ECE4574 – Large-Scale SW Development for Engineering Systems
Lecture 12 – Design Patterns for Engineering Systems

Creed Jones, PhD

Course Updates

- Project Sprint 1 Report is due TONIGHT, 11:59 PM
  - Group submission
- Sprint 2 starts TODAY
- Homework 2 is due this Friday, October 13
- Quiz 5 is next Monday, October 15
  - covers lectures 12-13
  - open 7 PM to 1 AM

Topics for Today

Some design patterns for engineering computation

- The State pattern
  - An SIRD disease simulation
- The Blackboard pattern
- The Template Method pattern
- The Composite pattern
- The Method-Command-Strategy pattern

- A few references

SOME DESIGN PATTERNS FOR ENGINEERING COMPUTATION

I want to discuss a few design patterns that are especially applicable to engineering systems and calculation tasks

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of a solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".

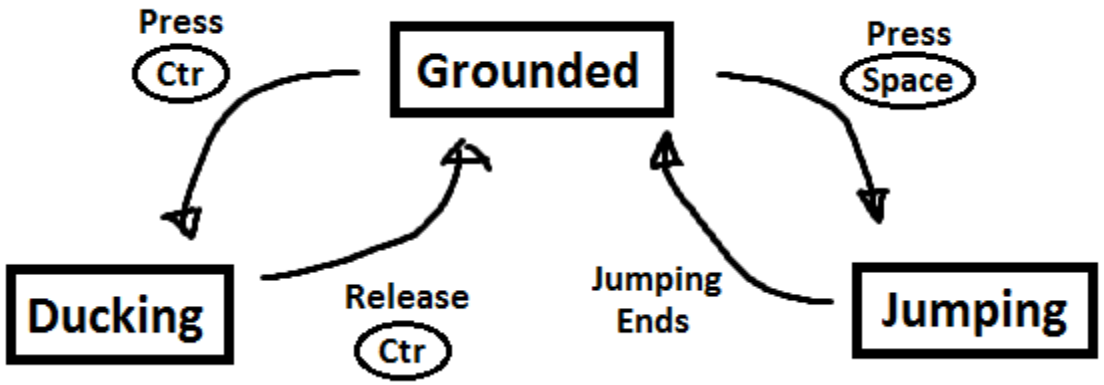
Alexander et al, *A Pattern Language*

- The State pattern
- The Blackboard pattern
- The Template Method pattern
- The Composite pattern
- The Method-Command-Strategy pattern

STATE

The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

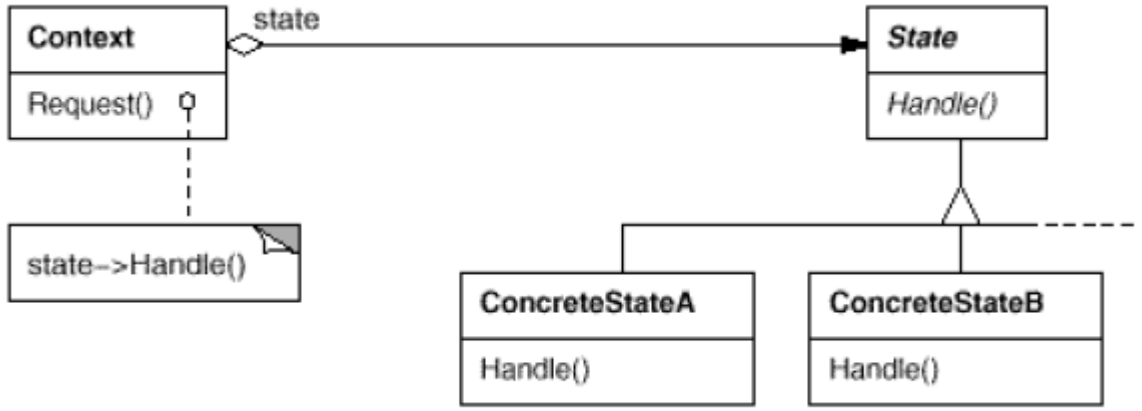
Different states are embodied in different concrete state classes, but they all implement a common interface (State) which defines the basic behavior for this entity. In one sense, all classes that implement State are actually the same "class" but the objects will behave differently at different times.



Imagine a game character that has several different types of movement, and we press keys to change the movement type or the "mode"

https://denisrizov.com/2016/03/13/state-pattern/

The State pattern allows an object to change its response to messages, etc., when its internal state changes



The different states are instantiated in different derived classes, each with different behavior – providing a common interface through the Handle() function

State participants

Context
- defines the interface of interest to clients.
- maintains an instance of a ConcreteState subclass that defines the current state.

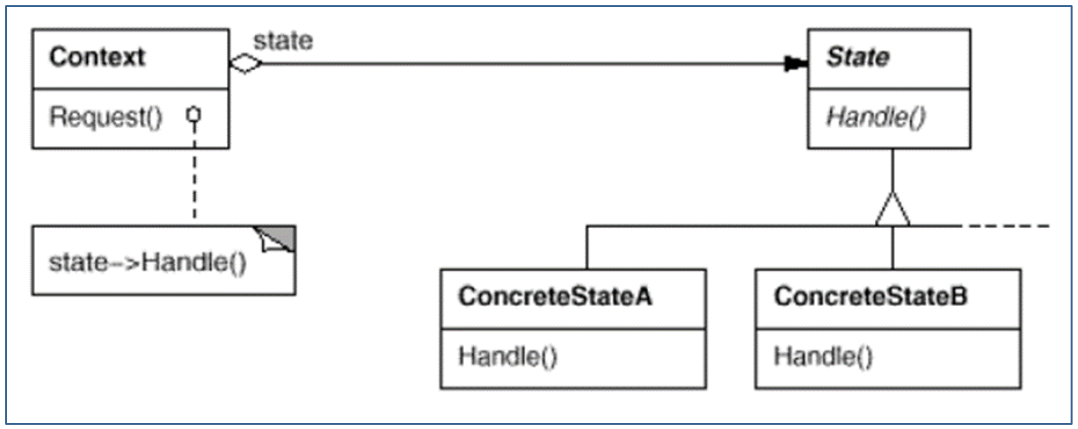
State
- defines an interface for encapsulating the behavior associated with a particular state of the Context.

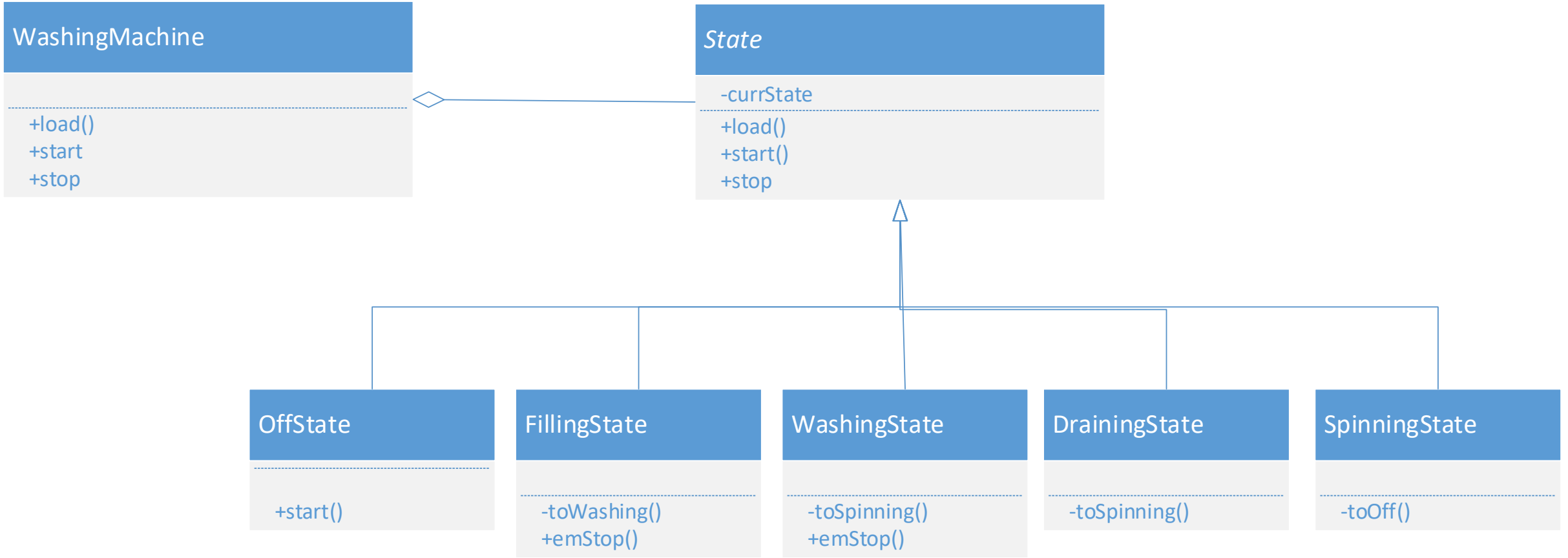
ConcreteState subclasses
- each subclass implements a behavior associated with a state of the Context

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.

Features of the State pattern

- One concrete class defined for each possible state
  - All of them inherit from *State*
- The state transition diagram (what it takes to move from state to state) is usually defined in the concrete state classes
  - In very simple situations, transition logic can be in the Context class
- I don’t like these names and usually change them
- The State pattern is unwieldy with many states or with a dynamic list of states
  - It does fine with dynamically changing state transitions


Context
state
Request()
State
Handle()
state->Handle()
ConcreteStateA
Handle()
ConcreteStateB
Handle()


WashingMachine
+load()
+start
+stop
State
-currState
+load()
+start()
+stop
OffState
+start()
FillingState
-toWashing()
+emStop()
WashingState
-toSpinning()
+emStop()
DrainingState
-toSpinning()
SpinningState
-toOff()

An infectious disease simulation using the State pattern

- I create a Population object – it contains a variable number of Patients
- Each Patient can be in one of four states:
  - Susceptible: it's possible to catch the disease
    - all Patients start in this state
  - Infected: one of two things can happen
    - after a random time, the Patient can Recover
    - at any time while Infected, the Patient has a chance of dying
  - Recovered: totally immune, nothing more can happen
  - Dead: it's over
- For my simulation, the Patient has a recovery time uniformly distributed between 0 and 40 days; after this time, they are Recovered
- However, every day while infected, they have a 0.01 probability of Death

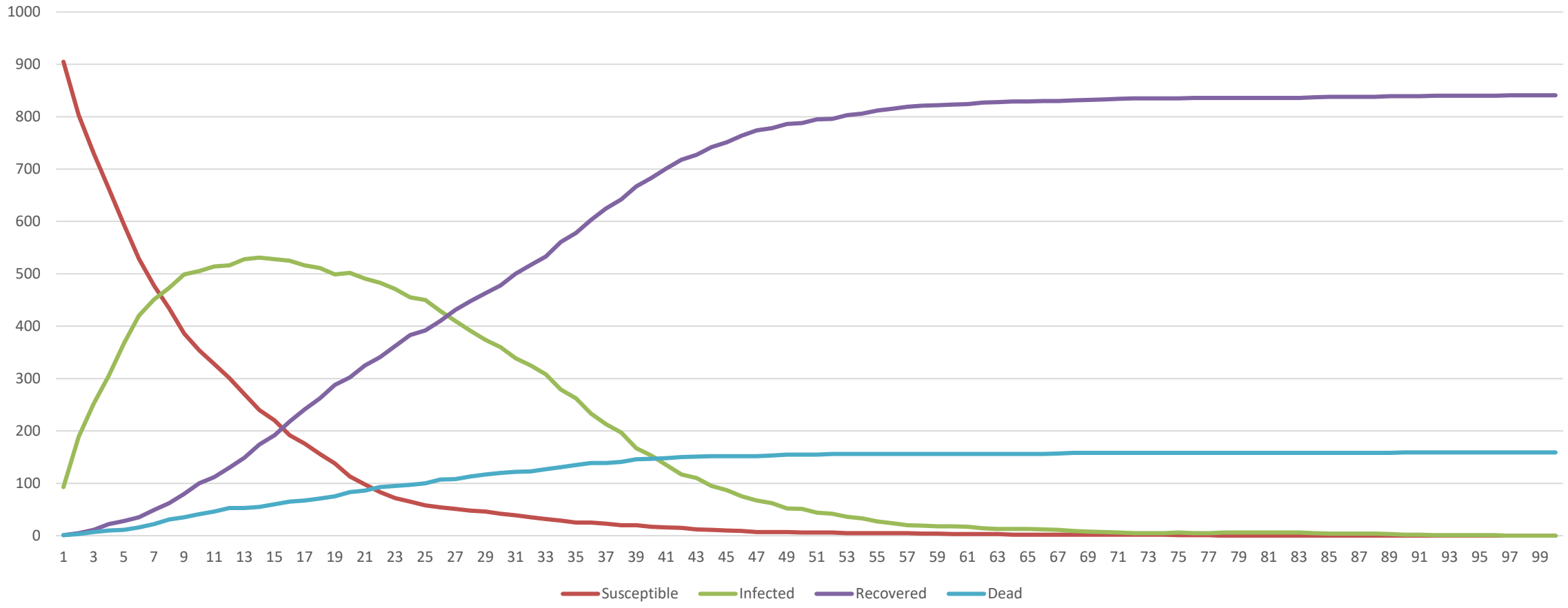
```
#pragma once

#include <iostream>
#include <string>
using namespace std;

#include "StateEvent.h"

class Patient;

class State {
protected:
    Patient* mPatient;
public:
    void setPatient(Patient* pp) { mPatient = pp; }
    virtual void onEvent(StateEvent e) = 0;
    virtual string getStateStr() = 0;
};

class Susceptible : public State {
public:
    Susceptible() {};
    void onEvent(StateEvent e);
    string getStateStr() { return "Susceptible"; }
};
```

```
class Infected : public State {
protected:
    int recoveryTime;
public:
    Infected();
    void onEvent(StateEvent e);
    string getStateStr() { return "Infected"; }
};

class Recovered : public State {
public:
    Recovered() {
    }
    void onEvent(StateEvent e) {
    }
    string getStateStr() { return "Recovered"; }
};

class Dead : public State {
public:
    Dead() {
    }
    void onEvent(StateEvent e) {
    }
    string getStateStr() { return "Dead"; }
};
```

```
#include "Patient.h"

void Susceptible::onEvent(StateEvent e) {
    switch (e) {
    default:
        break;
    case StateEvent::INFECT:
        this->mPatient->setCurrent(new Infected());
        cout << "Patient #" << this->mPatient->getMNum() << " is now Infected" << endl;
        delete this;
    }
}

Infected::Infected() { recoveryTime = rand() % 40; }

void Infected::onEvent(StateEvent e) {
    switch (e) {
    default:
        break;
    case StateEvent::TIME:
        if (--recoveryTime < 0) {
            this->mPatient->setCurrent(new Recovered());
            cout << "Patient #" << this->mPatient->getMNum() << " is now Recovered" << endl;
            delete this;
        }
        else if (rand() % 100 == 0) {
            this->mPatient->setCurrent(new Dead());
            cout << "Patient #" << this->mPatient->getMNum() << " is now Dead" << endl;
            delete this;
        }
    }
}
```

Here are simulation results for a Population of 1000 Patients; after 100 days, 841 Patients have recovered – but 159 have died

This is a well-known SIR (or SIRD) curve

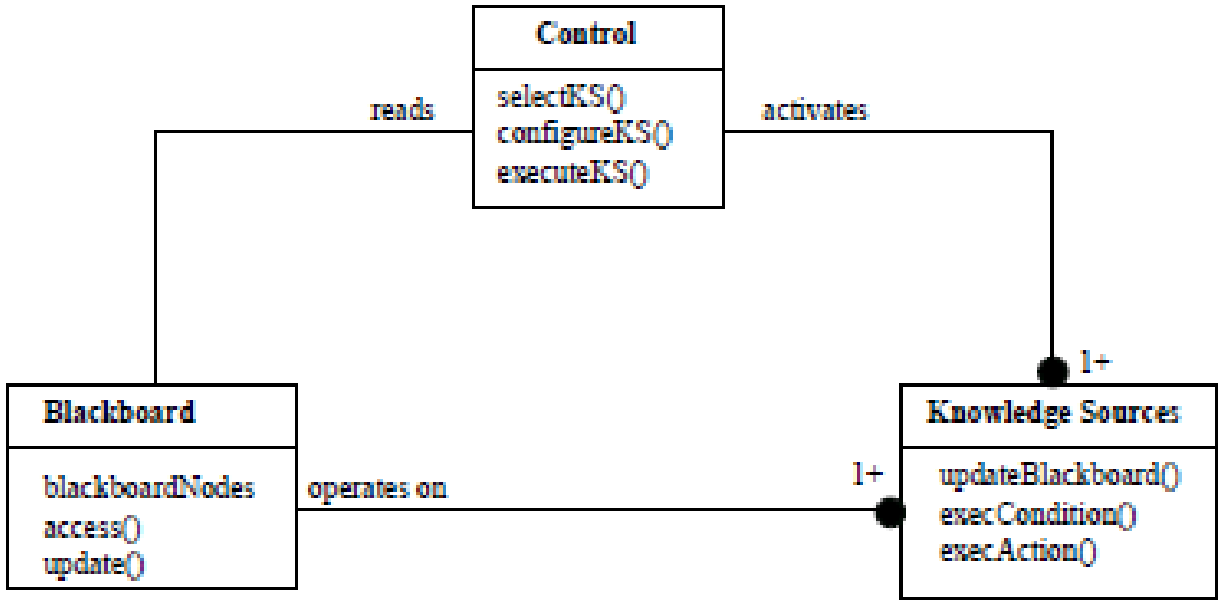


BLACKBOARD

The Blackboard Pattern supports processing on several "knowledge sources" by independent processes

- In the process of performing some task, information is posted to and updated in a central location called the "Blackboard"
- Knowledge sources can be human or automated, static or learning
- The Control (or control shell) manages communication and mediates

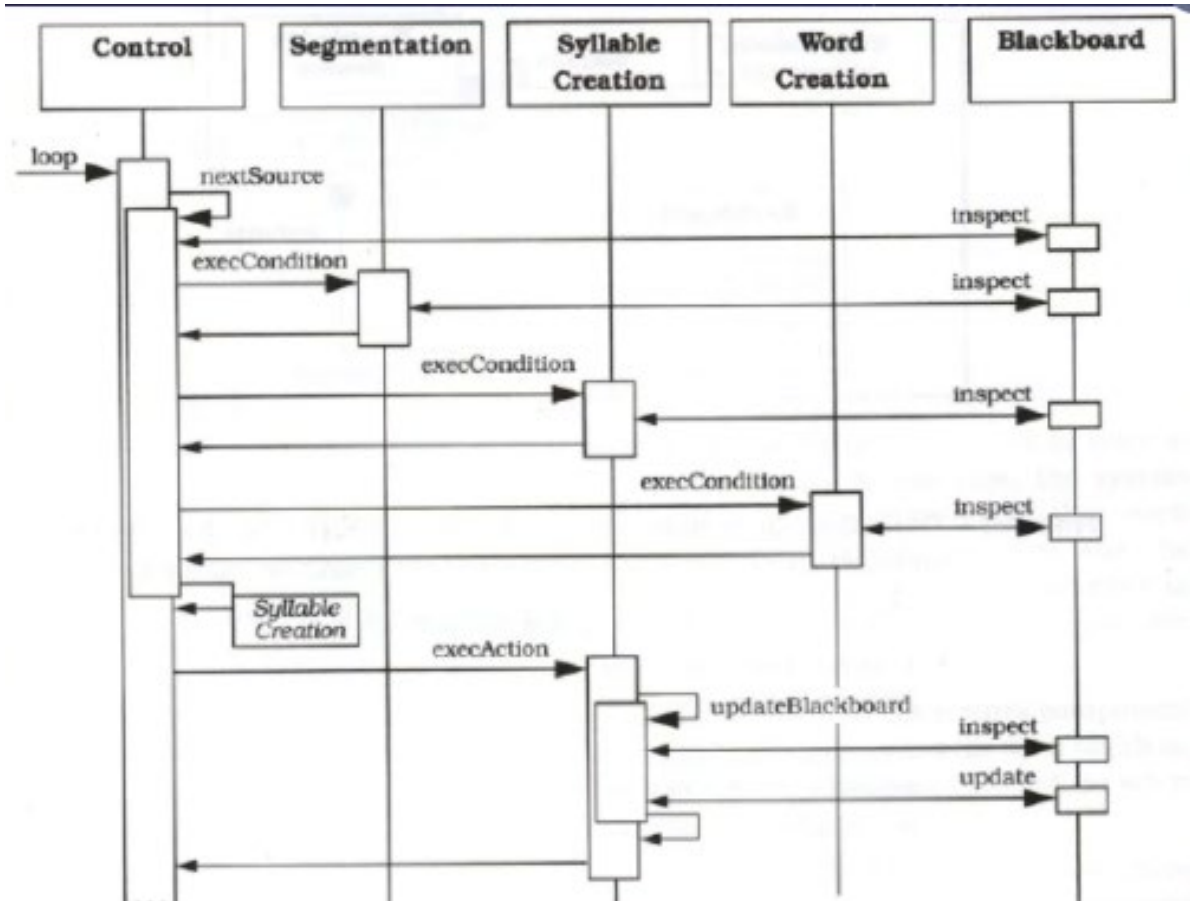
- Good support for adding new knowledge sources



The Blackboard pattern is often used in speech recognition systems

- It's common to use several different algorithms for speech recognition
  - These are the Knowledge sources
- Each KS updates the blackboard with the results of analysis of a speech sample
  - Which may depend on history
- The controller compares results and makes the overall decision

- Thanks to Andy Bulka – Melbourne Patterns Group



There's a nice example from Microsoft on applying the Blackboard pattern to real-time radar data analysis

- Regrettably, it's implemented in C#
  - ☹
- But, it's a great example of an engineering application
  - Real-time
  - Radar data
  - Scalable to large scenarios

https://social.technet.microsoft.com/wiki/contents/articles/13461.blackboard-design-pattern-a-practical-c-example-radar-defence-system.aspx

Participants in the Blackboard Pattern

Blackboard

- manages central data
- for solution space and control data
- provides interface for read/write by knowledge sources
- "hypothesis" or "blackboard entry" is a solution constructed during the problem solving process

Knowledge Source

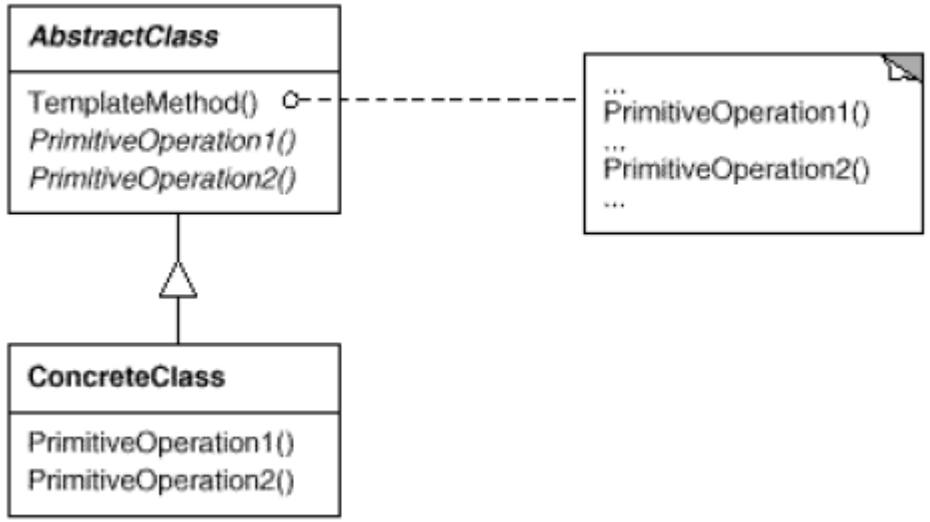
- Evaluates its own applicability
- Computes a result
- Updates the Blackboard (they do not communicate directly)

Control

- monitor changes on the Blackboard
- select a knowledge source and decide what action to take next according to a knowledge application strategy
- the basis for this strategy is the data on the blackboard.

TEMPLATE METHOD

The Template Method pattern facilitates deferral of some implementation details until subclasses are implemented



- The Template Method pattern can be seen as a basic application of object-oriented design
  - Polymorphism
- The more interesting thing it provides is a place to define the sequence of steps in an algorithm
  - And for those steps, precise implementation can be defined later and can depend on object type

Template Method participants

AbstractClass

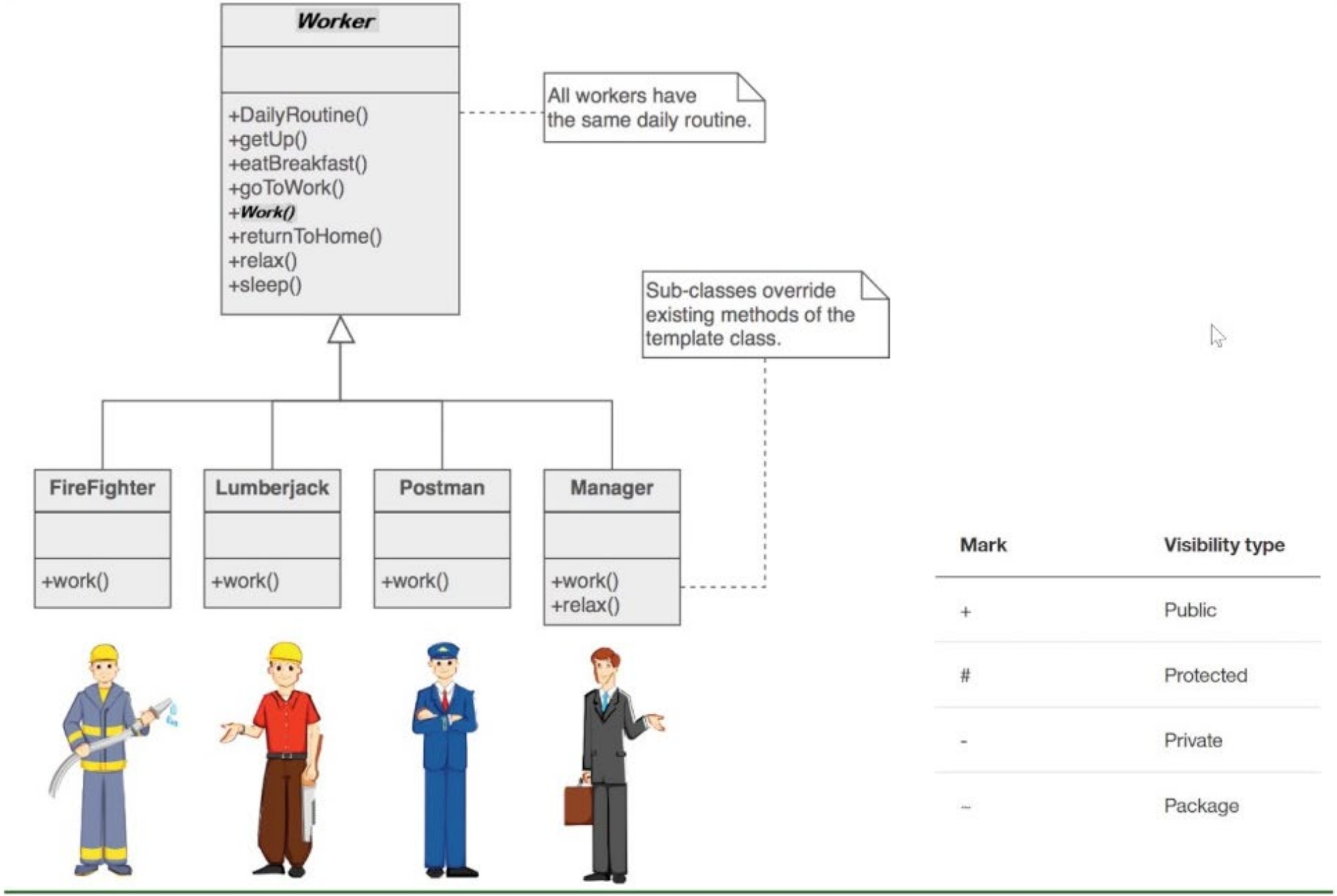
- defines abstract **primitive operations** that concrete subclasses define to implement steps of an algorithm.
- implements a ***template method*** defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.

ConcreteClass

- implements 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.



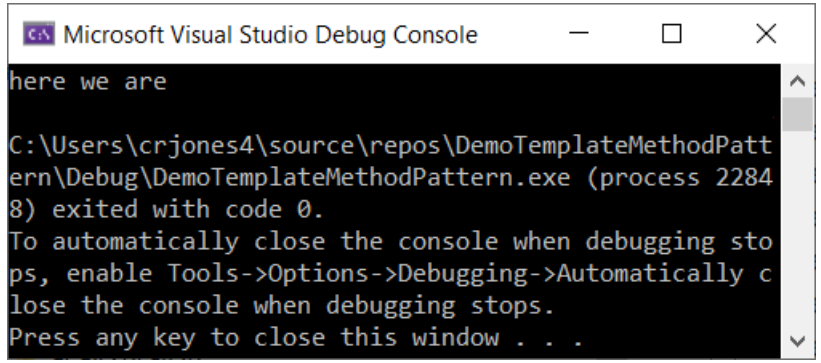
Mark	Visibility type
+	Public
#	Protected
-	Private
~	Package

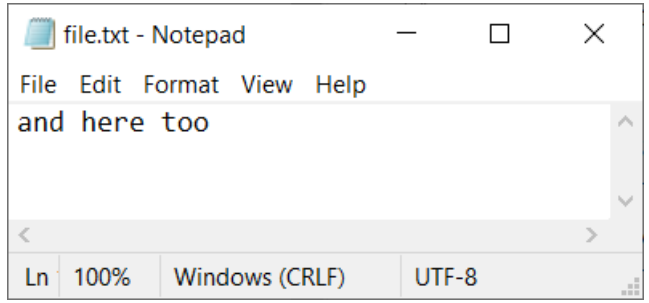
https://www.youtube.com/watch?v=-pqZCBIZpKk

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

class Logger {
protected:
    string mName;
    virtual void open(string nm) { mName = nm; };
    virtual void writeTo(string msg) = 0;
    virtual void flush() = 0;
    virtual void close() {};
    void clearName() { mName = "";  }
public:
    Logger() {}
    void write(string nm, string msg) {
                    // defines the algorithm flow
        open(nm);
        writeTo(msg);
        flush();
        close();
    }
    string getName() { return mName; }
};
```

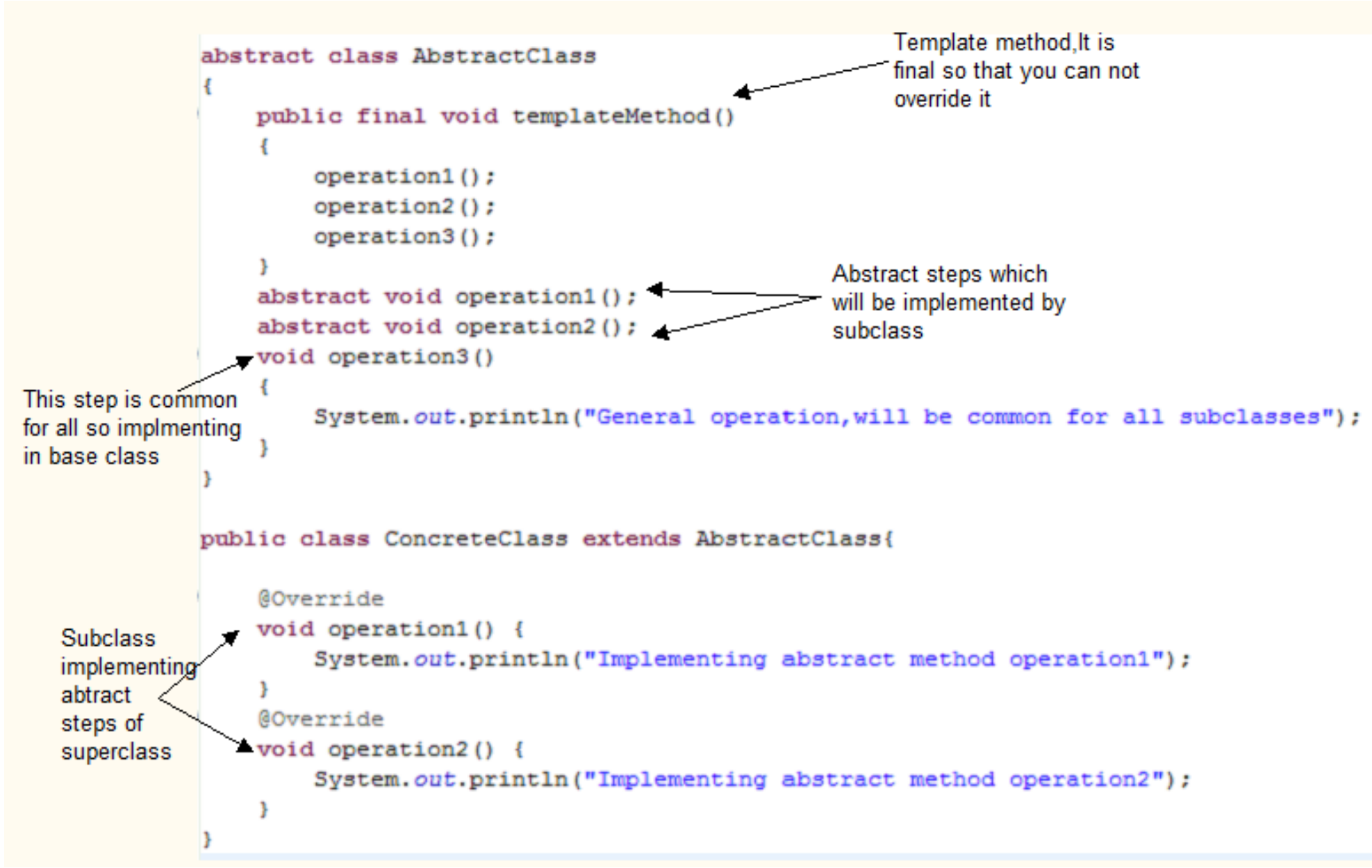
```
class FileLogger : public Logger {
private:
    ofstream oFile;
    void open(string nm) { mName = nm; oFile.open(nm);
}
    void writeTo(string msg) { oFile << msg; }
    void flush() { oFile.flush(); }
    void close() { oFile.close(); }
};

class ConsoleLogger : public Logger {
private:
    void writeTo(string msg) { cout << msg; }
    void flush() { cout << endl; }
};

int main()
{
    Logger* cLog = new ConsoleLogger();
    Logger* fLog = new FileLogger();
    cLog->write("console", "here we are");
    fLog->write("file.txt", "and here too");
}
```

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

class Logger {
protected:
    string mName;
    virtual void open(string nm) { mName = nm; };
    virtual void writeTo(string msg) = 0;
    virtual void flush() = 0;
    virtual void close() {};
    void clearName() { mName = "";  }
public:
    Logger() {}
    void write(string nm, string msg) {
                    // defines the algorithm flow
        open(nm);
        writeTo(msg);
        flush();
        close();
    }
    string getName()
};
```

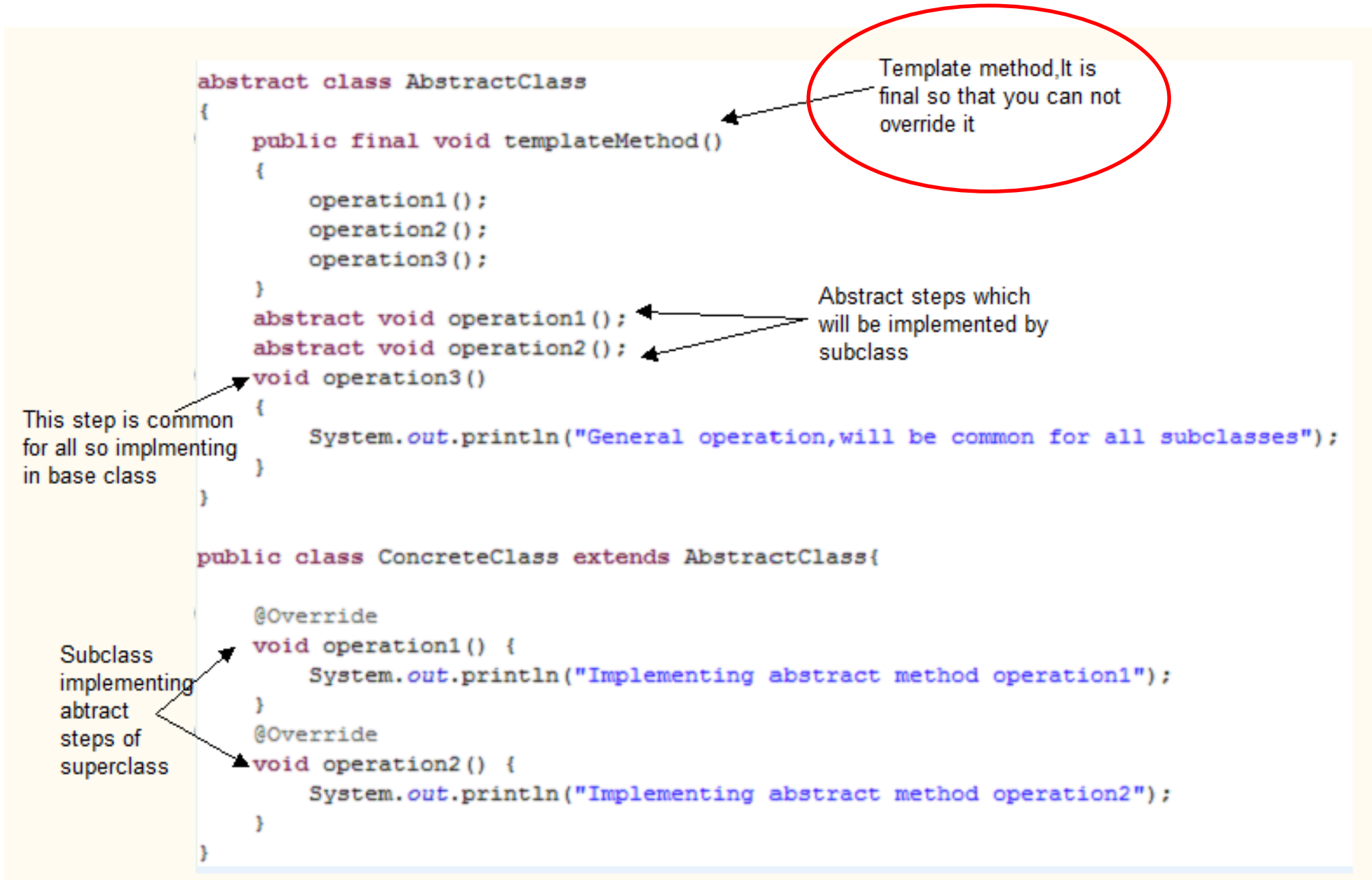





```
class FileLogger : public Logger {
private:
    ofstream oFile;
    void open(string nm) { mName = nm; oFile.open(nm);
}
    void writeTo(string msg) { oFile << msg; }
    void flush() { oFile.flush(); }
    void close() { oFile.close(); }
};

class ConsoleLogger : public Logger {
private:
    void writeTo(string msg) { cout << msg; }
    void flush() { cout << endl; }
};

int main()
{
    Logger* cLog = new ConsoleLogger();
    Logger* fLog = new FileLogger();
    cLog->write("console", "here we are");
    fLog->write("file.txt", "and here too");
}
```



http://iblogfreeshare.blogspot.com/2013/03/template-method-design-pattern-in-java.html

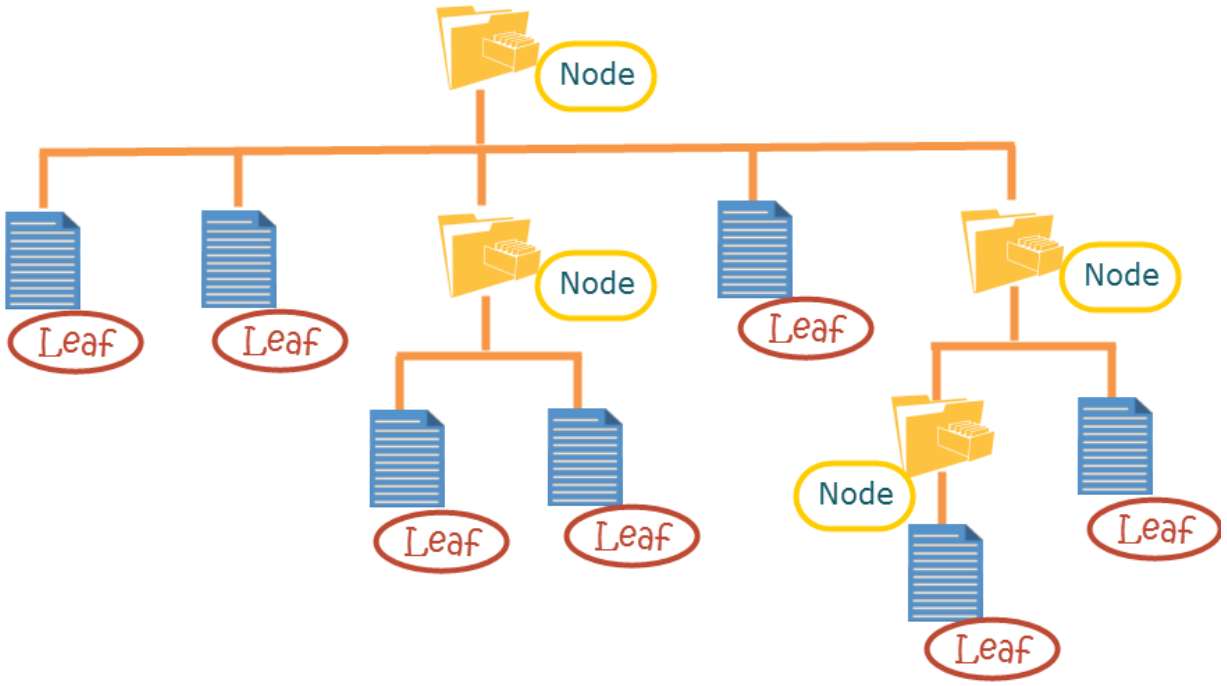



```
abstract class AbstractClass
{
    public final void templateMethod()
    {
        operation1();
        operation2();
        operation3();
    }
    abstract void operation1();
    abstract void operation2();
    void operation3()
    {
        System.out.println("General operation,will be common for all subclasses");
    }
}

public class ConcreteClass extends AbstractClass{

    @Override
    void operation1() {
        System.out.println("Implementing abstract method operation1");
    }
    @Override
    void operation2() {
        System.out.println("Implementing abstract method operation2");
    }
}
```

http://iblogfreeshare.blogspot.com/2013/03/template-method-design-pattern-in-java.html

COMPOSITE

The Composite Pattern allows you to compose objects into tree structures to represent part-whole hierarchies.

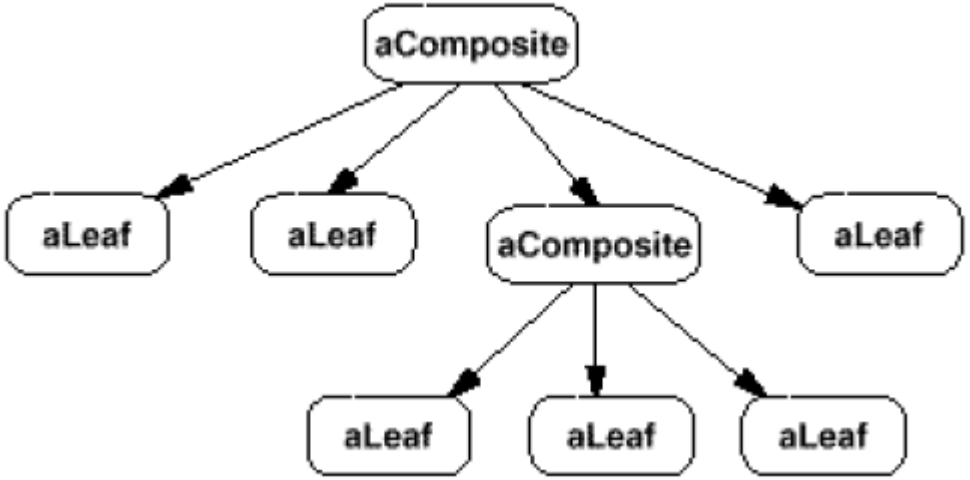


Composite lets clients treat individual objects and compositions of objects uniformly.

In a typical file system, a folder can contain documents or other folders, which can contain documents or other folders, which…

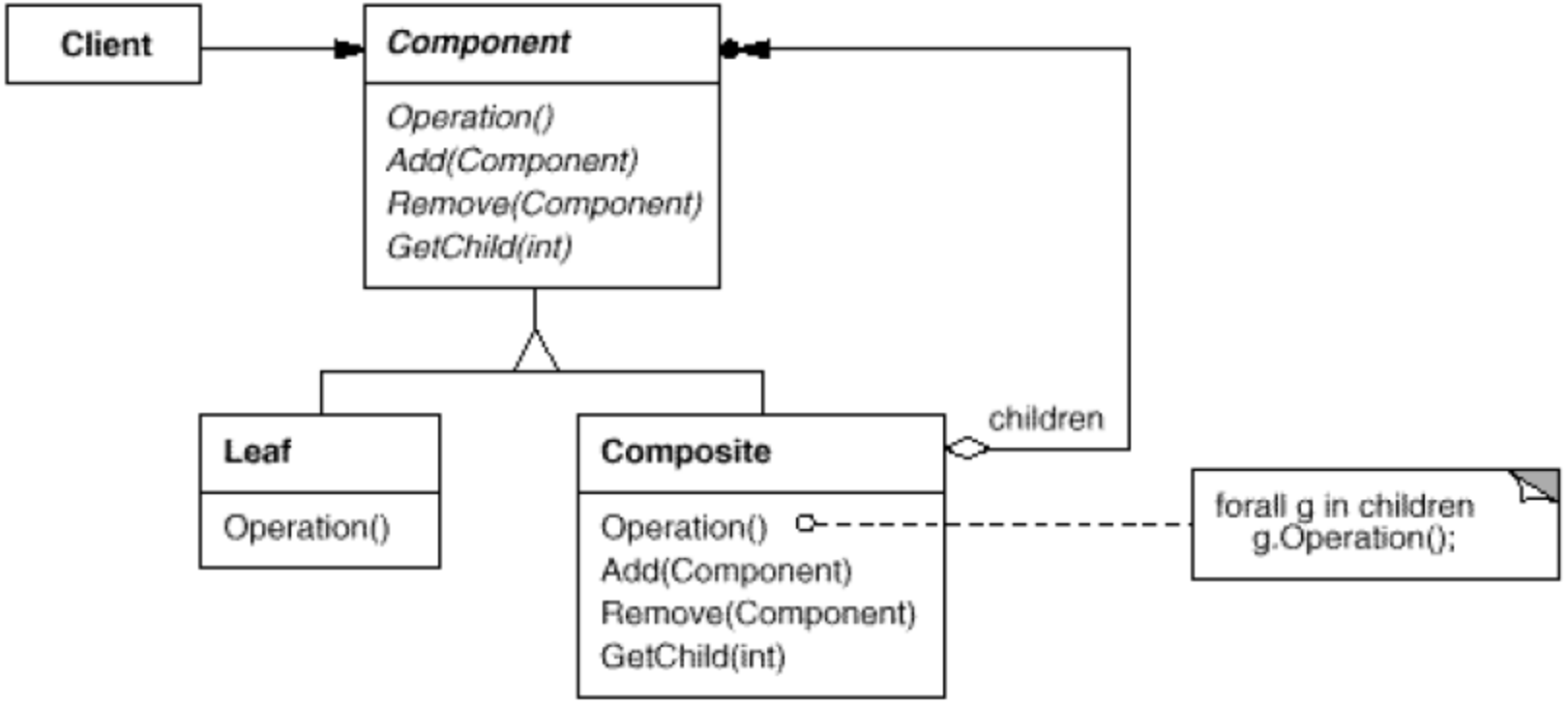
The Composite Pattern allows related objects to be treated as a whole, in a defined hierarchy

- When objects form a tree, it's nice to have a consistent way to operate on the whole tree or on a subtree…



- The Composite pattern describes an inheritance strategy to facilitate this

In the Composite pattern, both the Composite itself and its Leaf members inherit from the *Component* class



Participants in the Composite Pattern

Component
- declares the interface for objects in the composition.
- implements default behavior for the interface common to all classes, as appropriate.
- declares an interface for accessing and managing its child components.
- (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.

Leaf
- represents leaf objects in the composition. A leaf has no children.
- defines behavior for primitive objects in the composition.

Composite
- defines behavior for components having children.
- stores child components.
- implements child-related operations in the Component interface

Client
- manipulates 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.

METHOD-COMMAND-STRATEGY

A design pattern specifically intended for high-performance computing (HPC) was introduced in the late 2000's and has gotten some attention

Method-Command-Strategy: A Pattern for Numerical Methods on Object-Oriented High Performance Computing

Tiago Quintino[1,2], Andrea Lani[1], Herman Deconinck[1], Stefan Vandewalle[2]

[1] Von Karman Institute, Aerospace Dept., Chaussee de Waterloo 72, B-1640 Sint-Genesius-Rode, Belgium
phone: +3223599611, fax: +3223599600, e-mail: quintino@vki.ac.be

[2] Catholic University Leuven, Computer Science Dept., Celestijnenlaan 200A, B-3001 Leuven, Belgium

Abstract. Developing architectural solutions for high performance computing software able to conjugate great flexibility and efficiency is an open challenge because one is often confronted with run-time trade-offs. This paper presents and analyzes a design pattern, the Method-Command-Strategy, specifically tailored to scientific computing applications, that allows to encapsulate numerical methods in a modular and extensible way without affecting peak performance.

Keywords: design patterns, high performance computing

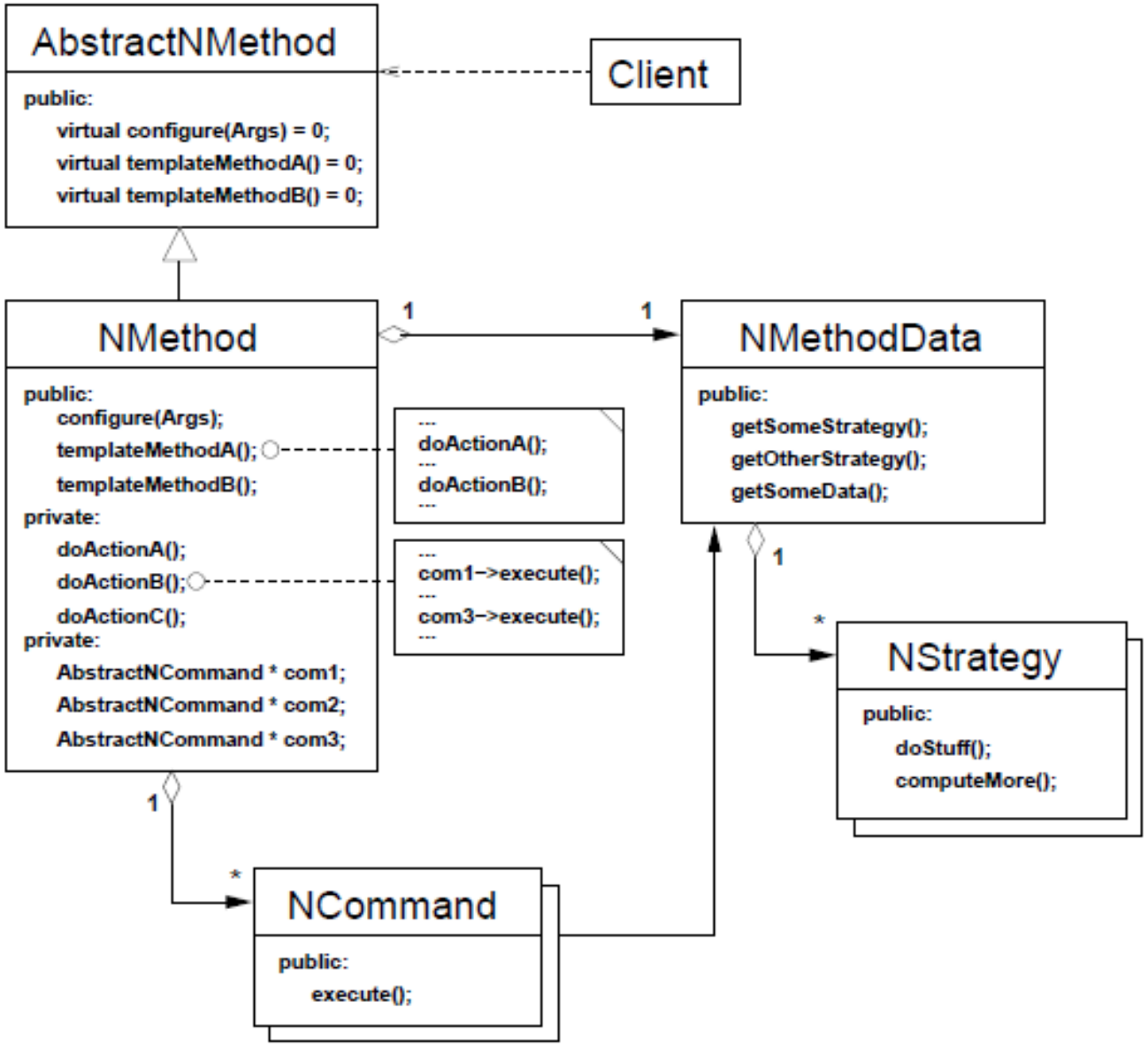
1 Introduction

The concept of Design Patterns, introduced in 1995 by [1], and quickly accepted by the developer community, brought higher levels of understanding to software engineering and began to structure the way software architecture is perceived.

So far, for reasons out of scope of this work, these object-oriented techniques have taken their time to find a foothold in the specific domain of high performance scientific computing (HPC).

In such applications, design decisions must take into account the run-time efficiency of the software, while keeping in mind more common computer science aims like flexibility, modularity and extensibility. This is mostly a compromise effort that often results in solutions that don't satisfy neither aims completely, rendering them unacceptable from the HPC point of view.

The goal of the Method-Command-Strategy pattern is to allow specific numerical optimizations to be determined by data at run time, without exposing all of this complexity



Method-Command-Strategy participants (from the Quintino paper)

- **Client**: the object requesting the algorithm execution, typically the driver of the scientific simulation or other NMethods that acquaint the current one.
- **AbstractNMethod**: defines an interface to the method, possibly IDL component interface, to be handled by the numerical simulation.
- **NMethod**: defines a skeleton for each step of the numerical method, calling hook actions and concrete (defined) actions. Each function defining the step is a Template Method and each action is the Invoker in the Command pattern.
- **AbstractNCommand**: exposes the interface to the NCommand hierarchic tree. Has acquaintance of NMethodData.
- **NCommand**: performs a hook action for the NMethod to call. Might be fully implemented, thus having no Receivers or partly implemented in which case it defers the rest to NStrategy objects. This is a Command pattern, where the Receivers can be NStrategies, NMethodData or other objects that have been setup in the concrete NCommand.
- **NMethodData**: aggregates all the NStrategies to be shared between the NCommands. Works as a repository for common data particular to the NMethod.
- **AbstractNStrategy**: defines the interface for a family of fine grained algorithms that will be used by the NCommands. This is a typical application of the Strategy pattern .
- **NStrategy**: defines one fine-grained algorithm that can be interchanged with others. It is a Receiver of the NCommands.

Method-Command-Strategy collaborations (from the Quintino paper)

- The NMethod defers the implementation of the hook actions to various NCommands. Just as in the Template Method, the NMethod fixes the order of execution of the actions.
- The NMethodData is *acquainted* by all the NCommands as a way of sharing objects between them, both data and NStrategies. It works as a unified point of access to the commands receivers.
- The Client has to configure the NMethod with parameters for him to choose the correct NCommands and place the chosen NStrategies in NMethodData.
- The NCommand encapsulates the requested action, implementing it or deferring it to the NStrategies.

When would you consider using this Method-Command-Strategy design pattern?

When:

- the multiple parts of a numerical method can be cast in static skeletons with predefined order of actions;
- the implementation of the method must be as efficient as possible without compromising flexibility;
- the actions that compose the method have an open set of various implementations - possibly to be added via a plug-in policy; and
- when specific actions need to be tailored to specific requirements, usually related to efficiency and simplification of the algorithms based on some specific type binding, thus requiring encapsulation of such implementation.

SOME FURTHER READING

Barrera et al, Using Design Patterns to Solve Newton-type Methods , in *Trends and Applications in Software Engineering*, Advances in Intelligent Systems and Computing 537, DOI 10.1007/978-3-319-48523-2_10

Using Design Patterns to Solve Newton-type Methods

Ricardo Serrato Barrera[1], Gustavo Rodríguez Gómez[2], Saúl Eduardo Pomares Hernández[2], Julio César Pérez Sansalvador[3], and Leticia Flores Pulido[4],

[1] Estratei Sistemas de Información, S.A. de C.V., Virrey de Mendoza 605-B, Col. Las fuentes, 59699, Zamora, Michoacán, México
[2] Instituto Nacional de Astrofísica, Óptica y Electrónica, Coordinación de Ciencias Computacionales, Luis Enrique Erro 1, 72840, Tonantzintla, Puebla, México
[3] Instituto Nacional de Astrofísica, Óptica y Electrónica, Laboratorio de Visión por Computadora, Luis Enrique Erro 1, 72840, Tonantzintla, Puebla, México
[4] Universidad Autónoma de Tlaxcala, Facultad de Ciencias Básicas, Ingeniería y Tecnología, Calzada Apizaquito, Colonia Apizaquito, 90300, Apizaco, Tlaxcala, México
{rsbserrato, grodrig, spomares, tachidok}@ccc.inaoep.mx, leticia.florespo@udlap.mx

Abstract. We present the development of a software system for Newton-type methods via the identification and application of software design patterns. We measured the quality of our developed system and found that it is flexible, easy to use and extend due to the application of design patterns. Our newly developed system is flexible enough to be used by the numerical analyst interested in the creation of new Newton-type methods, or the engineer that applies different Newton--type strategies in his software solutions.

Keywords: Newton-type methods, design patterns, object-oriented, software design, scientific software.

Kees and Miller spend a lot of time thinking about the design of their solver in C++ - but did not choose to use a known pattern

Christopher E. Kees and Cass T. Miller. 1999. C++ implementations of numerical methods for solving differential-algebraic equations: design and optimization considerations. ACM Trans. Math. Softw. 25, 4 (Dec. 1999), 377–403. DOI:https://doi.org/10.1145/332242.334001

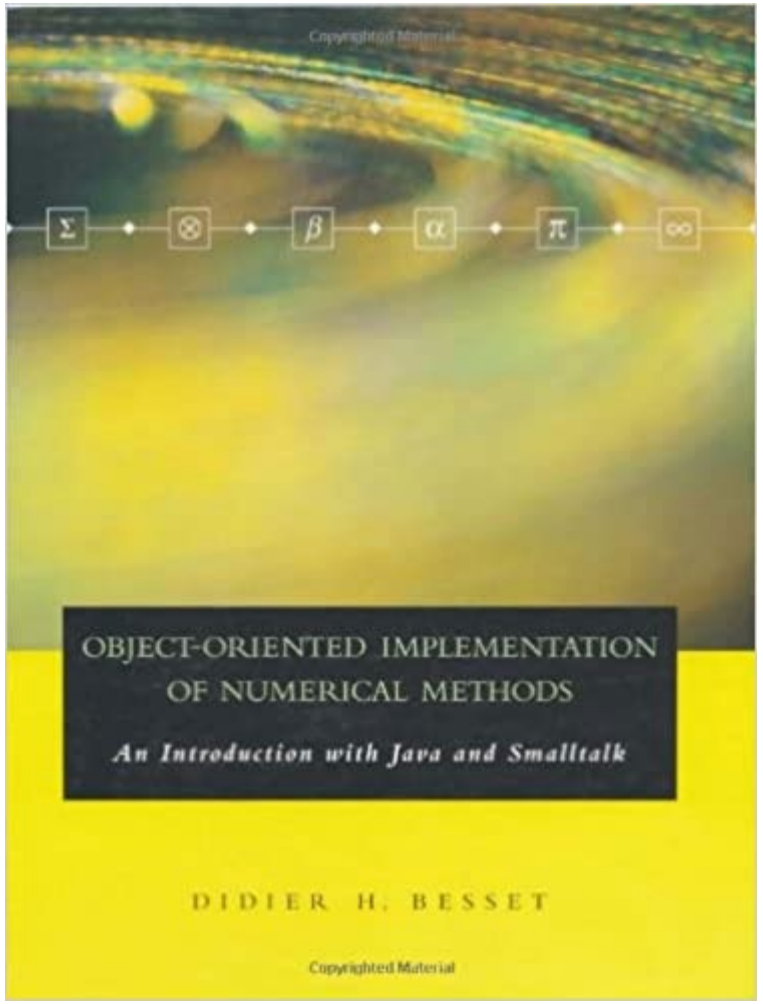
C++ Implementations of Numerical Methods for Solving Differential-Algebraic Equations: Design and Optimization Considerations

CHRISTOPHER E. KEES and CASS T. MILLER
The University of North Carolina, Chapel Hill

Object-oriented programming can produce improved implementations of complex numerical methods, but it can also introduce a performance penalty. Since computational simulation often requires intricate and highly efficient codes, the performance penalty of high-level techniques must always be weighed against the improvements they enable. These issues are addressed in a general object-oriented (OO) toolkit for the numerical solution of differential-algebraic equations (DAEs). The toolkit can be configured in several different ways to solve DAE initial-value problems with an adaptive multistep method. It contains a wrapped version of the Fortran 77 code DASPK and a translation of this code to C++. Two C++ constructs for assembling the tools are provided, as are two implementations of an important DAE test problem. Multiple configurations of the toolkit for DAE test problems are compared in order to assess the performance penalties of C++. The mathematical methods and implementation techniques are discussed in detail in order to provide heuristics for efficient OO scientific programming and to demonstrate the effectiveness of OO techniques in managing complexity and producing better code. The codes were tested on a variety of problems using publicly available Fortran 77 and C++ compilers. Extensive efficiency comparisons are presented in order to isolate computationally inefficient OO techniques. Techniques that caused difficulty in implementation and maintenance are also highlighted. The comparisons demonstrate that the majority of C++'s built-in support for OO programming has a negligible effect on performance, when used at sufficiently high levels, and provides flexible and extensible software for numerical methods.

Besset, D. H. (2001). Object-oriented implementation of numerical methods : an introduction with java and smalltalk. Morgan Kaufmann.

(in VT library)



Topics for Today

Some design patterns for engineering computation

- The State pattern
  - An SIRD disease simulation
- The Blackboard pattern
- The Template Method pattern
- The Composite pattern
- The Method-Command-Strategy pattern

- A few references