Design Patterns

The Timeless Way of Coding

Designed and Presented by

Dr. Heinz Kabutz

Illustrations by Edith Sher

Dr Heinz Kabutz

- Born and bred in fishing village Cape Town
- Professional Java programmer since 1997
  - Worked on many large Java systems
  - Trained hundreds of programmers in Java and Design Patterns
- PhD in Computer Science from the University of Cape Town
  - Focused on performance analysis of distributed communicating systems

Maximum Solutions (Pty) Ltd

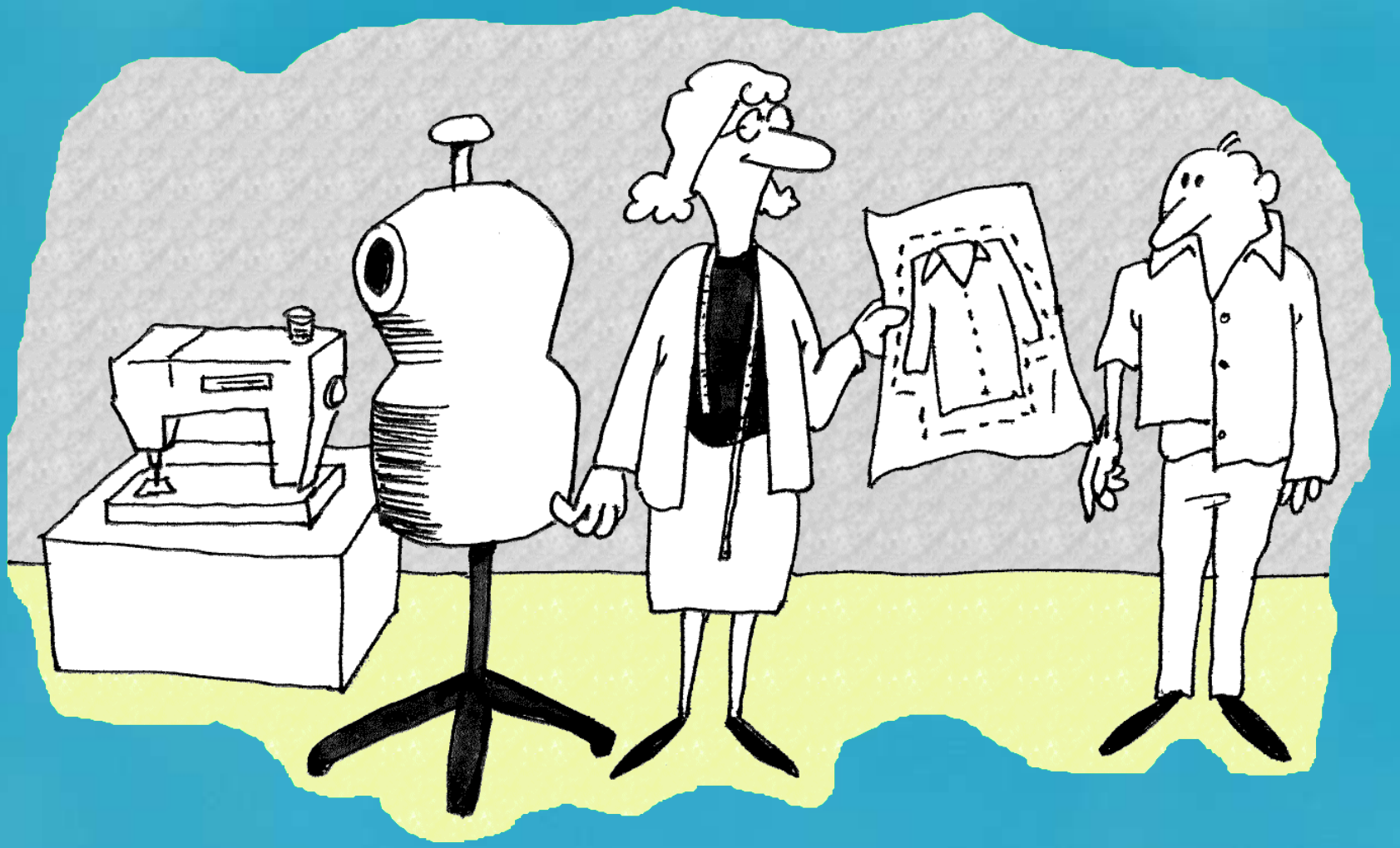
The Java Specialists

- Founded in 1998 by Heinz & Helene Kabutz
- A South African company
  - Active in South Africa, Germany, Austria, UK, Mauritius, China, Estonia, Switzerland
- Company has four interacting energies:
  - Software Development
  - Specialist Training
    - Maximum one week a month
  - Consulting
  - Research
    - The Java™ Specialists' Newsletter

The Java Specialists' Newsletter

- Advanced free publication written specifically **for** Java Specialists
- Only publication of its kind in the world
- Translated into 10 languages (incl Zulu)
- Produced in South Africa
  - Something that Africa can be proud of
- Currently read in 108 countries
- Over 10000 regular readers
- http://www.javaspecialists.co.za

1: Introduction to Patterns



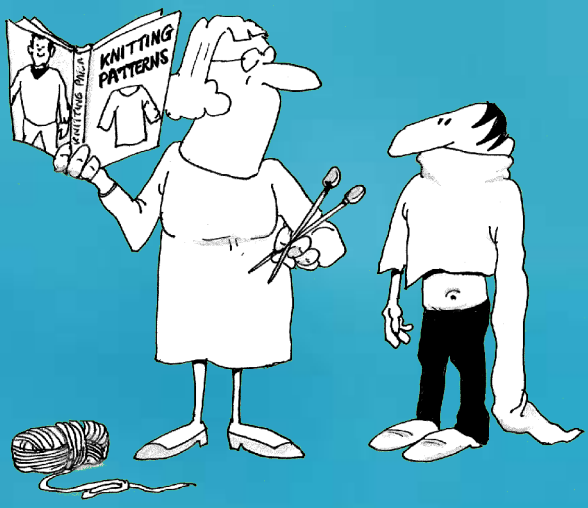
Structure of Talk

- Introduction to Design Patterns
- The Singleton
  - Why your developers like it
- The Adapter
  - Which to use when

Questions

- Please please please please ask questions!
- There *are* some stupid questions
  - They are the ones you didn't ask
  - Once you've asked them, they are not stupid anymore
- Assume that if you didn't understand something that it was my fault
- The more you ask, the more everyone learns (including me)

Learning Patterns



- Design Patterns are for programmers and developers
  - NOT analysts and architects!
- Improves programmer communication
- Broad-based Patterns Educational Empowerment (BBPEE)
- Courses & Study Groups
  - Courses short and sweet
  - Led by people already *in the know*
  - Best approach: Internal courses

Why are learning patterns?

- Manager forced you

- Want to become better OO programmer
- Fascination with Patterns
- Free breakfast?

Vintage Wines

- Design Patterns are like good red wine
  - You cannot appreciate them at first
  - As you study them you learn the difference between plonk and vintage
  - As you become a connoisseur you experience the various textures you didn't notice before
- Warning: Once you are hooked, you will no longer be satisfied with plonk!

Why are patterns so important?

- Provide a view into the brains of OO experts
- Help you understand existing designs
- *Patterns in Java*, Volume 1, Mark Grand writes
  - "What makes a bright, experienced programmer much more productive than a bright, but inexperienced, programmer is experience."

Coding Patterns

- We have all seen patterns in code:
  - `for (int i=0; i<names.length; i++) ...`
  - common data structures, like linked list
- This is the way we "do things"
- University teaches us to code, to theorise, but not to design
  - Lecturers often don't have enough real-world experience
- Design is normally learnt through experience
  - At the expense of the employer!

Introduction

- To begin learning Design Patterns, you need the basics:
  - Abstraction
  - Encapsulation
  - Modularity
  - Hierarchy (Composition and Inheritance)
- Should be able to follow basic UML class diagrams

Design Patterns Origin

The Timeless Way of Building

Christopher Alexander

There is a central quality which is the root criterion of life and spirit in a man, a town, a building, or a wilderness.

If you want to make a living flower, you don't build it physically, with tweezers, cell by cell. You grow it from the seed.

What's in a name?

The Timeless Way of Building

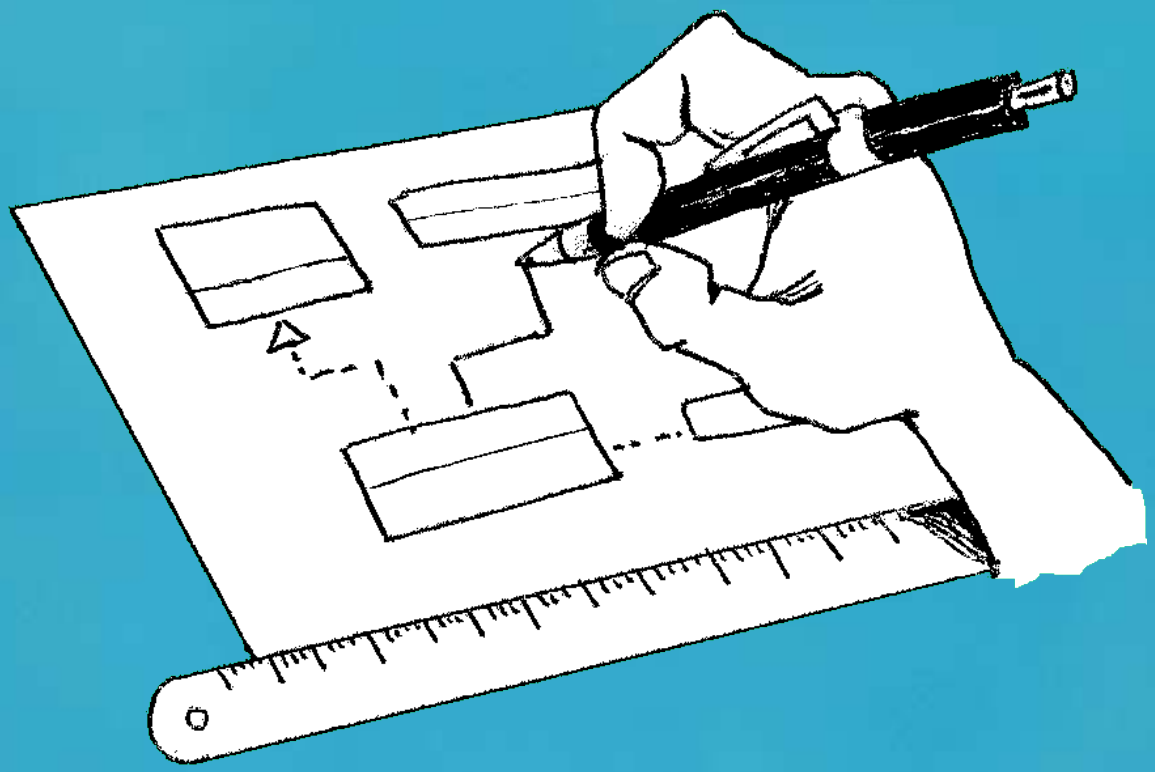
The search for a name is a fundamental part of the process of inventing or discovering a pattern.

So long as a pattern has a weak name, it means that it is not a clear concept, and you cannot tell me to make "one".

Why do we need a diagram?

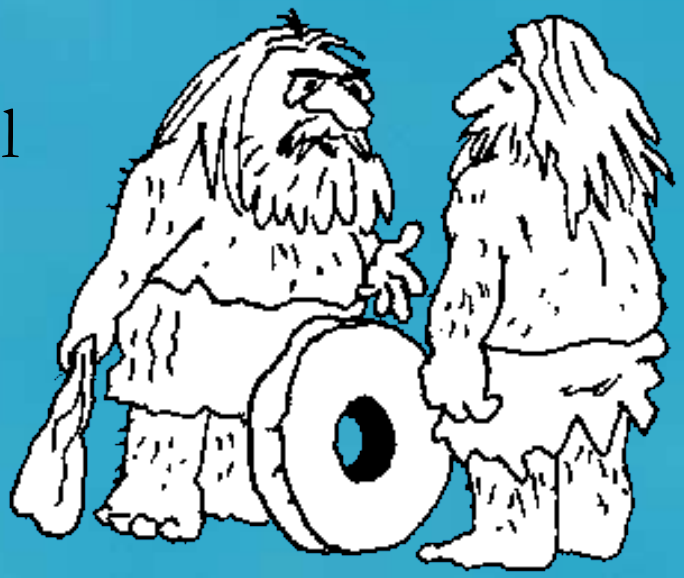
The Timeless Way of Building

If you can't draw a [class] diagram of it, it isn't a pattern



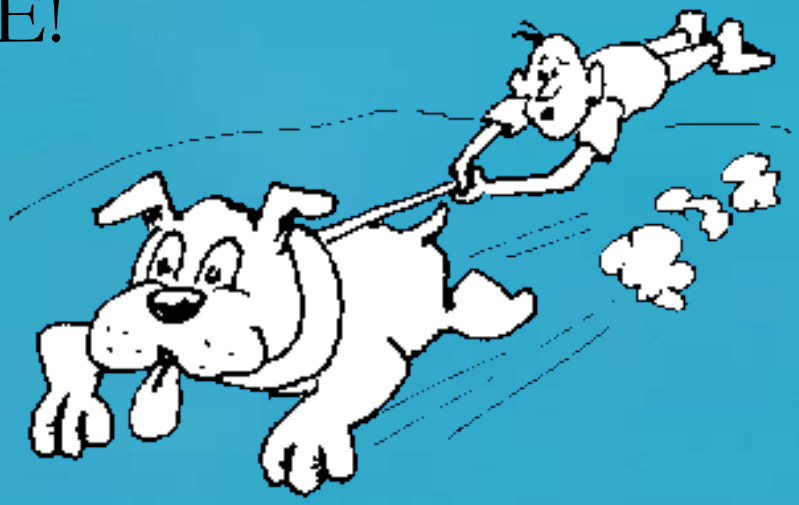
Misuse of Design Patterns

- Patterns Misapplied
  - "design" patterns should not be used during analysis
- Cookie Cutter Patterns
  - patterns are generalised solutions
- Misuse By Omission
  - reinventing a crooked wheel

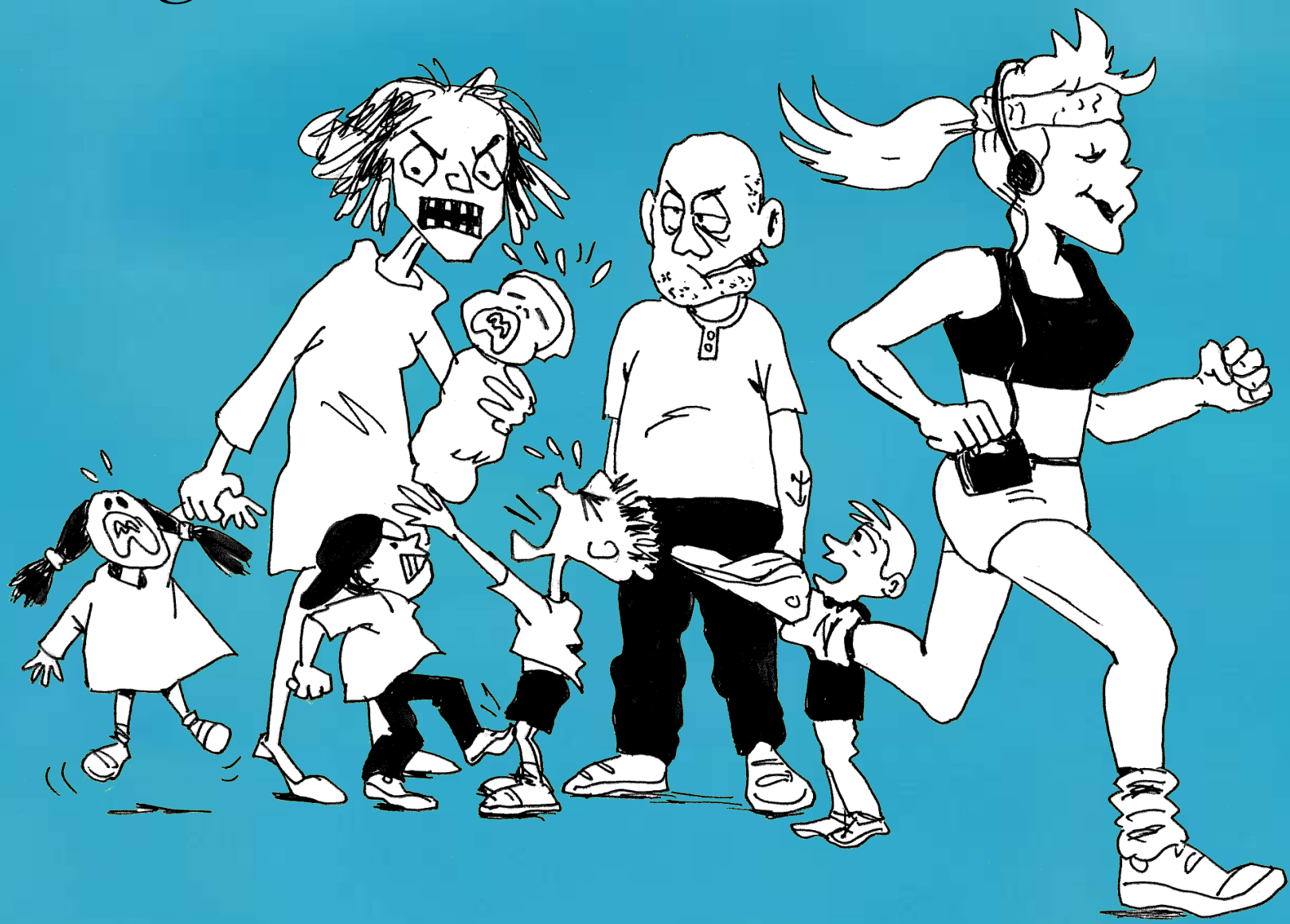


Summary

- Object Orientation is here to stay
- Design Patterns will fast-track you in learning how to design with objects
- Remember: BBPEE!



Singleton

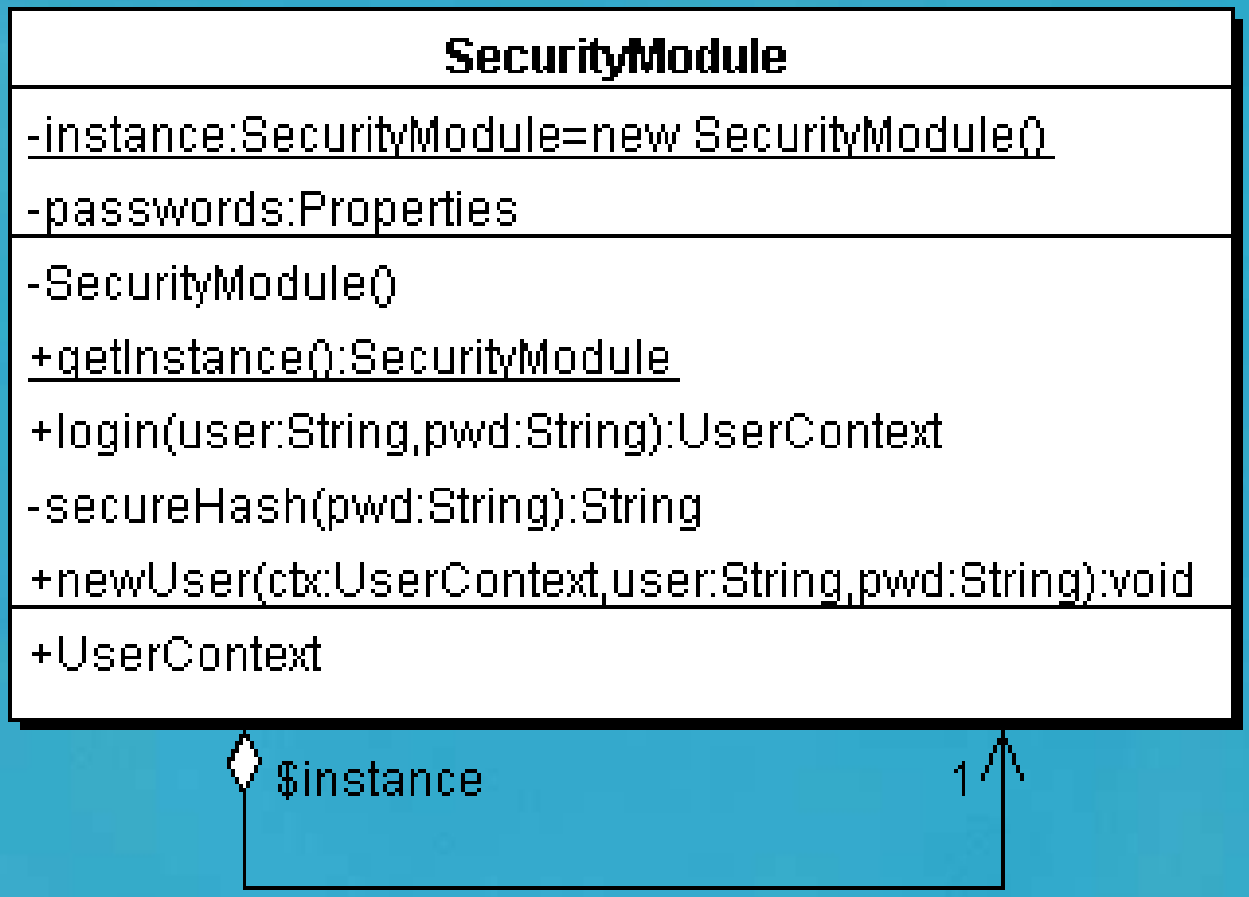


Singleton

- Intent
  - Ensure a class only has one instance, and provide a global point of access to it.

Motivation: Singleton

- It's important for some classes to have exactly **one** instance, e.g. SecurityModule



Sample Code: Singleton

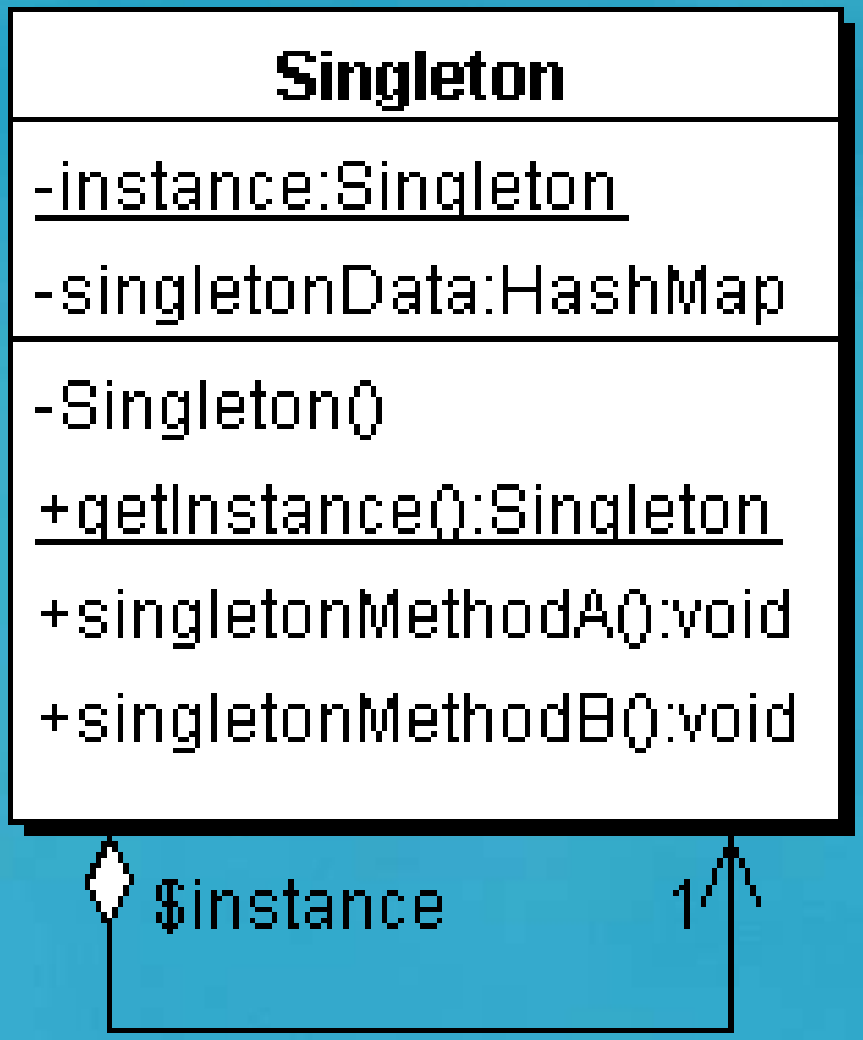
```
public class SecurityModule {
  private static SecurityModule instance =
    new SecurityModule();
  public static SecurityModule getInstance() {
    return instance;
  }
  private SecurityModule() {
    loadPasswords();
  }
  public UserContext login(String username,
      String password) {
    return new UserContext(username, password);
  }

  // etc.
```

Applicability: Singleton

- Use the Singleton pattern when
  - there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
  - when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Structure: Singleton



Consequences: Singleton

- Benefits
  - Controlled access to sole instance
  - Reduced name space
  - Permits refinement of operations and representation
  - Permits a variable number of instances
  - More flexible than class operations
- Drawbacks
  - Overuse can make a system less OO.

Known Uses in Java: Singleton

- java.lang.Runtime.getRuntime()
- java.awt.Toolkit.getDefaultToolkit()

Questions: Singleton

- The pattern for Singleton uses a private constructor, thus preventing extendability. What issues should you consider if you want to make the Singleton "polymorphic"?
- Sometimes a Singleton needs to be set up with certain data, such as filename, database URL, etc. How would you do this, and what are the issues involved?

Exercises: Singleton

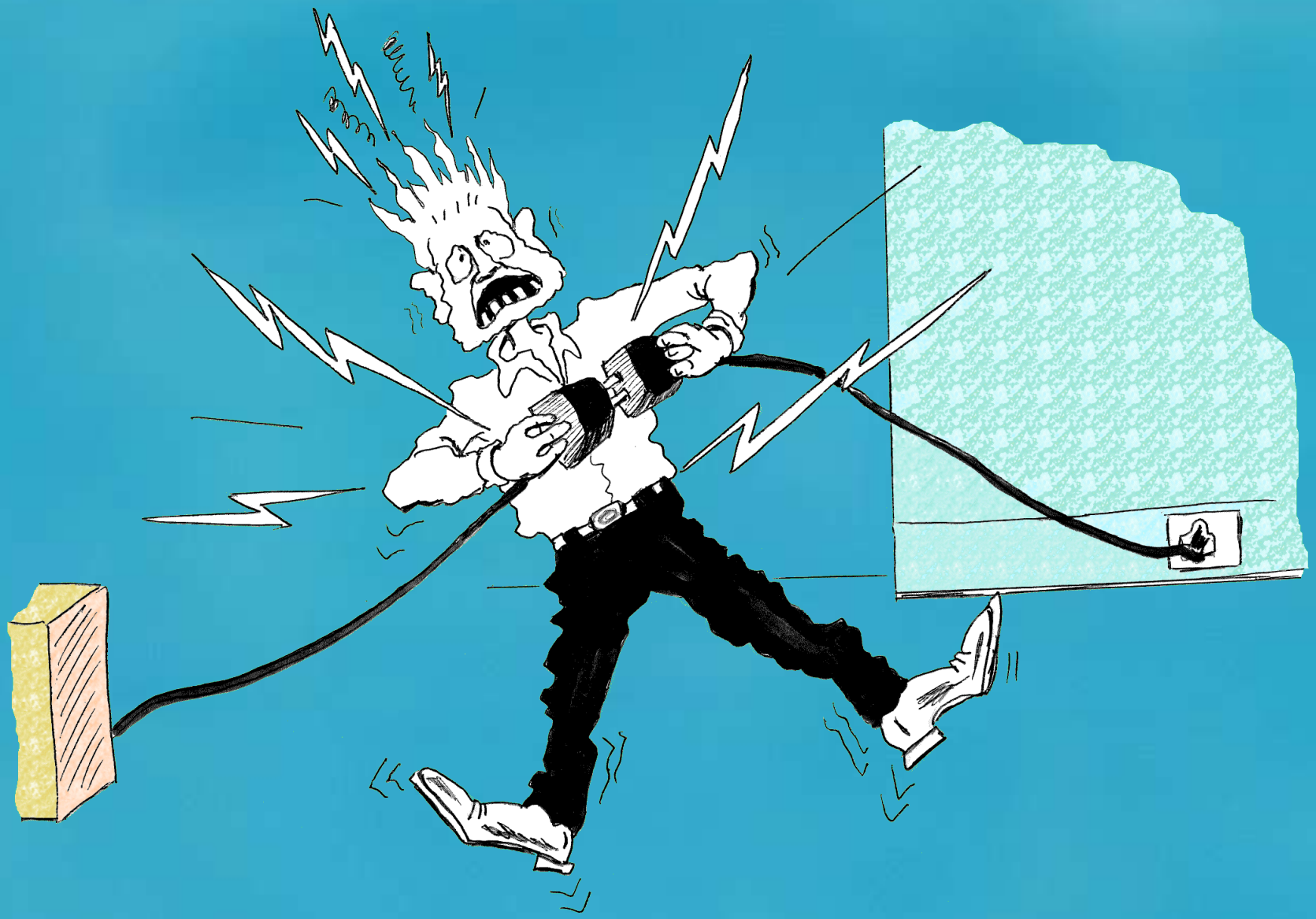
- Turn the following class into a Singleton:

```
public class Earth {
  public static void spin() {}
  public static void warmUp() {}
}

public class EarthTest {
  public static void main(String[] args) {
    Earth.spin();
    Earth.warmUp();
  }
}
```

- Now change it to be extendible

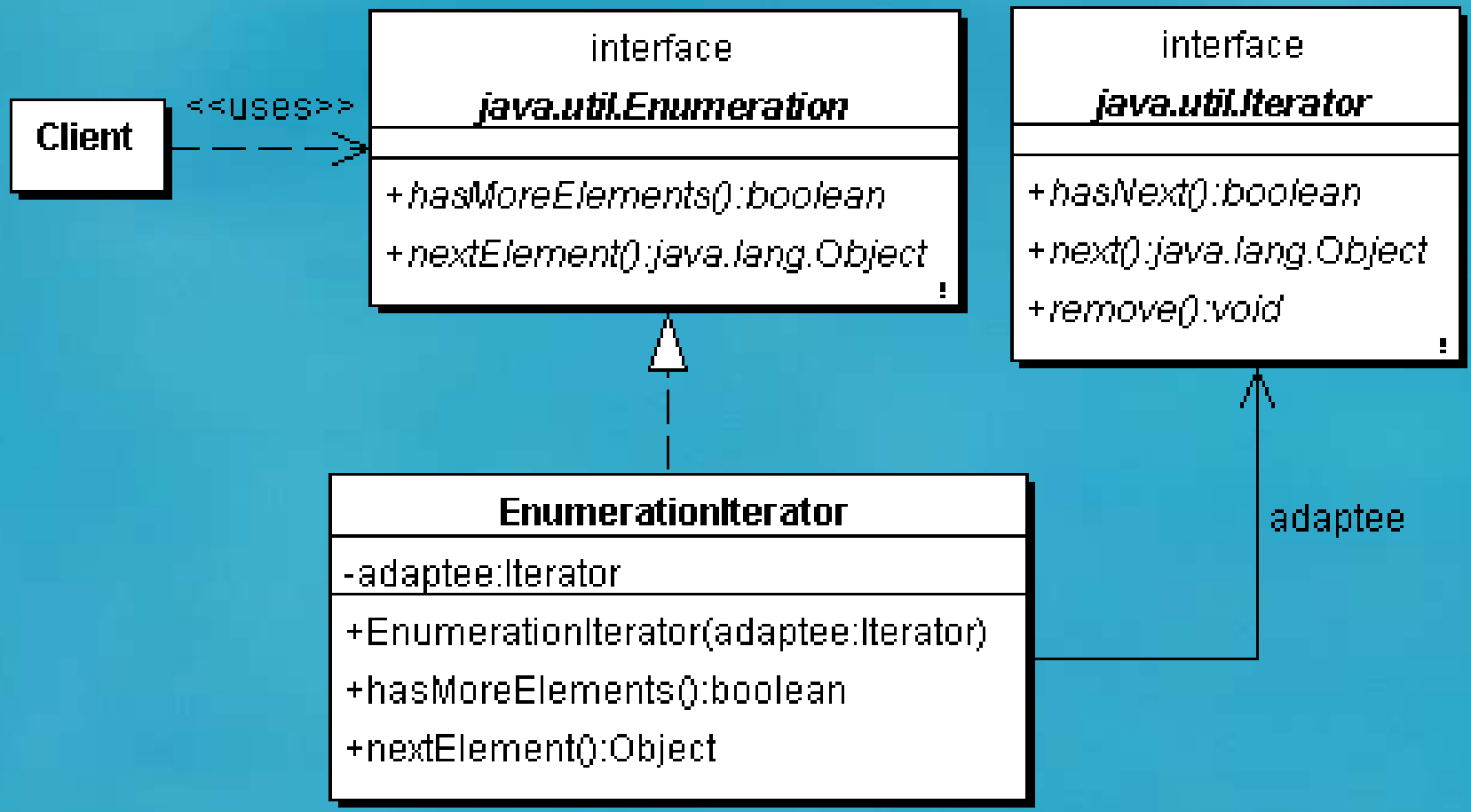
Adapter



Adapter

- Intent
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Also known as
  - Wrapper

Motivation: Adapter



- Convert an Iterator to an Enumeration

```
import java.util.Enumeration;

/** @since JDK 1.0 */
public class Printer {
  public static void print(Enumeration e) {
    System.out.println(
      "Enumeration {");
    while (e.hasMoreElements()) {
      System.out.print("  " + e.nextElement());
      if (e.hasMoreElements())
        System.out.println(",");
    }
    System.out.println("}");
  }
}
```

```
import java.util.*;

/** Adapter converts Iterator to Enumeration */
public class EnumerationIterator implements
    Enumeration {
  private final Iterator adaptee;
  public EnumerationIterator(Iterator adaptee) {
    this.adaptee = adaptee;
  }
  public boolean hasMoreElements() {
    return adaptee.hasNext();
  }
  public Object nextElement() {
    return adaptee.next();
  }
}
```

```
import java.util.*;
public class PrinterTest {
  public static void main(String[] args) {
    Vector old_collection = new Vector();
    for (char c = 'A'; c < 'M'; c++) {
      old_collection.addElement("" + c);
    }
    Printer.print(old_collection.elements());
    String[] names = {
      "Erich", "Richard", "Ralph", "John" };
    List new_collection = Arrays.asList(names);
    Enumeration en = new EnumerationIterator(
      new_collection.iterator());
    Printer.print(en);
  }
}
```

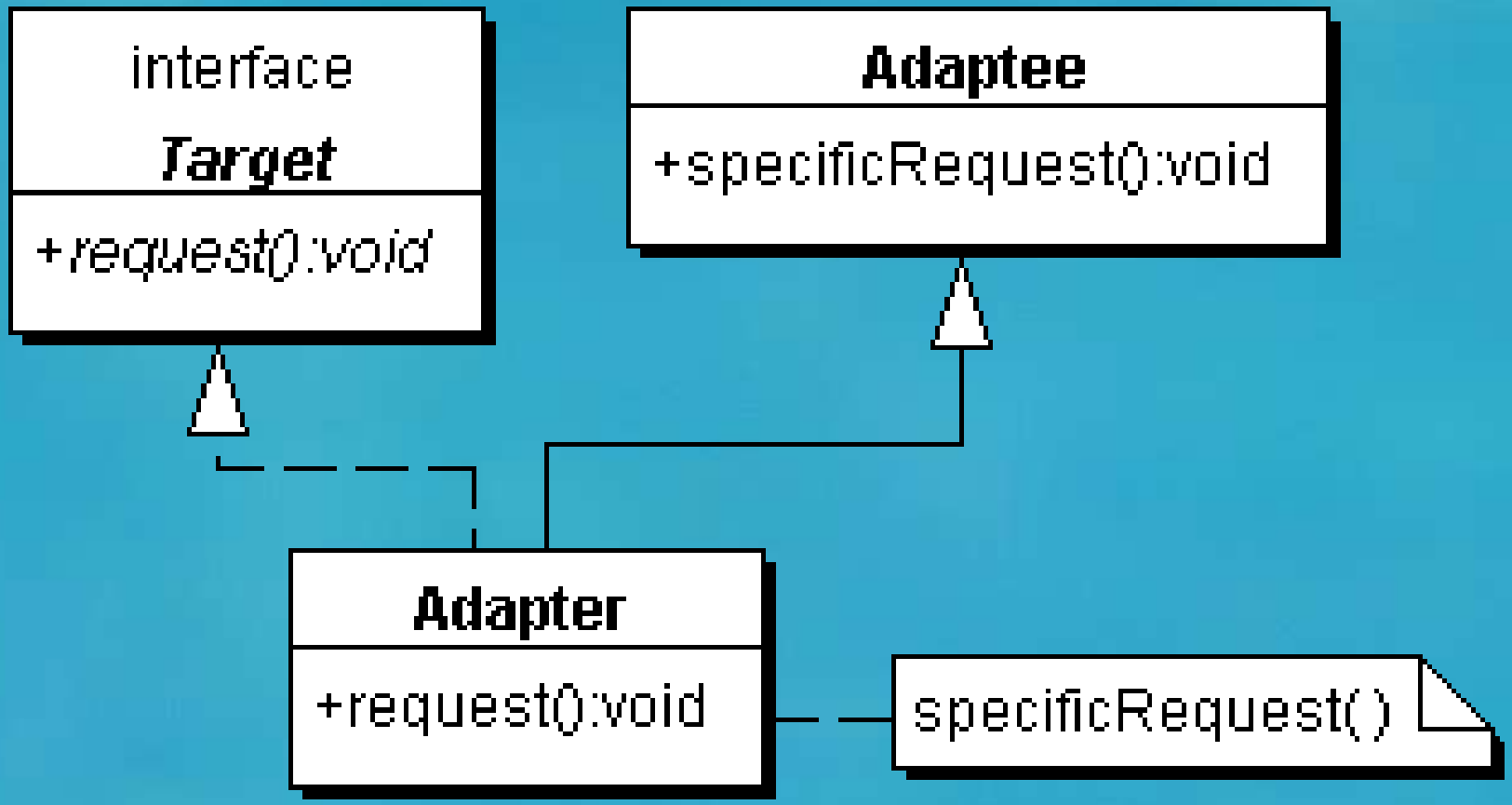
> java PrinterTest ↵

```
Enumeration {
   A,
   B,
   C,
   D,
   E,
   F,
   G,
   H,
   I,
   J,
   K,
   L}
Enumeration {
   Erich,
   Richard,
   Ralph,
```

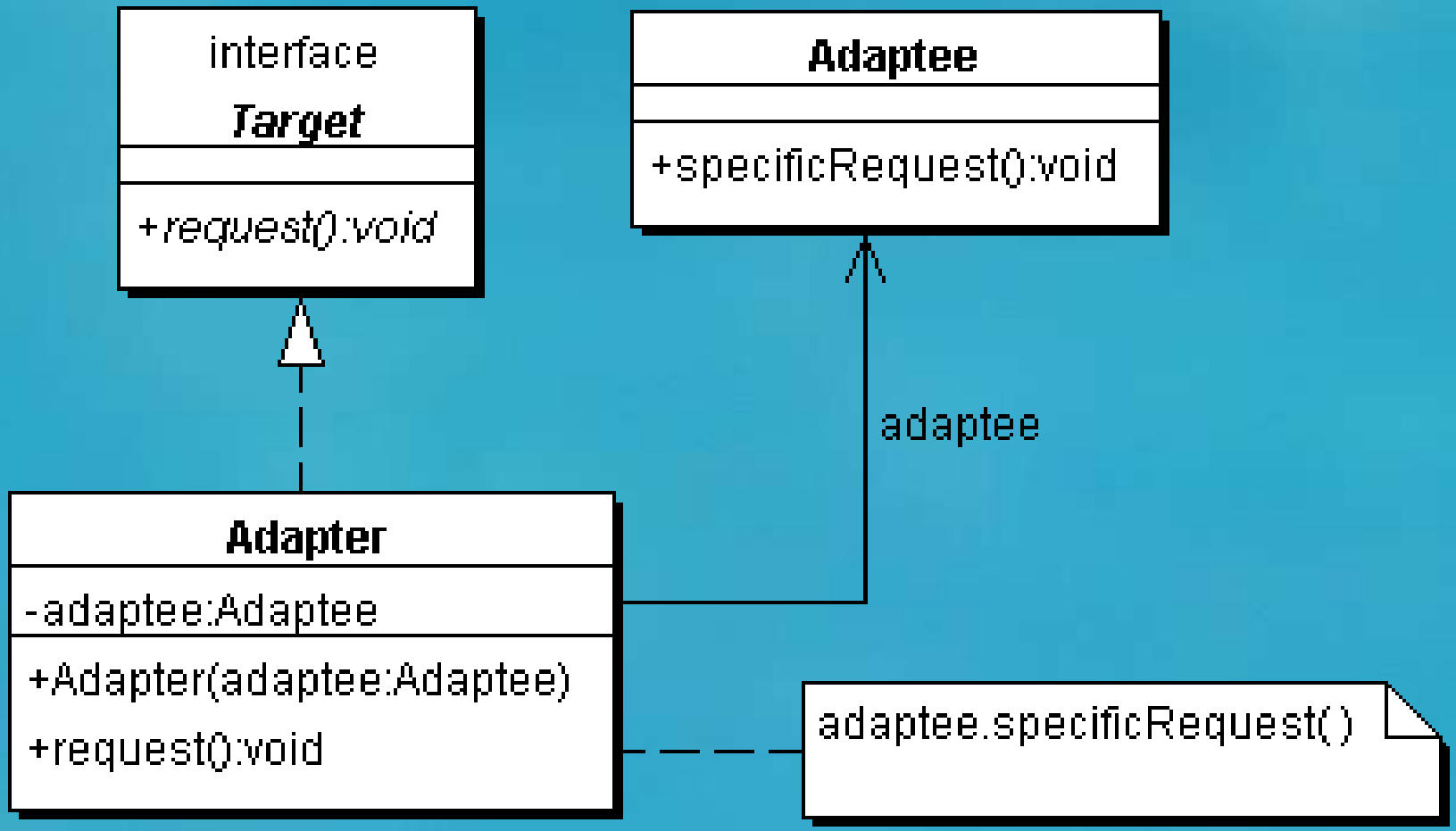
Applicability: Adapter

- Use the Adapter pattern when
  - some existing class does not match the interface you need
  - you need to use several existing subclasses, but you don't want to subclass each one

Structure: Class Adapter



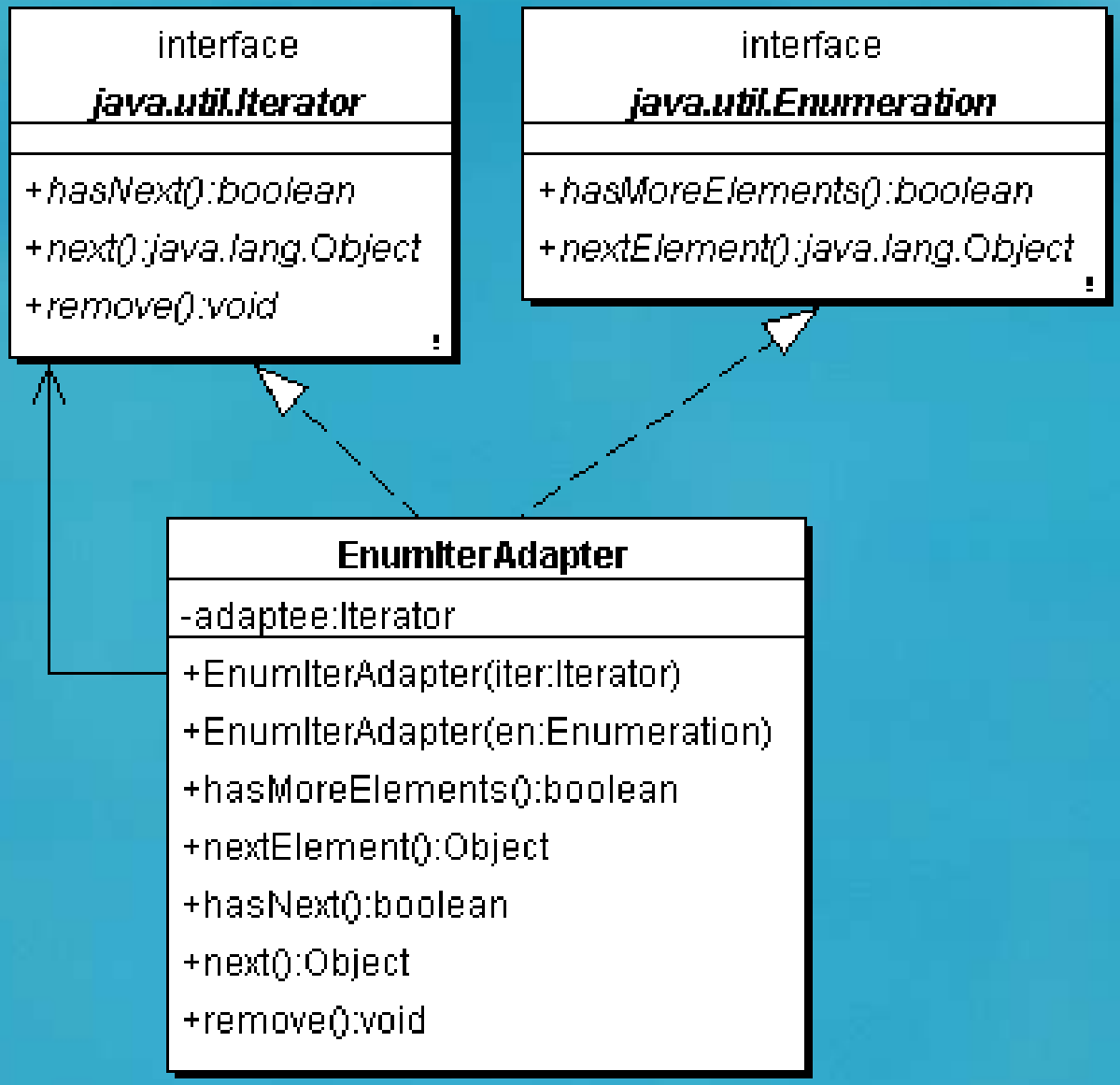
Structure: Object Adapter



Consequences: Adapter

- Class adapter
  - won't work when we want to adapt a class *and* all its subclasses
  - lets Adapter override some of Adaptee's methods
- Object adapter
  - single Adaptor can work with many Adaptees
  - makes it harder to override Adaptee behaviour

Two-way Adapter



```
import java.util.*;
public class EnumIterAdapter
    implements Enumeration, Iterator {
  private final Iterator adaptee;
  public EnumIterAdapter(Iterator iter) {
    adaptee = iter;
  }
  public EnumIterAdapter(final Enumeration en) {
    this(new Iterator() {
      public boolean hasNext() {
        return en.hasMoreElements();
      }
      public Object next() {
        return en.nextElement();
      }
      public void remove() {
        throw new UnsupportedOperationException();
      }
    });
  }
```

```
  public boolean hasMoreElements() {
    return adaptee.hasNext();
  }
  public Object nextElement() {
    return adaptee.next();
  }
  public boolean hasNext() {
    return adaptee.hasNext();
  }
  public Object next() {
    return adaptee.next();
  }
  public void remove() {
    adaptee.remove();
  }
}
```

Known Uses in Java: Adapter

- The **`java.io.InputStreamReader`** adapts **`java.io.InputStream`** to have the correct **`java.io.Reader`** interface
- The **`java.awt.MouseAdapter`** adapts **`java.awt.MouseListener`** without changing the interface.

Questions: Adapter

- What are the structural differences between an Adapter and a Proxy?
- Under what circumstances are they interchangeable?
- Java uses a **`MouseAdapter`** class to implement the **`MouseListener`** interface and to provide default operations. What type of Adapter is this?

Exercises: Adapter

- Consider the following Singer interface:

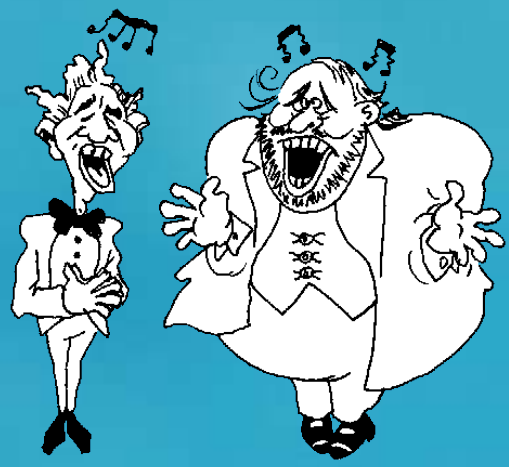
```
public interface Singer {
  void sing();
}
```

- It is used as follows

```
public class MusicFest {
  private final List singers = new LinkedList();
  public void addSinger(Singer singer) {
    singers.add(singer);
  }
  public void singAll() {
    Iterator it = singers.iterator();
    while(it.hasNext())
      ((Singer)it.next()).sing();
  }
}
```

- Now consider the Rapper class:

```
public class Rapper {
  public void talk() {
    System.out.println(
      "Vulgar lyrics deleted ...");
  }
}
```



- Now write a RapperAdapter class so that the MusicFestTest runs:

```
public class MusicFestTest {
  public static void main(String[] args) {
    MusicFest fest = new MusicFest();
    fest.addSinger(new Bass());
    fest.addSinger(new Soprano());
    fest.addSinger(new RapperAdapter());
    fest.singAll();
  }
}
```

Conclusion to Design Patterns

- Programmers become more effective when designing with patterns
- Knowing the basic patterns helps you understand new patterns easily
- New patterns discovered all the time:
  - http://www.hillside.net for *all sorts of* patterns
  - http://www.javasoft.com for J2EE patterns
- And remember, BBPEE !
  - (for those with a short memory, that is Broad-Based Patterns Educational Empowerment ☺)

End of Design Patterns Talk

- Thank you for attending this talk ☺
- Please encourage your developers to learn patterns
- Please contact me for further information about Design Patterns Courses:

  heinz@javaspecialists.co.za
- You should subscribe to **The Java™ Specialists' Newsletter** on:

  http://www.javaspecialists.co.za