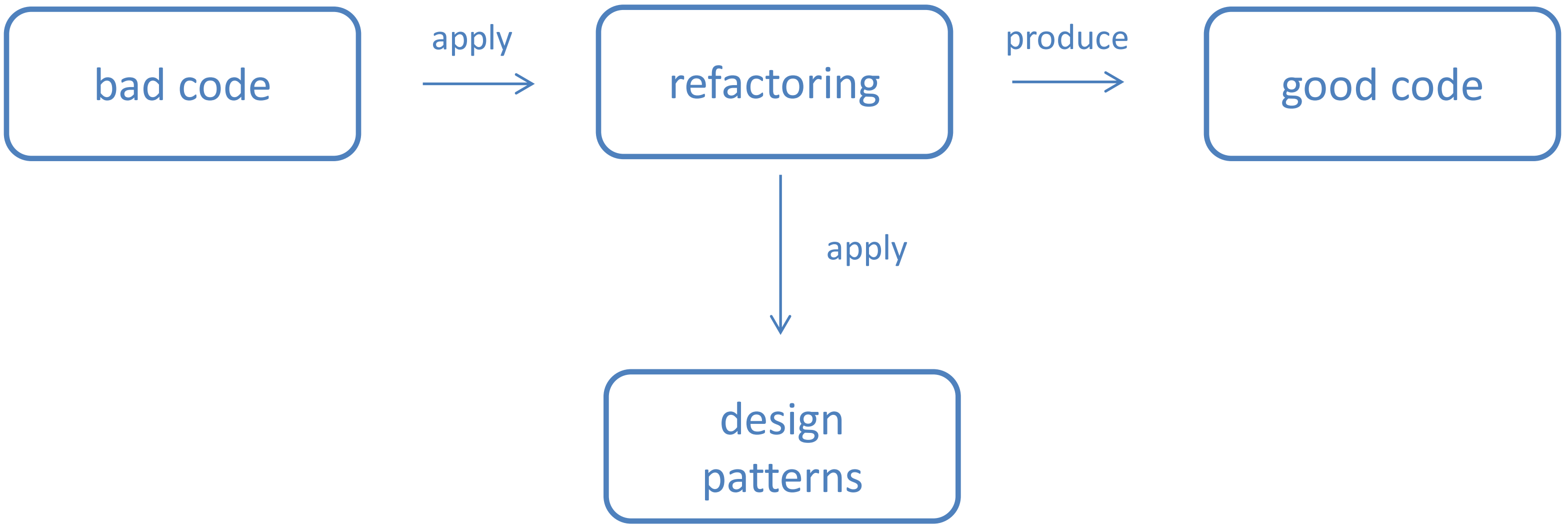
Design patterns

Agenda

- Design patterns
- Catalogue of design patterns
- Applying design pattern

The Big Picture



bad code

apply

refactoring

produce

good code

apply

design patterns

Design patterns

Design patterns

- Design patterns are well-known solutions to common problems in software development

- According to the notorious design patterns book by the Gang of Four and other sources they are divided into three logical groups:

  - creational - related to the creation of objects
  - structural - related to the structural relationship between classes
  - behavioral - related to the communication between objects

Design patterns

- In order to understand design patterns in detail it is good to look into the corresponding class/sequence diagram that represents them

- Code examples for the demonstrations of design patterns in GitHub: https://github.com/martinfmi/Java-design-patterns

Design patterns catalogue

Creational patterns

- Abstract factory
- Builder
- Dependency injection
- Factory method
- Lazy initialization
- Object pool
- Prototype
- Singleton

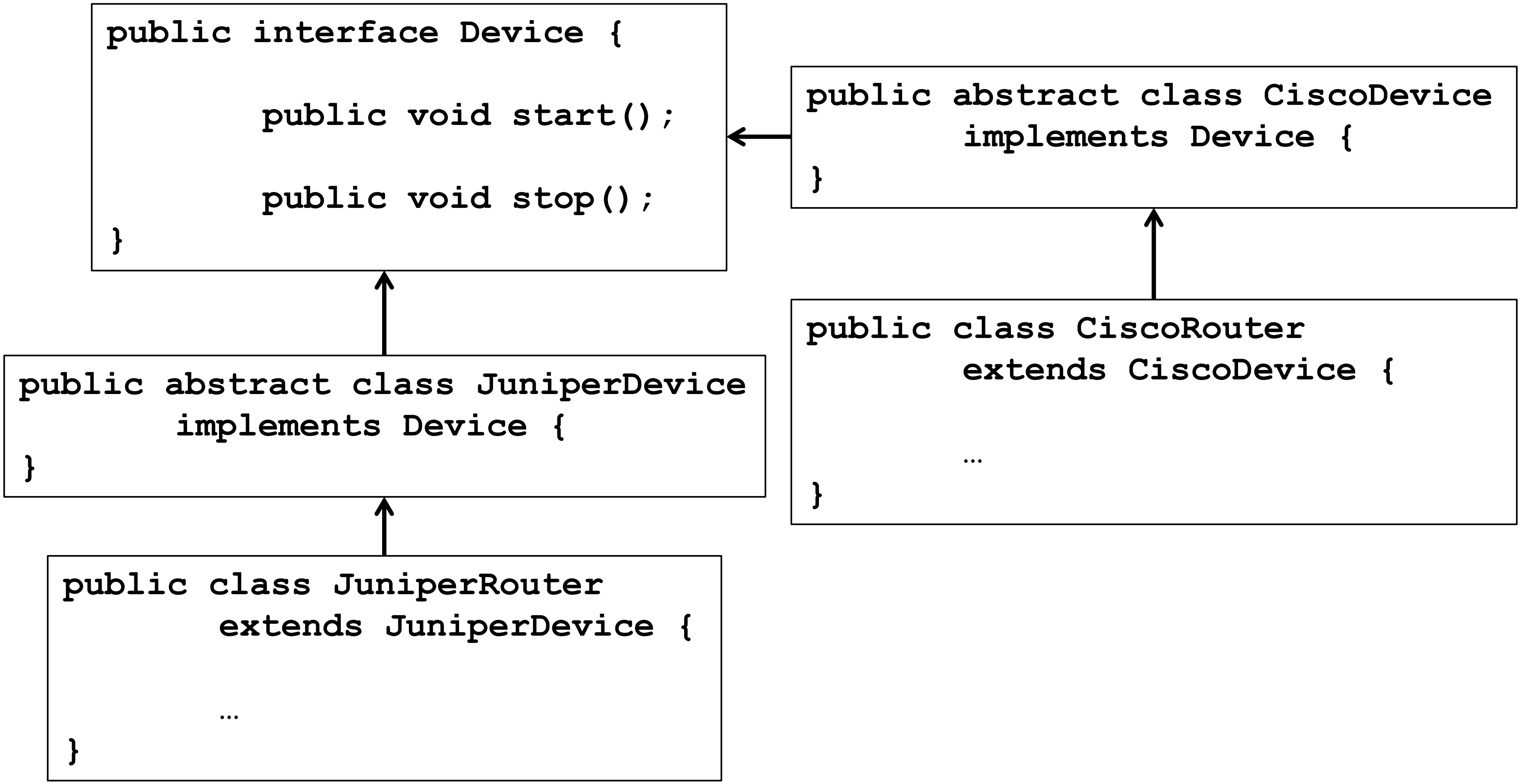
Abstract factory

- A factory is a an OOP concept that denotes an object used to create another object

- Abstract factory provides an interface for creating a variety of dependent object without using their concrete classes

- Good way to design functionality that:
  - hides of how objects are created
  - provides independence between client classes and created objects
  - provides a common mechanism to create instances of the group of dependent objects

Abstract factory example



```
public interface Device {

        public void start();

        public void stop();
}
```

```
public abstract class CiscoDevice
        implements Device {
}
```

```
public abstract class JuniperDevice
        implements Device {
}
```

```
public class CiscoRouter
        extends CiscoDevice {

        …
}
```

```
public class JuniperRouter
        extends JuniperDevice {

        …
}
```

Abstract factory example

```
public interface DeviceFactory {

        public Device device(String serialNumber);

        public static DeviceFactory factory(String vendor) {

                DeviceFactory factory = null;
                switch (vendor) {
                case "cisco":
                        factory = new CiscoDeviceFactory();
                        break;
                case "juniper":
                        factory = new JuniperDeviceFactory();
                        break;
                default:
                        throw new RuntimeException(…)
                }
                return factory;
        }
}
```

Abstract factory example

```
public class CiscoDeviceFactory implements DeviceFactory {

        @Override
        public Device device(String serialNumber) {
                Device device = null;
                if(serialNumber != null &&
                        serialNumber.contains("router")) {
                        device = new CiscoRouter();
                } else {
                        throw new RuntimeException(…)
                }
                return device;
        }

}
```

Abstract factory example

```
public class JuniperDeviceFactory implements DeviceFactory {

        @Override
        public Device device(String serialNumber) {
                Device device = null;
                if(serialNumber != null &&
                        serialNumber.contains("router")) {
                        device = new JuniperRouter();
                } else {
                        throw new RuntimeException(…)
                }
                return device;
        }

}
```

Abstract factory example

```
Device device = DeviceFactory.
                      factory("cisco").
                      device("router SN123");
```

Abstract factory in practice

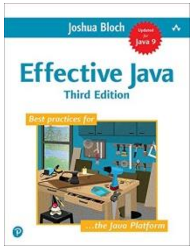
- JDK:
  - javax.xml.parsers.DocumentBuilderFactory
  - javax.xml.xpath.XPathFactory

  - (Note the above are slightly different as newInstance(…) methods are static but return different factory implementation)

- Spring framework:
  - org.springframework.beans.factory.support.BeanDefinitionBuilder

Builder

- Builder pattern is used to separate creation of complex objects from their representation
- In practice many frameworks and libraries implement builder pattern with an internal Builder class
- It is perfectly legal and accepted if the builder class is a top-level public class in its own file



Consider a builder when having too many constructor parameters

Builder example

```
public class Device {

       private String serialNumber;

       private String shortName;

       private double price;

       public Device(String serialNumber, String shortName,
double price) {
              …
       }

       public static class Builder {
              …
       }
}
```

Builder example

```
public static class Builder {

        private String serialNumber;

        private String shortName;

        private double price;

        public Builder serialNumber(String serialNumber) {
                      this.serialNumber = serialNumber;
                      return this;
        }

        …

        public Device build() {
                return new Device(serialNumber, shortName, price);
        }
}
```

Builder example

```
public static class Builder {

       private String serialNumber;

       private String shortName;

       private double price;

       public Builder serialNumber(String serialNumber) {
                      this.serialNumber = serialNumber;
                      return this;
       }

       …

       public Device build() {
               return new Device(serialNumber, shortName, price);
       }
}
```

Builder example

```
Device.Builder().
          serialNumber("SN123").
          shortName("router").
          price(1000d).
          build();
```

Builder in practice

- JDK:
  - java.util.stream.Stream.Builder
  - java.util.Calendar.Builder
  - java.util.Locale.Builder

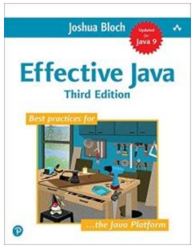
Dependency injection

- Dependency injection is a way to apply inversion of control
- It happens when one object supplies the dependencies of another object
- Typically achieved with the support of a DI framework such as Spring or CDI

Dependency injection

- It provides a fundamental mechanism to loose coupling between objects

- Also provides a way for applications to support different configurations



Prefer dependency injection to hardwiring resources

Dependency injection example

```
public class DeviceController {

        private Device device;

        public void setDevice(Device device) {
                this.device = device;
        }
}
```

```
public class DeviceInjector {

        public void inject(DeviceController controller,
                      Device device) {
                controller.setDevice(device);
        }

}
```

Dependency injection example

```
DeviceInjector injector = new DeviceInjector();
Device router = new CiscoRouter();
DeviceController controller = new DeviceController();
injector.inject(controller, router);
```

Dependency injection in practice

- Frameworks implementing the pattern:
  - Spring DI
  - JavaEE CDI
  - Google Guice
  - OSGi declarative services DI

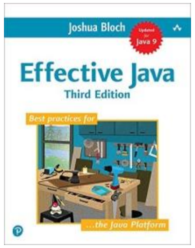
Factory method

- Allows a class to create objects without knowing concrete implementation

- Achieved by means of calling proper factory method on a child object that is responsible to create concrete implementation instance

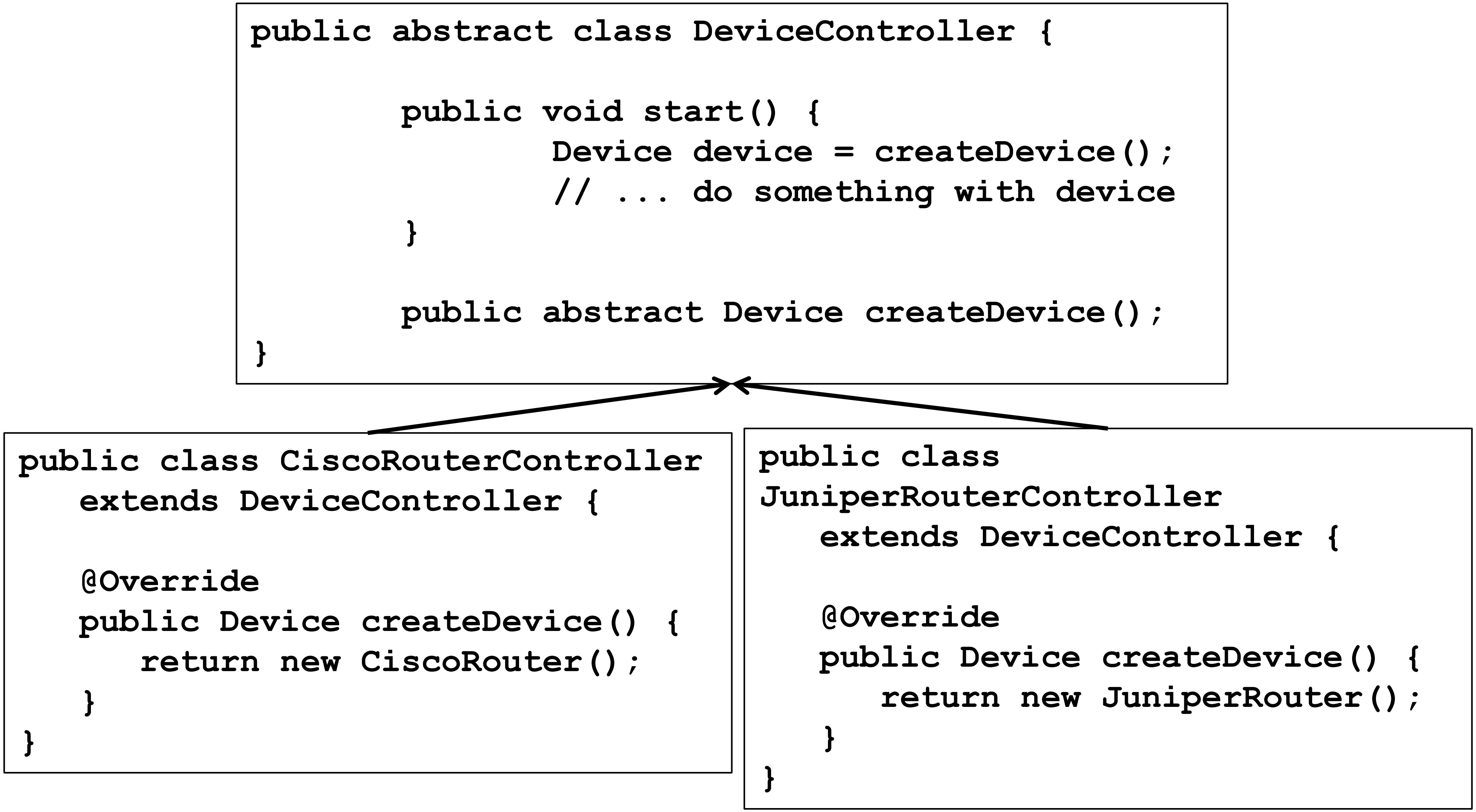
Factory method

- Not to be confused with static factory methods that provide simpler mechanism to create objects instead of using a constructor with parameters



Consider static factory methods instead of constructors

Factory method example



```
public abstract class DeviceController {

        public void start() {
                Device device = createDevice();
                // ... do something with device
        }

        public abstract Device createDevice();
}
```

```
public class CiscoRouterController
   extends DeviceController {

   @Override
   public Device createDevice() {
      return new CiscoRouter();
   }
}
```

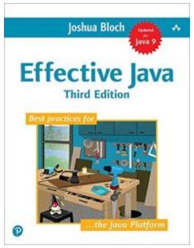
```
public class
JuniperRouterController
   extends DeviceController {

   @Override
   public Device createDevice() {
      return new JuniperRouter();
   }
}
```

Factory method in practice

- JDK:
  - javax.naming.spi.ObjectFactory (getObjectInstance() methods)
- Spring framework:
  - org.springframework.beans.factory.BeanFactory (getBean() methods)

Lazy initialization

- Is a pattern used to delegate the creation of an object for a later time
- Creation of the object happens typically when it is needed
- In many cases lazy initialization is combined with the factory method pattern



Use lazy initialization judiciously

Lazy initialization example

```
public class CiscoRouterController extends DeviceController {

        private Device device;

        @Override
        public Device createDevice() {
                if(device == null) {
                        device = new CiscoRouter();
                }
                return device;
        }

}
```

Lazy initialization example

```
public class CiscoRouterSynchronizedController
        extends DeviceController {

        private volatile Device device;

        @Override
        public Device createDevice() {
                if(device == null) {
                        synchronized (this) {
                                if(device == null) {
                                        device = new CiscoRouter();
                                }
                        }
                }
                return device;
        }
}
```

Lazy initialization in practice

- Spring framework:
  - we can define spring beans as “lazy” and they will be created when needed

Object pool

- An object pool alleviates the need to create expensive objects

- Notorious applications of the pattern are connection and thread pools

Object pool example

```
public class CiscoDevicePool {

        private Map<String, Device> devicePool =
                new HashMap<String, Device>();

        public Device getDevice(String serialNumber) {
                Device device = devicePool.get(serialNumber);
                if(device == null) {
                        device = new CiscoRouter();
                        // set proper device settings ...
                        devicePool.put(serialNumber, device);
                }
                return device;
        }
}
```

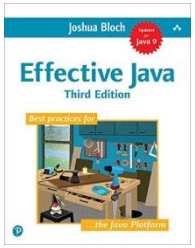
Object pool in practice

- JDBC connection pool

- JDK executor thread pools

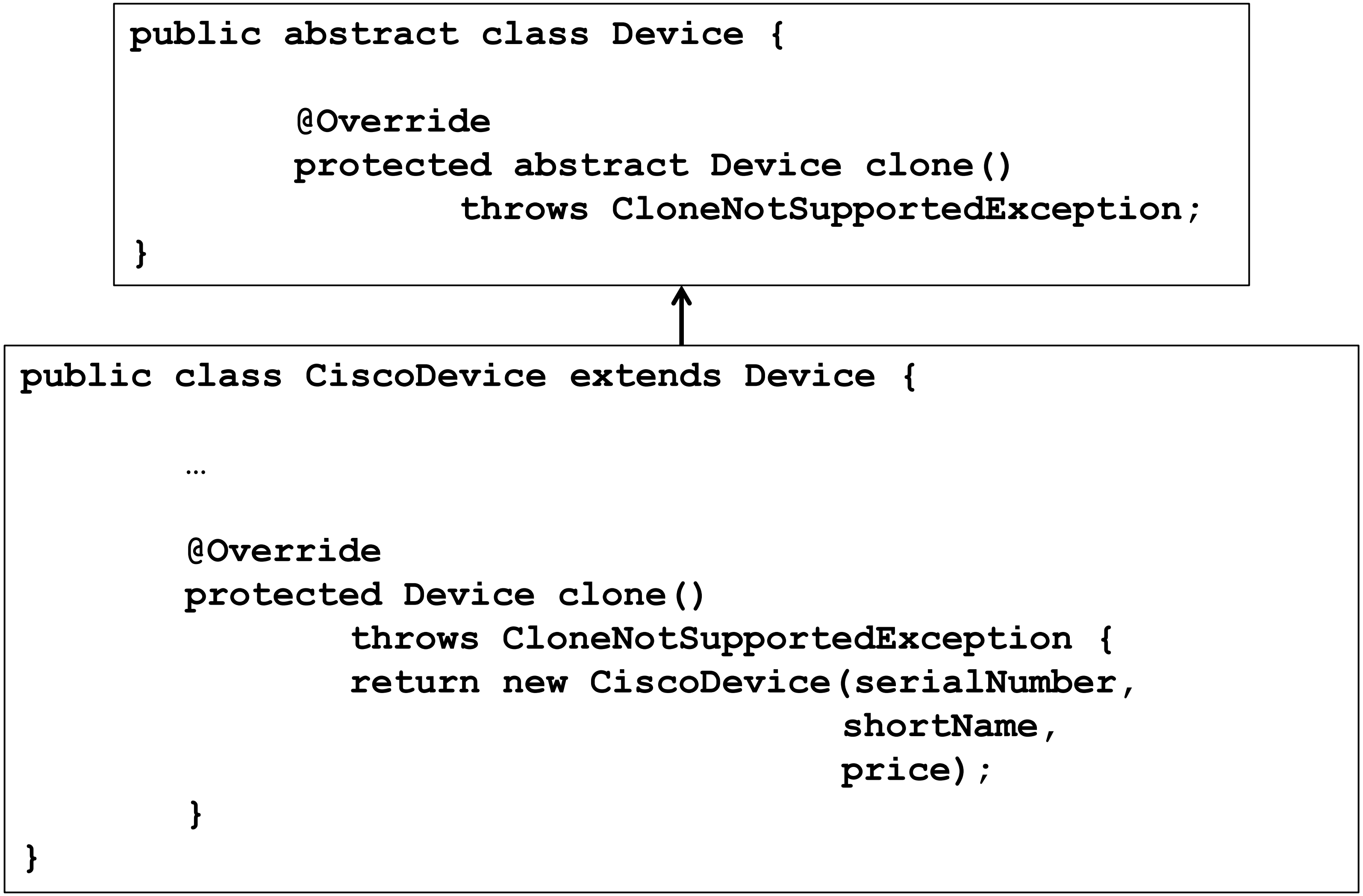
Prototype

- Provides a mechanism to create objects from a template object
- Used typically to avoid creation of expensive objects using 'new‘
- Provides the ability to copy objects without knowing the concrete subtype



Override clone judiciously

Prototype example



```
public abstract class Device {

        @Override
        protected abstract Device clone()
                throws CloneNotSupportedException;
}
```

```
public class CiscoDevice extends Device {

        …

        @Override
        protected Device clone()
                throws CloneNotSupportedException {
                return new CiscoDevice(serialNumber,
                                      shortName,
                                      price);
        }
}
```

Prototype in practice

- JDK:
  - java.lang.Object (through clone() method, classes must implement java.lang.Cloneable)

Singleton

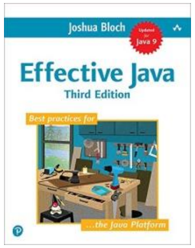
- Provides a mechanism to ensure a class has only one instance

- Used to avoid excessive creation of class instances whenever possible

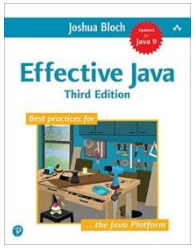
Singleton

- A generalization of the singleton pattern is called 'multiton'

- A multiton provides creation of multiple instances



Enforce the singleton property with a private constructor or an enum type



Avoid creating unnecessary objects

Singleton example

```
public class CiscoRouterController{

        private static CiscoRouterController controller =
                new CiscoRouterController();

        private CiscoRouterController() {}

        public static CiscoRouterController instance() {
                return controller;
        }

}
```

Singleton example

```
public class LazyCiscoRouterController  {

        private static LazyCiscoRouterController controller;

        private LazyCiscoRouterController() {}

        public static LazyCiscoRouterController instance() {
                if(controller == null) {
                        controller = new LazyCiscoRouterController();
                }
                return controller;
        }
}
```

Singleton example

```
public class LazyCiscoRouterSynchronizedController {

        private static volatile
                LazyCiscoRouterSynchronizedController controller;

        private LazyCiscoRouterSynchronizedController() {}

        public static LazyCiscoRouterSynchronizedController
                instance() {
                if(controller == null) {
                        synchronized
                        (LazyCiscoRouterSynchronizedController.class) {
                                if(controller == null) {
                                        controller = new
                                LazyCiscoRouterSynchronizedController();
                                }
                        }
                }
                return controller;
        }
}
```

Singleton in practice

- JDK:
  - java.lang.Runtime

- Spring framework:
  - Beans defined as singleton by default in the Spring configuration

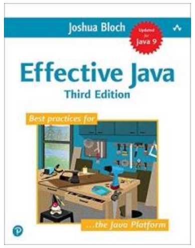
Structural patterns

- Adapter
- Bridge
- Decorator
- Composite
- Façade
- Flyweight
- Front controller
- Marker
- Proxy

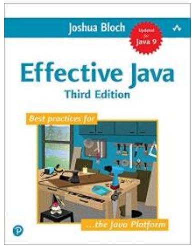
Adapter

- Provides a mechanism to 'adapt' one incompatible type to another

- Can be used to provide an alternative to multiple inheritance in Java



Avoid creating unnecessary objects
(An adapter does not need to be created more than once for a given object)



Favor static member classes over non-static

(An adapter class can be created as a non-static inner class)

Adapter example

```
public class JuniperRouterAdapter extends CiscoRouter {

        private JuniperRouter juniperRouter;

        public JuniperRouterAdapter(JuniperRouter juniperRouter) {
                this.juniperRouter = juniperRouter;
        }

        @Override
        public void start() {
                juniperRouter.start();
        }

        @Override
        public void stop() {
                juniperRouter.stop();
        }
}
```

Adapter example

```
JuniperRouter juniperRouter = new JuniperRouter();
CiscoRouter ciscoRouter =
       new JuniperRouterAdapter(juniperRouter);
ciscoRouter.start();
```

Adapter in practice

- JDK:
  - java.io.InputStreamReader(InputStream)
  - java.io.OutputStreamWriter(OutputStream)

Bridge

- Allows to decouple abstraction from implementation

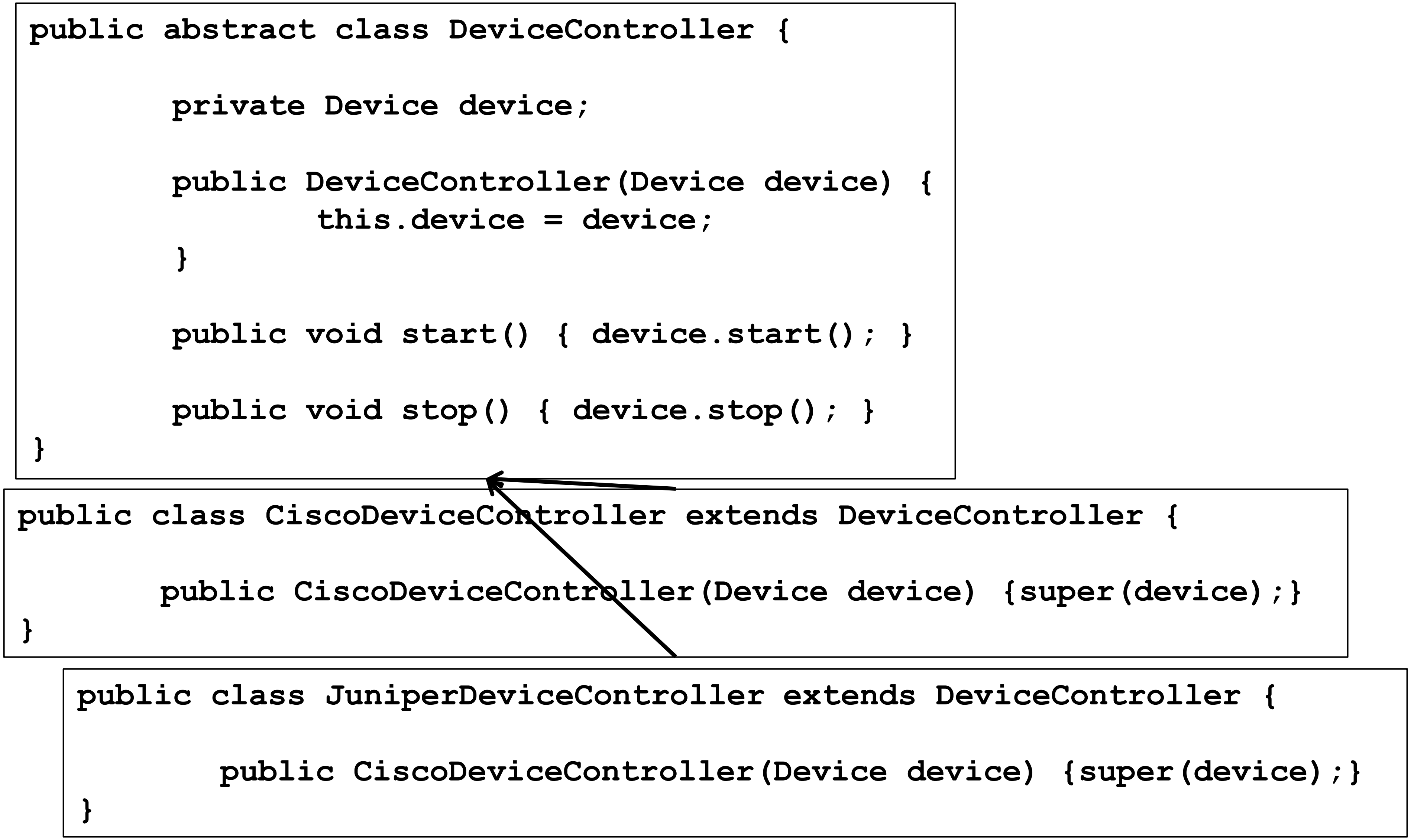
- The abstraction and implementation have their hierarchies

- Reduces the number of boilerplate classes that need to be written

Bridge

- Sometimes confused with adapter pattern
- Adapter pattern in contrast is useful for existing classes
- Bridge pattern is used when two hierarchies are known at design time typically

Bridge example



```
public abstract class DeviceController {

        private Device device;

        public DeviceController(Device device) {
                this.device = device;
        }

        public void start() { device.start(); }

        public void stop() { device.stop(); }
}
```

```
public class CiscoDeviceController extends DeviceController {

        public CiscoDeviceController(Device device) {super(device);}
}
```

```
public class JuniperDeviceController extends DeviceController {

        public CiscoDeviceController(Device device) {super(device);}
}
```

Bridge example

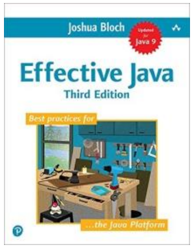
```
CiscoRouter device = new CiscoRouter();
CiscoDeviceController controller =
       new CiscoDeviceController(device);
controller.start(); // no need to have CiscoRouterController
```

Bridge in practice

- JDK:
  - in AWT the hierarchies of java.awt.Component and java.awt.peer.ComponentPeer

Decorator

- An object used to add behavior to another object
- An alternative to subclassing
- Can be applied in cases when subclassing is not possible or applicable
- Decorator pattern uses typically delegation for the existing operations of the decorated (also called wrapped) object



Favor composition over inheritance

Decorator example

```
public abstract class RestartableDevice extends Device {

        private Device device;

        public RestartableDevice(Device device) {
                this.device = device;
        }

        public void restart() {
                device.stop();
                device.start();
        }
}
```

Decorator in practice

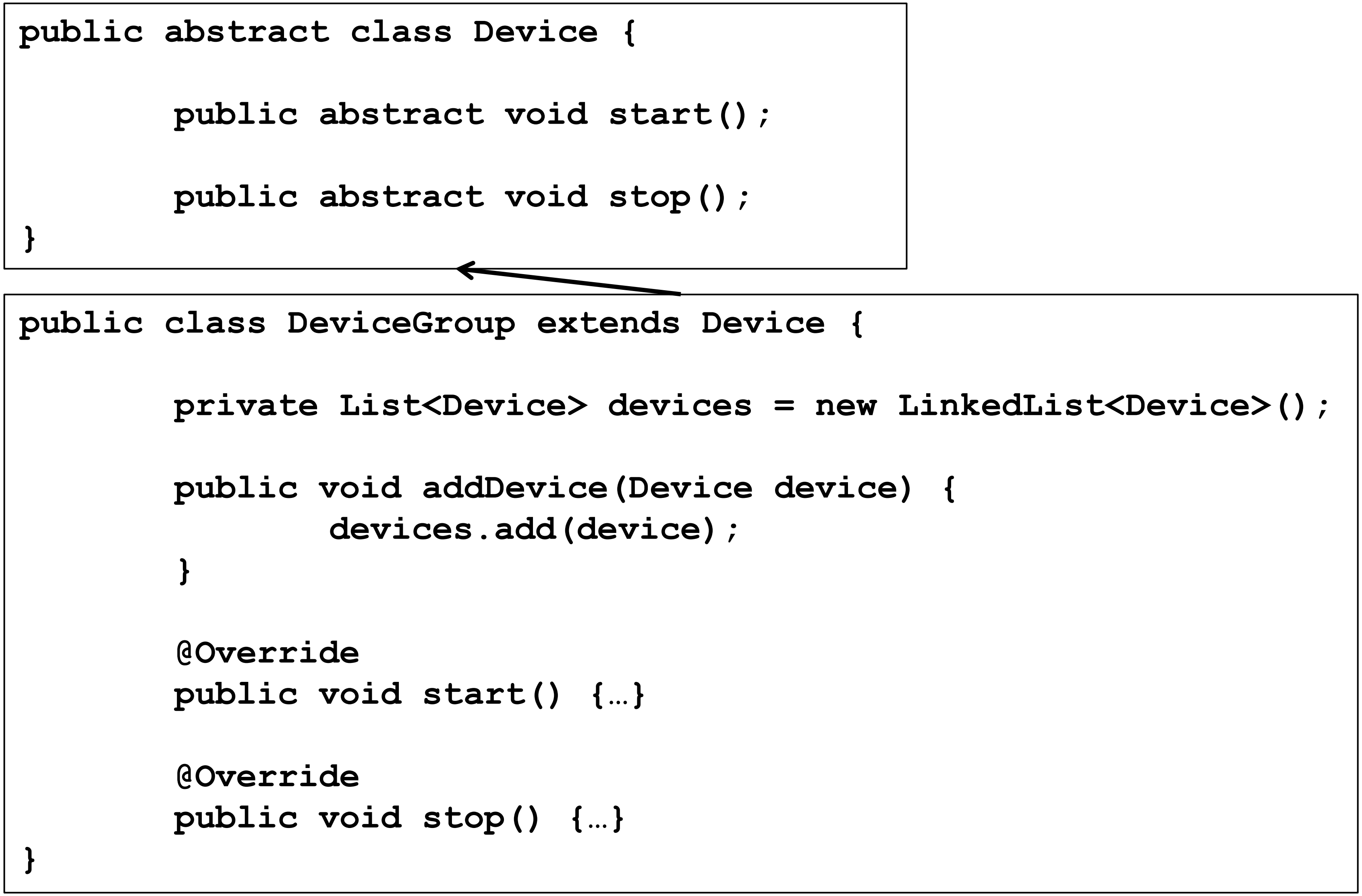
- JDK:
  - java.io.BufferedReader/BufferedWriter

Composite

- Provides the possibility to compose objects in a tree structure

- Treats simple objects and compositions of objects uniformly

Composite example



```
public abstract class Device {

        public abstract void start();

        public abstract void stop();
}
```

```
public class DeviceGroup extends Device {

        private List<Device> devices = new LinkedList<Device>();

        public void addDevice(Device device) {
                devices.add(device);
        }

        @Override
        public void start() {…}

        @Override
        public void stop() {…}
}
```

Composite in practice

- JDK:
  - java.awt.Component

- JavaEE:
  - javax.faces.component.UIComponent

Facade

- Provides a simpler interface for interacting with a complex system

- Serves as an entrypoint to a particular (sub)system

Facade example

```
public class DeviceManager {

        private DeviceGroup devices;

        public void initialize() {
                devices = new DeviceGroup();
                // do some complex device initialization ...
                for(Device device : devices.getDevices()) {
                        device.start();
                }
        }

        public static void main(String[] args) {
                DeviceManager manager = new DeviceManager();
                manager.initialize();
        }
}
```

Facade in practice

- JavaEE:
  - javax.faces.context.FacesContext

Flyweight

- Provides a way to store large number of objects efficiently

- Avoids creating a large number of objects

Flyweight example

```
public class Manufacturer {

        private String name;

        public Manufacturer(String name) {
                this.name = name;
        }

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

}
```

Flyweight example

```
public class Device {

        private static HashMap<String, Manufacturer>
                manufacturersCache = new HashMap<>();

        private String serialNumber;

        private Manufacturer manufacturer;

        public static Device of(String manufacturer,
                        String serialNumber) {
                Device device = new Device();
                device.setSerialNumber(serialNumber);
                Manufacturer manufacturerItem =
                   manufacturersCache.computeIfAbsent(manufacturer,
                        (key) -> new Manufacturer(manufacturer));
                device.setManufacturer(manufacturerItem);
                return device;
        }
        …
}
```

Flyweight in practice

- JDK:
  - java.lang.Integer (though valueOf(int) that caches values in the range of -128 to 127)
  - similar behavior for other classes through valueOf(…) method

Front controller

- A common pattern used by web application framework
- Used to handle every request from a client and dispatch accordingly to a proper handler class

Front controller example

```
public abstract class DeviceManager {

        private HashMap<String, Device> devices =
                new HashMap<String, Device>();

        public abstract Device createDevice(String serialNumber);

        public void addDevice(String serialNumber, Device device) {
                devices.put(serialNumber, device);
        }

}
```

Front controller example

```
public class CiscoDeviceManager extends DeviceManager {

        @Override
        public Device createDevice(String serialNumber) {
                Device device = null;
                if(serialNumber.contains("router")) {
                        device = new CiscoRouter();
                } else {
                        throw new RuntimeException(…);
                }

                return device;
        }
}
```

Front controller example

```
public class JuniperDeviceManager extends DeviceManager {

        @Override
        public Device createDevice(String serialNumber) {
                Device device = null;
                if(serialNumber.contains("router")) {
                        device = new JuniperRouter();
                } else {
                        throw new RuntimeException(…);
                }

                return device;
        }
}
```

Front controller example

```
public class DeviceController { // front controller

        private HashMap<String, DeviceManager> vendorToDeviceManager
                = new HashMap<String, DeviceManager>();

        public DeviceController() {
                vendorToDeviceManager.put("cisco",
                        new CiscoDeviceManager());
                vendorToDeviceManager.put("juniper",
                        new JuniperDeviceManager());
        }

        public void invokeOperation(String vendor,
                String serialNumber, String operation) {
                DeviceManager manager =
                        vendorToDeviceManager.get(vendor);
                if("create".equals(operation)) {
                        manager.createDevice(serialNumber);
                }
        }
}
```

Front controller example

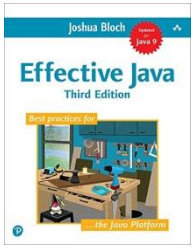
```
DeviceController controller = new DeviceController();
controller.invokeOperation("cisco", "router SN123", "create");
```

Front controller in practice

- Spring framework:
  - org.springframework.web.servlet.DispatcherServlet

Marker interface

- Provides a mechanism to associate certain metadata with a class
- The metadata is typically used at runtime to determine certain properties of the class
- A runtime annotation can also achieve the same purpose as a marker interface



Use marker interfaces to define types

Marker interface example

```
public interface RestartableDevice {
}
```

Marker interface in practice

- JDK:
  - java.lang.Cloneable
  - java.io.Serializable
  - java.rmi.Remote

Proxy

- Provides an interface (wrapper) to another object

- Used when the access to the particular object should be controlled

- Can be used to provide additional functionality to an object

Proxy

- Typical use cases for proxy pattern:
  - remote proxy: used to represent an object in a remote system
  - virtual proxy: used to represent a complex or heavy object that cannot be accessed directly
  - protection proxy: used to represent an object that requires access control

Proxy example

```
public class CiscoRouterTrackingProxy extends Device {

        private Logger logger = Logger.getLogger(…);

        private CiscoRouter ciscoRouter;

        public CiscoRouterTrackingProxy(CiscoRouter ciscoRouter) {
                this.ciscoRouter = ciscoRouter;
        }

        @Override
        public void start() {
                logger.info("Starting cisco router ...");
                ciscoRouter.start();
        }
        @Override
        public void stop() {
                logger.info("Stopping cisco router ...");
                ciscoRouter.stop();
        }
}
```

Proxy in practice

- Spring AOP:
  - uses either JDK dynamic proxies or CGLIB to create a proxy for a given target object

- Spring remoting:
  - Creates proxies for RMI/HTTP/JMS and other invoker classes

Behavioral patterns

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

Chain of responsibility

- Provides the possibility to abstract away command handlers
- Effectively decouples the client from the concrete handler classes
- Typically achieved by creating a sequence of handlers

Chain of responsibility example

```
public abstract class DeviceValidator {

        private DeviceValidator next;

        public abstract boolean validate(Device device);

        public DeviceValidator addNext(DeviceValidator validator) {
                next = validator;
                return this;
        }

        public boolean hasNext() {
                return next != null;
        }

        public DeviceValidator getNext() {
                return next;
        }

}
```

Chain of responsibility example

```
public class PriceValidator extends DeviceValidator {

        @Override
        public boolean validate(Device device) {
                return device.getPrice() > 0;
        }
}
```

```
public class SerialNumberValidator extends DeviceValidator {

        @Override
        public boolean validate(Device device) {
                return device.getSerialNumber().contains("SN");
        }

}
```

Chain of responsibility example

```
public class DeviceValidatorChain {

        public boolean validate(DeviceValidator start,
                        Device device) {
                DeviceValidator validator = start;
                boolean valid = true;
                do {
                        valid = validator.validate(device);
                        validator = validator.getNext();
                } while(valid && validator != null);

                return valid;
        }

}
```

Chain of responsibility example

```
DeviceValidator startValidator =
        new SerialNumberValidator();
startValidator.addNext(new PriceValidator());

DeviceValidatorChain validationChain =
        new DeviceValidatorChain();
boolean valid = validationChain.validate(startValidator,
        new CiscoRouter("SN 123", "router", 1000));
```

Chain of responsibility in practice

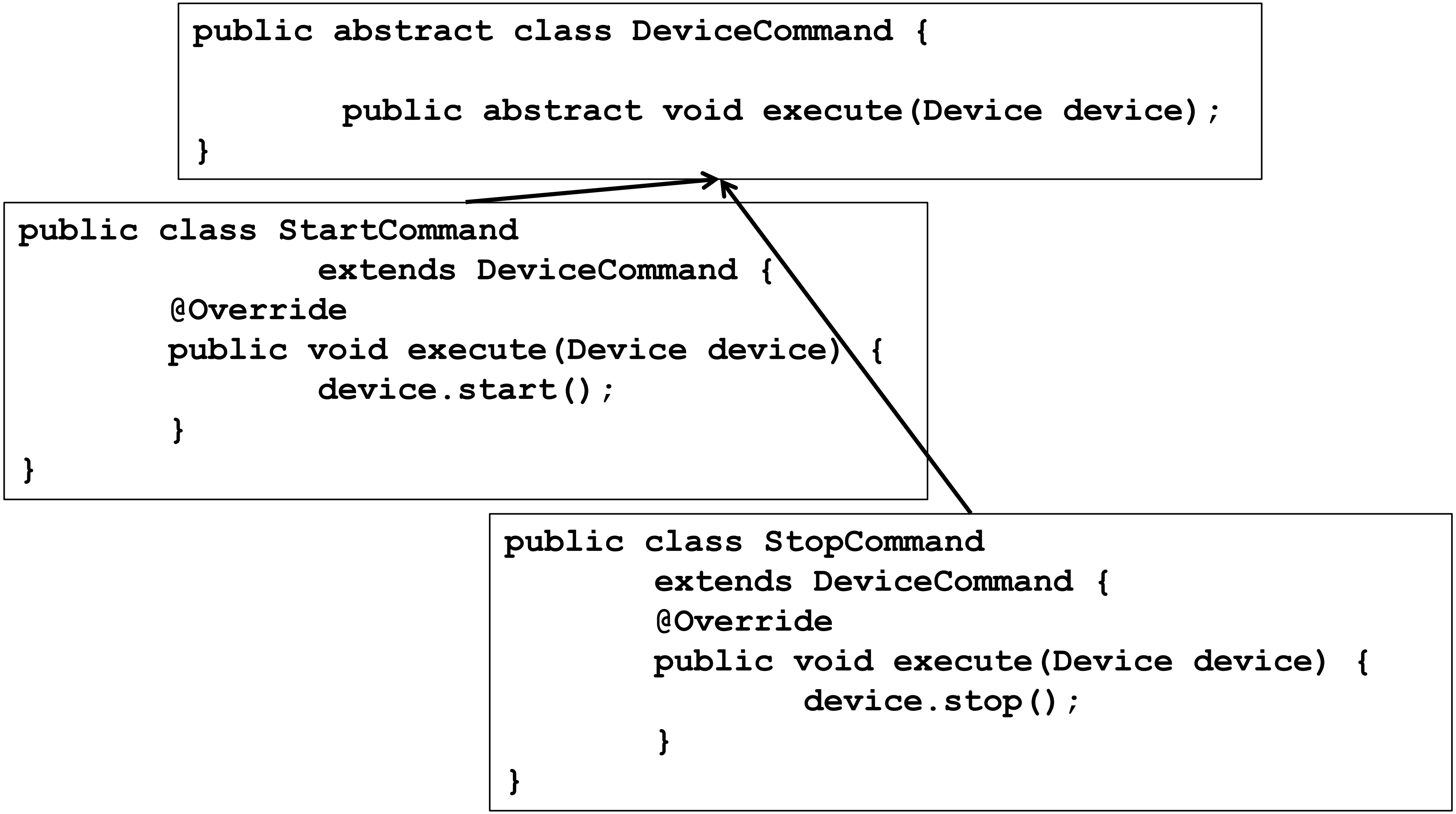
- JavaEE:
  - javax.servlet.Filter (doFilter() methods)

Command

- Provides a mechanism to decouple invoker of a particular operation from the operation itself

- A common interface for command representation is defined used by the caller

Command example



```
public abstract class DeviceCommand {

        public abstract void execute(Device device);
}
```

```
public class StartCommand
                extends DeviceCommand {
        @Override
        public void execute(Device device) {
                device.start();
        }
}
```

```
public class StopCommand
        extends DeviceCommand {
        @Override
        public void execute(Device device) {
                device.stop();
        }
}
```

Command example

```
public class DeviceController {

        private HashMap<String, DeviceCommand> commandHandlers
                 = new HashMap<String, DeviceCommand>();

        public void addCommand(String command,
                        DeviceCommand commandHandler) {
                commandHandlers.put(command, commandHandler);
        }

        public void execute(Device device, String operation) {
                DeviceCommand command =
                        commandHandlers.get(operation);
                if(command != null) {
                        command.execute(device);
                }
        }
}
```

Command example

```
DeviceController controller = new DeviceController();
controller.addCommand("start", new StartCommand());
controller.addCommand("stop", new StartCommand());

CiscoRouter router = new CiscoRouter();
controller.execute(router, "start");
```

Command in practice

- JDK:
  - java.lang.Runnable
  - javax.swing.Action

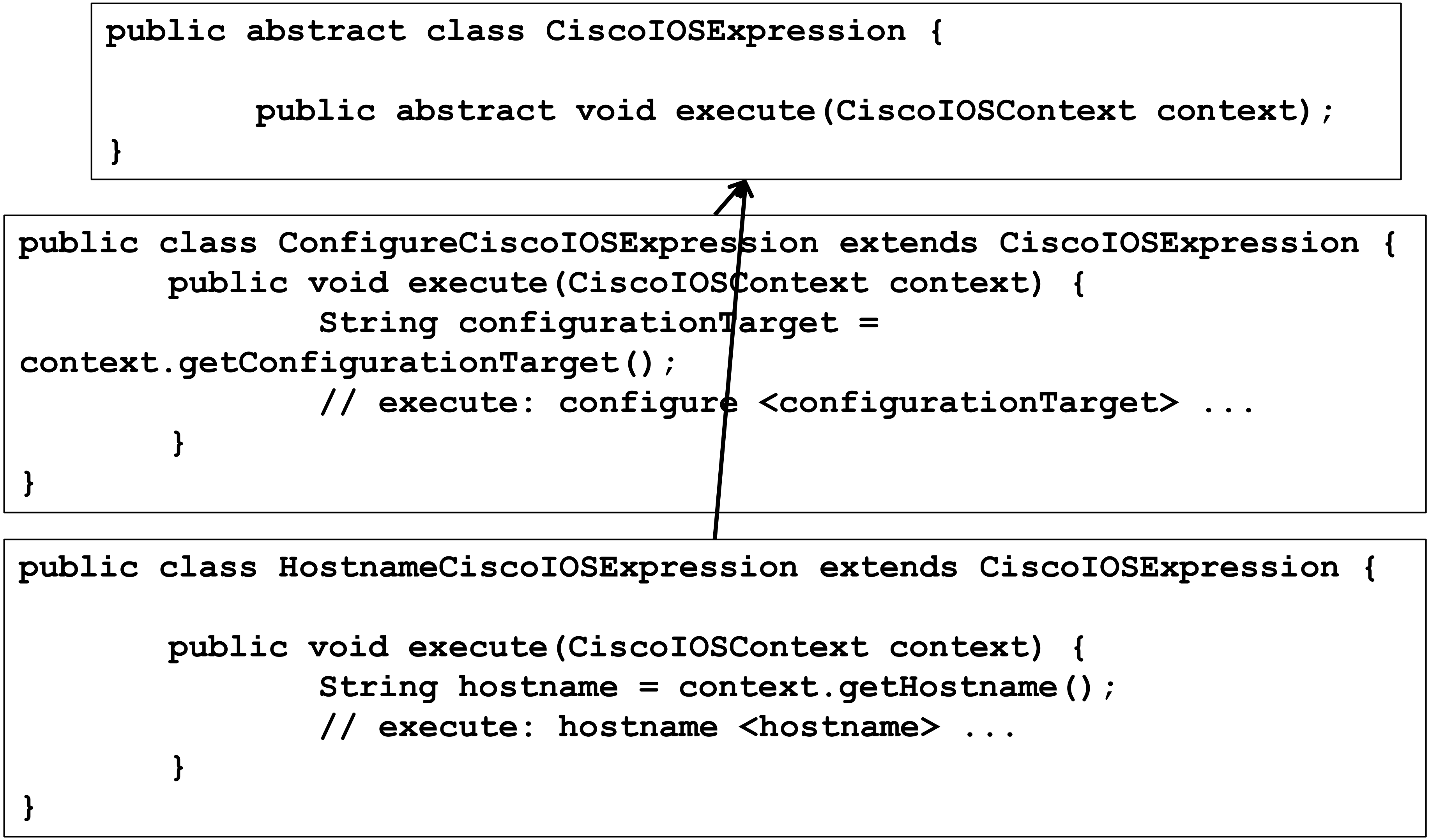
Interpreter

- Provides a mechanism to evaluate the grammar of a language

- Each element of the language is "interpreted" by a concrete interpreter class

- The structure of the interpreter classes is organized using the composite pattern

Interpreter example



```
public abstract class CiscoIOSExpression {

        public abstract void execute(CiscoIOSContext context);
}
```

```
public class ConfigureCiscoIOSExpression extends CiscoIOSExpression {
        public void execute(CiscoIOSContext context) {
                String configurationTarget =
context.getConfigurationTarget();
                // execute: configure <configurationTarget> ...
        }
}
```

```
public class HostnameCiscoIOSExpression extends CiscoIOSExpression {

        public void execute(CiscoIOSContext context) {
                String hostname = context.getHostname();
                // execute: hostname <hostname> ...
        }
}
```

Interpreter example

```
public class MultilineCiscoIOSExpression extends CiscoIOSExpression {

        private CiscoIOSExpression[] expressions;

        public MultilineCiscoIOSExpression(CiscoIOSExpression[]
                        expressions) {
                this.expressions = expressions;
        }

        public void execute(CiscoIOSContext context) {
                for( CiscoIOSExpression expession : expressions) {
                        expession.execute(context);
                }
        }

}
```

Interpreter example

```
public class CiscoIOSContext { // contains IOS-related params
        private String configurationTarget;

        private String hostname;

        public String getConfigurationTarget() {
                return configurationTarget;
        }

        public void setConfigurationTarget(String configurationTarget) {
                this.configurationTarget = configurationTarget;
        }

        public String getHostname() {
                return hostname;
        }

        public void setHostname(String hostname) {
                this.hostname = hostname;
        }
}
```

Interpreter example

```
public class CiscoIOSInterpreter {

          public void execute(String script) {

                    String[] lines = script.split("\\r?\\n");
                    CiscoIOSContext context = new CiscoIOSContext();

                    ArrayList<CiscoIOSExpression> expressions =
                              new ArrayList<CiscoIOSExpression>(lines.length);
                    for (String line : lines) {
                              if (line.startsWith("configure ")) {
                                        context.setConfigurationTarget(
                                                  line.replace("configure ", ""));
                                        expressions.add(new ConfigureCiscoIOSExpression());
                              } else if (line.startsWith("hostname ")) {
                                        context.setHostname(line.replace("hostname ", ""));
                                        expressions.add(new HostnameCiscoIOSExpression());
                              }
                    }

                    MultilineCiscoIOSExpression multilineExpression =
                              new MultilineCiscoIOSExpression(
                              expressions.toArray(new CiscoIOSExpression[0]));
                    multilineExpression.execute(context);
          }
}
```

Interpreter example

```
String script = "configure terminal\\n" +
                "hostname machine.hostname.com";

CiscoIOSInterpreter interpreter =
       new CiscoIOSInterpreter();
interpreter.execute(script);
```

Interpreter in practice

- JDK:
  - java.text.Format (DateFormat, MessageFormat, NumberFormat)

Iterator

- Provides a mechanism to access the elements of a composite object sequentially
- Hides specific details on the access mechanism
- Typically used to decouple traversal of collections from the particular collection type

Iterator example

```
public abstract class Iterator<T> {

       public abstract boolean hasNext();

       public abstract T next();
}
```

Iterator example

```
public class DeviceGroupIterator extends Iterator<Device>{

        private DeviceGroup group;

        private int currentIndex = 0;

        public DeviceGroupIterator(DeviceGroup group) {
                this.group = group;
        }

        @Override
        public boolean hasNext() {
                return currentIndex < group.getDevices().size();
        }

        @Override
        public Device next() {
                return group.getDevices().get(currentIndex++);
        }

}
```

Iterator example

```
public class DeviceController {

        public void startCiscoDevices(DeviceGroup deviceGroup) {

                DeviceGroupIterator iterator =
                        new DeviceGroupIterator(deviceGroup);

                while (iterator.hasNext()) {
                        iterator.next().start();
                }

        }
}
```

Iterator example

```
DeviceGroup group = new DeviceGroup();
group.addDevice(new CiscoRouter());
group.addDevice(new JuniperRouter());

DeviceController controller =
        new DeviceController();
controller.startCiscoDevices(group);
```

Iterator in practice

- JDK:
  - all implementations of java.util.Iterator
  - all implementations of java.util.Enumeration

Mediator

- Provides a mediator object through which communication between objects happens

  Reduces coupling between objects and simplifies communication

Mediator example

```
public class CiscoDevice {

        private CiscoDeviceManager manager;

        public void start() {
        }

        public void stop() {
        }

        public void executeScript(String script) {
                manager.executeScript(script);
        }
}
```

Mediator example

```
public class CiscoIOSInterpreter {

        private CiscoDeviceManager manager;

        public void execute(String script) {
        }

        public void startDevice() {
                manager.startDevice();
        }

        public void stopDevice() {
                manager.stopDevice();
        }
}
```

Mediator example

```
public class CiscoDeviceManager { // mediator
        private CiscoIOSInterpreter interpreter;
        private CiscoDevice ciscoDevice;

        public void setInterpreter(CiscoIOSInterpreter interpreter) {
                this.interpreter = interpreter;
        }
        public void setCiscoDevice(CiscoDevice ciscoDevice) {
                this.ciscoDevice = ciscoDevice;
        }
        public void executeScript(String script) {
                interpreter.execute(script);
        }
        public void startDevice() {
                ciscoDevice.start();
        }
        public void stopDevice() {
                ciscoDevice.start();
        }
}
```

Mediator in practice

- JDK:
  - java.util.concurrent.Executor (execute() method)
  - java.util.concurrent.ExecutorService (submit() method)
  - javax.swing.ButtonModel

Memento

- Provides a mechanism to store object’s internal state
- In additional to storing it is also responsible to provide capabilities for restoring of object’s state

Memento example

```
public class Device {

        private String serialNumber;
        private String shortName;
        private double price;
        private String configScript;

        public Device(String serialNumber, String shortName,
                       double price) {…}

        public DeviceSnapshot saveConfiguration() {
                DeviceSnapshot snapshot = new DeviceSnapshot();
                snapshot.setConfigurationScript(configScript);
                return snapshot;
        }

        public void restoreConfiguration(DeviceSnapshot snapshot) {
                this.configScript = snapshot.getConfigScript();
        }
}
```

Memento example

```
// represents a momento
public class DeviceSnapshot {

        private String configScript;

        public String getConfigScript() {
                return configScript;
        }

        public void setConfigScript(String configScript) {
                this.configScript = configScript;
        }
}
```

Memento example

```
Device device = new Device("SN 123", "router", 30);
device.setConfigScript("configure terminal\n″ +
                       ″hostname machine.test.com");
DeviceSnapshot startShapshot = device.saveConfiguration();

device.setConfigScript("configure terminal\n″ +
                       ″hostname another.test.com");
DeviceSnapshot anotherShapshot = device.saveConfiguration();

device.restoreConfiguration(startShapshot);
System.out.println(device.getConfigurationScript());
```

Memento in practice

- JDK:
  - all implementations of java.io.Serializable

- JavaEE:
  - all implementations of javax.faces.component.StateHolder

Observer

- Provides a mechanism for an object to notify a set of dependents (“observers”) for changes
- The notifying object is also called a “source” and dependents are called “sinks”

Observer

- Also applied as an architectural concepts and provides the building block for distributed event handling systems (such as message brokers)

- Some languages (like C#) provide built-in support for the observer pattern (Java is not one of them at present)

Observer example

```
public class Device {

        private List<Device> connectedDevices =
                new LinkedList<>();
        private String serialNumber;
        private String shortName;
        private double price;

        public Device(String serialNumber, String shortName,
                        double price) {…}

        public void addConnectedDevice(Device device) {
                connectedDevices.add(device);
        }
        public void restart() {
                // restart current device ...
                for(Device connectedDevice : connectedDevices) {
                        connectedDevice.restart();
                }
        }
}
```

Observer example

```
Device device = new Device("SN 123", "router", 30);
Device switch1 = new Device("SN 124", "switch1", 10);
Device switch2 = new Device("SN 125", "switch2", 10);
device.addConnectedDevice(switch1);
device.addConnectedDevice(switch2);
device.restart();
```

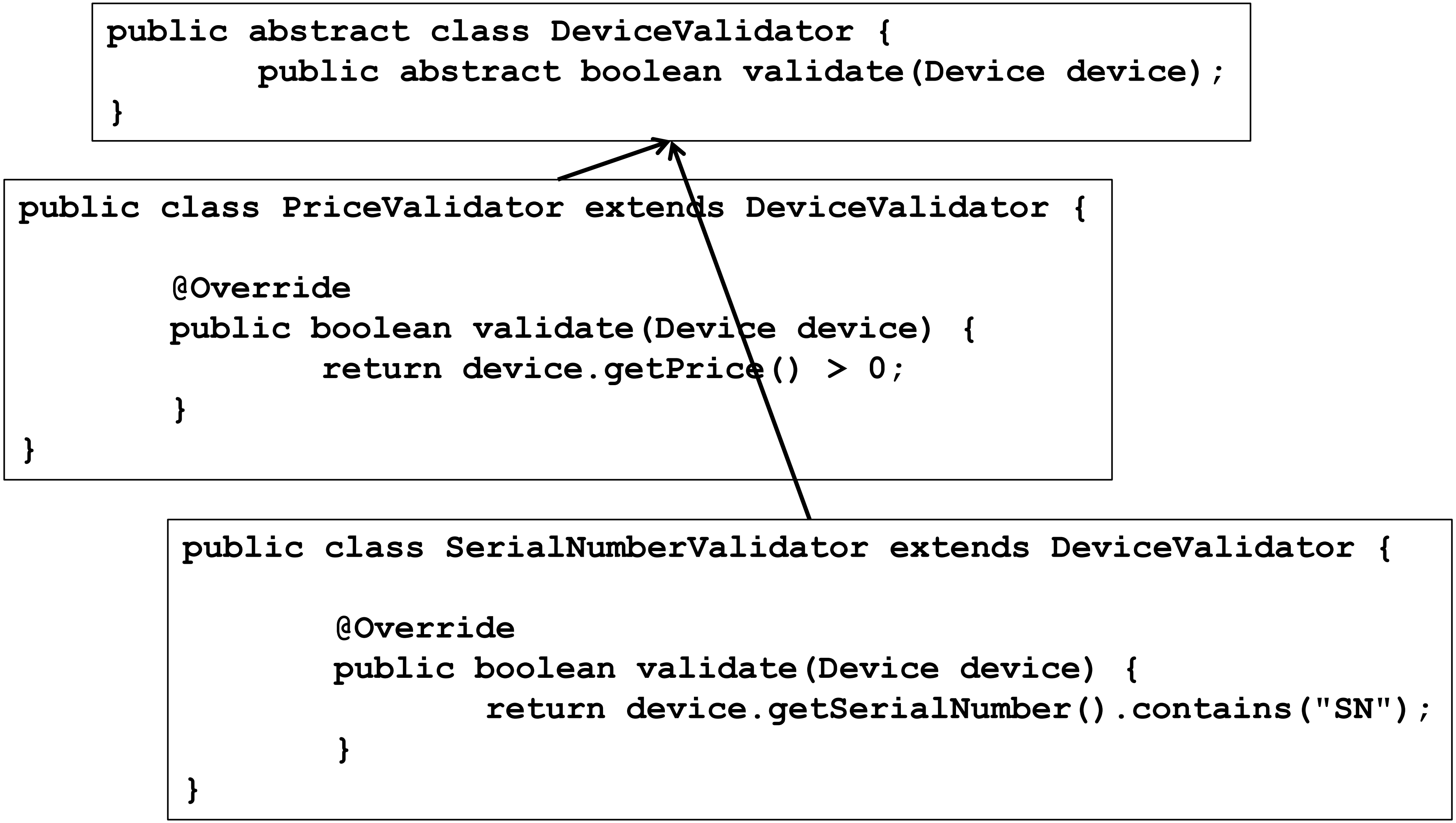
Observer in practice

- JDK:
  - java.util.Observer/java.util.Observable
  - java.util.EventListener
- Spring framework:
  - ApplicationContext's event mechanism
- JavaEE:
  - servlet listeners

Strategy

- Provides the possibility to vary an algorithm at runtime
- Decouples the client from the concrete algorithm implementation
- A base class provides the abstract method that must be implemented by the concrete implementations provided by the subclasses

Strategy example



```
public abstract class DeviceValidator {
        public abstract boolean validate(Device device);
}
```

```
public class PriceValidator extends DeviceValidator {

        @Override
        public boolean validate(Device device) {
                return device.getPrice() > 0;
        }
}
```

```
public class SerialNumberValidator extends DeviceValidator {

        @Override
        public boolean validate(Device device) {
                return device.getSerialNumber().contains("SN");
        }
}
```

Strategy in practice

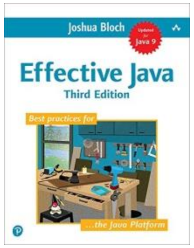
- JDK:
  - java.util.List (sort() method)
  - java.util.Comparator (compare() method)

Template method

- Defines a skeleton method that uses high-level (abstract) operations to define the behavior of the method

- Can be used in combination with strategy pattern when the algorithm implementations can be implemented using a similar structure

- A base class provides the template method and all subclasses need to provide implementations of the high-level operations



Favor the use of standard functional interfaces

Template method example

```
public abstract class DeviceConfigurationValidator {

        public boolean validate(Device device) {
                boolean result =
                        validateConfigurationSyntax() &&
                        validateCommandParameters();
                return result;
        }

        protected abstract boolean validateCommandParameters();

        protected abstract boolean validateConfigurationSyntax();
}
```

Template method example

```
public abstract class CiscoConfigurationValidator
        extends DeviceConfigurationValidator {

        protected boolean validateConfigurationSyntax() {
                boolean result = true;
                // validate Cisco configuration syntax ...
                return result;
        }

        protected boolean validateCommandParameters() {
                boolean result = true;
                // validate Cisco configuration command parameters ...
                return result;
        }
}
```

Template method example

```
public abstract class JuniperConfigurationValidator
        extends DeviceConfigurationValidator {

        protected boolean validateConfigurationSyntax() {
                boolean result = true;
                // validate Juniper configuration syntax ...
                return result;
        }

        protected boolean validateCommandParameters() {
                boolean result = true;
                // validate Juniper configuration command parameters ...
                return result;
        }
}
```

Template method in practice

- JDK:
  - all non-abstract methods of java.io.InputStream, java.io.OutputStream, java.io.Reader and java.io.Writer
  - all non-abstract methods of java.util.AbstractList, java.util.AbstractSet and java.util.AbstractMap

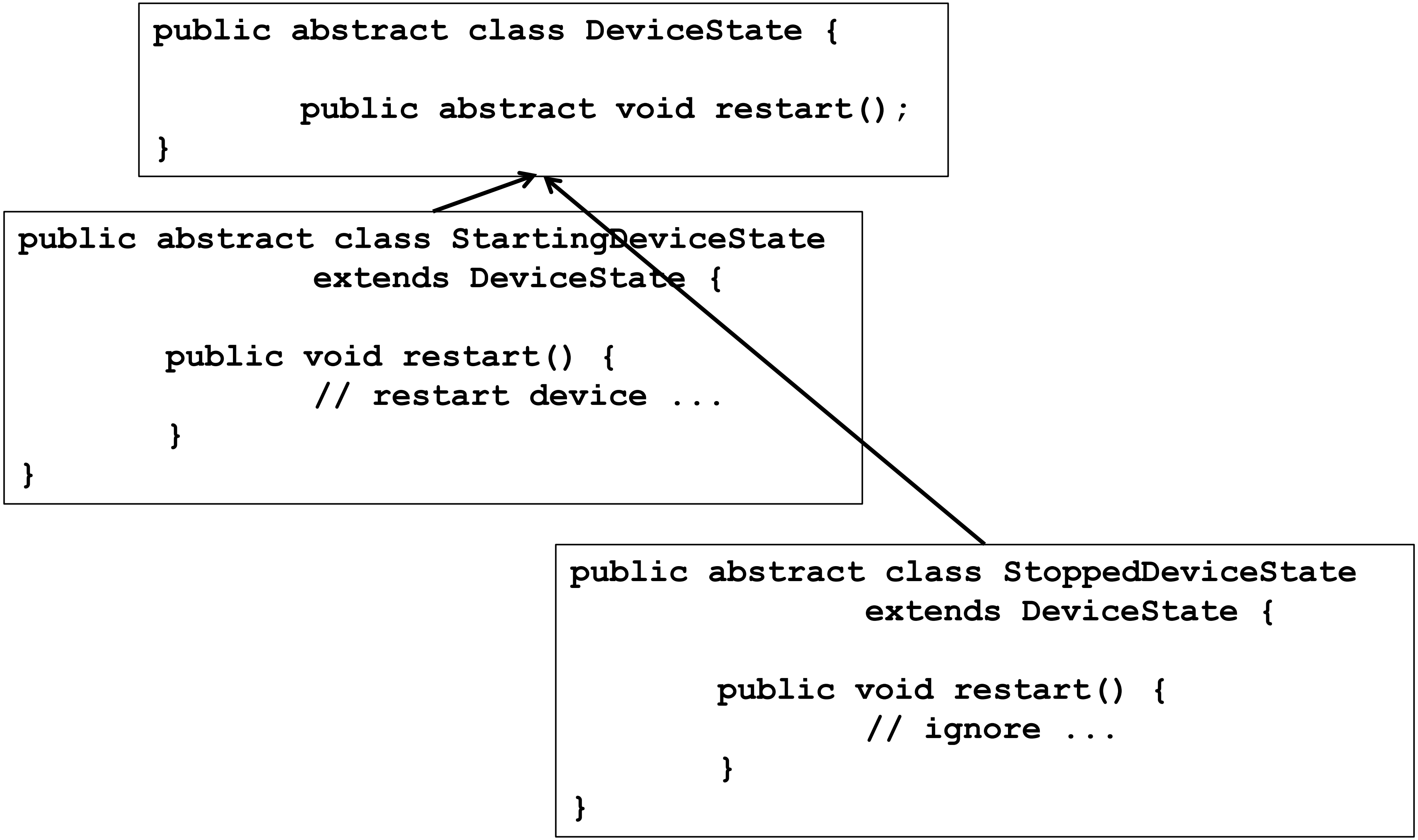
State

- Provides a mechanism to encapsulate varying behavior for the same object

- Used when an object needs to act differently when its internal state changes

- Can be implemented as a strategy pattern through the state’s interface

State example



```
public abstract class DeviceState {

        public abstract void restart();
}
```

```
public abstract class StartingDeviceState
                extends DeviceState {

        public void restart() {
                // restart device ...
        }
}
```

```
public abstract class StoppedDeviceState
                extends DeviceState {

        public void restart() {
                // ignore ...
        }
}
```

State example

```
public class Device {

        private String serialNumber;
        private String shortName;
        private double price;
        private DeviceState state;

        public Device(String serialNumber,
        String shortName, double price) {…}
        …
        public void setState(DeviceState state) {
                this.state = state;
        }
        public DeviceState getState() {
                return state;
        }
        public void restart() {
                state.restart();
        }
}
```

State in practice

- JavaEE:
  - javax.faces.lifecycle.LifeCycle (execute() method behavior is different depending on the current phase of the JSF lifecycle)

Visitor

- Provides a mechanism to separate an algorithm from the object structure on which it operates
- Each node in the object structure can apply an algorithm (visitor) that is represented by a common interface
- The visitors are typically organized as a strategy pattern

Visitor example

```
public class Device {

        private String serialNumber;

        private String shortName;

        private double price;

        public Device(String serialNumber,
                String shortName, double price) {…}
        …
        public void validate(DeviceValidator validator) {
                validator.validate(this);
        }
}
```

Visitor example

```
Device ciscoRouter = new CiscoRouter("SN 123", "router", 30);
Device juniperRouter = new JuniperRouter("SN 127", "router", 20);

DeviceValidator validator = new SerialNumberValidator();
ciscoRouter.validate(validator);
juniperRouter.validate(validator);
```

Visitor in practice

- JDK:
  - javax.lang.model.element.AnnotationValue/AnnotationValueVisitor
  - javax.lang.model.element.Element/ElementVisitor
  - javax.lang.model.type.TypeMirror/TypeVisitor
  - java.nio.file.FileVisitor/SimpleFileVisitor

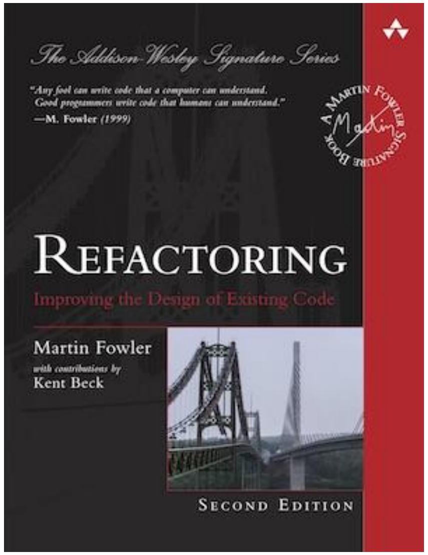
Design patterns:
areas of active research

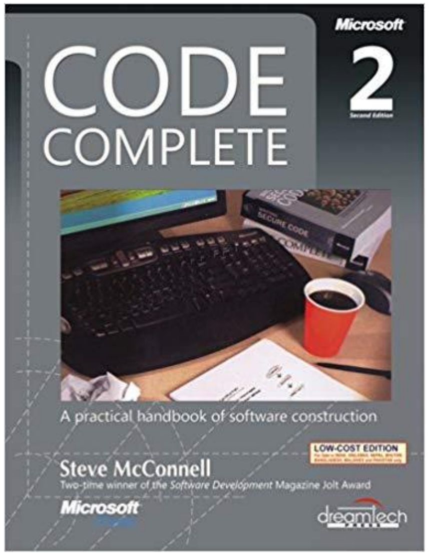
- Application of design patterns in large projects
- Tools trying to discover source code eligible for refactoring with design patterns
- Design pattern mining

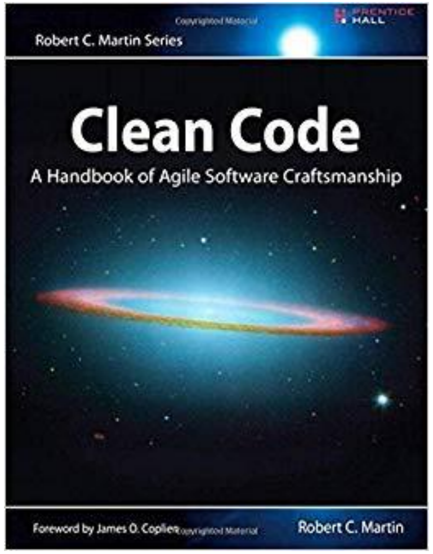
Applying design patterns

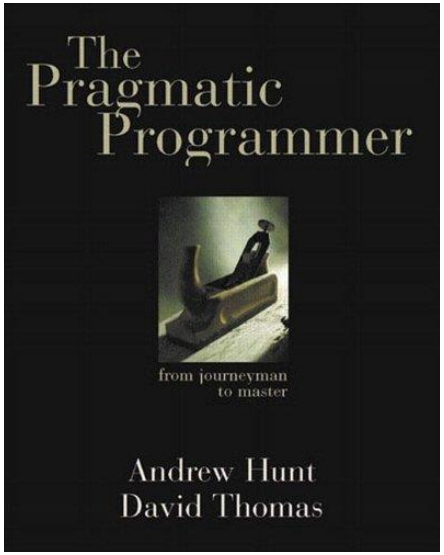
Questions ?

References

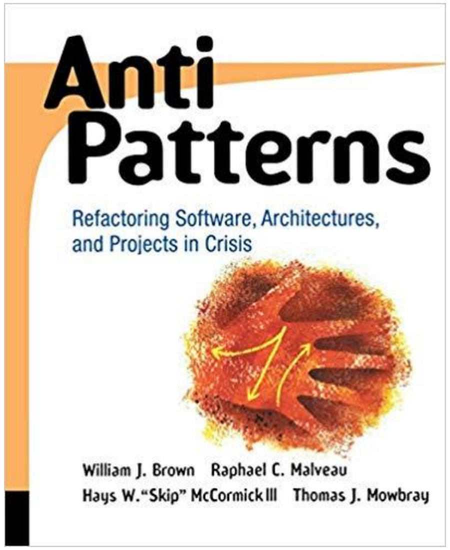

The Addison-Wesley Signature Series
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler (1999)
A Martin Fowler Signature Book
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition


Microsoft
Code Complete
2
Second Edition
A practical handbook of software construction
Low-Cost Edition
Steve McConnell
Two-time winner of the Software Development Magazine Jolt Award
Microsoft
dreamtech


Robert C. Martin Series
Prentice Hall
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin


The Pragmatic Programmer
from journeyman to master
Andrew Hunt
David Thomas

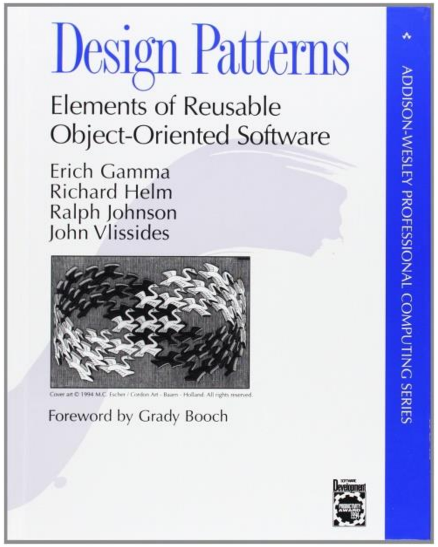
References



AntiPatterns

Refactoring Software, Architectures, and Projects in Crisis

William J. Brown Raphael C. Malveau
Hays W. "Skip" McCormick III Thomas J. Mowbray



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

O'REILLY® Copyrighted Material

10th Anniversary Updated for Java 8

Head First Design Patterns

A Brain-Friendly Guide

Avoid those embarrassing coupling mistakes

Learn why everything your friends know about Factory pattern is probably wrong

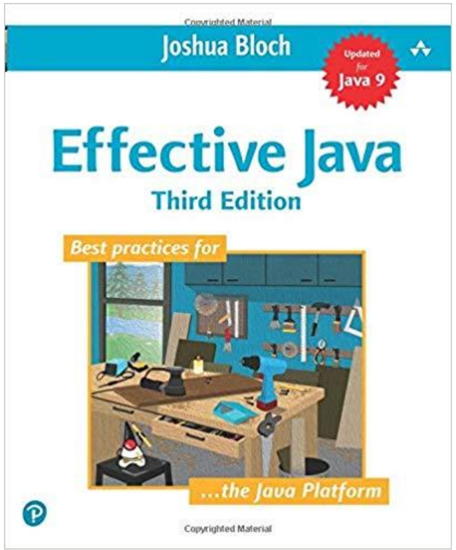
Discover the secrets of the Patterns Guru

Load the patterns that matter straight into your brain

Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern

See why Jim's love life improved when he cut down his inheritance

Eric Freeman & Elisabeth Robson
with Kathy Sierra & Bert Bates



Copyrighted Material

Joshua Bloch

Updated for Java 9

Effective Java

Third Edition

Best practices for ...the Java Platform

Copyrighted Material