Design Patterns

Part II

https://sourcemaking.com/design_patterns

Gang of Four patterns

There are three basic kinds of design patterns:

- Creational
- Structural
- Behavioral

Creational Patterns

- Creational design patterns separate the object creation logic from the rest of the system
- Deal with initializing and configuring classes and objects
- Instead of you creating objects, creational patterns create them for you

Creational Patterns

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

Structural Patterns

- Sometimes you need to build larger structures by using an existing set of classes
- Structural class patterns use inheritance to build a new structure
- Structural object patterns use composition / aggregation to obtain a new functionality

Structural Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Behavioral Patterns

- Behavioral patterns govern how objects communicate with each other
- Deal with dynamic interactions among societies of classes and objects
- How they distribute responsibility

Behavioral Patterns

- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

Elements of Design Patterns

Design patterns have 4 essential elements

- *Pattern name*
  - ✓ increases vocabulary of designers
- *Problem*
  - ✓ intent, context, when to apply
- *Solution*
  - ✓ UML-like structure, abstract code
- *Consequences* (results and tradeoffs)

Creational Patterns

Creational Patterns

- Abstract Factory
  - ✓ Factory for building related objects
- Builder
  - ✓ Factory for building complex objects incrementally
- Factory Method
  - ✓ Method in a derived class creates associates
- Prototype
  - ✓ Factory for cloning new instances from a prototype
- Singleton
  - ✓ Factory for a singular (sole) instance

Singleton pattern (creational)

- Ensure that a class has only one instance and
- provide a global point of access to it
  - ✓ *Why not use a global variable?*

Singleton pattern: problem

- Need a single instance of some class for use by many modules (similar to a global variable)
- Lifetime of this instance is for the entire program
- May want to extend this by subclassing later
- Want to do that without changing the client code

https://www.youtube.com/watch?v=CdYY-wPPS3w

Singleton Pattern: Solution

- Make the constructors of the class private
- Store the object created privately
- Provide access to get the instance through a public method
- Can be extended to create a pool of objects
- Use static members and functions

http://www.sourcetricks.com/p/design-patterns-using-c.html#.WP0GlYjyvb2
https://www.youtube.com/watch?v=CdYY-wPPS3w

Singleton Pattern

```
class Singleton{
public:
    static Singleton* getInstance();
private:
    Singleton();
    Singleton(const Singleton&);
    Singleton& operator= (const Singleton&);
    static Singleton* instance;
};
Singleton *p2 = p1->getInstance();
```

Singleton Pattern

```
class Singleton{
public:
   static Singleton& getInstance(); // getter
   // Methods to block
   Singleton(const Singleton& arg)= delete; // Copy const
   Singleton(const MySingleton&& arg) = delete; //Move
   Singleton& operator=(const Singleton& arg) = delete;
   Singleton& operator=(const Singleton&& arg) = delete;
   private:
         Singleton();
         virtual ~Singleton();};
```

Prototype Pattern

- A prototype pattern is used in software development when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects
- Example:
  - the inherent cost of creating a new object in the standard way (e.g., using the new keyword) is prohibitively expensive for a given application

Prototype Pattern

Problem

- ✓ Application "hard wires" the class of object to create in each "new" expression

Prototype Pattern

Solution

- ✓ Declare an abstract base class that specifies a pure virtual "clone" method, and, maintains a dictionary of all "cloneable" concrete derived classes
- ✓ Any class that needs a "*polymorphic constructor*" capability
  - ✓ derives itself from the abstract base class
  - ✓ registers its prototypical instance
  - ✓ implements the clone() operation

Prototype Pattern

```
class Document{                          // Prototype
public:
   virtual Document* clone() const = 0;
   virtual void store() const = 0;
   virtual ~Document() { }
};
class plainDoc : public Document{
public:
   Document* clone() const { return new plainDoc; }
   void store() const { cout << "plainDoc\n";}
};
```

Abstract Factory

- A utility class that creates an instance of several families of classes
- It can also return a factory for a certain group

https://en.wikibooks.org/wiki/C%2B%2B_Programming/Code/Design_Patterns

Abstract Factory

- Problem
  - ✓ We want to decide at run time what object is to be created based on some configuration or application parameter
  - ✓ When we write the code, we do not know what class should be instantiated

Abstract Factory

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

Factory Method

- Define an interface for creating an object, but let subclasses decide which class to instantiate
- Factory Method lets a class defer instantiation to subclasses
- Defining a "virtual" constructor
- The new operator considered harmful

https://sourcemaking.com/design_patterns/factory_method

Factory Method

- Problem
  - ✓ A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation

Factory Method

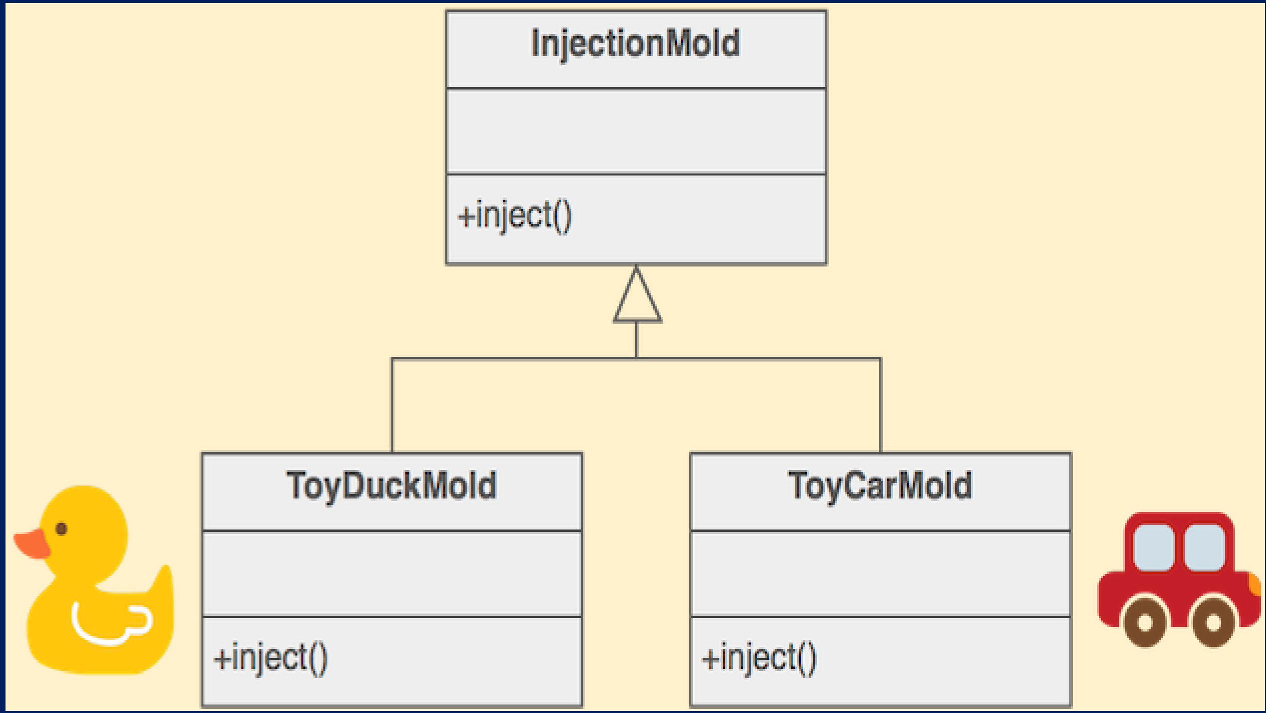
- Solution
  - ✓ *Factory Method* is to creating objects as *Template Method* is to implementing an algorithm
  - ✓ Factory Method makes a design more customizable and only a little more complicated

Factory Method

- Solution
  - ✓ *Factory Method* is similar to *Abstract Factory* but without the emphasis on families
  - ✓ Factory Methods are routinely specified by an architectural framework, and then implemented by the user of the framework

Factory Method

- Example
  - ✓ The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate



Factory Method

- Solution
  - ✓ A factory method is a static method of a class that returns an object of that class' type.
  - ✓ Unlike a constructor, the actual object it returns might be an instance of a subclass.
  - ✓ Another advantage of a factory method is that it can return existing instances multiple times.

Builder

- Is used to separate the construction of a complex object from its representation so that the same construction process can create different objects representations

https://en.wikibooks.org/wiki/C%2B%2B_Programming/Code/Design_Patterns

Builder

Problem

- ✓ We want to construct a complex object
- ✓ We do not want to have a complex constructor member or one that would need many arguments

Builder

Solution

- ✓ Define an intermediate object whose member functions define the desired object part by part before the object is available to the client
- ✓ Builder lets us defer the construction of the object until all the options for creation have been specified

Structural Patterns

Structural Patterns

- All about Class and Object composition
- Use inheritance to compose interfaces
- Define ways to compose objects to obtain new functionality

Structural Patterns

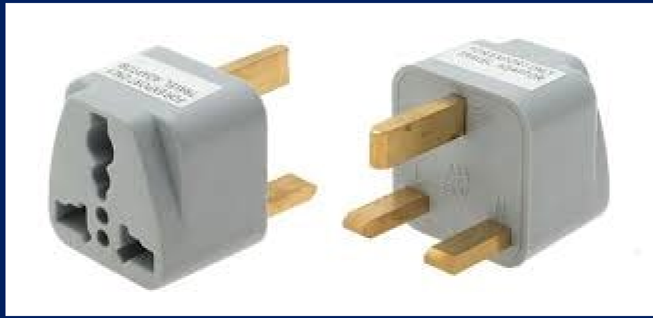
- Adapter
  - ✓ Match interfaces of different classes
- Bridge
  - ✓ Separates an object’s interface from its implementation
- Composite
  - ✓ A tree structure of simple and composite objects
- Decorator
  - ✓ Add responsibilities to objects dynamically

Structural Patterns

- Facade
  - ✓ A single class that represents an entire subsystem
- Flyweight
  - ✓ A fine-grained instance used for efficient sharing
- Private Class Data
  - ✓ Restricts accessor/mutator access
- Proxy
  - ✓ An object representing another object

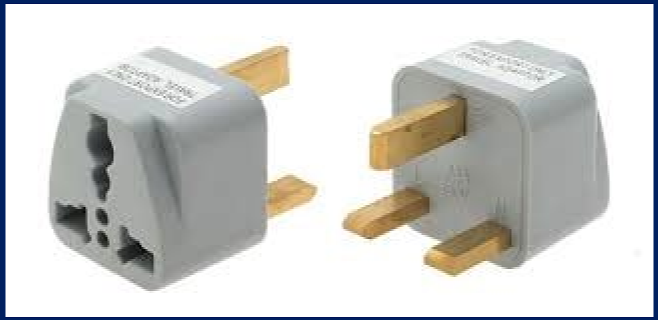
Adapter

- Convert the interface of a class into another interface expected by the client
- Adapter lets classes work together that couldn't otherwise because of incompatible interfaces
- Used to provide a new interface to existing legacy components
  - ✓ Interface engineering, reengineering
- Also known as a wrapper



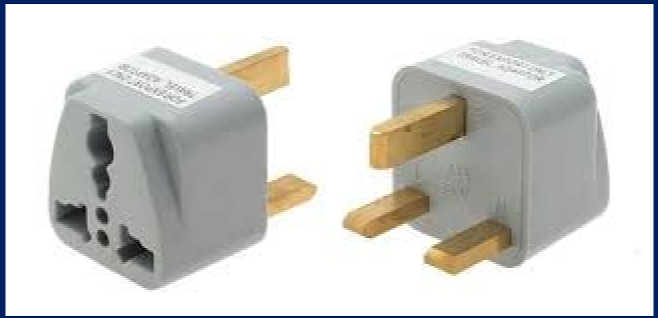
Adapter: Problem

- We need an interface of abstract class *Base*
- We already have another class *Done*, that implements the needed behavior, but with the wrong interface



Adapter: Solution

1. Derive new class Adapter from Base
2. Add Done object as a member of Adapter class
3. Use thin layer of code around Done object to implement the desired Base interface

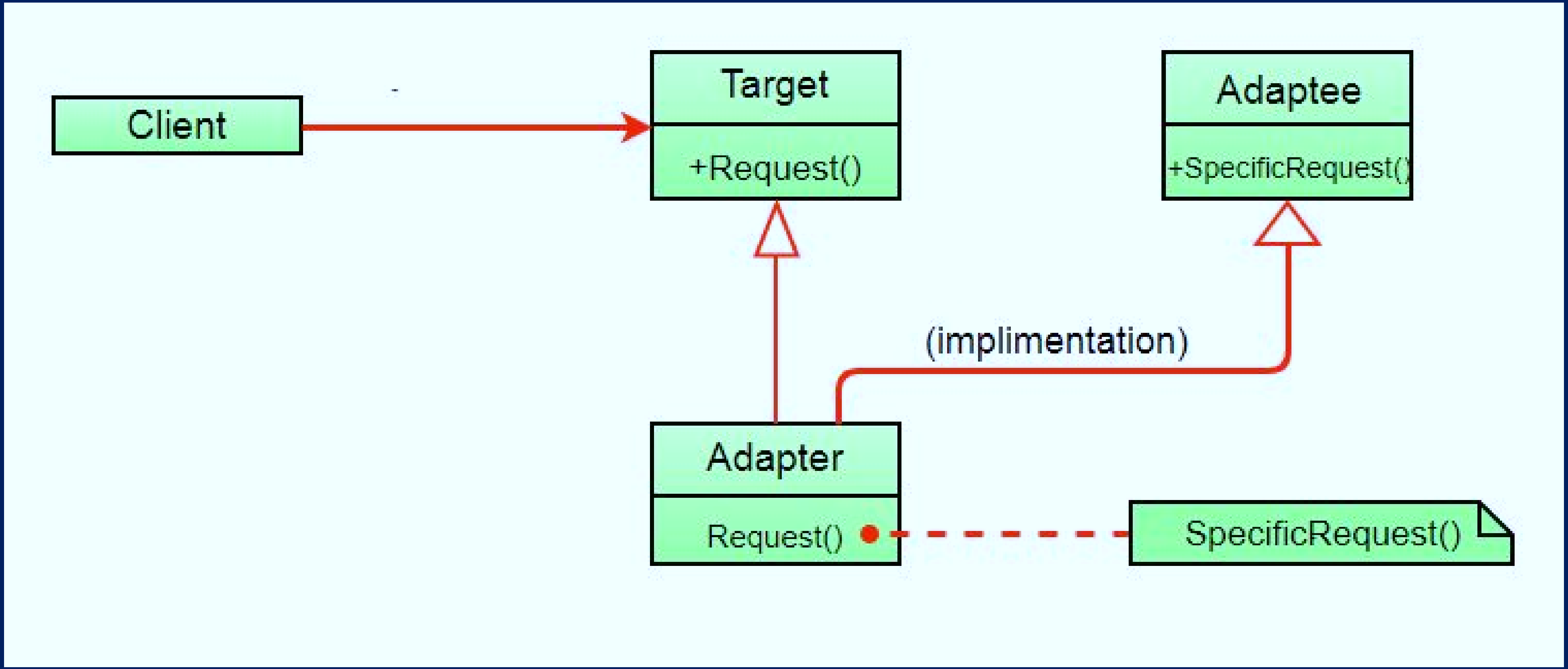


Adapter

1. Adapter pattern relies on object composition
2. Client calls operation on Adapter object
3. Adapter calls Adaptee to carry out the operation
4. In STL, stack adapted from vector:

   When stack executes push(), underlying vector does vector::push_back()

Adapter: Structure



Adapter

- One way to implement `class Adapter` :

```
class Adapter: public Base {     //class Adapter
public:
        // Base functions here
private:
        Done m_done;
};
```

Adapter

- Another way to do that:

```
class Adapter: public Base, private Done {};
```

- Or build a two-way adapter, using multiple inheritance where both bases are public:

```
class Adapter: public Base, public Done {};
```

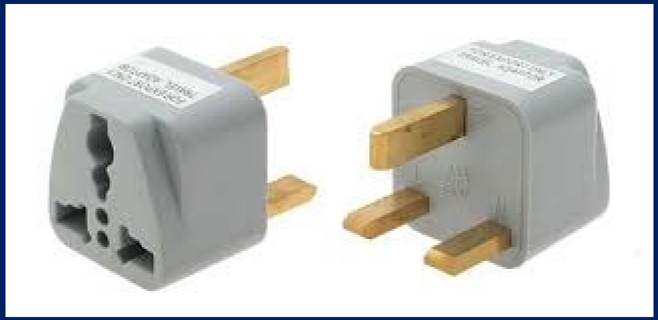
Adapter: abstract adapter

- Abstract adapter adds convenience of Base interface to an existing Done implementation:

```
class Adapter: public Base {     //class Adapter
public:
    Adapter( Done* dp );
    // Base functions here
private:
    Done* m_done;
};
```

Adapter

- The adapter pattern is useful to expose a different interface for an existing API to allow it to work with other code
- By using adapter pattern, we can take heterogeneous interfaces, and transform them to provide consistent API



Bridge

- Has a structure similar to an object adapter
- Bridge has a different intent
  - It is meant to *separate* an interface from its implementation so that they can be varied easily and independently
  - An adapter is meant to *change the interface* of an *existing* object

Bridge: Problem

- An abstraction and its implementation should be defined and extended independently from each other.
- A compile-time binding between an abstraction and its implementation should be avoided so that an implementation can be selected at run-time

Bridge: Solution

- There are 2 parts in Bridge design pattern :
  1. Abstraction
  2. Implementation/Implementor
- Separate an abstraction (Abstraction) from its implementation (Implementor) by putting them in separate class hierarchies.
- Implement the Abstraction in terms of (by delegating to) an Implementor object

Bridge

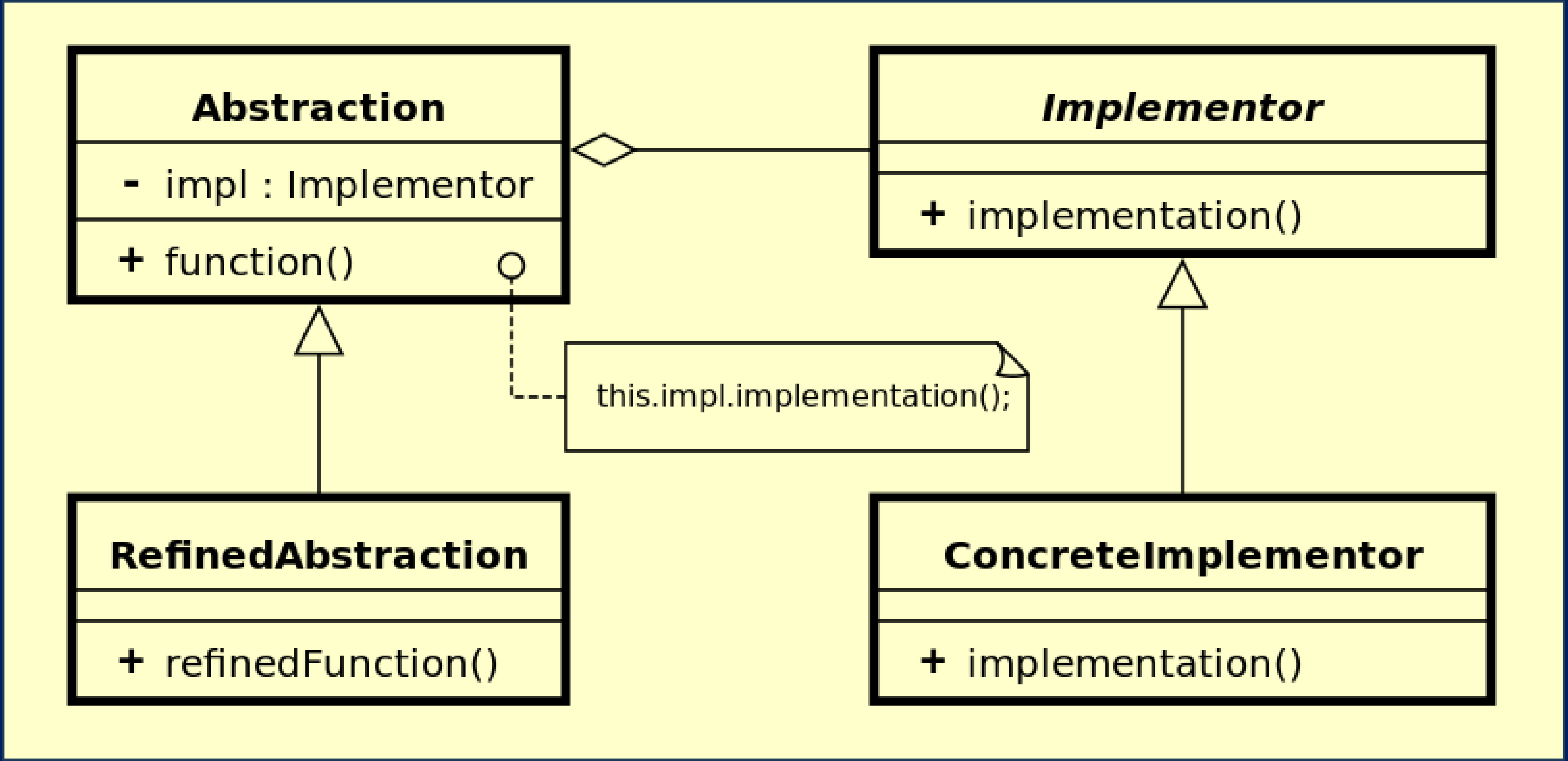
- Allows the Abstraction and the Implementation to be developed independently
  - The client code can access only the Abstraction
- Abstraction - an interface or abstract class

  Implementor is also an interface or abstract class

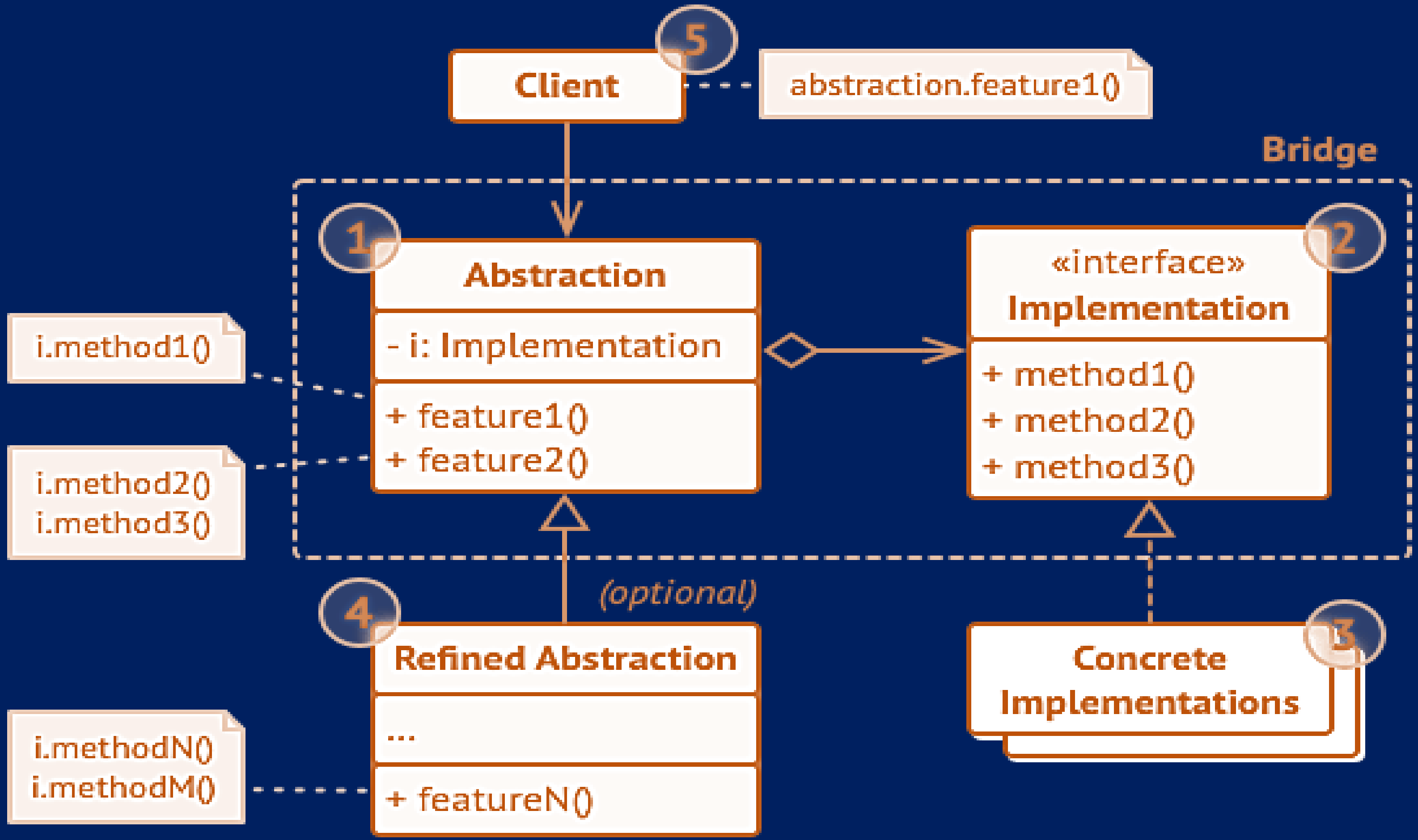
Bridge

- Abstraction contains a reference to the Implementor
  - Children of the abstraction are referred to as *refined abstractions*
  - Children of the implementor *are concrete implementors*
  - Can change the abstraction’s implementor at run-time
  - Changes to the implementor do not affect client code.
- It increases the loose coupling between class abstraction and it’s implementation

Bridge: Structure



Bridge: Structure



Bridge

- *Abstraction* – core of the bridge design pattern and defines the crux. Contains a reference to the Implementer.
- *Refined Abstraction* – Extends the abstraction takes the finer detail one level below.
  - Hides the finer elements from implemetors

Bridge

- *Implementer* – defines the interface for implementation classes
  - This interface does not need to correspond directly to the abstraction interface and can be very different
  - Abstraction imp provides an implementation in terms of operations provided by Implementer interface
- *Concrete Implementation* – platform-specific code, implements the Implementor interface

Bridge

- Client is only interested in working with the abstraction.
- It's the client's job to link the abstraction object with one of the implementation objects

http://www.vishalchovatiya.com/bridge-design-pattern-in-modern-cpp/

Bridge: example

```
struct Person {              // "Person.h"
   /* PIMPL ----------------------------------- */
   class PersonImpl;
   unique_ptr<PersonImpl>  m_impl; // bridge-not necessarily inner
                                   // class
   /* ---------------------------------------- */
    string m_name;
    Person();
    ~Person();
    void greet();
private:
    // secret data members or methods are in `PersonImpl` not here
    // as we are going to expose this class to client };
```

Bridge: example

```
#include "Person.h"

/* PIMPL Implementation ------------------------------- */
struct Person::PersonImpl {
    void greet(Person *p) {
        cout << "hello " << p->name.c_str() << endl;
    }
};
/* -------------------------------------------------- */

Person::Person() : m_impl(new PersonImpl) { }
Person::~Person() { delete m_impl; }
void Person::greet() { m_impl->greet(this); }
```

Bridge: Benefits

1. Provides flexibility to develop interface and the implementation independently
   - And the client/API-user code can access only the abstraction part
2. Preserves the Open-Closed Principle
3. Can hide the implementation details from the client

Bridge: Benefits

4. “prefer composition over inheritance”
5. A compile-time binding between an abstraction and its implementation should be avoided
   - an implementation can select at run-time

Proxy

Acts as convenient surrogate or placeholder for another object

- Remote Proxy
  - ✓ local representative for object in a different address space
- Virtual Proxy
  - ✓ represent large object that should be loaded on demand
- Protected Proxy
  - ✓ protect access to the original object

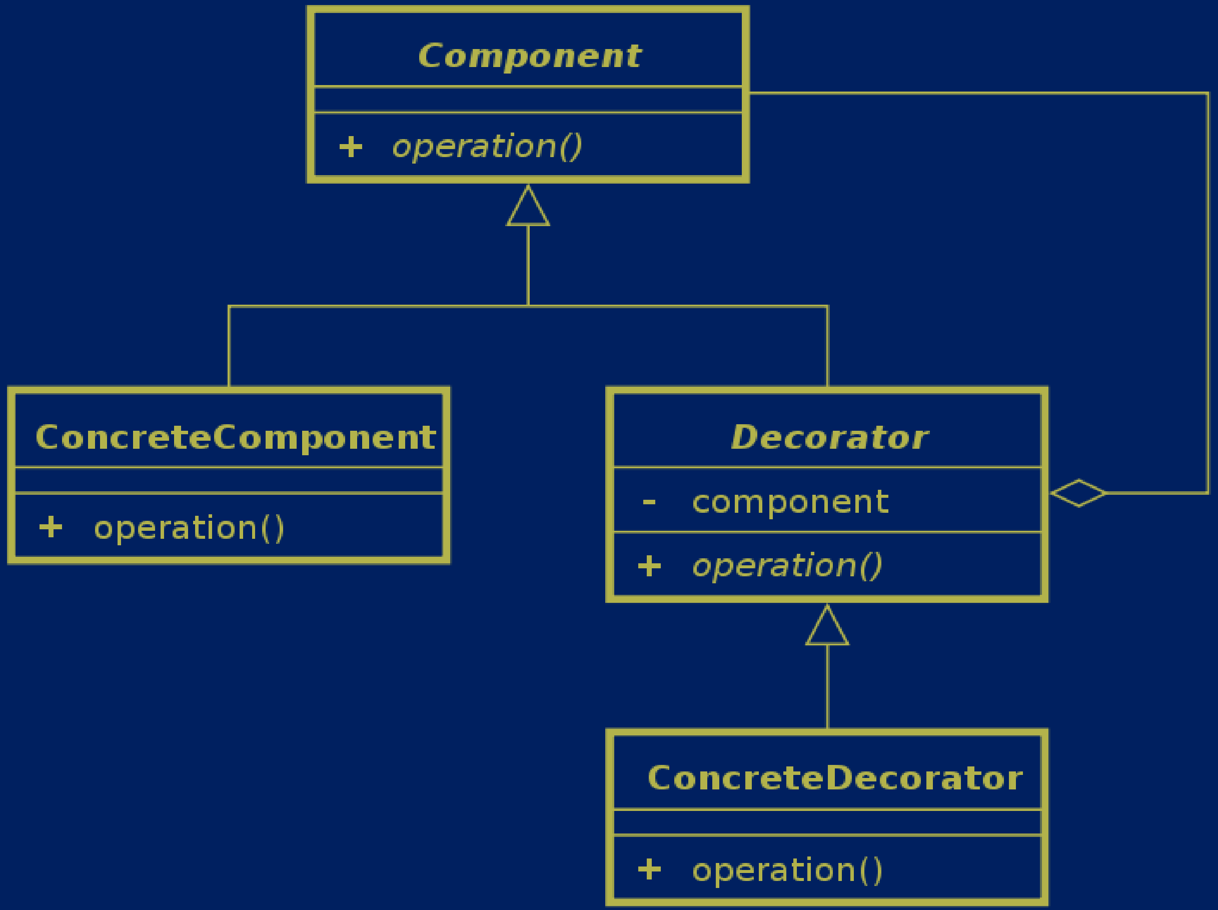
Composite

- Objects that can serve as containers, and can hold other objects like themselves
- An object that is either an individual item or a collection of many items
  - ✓ composite objects can be composed of individual items or of other composites
  - ✓ recursive definition: objects that can hold themselves
  - ✓ often leads to a tree structure of leaves and nodes

Decorator

- Decorator pattern allows a user to add new functionality to an existing object without altering its structure.
- This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class
- This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact

Decorator



Component
+ operation()
ConcreteComponent
+ operation()
Decorator
- component
+ operation()
ConcreteDecorator
+ operation()

Decorator

```cpp
class Control {
 public:
    virtual void draw() = 0;
};
```

```cpp
class ScrollPane: public
 Control { public:
    ScrollPane(Control*
        decorated):
        decorated(decorated)
    {
    }

    virtual void draw() {
        // TODO draw
        scrollbars
        decorated->draw();
    }
 private:
    Control* decorated;
};
```

Decorator: Pros and Cons

Pros

- The decorators can be arbitrarily plugged on run time on top of each other.
- Each decorator can implement a behavior variant and follow the single responsibility principle

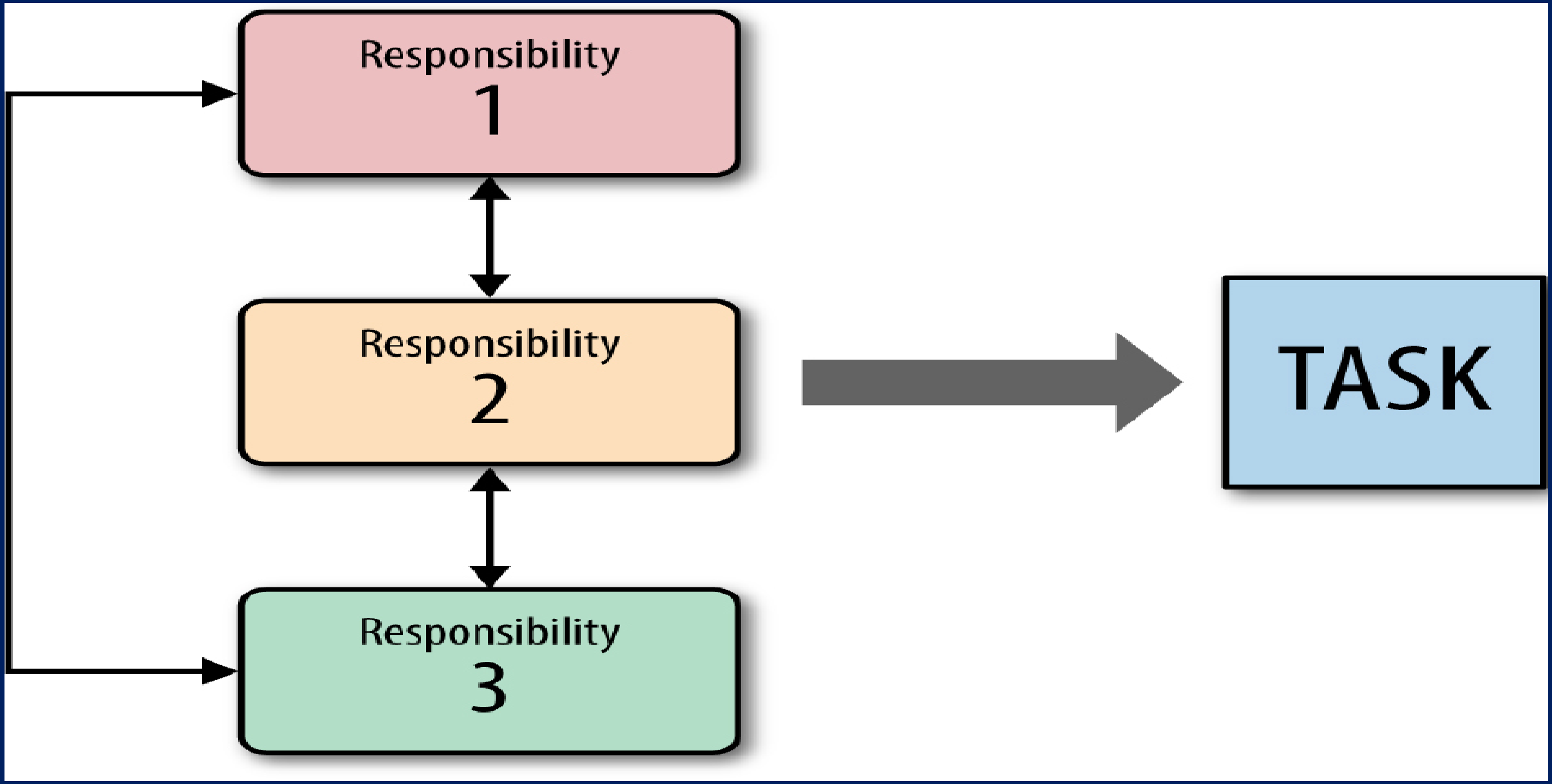
Decorator: Pros and Cons

Cons

- Due to these delegated member function calls, the control flow is difficult to follow.
- The delegated member function call may affect the performance of the program.
- It is pretty complicated to remove a decorator out of a stack of decorators.

Behavioral Patterns

Behavioral Patterns



Behavioral Patterns

All about Class's objects communication

- Chain of responsibility
  - ✓ A way of passing a request between a chain of objects
- Command
  - ✓ Encapsulate a command request as an object
- Interpreter
  - ✓ A way to include language elements in a program
- Iterator
  - ✓ Sequentially access the elements of a collection

Behavioral Patterns

- Mediator
  - ✓ Defines simplified communication between classes
- Memento
  - ✓ Capture and restore an object's internal state
- Null Object
  - ✓ Designed to act as a default value of an object
- Observer
  - ✓ A way of notifying change to a number of classes

Behavioral Patterns

- State
  - ✓ Alter an object's behavior when its state changes
- Strategy
  - Encapsulates an algorithm inside a class
- Template method
  - ✓ Defer the exact steps of an algorithm to a subclass
- Visitor
  - ✓ Defines a new operation to a class without change

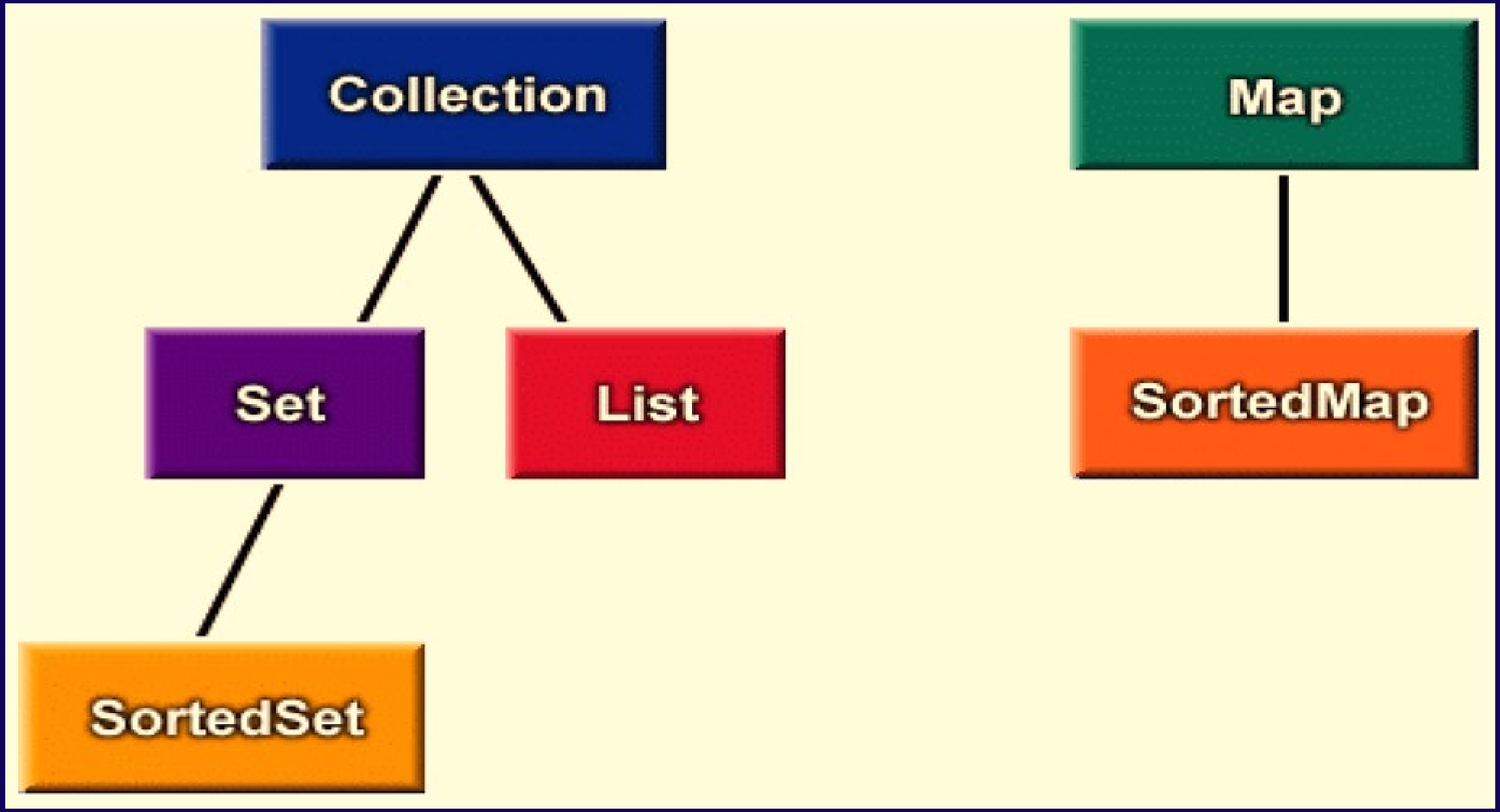
Iterator Pattern

Intent

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- The standard library abstraction that makes it possible to decouple collection classes and algorithms
- Promote to "full object status" the traversal of a collection
- Polymorphic traversal

Iterator Pattern

objects that traverse collections



Iterator Pattern

Problem

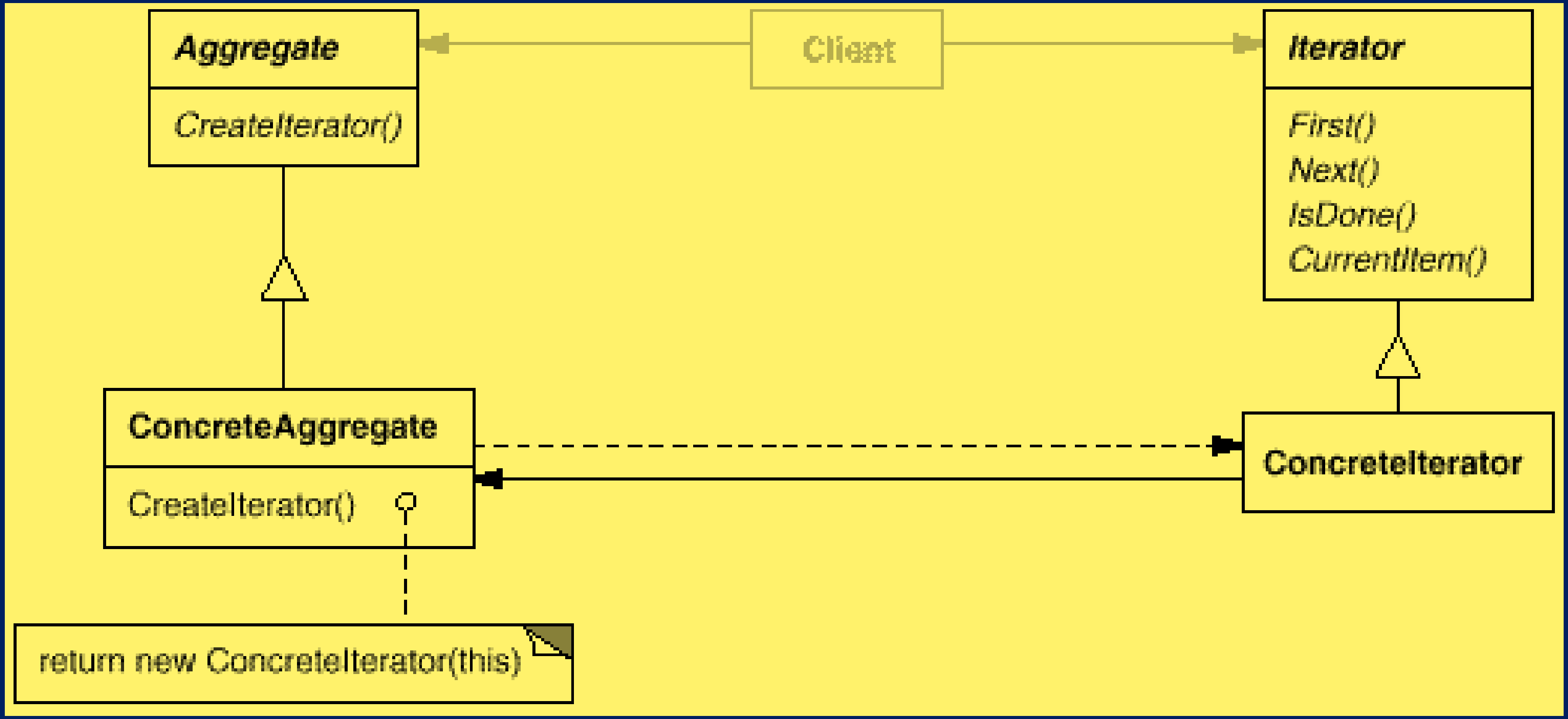
- Need to "abstract" the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each transparently

Iterator Pattern

Applicability

- require multiple traversal algorithms over a container
- require a uniform traversal interface over different containers
- when container classes & traversal algorithm must vary independently

Iterator : Structure



Iterator Pattern

- Aggregate
  - ✓ defines an interface for creating an *Iterator* object.
- ConcreteAggregate
  - ✓ implements the *Iterator* creation and returns instance of the proper *ConcreteIterator*
- Iterator
  - ✓ defines an interface for element traversal and access
- ConcreteIterator
  - ✓ implements the Iterator interface

Iterator Pattern

Check list

- Add a `create_iterator()` method to the "collection" class, and grant the "*iterator*" class privileged access
- Design an "*iterator*" class that can encapsulate traversal of the "collection" class
- Clients ask the collection object to create an iterator object.
- Clients use the `first()`, `is_done()`, `next()`, and `current_item()` protocol to access the elements of the collection class

https://sourcemaking.com/design_patterns/iterator/cpp/1

Memento Pattern

The Memento design pattern provides an ability to restore (rollback) an object to its previous state

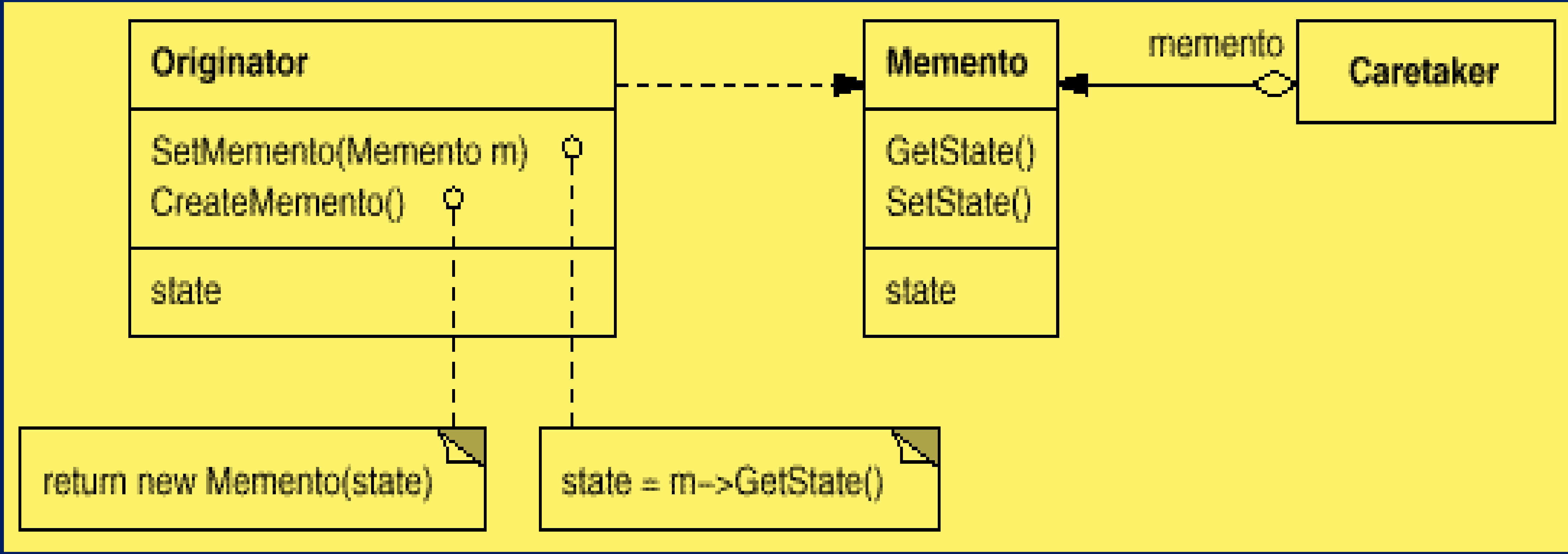
Memento: Problem

- An object of class Item has considerable hidden state information
- The client wants to:
  - ✓ remember checkpoint state
  - ✓ modify the state
  - ✓ optionally restore the state to the checkpoint state
- Minimal impact on the original Item class interface

Memento

- *Originator* – stores internal state of the *Originator* object
- *Caretaker* –knows why and when the *Originator* needs to save and restore itself
  - ✓ implements the "undo" mechanism
  - ✓ takes responsibility to store *Memento* objects in memory
  - ✓ Note: the *Caretaker* never operates on, or examines the contents of a *Memento*
- *Memento* – stores internal state of the *Originator* object

Memento : Structure



Memento : example

```
class Memento {
public:
      virtual ~Memento();
private:
      friend class Item;
      Memento();
};
class Item {
public:
      Memento* create_memento();
      void restore_memento( Memento* );
};
```

Memento: Check List

1. Identify the roles of “caretaker” and “originator”
2. Create a Memento class and declare the originator a friend
3. Caretaker knows when to "check point" the originator
4. Originator creates a Memento and copies its state to that Memento

Memento: Check List

5. Caretaker holds on to (but cannot peek into) the Memento.
6. Caretaker knows when to "roll back" the originator
7. Originator reinstates itself using the saved state in the Memento

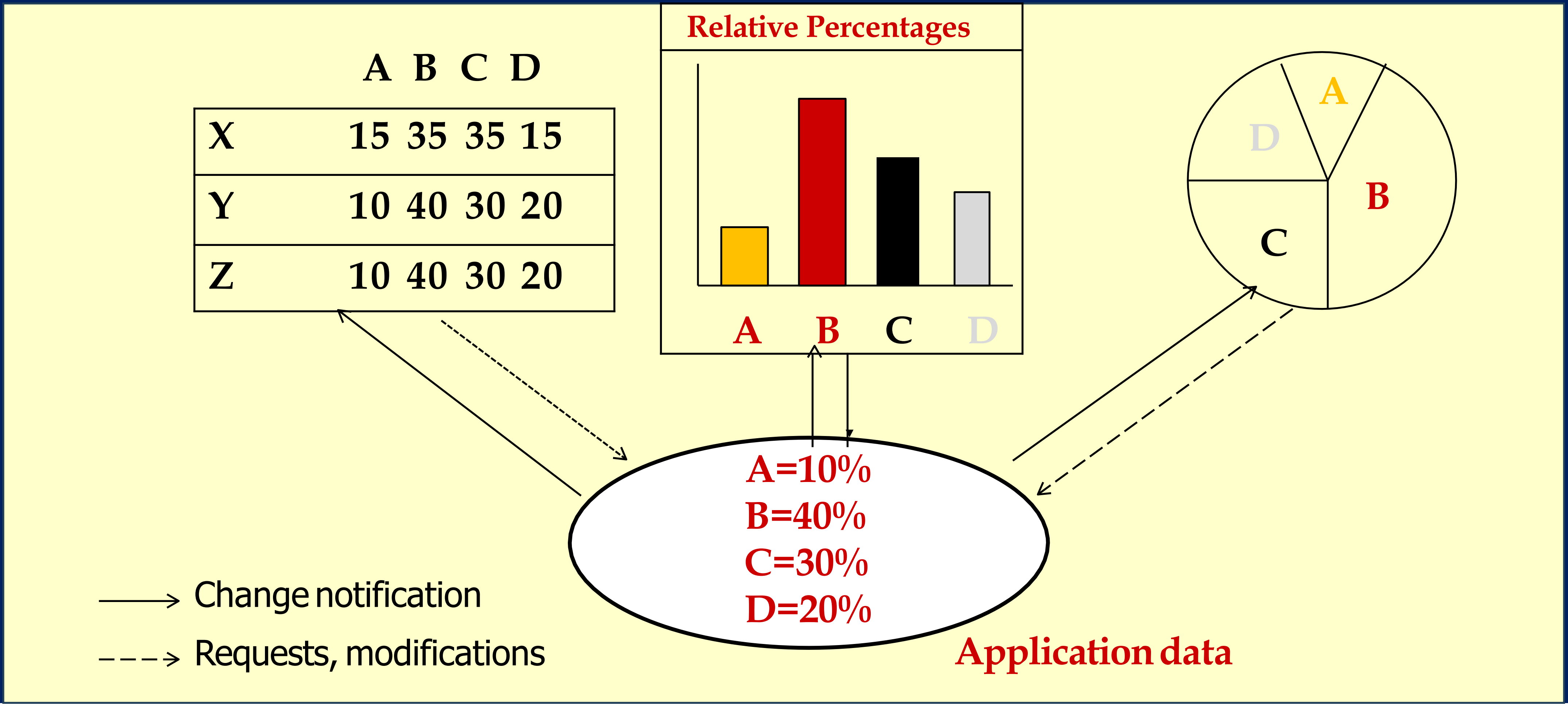
Memento: Pros and Cons

- ✓ You can produce snapshots of the object’s state without violating its encapsulation.
- ✓ You can simplify the originator’s code by letting the caretaker maintain the history of the originator’s state.

- ✖ The app might consume lots of RAM if clients create mementos too often
- ✖ Caretakers should track the originator’s lifecycle to be able to destroy obsolete mementos.
- ✖ Most dynamic programming languages, such as PHP, Python and JavaScript, can’t guarantee that the state within the memento stays untouched

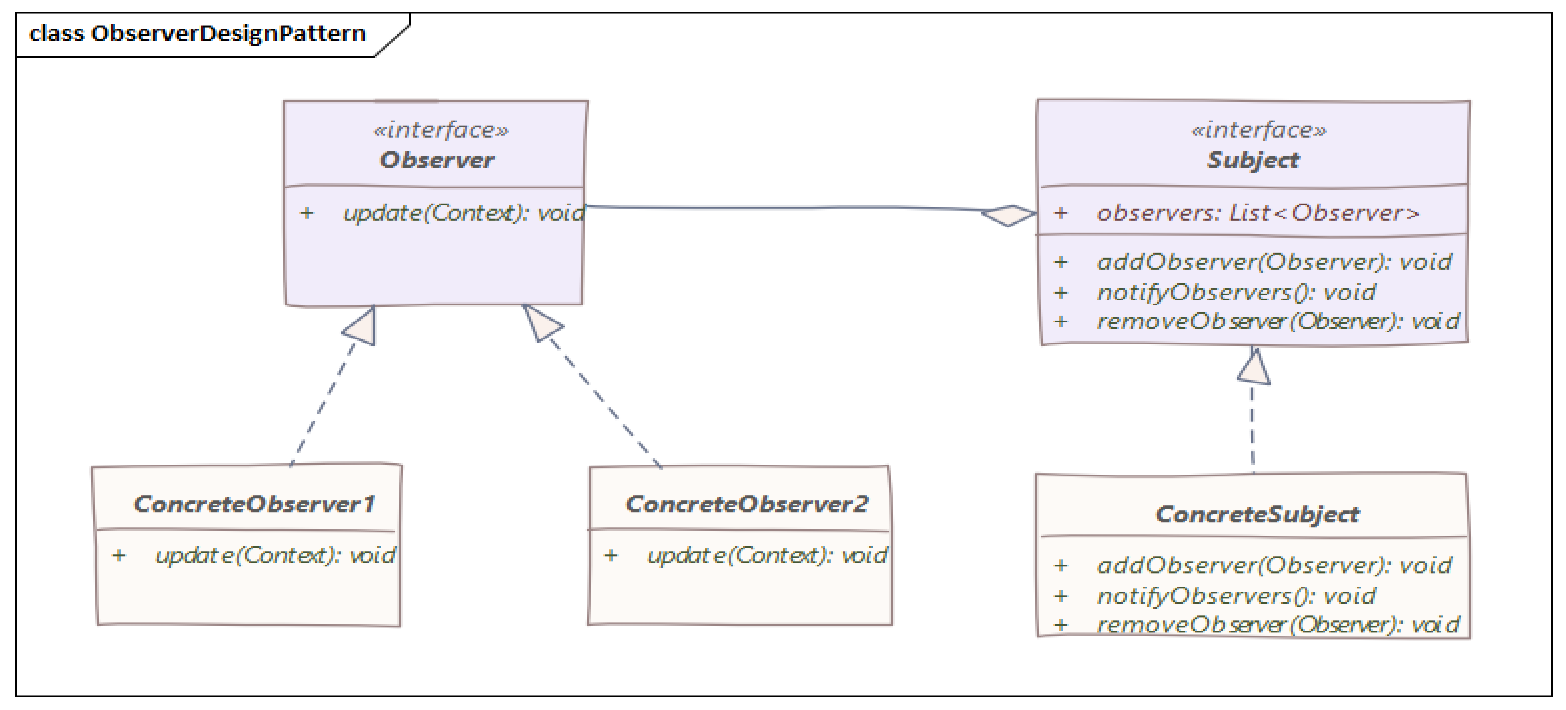
Observer

Multiple displays of same application data



Observer: intend

- Define a 1-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically



Observer: intend

Subject

- Has a list of observers
- Methods for attaching / detaching an observer

Observer

- An updating interface for objects that get notified of changes in a subject

ConcreteSubject

- Stores state of interest to observers
- Sends notification when state changes

Observer: example

ConcreteObserver

- Implements updating interface

```
class Observer {
 public:
  void update(Observable sub, StateInfo arg) = 0;
}
```

```
class PieChartView: public Observer {
  void update(Observable sub, StateInfo
  arg) {
   // Repaint the pie chart
  }
}
```

```
class Observable {
 public:
  void attach(Observer o) {}
  void detach(Observer o) {}
  void notify(StateInfo arg) {}
  …
  bool hasChanged() {}
}
```

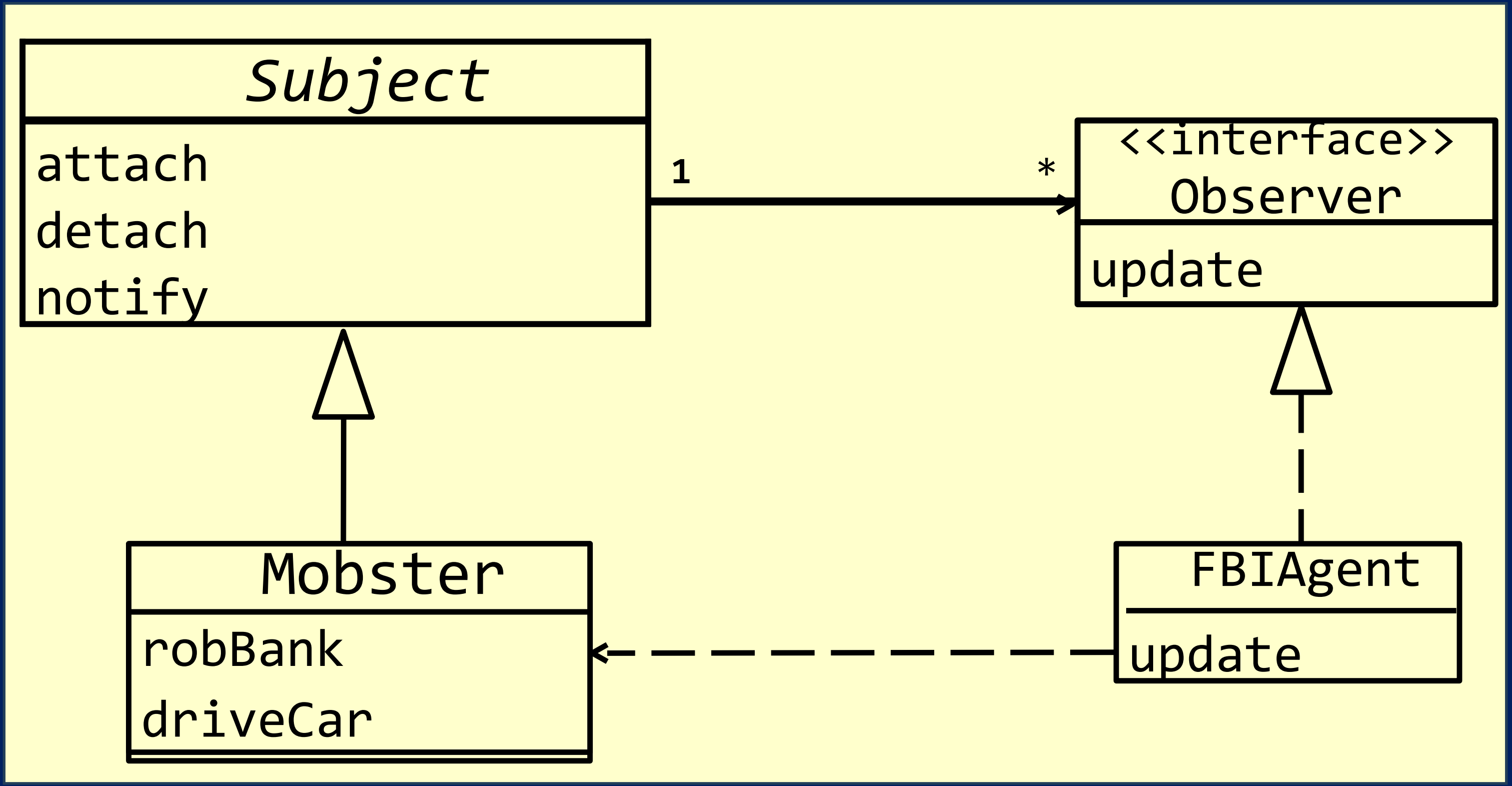
```
class StatsTable: public Observable {
  private:
    vector<Observer*> observers; StateInfo state;
  public:
    bool hasChanged() {
      // Override to decide when it has changed
    }
}
```

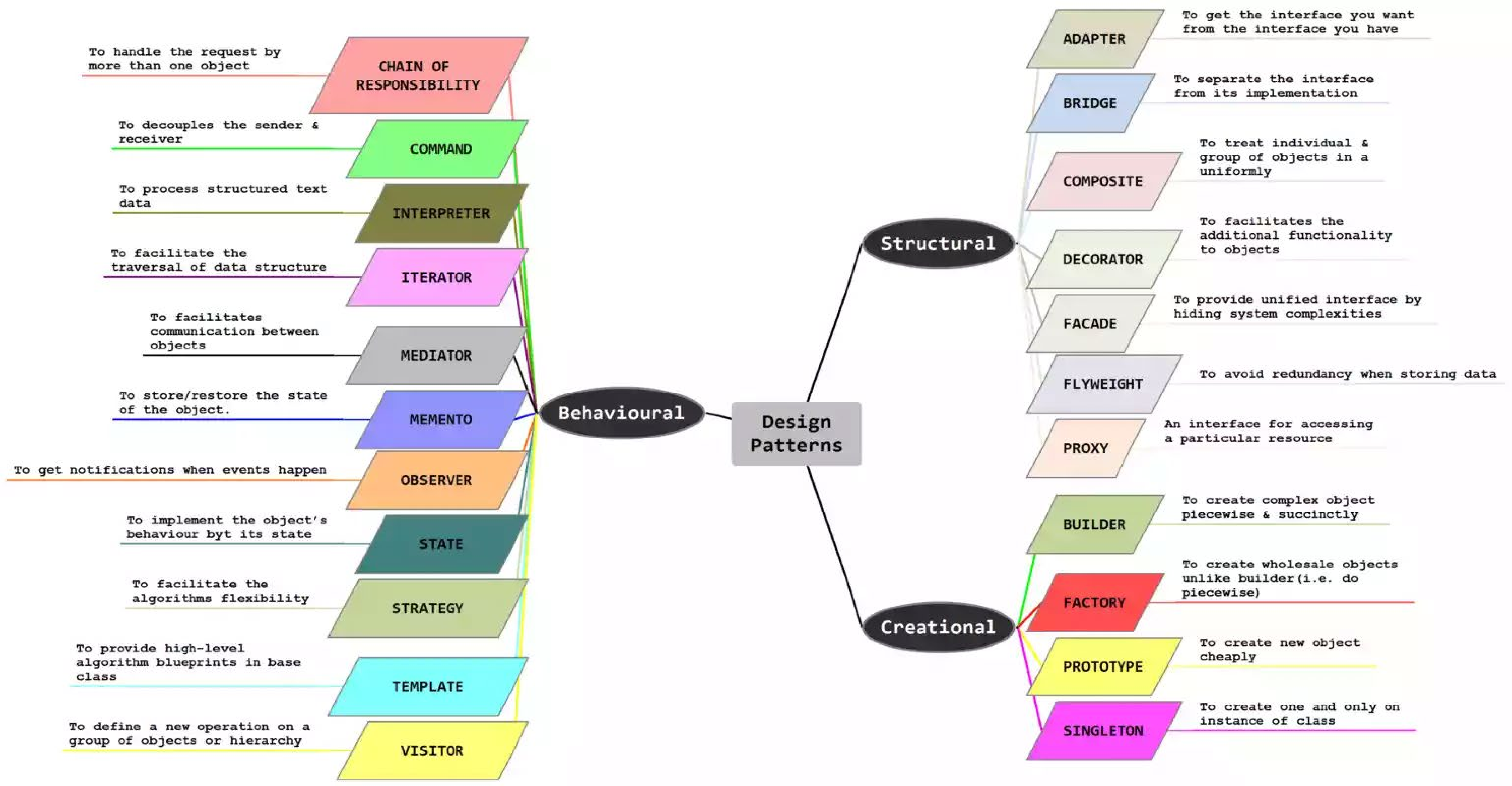
Observer: cycles of dependence

- If package A depends on package B, then you cannot run the tests for A unless you also have B
  - "A depends on B" means that you cannot use A unless you have B
  - "Package A depends on package B" means that something in A depends on something in B
- If package A depends on package B, then package B should NOT depend on package A
  - If classes C and D both depend on each other, put them in the same package

Observer: eliminating dependence

- `Observer` pattern eliminate some dependence
- Class `FBIAgent` probably depends on class `Mobster`, but `Mobster` does not depend on `FBIAgent`




Design Patterns
Behavioural
CHAIN OF RESPONSIBILITY
To handle the request by more than one object
COMMAND
To decouples the sender & receiver
INTERPRETER
To process structured text data
ITERATOR
To facilitate the traversal of data structure
MEDIATOR
To facilitates communication between objects
MEMENTO
To store/restore the state of the object.
OBSERVER
To get notifications when events happen
STATE
To implement the object's behaviour byt its state
STRATEGY
To facilitate the algorithms flexibility
TEMPLATE
To provide high-level algorithm blueprints in base class
VISITOR
To define a new operation on a group of objects or hierarchy
Structural
ADAPTER
To get the interface you want from the interface you have
BRIDGE
To separate the interface from its implementation
COMPOSITE
To treat individual & group of objects in a uniformly
DECORATOR
To facilitates the additional functionality to objects
FACADE
To provide unified interface by hiding system complexities
FLYWEIGHT
To avoid redundancy when storing data
PROXY
An interface for accessing a particular resource
Creational
BUILDER
To create complex object piecewise & succinctly
FACTORY
To create wholesale objects unlike builder(i.e. do piecewise)
PROTOTYPE
To create new object cheaply
SINGLETON
To create one and only on instance of class

Part III

Model View Controller Pattern

Model View Controller Pattern

Architectural pattern: MVC

Motivation

- Mixing the game representation, the game rules, and the drawing code all together is likely to create a mess
- Hard to tell what is happening at any given moment
- Difficult to maintain
- Extremely difficult to add new things
  - New kinds of data objects
  - New ways to represent the data

Architectural pattern: MVC

Separate the problem into three interrelated, but distinct, components

- Model
- View
- Controller

Architectural pattern: MVC

Model

- Represents the data being modelled and the rules for maintaining it
  - Ex: A `Universe` object that owns `Planets` and enforces the physics rules

View

- Handles presenting the data (such as through a GUI)
  - Ex: `Drawer` that knows how to display the `Universe` in the window

Architectural pattern: MVC

Controller

- Responds to input and directs changes to the model
  - Ex: Reacts to timer and key events and directs the `Universe` when to update and by how much time

The components often communicate through the Observer pattern

- Ex: The model notifies all views when its data has changed
- Ex: The view notifies the controller about button clicks

Model View Controller

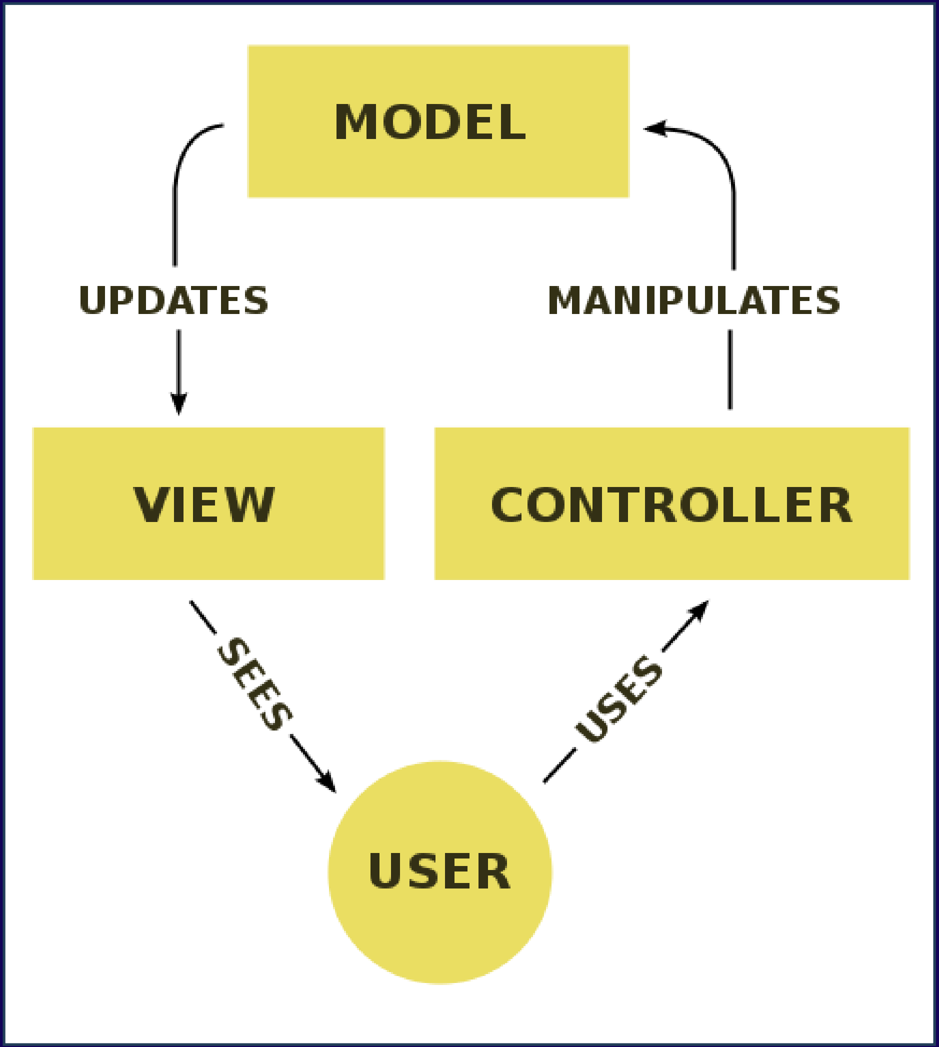


Image:
https://en.wikipedia.org/wiki/Model-view-controller