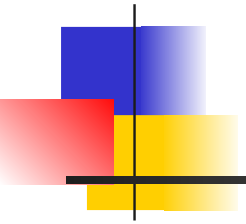


Design Patterns

James Brucker

Reusable Solutions

All engineering disciplines reuse proven good solutions

Civil Engineer

standard designs and construction methods based on experience

Circuit Designer

reuse component designs in integrated circuits

Architect

reuse design patterns in home and building design

Reusable Ideas in Software

Developers **reuse** knowledge, experience, & code

Application Level

reuse the design & code of a similar project

Design Level

apply known design principles and design patterns

Logic Level

apply known *algorithms* to implement behavior

Method Implementation (Coding) Level

use programming idioms for common tasks

A Programming Idiom

Problem: process every element of an array

Idiom:

1. initialize result
2. loop over the array
3. process each element of the array

```
// add the values of all the coupons we have sold...
Coupon [ ] coupons = ...
double total = 0;                    // initialize
for( int k=0; k<coupons.length; k++ ) {  // loop
     total += coupons[k].getTotal();  // process each one
}
```

An Algorithm

Problem:

find the shortest path from node A to node B in a graph

Solution:

apply Dykstra's Shortest Path algorithm

Reusable Code

Requirement:

sort a List of Persons by last name. Ignore case.

Solution:

Write a *Comparator* and use Collections.sort

```
List<Person> people = registry.getPeople( );
Comparator<Person> compByName = new Comparator<>() {
  public int compare(Person a, Person b) {
    return a.getLastname().compareToIgnoreCase(
                          b.getLastname());
  }
};
java.util.Collections.sort( people, compByName );
```

Reusable Code

Requirement:

keep a log of activity & events in a file,
so we have a record of what was done and any problems that occur.

Solution:

Use the open-source Log4J or slf4j framework.

```
public class Purse {
    private static final Logger log =
                    Logger.getLogger(Purse.class);
    public boolean insert(Money m) {
        if (m == null) log.error("argument is null");
        else log.info("inserting " + m);
```

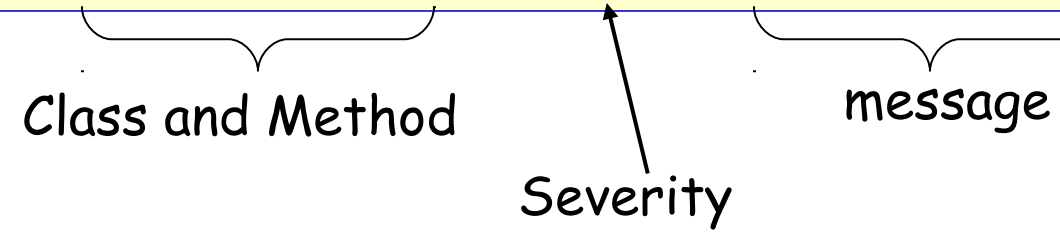
Logger Output

Log File:

You control *where logging is output*, and *how much detail is recorded*. Config file: log4j.properties.

Example:

```
6:02:27 Purse insert INFO inserting 10 Baht
6:03:00 Purse insert INFO inserting 20 Baht
6:03:10 Purse insert ERROR argument is null
6:03:14 Purse withdraw INFO withdraw 10 Baht
```



What is a Design Pattern?

A *situation* that occurs over and over, along with a *reusable* design of a solution.

Format for Describing a Pattern

Pattern Name: **Iterator**

Context

We need to access elements of a collection.

Motivation (Forces)

We want to access elements of a collection without the need to know the underlying structure of the collection.

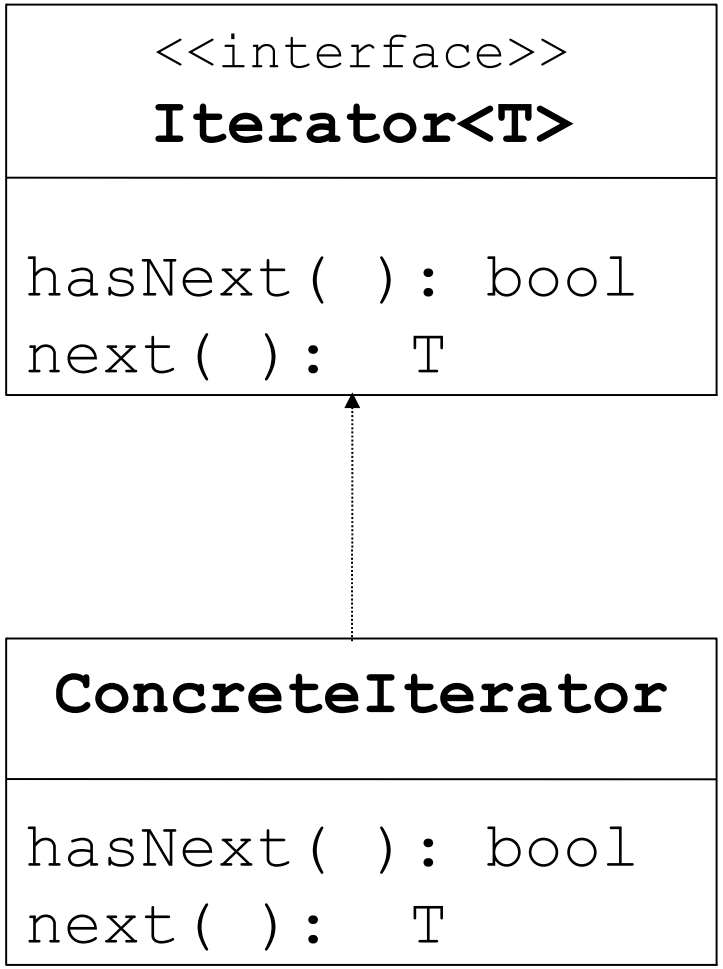
Solution

Each collection provides an iterator with methods to get the next element and check for more elements.

Consequences

Application is not coupled to the collection. Collection type can be changed w/o changing the application.

Diagram for Iterator



Examples of Iterator

What *Iterators* have you used?

How do you Get an Iterator?

Context:

We want to create an Iterator without knowing the class of the group of objects.

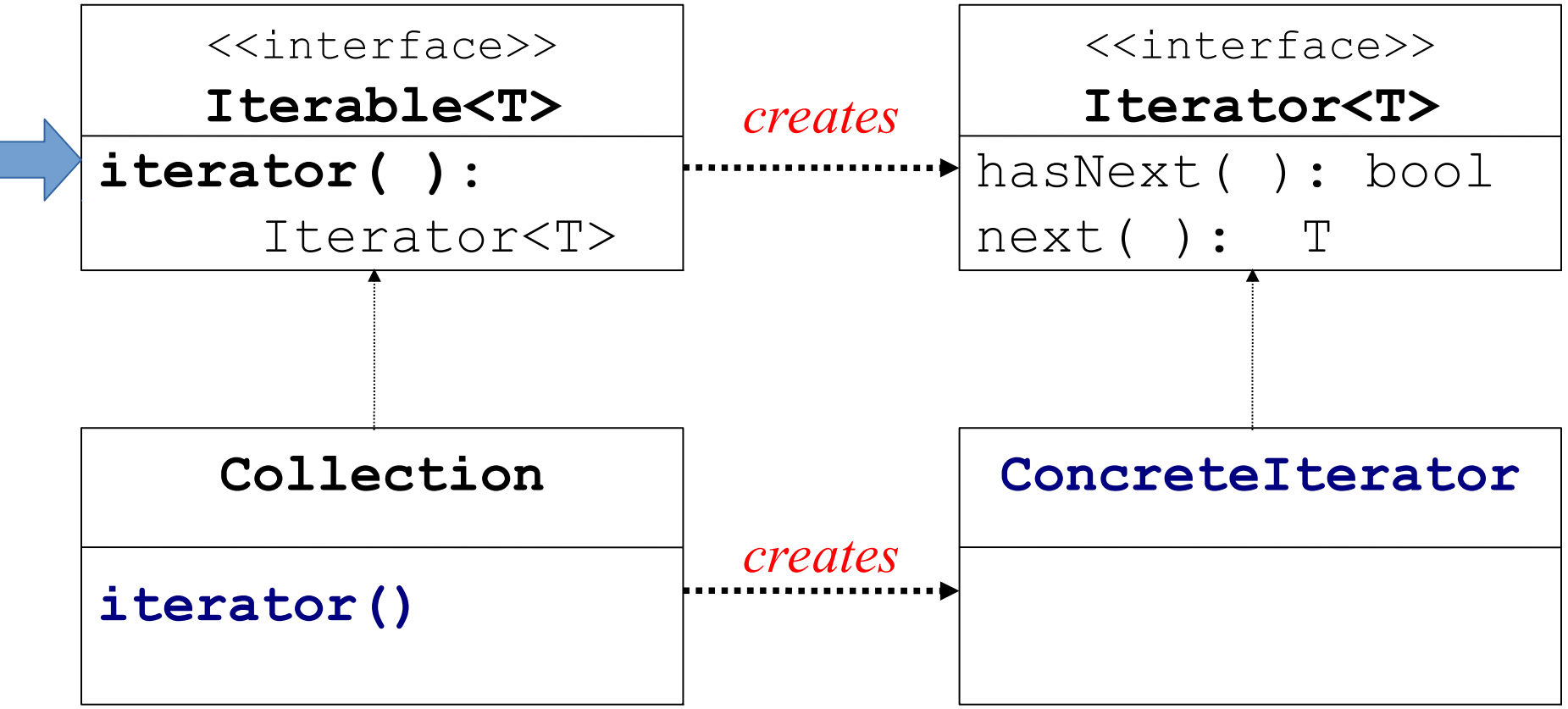
Forces:

We don't want the code to be coupled to a particular collection. We want to always create iterators in the *same way*.

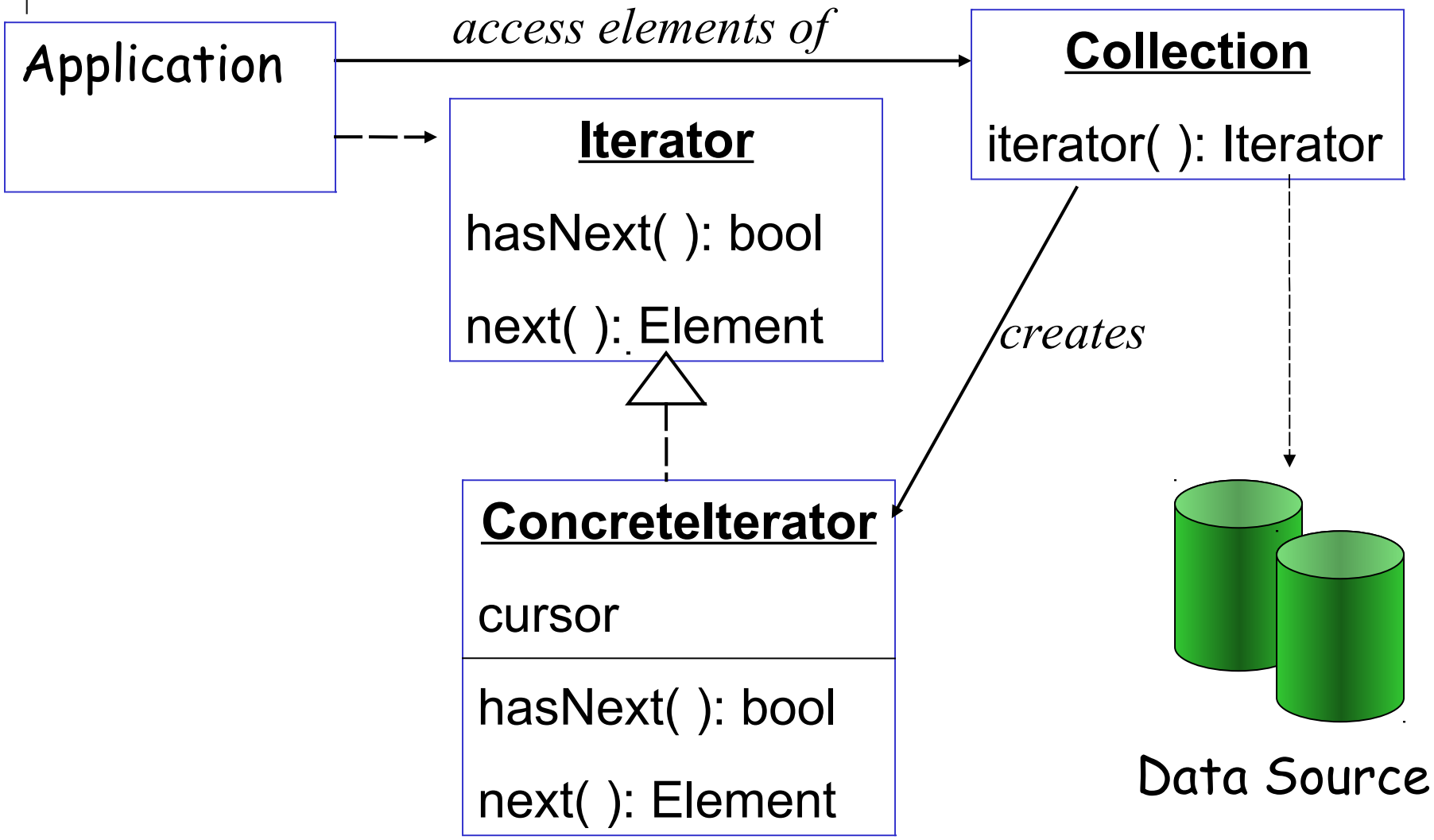
```
Collection<String> stuff = Foo.getElements();

Iterator<String> iterator = stuff.iterator();
```

Solution: Define a *Factory Method*

A *factory method* is a method that creates other objects.



Structure of Iterator Pattern



Example

```
List<String> list = new ArrayList<>( );
list.add( "apple" );
. . . // add more elements

Iterator<String> iter = list.iterator( );

while( iter.hasNext( ) )  {
    System.out.println( iter.next() );
}
```

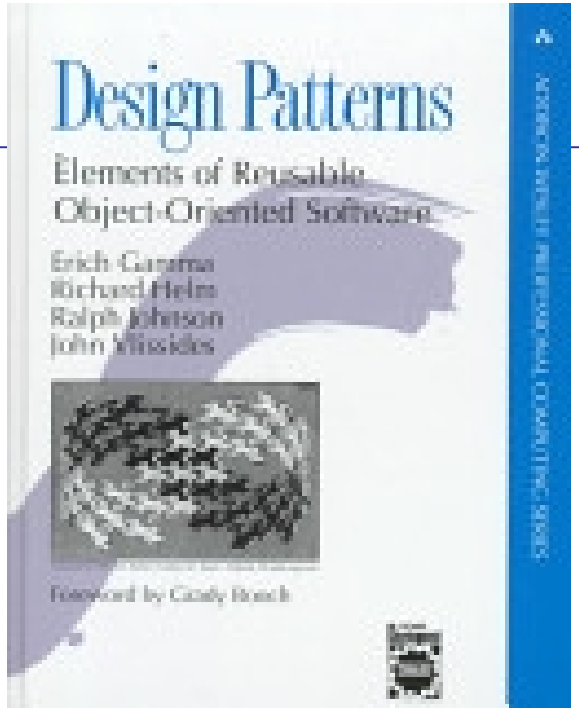
Design Patterns - Gang of Four book

The "Gang of Four"

The first book to popularize the idea of software patterns:

Gamma, Helm, Johnson, Vlissides

Design Patterns: Elements of Reusable Object-Oriented Software. (1995)

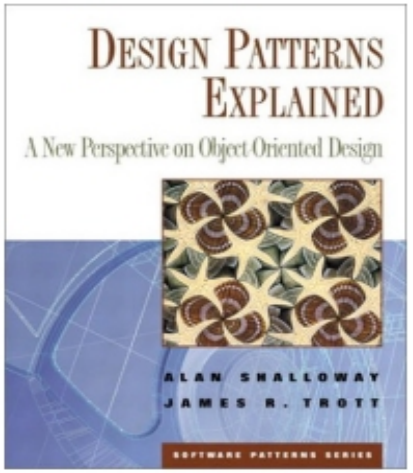


Good Design Patterns Books

Good for Java programmers

Design Patterns Explained, 2E (2004)
by Allan Shallow & James Trott

also wrote: *Pattern Oriented Design*.



Head First Design Patterns (2004)
by Eric & Elizabeth Freeman

Visual & memorable examples,
code is *too simple*.

Structure of Patterns in Gang of Four book

Name of Pattern

Intent

what the pattern does.

Motivation

Why this pattern. When to apply this pattern

Structure

Logical structure of the pattern. UML diagrams.

Participants and Collaborators

What are the elements of the pattern? What do they do?

Consequences

The benefits and disadvantages of using the pattern.

Iterator Pattern

Pattern Name: Iterator

Context

We need to access elements of a collection.

Motivation (Forces)

We want to use or view elements of a collection without the need to know the underlying structure of the collection.

Solution

Each collection provides an iterator with methods to check for more elements and get the next element.

Design Patterns To Know

1. Iterator
2. Adapter
3. Factory Method
4. Decorator
5. Singleton
6. Strategy - Layout Manager, used in a Container
7. State
8. Command
9. Observer
10. Facade

SKE Favorite Design Patterns

The SKE12 *Software Spec & Design* class were asked:

"*What patterns are most instructive or most useful?*"

SKE12 Favorite Patterns

Pattern	Votes
MVC	18
State	17
Factory Method	16
Command	15
Strategy	15
Facade	12
Singleton	12
Iterator	11
Observer	11
Adapter	8
Decorator	4
Template Method	3

Categories of Patterns

Creational - how to create objects

Structural - relationships between objects

Behavioral - how to implement some behavior

Situations (Context) not Patterns

Learn the **situation** and the **motivation** (forces) that motivate the solution.

Pay attention to **Applicability** for details of context where the pattern applies.
(Avoid applying the wrong pattern.)

Adding New Behavior

Situation:

we want to add some new behavior to an existing class

Forces:

1. don't want to add more responsibility to the class
2. the behavior may apply to similar classes, too

Example:

Scrollbars

Changing the Interface

Situation:

we want to use a class in an application that requires interface A. But the class doesn't implement A.

Forces:

1. not appropriate to modify the existing class for the new application
2. we may have many classes we need to modify

Example:

change an Enumeration to look like an Iterator

Convenient Implementation

Situation:

some interfaces require implementing a *lot* of methods. But most of the methods aren't usually required.

Forces:

1. how can we make it *easier to implement interface?*
2. how to supply default implementations for methods?

Example:

MouseListener (6 methods), List (24 methods)

A Group of Objects act as One

Situation:

we want to be able to use a Group of objects in an application, and

the application can treat the whole group like a single object.

Forces:

There are many objects that behave similarly. To avoid complex code we'd like to treat as one object.

Example:

KeyPad in a mobile phone app.

Creating Objects without Knowing Type

Situation:

we are using a framework like OCSF.

the framework needs to create objects.

how can we change the type of object that the framework creates?

Forces:

1. want the framework to be extensible.

2. using "new" means coupling between the class and the framework.

Example:

JDBC (Java Database Connection) creates connections for different kinds of databases.

Do Something *Later*

Situation:

we want to run a task at a given time (in the future)

Forces:

we don't want our "task" to be responsible for the schedule of when it gets run.

This situation occurs a lot, so we need a reusable solution.

Example:

We're writing a digital clock. We want an alarm to sound at a specified time.