

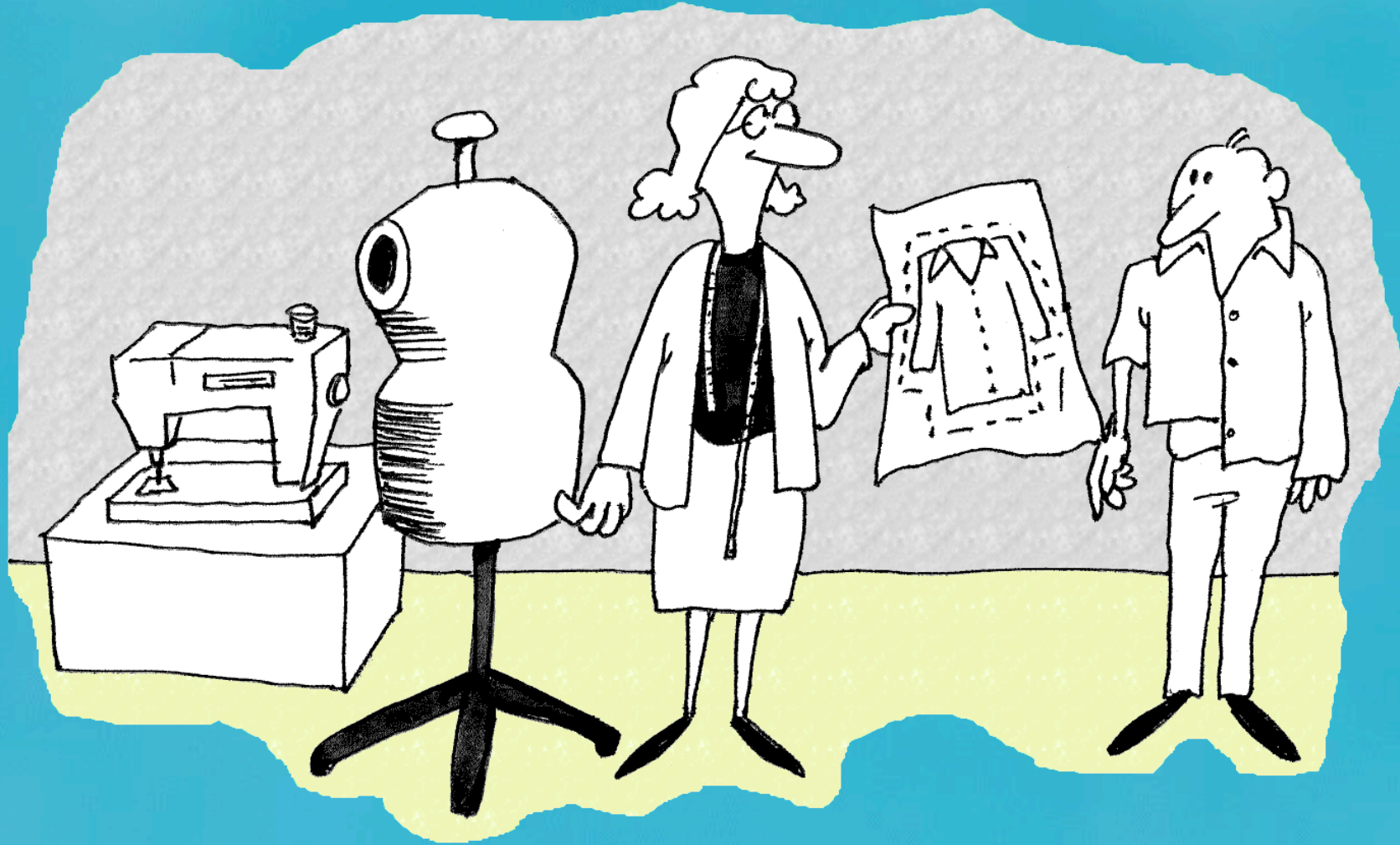
Design Patterns

The Timeless Way of Coding

Designed and Presented by
Dr. Heinz Kabutz

Illustrations by Edith Sher

1: Introduction to Patterns



Who am I?

- YAJP
- Trainer of Java and Design Patterns Courses in various places of the world
- PhD in Computer Science from UCT
- Java Consultant
- Publish Java newsletter “Made in South Africa” that is reaching 97 countries
 - This raises South Africa’s technological image

Your Expectancies

- Don't expect
 - Polished speaker, wearing a suit
 - Java Guru (“The leading Java guy in the country” ;-)
 - Salesman
- Do expect
 - Someone who is passionate about programming and the problems programmers face
 - Someone who prefers technology to meetings
 - Someone who loves using good Object-Oriented principles, and who is committed to teaching others how to do the same.

My Hopes for This Talk

- To find out which JUG is friendlier: Cape Town or Durban?
- To make new contacts in Durban
- To get you addicted to Design Patterns
 - Drug dealer scum have good business strategy
- To raise awareness of how Maximum Solutions can help you:
 - Training (Java and Design Patterns)
 - Consulting / Mentoring / Coaching
 - Programming

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)

What is a Design Pattern?

- A design idea that has been applied many times, with success
- Designs that result in reusable code
- In our case, we will look at Object Oriented Design Patterns

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”
- Most courses teach the syntax of a language, not the semantics
- Design is normally learnt through experience

Introduction

- For this talk I assume you have a good understanding of the basic OO concepts of encapsulation, abstraction, composition and inheritance
- Should be able to follow basic UML class diagrams
- Design Patterns is the recommended text; additional references are shown where applicable

Textbook – “Design Patterns”

- “Design Patterns” book by Gang of Four (GoF)
- Contains a collection of basic “patterns” that experienced OO developers use regularly
- Cannot proceed very far in Java / C++ / VB.NET without understanding patterns
- Facilitates better communication
- Based on work of renegade architect Christopher Alexander in “The Timeless Way of Building”



Pattern Structure

- Classic
 - Intent
 - Also Known As
 - Motivation
 - Applicability
 - Structure
 - Participants
 - Collaborations
 - Consequences
 - Implementation
 - Sample Code
 - Known Uses
 - Related Patterns
- This Course
 - Intent
 - Also Known As
 - Motivation
 - Sample Code
 - Applicability
 - Structure
 - Consequences
 - Known Uses In Java
- The other sections are left for self-study

What's in a name?

The Timeless Way of Building

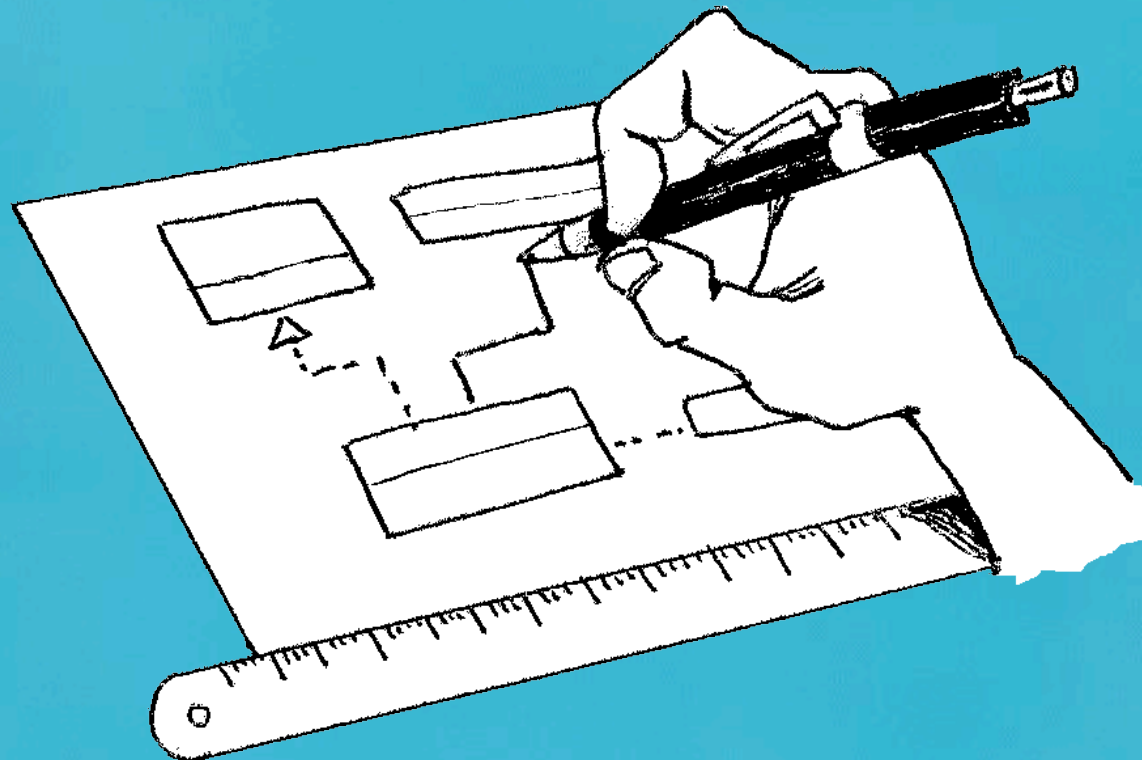
The search for a name is a fundamental part of the process of inventing or discovering a pattern.

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?

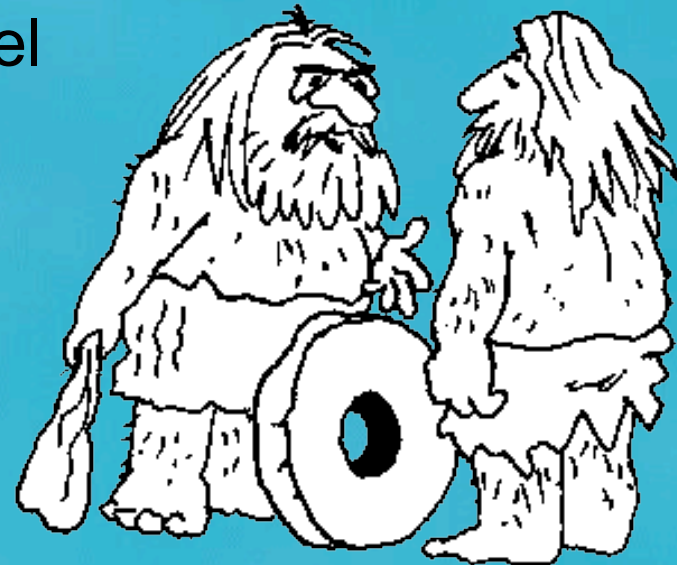
The Timeless Way of Building

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



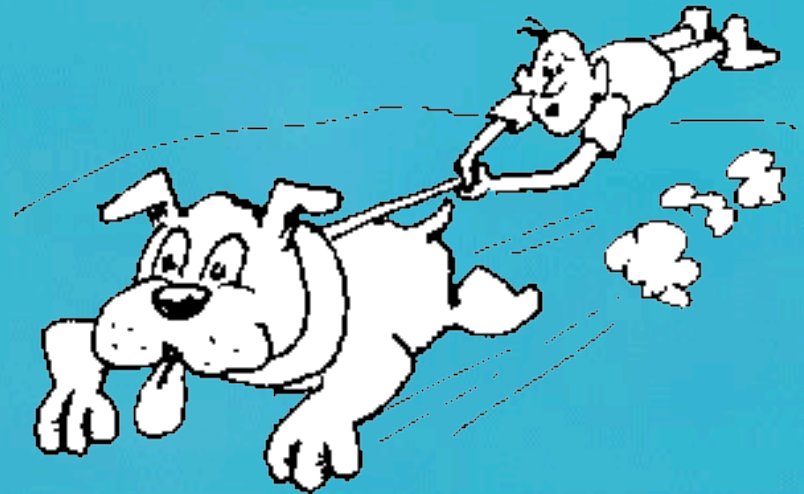
Misuse of Design Patterns

- 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

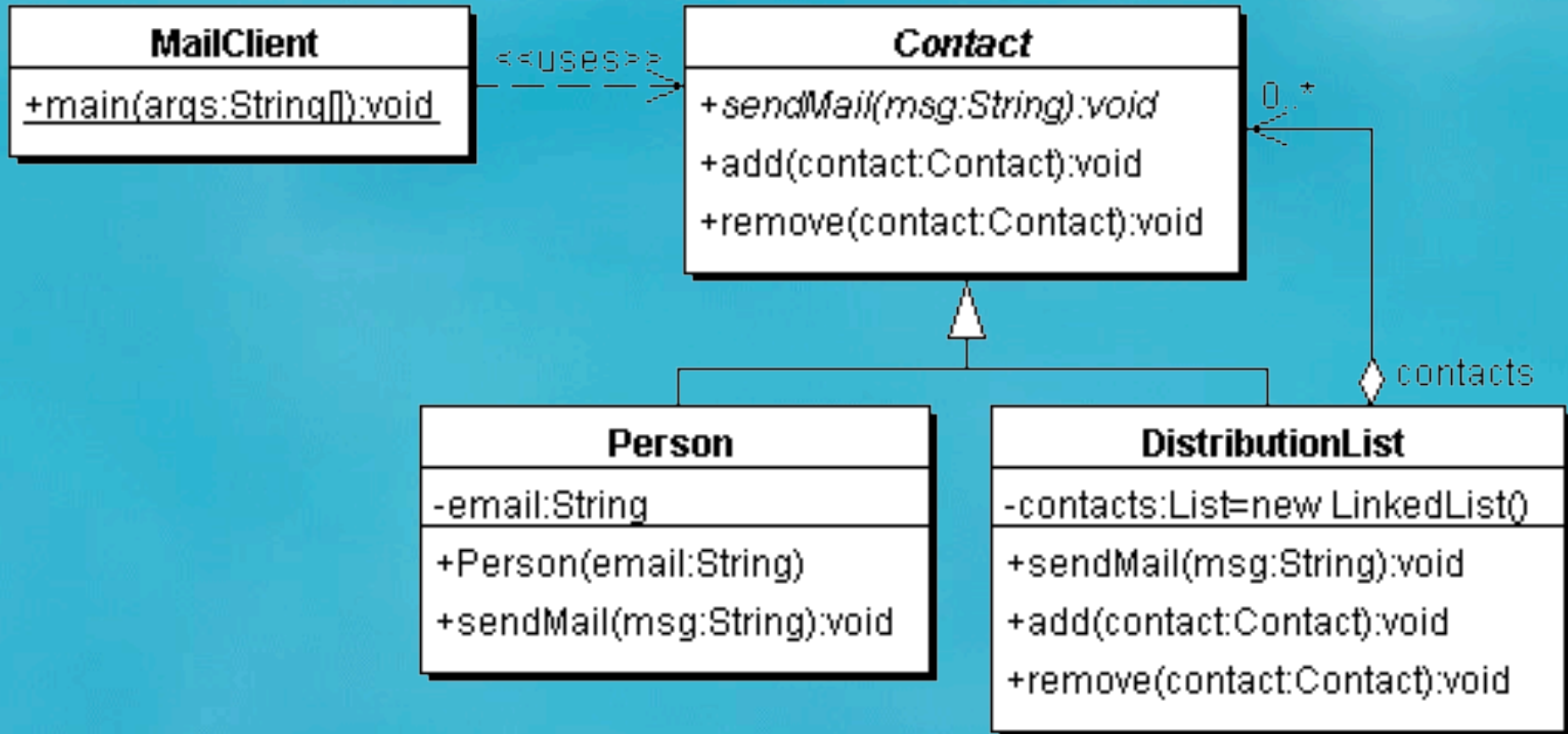


2: Composite

Composite

- Intent
 - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Intent according to Vlissides
 - Assemble objects into tree structures. Composite simplifies clients by letting them treat individual objects and assemblies of objects uniformly.

Motivation: Composite



Sample Code: Contact

```
public abstract class Contact {  
    public void add(Contact contact) {}  
    public void remove(Contact contact) {}  
    public abstract void sendMail(String msg);  
}
```

Sample Code: Person

```
public class Person extends Contact {
    private final String email;
    public Person(String email) {
        this.email = email;
    }

    public void sendMail(String msg) {
        System.out.println("To: " + email);
        System.out.println("Msg: " + msg);
        System.out.println();
    }
}
```

Sample Code: DistributionList

```
import java.util.*;
public class DistributionList extends Contact {
    private List contacts = new LinkedList();
    public void add(Contact contact) {
        contacts.add(contact);
    }
    public void remove(Contact contact) {
        contacts.remove(contact);
    }

    public void sendMail(String msg) {
        Iterator it = contacts.iterator();
        while(it.hasNext()) {
            ((Contact)it.next()).sendMail(msg);
        }
    }
}
```

Sample Code: MailClient

```
public class MailClient {
    public static void main(String[] args) {
        Contact tjsn = new DistributionList();
        tjsn.add(new Person("john@aol.com"));
        Contact students = new DistributionList();
        students.add(new Person("amrita@intnet.mu"));
        tjsn.add(students);
        tjsn.add(new Person("anton@bea.com"));
        tjsn.sendMail(
            "welcome to the 5th edition of ...");
    }
}
```


> java MailClient ↵

To: john@aol.com

Msg: welcome to the 5th edition of ...

To: amrita@intnet.mu

Msg: welcome to the 5th edition of ...

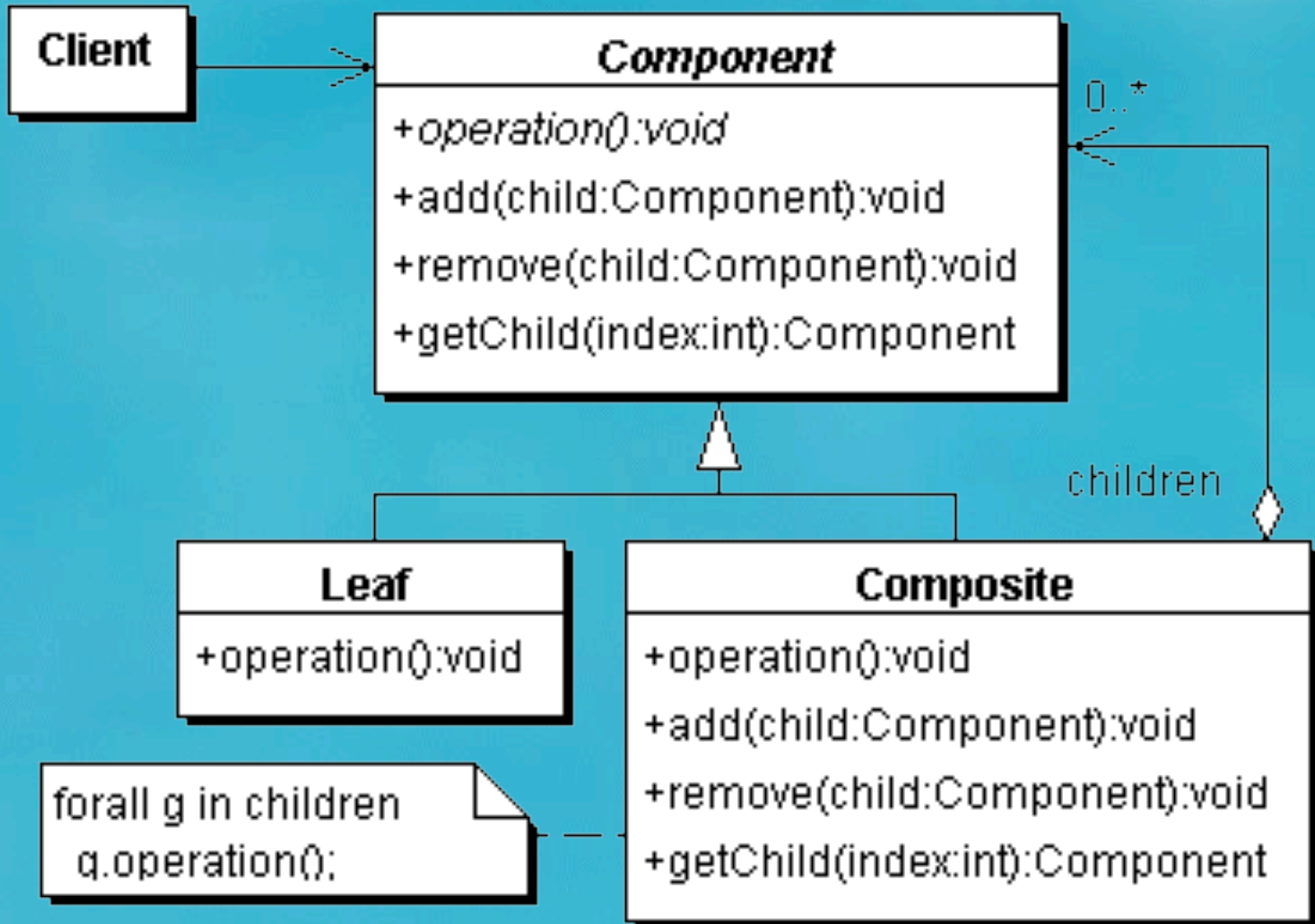
To: anton@bea.com

Msg: welcome to the 5th edition of ...

Applicability: Composite

- Use the Composite pattern when
 - you want to represent part-whole hierarchies of objects.
 - you want clients to be able to ignore the difference between compositions of objects and individual objects.

Structure: Composite



Consequences: Composite

- Benefits
 - defines class hierarchies consisting of primitive objects and composite objects
 - makes the client simple
 - makes it easier to add new kinds of components
- Drawbacks
 - can make your design overly general

Known Uses: Composite

- `java.awt.Component`
- `java.io.File`

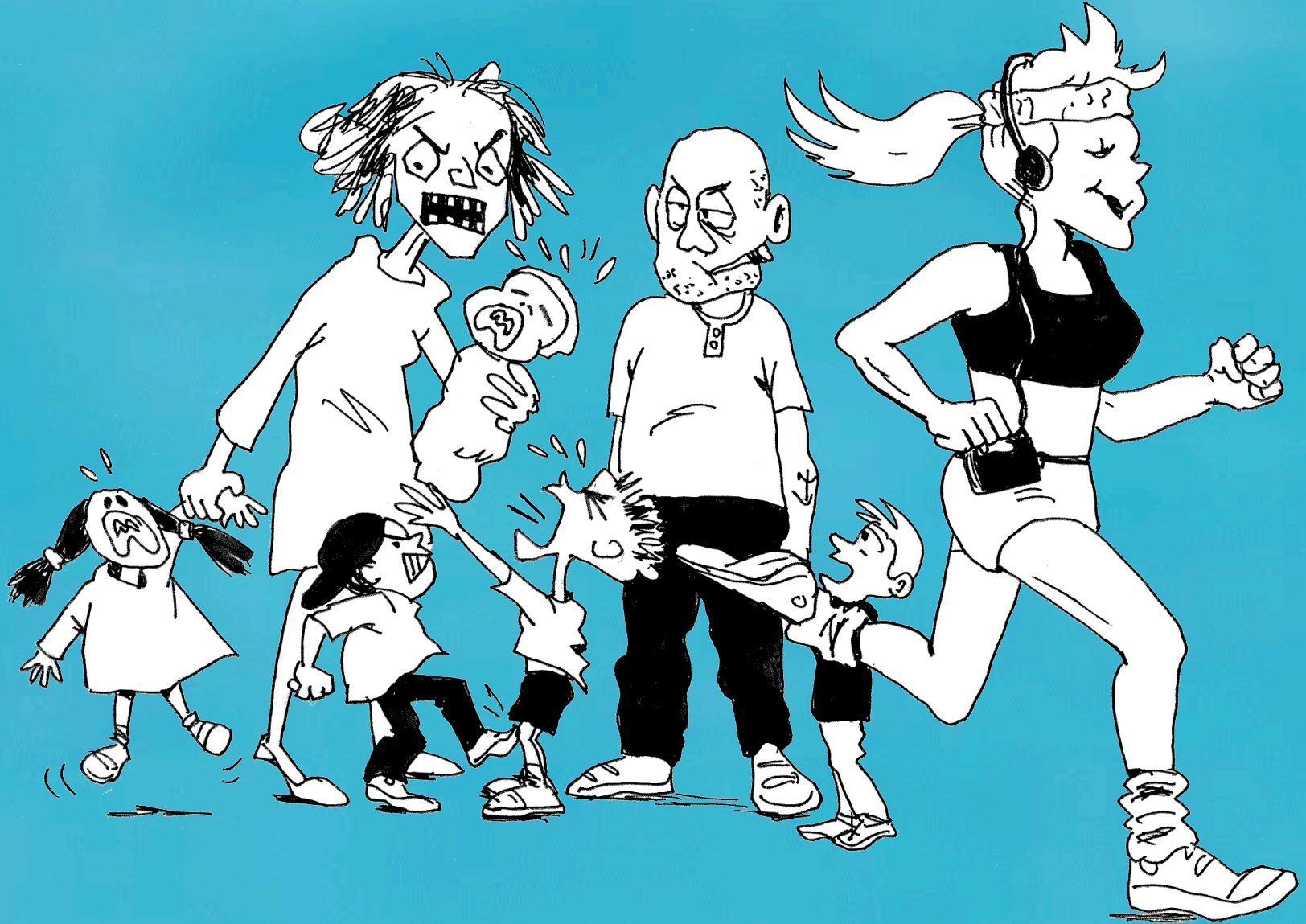
Questions: Composite

- The Composite Pattern is one of the most commonly used patterns in Object Orientation. How would you go about designing the Mailing List example without this patterns, i.e. without having a common superclass?
- What maintenance issues would this cause?

Exercises: Composite

- Add **isLeaf():boolean** and **children():Iterator** methods to **Contact**. **children()** returns an Iterator of all children of the current contact (not recursively). Leaves would return a **NullIterator** (which is a Singleton).
- Write an external **ContactIterator** class that returns all the leaves below a **Contact**.
- Map the Contact example to a relational database.

3: Singleton



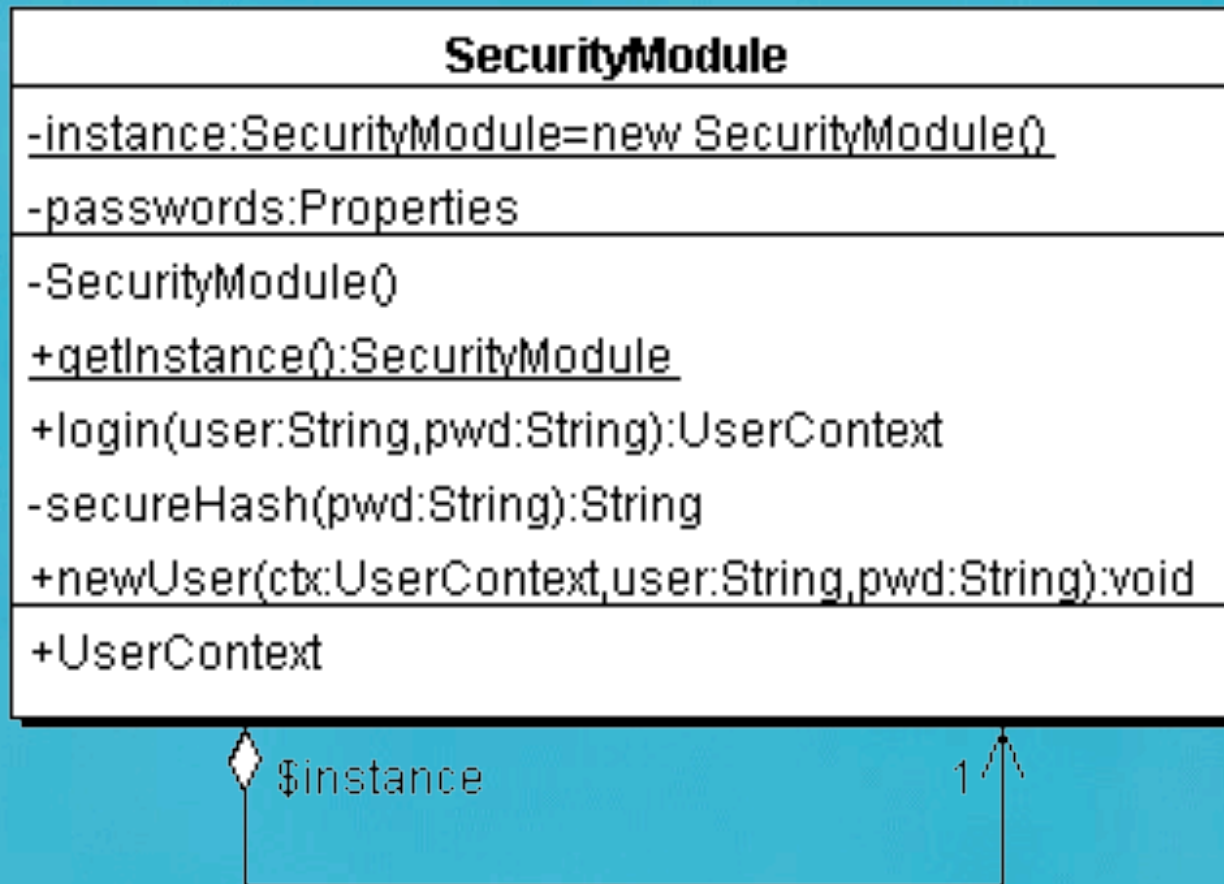
Singleton

- 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();
    }

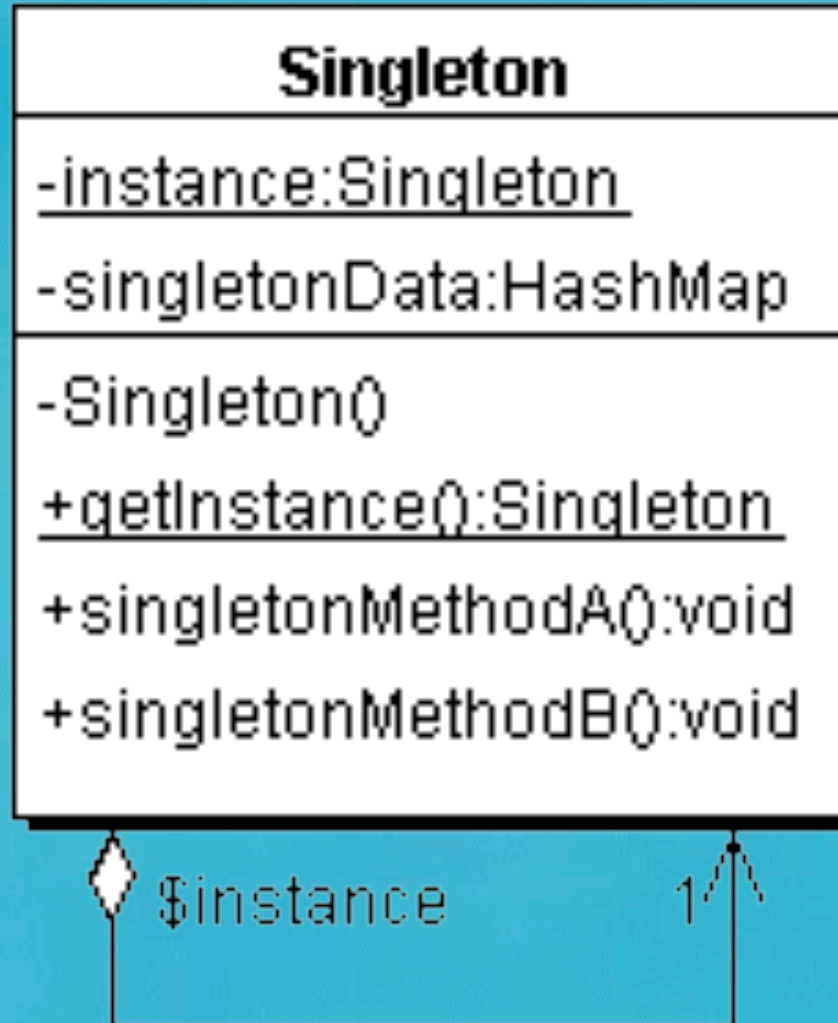
    public UserContext login(String username,
        String password) {
        return new UserContext(username, password);
    }

    // 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

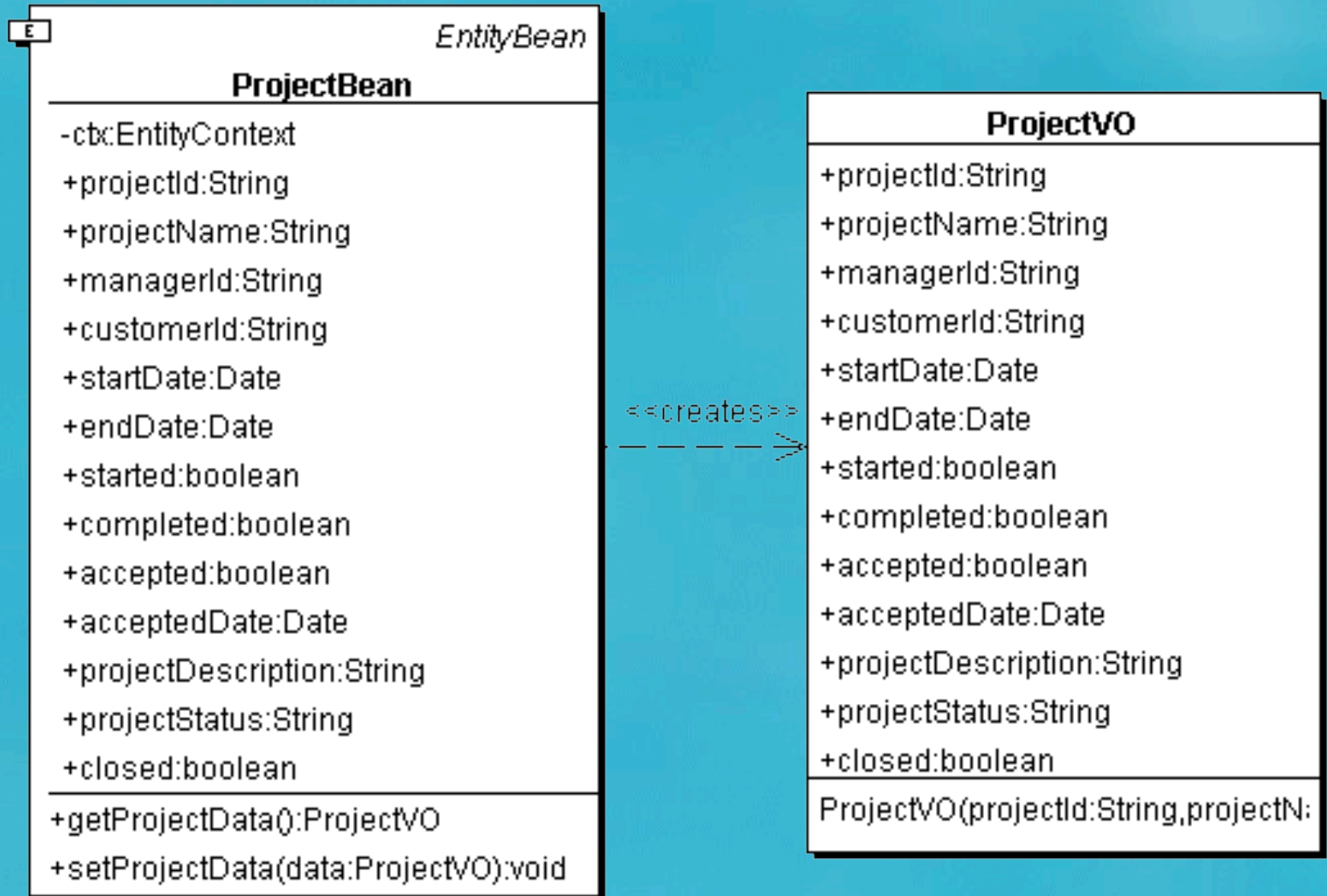
4: Value Object



Value Object

- Intent
 - Efficiently transfer remote, fine-grained data by sending a coarse-grained view of the data.
- Also known as
 - Data Transfer Object, Replicate Object

Motivation: Value Object



Sample Code: Value Object

```
public class ProjectVO implements Serializable {
    private String projectId;
    private String projectName;
    private String managerId;
    // ...
    ProjectVO(String projectId,
              String projectName, ... ) {
        this.projectId = projectId;
        this.projectName = projectName;
        this.managerId = managerId;
        // ...
    }
    public String getProjectId() {
        return projectId;
    }
    public void setProjectId(String projectId) {
        this.projectId = projectId;
    }
}
```

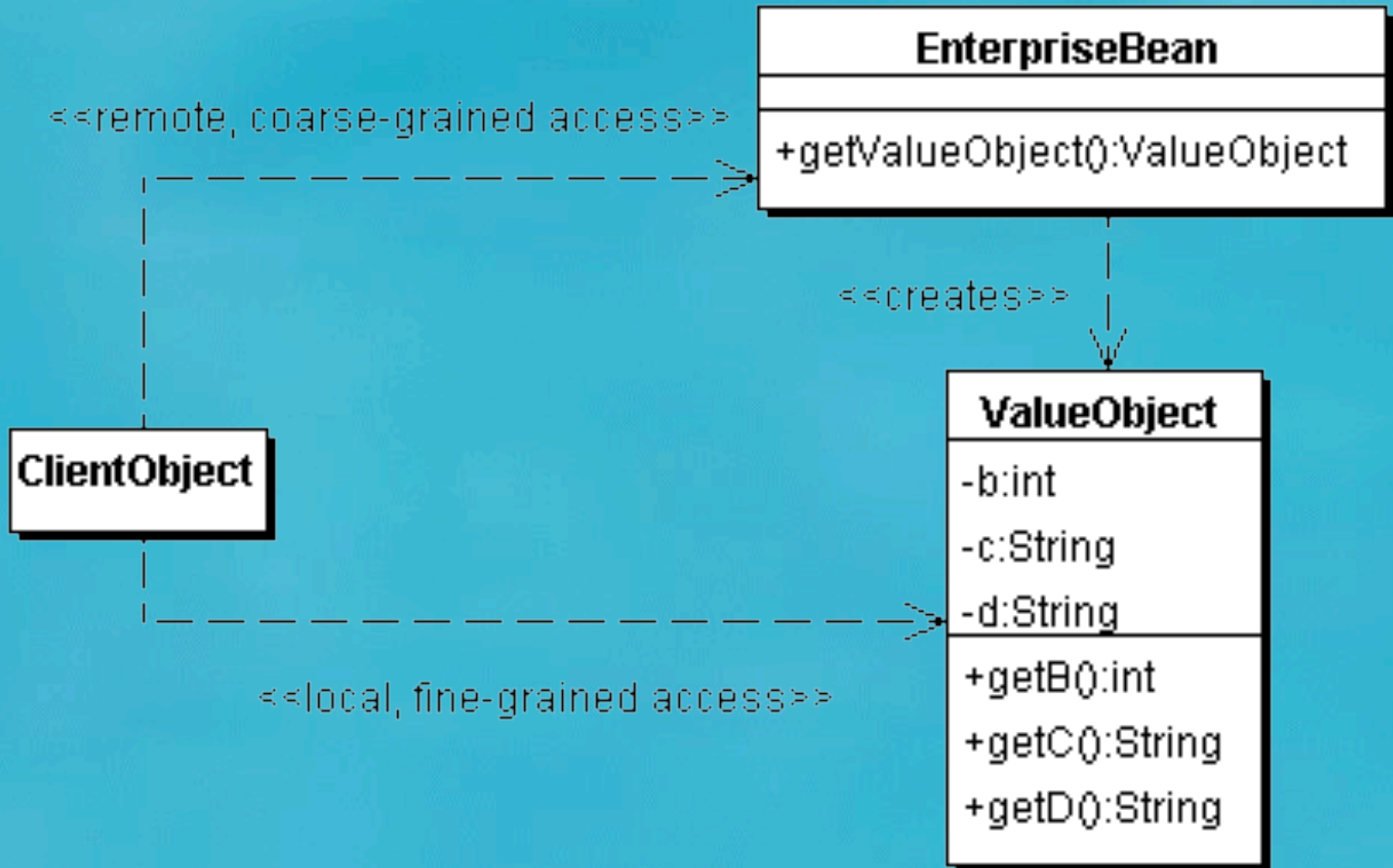
Sample Code: Value Object

```
public class ProjectBean implements EntityBean {
    public String projectId;
    public String projectName;
    public String managerId;
    ...
    public ProjectVO getProjectData() {
        return new ProjectVO(projectId,
            projectName, ... );
    }
    public void setProjectData(ProjectVO data) {
        // version control would be necessary and
        // only changed values should be set
        projectId = data.projectId;
        projectName = data.projectName;
        ...
    }
}
```

Applicability: Value Object

- Use value objects
 - to provide a coarse-grained view of remote, fine-grained data.
 - as a local-access alternative to entity beans.
 - Value object represents a business object without business methods
 - Only provides methods to read its data – ideal candidate for local access

Structure: Value Object



Consequences: Value Object

- Benefits
 - Simplifies entity bean and remote interface
 - Transfers more data in fewer remote calls
 - Reduces network traffic
- Drawbacks
 - May introduce stale value objects
 - May increase complexity due to synchronization and version control

Implementation: Value Object

- Updatable Value Objects
 - contain **setData()** as well as **getData()**
- Multiple Value Objects
- Entity inherits Value Object
 - Reduces duplicate code
- Generic Value Object factories
 - Value Object would then just be an interface
 - Concrete Value Object generated by reflection

Known Uses: Value Object

- Petstore example
 - CustomerEJB aggregates values into ContactInformation

Questions: Value Object

- Value Object introduces additional complexity into your application, with stale data, etc. How would you break up an Invoice class (containing Address, LineItems, etc.) into Value Objects? Would you use Entity Beans? If so, what would you use them for? How would Value Objects come into play?

Exercises: Value Object

- Change **ProjectBean** in motivation to extend **ProjectVO**.
- Split the **ProjectVO** shown earlier into **ProjectDetailsVO** and **ProjectStatusVO**. You will have to change the value objects into interfaces and make a **ValueObjectFactory** class that creates them.

Exercises: Value Object

- Design a **Project** VO that contains a **ProjectStatus** VO Proxy which is retrieved lazily using the Virtual Proxy pattern.
- Design an Iterator that retrieves **ProjectTask** VOs lazily.

5: Conclusion

- Design Patterns will help you write real Object Orientated code
- The textbook by GoF is very intimidating
 - <http://www.javaspecialists.co.za/books.html>
- Best way to learn Design Patterns is through a course:
 - <http://www.javaspecialists.co.za>
- Questions ...
- Email: heinz@javaspecialists.co.za

Design Patterns Cape Town



Design Patterns Germany



Design Patterns London



A large, curved outdoor swimming pool with clear blue water. In the background, there is a large thatched hut with a conical roof. The pool is surrounded by yellow lounge chairs and closed umbrellas. The ocean is visible in the distance under a cloudy sky. The scene is viewed from a wooden deck with dark wooden beams in the foreground.

Design Patterns Mauritius

Design Patterns Estonia at -18° Celsius



What Next: Design Patterns Course in Durban?

Please contact heinz@javaspecialists.co.za
(083) 340-5633