DESIGNING A SOFTWARE APPLICATION IS LIKE DESIGNING A BOAT...
572
YOU SPEND COUNTLESS HOURS LOCATING THE IDEAL MATERIALS AND TESTING THE ENGINE TO THE CUSTOMER'S EXACT SPECIFICATIONS, ONLY WHEN YOU'RE DONE THE CUSTOMER TELLS YOU, "GREAT JOB, WE LOVE IT, JUST ONE LITTLE THING:
WE NEED IT TO FLY."
imgflip.com

ARHUS
NIVERSITY
SCHOOL OF ENGINEERING

Previously…

GoF Template method & GoF Strategy

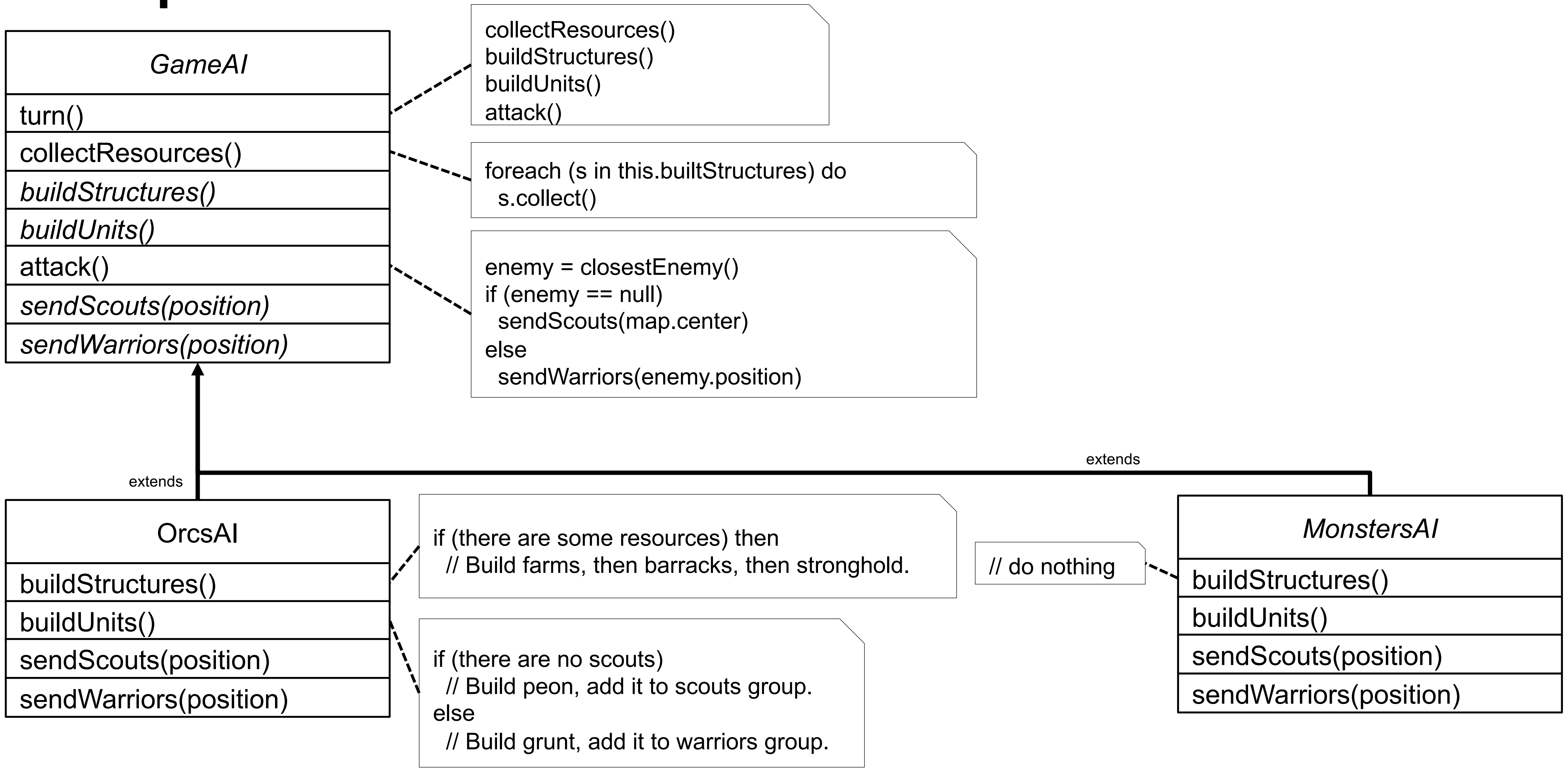
version: 1.0.3

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Agenda

- GoF Template Method
- GoF Strategy
- Comparison
- Lab exercise: “SuperSorter”

Example: Game



```
collectResources()
buildStructures()
buildUnits()
attack()
```

```
foreach (s in this.builtStructures) do
  s.collect()
```

```
enemy = closestEnemy()
if (enemy == null)
  sendScouts(map.center)
else
  sendWarriors(enemy.position)
```

```
if (there are some resources) then
  // Build farms, then barracks, then stronghold.
```

```
if (there are no scouts)
  // Build peon, add it to scouts group.
else
  // Build grunt, add it to warriors group.
```

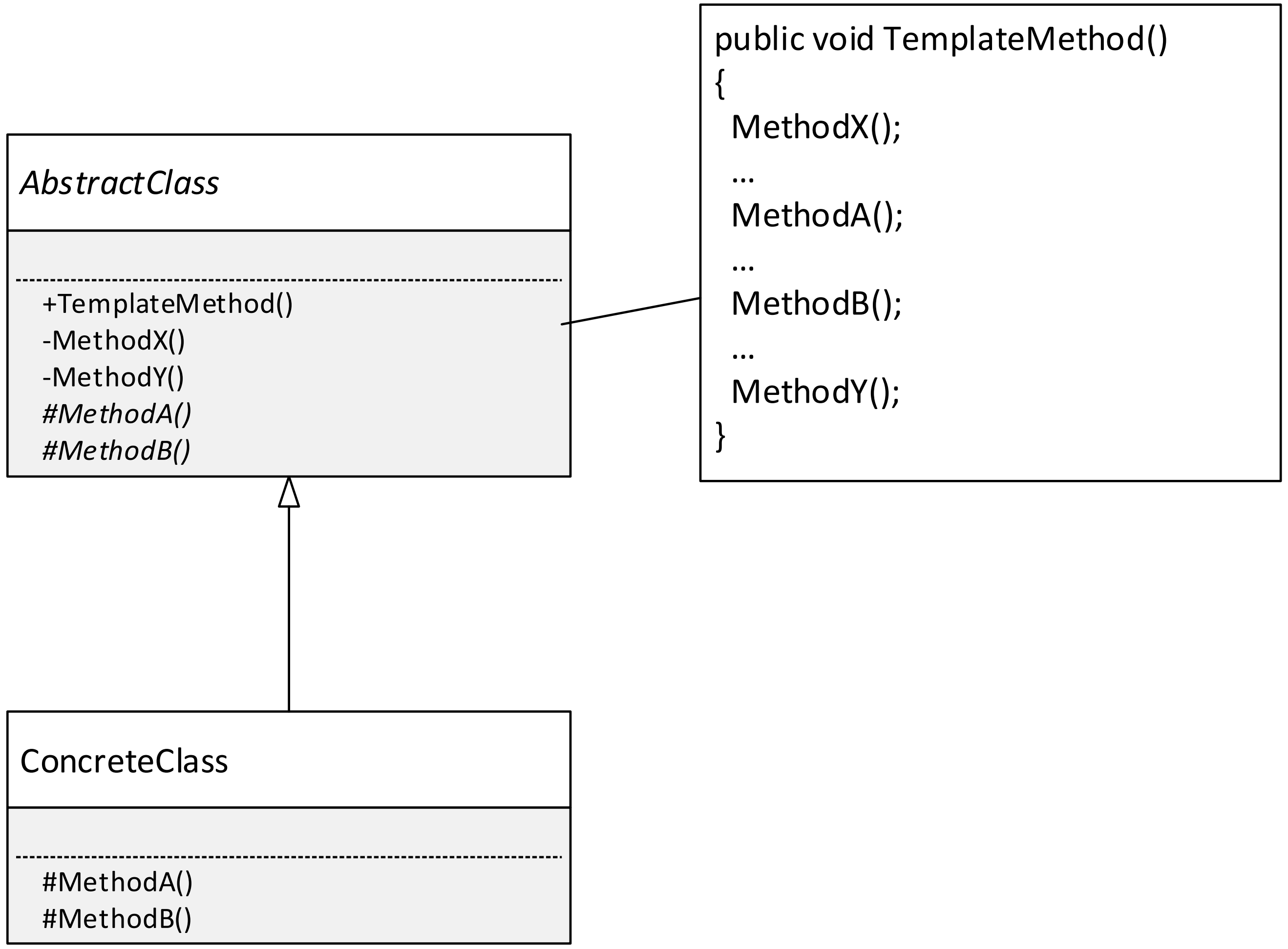
https://refactoring.guru/design-patterns/template-method

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

GoF Template Method

- Pattern name:
  - Template Method
- Intent:
  - Define the *skeleton* of an algorithm in an operation, deferring some steps to client subclasses.

Structure



```
public void TemplateMethod()
{
  MethodX();
  …
  MethodA();
  …
  MethodB();
  …
  MethodY();
}
```

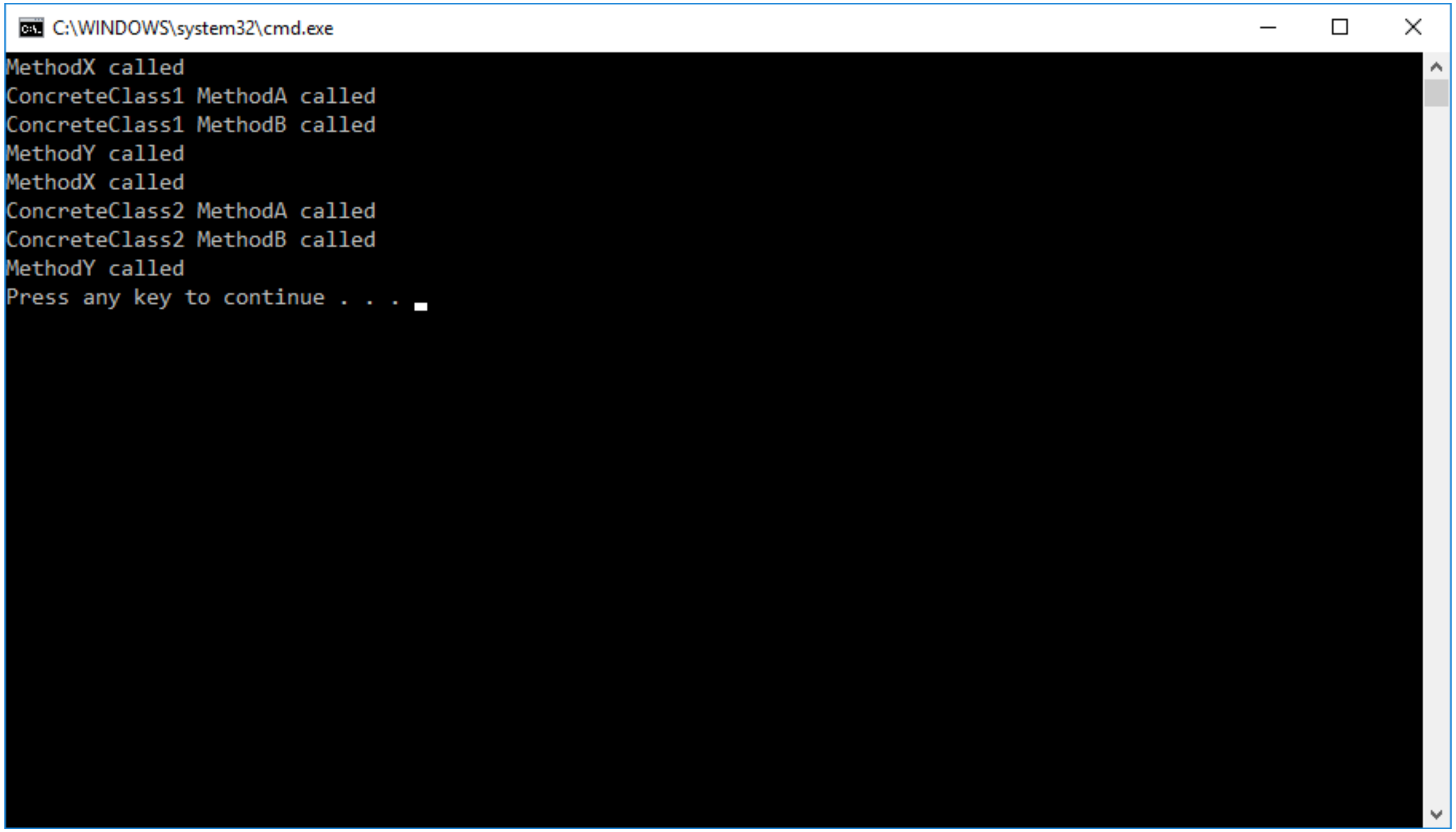
Implementation and program flow

```
public abstract class AbstractClass
{
    public void TemplateMethod()
    {
        MethodX();
        MethodA();
        MethodB();
        MethodY();
    }

    protected abstract void MethodA();
    protected abstract void MethodB();

    private void MethodX()
    {
        Console.WriteLine("MethodX called");
    }

    private void MethodY()
    {
        Console.WriteLine("MethodY called");
    }
}
```

```
public class ConcreteClass1 : AbstractClass
{
    protected override void MethodA()
    {
        Console.WriteLine("ConcreteClass1 MethodA
called");
    }

    protected override void MethodB()
    {
        Console.WriteLine("ConcreteClass1 MethodB
called");
    }
}
```

```
static void Main(string[] args)
{
    AbstractClass ac = new ConcreteClass1();
    ac.TemplateMethod();

    AbstractClass ac2 = new ConcreteClass2();
    ac2.TemplateMethod();
}
```

Implementation and program flow



C:\WINDOWS\system32\cmd.exe

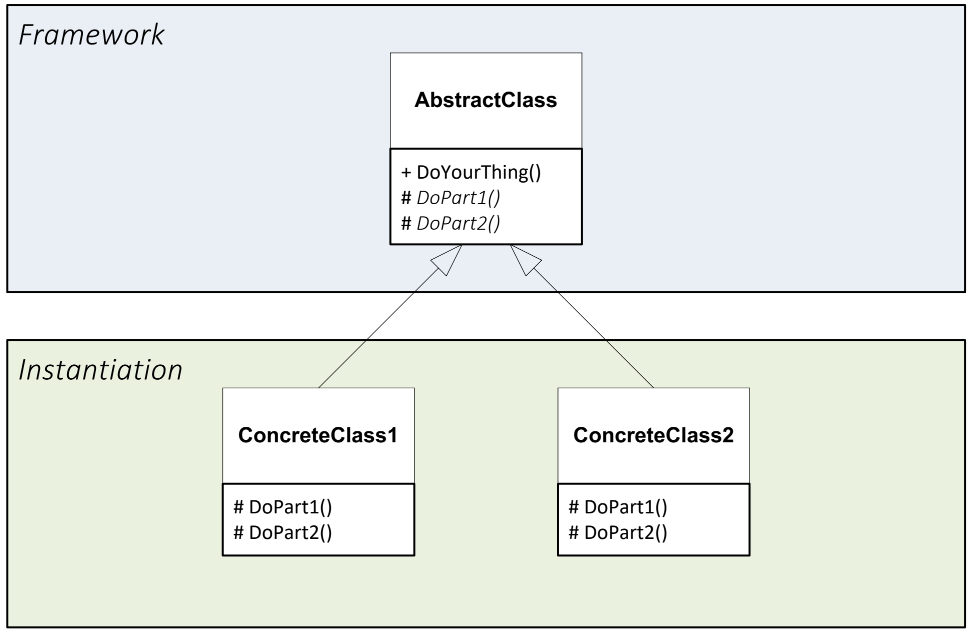
```
MethodX called
ConcreteClass1 MethodA called
ConcreteClass1 MethodB called
MethodY called
MethodX called
ConcreteClass2 MethodA called
ConcreteClass2 MethodB called
MethodY called
Press any key to continue . . .
```

GoF Template Method

- Template Method is commonly used in frameworks
  - Frameworks controls the flow (when to do something)
  - You instantiate the framework by implementing framework methods in derived classes
- Also called Hollywood pattern
  - Don’t call us – we’ll call you

GoF Template Method

- Template Method is commonly used in frameworks
  - Frameworks controls the flow (when to do something)
  - You instantiate the framework by implementing framework methods in derived classes



AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

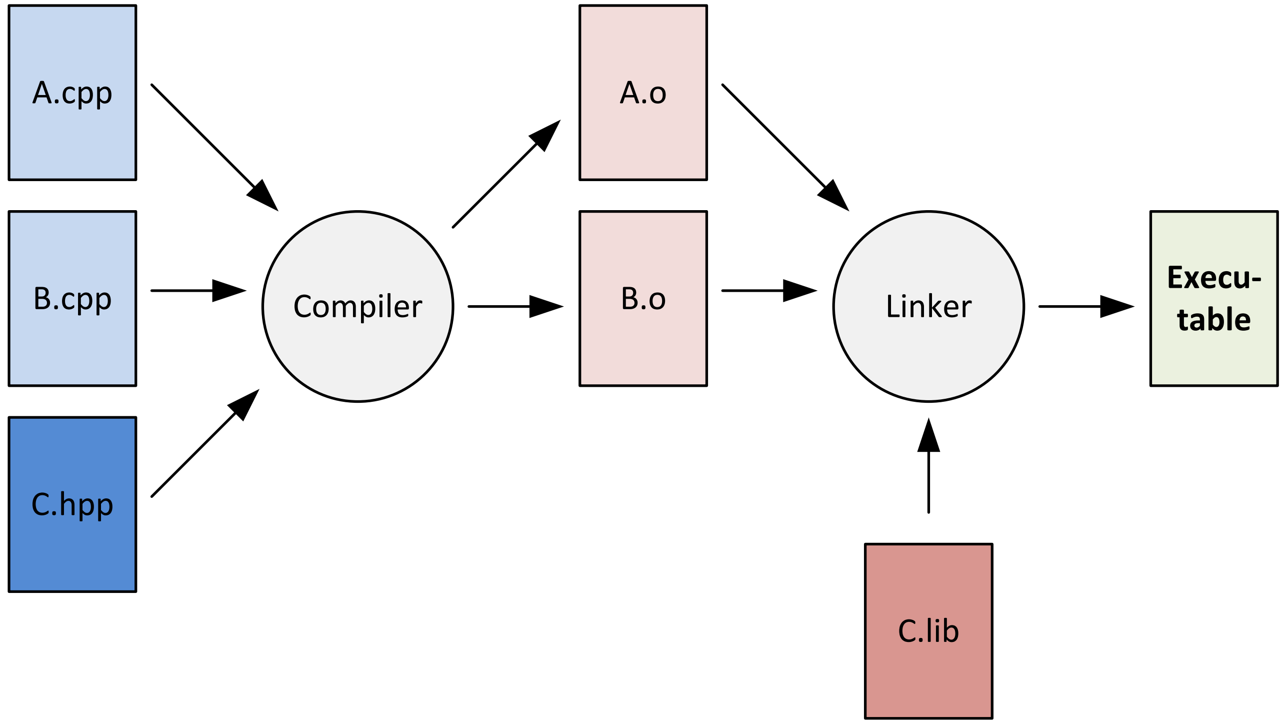
Example: Turn-based Games

```
abstract class TurnBasedGame
{
  private List<Player> _players;

  // Template Method – defines ”structure”
  public void PlayGame()
  {
    int i=0;
    InitGame();
    while(!GameOver())
    {
        TakeTurn(_players[i]);
        i = (i+1) % _players.Count;
    }
    AnnounceWinner();
  }

  // Methods to be implemented by subclasses
  protected abstract void InitGame();
  protected abstract bool GameOver();
  protected abstract void TakeTurn(Player p);
  protected abstract void AnnounceWinner()

}
```

```
class Chess: TurnBasedGame
{
  protected override void InitGame()
  {
    // Set up chess pieces
  }

  protected abstract bool GameOver()
  {
    // Check for check-mate
  }

  protected abstract void TakeTurn(Player p)
  {
    // Move one of p’s pieces IAW chess rules
  }

  protected abstract void AnnounceWinner()
  {
    // Announce the winner of the game
  }
}
```

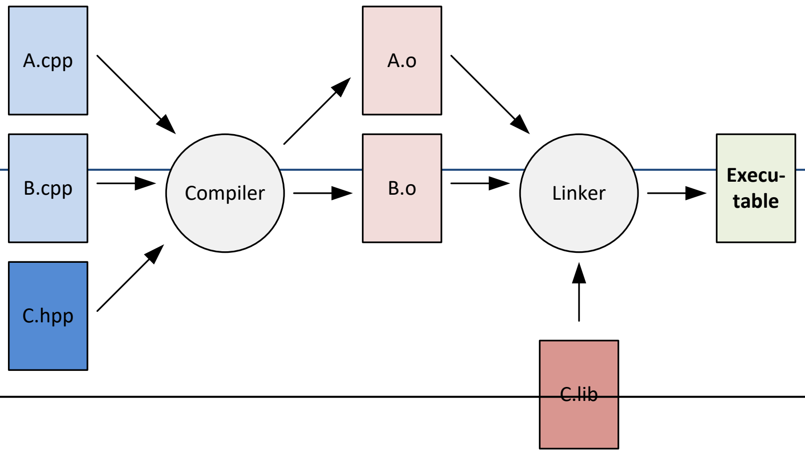
Example 4: Build Process

- A simplified picture of the build process in e.g. C++

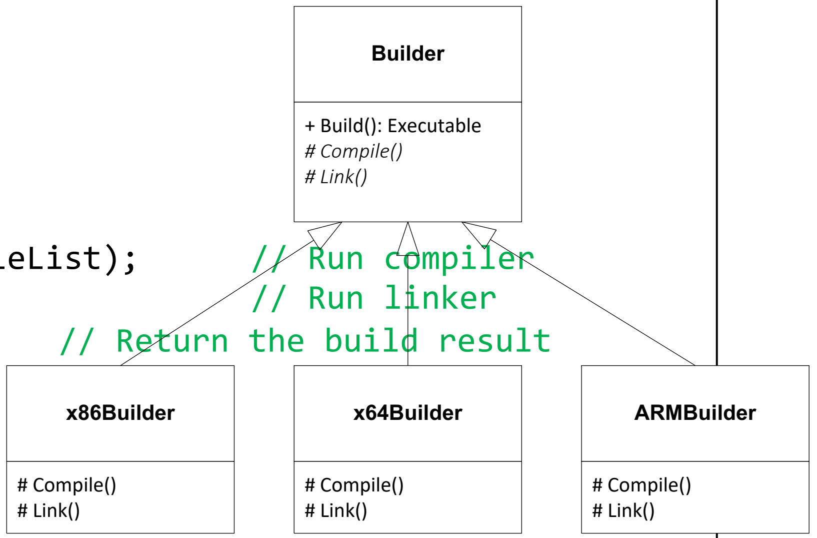


Describe, in general terms, how this build process could take place for different platforms (compiler suites) using the *Template Method pattern*

The *compile* → *link* → *return executable* sequence is fixed, but *actual* compilation and linkage is deferred to subclass(es)



```
abstract class Builder
{
  // Template Method
  public Executable Build(MakeFile m)
  {
    try
    {
      var objFiles = Compile(m.SourceFileList, m.HeaderFileList);     // Run compiler
      var executable = Link(objFiles, m.LibFiles);                    // Run linker
      return executable;                                   // Return the build result
    }
    catch(BuildError be)
    {
      // Build error occurred – handle it
      Console.WriteLine(be); // Do not do at home.
    }
  }

  protected abstract List<File> Compile(List<File> sourceFiles, List<File> headerFiles);
  protected abstract List<File> Link(List<File> objFiles, List<File> libFiles);
}
```



Abstract methods

- Optional operation
  - Empty implementation in root object – Hooked Method
- Similar for all subclasses
  - Final declaration

GoF Strategy

- Pattern name:
  - Strategy
- Intent:
  - Define a *family* of algorithms, encapsulate each one, and make them interchangeable at runtime.

Example: Robocode Game

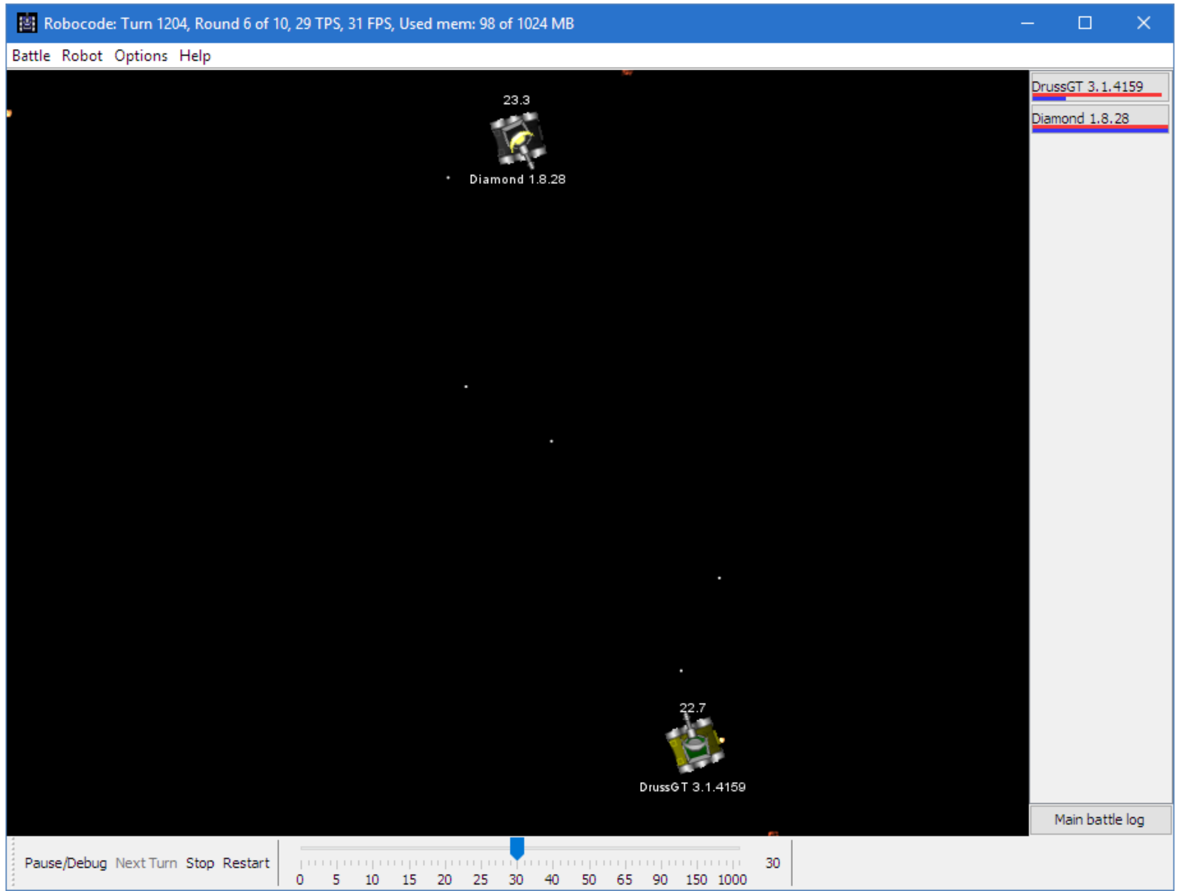
- Robocode is a programming game where the goal is to **code a robot** to compete against other robots in a battle arena.
- The **player is the programmer of the robot**, who will have no direct influence on the game.
- Robocode's battles take place in a battlefield, where small automated 6-wheeled robots fight it out until only one is left.

```
public class MyRobot extends Robot {
    public void run() {
        while (true) {
            ahead(100);
            turnGunRight(360);
            back(100);
            turnGunRight(360);
        }
    }

    public void onScannedRobot(ScannedRobotEvent e) {
        fire(1);
    }
}
```

The game calls your onScannedRobot() method whenever another robot is seen.

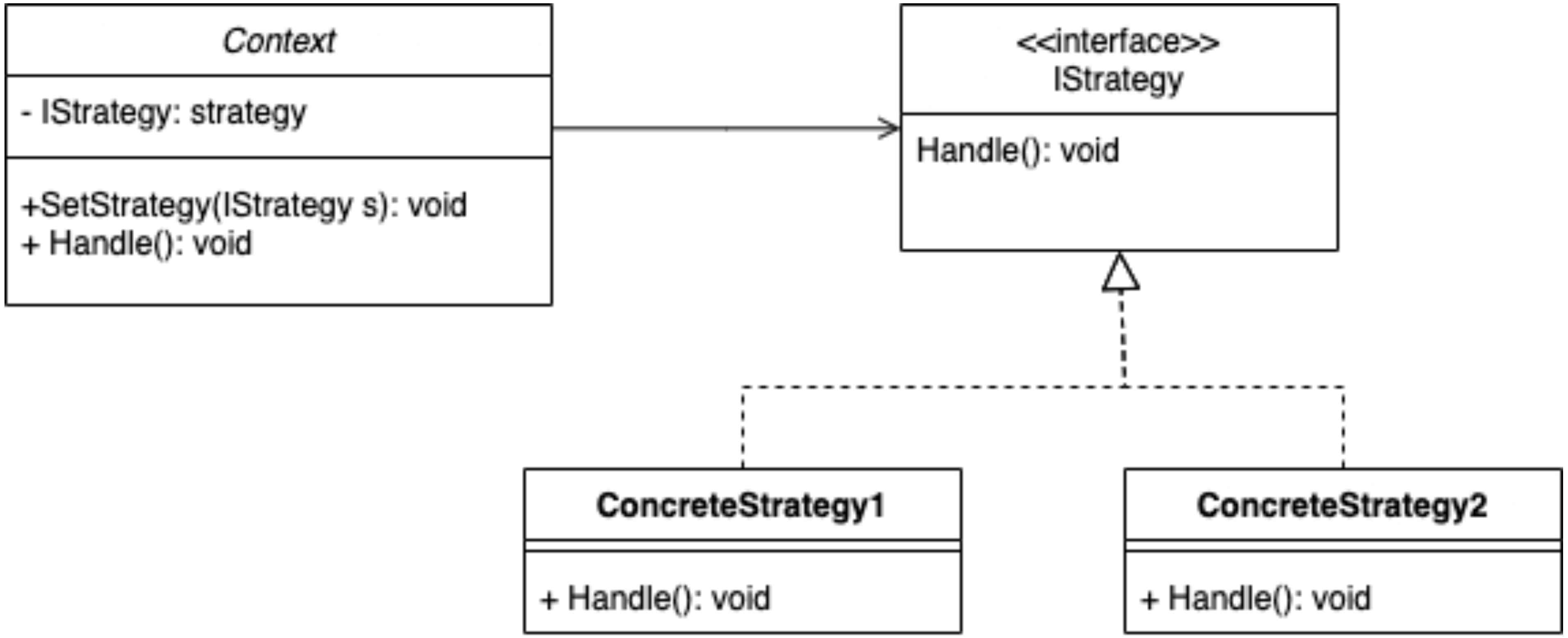
The order that Robocode runs is as follows:

1. Battle view is (re)painted.
2. All robots execute their code until they take action (and are then paused).
3. Time is updated (time++).
4. All bullets move (including the bullet fired in the last tick) and are checked for collisions.
5. All robots move (gun, radar, heading, acceleration, velocity, distance, in that order. gun heat is also decreased in this step).
6. All robots perform scans (and collect team messages).
7. All robots are resumed to take new action.
8. Each robot processes its event queue.

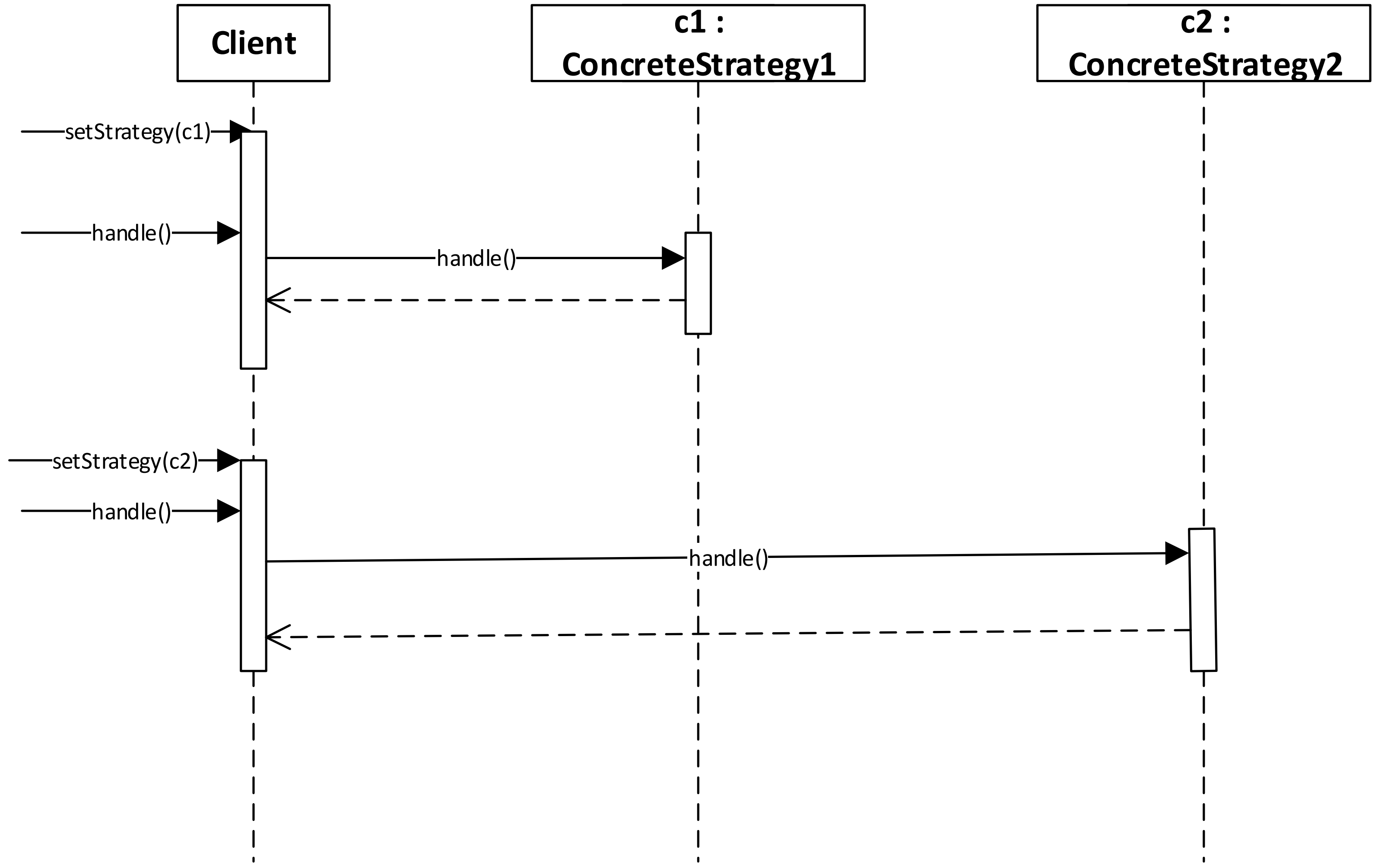


Robocode: Turn 1204, Round 6 of 10, 29 TPS, 31 FPS, Used mem: 98 of 1024 MB

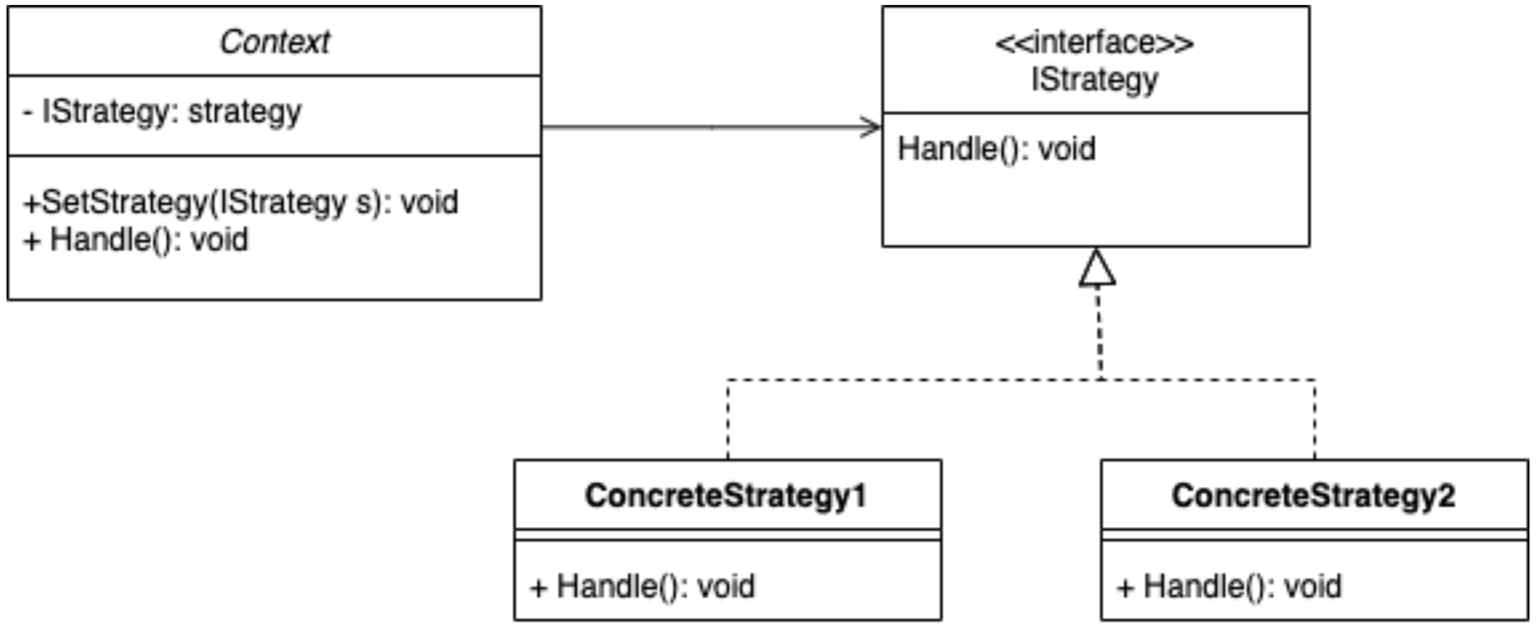
Structure


Context
- IStrategy: strategy
+SetStrategy(IStrategy s): void
+ Handle(): void
<<interface>>
IStrategy
Handle(): void
ConcreteStrategy1
+ Handle(): void
ConcreteStrategy2
+ Handle(): void

Sequence


Client
c1 : ConcreteStrategy1
c2 : ConcreteStrategy2
setStrategy(c1)
handle()
handle()
setStrategy(c2)
handle()
handle()

The Strategy pattern enables the *behavior* of the context to be defined at runtime by delegating it to another object.



Example: Event logging

```
class SomeSubSystem
{
  public ILog Log {set; private get;}

  public SomeSubSystem()
  {
    Log = new NullLog(); // Default
  }


  public void DoYourThing()
  {
    Log.Log(”Event occurred”);
  }
}
```

```
interface ILog
{
  void Log(string s);
}
```

```
class ConsoleLog : ILog
{
  public void Log(string s)
  {
    Console.WriteLine(s);
  }
}
```

```
class FileLog : ILog
{
  ...
}
```

Other examples

- Sorting
- Games
- Checks/rules
- …?

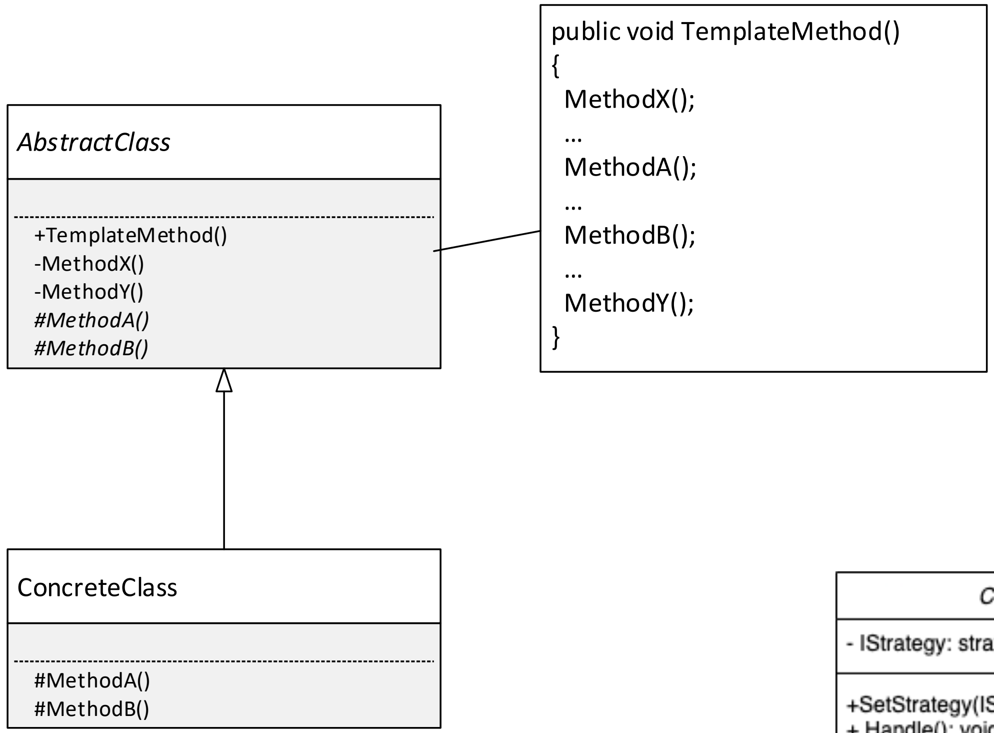
Consequences

- Alternative to sub-classing
- Eliminate switch/case
- Increases number of classes (stateless)
- Number of possible implementations
- Overhead between Strategy & Context
  - Strategy interface handles simple to complex

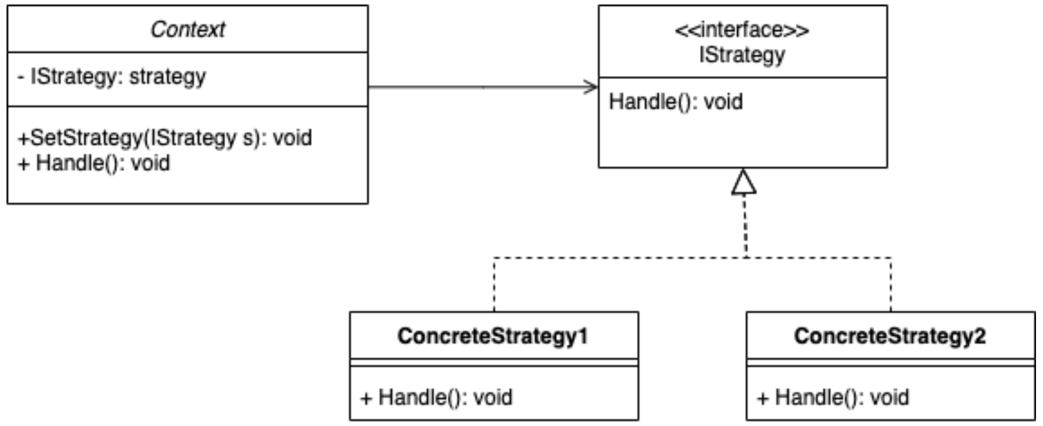
Comparison

- *GoF Template Method* and *GoF Strategy* are both behavioral patterns.
- Both are used to make the behavior of a system *extensible*.
- GoF Template Method uses *inheritance*
  - Callback implementations
  - Frameworks
  - Behavior fixed at compile-time
- GoF Strategy uses *delegation*
  - Behavior can be changed at runtime

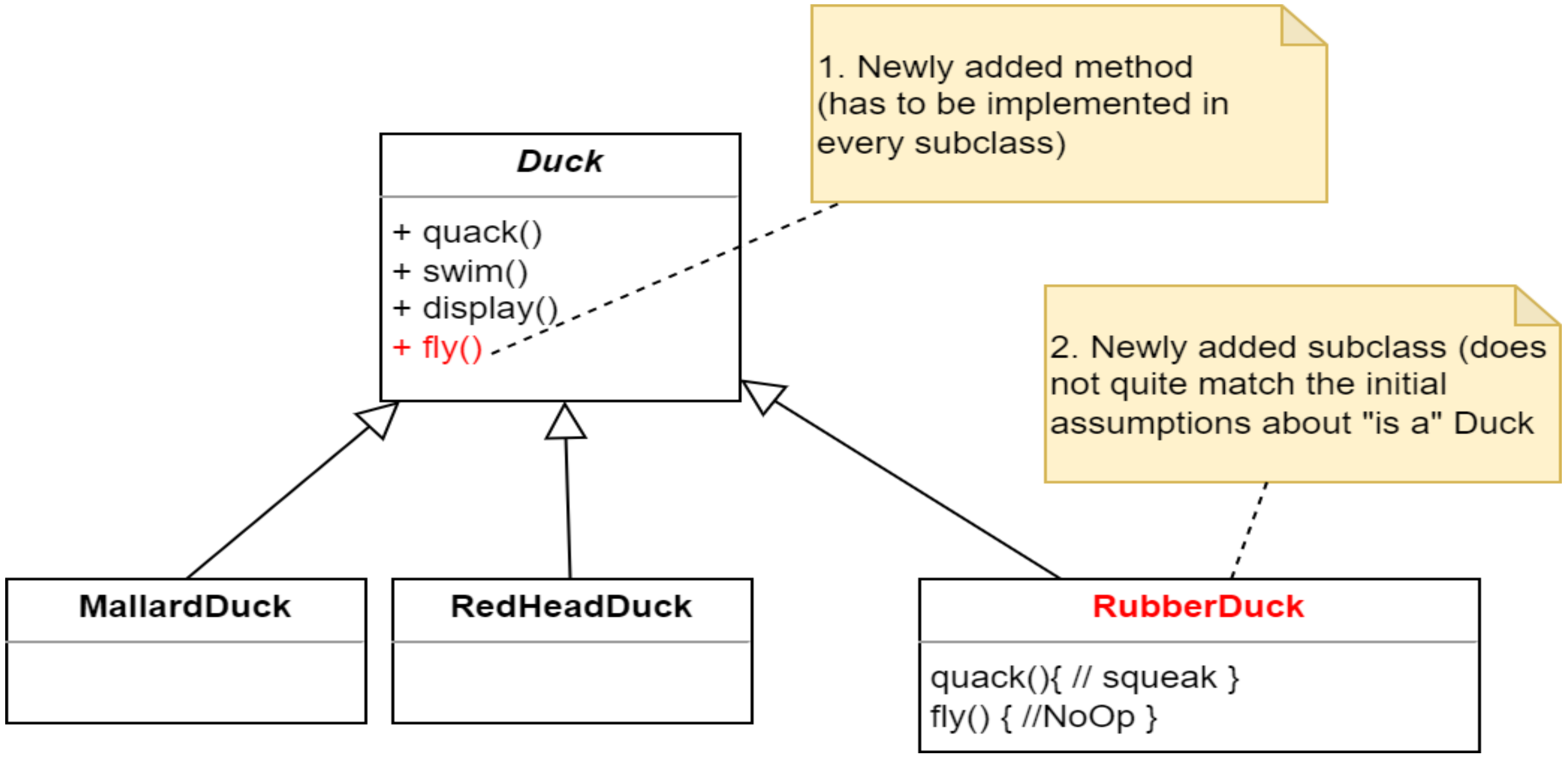
Inheritance vs Delegation



```
public void TemplateMethod()
{
  MethodX();
  …
  MethodA();
  …
  MethodB();
  …
  MethodY();
}
```

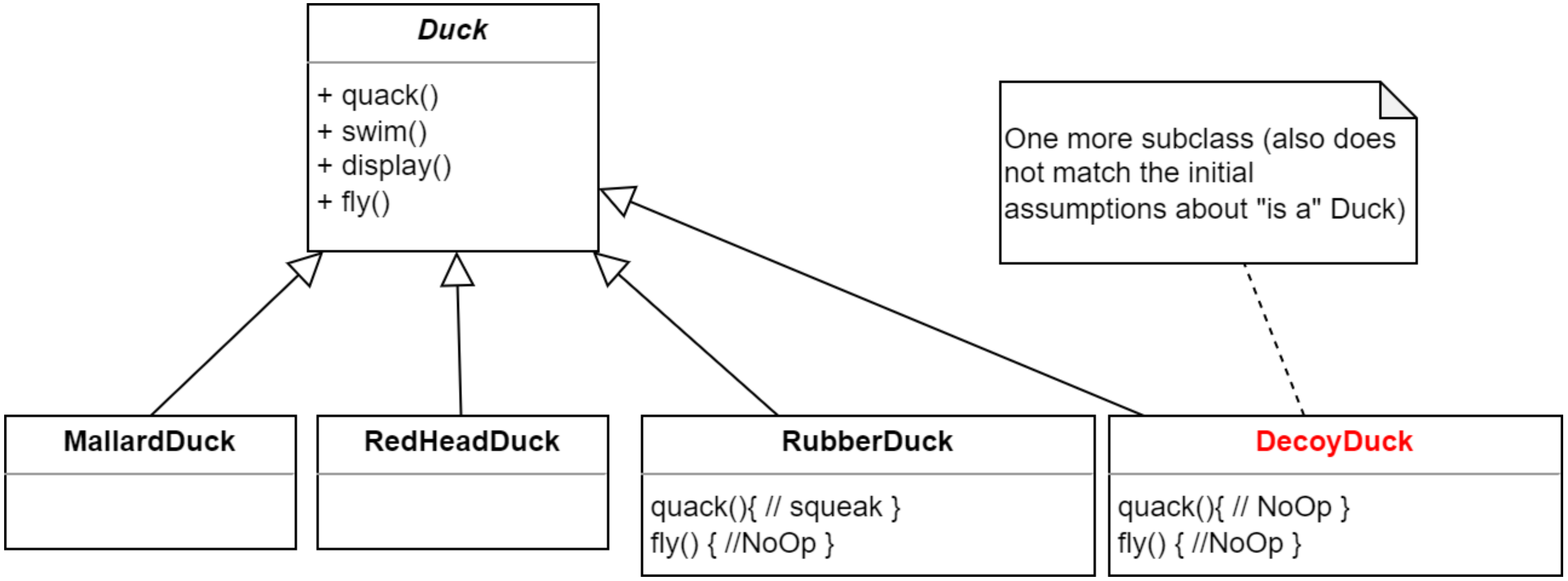


Example: Inheritance vs Delegation


Duck
+ quack()
+ swim()
+ display()
+ fly()
1. Newly added method (has to be implemented in every subclass)
2. Newly added subclass (does not quite match the initial assumptions about "is a" Duck
MallardDuck
RedHeadDuck
RubberDuck
quack(){ // squeak }
fly() { //NoOp }

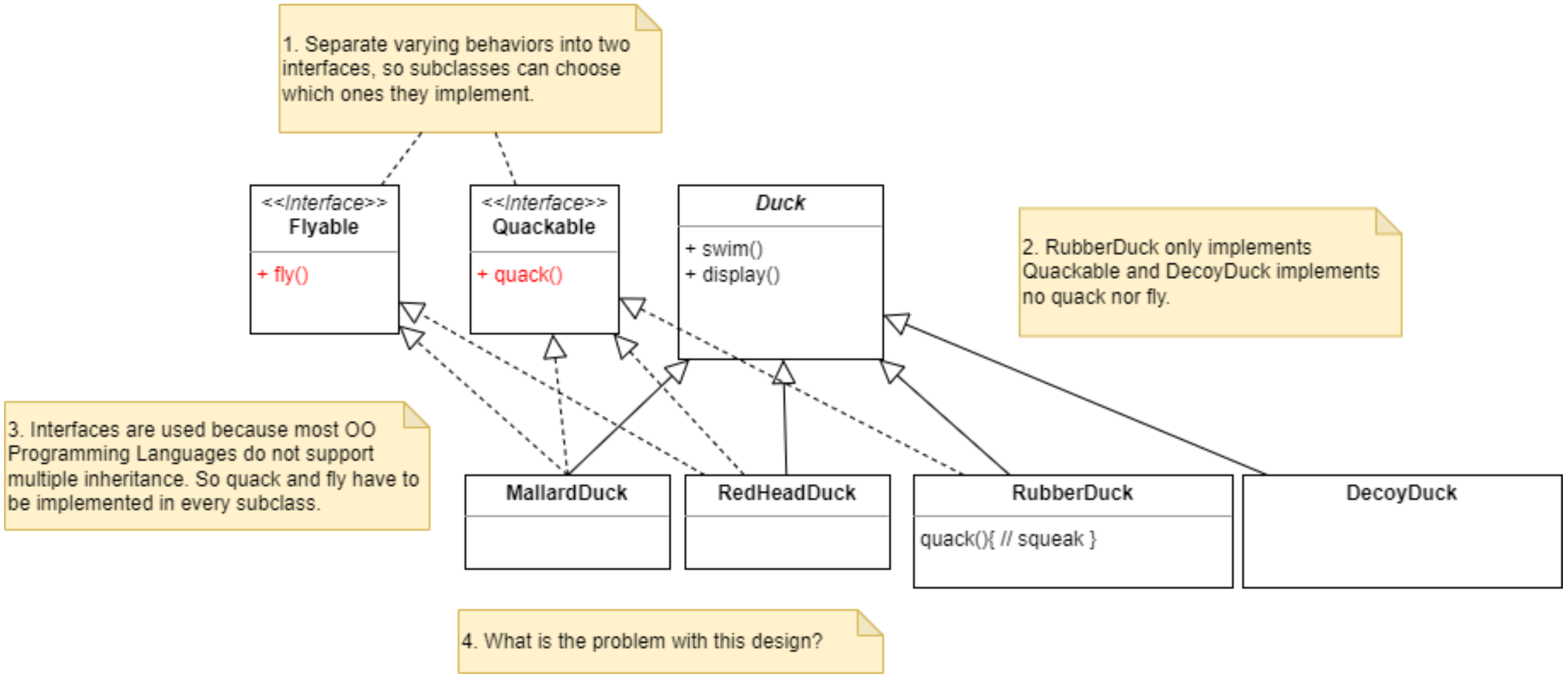
Example adapted from: Freeman, Eric, Elisabeth Robson, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, Inc., 2008.

Example: Inheritance vs Delegation



Example adapted from: Freeman, Eric, Elisabeth Robson, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, Inc., 2008.

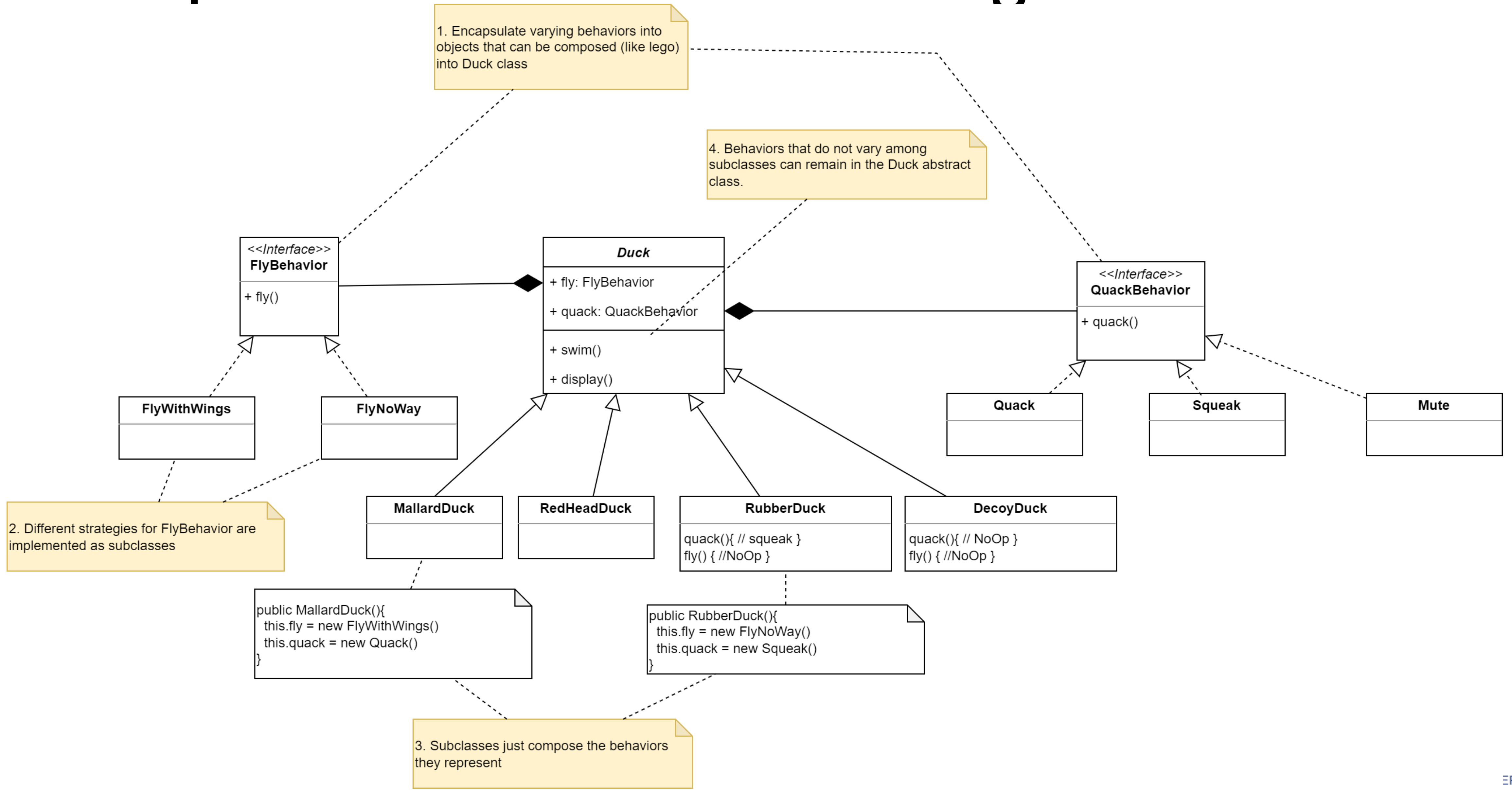
Example: Inheritance vs Delegation


1. Separate varying behaviors into two interfaces, so subclasses can choose which ones they implement.
<<Interface>>
Flyable
+ fly()
<<Interface>>
Quackable
+ quack()
Duck
+ swim()
+ display()
2. RubberDuck only implements Quackable and DecoyDuck implements no quack nor fly.
3. Interfaces are used because most OO Programming Languages do not support multiple inheritance. So quack and fly have to be implemented in every subclass.
MallardDuck
RedHeadDuck
RubberDuck
quack(){ // squeak }
DecoyDuck
4. What is the problem with this design?

Example adapted from: Freeman, Eric, Elisabeth Robson, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, Inc., 2008.

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Example: Inheritance vs Delegation


1. Encapsulate varying behaviors into objects that can be composed (like lego) into Duck class
4. Behaviors that do not vary among subclasses can remain in the Duck abstract class.
<<Interface>>
FlyBehavior
+ fly()
Duck
+ fly: FlyBehavior
+ quack: QuackBehavior
+ swim()
+ display()
<<Interface>>
QuackBehavior
+ quack()
FlyWithWings
FlyNoWay
Quack
Squeak
Mute
2. Different strategies for FlyBehavior are implemented as subclasses
MallardDuck
RedHeadDuck
RubberDuck
quack(){ // squeak }
fly() { //NoOp }
DecoyDuck
quack(){ // NoOp }
fly() { //NoOp }
3. Subclasses just compose the behaviors they represent

```
public MallardDuck(){
  this.fly = new FlyWithWings()
  this.quack = new Quack()
}
```

```
public RubberDuck(){
  this.fly = new FlyNoWay()
  this.quack = new Squeak()
}
```