CSC 133
Object-Oriented Computer Graphics Programming

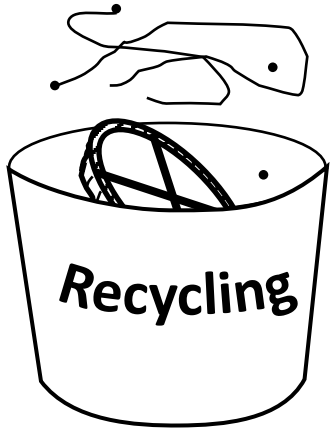
Design Patterns II

Dr. Kin Chung Kwan

Spring 2023

Computer Science Department
California State University, Sacramento

SACRAMENTO STATE



Design Patterns

Common Design Patterns

Creational:

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

Structural:

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

Behavioral:

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

The Strategy

The Strategy Pattern

Motivation

- A variety of algorithms exists to perform a particular operation
- The client needs to be able to select/change the choice of algorithm *at run-time.*

The Strategy Pattern (cont.)

Examples where different *strategies* might be used:

- Save a file in different formats (plain text, PDF, PostScript…)
- Compress a file using different compression algorithms
- Sort data using different sorting algorithms
- Capture video data using different encoding algorithms
- Plot the same data in different forms (bar graph, table, …)
- Have a game's non-player character (NPC) change its AI
- Arrange components in an on-screen window using different layout algorithms

Example

NPC have different strategy AI

- Different `attack()` type of NPC in Game

```
magic();
debuff();
fight();
```

Naïve Coding

```
public class Wizard {
     Wizard() { }
     void fight(){ … }
     void magic(){ … }
     void debuff(){ … }
}
```

Naïve Coding in Client

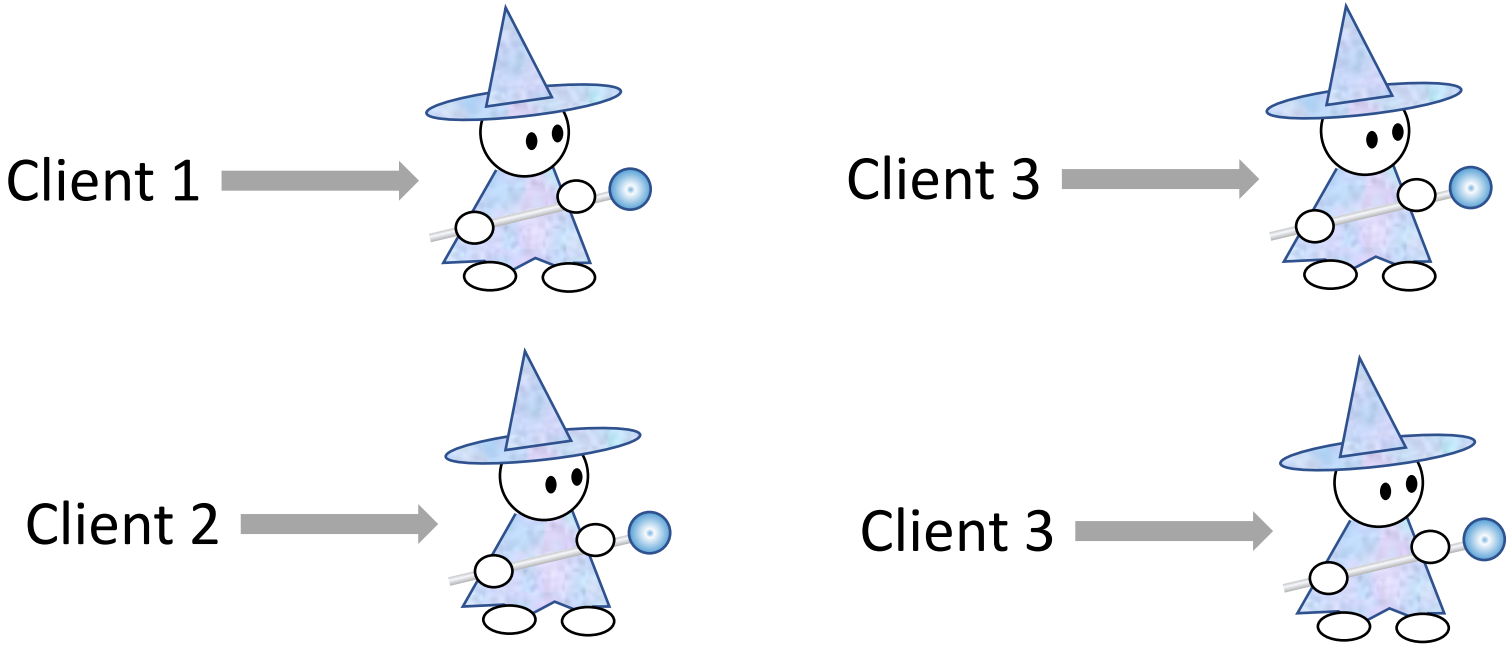
If we put the code in the client (outside NPC)

```
void attack() {

  switch (strategy) {
     case FIGHT:   wizard.fight();               break;
     case FIRE:    wizard.fireWeapon();          break;
     case DEBUFF:  wizard.castDebuffSpell();     break;
     case MAGIC:   wizard.castMagicSpell();      break;
  }
}
```

- Problem with this approach?
  - Adding a new plan requires changing the client!

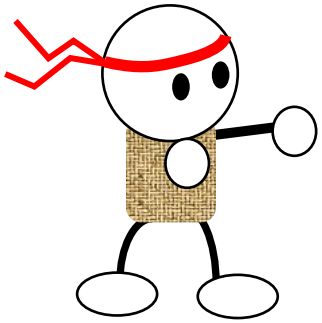
So?

- If multiple client is using the same class
  - Need to change many time
  - Just put it into NPC class?



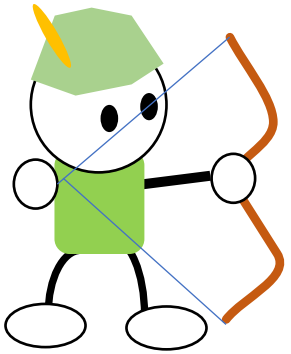
Another Problem

What if multiple classes have the same `attack()` type?



```
fight();
```

```
fight();
```



```
fight();
```

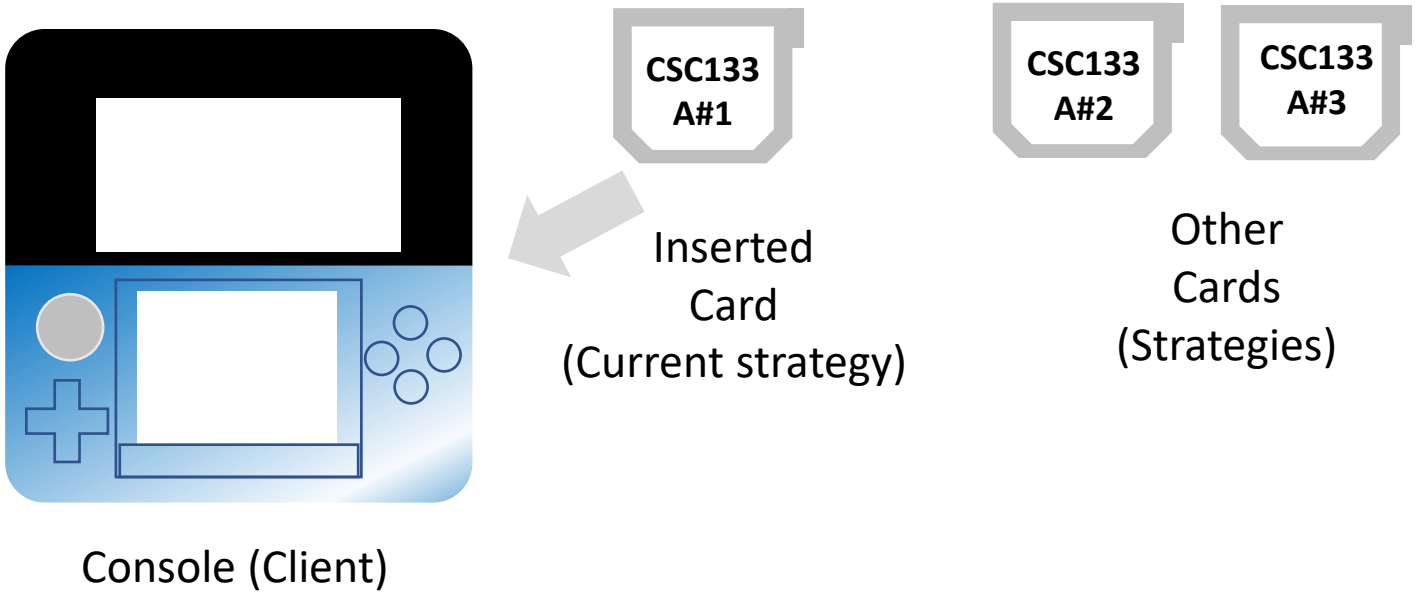
Duplicated code is not desired!

Solution Approach

- Provide various objects that know how to "apply strategy" (e.g., apply fight, fireWeapon, or castMagicSpell strategies)
  - Each in a different way, but with a uniform interface
- The context (e.g., NPC) maintains a "current strategy" object
- Provide a mechanism for the client (e.g., Game) to *change* and *invoke* the current strategy object of a context

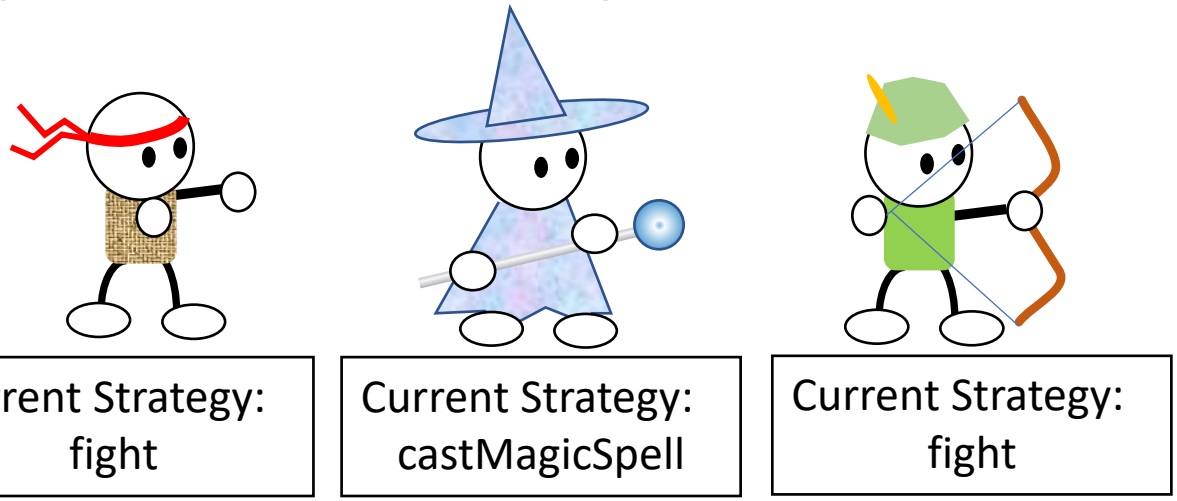
Idea

- Like game card in console
  - Only one card in the console
  - Game depends on the current card



Solution

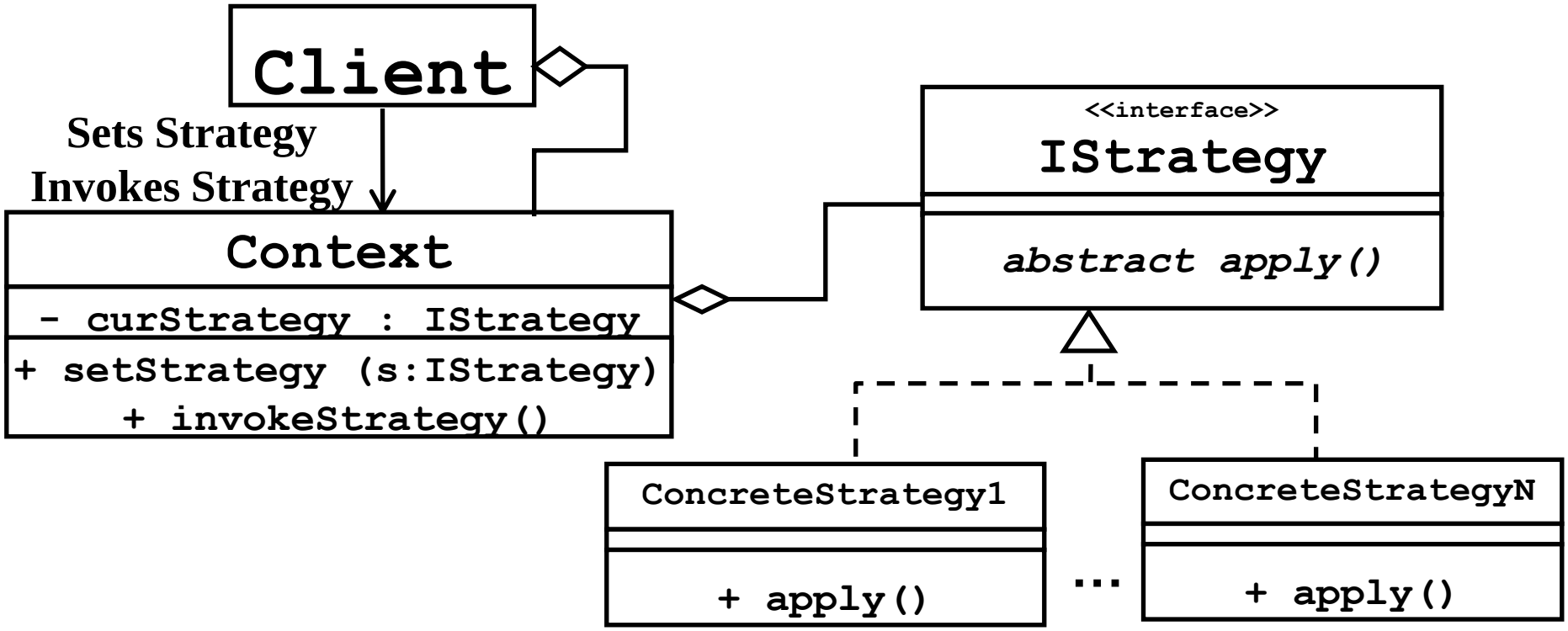
- Every NPC have an object for current strategy



- `For each npc`
  - `Npc.currentStrategy.run()`

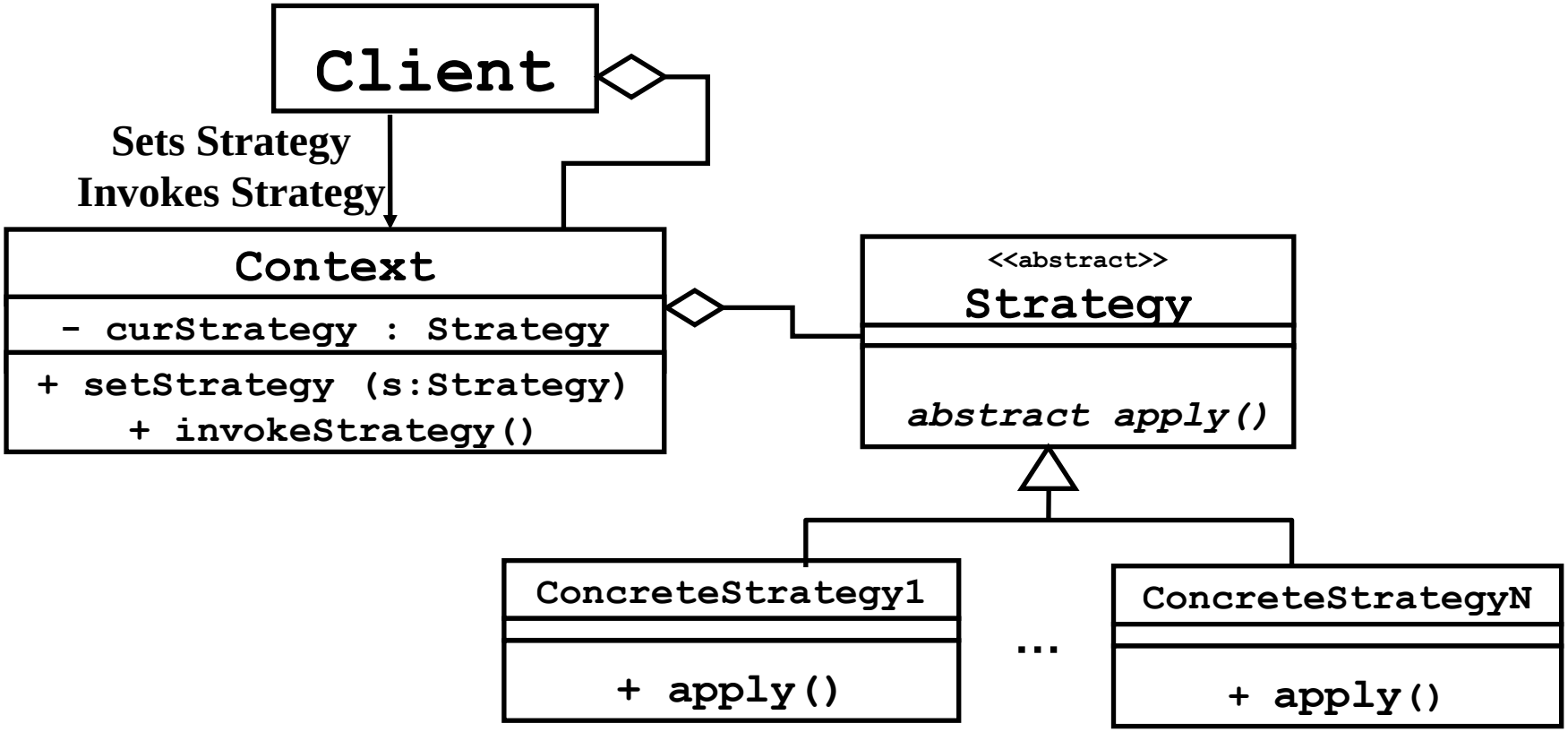
Strategy Pattern Organization

- **Using Interfaces**



Strategy Pattern Organization

- **Using abstract class**



Example Code

```
public class Character {
    private IStrategy curStrategy;
    public void setStrategy(IStrategy s) {
        curStrategy = s;
    }
    public void invokeStrategy() {
        curStrategy.apply();
    }
}
```

```
public class Warrior extends Character {

//code here for Warrior specific methods

}
```

```
public class Shaman extends Character {

//code here for Shaman specific methods

}
```

```
public class Hunter extends Character {
    private int bulletCount ;

    public boolean isOutOfAmmo() {
        if (bulletCount <= 0) return true;
        else return false;
    }
    public void fireWeapon() {
        bulletCount -- ;
    }

//code here for other Hunter specific
//methods

}
```

Example: NPCs in a Game

```
public interface IStrategy {
  public void apply();
}

public class FightStrategy implements IStrategy {
  public void apply() {
        //code here to do "fighting"
  }
}

public class FireWeaponStrategy implements IStrategy {
  private Hunter hunter;
  public FireWeaponStrategy(Hunter h) {
        this.hunter = h;//record the hunter to which this strategy
  applies
  }
  public void apply() {
        //tell the hunter to fire a burst of 10 shots
        for (int i=0; i<10; i++) {
              hunter.fireWeapon();
        }
  }
}

public class CastMagicSpellStrategy implements IStrategy {
  public void apply() {
        //code here to cast a magic spell
  }
}
```

Assigning / Changing Strategies

```
/** This Game class demonstrates the use of the Strategy Design Pattern
 *  by assigning attack response strategies to each of several game characters.
 */
public class Game {
    //the list of non-player characters in the game
    ArrayList<Character> npcList = new ArrayList<Character>();
        public Game() {      //construct some characters, assigning each a starting strategy
        Warrior w1 = new Warrior();
        w1.setStrategy(new FightStrategy());
        npcList.add(w1);
        Hunter h1 = new Hunter();
        h1.setStrategy(new FireWeaponStrategy(h1));
        npcList.add(h1);
        Shaman s1 = new Shaman();
        s1.setStrategy(new CastSpellStrategy());
        npcList.add(s1);
    }
    public void attack() {               //force each character to execute its attack response
        for (Character c : npcList) {
            c.invokeStrategy();
        }
    }
    public void updateCharacters() {     //update any strategies that need changing
        for (Character c : npcList) {
            if(c instanceof Hunter) {
                if ( ((Hunter)c).isOutOfAmmo() ) {
                    //change the character's strategy
                    c.setStrategy(new FightStrategy());
                }
            }
        }
    }
}
```

CN1 Layouts

- Strategy abstract super class:

```
Layout
```

- Client is the `Form`
- Context: `Container` (e.g., `ContentPane` of `Form`)
- Context methods:

```
public void setLayout (Layout lout)
public void revalidate()
```

- Concrete strategies (`extends Layout`):

```
class FlowLayout
class BorderLayout
class GridLayout
...
```

- "Apply" method (declared in the `Layout` super class):

```
abstract void layoutContainer(Container parent)
```

The Proxy

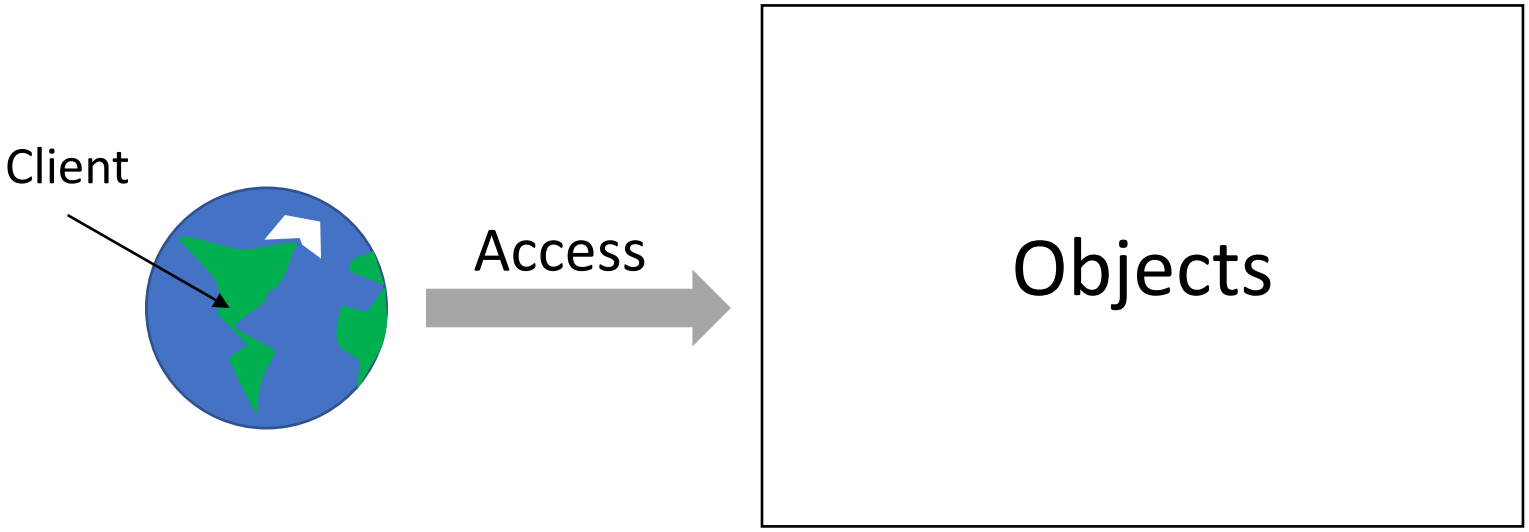
The Proxy Pattern

Motivation:

- Undesirable target object manipulation
  - Access required, but not to all operations
- Expensive target object manipulation
  - Lengthy image load time
  - Significant object creation time
  - Large object size
- Inaccessible target object
  - Resides in a different address space

Problem 1

- You only want one feature, but the object is too large.

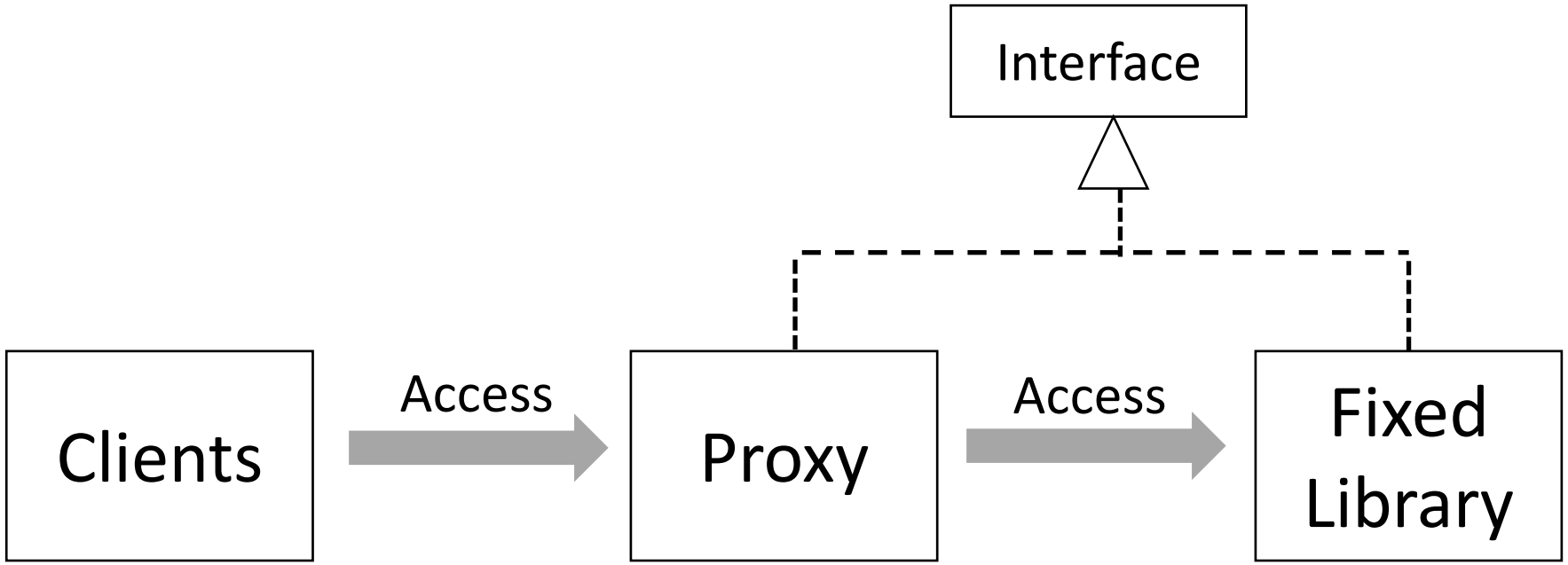


Problem 2

- You want to add new features, but it is not possible

Solution

- Add a proxy with the same interface
  - Clients and Library will not know it

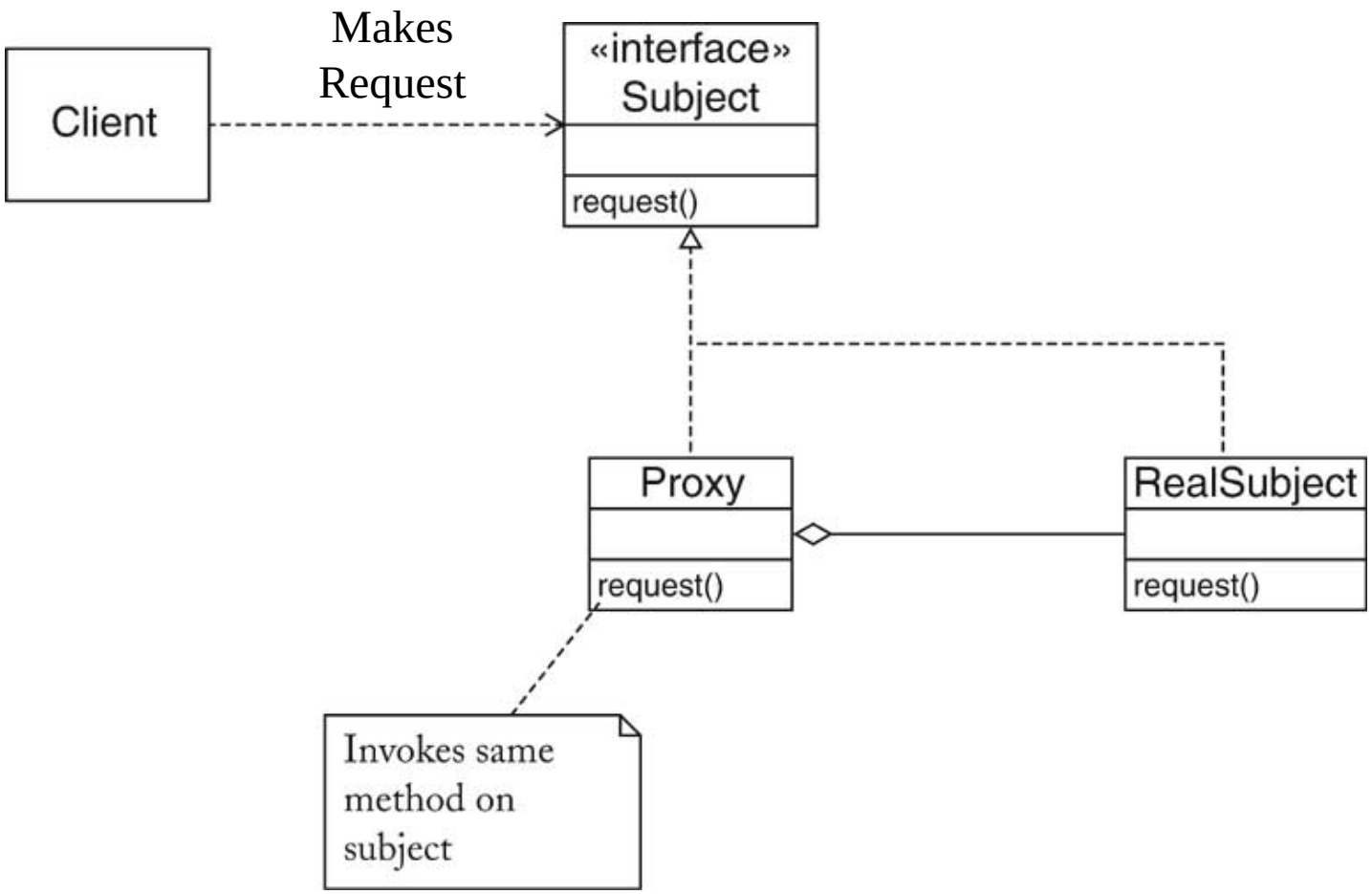


Proxy Types

- Protection
  - Controls access
- Virtual
  - A lightweight stand-in to reduce workload
- Remote
  - Local stand-in for object in another address space
- Logging
- Caching
  - Adding logging/cache to the access

27

Proxy Pattern Organization



Proxy Example

```
interface IGameWorld {
     Iterator getIterator();
     void addGameObject(GameObject o);
     boolean removeGameObject (GameObject o);
}


/**A proxy which prohibits removal of GameWorldObjects from the GameWorld*/
public class GameWorldProxy implements IObservable, IGameWorld {
     private GameWorld realGameWorld ;
     public GameWorldProxy (GameWorld gw)
       { realGameWorld = gw; }
     public Iterator getIterator ()
       { return realGameWorld.getIterator(); }
     public void addGameObject(GameObject o)
       {  realGameWorld.addGameObject(o) ;  }
     public boolean removeGameObject (GameObject o)
       {  return false ; }
     //...[also has methods implementing IObservable]
}
```

Proxy Example

```
/** This class defines a Game containing a GameWorld with a ScoreView observer. */
public class Game {
    public Game() {
        GameWorld gw = new GameWorld();   //construct a GameWorld
        ScoreView sv = new ScoreView();   //construct a ScoreView
        gw.addObserver(sv);               //register ScoreView as a GameWorld observer
    }
}
```

```
/** This class defines a GameWorld which is an observable and maintains a list of
 *  observers; when the GameWorld needs to notify its observers of changes it does so
 *  by passing a GameWorldProxy to the observers. */
public class GameWorld implements IObservable, IGameWorld {
    private Vector<GameObject> myGameObjectList = new Vector<GameObject>();
    private Vector<IObserver> myObserverList = new Vector<IObserver>();
    public Iterator<GameObject> getIterator() { ... }
    public void addGameObject(GameObject o) { ... }
    public boolean removeGameObject(GameObject o) {
        //code here to remove the specified GameObject from the GameWorld...
    }
    public void addObserver(IObserver o) { myObserverList.add(o); }

    //Pass a GameWorldProxy to observers, thus prohibiting observer removal of GameObjects
    public void notifyObservers() {
        GameWorldProxy proxy = new GameWorldProxy(this);
        for (IObserver o : myObserverList) {
            o.update((IObservable)proxy, null);
        }
    }
}
```

The Factory

Factory Method Pattern

- Motivation
  - Sometimes a class can’t anticipate the class of objects it must create
  - It is sometimes better to delegate specification of object types to subclasses
  - It is frequently desirable to avoid binding application-specific classes into a set of code

Problem

- You have a code using lots of the same objects
  - E.g., you created **a lot of** “gameobjects” objects

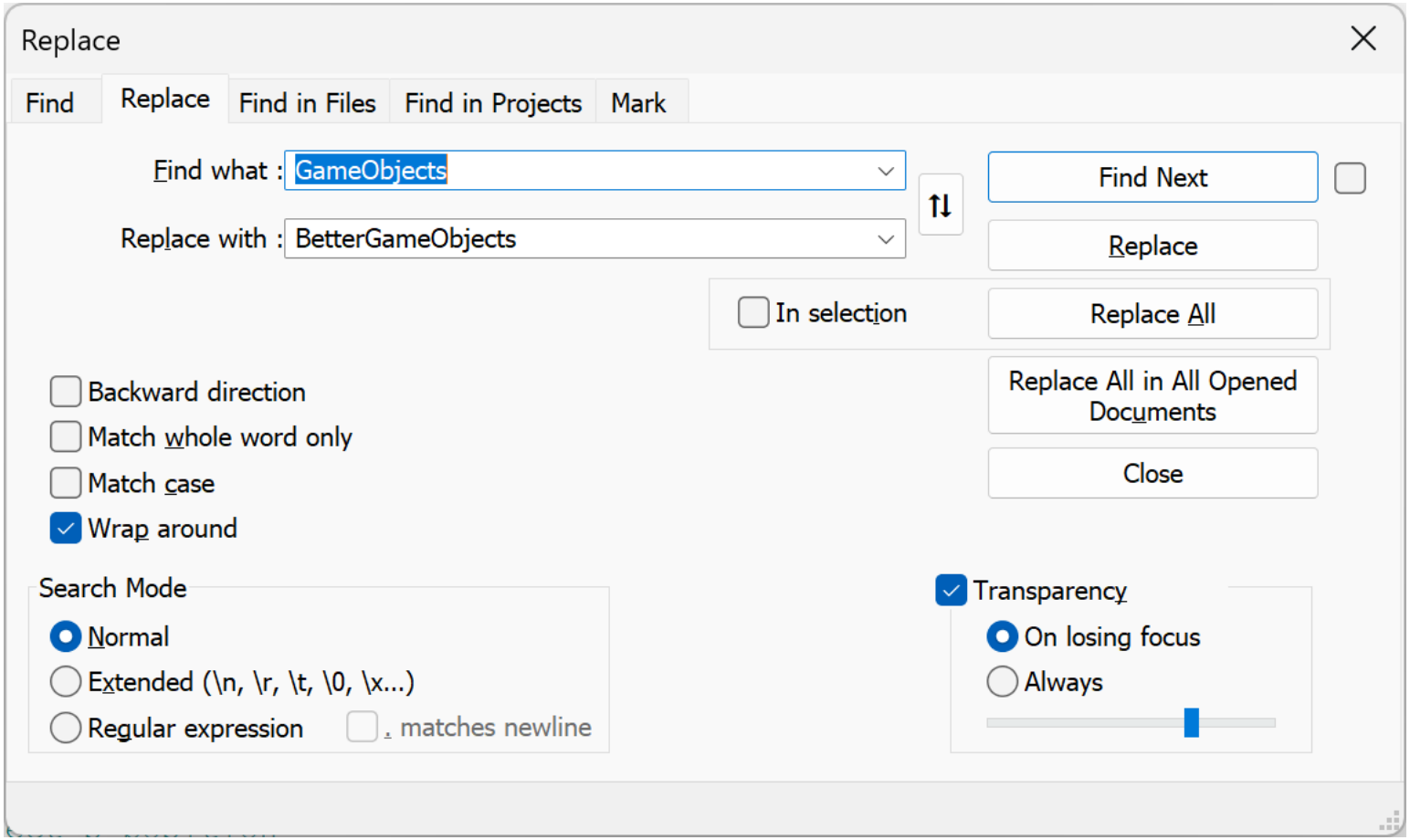
```
GameObjects go = new GameObjects();
```

- A few later, you want to replace game objects to is subclass

```
GameObjects → BetterGameObjects
```

- What can you do?

Solution?


Replace
Find
Replace
Find in Files
Find in Projects
Mark
Find what :
GameObjects
Replace with :
BetterGameObjects
Find Next
Replace
In selection
Replace All
Backward direction
Match whole word only
Match case
Wrap around
Replace All in All Opened Documents
Close
Search Mode
Normal
Extended (\n, \r, \t, \0, \x...)
Regular expression
. matches newline
Transparency
On losing focus
Always

Noooooooooooo!

- It is messy
  - You may replace something wrong
  - You may forget to replace some files

- You may still want the original one.

Factory Method

- Instead of `new GameObject()` directly
  - Make a factory method:

```
public gameobjects createGameObject(){
   return new GameObject();
}
```

- See the point?

In the Future

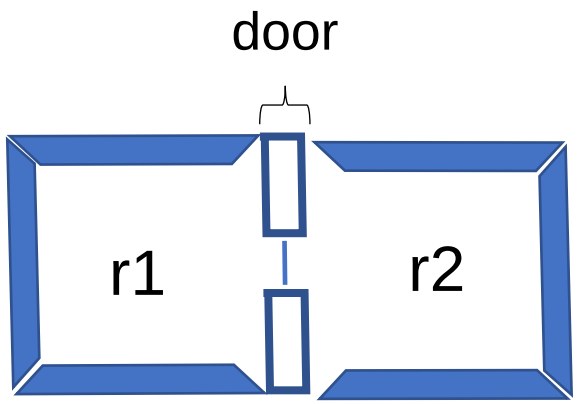
- You can update it directly by:
  - Create a subclass
  - Method overriding

```
public gameobjects createGameObject(){
   return new BetterGameObject();
}
```

- Defer the creation of objects to the subclasses

Example: Maze Game

```
public class MazeGame {

    // This method creates a maze for the game, using a hard-coded structure for the
    // maze (specifically, it constructs a maze with two rooms connected by a door).
    public Maze createMaze () {

        Maze theMaze = new Maze() ;   //construct an (empty) maze

        Room r1 = new Room(1) ;       //construct components for the maze
        Room r2 = new Room(2) ;
        Door theDoor = new Door(r1, r2);

        r1.setSide(NORTH, new Wall()); //set wall properties for the rooms
        r1.setSide(EAST,  theDoor);
        r1.setSide(SOUTH, new Wall());
        r1.setSide(WEST,  new Wall());

        r2.setSide(NORTH, new Wall());
        r2.setSide(EAST,  new Wall());
        r2.setSide(SOUTH, new Wall());
        r2.setSide(WEST,  theDoor);

        theMaze.addRoom(r1); //add the rooms to the maze
        theMaze.addRoom(r2);

        return theMaze ;
    }

    //other MazeGame methods here (e.g. a main program which calls createMaze())...
}
```



Based on an example in "*Design Patterns: Elements of Reusable Object-Oriented Software*" by Gamma et. al. (the so-called "Gang of Four" book).

Problems with `createMaze()`

- Inflexibility
- Lack of reusability

- **Reason**:
  - Hardcodes the maze types
  - Suppose we want to create a maze with (e.g.)
    - Magic Doors
    - Enchanted Rooms

With Factory Methods

```
public class MazeGame {

   //factory methods - each returns a MazeComponent of a given type
   public Maze makeMaze()      { return new Maze() ; }
   public Room makeRoom(int id)  { return new Room(id) ; }
   public Wall makeWall()      { return new Wall() ; }
   public Door makeDoor(Room r1, Room r2) { return new Door(r1,r2) ; }

   // Create a maze for the game using factory methods
   public Maze createMaze () {
      Maze theMaze = makeMaze() ;
      Room r1 = makeRoom(1) ;
      Room r2 = makeRoom(2) ;
      Door theDoor = makeDoor(r1, r2);
      r1.setSide(NORTH, makeWall());
      r1.setSide(EAST,  theDoor);
      r1.setSide(SOUTH, makeWall());
      r1.setSide(WEST,  makeWall());
      r2.setSide(NORTH, makeWall());
      r2.setSide(EAST,  makeWall());
      r2.setSide(SOUTH, makeWall());
      r2.setSide(WEST,  theDoor);
      theMaze.addRoom(r1);
      theMaze.addRoom(r2);
      return theMaze ;
   }
}
```

Overriding Factory Methods

```
//This class shows how to implement a maze made of different types of rooms.  Note
// in particular that we can call exactly the same (inherited) createMaze() method
// to obtain a new "EnchantedMaze".
public class EnchantedMazeGame extends MazeGame {

    //override MakeRoom to produce "EnchantedRooms"
    @Override
    public Room makeRoom(int id) {

        //create the spell necessary to enter the enchanted room
        Spell spell = makeSpell() ;

        //construct and return an EnchantedRoom
        return new EnchantedRoom(id, spell);
    }

    //override MakeDoor to produce a door requiring a spell
    @Override
    public Door makeDoor(Room r1, Room r2) {

        //construct and return a Door requiring a spell to be entered
        return new DoorNeedingSpell(r1, r2);
    }

    //new factory method for making spells
    public Spell makeSpell() { return new Spell() ;}
    ...
}
```

The State

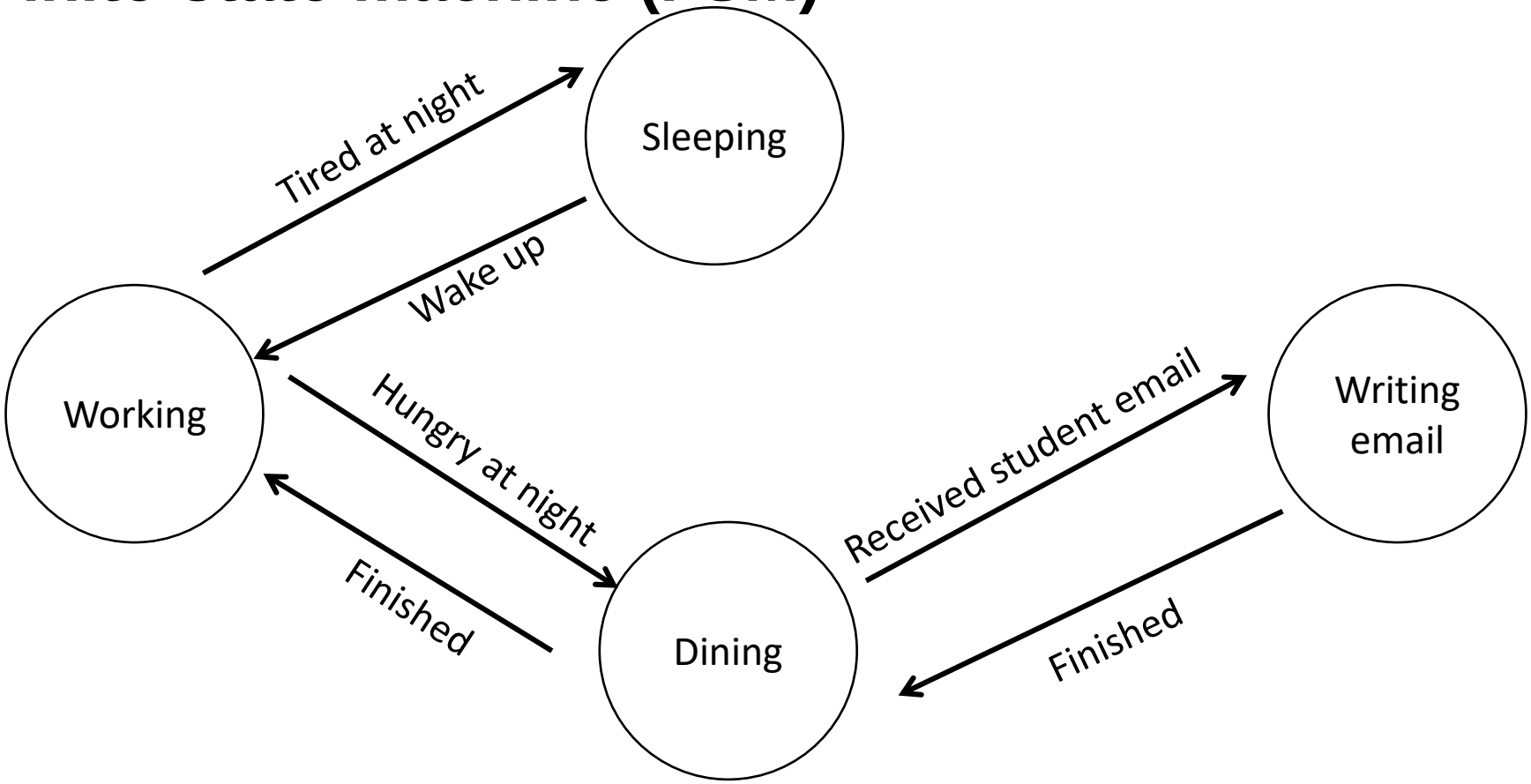
State Pattern

Motivation:

- Objects have different state
- Objects change their behaviors when its state changed
- Adding new states should not affect the existing one

State Example

Finite-state machine (FSM)



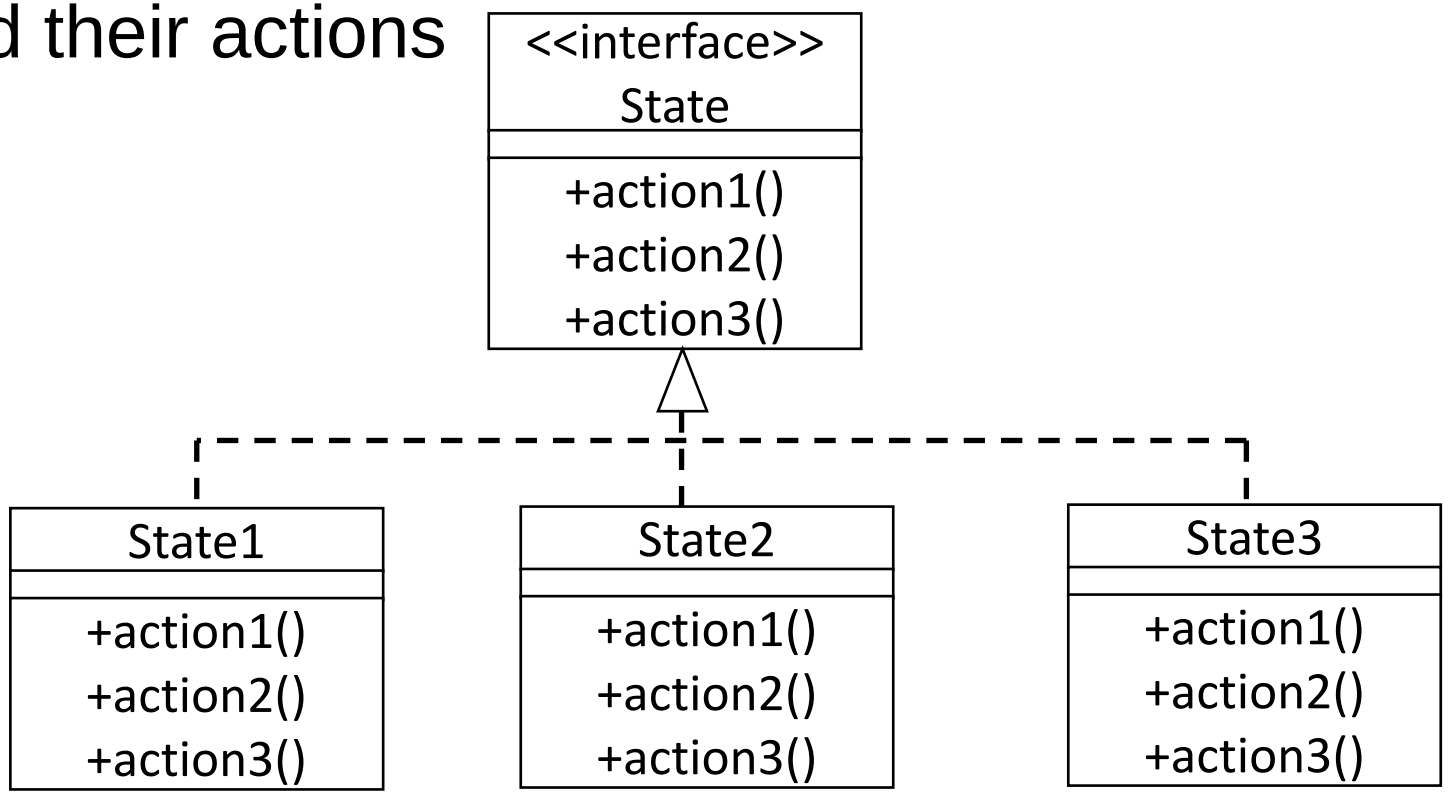
Traditional Way

```
if (state == “working”){
     if ( ... ) { ... }
}else if (state == “dining”){
     if ( ... ) { ... }
}else if ( ... ){
     if ( ... ) { ... }
}else{
     ...
}
```

So many if-statement or switch-statement. Hard to manage them

Solution

Create an `interface` to store different stage and their actions



DiningState Code

```
class diningState implements State{
      private Teacher kc;
      public void tired(){ }
      public void finished(){
            kc.goToWork();
      }
      public void recEmail(){
            kc.replyEmail();
      }
      ...
}
```

emailState Code

```
class emailState implements State{
     private Teacher kc;
     public void tired(){ }
     public void finished(){
          kc.goBackDining();
     }
     public void recEmaIl(){ }
     ...
}
```

Revision

Types of Design Patterns

- **Creational**
  - Deal with process of object creation
- **Structural**
  - Deal with structure of classes – how classes and objects can be combined to form larger structures
  - Design objects that satisfy constraints
  - Specify connections between objects
- **Behavioral**
  - Deal with interaction between objects
  - Encapsulate processes performed by objects

Concurrency Pattern

New design patterns for multi-threaded

- Active Object
- Balking pattern
- Barrier
- Double-checked locking
- Guarded suspension
- Leaders/followers pattern
- Monitor Object
- Nuclear reaction
- Reactor pattern
- Read write lock pattern
- Scheduler pattern
- Thread pool pattern
- Thread-local storage

Design Patterns

Same implementation used many times

- Effective
- Form a pattern

You used it without knowing it

Common Design Patterns

Creational:
- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

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

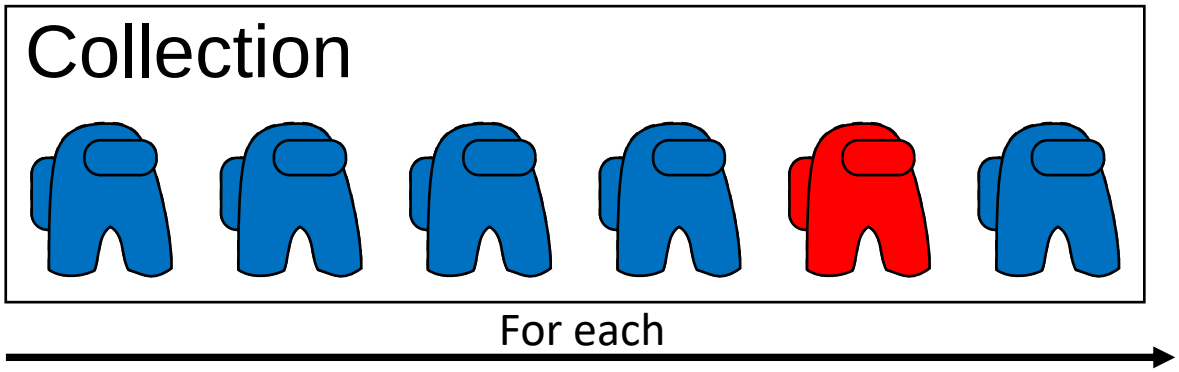
Behavioral:
- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Iterator Pattern

- Behavioral

- With a collection class to collect elements

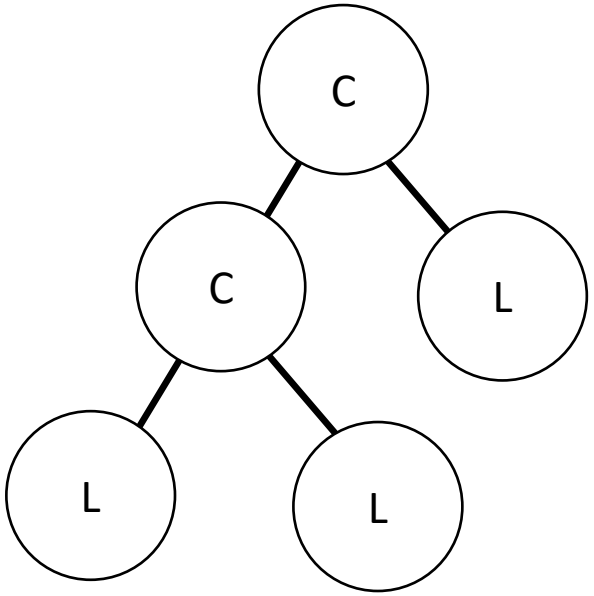
- Use an iteration to access all elements



Composite Pattern

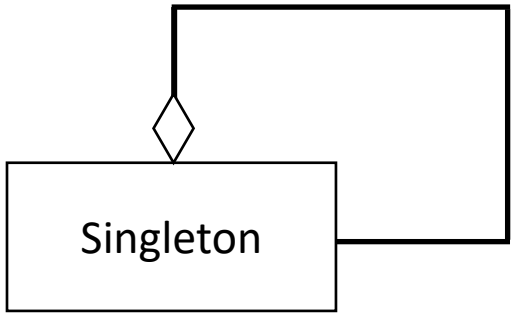
- Structural

- A structure that contain same-type objects
  - A tree node contains another tree nodes



Singleton Pattern

- Creational
- A class that can create one and only objects
  - Private constructor
  - Store in class variable
  - Static `get()` method

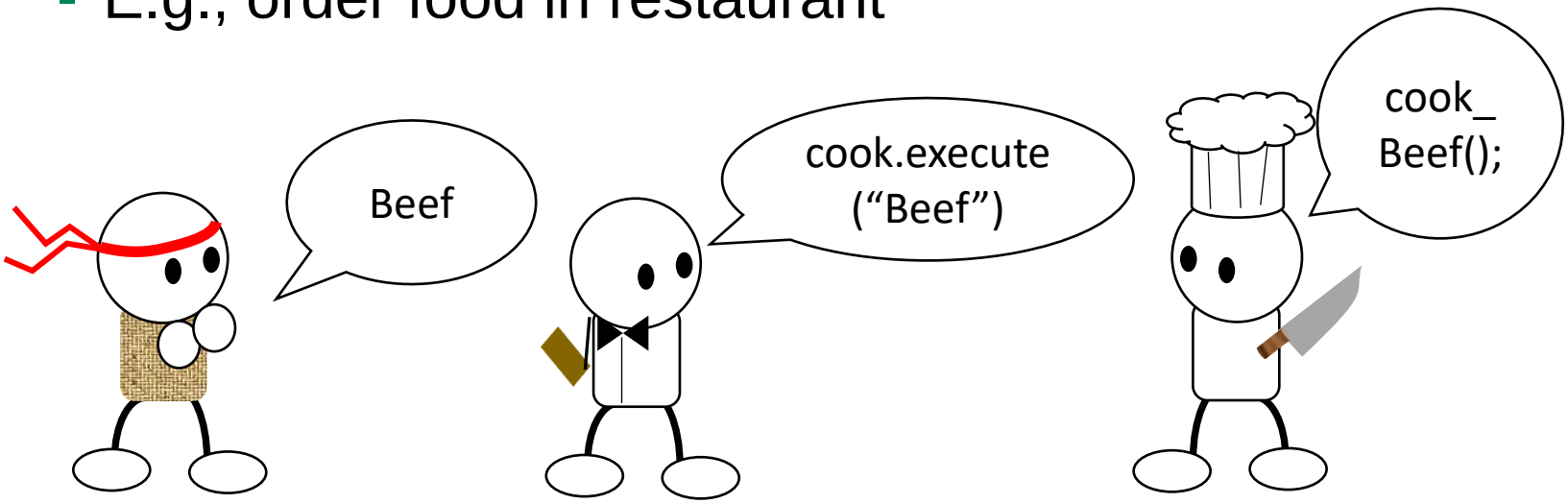


Observer Pattern

- Behavioral
- Keep tracking the data storage, notify if there is any changes.
  - Similar to YouTube subscription
  - MVC

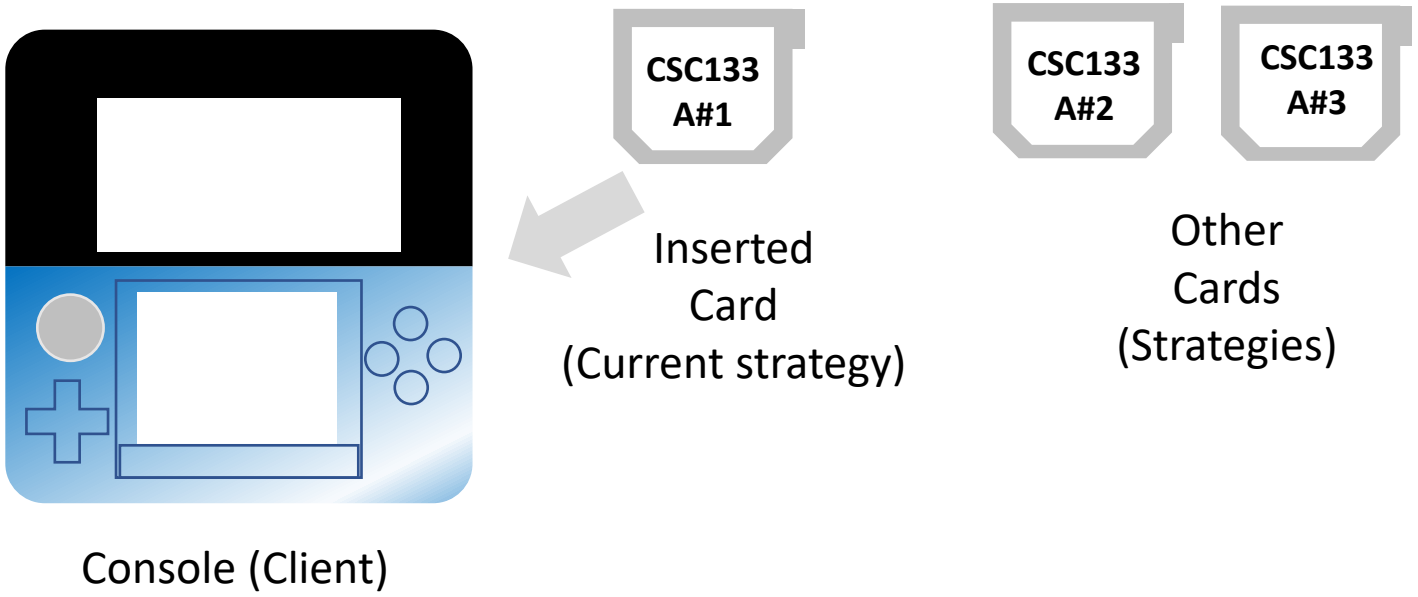
Command Pattern

- Behavioral
- Set up a list of command for `execute()`, only receiver know how to do it.
  - E.g., order food in restaurant



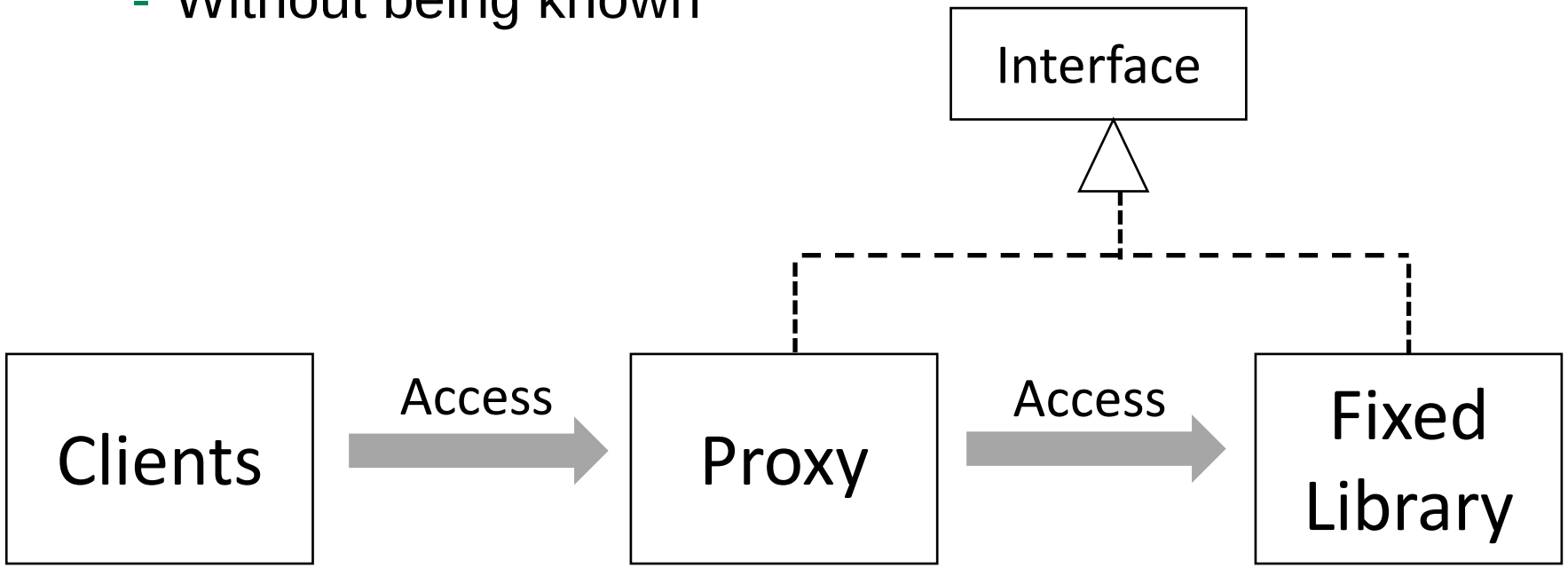
Strategy Pattern

- Behavioral Pattern
  - Define the current strategy
  - All strategy should provide same method to apply



Proxy Pattern

- A proxy between two class
  - To add new features
  - Without being known



Factory Method Pattern

Instead of new object directly, put them into factory method:

- Methods to create new objects
- Easy to change

State Pattern

- Finite-State Machine
  - Act depending on the currents state
  - Similar to Strategy Pattern

State

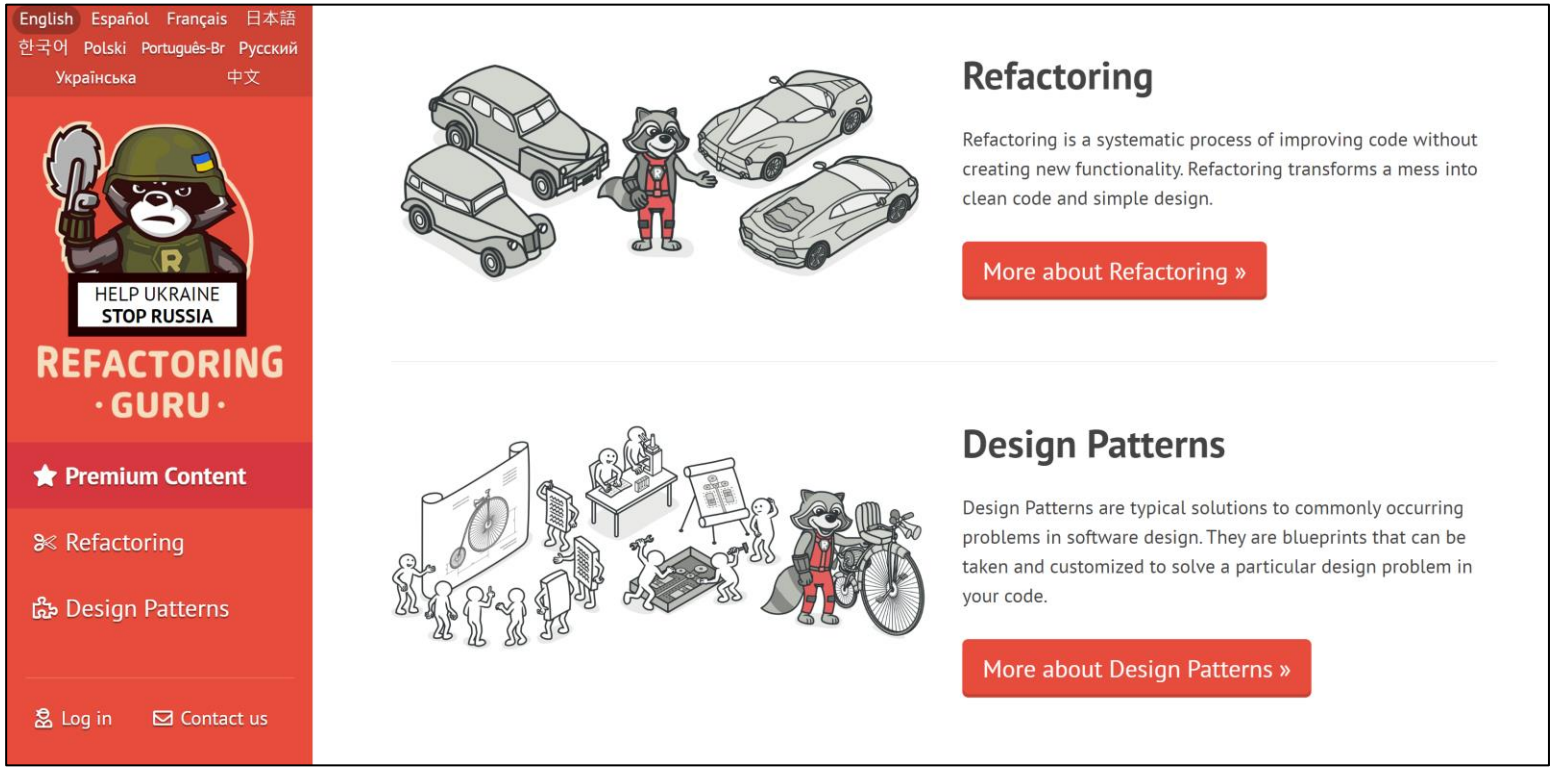
- What and When
- Limited change of state

Strategy

- How
- Freely change strategy

Refactoring Guru

https://refactoring.guru/



Any Questions?