**Singleton Pattern**

Often there will be situations where there needs to be one and only one instance of a class. Rather than relying on the programmer to create only one instance of a class, the *Singleton* pattern will create only one instance of a class, and makes that instance accessible by other objects.

Motivation

Sometimes it's important to have only one instance for a class. For example, in a system there should be only one window manager (or only a file system or only a print spooler). Usually singletons are used for centralized management of internal or external resources and they provide a global point of access to themselves.

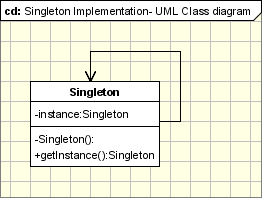
The singleton pattern is one of the simplest design patterns: it involves only one class which is responsible to instantiate itself, to make sure it creates not more than one instance; in the same time it provides a global point of access to that instance. In this case the same instance can be used from everywhere, being impossible to invoke directly the constructor each time.

Intent

Ensure that only one instance of a class is created. Provide a global point of access to the object.

Implementation

The implementation involves a static member in the "Singleton" class, a private constructor and a static public method that returns a reference to the static member.



The Singleton Pattern defines a getInstance operation which exposes the unique instance which is accessed by the clients. getInstance() is is responsible for creating its class unique instance in case it is not created yet and to return that instance.

class Singleton

{

private static Singleton instance;

private Singleton()

{

...

}

public static synchronized Singleton getInstance()

{

if (instance == null)

instance = new Singleton();

return instance;

}

...

public void doSomething()

{

...

}

}

You can notice in the above code that getInstance method ensures that only one instance of the class is created. The constructor should not be accessible from the outside of the class to ensure the only way of instantiating the class would be only through the getInstance method.

The getInstance method is used also to provide a global point of access to the object and it can be used in the following manner:

Singleton.getInstance().doSomething();

**Iterator**

The *Iterator* pattern is used to provide a means to access all the elements of some collection of objects sequentially without exposing the collection's internal representation. The Iterator is so common and useful that Java provides iterators for its collection objects as a standard feature.

Motivation

One of the most common data structures in software development is what is generic called a collection. A collection is just a grouping of some objects. They can have the same type or they can be all cast to a base type like object. A collection can be a list, an array, a tree and the examples can continue.

But what is more important is that a collection should provide a way to access its elements without exposing its internal structure. We should have a mechanism to traverse in the same way a list or an array. It doesn't matter how they are internally represented.

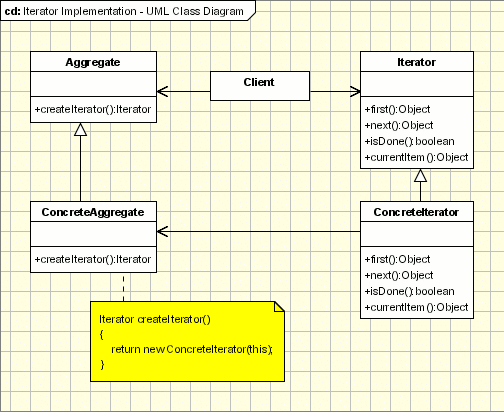
The idea of the iterator pattern is to take the responsibility of accessing and passing trough the objects of the collection and put it in the iterator object. The iterator object will maintain the state of the iteration, keeping track of the current item and having a way of identifying what elements are next to be iterated.

Intent

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

The abstraction provided by the iterator pattern allows you to modify the collection implementation without making any changes outside of collection. It enables you to create a general purpose GUI component that will be able to iterate through any collection of the application.

Implementation



Applicability & Examples

The iterator pattern allow us to:

access contents of a collection without exposing its internal structure.

support multiple simultaneous traversals of a collection.

provide a uniform interface for traversing different collection.

Example 1: This exmple is using a collection of books and it uses an iterator to iterate through the collection. The main actors are:

IIterator - This interface represent the AbstractIterator, defining the iterator

BookIterator - This is the implementation of Iterator(implements the IIterator interface)

IContainer - This is an interface defining the Agregate

BooksCollection - An implementation of the collection

Here is the code for the abstractions IIterator and IContainer:

interface IIterator

{

public boolean hasNext();

public Object next();

}

interface IContainer

{

public IIterator createIterator();

}

And here is the code for concrete classes for iterator and collection. Please note that the concrete iterator is an nested class. This way it can access all the members of the collection and it is encapsulated so other classes can not access the BookIterator. All the classes are not aware of BookIterator they uses the IIterator:

class BooksCollection implements IContainer

{

private String m\_titles[] = {"Design Patterns","1","2","3","4"};

public IIterator createIterator()

{

BookIterator result = new BookIterator();

return result;

}

private class BookIterator implements IIterator

{

private int m\_position;

public boolean hasNext()

{

if (m\_position < m\_titles.length)

return true;

else

return false;

}

public Object next()

{

if (this.hasNext())

return m\_titles[m\_position++];

else

return null;

}

}

}

Specific problems and implementation

Iterator and multithreading

Several problems may appear when collections are added from different threads. First of all let's see which the basic steps when using an iterator are:

Step one: the collection return a new iterator (using in our example the createIterator method). Usually this step is not affected when it is used in multithreading environments because it returns a new iterator object.

Step two: The iterator is used for iterating through the objects. Since the iterators are different objects this step is not a problematic one in multithreading environments.

It seems that the iterator does not raise special problems when a collection is used from different threads. Of course here we are talking about an "seems". To reformulate the iterator does not raise special problems when the collection used from different threads as long the collection is not changed.

Let's analyze each case:

A new element is added to the collection (at the end). The iterator should be aware of the new size of the collection and to iterate till the end.

A new element is added to the collection before the current element. In this case all the iterators of the collection should be aware of this.

The same actions should occur when an element is removed from the collection. The iterators should be aware of the changes.

The main task when creating a multithreading iterator is to create a robust iterator (that allows insertions and deletions without affection transversal). Then the blocks which are changing or accessing resources changed by another thread have to be synchronized.

**Observer**

The *Observer* pattern is used when any number of objects (the Observers) need to be notified automatically whenever another object (the Observable) changes its state.

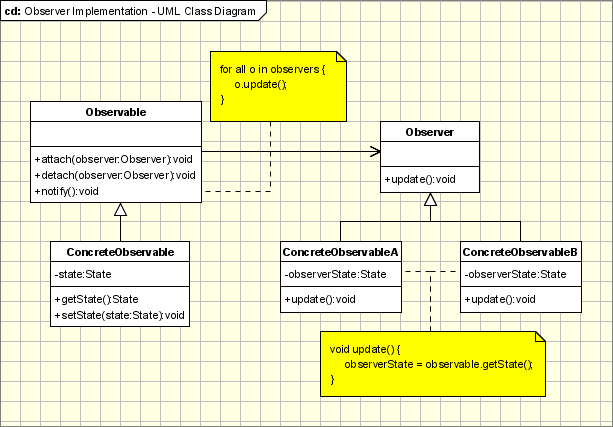
Motivation

We cannot talk about Object Oriented Programming without considering the state of the objects. After all object oriented programming is about objects and their interaction. The cases when certain objects need to be informed about the changes occured in other objects are frequent. To have a good design means to decouple as much as possible and to reduce the dependencies. The Observer Design Pattern can be used whenever a subject has to be observed by one or more observers.

Let's assume we have a stock system which provides data for several types of client. We want to have a client implemented as a web based application but in near future we need to add clients for mobile devices, Palm or Pocket PC, or to have a system to notify the users with sms alerts. Now it's simple to see what we need from the observer pattern: we need to separate the subject(stocks server) from it's observers(client applications) in such a way that adding new observer will be transparent for the server.

Intent

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



Implementation

The participants classes in this pattern are:

Observable - interface or abstract class defining the operations for attaching and de-attaching observers to the client. In the GOF book this class/interface is known as Subject.

ConcreteObservable - concrete Observable class. It maintain the state of the object and when a change in the state occurs it notifies the attached Observers.

Observer - interface or abstract class defining the operations to be used to notify this object.

ConcreteObserverA, ConcreteObserver2 - concrete Observer implementations.

The flow is simple: the main framework instantiate the ConcreteObservable object. Then it instantiate and attaches the concrete observers to it using the methods defined in the Observable interface. Each time the state of the subject it's changing it notifies all the attached Observers using the methods defined in the Observer interface. When a new Observer is added to the application, all we need to do is to instantiate it in the main framework and to add attach it to the Observable object. The classes already created will remain unchanged.

Applicability & Examples

The observer pattern is used when:

the change of a state in one object must be reflected in another object without keeping the objects tight coupled.

the framework we are writing needs to be enhanced in future with new observers with minimal changes.

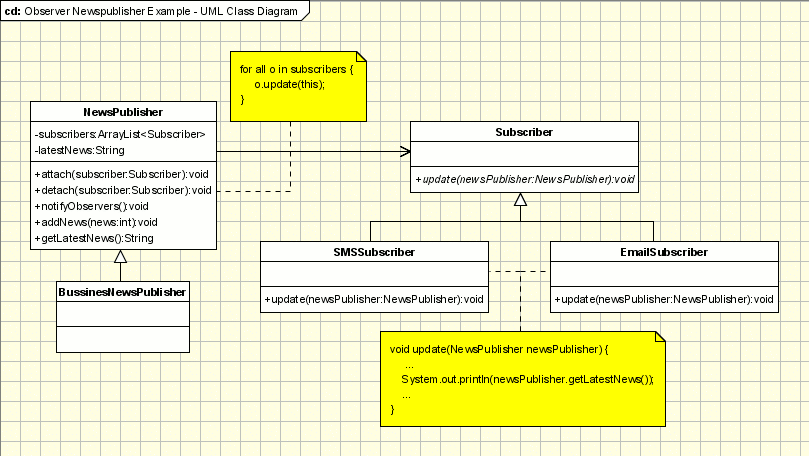
Some Classical Examples:

Model View Controller Pattern - The observer pattern is used in the model view controller (MVC) architectural pattern. In MVC the this pattern is used to decouple the model from the view. View represents the Observer and the model is the Observable object.

Event management - This is one of the domains where the Observer patterns is extensively used. Swing and .Net are extensively using the Observer pattern for implementing the events mechanism.

Example - News Agency

Lets' take the example of a news agency. A news agency gather news news and publish them to different subscribers. We need to create a framework for and agency to be able to inform immediately, when event occurs, its subscribers about the event. The subscribers can receive the news in different ways: Emails, SMS, ... The solution need to be extensively enough to support new types of subscribers(maybe new communication technologies will appear).



Obviously, the agency is represented by an Observable(Subject) class named NewsPublisher. This one is created as an abstract class because the agency want to create several types of Observable objects: in the beginning only for business news, but after some time sport and political new will be published. The concrete class is BusinessNewsPublisher.

The observer logic is implemented in NewsPublisher. It keeps a list of all it subscribers and it informs them about the latest news. The subscribers are represented by some observers (SMSSubscriber, EmailSubscriber). Both the observers mentioned above are inherited from the Subscriber. The subscriber is the abstract class which is known to the publisher. The publisher doesn't know about concrete observers, it knows only about their abstraction.

In the main class a publisher(Observable) is built and a few subscribers(Observers). The subscribers are subscribed to the publisher and they can be unsubscribed. In this architecture new types of subscribers can be easily added(instant messaging, ...) and new types of publishers(Weather News, Sport News, ...).

Specific Implementation Problems

Many subjects to Many observers

It's not a common situation but there are cases when a there are many observers that need to observe more than one subject. In this case the observer need to be notified not only about the change, but also which is the subject with the state changed. This can be realized very simple by adding to the subjects reference in the update notification method. The subject will pass a reference to itself(this) to the when notify the observer.

Who triggers the update?

The communication between the subject and its observers is done through the notify method declared in observer interface. But who it cat be triggered from either subject or observer object. Usually the notify method is triggered by the subject when it's state is changed. But sometimes when the updates are frequent the consecutive changes in the subject will determine many unnecessary refresh operations in the observer. In order to make this process more efficient the observer can be made responsible for starting the notify operation when it consider necessary.

Making sure Subject state is self-consistent before notification

The subject state should be consistent when the notify operation is triggered. If changes are made in the subject state after the observer is notified, it will will be refreshed with an old state. This seems hard to achieve but in practice this can be easily done when Subject subclass operations call inherited operations. In the following example, the observer is notified when the subject is in an inconsistent state:

class Observable{

...

int state = 0;

int additionalState = 0;

public updateState(int increment)

{

state = state + increment;

notifyObservers();

}

...

}

class ConcreteObservable extends Observable{

...

public updateState(int increment){

super.updateState(increment); // the observers are notified

additionalState = additionalState + increment; // the state is changed after the notifiers are updated

}

...

}

This pitfall can be avoided using template methods in the abstract subject superclass for calling the notify operations. Then subject subclass will implement the operations(s) of the template:

class Observable{

...

int state = 0;

int additionalState = 0;

public void final updateState(int increment)

{

doUpdateState(increment);

notifyObservers();

}

public void doUpdateState(int increment)

{

state = state + increment;

}

...

}

class ConcreteObservable extends Observable{

...

public doUpdateState(int increment){

super.doUpdateState(increment); // the observers are notified

additionalState = additionalState + increment; // the state is changed after the notifiers are updated

}

...

}

The Operations defined in the subject base class which triggers notify operation should be documented.

**Design Principles**

This?

<http://www.oodesign.com/design-principles.html>

**Messages**

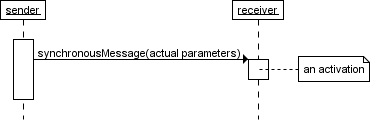
**Very useful link**

[**http://www.tracemodeler.com/articles/a\_quick\_introduction\_to\_uml\_sequence\_diagrams/**](http://www.tracemodeler.com/articles/a_quick_introduction_to_uml_sequence_diagrams/)

When a target sends a message to another target, it is shown as an arrow between their lifelines. The arrow originates at the sender and ends at the receiver. Near the arrow, the name and parameters of the message are shown.

Synchronous message

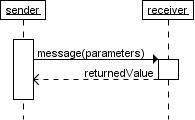
A synchronous message is used when the sender waits until the receiver has finished processing the message, only then does the caller continue (i.e. a blocking call). Most method calls in object-oriented programming languages are synchronous. A closed and filled arrowhead signifies that the message is sent synchronously.



The white rectangles on a lifeline are called activations and indicate that an object is responding to a message. It starts when the message is received and ends when the object is done handling the message.

When a messages are used to represent method calls, each activation corresponds to the period during which an activation record for its call is present on the call stack.

If you want to show that the receiver has finished processing the message and returns control to the sender, draw a dashed arrow from receiver to sender. Optionally, a value that the receiver returns to the sender can be placed near the return arrow.



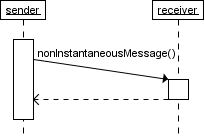
If you want your diagrams to be easy to read, only show the return arrow if a value is returned. Otherwise, hide it.

Instantaneous message

Messages are often considered to be instantaneous, i.e. the time it takes to arrive at the receiver is negligible. For example, an in-process method call. Such messages are drawn as a horizontal arrow.



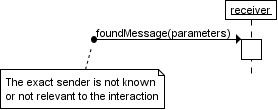
Sometimes however, it takes a considerable amount of time to reach the receiver (relatively speaking of course) . For example, a message across a network. Such a non-instantaneous message is drawn as a slanted arrow.



You should only use a slanted arrow if you really want to emphasize that a message travels over a relatively slow communication channel (and perhaps want to make a statement about the possible delay). Otherwise, stick with a horizontal arrow.

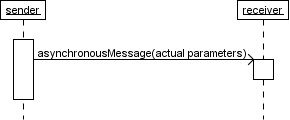
Found message

A found message is a message of which the caller is not shown. Depending on the context, this could mean that either the sender is not known, or that it is not important who the sender was. The arrow of a found message originates from a filled circle.



Asynchronous messages

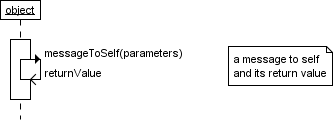
With an asynchronous message, the sender does not wait for the receiver to finish processing the message, it continues immediately. Messages sent to a receiver in another process or calls that start a new thread are examples of asynchronous messages. An open arrowhead is used to indicate that a message is sent asynchrously.



A small note on the use of asynchronous messages : once the message is received, both sender and receiver are working simultaneously. However, showing two simultaneous flows of control on one diagram is difficult. Usually authors only show one of them, or show one after the other.

Message to self

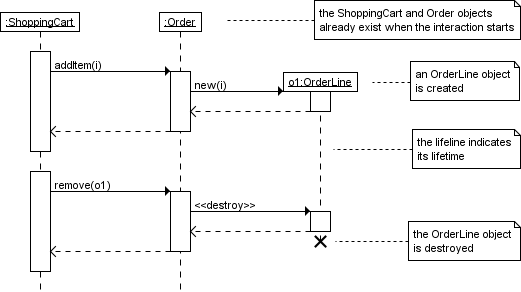
A message that an object sends itself can be shown as follows :



Keep in mind that the purpose of a sequence diagram is to show the interaction between objects, so think twice about every self message you put on a diagram.

Creation and destruction

Targets that exist at the start of an interaction are placed at the top of the diagram. Any targets that are created during the interaction are placed further down the diagram, at their time of creation.



A target's lifeline extends as long as the target exists. If the target is destroyed during the interaction, the lifeline ends at that point in time with a big cross.

**Singleton**

SingletonApp

**package** patterns.singleton;

**public** **class** SingletonApp {

**public** **static** **void** main(String[] args) {

Singleton a = Singleton.*getInstance*();

Singleton b = Singleton.*getInstance*();

System.*out*.println("Identical object (i.e., same reference): " + (a == b));

}

}

Singleton

**package** patterns.singleton;

**public** **class** Singleton {

**private** **static** Singleton *instance*;

**private** Singleton() {

}

**public** **static** Singleton getInstance() {

**if** (*instance* == **null**) {

*instance* = **new** Singleton();

}

**return** *instance*;

}

}

**Another example of Singleton**

**SystemApp**

**package** lab4.alarmSystem;

**public** **class** SystemApp {

**private** **static** **void** delayInSec(**int** delayInSec) **throws** InterruptedException {

Thread.*sleep*(1000L \* delayInSec);

}

**public** **static** **void** main(String[] args) **throws** InterruptedException {

**int** pinCode = 1234;

//этот вызов создает объект AlarmSystem

AlarmSystem systemFirst = AlarmSystem.*Create*(pinCode);

//а этот уже нет, новый инстанс не создается - он просто возвращает ссылку на ранее созданный

//не смотря на то, что мы пытались создать system с другим пинкодом и везде ссылаемся на вторую систему с пинкодом 2468

//фактически мы всегда ссылаемся на систему созданную в переменной systemFirst с пинкодом 1234

AlarmSystem system = AlarmSystem.*Create*(pinCode\*2);

//до того, как мы перешли на использование сингелтона

//эта строчка создавала один объект (instance)

//AlarmSystem system = new AlarmSystem(pinCode);

//сейчас нету возможности запретить написать код, который ниже, он позволяет создать еще одну систему

//в задании просят решить эту проблему

//эта строчка создавала второй объект (instance)

//AlarmSystem system2 = new AlarmSystem(pinCode);

Sensor bathSensor = **new** Sensor("Bathroom", SensorType.*WINDOW*, SensorState.*CLOSED*);

Sensor floorSensor = **new** Sensor("Floor", SensorType.*DOOR*, SensorState.*CLOSED*);

system.addSensor(bathSensor);

system.addSensor(floorSensor);

system.activate();

*delayInSec*(20);

floorSensor.changeState();

*delayInSec*(20);

system.deactivate(pinCode + 1);

*delayInSec*(5);

system.deactivate(pinCode);

}

}

**SystemState**

**package** lab4.alarmSystem;

**public** **enum** SystemState {

*INACTIVE*,

*LEAVING\_HOME*,

*ACTIVE*,

*COMING\_HOME*,

*ALARM* {

@Override

**protected** **void** entry(AlarmSystem parent) {

**super**.entry(parent);

parent.setSirenSoundOn(**true**);

}

@Override

**protected** **void** exit(AlarmSystem parent) {

**super**.exit(parent);

parent.setSirenSoundOn(**false**);

}

};

**protected** SystemState nextState(AlarmSystem parent, SystemState nextState) {

exit(parent);

nextState.entry(parent);

**return** nextState;

};

**protected** **void** entry(AlarmSystem parent) {

System.*out*.println("<System> State: " + **this**);

}

**protected** **void** exit(AlarmSystem parent) {

}

}

**Siren**

**package** lab4.alarmSystem;

**public** **class** Siren {

**private** **boolean** isSoundOn;

**public** **boolean** isSoundOn() {

**return** isSoundOn;

}

**public** **void** setSoundOn(**boolean** isSoundOn) {

**this**.isSoundOn = isSoundOn;

}

}

**Sensor**

**package** lab4.alarmSystem;

**public** **class** Sensor {

**private** **final** String identifier;

**private** SensorType type;

**private** SensorState state;

**private** AlarmSystem system;

**public** Sensor(String identifier, SensorType type, SensorState state) {

**this**.identifier = identifier;

**this**.type = type;

**this**.state = state;

}

**public** String getIdentifier() {

**return** identifier;

}

**public** SensorType getType() {

**return** type;

}

**public** SensorState getState() {

**return** state;

}

**public** **void** setAlarmSystem(AlarmSystem system) {

**this**.system = system;

}

**public** **void** changeState() {

state = state.nextState();

**if** (system != **null**) {

system.sensorChanged(**this**);

}

}

}

**SensorType**

**package** lab4.alarmSystem;

**public** **enum** SensorType {

*DOOR*,

*WINDOW*;

}

**SensorState**

**package** lab4.alarmSystem;

**public** **enum** SensorState {

*CLOSED* {

@Override

**protected** SensorState nextState() {

**return** *OPEN*;

}

},

*OPEN* {

@Override

**protected** SensorState nextState() {

**return** *CLOSED*;

}

};

**protected** **abstract** SensorState nextState();

}

**AlarmSystem**

**package** lab4.alarmSystem;

**import** java.util.ArrayList;

**public** **class** AlarmSystem {

**private** **final** **int** pinCode;

**private** SystemState state = SystemState.*INACTIVE*;

**private** Siren siren = **new** Siren();

**private** ArrayList<Sensor> sensors = **new** ArrayList<Sensor>();

//это статическая переменная, которая будет хранить нашу систему в единственном экземпляре

**private** **static** AlarmSystem *singleton* = **null**;

//для этого используем паттерн программирования Singleton

**public** **static** AlarmSystem Create(**int** pinCode)

{

//когда мы хотим создать систему в первый раз - переменная singleton равна null и по-этому сработает код singleton = new AlarmSystem(pinCode);

**if** (*singleton* == **null**)

{

*singleton* = **new** AlarmSystem(pinCode);

**return** *singleton*;

}

//после того, как мы попробуем еще раз создать новую систему - функция поймет, что уже есть старая и просто вернет на нее ссылку, не создавая новый объект

**else**

**return** *singleton*;

}

//здесь конструктор сделали приватным для того, чтобы у программиста даже не было возможности самостоятельно создавать экземпляры этого класса

//специально для того, что бы можно было создать только один экземпляр мы выше создали метод Create, который контролирует, чтобы экземпляр был только один

//здесь вся хитрость в том, что приватные методы, в том числе и конструкторы, пользователю класса доступны только если их вызывает какой-то публичный метод

//если не сделать публичного метода, который будет вызывать такой метод, то никакого способа его вызвать не будет - так захотел проэктировщик класса

//чтобы создать объект, конструктор которого приватный, можно использовать только ПУБЛИЧНЫЙ СТАТИЧЕСКИЙ метод

//именно таким и есть public static AlarmSystem Create(int pinCode)

**private** AlarmSystem(**int** pinCode) {

**this**.pinCode = pinCode;

System.*out*.println("<System> Created with pin code " + pinCode);

}

**public** **void** addSensor(Sensor sensor) {

**if** (!sensors.contains(sensor)) {

sensors.add(sensor);

sensor.setAlarmSystem(**this**);

System.*out*.printf("<System> Added sensor: %s (%s)\n", sensor.getIdentifier(), sensor.getType());

}

}

**public** **void** activate() {

**if** (state == SystemState.*INACTIVE*) {

state = state.nextState(**this**, SystemState.*LEAVING\_HOME*);

nextStateDelayed();

}

}

**public** **void** deactivate(**int** pinCode) {

**if** (pinCode == **this**.pinCode) {

state = state.nextState(**this**, SystemState.*INACTIVE*);

} **else** {

System.*out*.println("<System> Incorrect pin code for deactivation: " + pinCode);

}

}

**public** **void** sensorChanged(Sensor sensor) {

System.*out*.printf("<Sensor> %s: %s\n", sensor.getIdentifier(), sensor.getState());

**if** ((state == SystemState.*ACTIVE*) && (sensor.getState() == SensorState.*OPEN*)) {

**if** (sensor.getType() == SensorType.*DOOR*) {

state = state.nextState(**this**, SystemState.*COMING\_HOME*);

nextStateDelayed();

} **else** {

state = state.nextState(**this**, SystemState.*ALARM*);

}

}

}

**public** **void** setSirenSoundOn(**boolean** isSoundOn) {

siren.setSoundOn(isSoundOn);

System.*out*.println("<Siren> Sound on: " + siren.isSoundOn());

}

**private** **void** nextStateDelayed() {

DelayThread delay = **new** DelayThread(**this**, 10);

delay.start();

}

**protected** **void** delayExpired() {

**if** (state == SystemState.*LEAVING\_HOME*) {

state = state.nextState(**this**, SystemState.*ACTIVE*);

} **else** **if** (state == SystemState.*COMING\_HOME*) {

state = state.nextState(**this**, SystemState.*ALARM*);

}

}

}

**class** DelayThread **extends** Thread {

**private** AlarmSystem parent;

**private** **int** delayInSec;

**public** DelayThread(AlarmSystem parent, **int** delayInSec) {

**super**();

**this**.parent = parent;

**this**.delayInSec = delayInSec;}

@Override

**public** **void** run() {

**try** {

*sleep*(1000L \* delayInSec);

} **catch** (InterruptedException e) {

}

parent.delayExpired();

}}

**Observer**

**AlarmApp**

**package** patterns.observer.v1\_simple;

**public** **class** AlarmApp {

**public** **static** **void** main(String[] args) {

AlarmSystem system = **new** AlarmSystem();

ContactSensor bathroom = **new** ContactSensor("Bathroom", **false**);

ContactSensor floor = **new** ContactSensor("Floor", **false**);

system.addSensor(bathroom);

system.addSensor(floor);

bathroom.changeState();

system.setActive(**true**);

bathroom.changeState();

floor.changeState(); }

}

**AlarmSystem**

**package** patterns.observer.v1\_simple;

**import** java.util.ArrayList;

**public** **class** AlarmSystem {

**private** **boolean** isActive;

**private** ArrayList<ContactSensor> sensors = **new** ArrayList<ContactSensor>();

**public** **void** setActive(**boolean** isActive) {

**this**.isActive = isActive;

System.*out*.println("<System> Active: " + isActive); }

**public** **void** addSensor(ContactSensor sensor) {

sensors.add(sensor);

sensor.setSystem(**this**); }

**public** **void** onSensorStateChanged(ContactSensor sensor) {

System.*out*.println(sensor);

**if** (isActive && sensor.isOpen()) {

System.*out*.println("<System> Alarm!!!");}

}

}

**ContactSensor**

**package** patterns.observer.v1\_simple;

**public** **class** ContactSensor {

**private** String location;

**private** **boolean** isOpen;

**private** AlarmSystem system;

**public** ContactSensor(String location, **boolean** isOpen) {

**super**();

**this**.location = location;

**this**.isOpen = isOpen;

}

**public** **boolean** isOpen() {

**return** isOpen;

}

**public** **void** changeState() {

isOpen = !isOpen;

**if** (system != **null**) {

system.onSensorStateChanged(**this**); }

}

**public** **void** setSystem(AlarmSystem system) {

**this**.system = system;

}

@Override

**public** String toString() {

**return** String.*format*("<Contact> %s open: %b", location, isOpen);}

}

**Iterator**

**IteratorApp**

**package** patterns.iterator.v1\_simple;

**import** java.util.ArrayList;

/\*\* Executable application for demonstration purposes.

\* Creates a sample warehouse and prints information on containers

\* stored to the console.

**public** **class** IteratorApp {

**private** **static** Warehouse createSampleWarehouse() {

//коллекция - это какой-то набор объектов одного типа

//они умеют как минимум добавлять новые объекты, но дальше все определяется тем, что еще нужно делать над такими объектами

//конкретно эта коллекция имеет методы addContainer и removeContainer для добавления/удаления контейнеров

//и метод getStock, который вернет все объекты как список

Warehouse warehouse = **new** Warehouse();

warehouse.addContainer(**new** Container(543234, "Hamburg"));

warehouse.addContainer(**new** Container(947335, "Hamburg"));

warehouse.addContainer(**new** Container(154386, "Amsterdam"));

warehouse.addContainer(**new** Container(835445, "Hamburg"));

warehouse.addContainer(**new** Container(486523, "Amsterdam"));

**return** warehouse;

}

**public** **static** **void** main(String[] args) {

Warehouse warehouse = *createSampleWarehouse*();

System.*out*.println("Warehouse stock:");

ArrayList<Container> warehouseStock = warehouse.getStock();

//этот цикл и делает итерирование (перебор элементов в коллекции) для вывода всех объектов колекции на экран

**for** (Container container : warehouseStock) {

System.*out*.println(container);

}

}

}

**WareHouse**

**package** patterns.iterator.v1\_simple;

**import** java.util.ArrayList;

/\*\* Represents a warehouse as storage for containers.

\* Containers can be added and removed in arbitrary order.

**public** **class** Warehouse {

**private** ArrayList<Container> containers = **new** ArrayList<Container>();

**public** **void** addContainer(Container container) {

containers.add(container);

}

**public** **void** removeContainer(Container container) {

containers.remove(container);

}

//здесь они схитрили

//так как они используют класс ArrayList умеющий возвращать итератор, они тебе и возвращают такой класс

//таки образом уменьшив себе работу

//но далеко не всегда все так просто, думаю в более сложных примерах ты увидишь как внутри устроены итераторы ;)

**public** ArrayList<Container> getStock() {

**return** containers;

}

}

**Container**

**package** patterns.iterator.v1\_simple;

/\*\* Represents a container as storage for goods.

\* A container has an identifier and a shipping destination.

**public** **class** Container {

**private** **int** id;

**private** String destination;

**public** Container(**int** id, String destination) {

**super**();

**this**.id = id;

**this**.destination = destination;

}

**public** **int** getId() {

**return** id;

}

**public** String getDestination() {

**return** destination;

}

@Override

**public** String toString() {

**return** String.*format*("Container #%d to %s", id, destination);

}

}