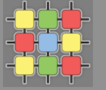
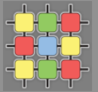


Architecture of software systems

Lecture 2: Design patterns

David Šišlák
david.sislak@fel.cvut.cz

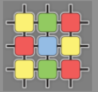
Immutable object



- thread-safe object
- all fields are **final** (and often **private**)
- no “setters”
- block changes by finalization of the class
- block override
  use private constr. and static factories
- don’t need a copy constr.
- no clone needed
- hashCode can use lazy initialization and cache
- no defensive copy when used as a field
- suitable for map and set elements

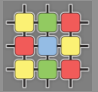
```java
final public class ImmutableRGB {

    //Values must be between 0 and 255.
    final private int red;
    final private int green;
    final private int blue;
    final private String name;

    private void check(int red, int green, int blue) {
        if (red < 0 || red > 255
                || green < 0 || green > 255
                || blue < 0 || blue > 255) {
            throw new IllegalArgumentException();
        }
    }

    public ImmutableRGB(int red, int green, int blue, String name) {
        check(red, green, blue);
        this.red = red;
        this.green = green;
        this.blue = blue;
        this.name = name;
    }


    public int getRGB() {
        return ((red << 16) | (green << 8) | blue);
    }

    public String getName() {
        return name;
    }

    public ImmutableRGB invert() {
        return new ImmutableRGB(255 - red, 255 - green, 255 - blue,
                "Inverse of " + name);
    }
}
```

Immutable object - String



» special behavior of concatenation operator (+)

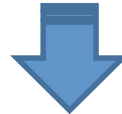
```
String a = "a";
String b = "b";
String c = "c";

String q = a + b +c;
```



Immutable object - String

- special behavior of concatenation operator (+)
  - implemented through the **StringBuilder** and its *append* method

```
String a = "a";
String b = "b";
String c = "c";

String q = a + b +c;
```

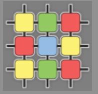


```
StringBuilder temp = new StringBuilder(a);
temp.append(b);
temp.append(c);
String q = temp.toString();
```

- but what about?

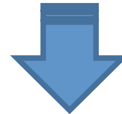
```
String q = a;
q += b;
q += c;
```

Immutable object - String



- special behavior of concatenation operator (+)
  - implemented through the **StringBuilder** and its *append* method

```
String a = "a";
String b = "b";
String c = "c";

String q = a + b +c;
```



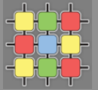
```
StringBuilder temp = new StringBuilder(a);
temp.append(b);
temp.append(c);
String q = temp.toString();
```

- but what about?

```
String q = a;
q += b;
q += c;
```

  - implies 4 object allocations !!!
- consider manual usage of **StringBuilder** or **StringBuffer** (thread-safe)

Immutable object



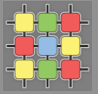
- concatenation of non-String types:
  - String.valueOf({primitives})
  - or

```
public static String valueOf(Object obj) {
return (obj == null) ? "null" : obj.toString();
}
```

- make collections/list/map/set immutable (UnsupportedOperationException):
  - **Collections.unmodifiableCollection(…)**
  - **Collections.unmodifiableList(…)**
  - **Collections.unmodifiableMap(…)**
  - **Collections.unmodifiableSet(…)**
  - **Collections.unmodifiableSortedMap(…)**
  - **Collections.unmodifiableSortedSet(…)**

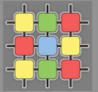
  - elements are not protected !!! use immutable elements too

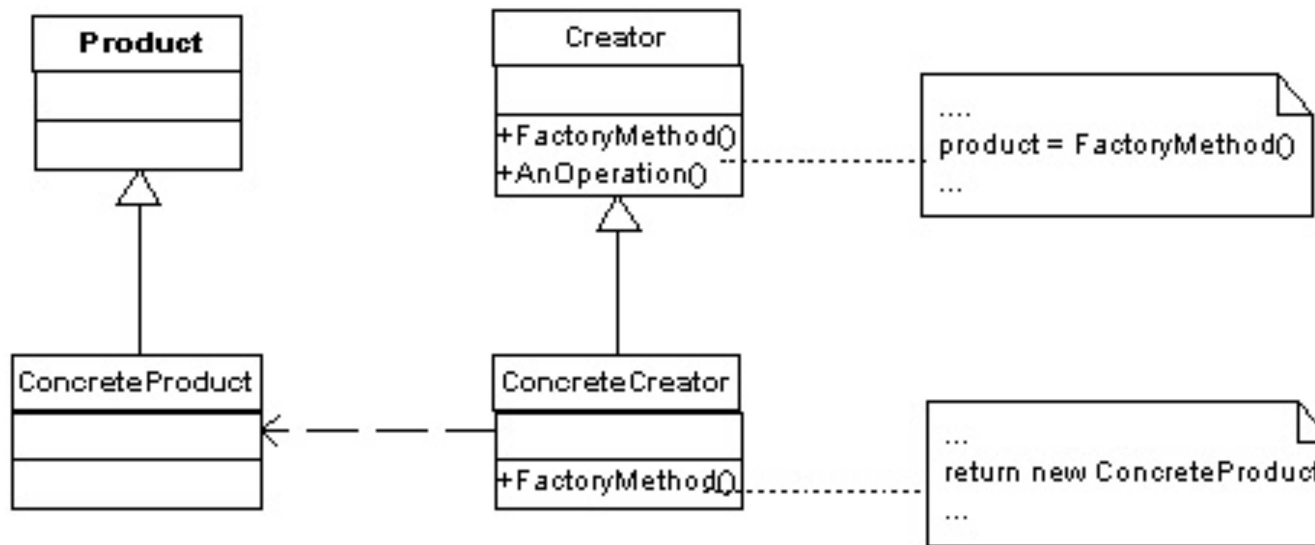
Factory method



- *creational* design pattern
- based on the concept of factories
- define a method for **creating objects in the interface which are subclasses of specific product**
- implementers can decide which class is instantiated (e.g. based on params)
- common usage
  - **toolkits and frameworks**
    - library code needs to create objects of types which are sub-classed by application using the framework
  - **test-driven development**
    - unit tests can use mock objects to simulate operations
- limitations
  - if use private constructors -> class cannot be extended
  - if use protected constructors -> subclass must re-implement all factory methods with exactly the same signatures (static cannot be overriden)

Factory method

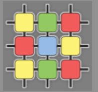




» Creator can define default implementation returning default factory product

» Note:
  » static factory methods – cannot be overridden !
  » Factory object can be instance

Factory method – example



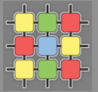
» product interface

```java
public interface Trace {
      // turn on and off debugging
      public void setDebug( boolean debug );
      // write out a debug message
      public void debug( String message );
      // write out an error message
      public void error( String message );
}
```

» product A

```java
public class SystemTrace implements Trace {
      private boolean debug;
      public void setDebug( boolean debug ) {
            this.debug = debug;
      }
      public void debug( String message ) {
            if( debug ) {  // only print if debug is true
                  System.out.println( "DEBUG: " + message );
            }
      }
      public void error( String message ) {
            // always print out errors
            System.out.println( "ERROR: " + message );
      }
}
```

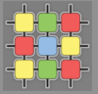
Factory method – example



» product B

```
public class FileTrace implements Trace {

      private java.io.PrintWriter pw;
      private boolean debug;
      public FileTrace() throws java.io.IOException {
            // a real FileTrace would need to obtain the filename somewhere
            // for the example I'll hardcode it
            pw = new java.io.PrintWriter( new java.io.FileWriter( "c:\trace.log" ) );
      }
      public void setDebug( boolean debug ) {
            this.debug = debug;
      }
      public void debug( String message ) {
            if( debug ) {  // only print if debug is true
                  pw.println( "DEBUG: " + message );
                  pw.flush();
            }
      }
      public void error( String message ) {
            // always print out errors
            pw.println( "ERROR: " + message );
            pw.flush();
      }
}
```

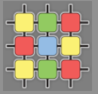
Factory method – example



- direct usage
  - need change all instantiations to modify behavior

```
//... some code ...
SystemTrace log = new SystemTrace();
//... code ...
log.debug( "entering loog" );
// ... etc ...
```

Factory method – example



- direct usage
  - need change all instantiations to modify behavior

```
//... some code ...
SystemTrace log = new SystemTrace();
//... code ...
log.debug( "entering loog" );
// ... etc ...
```

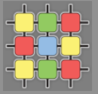
- factory method

```
public class TraceFactory {
      public static Trace getTrace() {
            return new SystemTrace();
      }
}
//... some code ...
Trace log =     TraceFactory.getTrace();
//... code ...
log.debug( "entering loog" );
// ... etc ...
```

  - just one place to modify behavior

```
public class TraceFactory {
      public static Trace getTrace() {
            try {
                  return new FileTrace();
            } catch ( java.io.IOException ex ) {
                  Trace t = new SystemTrace();
                  t.error( "could not instantiate FileTrace: " + ex.getMessage() );
                  return t;
            }
      }
}
```

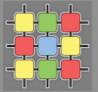
Lazy initialization



- *delay operation until the first time it is needed*
  - **lazy object creation**
  - **lazy calculation of a value**
  - **lazy class loading**
  - **lazy other expensive process**
- use a flag indicating that the process has taken place
- if not used -> save memory usage and/or processing time

- **lazy class loading**
  - classes are loaded only when they are first referenced
  - *use interfaces or parent classes for field types*

Lazy initialization – example



```
public class MyFrame extends Frame
{
  private MessageBox mb_ = new MessageBox();
  //private helper used by this class
  private void showMessage(String message)
  {
    //set the message text
    mb_.setMessage( message );
    mb_.pack();
    mb_.show();
  }
}
```

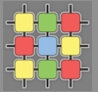
VS

```
public final class MyFrame extends Frame
{
  private MessageBox mb_ ; //null, implicit
  //private helper used by this class
  private void showMessage(String message)
  {
    if(mb_==null)//first call to this method
      mb_=new MessageBox();
    //set the message text
    mb_.setMessage( message );
    mb_.pack();
    mb_.show();
  }
}
```

» higher importance for complex objects (e.g. Image, DB connection)

» used often for lazy hash code computation in immutable objects

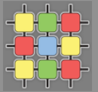
Singleton



- *class has only one instance with a global point of access to it*
- often used to control access to native resources like database connections or sockets
- unique repository of state, alternatively can be implemented as static
- lazy instantiation:

```
public static class MySingleton {
    private static MySingleton _instance;

    private MySingleton() {
        // ...
    }

    public static MySingleton getInstance() {
        if (_instance==null) {
            _instance = new MySingleton();
        }
        return _instance;
    }
    // ...
}
```

Singleton

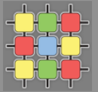


- *class has only one instance with a global point of access to it*
- often used to control access to native resources like database connections or sockets
- unique repository of state, alternatively can be implemented as static
- lazy instantiation:

```
public static class MySingleton {
    private static MySingleton _instance;

    private MySingleton() {
        // ...
    }

    public static synchronized MySingleton getInstance() {
        if (_instance==null) {
            _instance = new MySingleton();
        }
        return _instance;
    }
    // ...
}
```

- how to avoid locking?

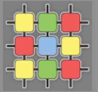
Double-checked locking



- **reduce the overhead** of acquiring a lock by use of **locking criterion**
- common usage
  - multi-threaded environment
  - combination with lazy initialization

- typical use pattern
  - check the locking criterion without obtaining the lock
  - obtain the lock
  - double-check whether the variable has been already initialized
  - otherwise, initialize

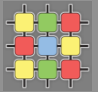
Singleton – double-checked locking



» avoid expense of locking

```
public static class MySingleton {
    private static MySingleton _instance;

    private MySingleton() {
        // ...
    }

    public static MySingleton getInstance() {
        if (_instance==null) {
            synchronized (MySingleton.class) {
                _instance = new MySingleton();
            }
        }
        return _instance;
    }
    // ...
}
```

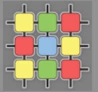
Singleton – double-checked locking



» avoid expense of locking

```
public static class MySingleton {
    private static MySingleton _instance;

    private MySingleton() {
        // ...
    }

    public static MySingleton getInstance() {
        if (_instance==null) {
            synchronized (MySingleton.class) {
                if (_instance==null) {
                    _instance = new MySingleton();
                }
            }
        }
        return _instance;
    }
    // ...
}
```

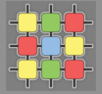
Singleton – double-checked locking



» avoid expense of locking

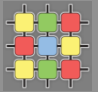
```
public static class MySingleton {
    private static volatile MySingleton _instance;

    private MySingleton() {
        // ...
    }

    public static MySingleton getInstance() {
        if (_instance==null) {
            synchronized (MySingleton.class) {
                if (_instance==null) {
                    _instance = new MySingleton();
                }
            }
        }
        return _instance;
    }
    // ...
}
```

Singleton initialization – eager initialization



```
public static class MySingleton {
    private static MySingleton _instance =
     new MySingleton();

    private MySingleton() {
        // ...
    }

    public static MySingleton getInstance() {
     return _instance;
    }

    // ...
}
```

Singleton initialization - lazy initialization



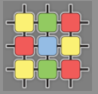
- initialization on demand holder idiom

```
class Foo {
    private static class HelperHolder {
       public static Helper helper = new Helper();
    }
    public static Helper getHelper() {
        return HelperHolder.helper;
    }
}
```

- inner class are not loaded until they are referenced

- BUT
  - if construction fail, it throws **NoClassDefFoundError** during run-time !

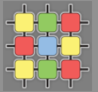
Serializable Singleton - example



```
public static class MySingleton implements Serializable {
    private static MySingleton _instance = new MySingleton();

    private MySingleton() {
        // ...
    }

    public static MySingleton getInstance()
    {
      return _instance;
    }
    // ...
}
```

» is this correct singleton?

Serializable Singleton - example



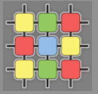
```
public static class MySingleton implements Serializable {
    private static MySingleton _instance = new MySingleton();

    private MySingleton() {
        // ...
    }

    public static MySingleton getInstance()
    {
      return _instance;
    }
    // ...
}
```

- is this correct singleton?
  - **NO** – due to serialization which will create new instance

- Fix – using read resolve

```
private Object readResolve() {
    return _instance;
}
```

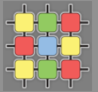
Serializable Singleton – using ENUM



```java
public enum MySingleton {
    INSTANCE;

    private boolean test;

    MySingleton() {
        test = true;
    }

    public void myMethod() {
        // ...
    }
}
```

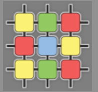
- serialization issue is fixed by default
- creation is thread-safe
- cannot inherit other enums or classes (only can implement interfaces)
- no lazy intialization
- different classloaders is still an issue

Singleton – issues



- don't allow subclassing due to static getInstance():
  - factory class with method returning singleton instance (requires non-private constructor)
- issue with more VMs in distributed system (e.g. RMI) -> singleton should not be used to store state !!!
- classloaders -> one singleton per each classloader
- *in older JVMs private static references for non-reachable objects was not enough to keep that instance*

Multiton

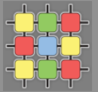


- *parametric singleton*, single instance with given parameter
- often in combination with immutable object

```
public static class MyMultiton {
    private static final Map<Object, MyMultiton> instances = new HashMap<Object, MyMultiton>();

    private final Object attribute;

    private MyMultiton(final Object attribute) {
        this.attribute = attribute;
    }

    public static MyMultiton getInstance(Object attribute) {
        synchronized (instances) {
            MyMultiton instance = instances.get(attribute);
            if (instance == null) {
                instance = new MyMultiton(attribute);
                instances.put(attribute, instance);
            }
            return instance;
        }
    }
}
```

- beware memory consumption !
- use WeakReference or SoftReference

Breaking singleton or multiton

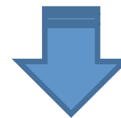


» usage of reflection to create instance

```
public static void main(String[] args) throws Exception {
    MySingleton s = MySingleton.getInstance();

    Class<?> cl = MySingleton.class;

    Constructor<?> cons = cl.getDeclaredConstructor();
    cons.setAccessible(true);

    MySingleton s2 = (MySingleton) cons.newInstance();
}
```

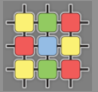
» Fix 1

```
SecurityManager mgr = new SecurityManager();
System.setSecurityManager(mgr);
```



```
Exception in thread "main" java.security.AccessControlException: access denied (java.lang.r
eflect.ReflectPermission suppressAccessChecks)
 at java.security.AccessControlContext.checkPermission(AccessControlContext.java:323)
 at java.security.AccessController.checkPermission(AccessController.java:546)
 at java.lang.SecurityManager.checkPermission(SecurityManager.java:532)
 at java.lang.reflect.AccessibleObject.setAccessible(AccessibleObject.java:107)
```

Breaking singleton or multiton



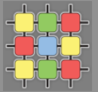
- usage of reflection to create instance

```
public static void main(String[] args) throws Exception {
    MySingleton s = MySingleton.getInstance();

    Class<?> cl = MySingleton.class;

    Constructor<?> cons = cl.getDeclaredConstructor();
    cons.setAccessible(true);

    MySingleton s2 = (MySingleton) cons.newInstance();
}
```

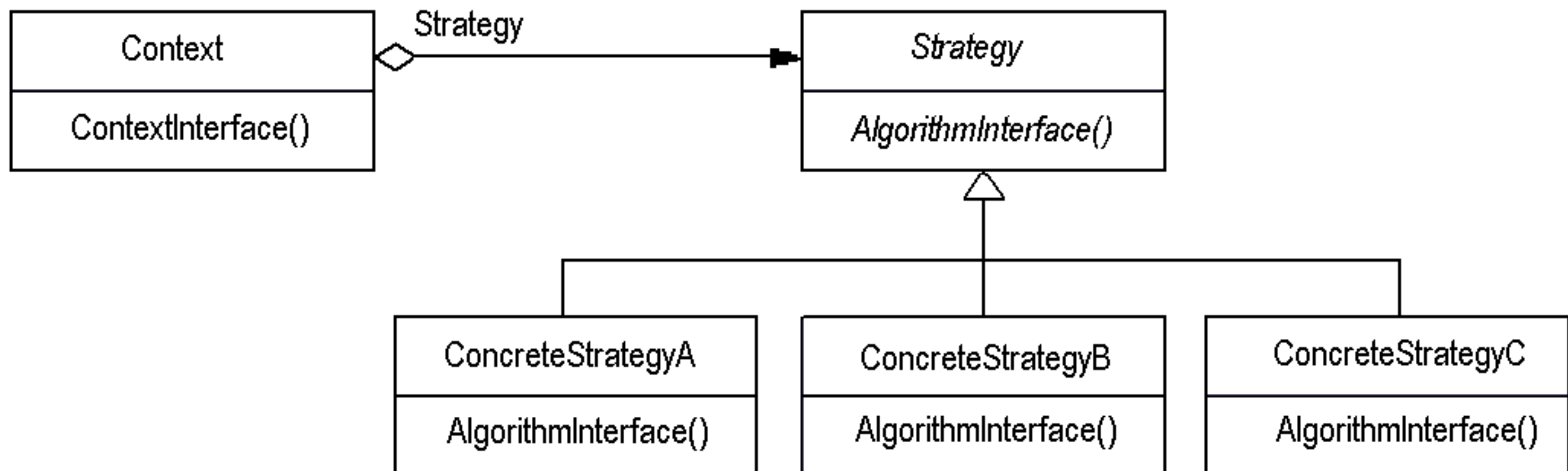
- Fix 2

```
public static class MySingleton {
    private static MySingleton _instance = new MySingleton();

    private MySingleton() {
        if (_instance != null) {
            throw new IllegalStateException("Singleton instance already created")
        }
        // ...
    }

    public static MySingleton getInstance()
    {
      return _instance;
    }
    // ...
}
```

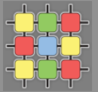
Strategy



- *behavioral* design pattern
- **selection of algorithm in runtime**
- define common interface for family of algorithms
- make different implementations interchangeable
- usually strategy is selected independently from clients that use it



Strategy - example



» Strategy

```
public interface SortInterface {
public void sort(double[] list);
}
```

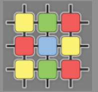
» Context

```
public class SortingContext {
private SortInterface sorter = null;

public void sortDouble(double[] list) {
 sorter.sort(list);
}

public SortInterface getSorter() {
 return sorter;
}

public void setSorter(SortInterface sorter) {
 this.sorter = sorter;
}
}
```

» Initialization

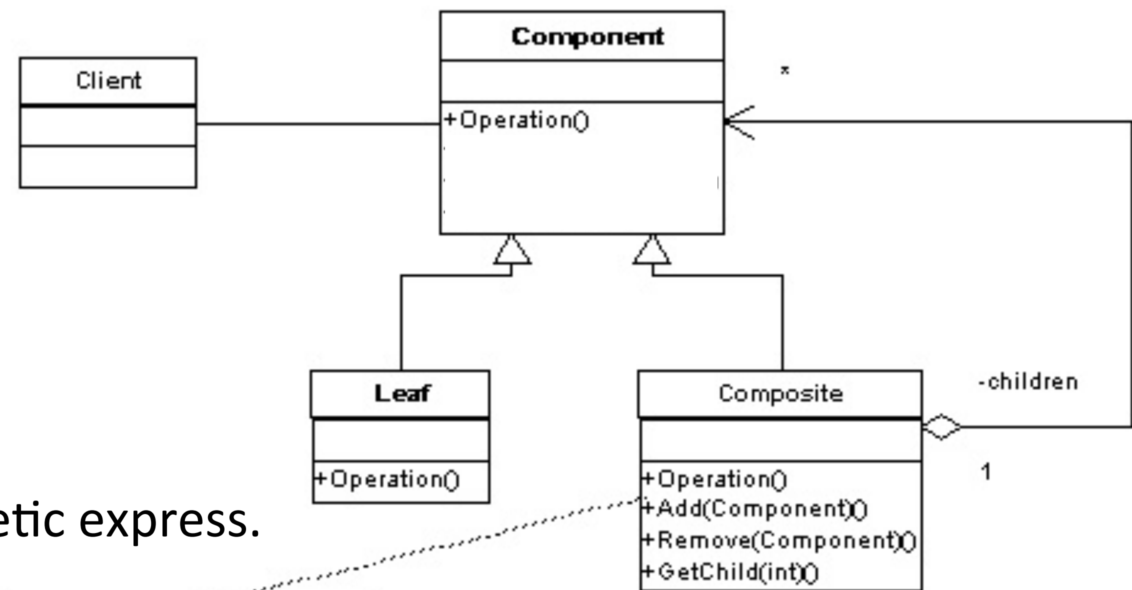
```
public class SortingClient {
public class SortingClient {
 public static void main(String[] args) {
  double[] list = {1,2.4,7.9,3.2,1.2,0.2,10.2,22.5,19.6,14,12,16,17};
  SortingContext context = new SortingContext();
  context.setSorter(new BubbleSort());
  context.sortDouble(list);
  for(int i =0; i< list.length; i++) {
   System.out.println(list[i]);
  }
 }
}
```

Composite



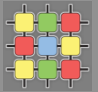
- *structural* design pattern
- **compose objects into tree structures**
- group of objects are treated as a single instance of an object
- clients do not need to use difference between compositions and individuals

- component – interface
- leaf – behavior for primitive objects, no children
- composite – stores child components
- client – manipulates objects using component



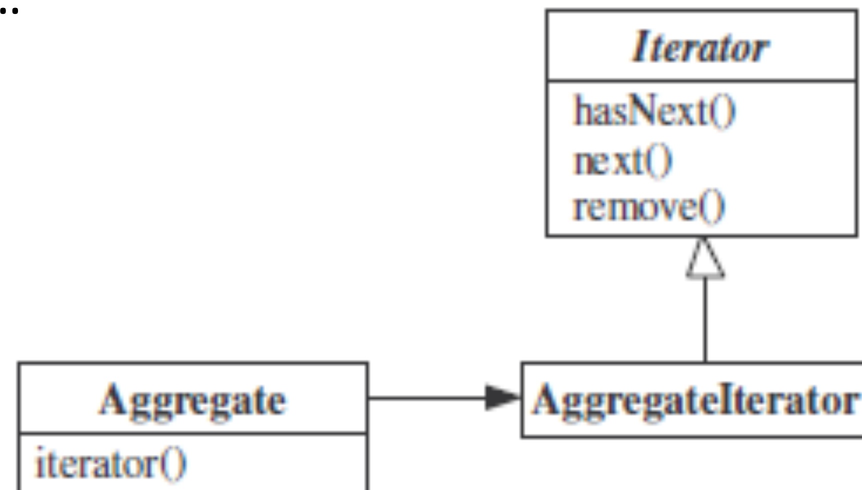
- Example – Graphics, arithmetic express.

Iterator



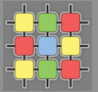
- » *behavioral* design pattern
- » iterator is used to **access elements of an aggregate object** (Collection)
- » sequential without exposing underlying implementation
- » very common in Java libraries (Aggregate – Iterable)
  - » List, HashSet, HashMap, Tree, …

```
Set items = new TreeSet();
...
Iterator seq = items.iterator();
while (seq.hasNext())
{
    Item item = (Item) seq.next();
    ...
    if (...) seq.remove();
}
```



- » Collection can be changed only through iterator -> if changed throws ConcurrentModificationException

Iterator



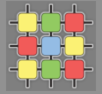
- » ListIterator - List.listIterator()
  - » hasPrevious(), previous()
  - » nextIndex(), previousIndex()
  - » add(...), set(…)

- » since Java 1.5

```
Set<Item> items = new TreeSet<Item>();
...
for (Item item : items)
{   ...
}
```

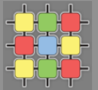
  - » works on arrays, anything implements Iterable

Abstract Factory



- *creational* design pattern
- extension of Factory pattern
- encapsulate a **group of individual factories having common theme**
  - multiple **factory methods**
- user creates concrete implementation of abstract factory and then uses generic interface to create concrete objects

Abstract Factory



AbstractFactory
+CreatProductA()
+CreatProductB()

Client

AbstractProductA

ConcreteFactory1
+CreatProductA()
+CreatProductB()

ConcreteFactory2
+CreatProductA()
+CreatProductB()

ProductA2

ProductA1

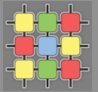
AbstractProductB

ProductB2

ProductB1

» Example: multiple look-and-feels in GUIs (e.g. Linux vs. Windows)

Command



- *behavioral* design pattern
- **encapsulate all information needed to call a method at a later time**

- client
  - instantiates the command
  - provides information required
- invoker
  - decides when to call the method
- receiver
  - contains method's code

- Example
  - Thread pools (java.lang.Runnable)
  - GUI Action