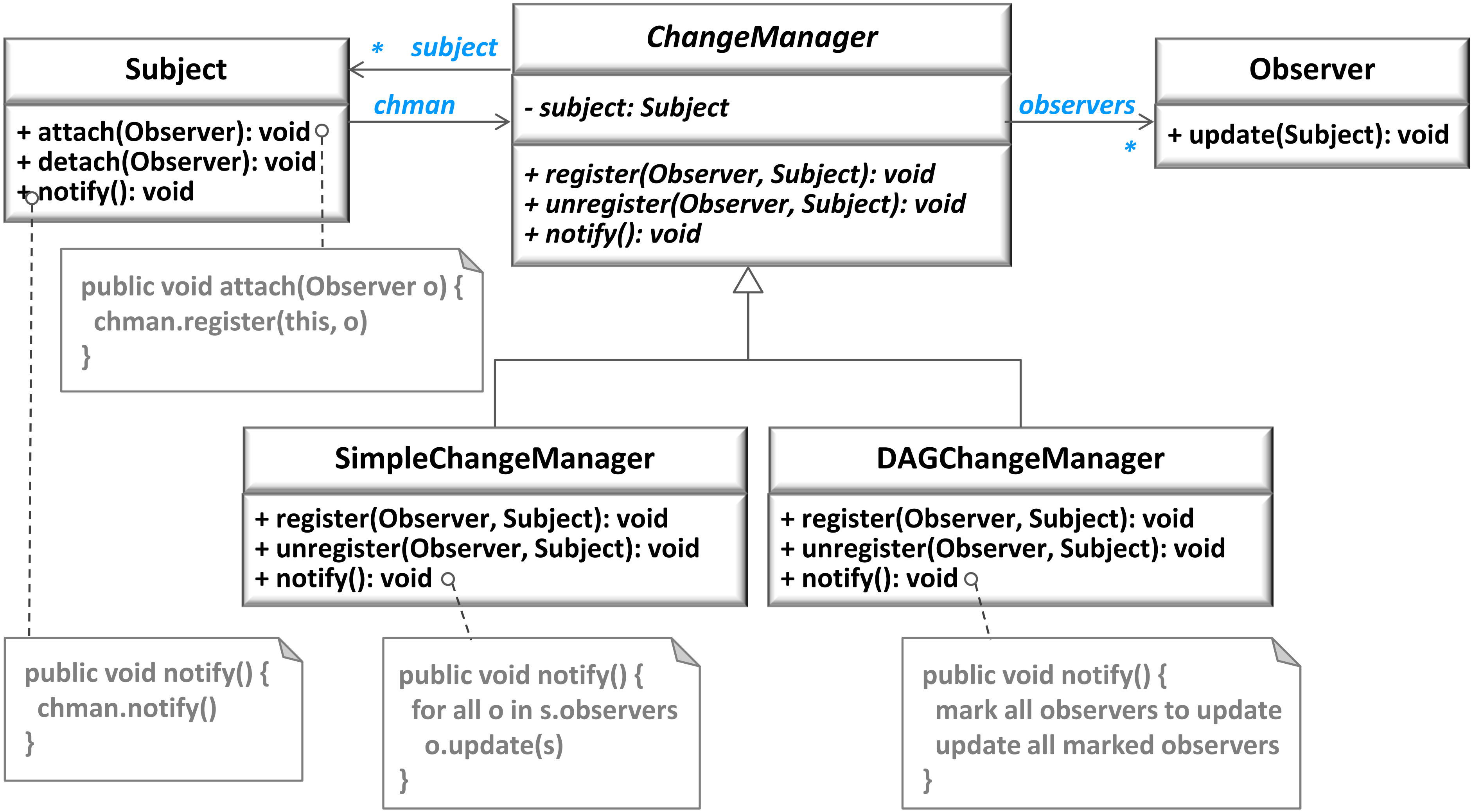
Implementation (1)

- **How does the subject keep track of its observers?**
  - Array, linked list
- **What if an observer wants to observe more than one subject?**
  - Have the subject tell the observer who it is via the update interface
- **Who triggers the update?**
  - The subject whenever its state changes
  - The observers after they cause one or more state changes
  - Some third party object(s)
- **Make sure the subject updates its state before sending out notifications**

Implementation (2)

- **How much info about the change should the subject send to the observers?**
  - Push Model - Lots
  - Pull Model - Very Little
- **Can the observers subscribe to specific events of interest?**
- **Can an observer also be a subject?**
  - If so, it's publish-subscribe
  - Yes!
- **What if an observer wants to be notified only after several subjects have changed state?**
  - Use an intermediary object which acts as a mediator
  - Subjects send notifications to the mediator object which performs any necessary processing before notifying the observers

Implementation (3)



Subject

+ attach(Observer): void
+ detach(Observer): void
+ notify(): void

* subject

chman

ChangeManager

- subject: Subject

+ register(Observer, Subject): void
+ unregister(Observer, Subject): void
+ notify(): void

observers

*

Observer

+ update(Subject): void

```
public void attach(Observer o) {
  chman.register(this, o)
}
```

SimpleChangeManager

+ register(Observer, Subject): void
+ unregister(Observer, Subject): void
+ notify(): void

DAGChangeManager

+ register(Observer, Subject): void
+ unregister(Observer, Subject): void
+ notify(): void

```
public void notify() {
  chman.notify()
}
```

```
public void notify() {
  for all o in s.observers
    o.update(s)
}
```

```
public void notify() {
  mark all observers to update
  update all marked observers
}
```

Sample Code (1)

```
class Subject;


  class Observer {
  public:
     virtual ~ Observer();
     virtual void Update(Subject*
   theChangedSubject) = 0;
  protected:
     Observer();
  };
```

```
class Subject {
   public:
      virtual ~Subject();

      virtual void Attach(Observer*);
      virtual void Detach(Observer*);
      virtual void Notify();
   protected:
      Subject();
   private:
      List<Observer*> *_observers;
   };
```

Sample Code (2)

```
class ClockTimer : public Subject {
  public:
     ClockTimer();

     virtual int GetHour();
     virtual int GetMinute();
     virtual int GetSecond();

     void Tick();
  };
```

```
class DigitalClock: public Widget, public Observer
    {
  public:
     DigitalClock(ClockTimer*);
     virtual ~DigitalClock();

     virtual void Update(Subject*);
        // overrides Observer operation

     virtual void Draw();
        // overrides Widget operation;
        // defines how to draw the digital clock
  private:
     ClockTimer* _subject;
  };
```

Known Uses

- **Smalltalk Model/View/Controller user interface framework**
  - Model = Subject
  - View = Observer
  - Controller is whatever object changes the state of the subject
- **Java 1.1 AWT/Swing Event Model**

Related Patterns

- **Mediator**
  - To encapsulate complex update semantics

State

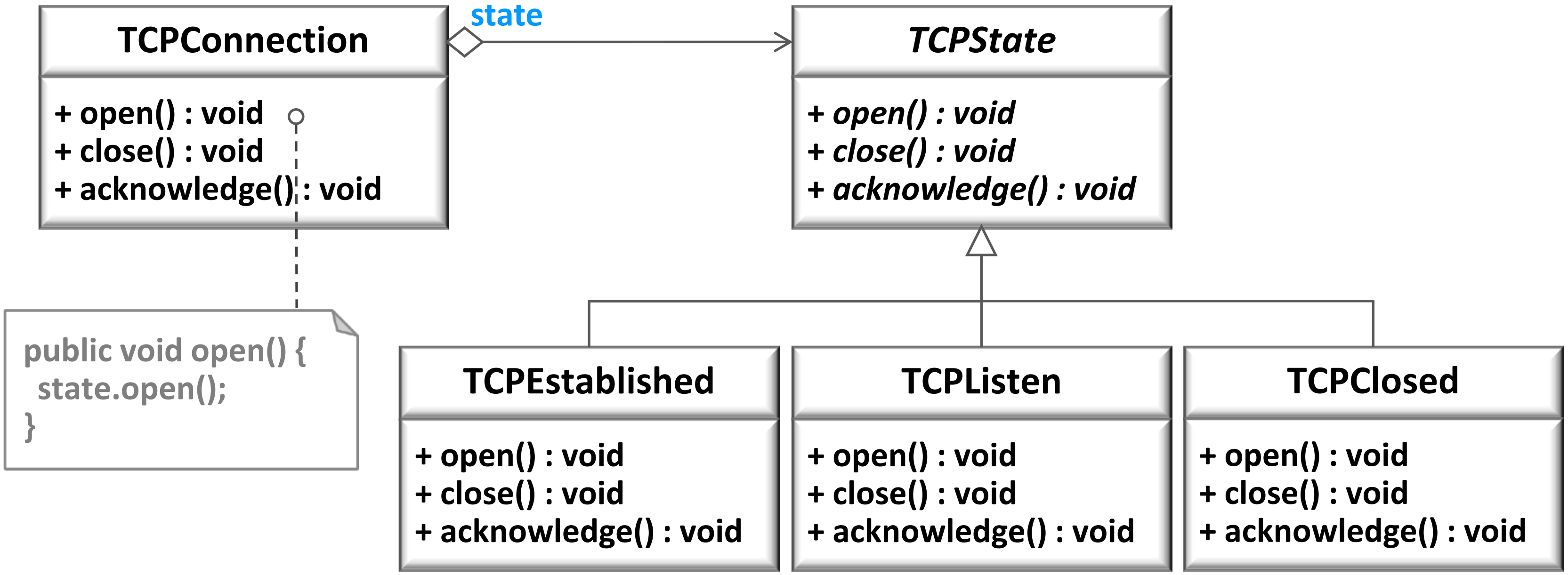
Unit 21

Intent

- Allow an object to alter its behavior when its internal state changes.
- The object will appear to change its class.

Motivation

- A TCPConnection object receives requests from other objects, it responds differently depending on its current state.

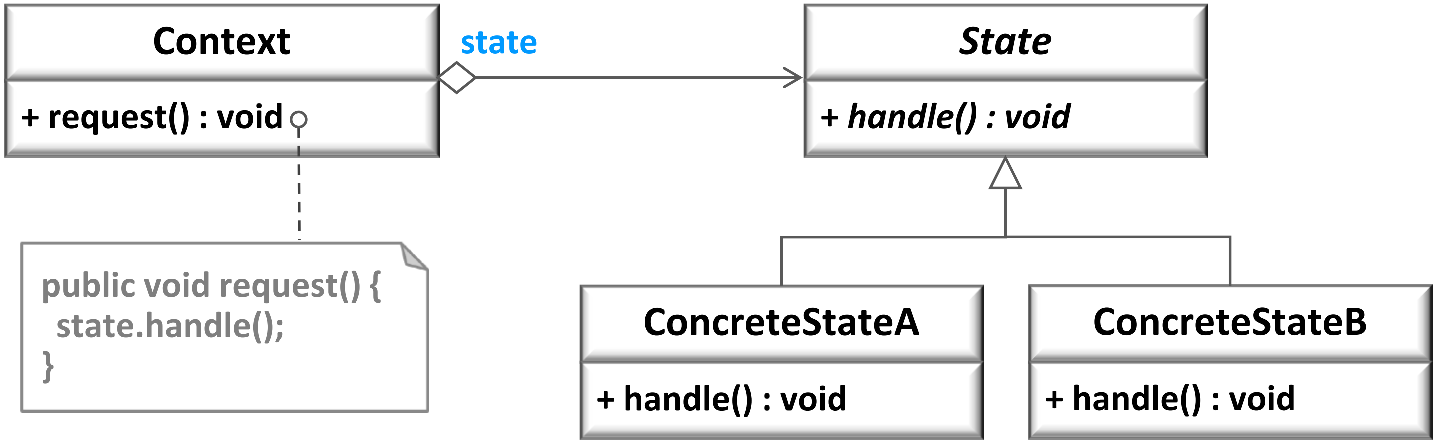


```
public void open() {
  state.open();
}
```

Applicability

- **Use the State pattern whenever:**
  - An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
  - Operations have large, multipart conditional statements that depend on the object's state.
    - The State pattern puts each branch of the conditional in a separate class.

Structure



Context

+ request() : void

state

State

+ handle() : void

```
public void request() {
  state.handle();
}
```

ConcreteStateA

+ handle() : void

ConcreteStateB

+ handle() : void

Participants

- **Context (TCPConnection)**
  - To define the interface of interest to clients
  - To maintain an instance of a ConcreteState subclass that defines the current state
- **State (TCPState)**
  - To define an interface for encapsulating the behavior associated with a particular state of the Context
- **ConcreteState subclasses (TCPEstablished, TCPListen, TCPClosed)**
  - Each subclass implements a behavior associated with a state of the Context.

Collaborations

- **Context delegates state-specific requests to the current ConcreteState object.**
- **A context may pass itself as an argument to the State object handling the request.**
  - This lets the State object access the context if necessary.
- **Context is the primary interface for clients. Clients can configure a context with State objects. Once a context is configured, its clients don't have to deal with the State objects directly.**
- **Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.**

Consequences

- **Advantages**
  - Puts all behavior associated with a state into one object
  - Allows state transition logic to be incorporated into a state object rather than in a monolithic if or switch statement
  - Helps avoid inconsistent states since state changes occur using just the one state object and not several objects or attributes
- **Disadvantages**
  - Increased number of objects

Implementation

- **Who defines the state transitions?**
  - The Context class
    - OK for simple situations
  - The ConcreteState classes
    - Generally more flexible, but causes implementation dependencies between the ConcreteState classes
- **Can't we just use a state-transition table for all this?**
  - Harder to understand
  - Difficult to add other actions and behavior
- **When are the ConcreteState objects created and destroyed?**
  - Create ConcreteState objects as needed and destroy thereafter.
  - Create all ConcreteState objects once and have the Context object keep references to them and never destroy them.

Sample Code (1)

```
class TCPOctetStream;
 class TCPState;

 class TCPConnection {
 public:
   TCPConnection();
   void ActiveOpen();
   void PassiveOpen();
   void Close();
   void Send();
   void Acknowledge();
   void Synchronize();
     void ProcessOctet(TCPOctetStream*);
```

```
private:
    friend class TCPState;
    void ChangeState(TCPState*);
  private:
    TCPState* _state;
  };
```

Sample Code (2)

```
class TCPState {
  public:
     virtual void Transmit(TCPConnection*,
   TCPOctetStream*);
     virtual void ActiveOpen(TCPConnection*);
     virtual void PassiveOpen(TCPConnection*);
     virtual void Close(TCPConnection*);
     virtual void Synchronize(TCPConnection*);
     virtual void Acknowledge(TCPConnection*);
     virtual void Send(TCPConnection*);
  protected:
     void ChangeState(TCPConnection*,
   TCPState*);
  };
```

```
class TCPEstablished : public TCPState {
   public:
      static TCPState* Instance();

      virtual void Transmit(TCPConnection*,
    TCPOctetStream*);
      virtual void Close(TCPConnection*);
   };

   class TCPListen : public TCPState {
   public:
      static TCPState* Instance();

      virtual void Send(TCPConnection*);
      // ...
   };
```

Known Uses

- Johnson and Zweig characterize the State pattern and its application to TCP connection protocols.
- HotDraw and Unidraw drawing editor frameworks
- Coplien's Envelope-Letter idiom

Related Patterns

- The Flyweight pattern explains when and how State objects can be shared.
- State objects are often Singletons.

Strategy

Unit 22

Intent

- Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it.

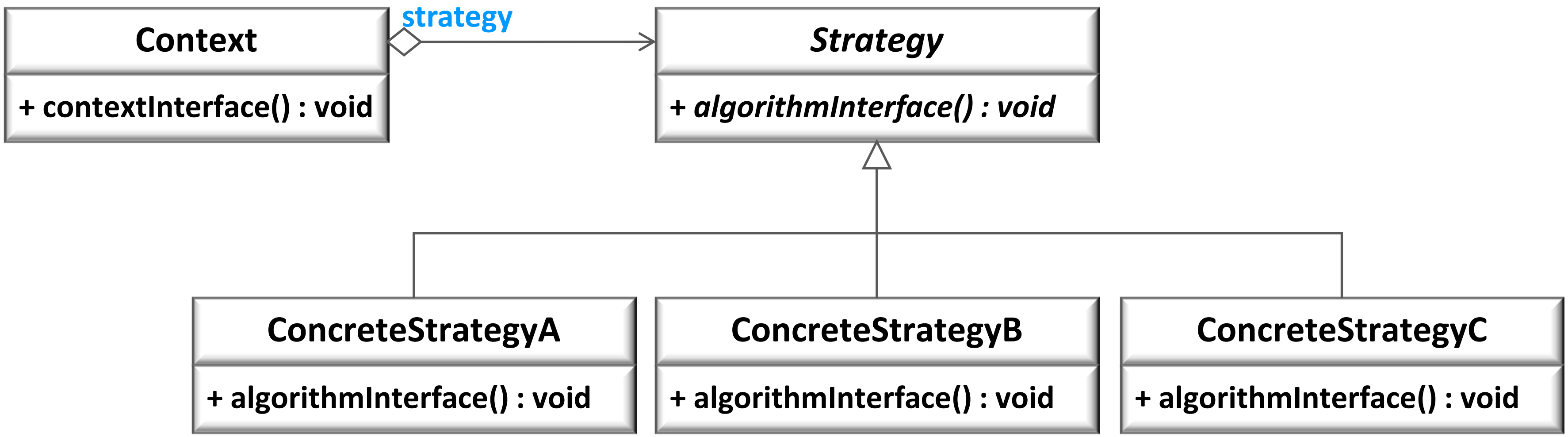
Motivation

- **Typical Problems of Algorithms for Breaking a stream of Text into Lines**
  - To get more complex if client applications include the linebreaking code
    - That makes clients bigger and harder to maintain, especially if they support multiple linebreaking algorithms.
  - Not easy to use different algorithms at different times
  - Difficult to add new algorithms and vary existing ones

Applicability

- **Use the Strategy pattern when:**
  - To make many related classes differ only in their behavior
  - To need different variants of an algorithm
  - Not to expose data used by the algorithm to clients
    - Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
  - To define many behaviors which appear as multiple conditional statements in its operations
    - Instead of many conditionals, move related conditional branches into their own Strategy class.

Structure


strategy
Context
+ contextInterface() : void
Strategy
+ algorithmInterface() : void
ConcreteStrategyA
+ algorithmInterface() : void
ConcreteStrategyB
+ algorithmInterface() : void
ConcreteStrategyC
+ algorithmInterface() : void

Participants

- **Strategy (Compositor)**
  - To declare an interface common to all supported algorithms
  - Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- **ConcreteStrategy (SimpleCompositor, TeXCompositor, ArrayCompositor)**
  - To implement the algorithm using the Strategy interface
- **Context (Composition)**
  - To be configured with a ConcreteStrategy object
  - To maintain a reference to a Strategy object
  - May define an interface that lets Strategy access its data.

Collaborations

- **Strategy and Context interact to implement the chosen algorithm.**
  - A context may pass all data required by the algorithm to the strategy when the algorithm is called.
  - Alternatively, the context can pass itself as an argument to Strategy operations.
  - That lets the strategy call back on the context as required.
- **A context forwards requests from its clients to its strategy.**
  - Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively.
  - There is often a family of ConcreteStrategy classes for a client to choose from.

Consequences

- **Advantages**
  - To provide an alternative to subclassing the Context class to get a variety of algorithms or behaviors
  - To eliminate large conditional statements
  - To provide a choice of implementations for the same behavior
- **Disadvantages**
  - To increase the number of objects
  - All algorithms must use the same Strategy interface.

Implementation

- **Defining the Strategy and Context interfaces.**
  - How to give ConcreteStrategy an efficient access to any data it needs from a context, and vice versa?
    - To have Context pass data in parameters to Strategy operations
    - To have a context pass itself as an argument and to let the strategy request data from the context explicitly
- **Strategies as template parameters**
  - Applicable if (1) the Strategy can be selected at compile-time, and (2) it does not have to be changed at run-time
  - No need to define an abstract class that defines the interface to the Strategy
- **Making Strategy objects optional.**
  - Context checks to see if it has a Strategy object before accessing it.
    - If there is one, then Context uses it normally.
    - If there isn't a strategy, then Context carries out default behavior.

Sample Code (1)

```
class Composition {
 public:
   Composition(Compositor*);
   void Repair();
 private:
   Compositor* _compositor;
   Component* _components;    // the list of components
   int _componentCount;       // the number of components
   int _lineWidth;            // the Composition's line width
   int* _lineBreaks;          // the position of linebreaks
                       // in components
   int _lineCount;            // the number of lines
 };
```

Sample Code (2)

```
class Compositor {
  public:
    virtual int Compose(
        Coord natural[], Coord stretch[], Coord
   shrink[],
        int componentCount, int lineWidth, int
   breaks[]
    ) = 0;
  protected:
    Compositor();
  };
```

```
class SimpleCompositor : public Compositor {
    public:
        SimpleCompositor();


        virtual int Compose(
            Coord natural[], Coord stretch[], Coord
      shrink[],
            int componentCount, int lineWidth, int
      breaks[]
        );
        // ...
    };


Composition* quick = new Composition(new
      SimpleCompositor);
```

Known Uses

- **ET++ and InterViews**
  - To encapsulate different linebreaking algorithms as we've described
- **In the RTL System for compiler code optimization**
- **The ET++SwapsManager calculation engine framework computes prices for different financial instruments**
- **The Booch components**
- **RApp is a system for integrated circuit layout**
- **Borland's ObjectWindows**

Related Patterns

- **Flyweight**
  - Strategy objects often make good flyweights.

Template Method

Unit 23

Intent

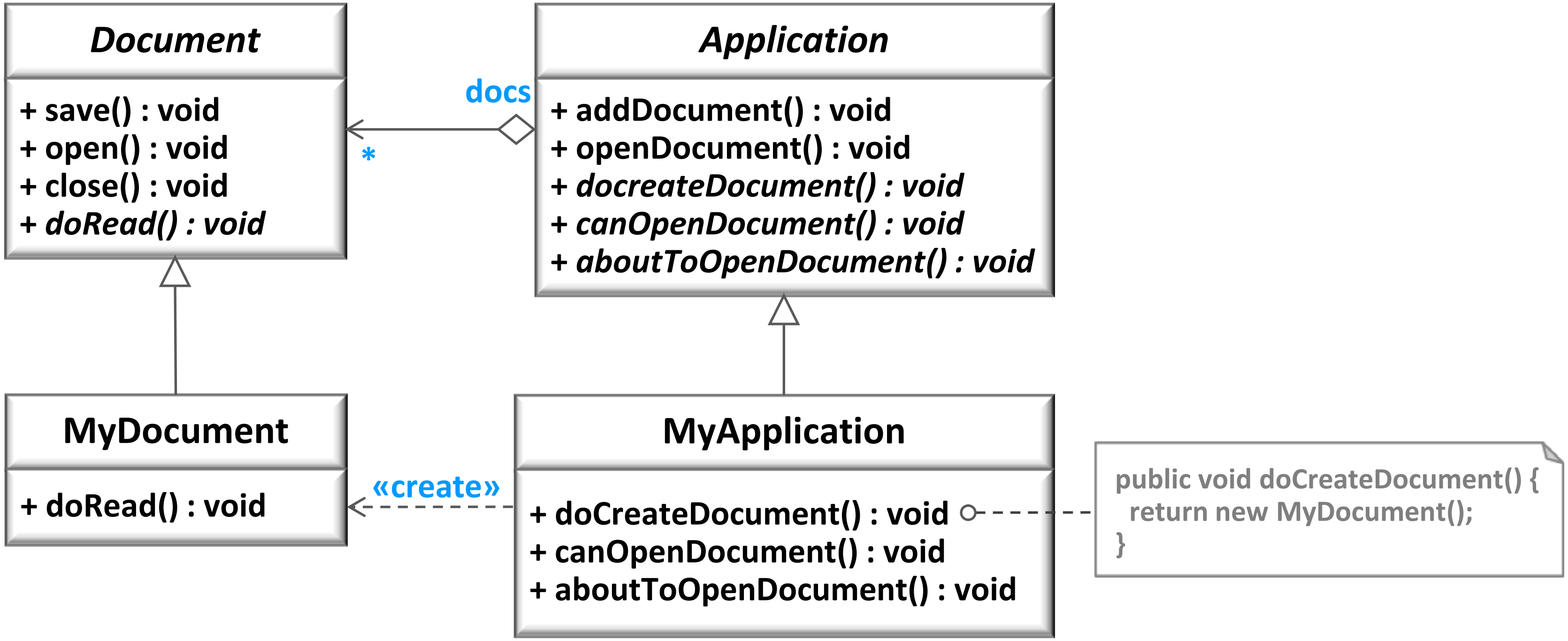
- **Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.**
- **Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.**

Motivation (1)

- **Consider this situation**
  - 1) you want to specify (or fix) the order of operations that a method uses.
  - 2) but, you want to vary some of these operations to meet subclasses' needs.

Motivation (2)

- **Consider an application framework that provides Application and Document classes:**
  - Application class for opening existing documents
  - Document class for representing the information in a document



```
public void doCreateDocument() {
  return new MyDocument();
}
```

Motivation (3)

- **The OpenDocument() method might look like this:**

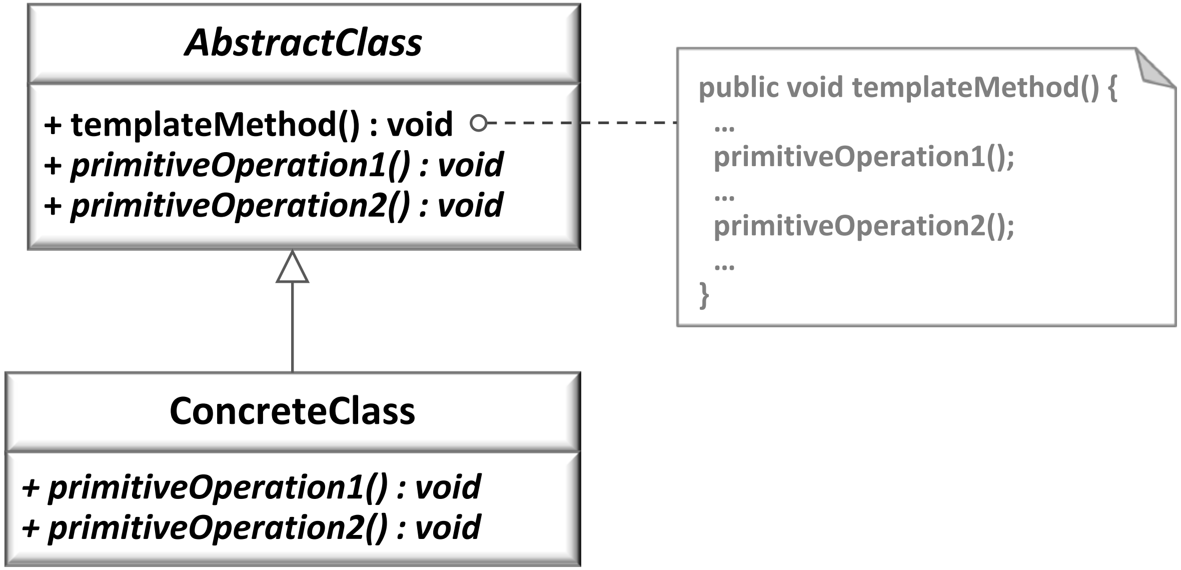
```
public void OpenDocument (String name) {
        if (!CanOpenDocument(name)) { return; }
        Document doc = DoCreateDocument();
        if (doc != null) {
                docs.AddDocument(doc);
                AboutToOpenDocument(doc);
                doc.Open();
                doc.DoRead();
        }
  }
```

- **The OpenDocument() method is a Template Method.**
  - The template method fixes the order of operations, but allows Application subclasses to vary those steps as needed.

Applicability

- **Use the Template Method pattern when:**
  - To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary
  - To localize common behavior among subclasses and place it in a common class (in this case, a superclass) to avoid code duplication
  - To control how subclasses extend superclass operations.
    - You can define a template method that calls "hook" operations at specific points, thereby permitting extensions only at those points.
- **The Template Method is a fundamental technique for code reuse.**

Structure


AbstractClass
+ templateMethod() : void
+ *primitiveOperation1() : void*
+ *primitiveOperation2() : void*
ConcreteClass
+ *primitiveOperation1() : void*
+ *primitiveOperation2() : void*

```
public void templateMethod() {
  …
  primitiveOperation1();
  …
  primitiveOperation2();
  …
}
```

Participants

- **AbstractClass (Application)**
  - To define abstract primitive operations that concrete subclasses define to implement steps of an algorithm
  - To implement a template method defining the skeleton of an algorithm
    - The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
- **ConcreteClass (MyApplication)**
  - To implement the primitive operations to carry out subclass-specific steps of the algorithm

Collaborations

- ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm.

Consequences

- **A fundamental technique for code reuse**
  - Particularly important in class libraries, because template methods are the means for factoring out common behavior in library classes.
- **It's important for template methods to specify which operations are hooks (may be overridden) and which are abstract operations (must be overridden).**

Implementation

- **Using Access Control**
  - Declare primate operations, which a template method calls, as protected, so that they are only called by the template method.
  - Declare primitive operations that *must be overridden* as abstract.
  - Declare a template method as final, so that it should not be overridden by subclasses.
- **Minimizing Primitive Operations**
  - Try to minimize the number of operations that a subclass must override, otherwise using the template method becomes tedious for the developer.
- **Naming Conventions**
  - Identify the operations that should be overridden by adding a prefix to their names, such as “do-”.

Sample Code

```
 void View::Display () {
       SetFocus();
       DoDisplay();
       ResetFocus();
   }

void View::DoDisplay () { }

void MyView::DoDisplay () {
       // render the view's contents
   }
```

Known Uses

- Template methods are so fundamental that they can be found in almost every abstract class.
- Wirfs-Brock et al. provide a good overview and discussion of template methods.

Related Patterns

- **Factory Methods are often called by template methods.**
  - In the Motivation example, the factory method DoCreateDocument is called by the template method OpenDocument.
- **Strategy**
  - Template methods use inheritance to vary part of an algorithm. Strategies use delegation to vary the entire algorithm.

Visitor

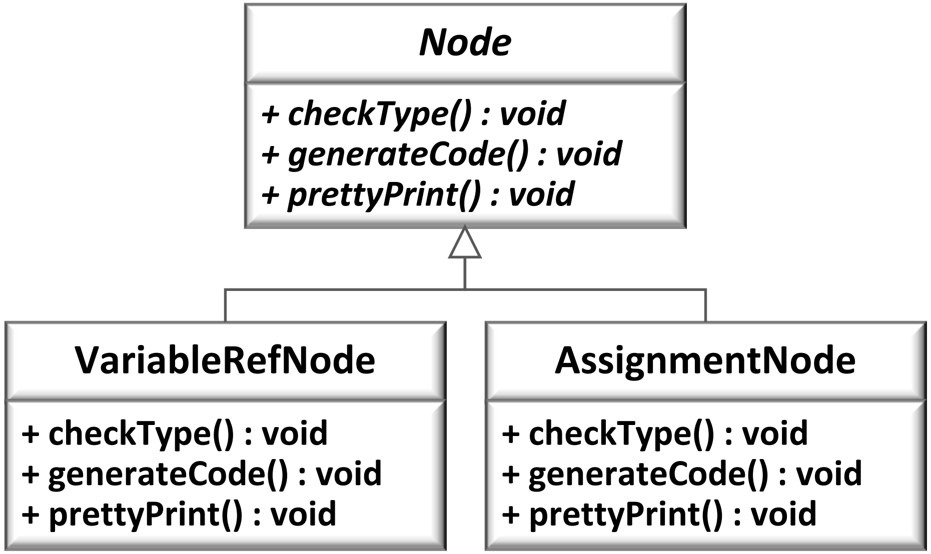
Unit 24

Intent

- To represent an operation to be performed on the elements of an object structure.
- To define a new operation without changing the classes of the elements on which it operates

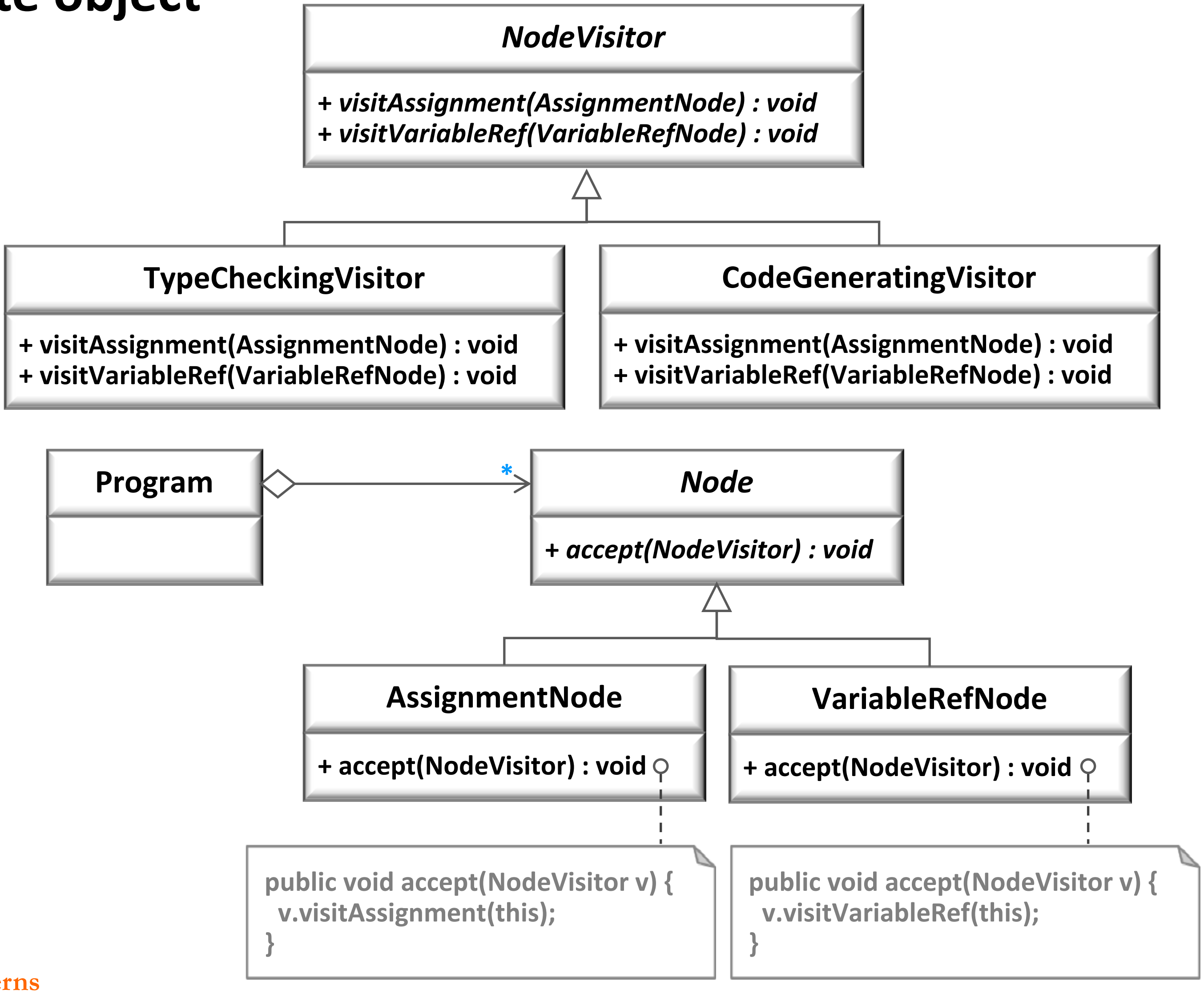
Motivation (1)

- **Consider a compiler that represents programs as abstract syntax trees**
  - To need the operations to treat diverse types of nodes in different ways
    - E.g.) operations for type-checking, code optimization, flow analysis, checking
  - The set of node classes depends on the language being compiled, but it doesn't change much for a given language.



Motivation (2)

- Solved by packaging related operations from each class in a separate object



NodeVisitor

+ visitAssignment(AssignmentNode) : void
+ visitVariableRef(VariableRefNode) : void

TypeCheckingVisitor

+ visitAssignment(AssignmentNode) : void
+ visitVariableRef(VariableRefNode) : void

CodeGeneratingVisitor

+ visitAssignment(AssignmentNode) : void
+ visitVariableRef(VariableRefNode) : void

Program

Node

+ accept(NodeVisitor) : void

AssignmentNode

+ accept(NodeVisitor) : void

VariableRefNode

+ accept(NodeVisitor) : void

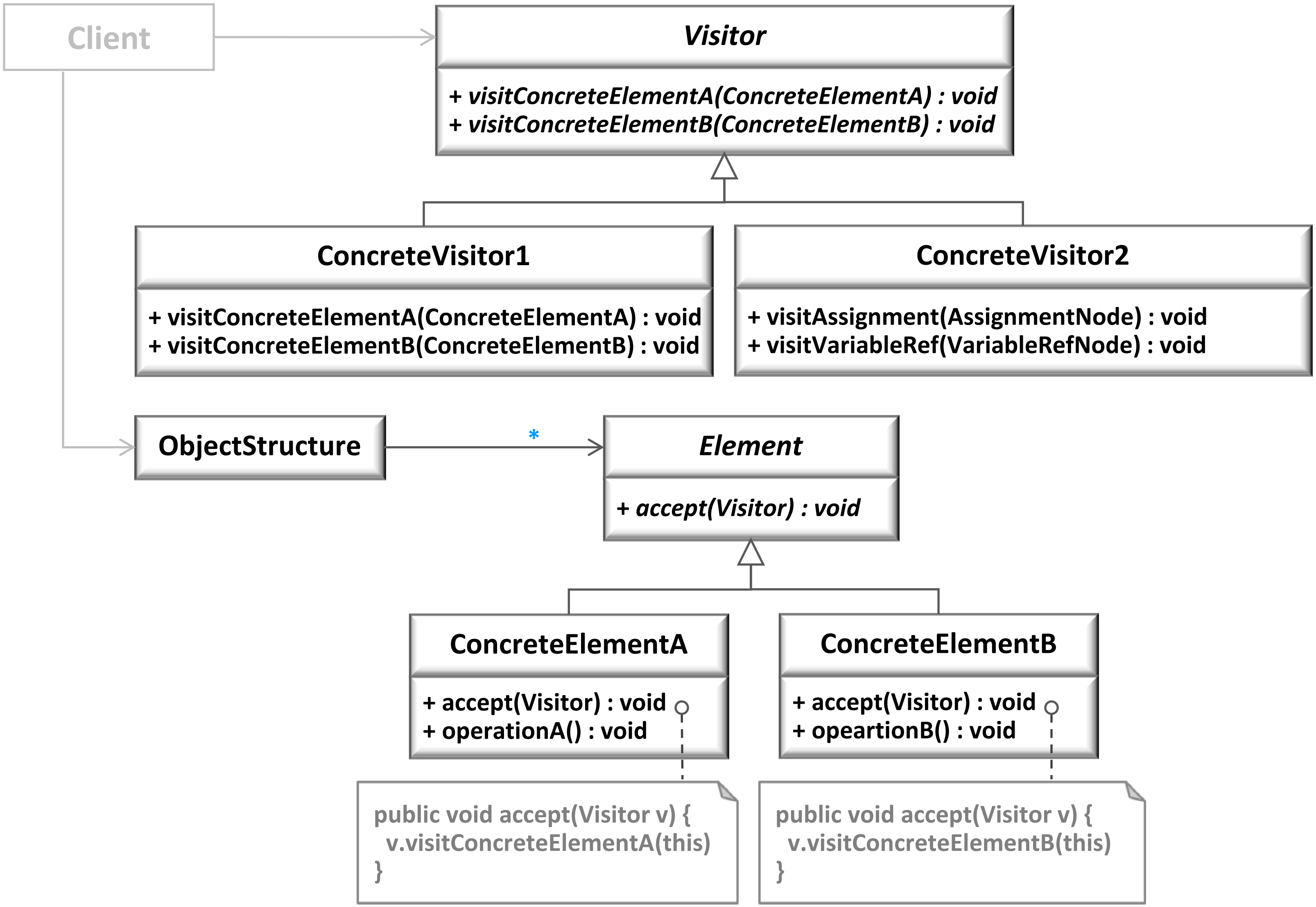
```
public void accept(NodeVisitor v) {
  v.visitAssignment(this);
}
```

```
public void accept(NodeVisitor v) {
  v.visitVariableRef(this);
}
```

Applicability

- **Use the visitor pattern when:**
  - You want to perform operations on many objects in an object structure which depend on their concrete classes.
  - You need to perform many distinct and unrelated operations on objects in an object structure, without polluting their classes with these operations
    - Visitor lets you keep related operations together by defining them in one class.
  - The classes defining the object structure rarely change, but you often want to define new operations over the structure.
    - Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly.

Structure



Client

Visitor

+ *visitConcreteElementA(ConcreteElementA) : void*
+ *visitConcreteElementB(ConcreteElementB) : void*

ConcreteVisitor1

+ visitConcreteElementA(ConcreteElementA) : void
+ visitConcreteElementB(ConcreteElementB) : void

ConcreteVisitor2

+ visitAssignment(AssignmentNode) : void
+ visitVariableRef(VariableRefNode) : void

ObjectStructure

*

Element

+ *accept(Visitor) : void*

ConcreteElementA

+ accept(Visitor) : void
+ operationA() : void

ConcreteElementB

+ accept(Visitor) : void
+ opeartionB() : void

```
public void accept(Visitor v) {
  v.visitConcreteElementA(this)
}
```

```
public void accept(Visitor v) {
  v.visitConcreteElementB(this)
}
```

Participants (1)

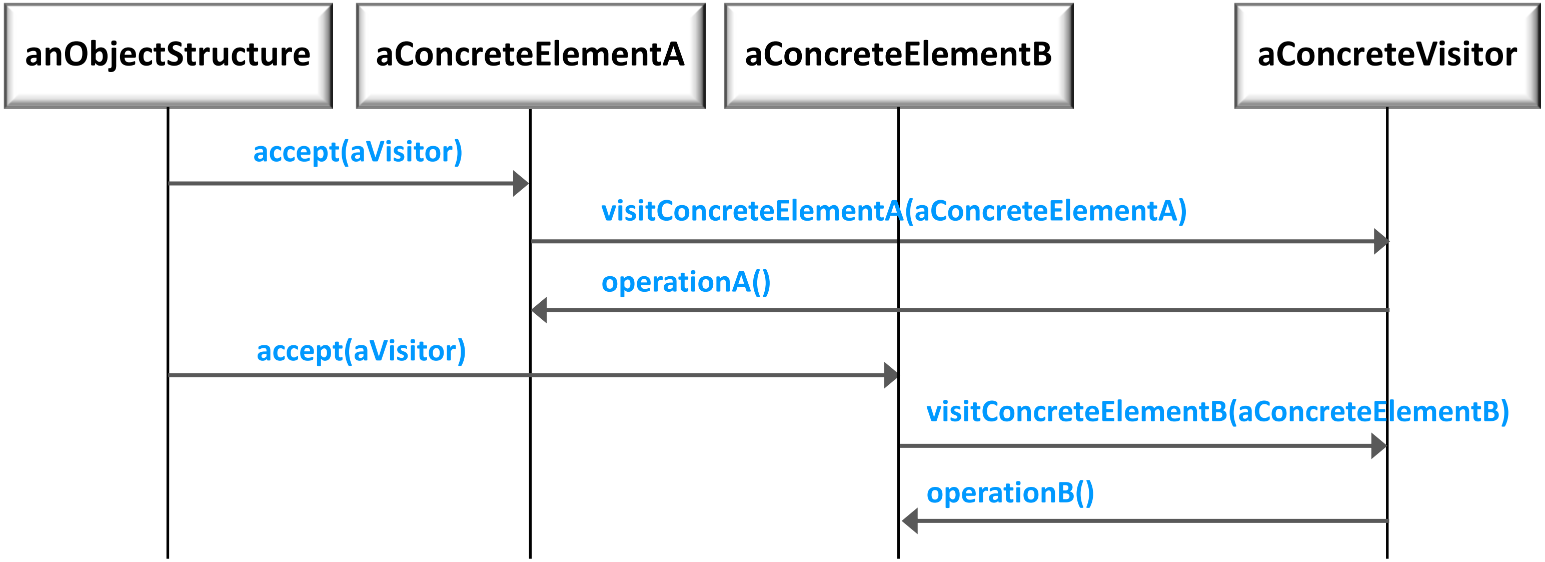
- **Visitor (NodeVisitor)**
  - To declare a Visit operation for each class of ConcreteElement in the object structure
  - Roles of the Visit operation
    - To let the visitor determine the concrete class of the element being visited
    - To enable to access the element directly through its particular interface
- **ConcreteVisitor (TypeCheckingVisitor)**
  - To implement each operation declared by Visitor
  - Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure.
  - To provide the context for the algorithm
  - To store its local state which often accumulates results during the traversal of the structure

Participants (2)

- **Element (Node)**
  - To define an Accept operation that takes a visitor as an argument
- **ConcreteElement (AssignmentNode,VariableRefNode)**
  - To implement an Accept operation that takes a visitor as an argument
- **ObjectStructure (Program)**
  - To be able to enumerate its elements
  - May provide a high-level interface to allow the visitor to visit its elements.
  - May either be a composite or a collection such as a list or a set

Collaborations

- **A client must create a ConcreteVisitor object and then traverse the object structure, visiting each element with the visitor.**
- **When an element is visited, it calls the Visitor operation that corresponds to its class.**
    - The element supplies itself as an argument to this operation to let the visitor access its state, if necessary.



Consequences

- **Advantages**
  - Easy to add new operations
  - Possible to gather related operations and separate unrelated ones
  - Visiting across class hierarchies
    - To be able to visit objects that don’t have a common parent class
    - To be able to add any type of object to a Visitor interface
  - Possible to accumulate state
    - Without a visitor, the state would be passed as extra arguments to the operations, or they might appear as global variables.
- **Disadvantages**
  - Hard to add new ConcreteElement classes
  - Breaking encapsulation
    - To enforce to provide public operations that access an element’s internal states

Implementation

- **Double Dispatch**
  - The operation that gets executed depends on the kind of request and the types of *two* receivers.
  - Double-dispatch operation in Visitor: accept()
    - The operation that gets executed depends on both the type of Visitor and the type of Element it visits.
    - To dynamically bind operations at runtime
  - To support adding operations to classes without changing them
- **Who is responsible for traversing the object structure?**
  - Possible to put responsibility for traversal in any of three places
    - In the object structure,
    - In the visitor,
    - Or, in a separate iterator object

Sample Code (1)

```
class Equipment {
  public:
    virtual ~Equipment();
    const char* Name() { return _name; }
    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Accept(EquipmentVisitor&);
  protected:
    Equipment(const char*);
  private:
    const char* _name;
  };
```

```
class EquipmentVisitor {
  public:
    virtual ~EquipmentVisitor();

    virtual void VisitFloppyDisk(FloppyDisk*);
    virtual void VisitCard(Card*);
    virtual void VisitChassis(Chassis*);
    virtual void VisitBus(Bus*);

    // and so on for other concrete subclasses of
   Equipment
  protected:
    EquipmentVisitor();
  };
```

Sample Code (2)

```
void FloppyDisk::Accept (EquipmentVisitor& visitor) {
      visitor.VisitFloppyDisk(this);
   }
void Chassis::Accept (EquipmentVisitor& visitor) {
       for (
          ListIterator i(_parts);
          !i.IsDone();
          i.Next()
       ) {
          i.CurrentItem()->Accept(visitor);
       }
       visitor.VisitChassis(this);
   }
```

Known Uses

- **Smalltalk-80 compiler**
  - ProgramNodeEnumerator
- **IRIS Inventor**
- **X Consortium's Fresco Application Toolkit specification**

Related Patterns

- **Composite**
  - Visitors can be used to apply an operation over an object structure defined by the Composite pattern.
- **Interpreter**
  - Visitor may be applied to do the interpretation.

Hardware Proxy Pattern

Unit 25

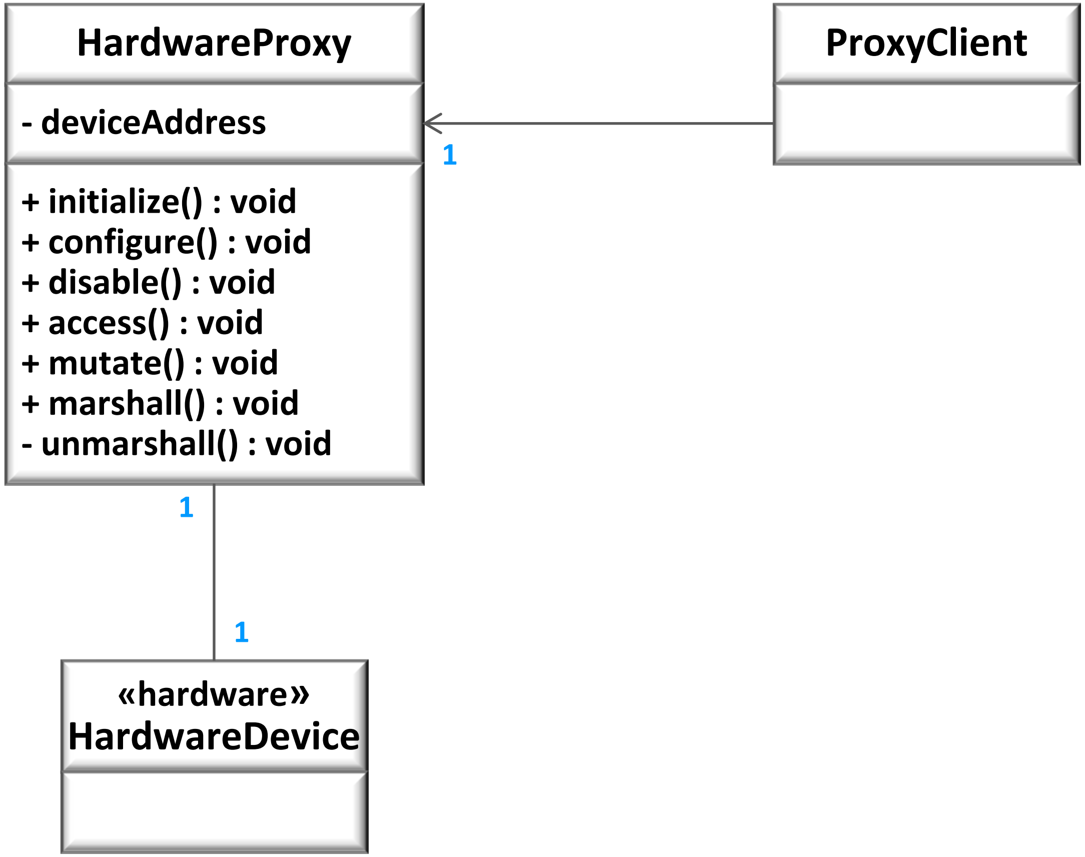
Overview

- **To use a class (or struct) to encapsulate all access to a hardware device, regardless of its physical interface**
  - Hardware may be memory, port, or interrupt mapped, or may even be mapped via a serial connection, bus, network, or wireless link.
- **Roles of Proxies**
  - To publish services that allow values to be read from and written to the device
  - To initialize, configure, and shut down the device as appropriate
  - To provide an encoding and connection-independent interface for clients

Applicable Situations

- **Used to eliminate side effects of changing bit encoding, memory address, encryption, and compression used by a hardware device**
  - Let clients be unaware of those hardware-specific access information

Structure (1)


HardwareProxy
- deviceAddress
+ initialize() : void
+ configure() : void
+ disable() : void
+ access() : void
+ mutate() : void
+ marshall() : void
- unmarshall() : void
ProxyClient
1
1
1
«hardware»
HardwareDevice

Structure (2)

- **HadwareDevice**
  - To represent the actual hardware
- **ProxyClient**
  - To invoke its services to access the hardware device
- **HardwareProxy**
  - To enable and initialize the device prior to first use (via initialize())
  - To provide a means by which the device may be configured (via configure())
  - To provide a means by which the device may be safely turned off or disabled (via disable())
  - To return a particular value from the device usually after calling unmarshall() (via access())
  - To write data values to the device after calling marshall() (via mutate())
  - To take parameters from various other functions and perform any required encryption, compression, or bit-packing required to send the data to the device (via marshall())
  - To perform any necessary unpacking, decryption, and decompression of the data retrieved from the device prior to returning them to the client in presentation format (via unmarshall())

Consequences

- **Advantages**
  - To benefit from encapsulation of the hardware interface and encoding details
  - To provide flexibility for the actual hardware interface to change radically with absolutely no changes in the clients
- **Disadvantages**
  - Negative impact on run-time performance

Polling Pattern

Unit 26

Overview

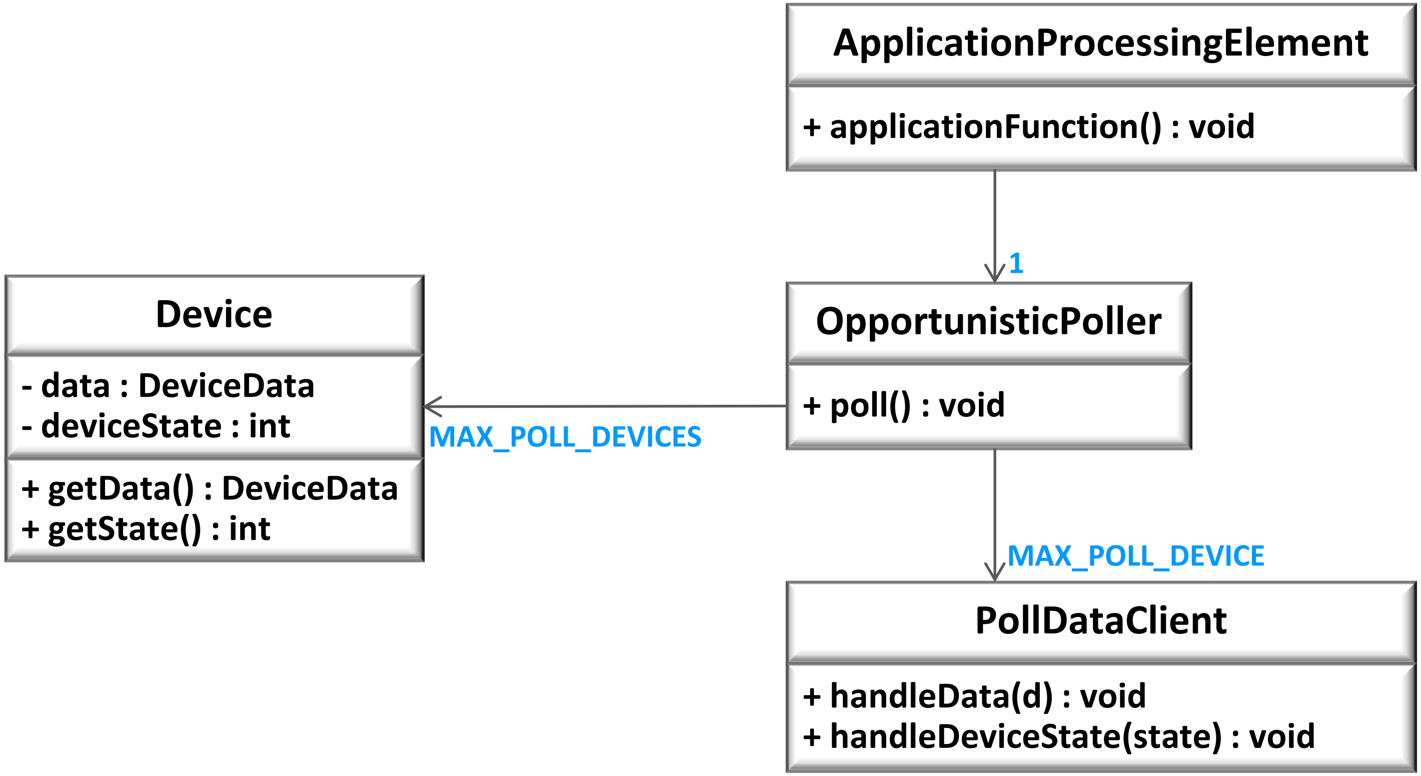
- **The simplest way to check for new data or signals from the hardware**
- **Polling can be periodic or opportunistic.**
  - Periodic polling uses a timer to indicate when the hardware should be sampled.
  - Opportunistic polling is done when it is convenient for the system, such as between major system functions or at some point in a repeated execution cycle.

Applicable Situations

- To get new sensor data or hardware signals into the system when the data or events are not highly urgent and the time between data sampling can be guaranteed to be fast enough

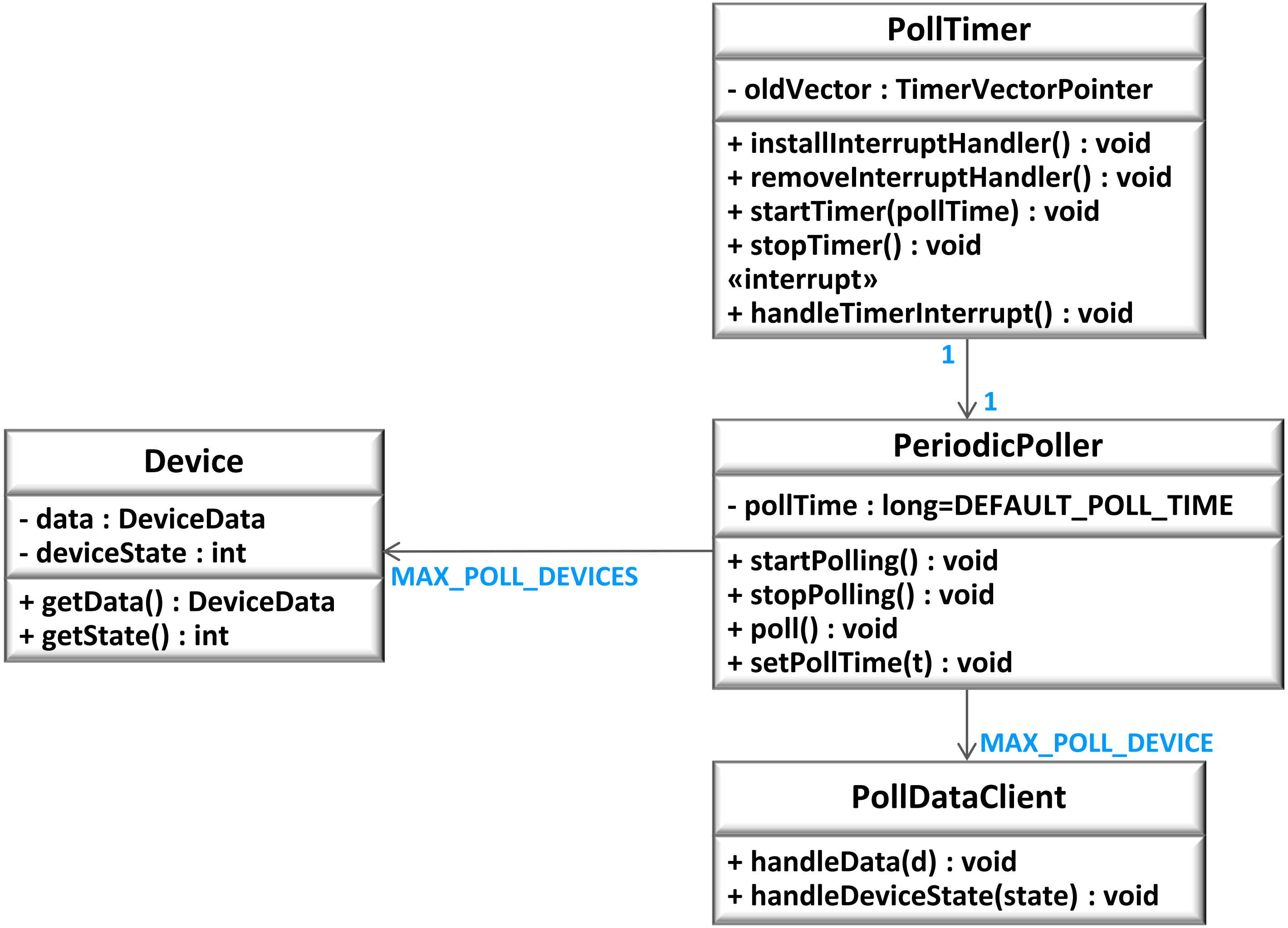
Structure (1)

- Opportunistic Polling Pattern



Structure (2)

- **Periodic Polling Pattern**
  - Usually implemented with ‘Interrupt Pattern’ tied to a poll timer



Structure (3)

- Device
  - To provide data and/or device state information via accessible functions
- ApplicationProcessingElement
  - To have the applicationFunction that has a loop in which it invokes the poll() operation
- OpportunisticPoller
  - To scan the attached devices for data and device state and pass this information on to the appropriate client for each
- PollDataClient
  - The client for data and state information from one or more of the devices
- PeriodicPoller
  - To scan the attached devices for data and device state and pass this information on to the appropriate client for each
- PollTimer
  - To insert the address of the ISR into the interrupt vector table (via installInterruptHandler())
  - To restore the original vector (via removeInterruptHandler())
  - To install the handleTimerInterrupt() (via startTimer()

Consequences

- **Advantages**
  - Easy to get data from sensors
- **Disadvantages**
  - Less timely than interrupts
    → Care must be taken that if there are deadlines associated with the data or signals.

Interceptor Pattern

Unit 27

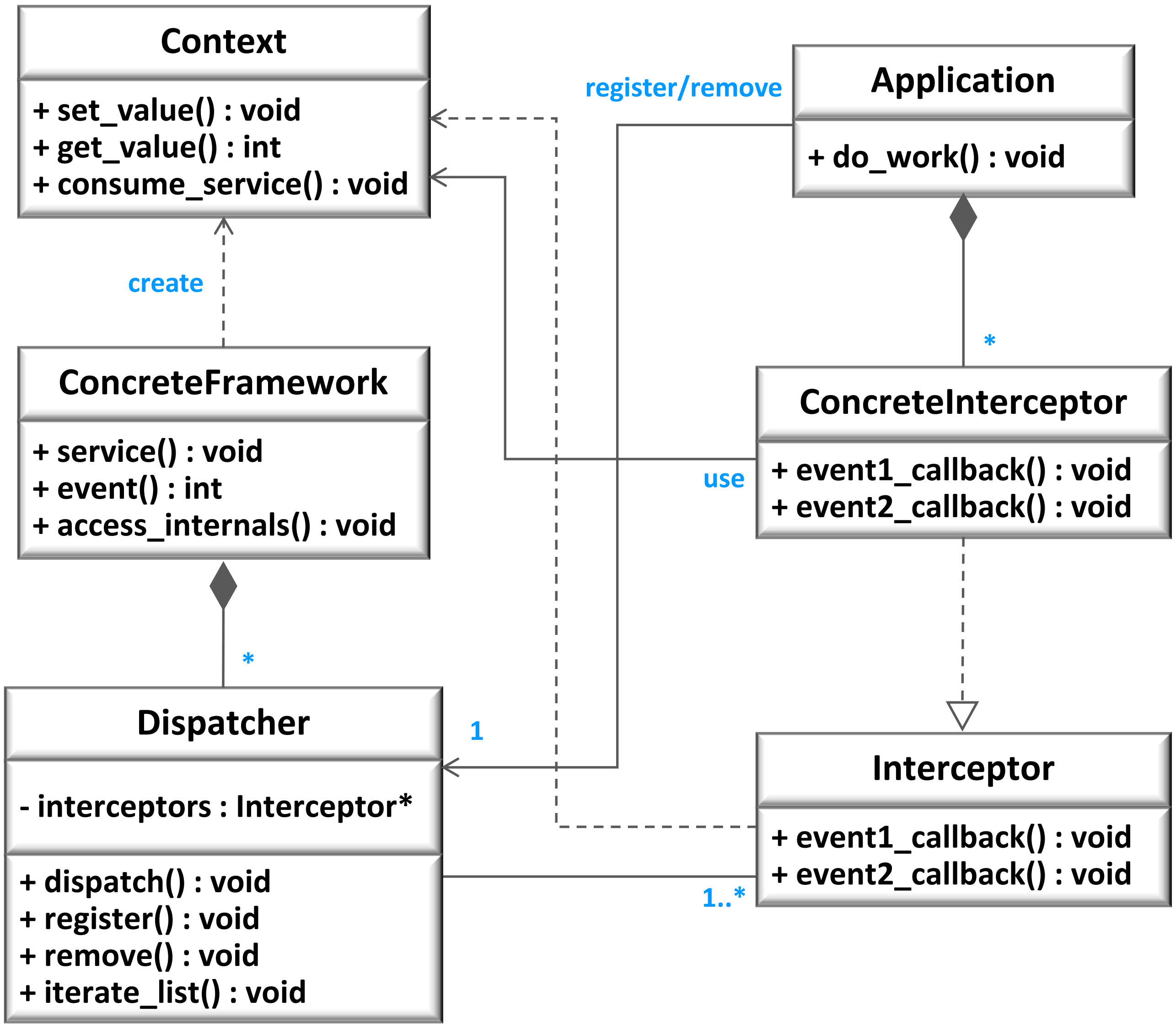
Overview

- **To allow services to be added transparently to a framework and triggered automatically when certain events occur**
- **Problem**
  - To anticipate all the services frameworks must offer to users
  - To allow integration of additional services without requiring modifications to a framework's core architecture
  - To monitor and control behavior of applications using a framework

Situation

- **To allow applications to extend a framework transparently by registering 'out-of-band' services with the framework via predefined interfaces**
  - To let the framework trigger these services automatically when certain events occur
  - To open the framework's implementation so that the out-of-band services can access and control certain aspects of the framework's behavior
- **To define context objects to allow a concrete interceptor to introspect and control certain aspects of the framework's internal state and behavior in response to events**

Structure


Context
+ set_value() : void
+ get_value() : int
+ consume_service() : void
register/remove
Application
+ do_work() : void
create
*
ConcreteFramework
+ service() : void
+ event() : int
+ access_internals() : void
ConcreteInterceptor
use
+ event1_callback() : void
+ event2_callback() : void
*
Dispatcher
1
- interceptors : Interceptor*
+ dispatch() : void
+ register() : void
+ remove() : void
+ iterate_list() : void
Interceptor
+ event1_callback() : void
+ event2_callback() : void
1..*

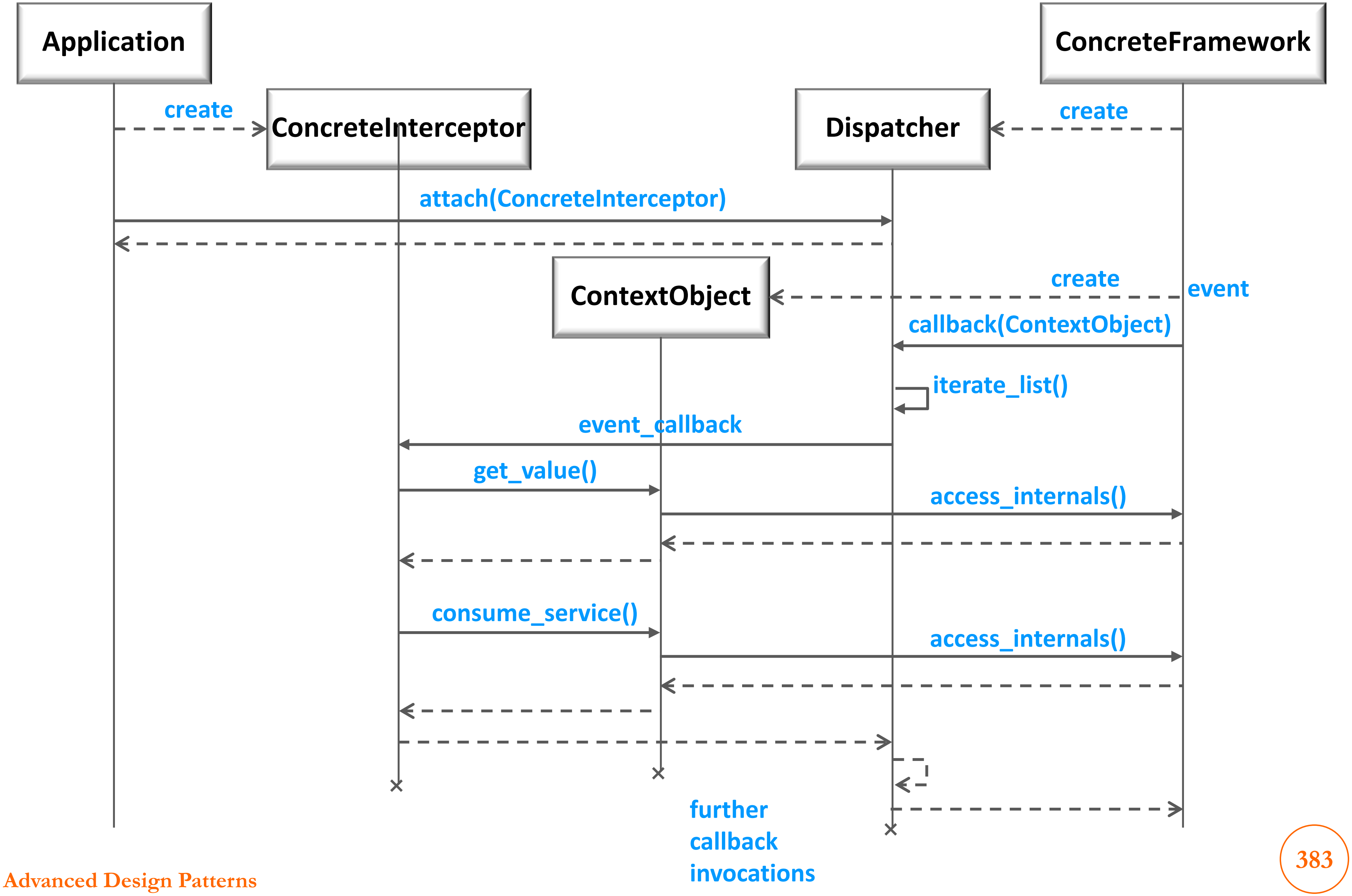
Participants (1)

- **ConcreteFramework**
  - To define application services
  - To integrate dispatchers that allow applications to intercept events
  - To delegate events to associated dispatchers
- **Interceptor**
  - To define an interface for integrating out-of-band services
- **ConcreteInterceptor**
  - To implement a specific out-of-band service
  - To use context object to control the concrete framework

Participants (2)

- **Dispatcher**
  - To allow applications to register and remove concrete interceptors
  - To dispatch registered concrete interceptor callbacks when events occur
- **Context Object**
  - To allow services to obtain information from the concrete framework
  - To allow services to control certain behavior of the concrete framework
- **Application**
  - To run atop the concrete framework
  - To implement concrete interceptors and registers them with dispatchers

Dynamics


Application
ConcreteInterceptor
Dispatcher
ConcreteFramework
create
create
attach(ConcreteInterceptor)
ContextObject
create
event
callback(ContextObject)
iterate_list()
event_callback
get_value()
access_internals()
consume_service()
access_internals()
further
callback
invocations

Implementation (1)

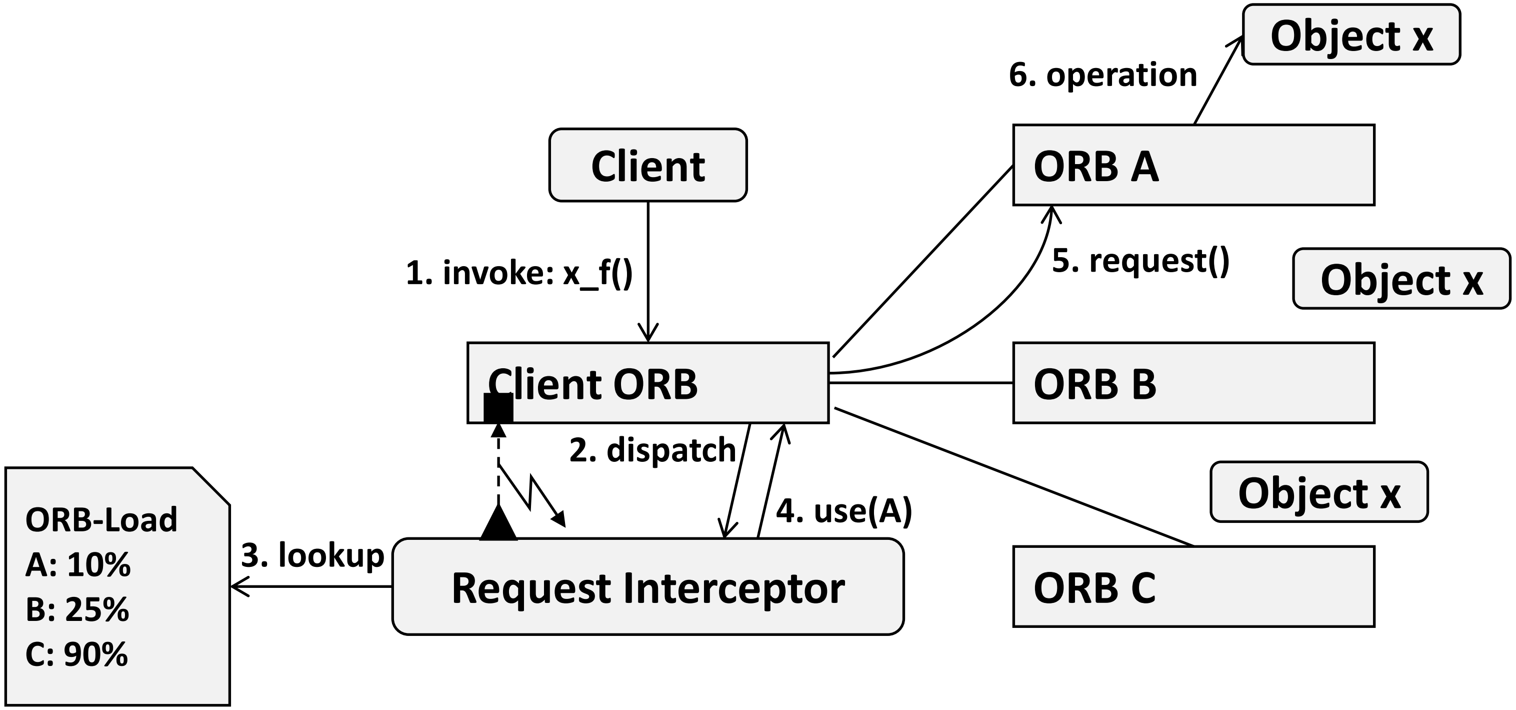
- **Step 1. Model the internal behavior of the concrete framework using a state machine or an equivalent notation, if such a model is not available already.**
- **Step 2. Identify and model interception points.**
  - Identify concrete framework state transitions.
  - Partition interception points into reader and writer sets.
  - Integrate interception points into the state machine model.
  - Partition interception points into disjoint interception groups.

Implementation (2)

- **Step 3. Specify the context objects.**
  - Determine the context object semantics.
  - Determine the number of context object types.
  - Define how to pass context objects to concrete interceptors.
- **Step 4. Specify the interceptors.**
- **Step 5. Specify the dispatchers.**
  - Specify the interceptor registration interface.
  - Specify the dispatcher callback interface
- **Step 6. Implement the callback mechanisms in the concrete framework.**
- **Step 7. Implement the concrete interceptors.**

Interceptor: Example (1)

- Applications can use the Interceptor pattern to integrate a customized load-balancing mechanism into MiddleORB.



Interceptor: Consequences

- **Pros**
  - Extensibility and flexibility
  - Separation of concerns
  - Support for monitoring and control of frameworks
  - Layer symmetry
  - Reusability
- **Cons**
  - Complex design issues
  - Malicious or erroneous interceptors
  - Potential interception cascades

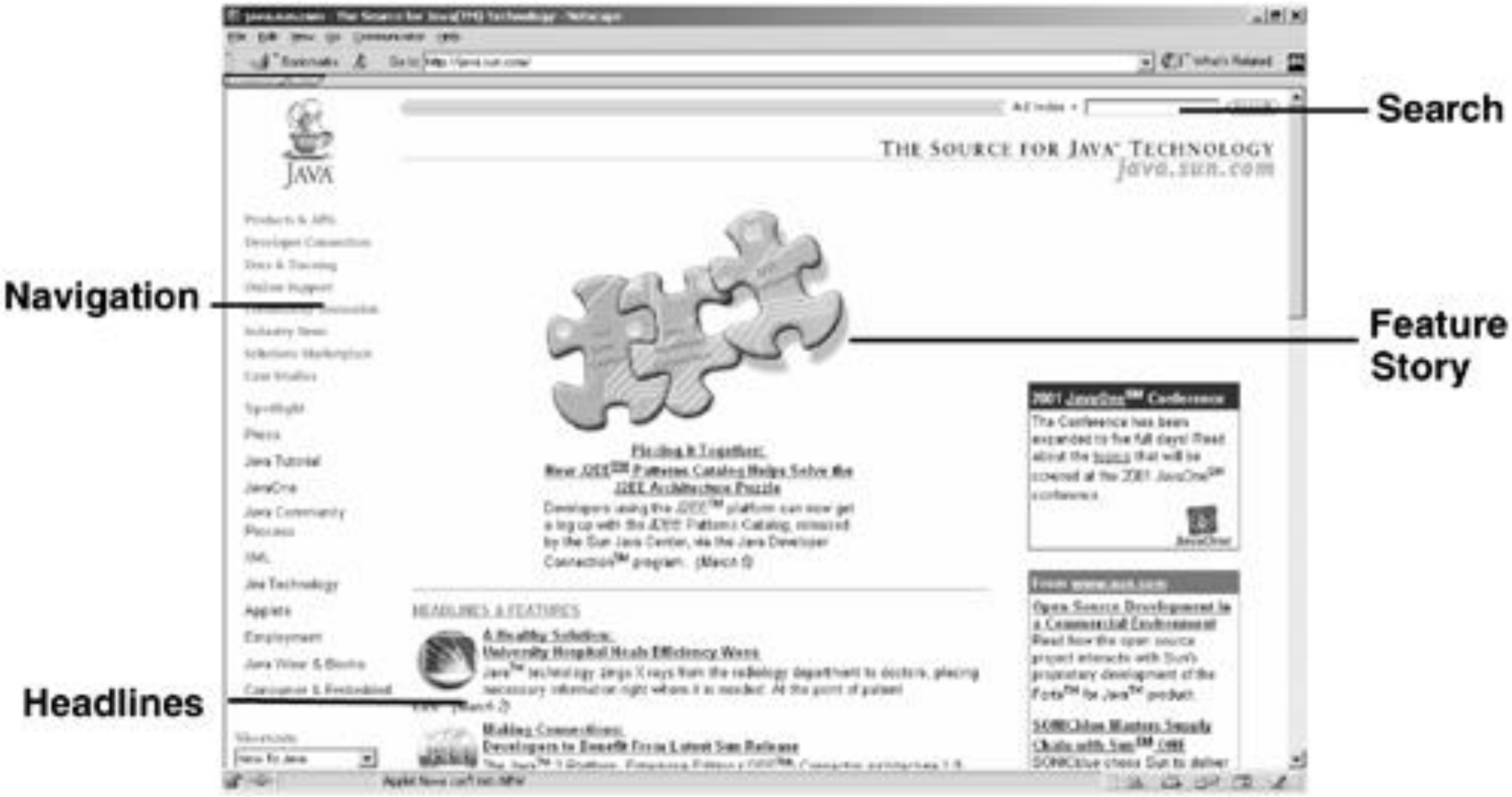
Composite View Pattern

Unit 28

Problem & Solution

- **Problem**
  - To build a view from modular, atomic component parts that are combined to create a composite whole, while managing the content and the layout independently
- **Solution**
  - Use it that are composed of multiple atomic subviews.
  - Each subview of the overall template can be included dynamically in the whole, and the layout of the page can be managed independently of the content.

Example



THE SOURCE FOR JAVA™ TECHNOLOGY
java.sun.com

Piecing it Together:
New J2EE℠ Patterns Catalog Helps Solve the J2EE Architecture Puzzle
Developers using the J2EE™ platform can now get a leg up with the J2EE Patterns Catalog, released by the Sun Java Center, via the Java Developer Connection℠ program. (March 8)

2001 JavaOne℠ Conference
The Conference has been expanded to five full days! Read about the topics that will be covered at the 2001 JavaOne℠ conference.

From www.sun.com
Open Source Development in a Commercial Environment
Read how the open source project interacts with Sun's proprietary development of the Forte™ for Java™ product.

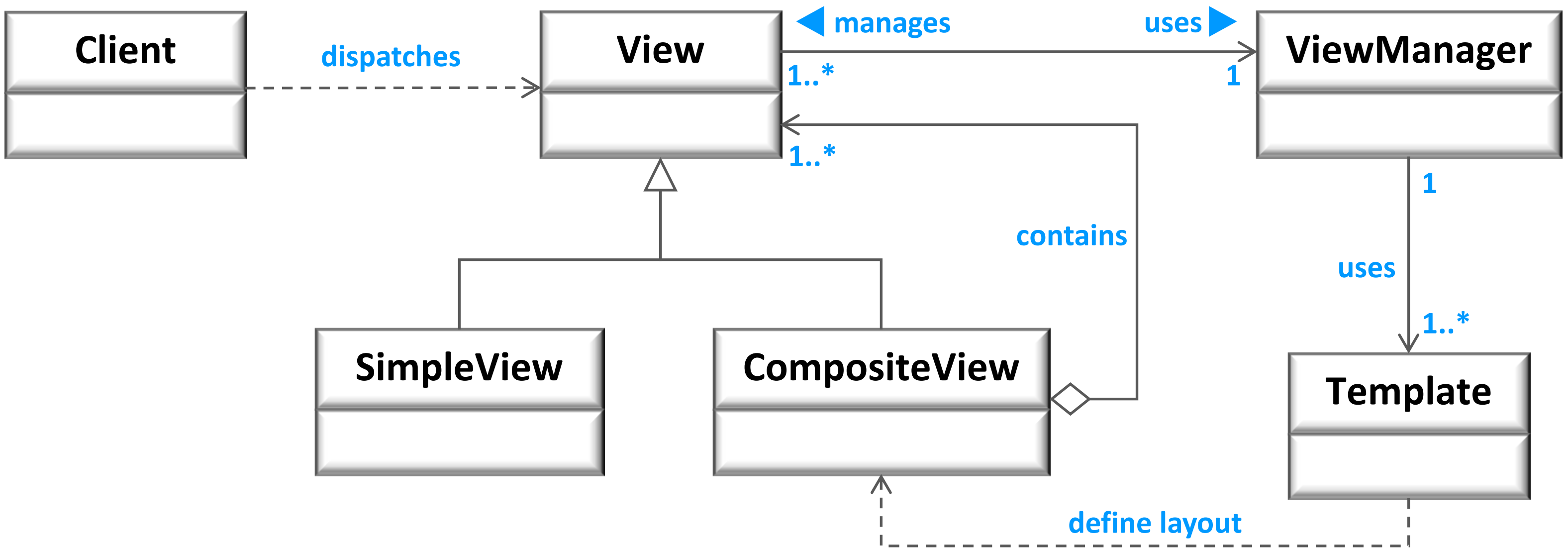
HEADLINES & FEATURES

A Healthy Solution:
University Hospital Heals Efficiency Woes
Java™ technology brings X rays from the radiology department to doctors, placing necessary information right where it is needed: At the point of patient care. (March 2)

Making Connections:
Developers to Benefit From Latest Sun Release

[Alur 03]

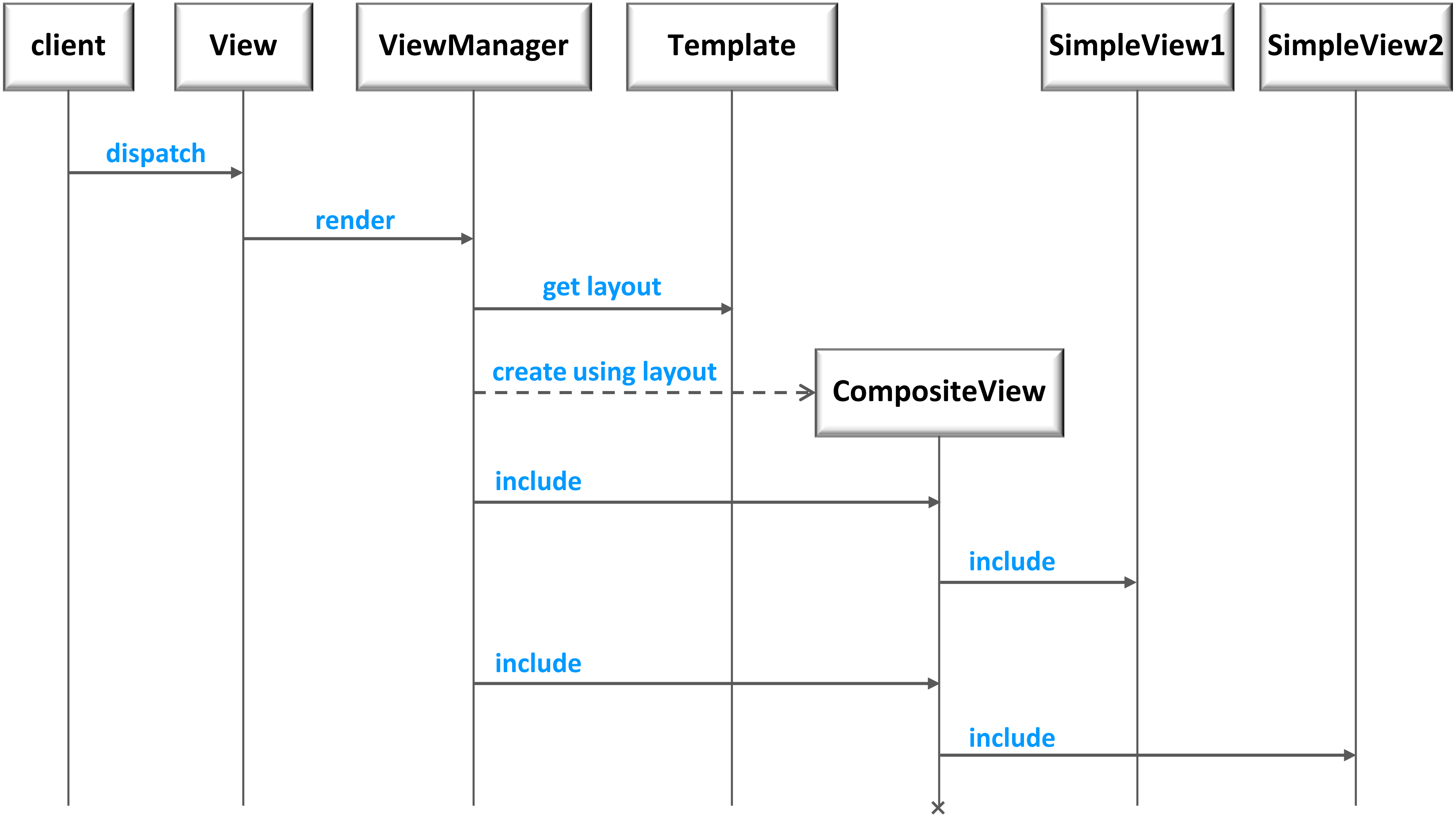
Structure


Client
dispatches
View
◀ manages
uses ▶
1..*
1
ViewManager
1..*
contains
1
uses
1..*
SimpleView
CompositeView
Template
define layout

Participants

Participant	Description
View	To represent the display
SimpleView	• To represent an atomic portion of a composite whole. • To be referred to as a view segment or subview
CompositeView	To be composed of multiple Views
Template	To represent the view layout
ViewManager	To use a Template to enforce a layout into which it places the appropriate content, allowing a ViewManager to manage content and layout independently

Behavior


client
View
ViewManager
Template
SimpleView1
SimpleView2
dispatch
render
get layout
create using layout
CompositeView
include
include
include
include

Introduction to Software Architecture

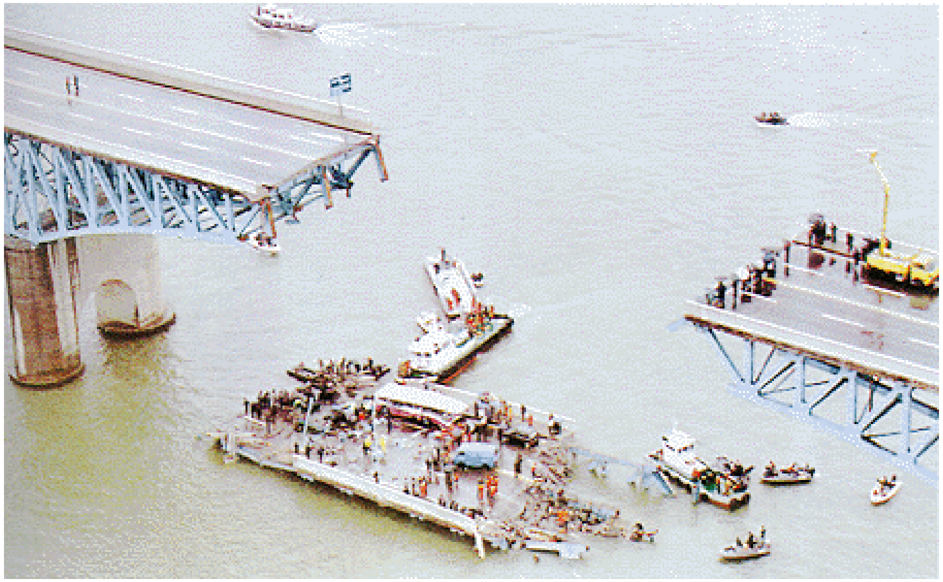
Unit 29

Collapse of Bridges (1)

- **35W Bridge, Mississippi River, USA**
  - Built in 1967, Collapsed in 2004
  - Poor Architecture Design

Collapse of Bridges (2)

- **Sungsoo Bridge, Seoul, Korea**
  - Built I 1979, Collapsed in 1994
  - Poor Construction/Implementation Technology



Collapse of Bridges (3)

- **Bay Bridge**
  - Built in 1936, Collapsed in 1989
  - 6.9 Earthquake

Golden Gate Bridge

- **Construction**
  - January 5, 1933: Construction begins
  - May 27, 1937: Bridge opens to pedestrians
  - May 28, 1937: Bridge open to automobiles
- **Architecture**
  - Suspension, Truss Arch & Truss Causeways
- **Total length: 1.7 miles**
- **746 feet (227 m)
  above the water**

World Trade Center

- **Build in 1973, Collapsed in 2001 (9.11)**
- **Impact of the Architecture**
  - Due to ‘Tube-Frame Structural System’

Software Architecture

- **Definitions**
  - Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution. [ISO/IEC 42010]

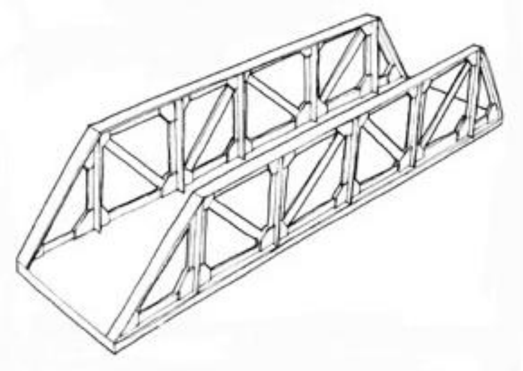
  - Structure or structures of the system, which comprise software elements, the externally visible qualities of those elements, and the relationships among them [Bass et el]
  - Elements + Form + Rationale [Perry and Wolf]
  - Design that involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns [Shaw and Garlan]
  - Software architecture deals with the design and implementation of the high-level structure of software. Architecture deals with abstraction, decomposition, composition, style, and aesthetics [Kruchten]

Requirements and Architecture

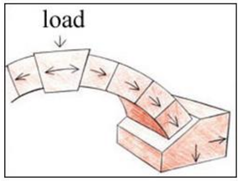
- **Software Requirement Specification (SRS)**
  - Functional Requirement
    - Use Cases
  - Non-Functional Requirement
    - Quality Requirement ✓
    - Constraints ✓

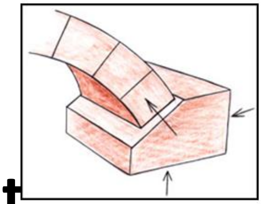
Styles for Bridges

- **Beam Style**

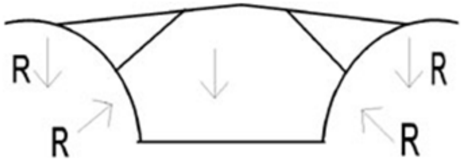


- **Arch Style**

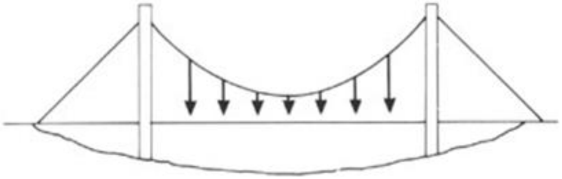




- **Cantilever Style**

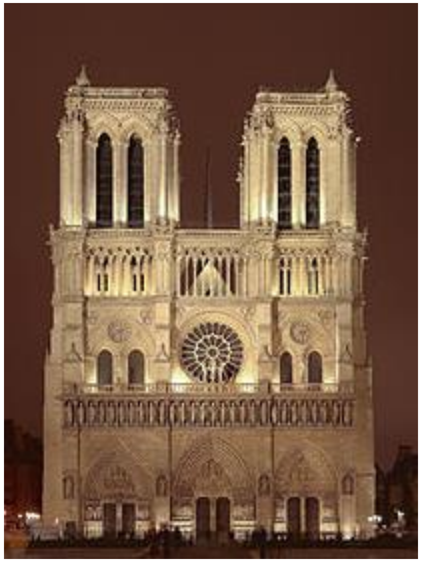


- **Suspension Style**

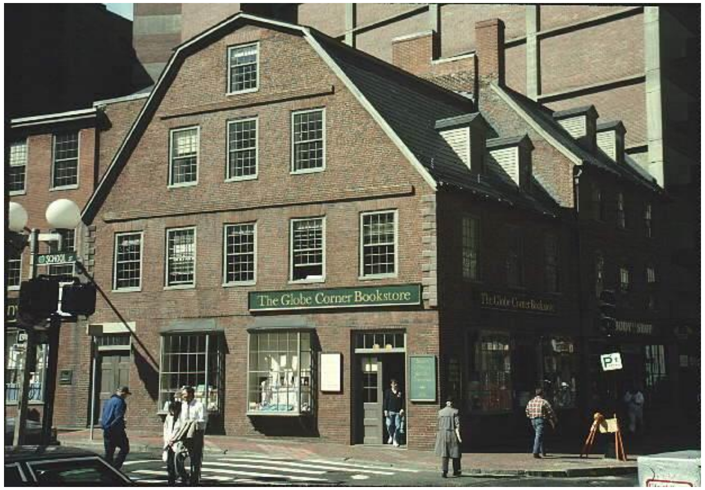


Styles for Buildings

- **Architectures in terms of**
  - Form, Techniques, Materials, Time period, Region, etc.



Early Gothic Architecture
Notre Dame, Paris



18th Century Georgian Architecture
“Old Corner Bookstore”, Boston MA

Styles for Software Systems (2)

- **A style consists of:**
  - A set of component types (e.g. process, procedure) that perform some function at runtime
  - A set of connectors (e.g. data streams, sockets) that mediate communication among components
  - A topological layout of the components showing their runtime relationships
  - A set of invariants (i.e. constraints)
    - e.g. direction of data-flow, dynamism, concurrency

Styles in [Rozanski 2005]

- Pipes and Filters
- Client/Server
- Tiered Computing
- Peer-to-Peer
- Layered Implementation
- Publisher/Subscriber
- Asynchronous Data Replication
- Distribution Tree
- Integration Hub
- Tuple Space

Styles in[Taylor 2009]

- Traditional Language-Influenced Styles
  - Main program and subroutines
  - Object-oriented
- Layered
  - Virtual machines
  - Client-server
- Dataflow Styles
  - Batch-sequential
  - Pipe-and-filter
- Shared Memory
  - Blackboard
  - Rule-based
- Interpreter
  - Interpreter
  - Mobile code
- Implicit Invocation
  - Publish-subscribe
  - Event-based
- Peer-to-Peer
- Complex Styles
  - C2 (Components and Connectors)
  - Distributed Objects

Styles in [Fairbanks 2010]

- Layered style
- Big ball of mud style
- Pipe-and-filter style
- Batch-sequential style
- Model-centered style
- Publish-subscribe style
- Client-server style
- N-tier
- Peer-to-peer style
- Map-reduce style
- Mirrored style
- Rack style
- Farm style

Styles in [Clements 2010]

- Module Structures
  - Decomposition Style
  - Uses Style
  - Generalization Style
  - Layered Style
  - Aspects Style
  - Data Model
- Allocation Structures
  - Deployment Style
  - Install Style
  - Work Assignment Style
  - Other Allocation Styles
- Component-and-Connector Structures
  - Data Flow Styles
    - Pipe-and-Filter Style
  - Call-Return Styles
    - Client-Server Style
    - Peer-to-Peer Style
    - Service-Oriented Architecture Style
  - Event-Based Styles
    - Publish-Subscribe Style
  - Repository Styles
    - Shared-Data Style

Architectural View

- **Definition**
  - A representation of one or more structural aspects of an architecture
  - Functional View, Information View, etc.
- **Analogy of Architectural View**
  - Blueprints of Building Design
- **Strategy**
  - A complex system is much more effectively described by using a set of interrelated views, which collectively illustrate its functional features and quality properties and demonstrate that it meets its goals.

Functional View

- **Describes system’s functional elements, their responsibilities, interfaces, and primary interactions.**
- **Cornerstone of the Architecture**
  - First part of AD that stakeholders try to read
- **Derives the shape of other views.**

Information View

- **Describes the way that the architecture stores, manipulates, manages, and distributes information.**
- **High-level view of static data structure**
- **Considers**
    - Content, structure, ownership, latency, references, and data migration

Concurrency View

- **Describes the concurrency structure of the system.**
- **Maps functional elements to concurrency units.**
  - To identify the parts of the system that can execute concurrently, and how this is coordinated and controlled.
- **Consider**
  - Process, Thread structures, and Inter-Process Communication (IPC)

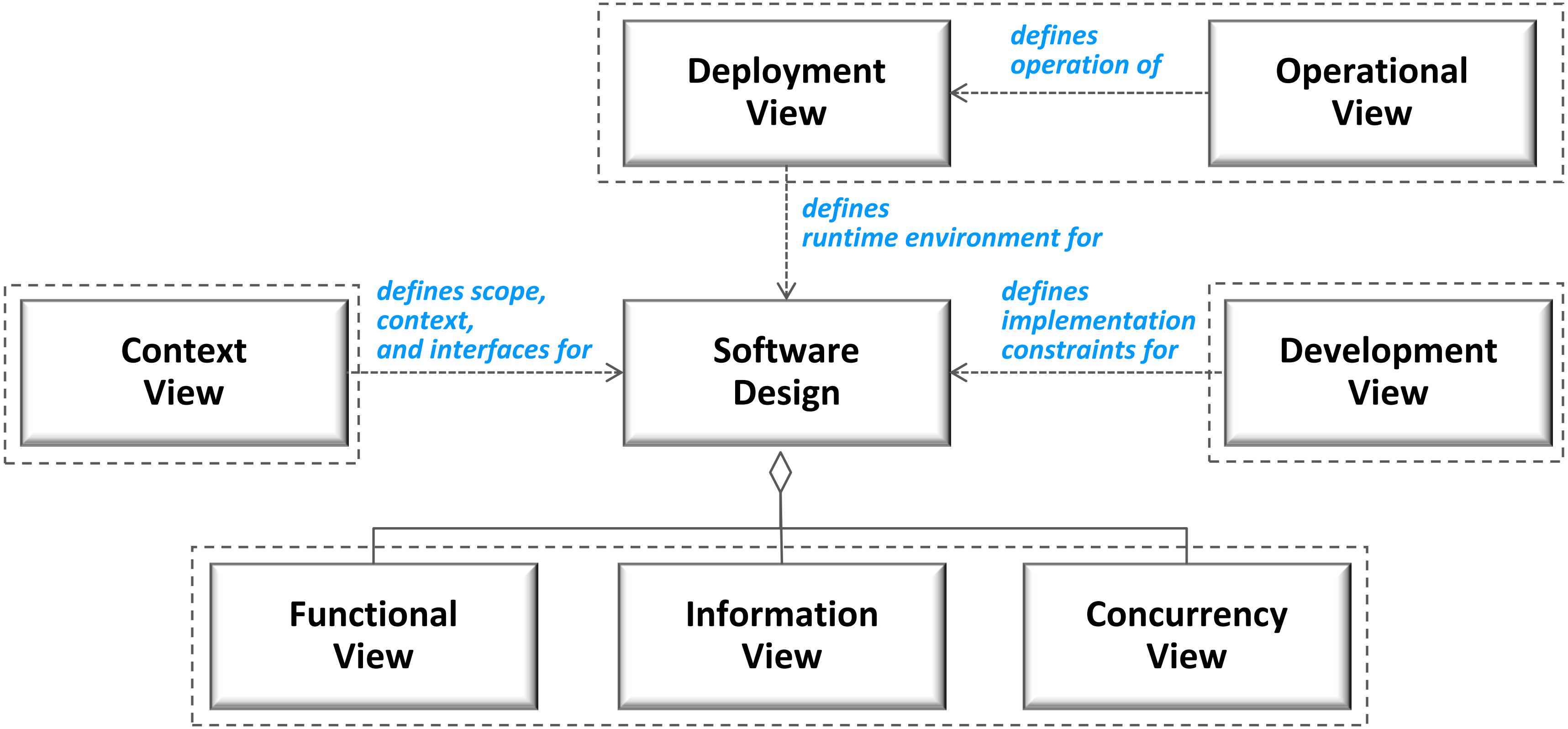
Deployment View

- **Describes the environment into which the system will be deployed.**
- **Consider**
  - Hardware Environment
  - Technical environment
  - Mapping of software elements to the runtime environment that will execute them

Operational View

- Describes how the system will be operated, administered, and supported when it is running its production environment.
- Identifies system-wide strategy for addressing operations concerns and their solutions.

Architectural View Groups



Architectural Viewpoint

- **Definition**
  - A collection of patterns, templates and conventions for constructing one type of view.
- **Provides a framework for capturing reusable architectural knowledge.**
- **Benefits**
  - Separations of Concerns
  - Communication with Stakeholder Groups
  - Management of Complexity
  - Improved developer focus

Architectural Perspective (1)

- **Definition**
  - A collection of activities, tactics, and guidelines that are used to ensure that a system exhibits a particular set of related *quality properties* that require consideration *across* a number of system's architectural views
- **Orthogonal to Viewpoints**
- **Issues addressed by architectural perspectives are often referred to as Non-Functional Requirement (NFR).**

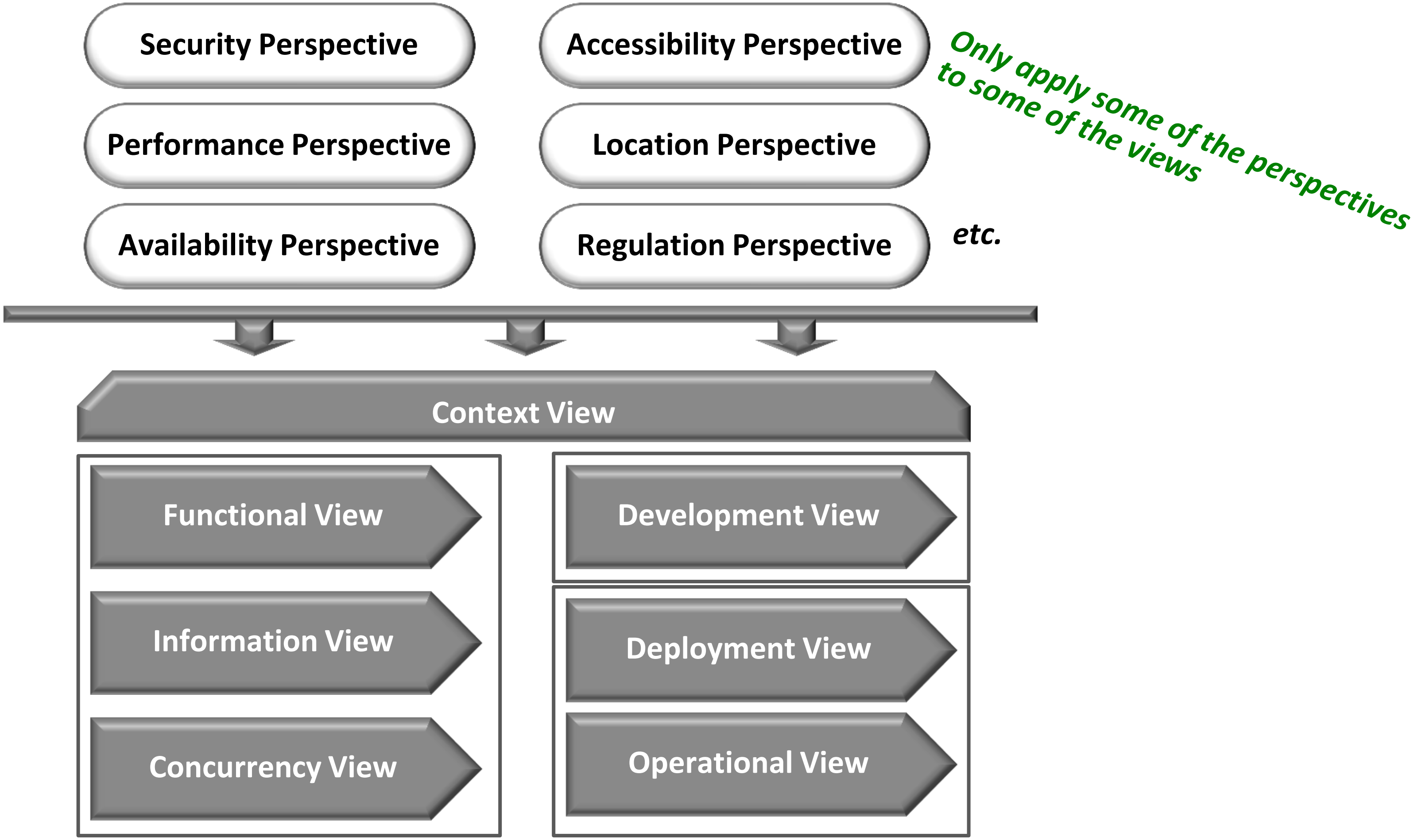
Architectural Perspective (2)

- **Specification Scheme**

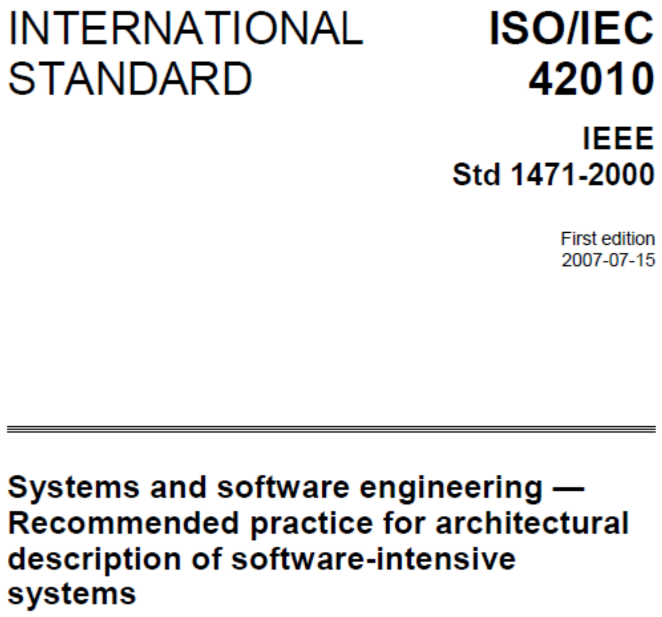
Item	Explanation
Applicability	To explain which of the views are most likely to be affected by applying the perspective
Concern	To define the quality properties that the perspective addresses
Activities	Steps for applying the perspective to the views - Identifying the important quality properties - Analyzing the views against the properties - Making architectural decisions that modify and improve the views
Architectural Tactics	To identify and describe the most important tactics, which are established and proven approaches architects can use to help achieve a particular quality property
Problems and Pitfalls	To explain the most common things that can go wrong and gives guidance on how to recognize and avoid them
Checklists	A list of questions to help architects make sure to address the most important concerns
Further Reading	To provide a number of pointers to further information

Applying Perspectives to Views (1)

- To apply each relevant perspective to some or all of the views in order to address that perspective's system-wide quality property concerns



Standard by ISO/ICE and IEEE



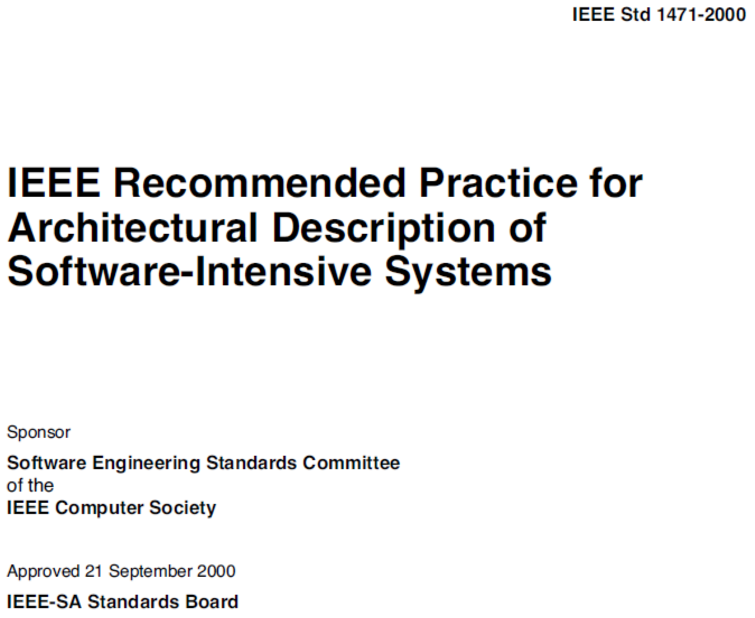
INTERNATIONAL STANDARD

ISO/IEC 42010

IEEE Std 1471-2000

First edition
2007-07-15

Systems and software engineering — Recommended practice for architectural description of software-intensive systems



IEEE Std 1471-2000

IEEE Recommended Practice for Architectural Description of Software-Intensive Systems

Sponsor

Software Engineering Standards Committee
of the
IEEE Computer Society

Approved 21 September 2000

IEEE-SA Standards Board

Styles, Viewpoints, and NFR-Design

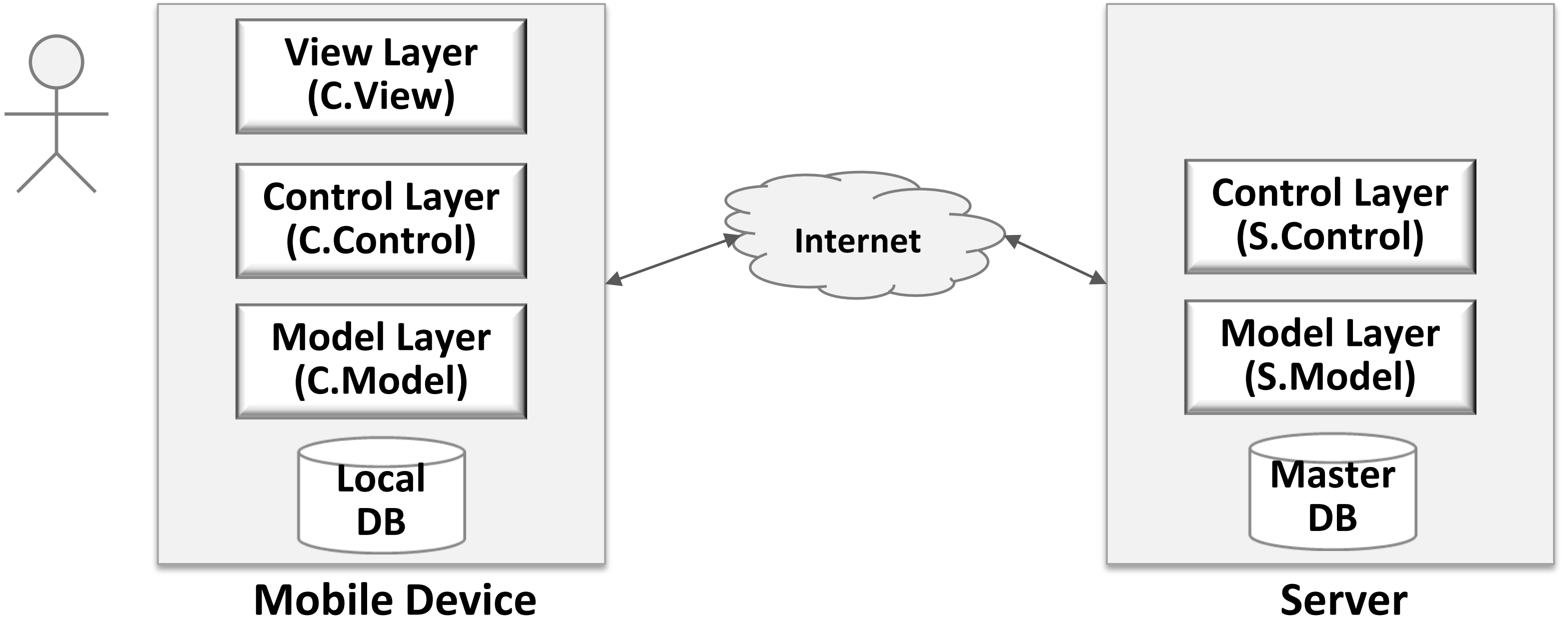
Unit 30

Balanced MVC

- When an application consists of two types of functionality; one functionality requiring only a local computation, and the other requiring a remote computation which is resource intensive or demands a shared information/database.
- When utilizing both the computing power of a mobile device and that of a more powerful server, as an effort to increase the overall performance through parallelism.

Structure (1)

- **Partitioning Functionality & Dataset into two parties.**
  - Mobile Device
  - Server



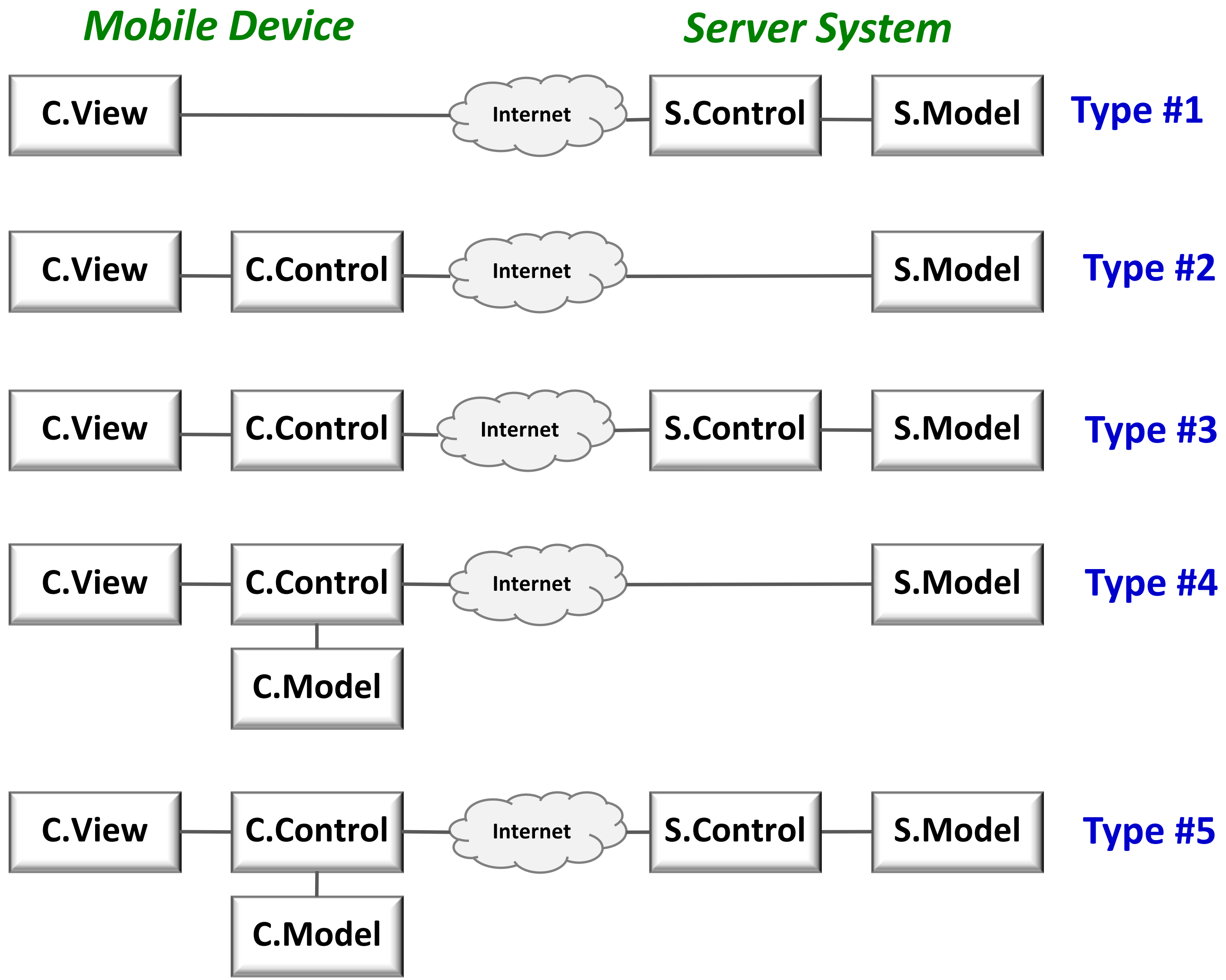
Structure (2)

- Mobile device may maintain a local cache DB for benefits, but with a synchronization effort.
- Control and model layers are deployed and executed on both mobile device and server.
- The server side typically does not have a view layer.

Structure (3)

- **C.Control vs. S.Control**
  - *C.Control* carries out the business process logics for a user-specific system.
  - *S.Control* implements common business logics that can be reused by multiple users.
- **C.Model vs. S.Model**
  - *C.Model* manages data for a specific user.
    - Stored on the secondary database or cache
  - *S.Model* manages persistent data for all the users.
    - Stored on the database in a secondary storage

Structure (4)



Mobile Device
Server System
C.View
Internet
S.Control
S.Model
Type #1
C.View
C.Control
Internet
S.Model
Type #2
C.View
C.Control
Internet
S.Control
S.Model
Type #3
C.View
C.Control
Internet
S.Model
C.Model
Type #4
C.View
C.Control
Internet
S.Control
S.Model
C.Model
Type #5

Structure (5)

- **Type 1**
  - Only *S.Control* and *S.Model* exist.
  - Realizing thin client side for resolving resource constraints
  - Always needs network capability
- **Type 2**
  - Only *C.Control* and *S.Model* exist.
  - Applicable situations
    - Low interaction between *C.Control* and *S.Model*
- **Type 3**
  - Both *C.Control* and *S.Control*, and also *S.Model* exist.
  - Applicable situations
    - Intensive interaction between *C.View* and *C.Control, and S.Control and S.Model*
    - Low interaction between *C.Control* and *S.Control*
  - A high parallelism on *C.Control* and *S.Control*

Structure (6)

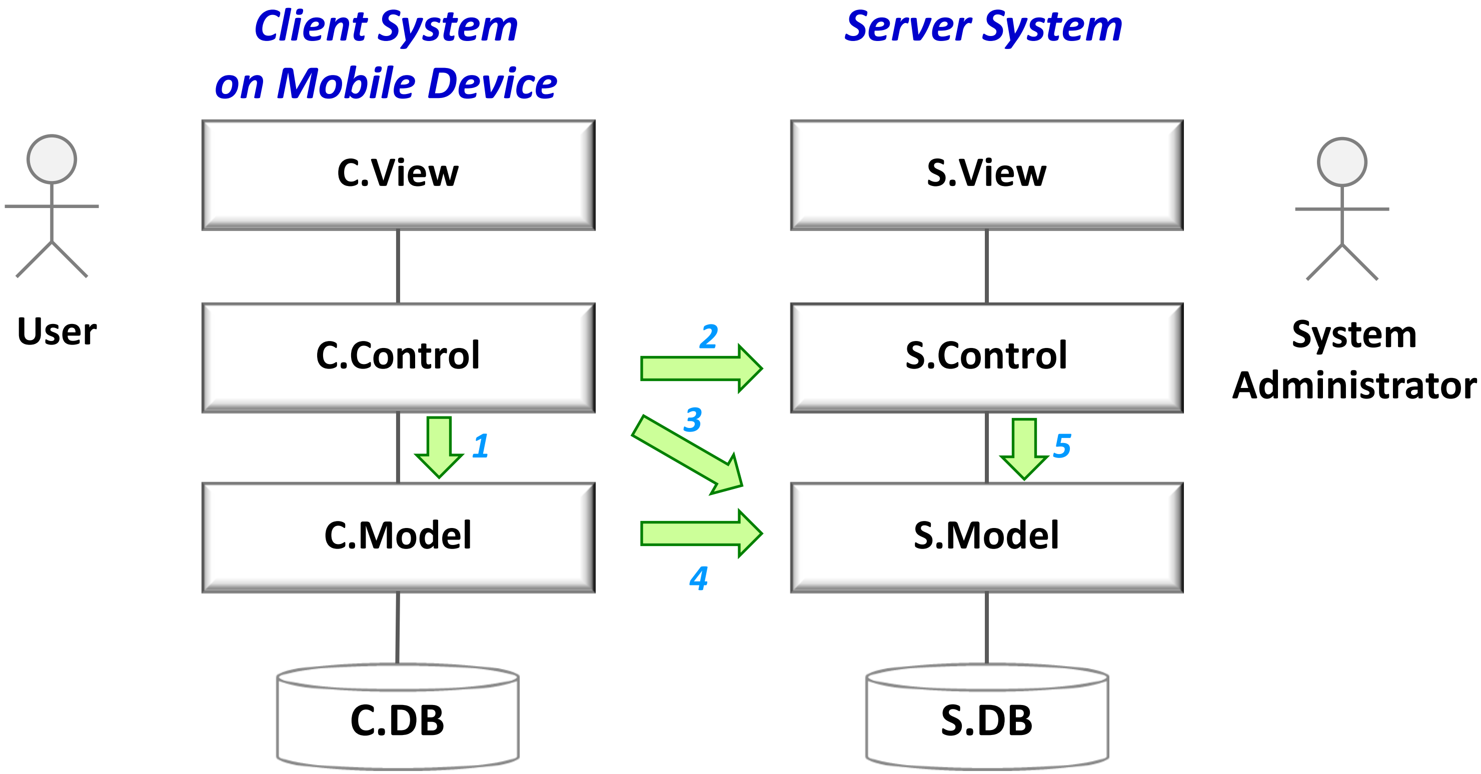
- **Type 4**
  - *C.Control*, and also *C.Model* and *S.Model* exist.
  - Applicable situations
    - Intensive interaction among *C.View*, *C.Control* and *C.Model*
    - Low interaction between *C.Control* and *S.Model*
  - High parallelism on *C.Model* and *S.Model*
- **Type 5**
  - Both *C.Control* and *S.Control*, and also *C.Model* and *S.Model* exist.
  - Integrated version of Pattern 3 and 4
  - Applicable Situation
    - High coupling/dependency between each pair of *Control* and *Model*.

Structure (7)

Type	Computation on Client Side	Network Overhead	Parallelism
1	Low	Low	Low
2	Middle	High	Low
3	Middle	Low	High
4	High	High	Middle
5	High	Low	High

Interaction (1)

- Diverse Interaction Paths



Interaction (2)

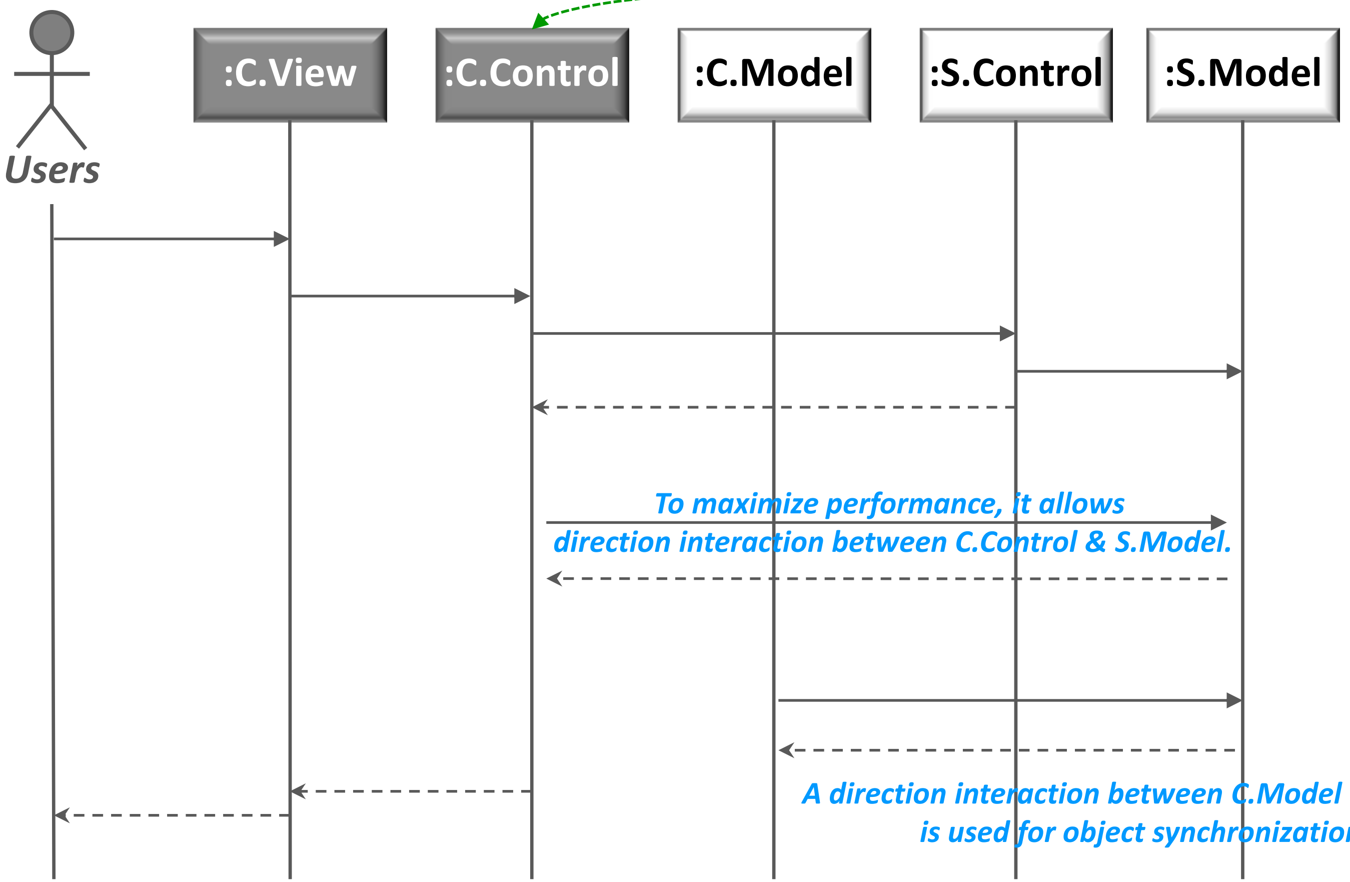
- **Path 1 : between *C.Control* and *C.Model***
  - Not requiring any communications with server system
  - Applicable to functionality which is invoked on the client system without relying on the server system.
- **Path 2 : between *C.Control* and *S.Control*.**
  - Applicable to the case that the functionality of *C.Control* can be fulfilled with the support of the *S.Control*.
  - *C.Control* provides client-specific functionality and *S.Control* provides common and reusable functionality in service-based systems.
- **Path 3 : between C.Control and *S.Model***
  - Without going through *S.Control*.
  - Applicable to the case that the *C.Control* needs to update the objects in *S.Model* efficiently.

Interaction (3)

- **Path 4: between *C.Model* and *S.Model***
  - To synchronize the states of two corresponding objects when the client system maintains copies of the entity objects. This is useful to maintain the state consistency.
- **Path 5: between *S.Control* and *S.Model***
  - To the typical invocation of entity objects on the server side.

Interaction (4)

C.Control manages its business process by invoking other components.

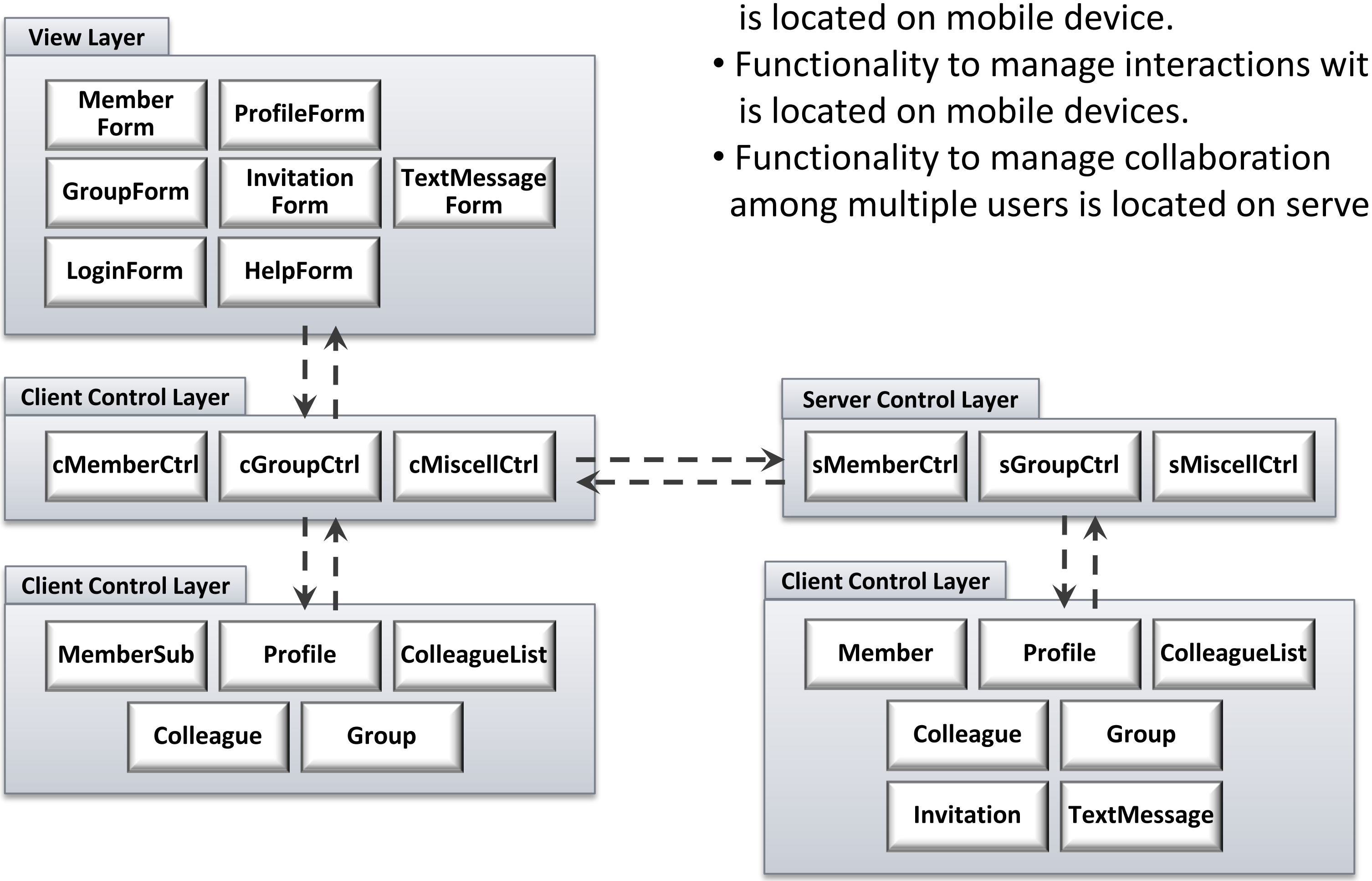


To maximize performance, it allows direction interaction between C.Control & S.Model.

A direction interaction between C.Model & S.Model is used for object synchronization.

Example

- **Mobile Mate Service**



- Functionality to manage client-specific data is located on mobile device.
- Functionality to manage interactions with users is located on mobile devices.
- Functionality to manage collaboration among multiple users is located on server system.

Pros and Cons

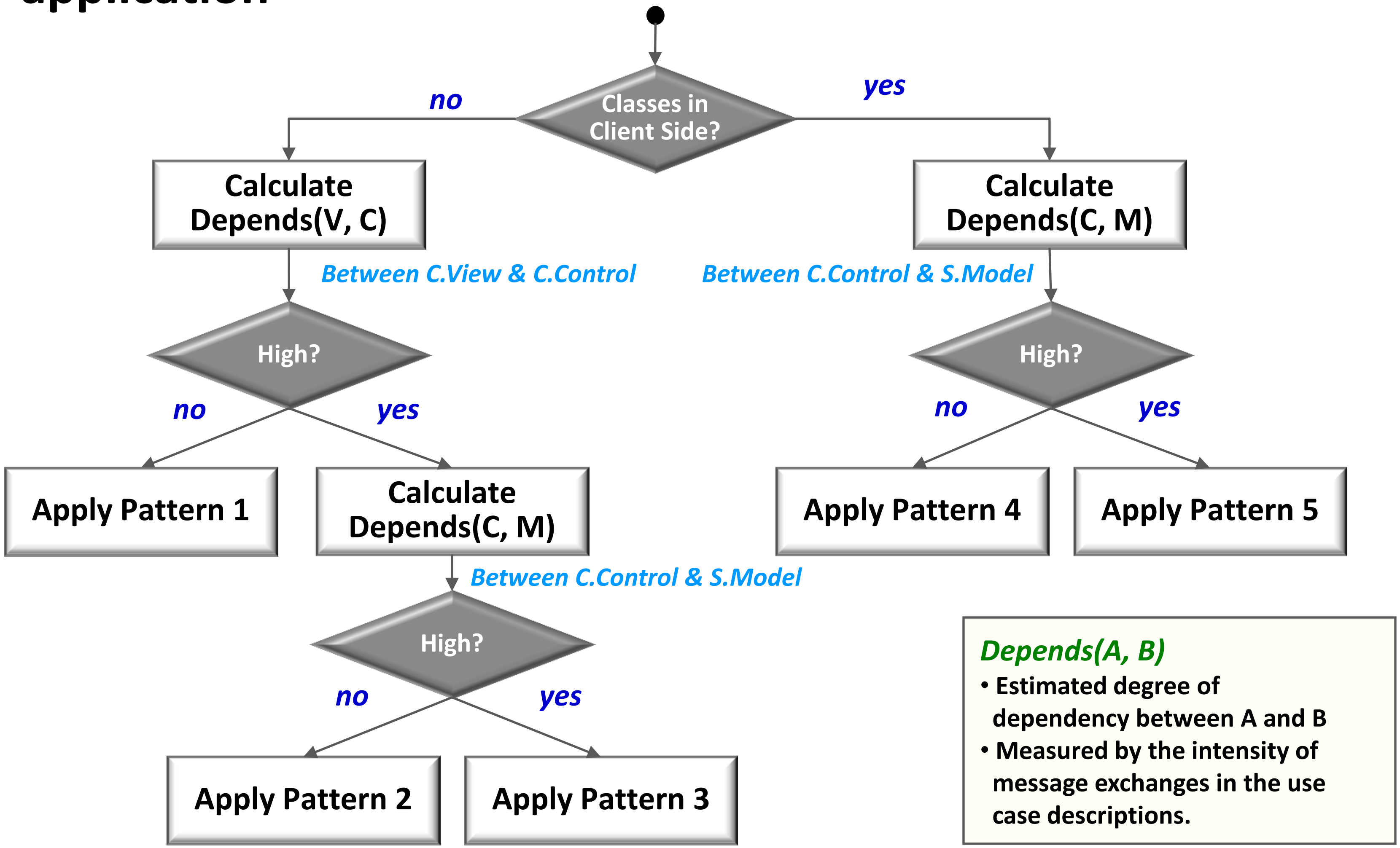
- **Pros**
  - Complex applications can be hosted.
    - Complex functionalities are deployed and run on server system.
  - Problems with resource limitation are resolved.
- **Cons**
  - Complex and precise engineering is required.
    - Especially, architecture design becomes the key issue.
  - Network overhead
  - May need to synchronize replicated objects.

Design Issues (1)

- **Criteria**
  - To minimize the amount of dataset flown over the network
  - To minimize the number of messages
  - To maximize coupling between functional unit (control layer) and dataset (model layer)
  - To minimize a cost of synchronizing states of objects on server side and states of their clone on client side
  - To maximize parallelism between two nodes

Design Issues (2)

- Step 1. To define the most appropriate type for the target application



Depends(A, B)

- Estimated degree of dependency between A and B
- Measured by the intensity of message exchanges in the use case descriptions.

Design Issues (3)

- **Step 2. Realize the pattern by deriving components**
  - Classes in *C.View*
    - From interactions between actor and client system specified in use case descriptions
  - Classes in *C.Model* and *S.Model*
    - Classes in the object models
  - Classes in *C.Control* and *S.Control*
    - From use case descriptions and object models
    - Start with checking *functional groups* in the use case description.
    - Refine the *functional groups*.
    - Derive one control class from one functional group.
- **Step 3. Define operations in each component**
  - For classes in *C.Control* and *S.Control*
    - From interactions between *actor* and *client system*, and between *client system* and *service system* in the use case description

Overview of Functional Viewpoint

Functional Viewpoint	
Definition	Describes the system’s functional element, their responsibilities, interfaces, and primary interactions
Concerns	Functional capabilities, external interfaces, internal structure, and design philosophy
Models	Functional structure model
Problems and Pitfalls	Poorly defined interface Poorly understood responsibilities Infrastructure modeled as functional elements Overloaded view Diagrams without element definitions Difficulty in reconcile
Stakeholders	All stakeholders
Applicability	All systems

Overview (2)

- **To define architectural elements that deliver the system's functionality**
  - Key functional element, their responsibilities, interfaces, and primary interactions
- **The cornerstone of most ADs**
  - The first part of the description that stakeholders try to read
  - Easiest view for stakeholders to read
- **To drive the shape of other system structures**
  - Such as the information structure, concurrency structure, and deployment structure etc.

Concern 1. Functional Capability

- **To define;**
  - What the system is required to do
  - What it is not required to do explicitly or implicitly
- **On projects**
  - Being given an good-quality requirements specification at the start of architecture definition can focus on showing how architectural elements work together to provide functionality.

Concern 2. External Interfaces

- **Data Flow between the system and others**
  - Inward Data
    - Resulting in an internal change of system state
  - Outward Data
    - As a result of internal changes of system state
- **Control Flow between the system and others**
  - Inbound Control Flow
    - A request by an external system to others to perform a task
  - Outbound Control Flow
    - A request by one system to another to perform a task
- **To consider both the interface syntax and semantics**
  - The structure of the data or request
  - Meaning or effect

Concern 3. Internal Structure

- **What internal elements do**
  - How they map onto the requirements
- **How they interact with each other**
- **Internal structure has a impact on the system properties such as;**
  - Availability, Resilience, Scalability, Security
    - e.g., a complex system is generally harder to secure than a simple one

Concern 4. Design Philosophy (1)

- **Consider different stakeholders' views**
  - What the system does and the interfaces it presents to user and to other system
  - How well the architecture adheres to sound principles of design
  - Good architecture to build, operate, and enhance easily
    - In the technical stakeholders' perspective
  - Faster, cheaper, and easier architecture to get a well-designed system into production
    - In the technical acquirers' perspective

Concern 4. Design Philosophy (2)

- **Design Quality Underpinning Design Philosophy**

Design Quality	Description	Significance
Specification of concerns	• Each internal element responsible for a distinct part of the system's operation? • Common processing performance	High separation (+)to build, support, and enhance easier (-)Performance and scalability
Cohesion	the functions provided by an element strongly related to each other	High cohesions sensible and tends to result in simpler, less error-prone design.
Coupling	How strong are the element inter-relationship?	Loosely coupled system (+)easier to build, support, and enhance (-)poor scalability
Volume of Interactions	What proportion of processing steps involve interactions between elements as opposed to within an element?	Communicating between certain types of elements (+)magnitude expensive, and less reliable
Functional Flexibility	How amenable is the system to supporting functional changes?	System designed changed easily (-) harder to build (+) Adaptable
Overall Coherence	Does the architecture "look right" when decomposed into elements?	Not "look like" - Underlying problems and hard to understand to stakeholders.

Concern 5. Stakeholder Concerns

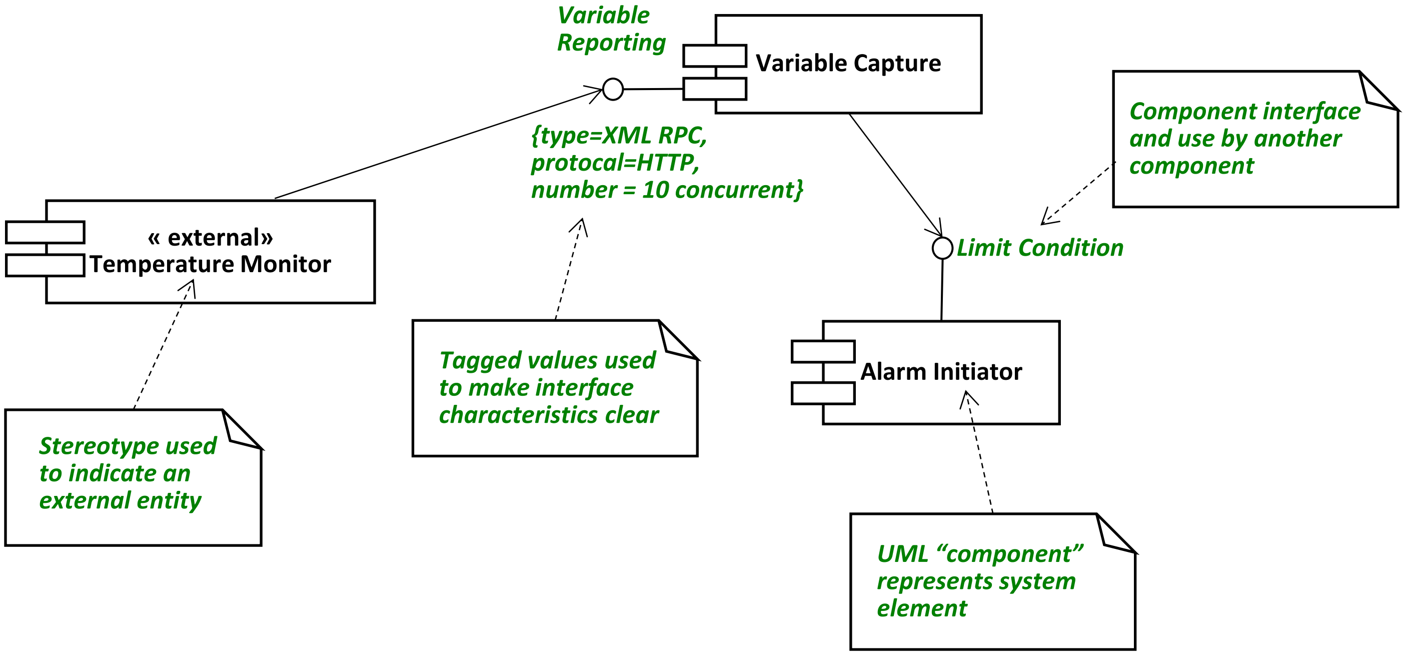
Stakeholder Class	Concerns
Acquirers	Primarily functional capabilities and external interfaces
Assessors	All concerns
Communicators	All concerns, to some extent
Developers	Primarily design philosophy and internal structure, but also functional capabilities and external interfaces
System administrators	Primarily design philosophy and internal structure
Testers	Primarily design philosophy and internal structure, but also functional capabilities and external interfaces
Users	Primarily functional capabilities and external interfaces

System's Elements

- **Functional Elements**
  - A well defined part of the runtime system that has particular responsibilities and expose defined interfaces
    - Software code module, Application package, Data store.
- **Interfaces**
  - A well defined mechanism that the functions of an element can be accessed by other elements
    - Defined by inputs, outputs, and semantics of each operation
- **Connectors**
  - Pieces of architecture that link the elements together to allow them to interact
    - To define the interaction between elements that use it
- **External entities**
  - Other system such as software programs, hardware devices

Functional Structure Model (1)

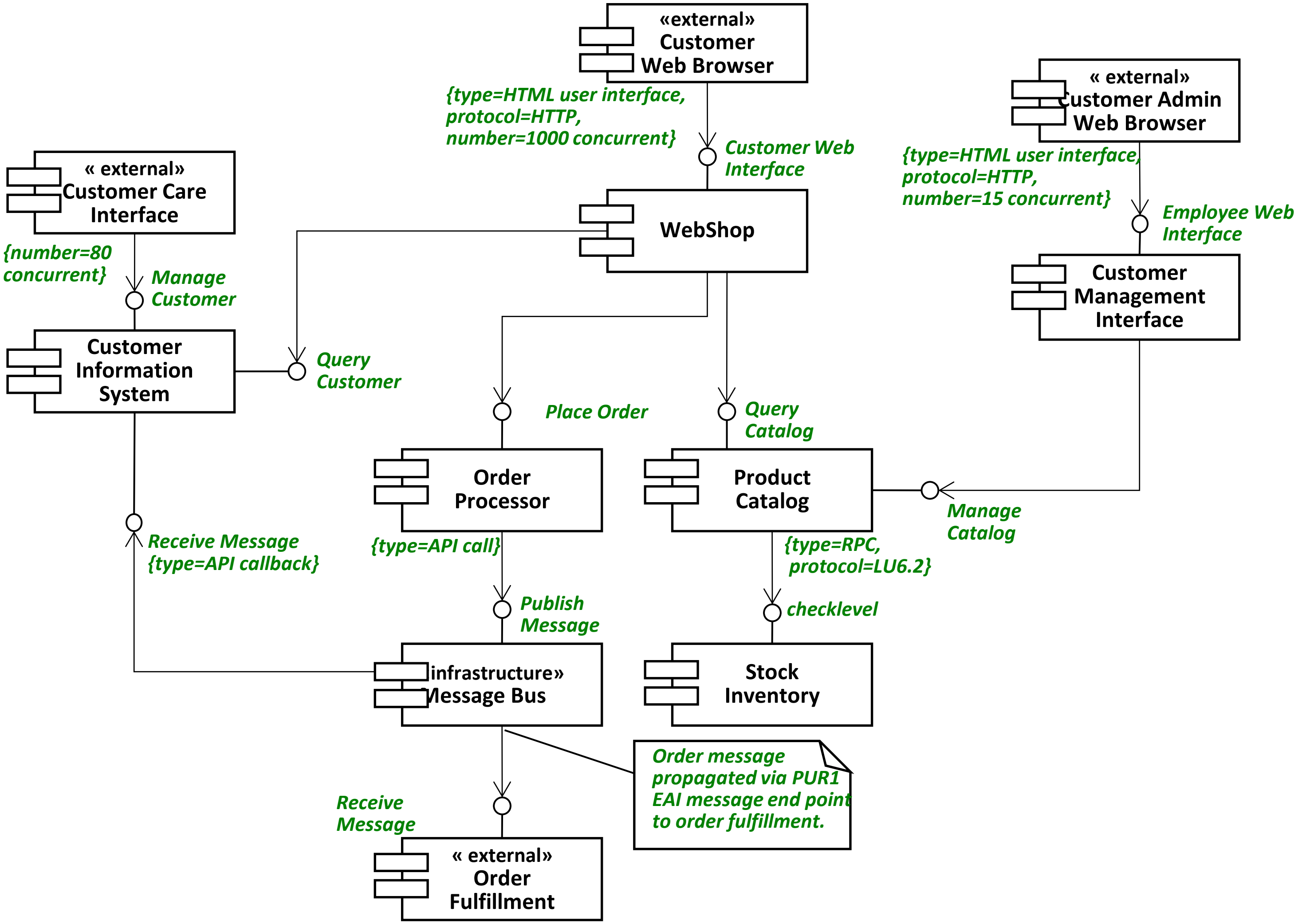
- **UML Component Diagram**
  - To show a system's elements, interfaces, and inter-element connections
  - Example of a Functional Structure in UML



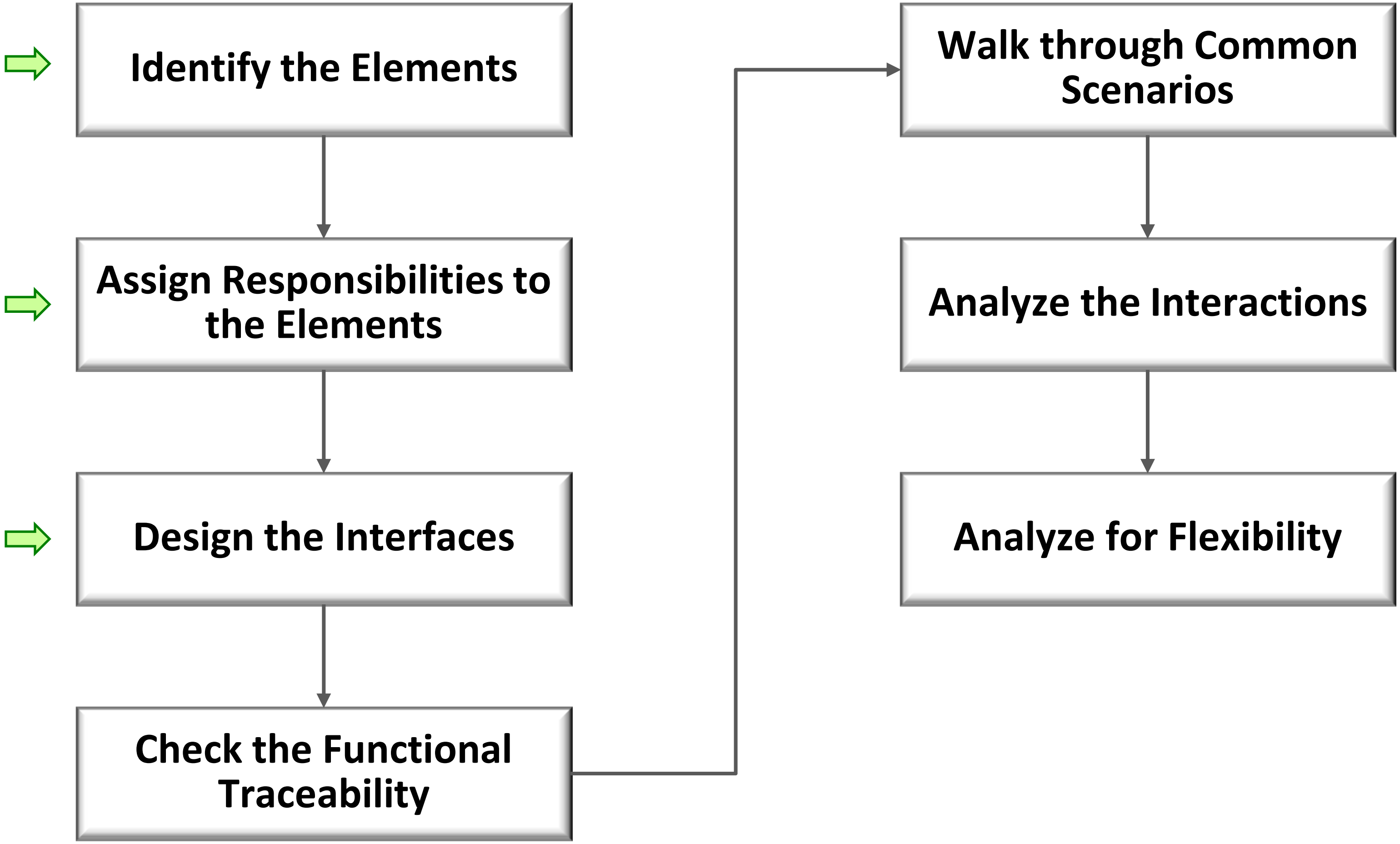
Variable Reporting
Variable Capture
{type=XML RPC, protocol=HTTP, number = 10 concurrent}
Component interface and use by another component
« external» Temperature Monitor
Limit Condition
Alarm Initiator
Tagged values used to make interface characteristics clear
Stereotype used to indicate an external entity
UML "component" represents system element

Functional Structure Model (2)

- Example of UML Component Diagram



Steps in Applying Functional Viewpoint


⇨ Identify the Elements
⇨ Assign Responsibilities to the Elements
⇨ Design the Interfaces
Check the Functional Traceability
Walk through Common Scenarios
Analyze the Interactions
Analyze for Flexibility

Overview of Security Perspective

Desired Quality	The ability of the system to reliably control, monitor, and audit who can perform what actions on these resources and the ability to detect and recover from failures in security mechanisms
Applicability	Any systems with publicly accessible interfaces, with multiple users where the identify of the user is significant, or where access to operations or information needs to be controlled
Concerns	Policies, threats, mechanisms, accountability, availability, detection and recovery
Activities	Identify Sensitive Resources → Define Security Policy → Identify Threats to System → Design Security Implementation → Assess Security Risks
Architectural Tactics	Apply recognized security principles, authenticate the principals, authorize access, ensure information secrecy, ensure information integrity, ensure accountability, protect availability, integrate security technologies, provide security administration, use third-party security infrastructure
Problems and Pitfalls	complex security policies, unproven security technologies, system not designed for failure, lack of administration facilities, technology-driven approach, failure to consider time sources, overreliance on technology, no clear requirements or models, security as an afterthought, security embedded in the application code, piecemeal security, ad hoc security technology

Applicability to Views

View	Applicability
Functional	• To allow to see which of the system's functional elements need to be protected • The functional structure of the system may be impacted by the need to implement security policies.
Information	• To help you see what needs to be protected, i.e. sensitive data in the system • To be modified as a result of security design (e.g. partitioning information by sensitivity)
Concurrency	• To indicate the needs to isolate different pieces of the system into different runtime elements, which will affect the system's concurrency structure
Development	• To identify guidelines or constraints that the software developers will need to be aware of in order to ensure the security policy is enforced
Deployment	• The security design may have a major impact on this view. (e.g. security-oriented hardware or software)
Operational	• How the system is operated once it is in production will have a major affect on the security.

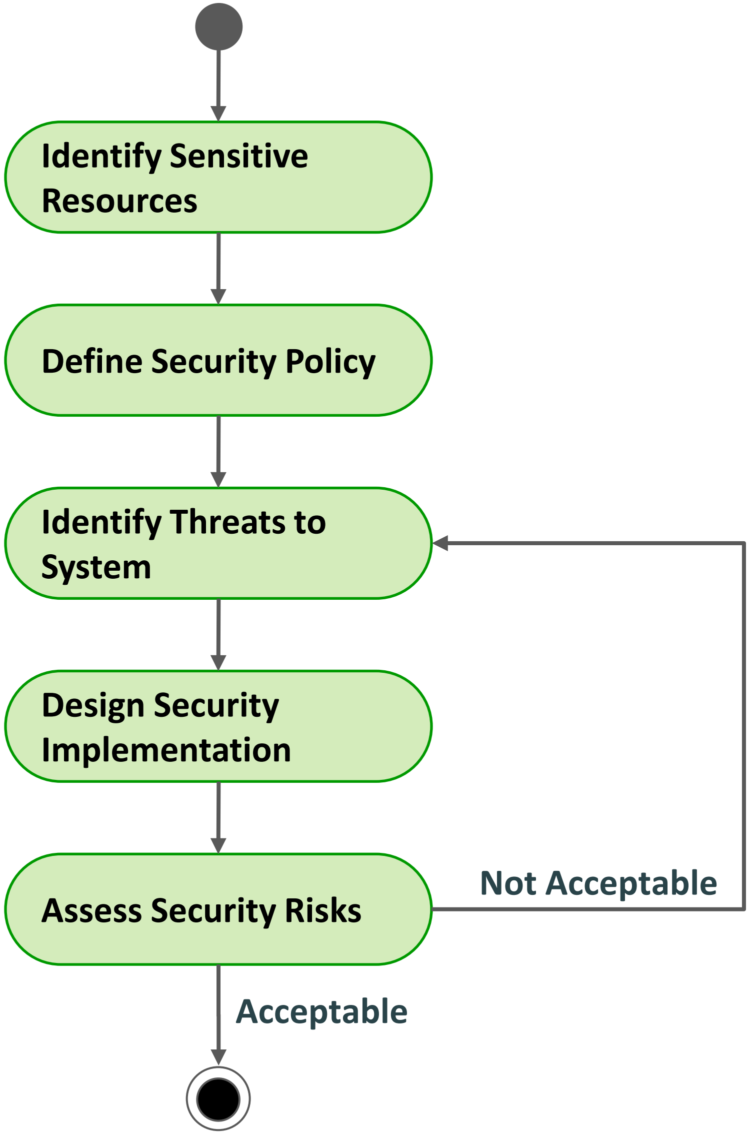
Concerns (1)

- **Policies**
  - To define the system's security needs
    - To define controls and guarantees that the system requires for its resources
  - To define information access policy in terms of;
    - The different types of principals the system contains
    - For each type of information, what sort of access each principal group requires
  - To define information integrity constraints
- **Threats**
  - To allow you to identify security enforcement mechanisms that can be counter the threats
  - Example
    - password cracking, network attacks, denial-of-service attacks, social-engineering attacks

Concerns (2)

- **Security Mechanisms**
  - A set of technologies, configuration setting, and procedures required to enforce the rules established by the security policy
  - To select the right set of technologies from the wide array available and to apply them appropriately for the system
- **Accountability**
  - A means of ensuring that every action can be unambiguously traced back to the principal who performed it
- **Availability**
  - To become more important as systems routinely connect to public networks like the Internet
- **Detection and Recovery**
  - The important security concern in the ability of the system to detect security breaches and recover from them.

Micro Process



Identify Sensitive Resources

Define Security Policy

Identify Threats to System

Design Security Implementation

Assess Security Risks

Not Acceptable

Acceptable

Guidelines for ‘Identify Sensitive Resources’

- **Decide what the sensitive resources in the system are.**
  - By using the Functional and Information views as primary inputs, along with any security requirements information available
  - Or by using organizational security policies and relevant external regulations, as well as the opinions of interested stakeholders
  - Typically, functional operations and data items are sensitive resources.
- **For each type of sensitive resource, define;**
  - The reasons the resource is sensitive
  - Who should be considered as its owner
  - Type of access controls required for the resource

Guidelines for ‘Identify Sensitive Resources’

- **Example of Sensitive Resource Identification**

Resource	Sensitivity	Owner	Access Control
Customer account records	Personal information of value for identify theft or invasion of privacy	Customer Care Group	No direct data access
Descriptive product catalog entries	Defines what is for sale and its description; if changed maliciously, could harm the business	Stock Management Group	No direct data access
Pricing product catalog entries	Defines pricing for catalog items; if maliciously or accidentally modified, could harm business or allow fraud	Pricing Team in Stock Management Group	No direct data access
Business operations on customer account records	Needs to be controlled to protect data access and integrity	Customer Care Group	Access to individual record or all records by authenticated principal
Descriptive catalog operations	Needs to be controlled to protect data access and integrity	Stock Management Group	Access to catalog modification operations by authenticated principal

Guidelines for ‘Define the Security Policy’

- **Identify the principal classes.**
  - To make defining the security policy more manageable, start by grouping principals into classes you can treat as groups for security policy purposes.
  - Partitions the principals into sets based on the types of access they require to different types of sensitive resources.
- **Identify the resource classes.**
  - Partition the system’s sensitive resource types into groups that you can treat together for access control purposes.
- **Identify the access control sets.**
  - For each resource class, define the operations that can be performed on members of that class and the principal classes that should be allowed to access each operation on the class.

Guidelines for 'Define the Security Policy'

- **Identify the sensitive operations.**
  - Consider any system-level operations that are independent of the system's managed resources (s.a. administrative operations).
  - Define which principal classes should be allowed to access these operations.
- **Identify the integrity requirements.**
  - Consider any situations in the system where information is or could be changed.
  - Identify the set of integrity guarantees required for that information.

Guidelines for ‘Identify Threats’

- **Identify the threats.**
  - Consider;
    - The security threats your system faces from the perspective of the sensitive resource
    - The possible access to these resources that potential attackers might wish to gain
    - The main characteristics of the potential attackers
    - The types of attacks they are likely to carry out
- **Characterize the threats.**
  - Characterize each threat in terms of;
    - The resources that would be compromised if the attack were successful
    - The result of this compromise
    - The likelihood of the attack occuring

Guidelines for 'Identify Threats'

- **Example of Access Control Policy**

	User Account Records	Desc. Catalog Records	Pricing Records	User Account Operations	Desc. Catalog Operations
Data Administrator	Full with audit	Full with audit	Full with audit	Full with audit	Full with audit
Catalog Clerk	None	None	None	All	Read-only Operations
Catalog Manager	None	None	None	Read-only operations with audit	All
Product Price Administrator	None	None	None	None	Read-only Operations
Customer Care Clerk	None	None	None	All	Read-only Operations
Registered Customer	None	None	None	All on own record	Read-only Operations
Unknown Web-Site User	None	None	None	None	Read-only Operations

Guidelines for ‘Design the Security Impl.’

- **Design a way to mitigate the threats.**
  - For each of the threats to sensitive resources, design a security mechanism.
- **Design a detection and recovery approach.**
  - To include;
    - Technical intrusion detection solutions
    - Internal system checks and balances to reveal unexpected inconsistencies
    - A set of processes for regularly checking the system for intrusions and reacting to any discovered
- **Access the technology.**
  - To access which candidate security technologies are suitable for addressing a particular threat in a particular context
- **Integrate the technology.**

Guidelines for ‘Access the Security Risks’

- **Access the Risks**
  - For each rick in threat model;
    - Reevaluate its likelihood of occurrence given the planned security infrastructure
    - Compare its against the security needs established earlier in the process

Risk	Estimated Cost	Estimated Likelihood	Notional Cost
Attacker gains direct DB access.	$8,000,000	0.2%	$16,000
Website flaw allows free order to be placed and fulfilled.	$800,000	4%	$32,000
Social-engineering attack on a customer service representative results in hijacking of customer accounts.	$4,000,000	1.5%	$60,000
…	…	…	…

Thank You !