OBJECT ORIENTATION AND OBJECT PATTERNS

Design failures

- Rigidity
  - Hard to modify functionality without major re-work (e.g. fixed strings in menu items)
- Fragility
  - P(error) high after modification
- Immobility
  - Code hard to re-use on other projects (often due to coupling)
- Viscosity of design
  - Hard to modify code functionality and keep original design philosophy

© University of Liverpool

Examples

- Rigidity
  - String yes="Yes";
  - String no="No";
- Fragility

```
if (c>=65) && (c<=90)
  c=c+32;
```

DRY

- Every piece of knowledge must have a single, unambiguous, authoritative representation within a system

© University of Liverpool

Good design principles

- DRY
  - Don't Repeat Yourself
  - So
    - 1 version of (data, algorithm, document, image file, test plan)
    - Authoritative
    - Example for encryption algorithm use a code generator to generate javascript version from Java
  - Ok
    - To have cached copies, as long as they are generated from the original

© University of Liverpool

DRY and calculated data

- Transaction amounts
  - +10 -100 -40

- Should not store balance, should calculate it
- Because
  - If the 2 don't agree which one is right?

Other DRY violations

- Storing repeated data across database tables
- Storing derived data across database tables (caching results is ok)
- Storing same data at client and server
  - Generally always better to store at server and download on initialising client

© University of Liverpool

Design principles

- Open Closed Principle (OCP) (Bertrand Meyer)
  - Definition
    - Open for extension
    - Closed from modification
  - In practise implemented using
    - Use of interface definitions (head, no body)
      - Dynamic polymorphism (run time type checking)
    - Use of generics
      - Static polymorphism (compile time checking)

© University of Liverpool

Dynamic polymorphism

```
Interface Printable {
    public void doPrint();
}

Class Circle implements Printable {

Public void doPrint() {
    //
}
}
```

// If object is type Printable, doPrint implementation is determined at run time

© University of Liverpool

Static polymorphism

- Generics/templates
  - One code base, but types can modify

```
public class Stack<E> {
        public <E> pop() {

        }
}

public class Main {
  public static void main(String argvs) {
    Stack <int> myStack; // type is fixed at compile time
  }
}
```

Liskov Substitution Principle (LSP) **Barbar Liskov**

- Subclasses should be substitutable for their base classes
- Circle/(Ellipse) example
- Class Ellipes {
  - setFocus1(Point focus)
  - setFocus2(Point focus)
  - Point getFocus1()
  - Point getFocus2()
- }

LSP violation example

- Class Circle extends Ellipse {
  - setFocus1(Point center) {
    - super.setFocus1(center);
    - super.setFocus2(center);
  - }
  - setFocus2(Point center) {
    - super.setFocus1(center);
    - super.setFocus2(center);
  - }

© University of Liverpool

LSP violation detection

```
public void (Ellipse e)
{
        Point a(-1,0);
        Point b(1,0);
        e.Focus1(a);
        e.Focus2(b);
        assert(e.GetFocus1() == a); // if e is type Circle
        assert(e.GetFocus2() == b);  // assertion will fail!!
}
```


© University of Liverpool

Design by contract

- Circle breaks the implicit contract of the Ellipse class and therefore violates LSP
- Design by contract
  - Can be defined in languages such as Eiffel, each method has a contract which is checked on each invocation
  - For other languages use assertions

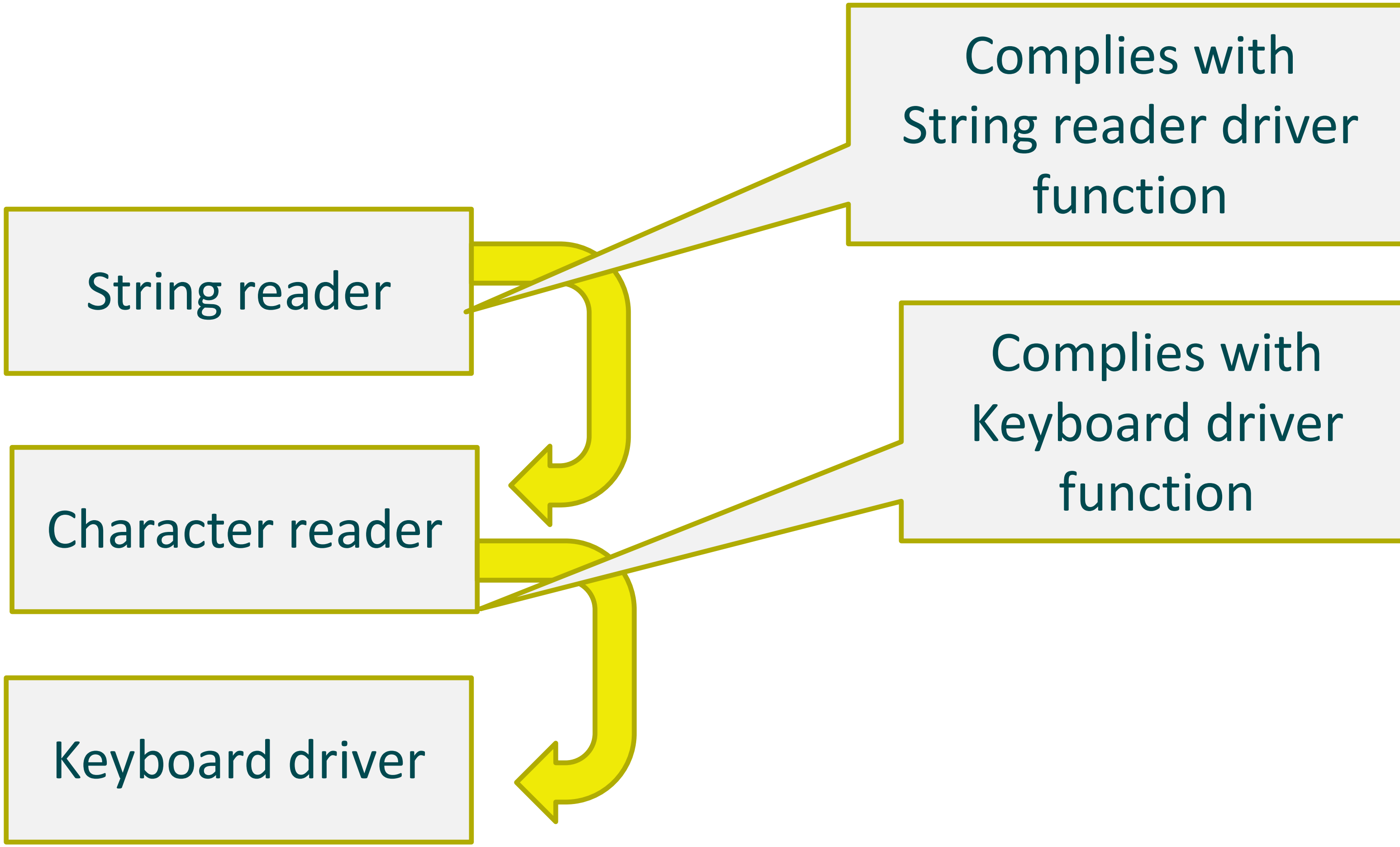
© University of Liverpool

Dependency Inversion Principle

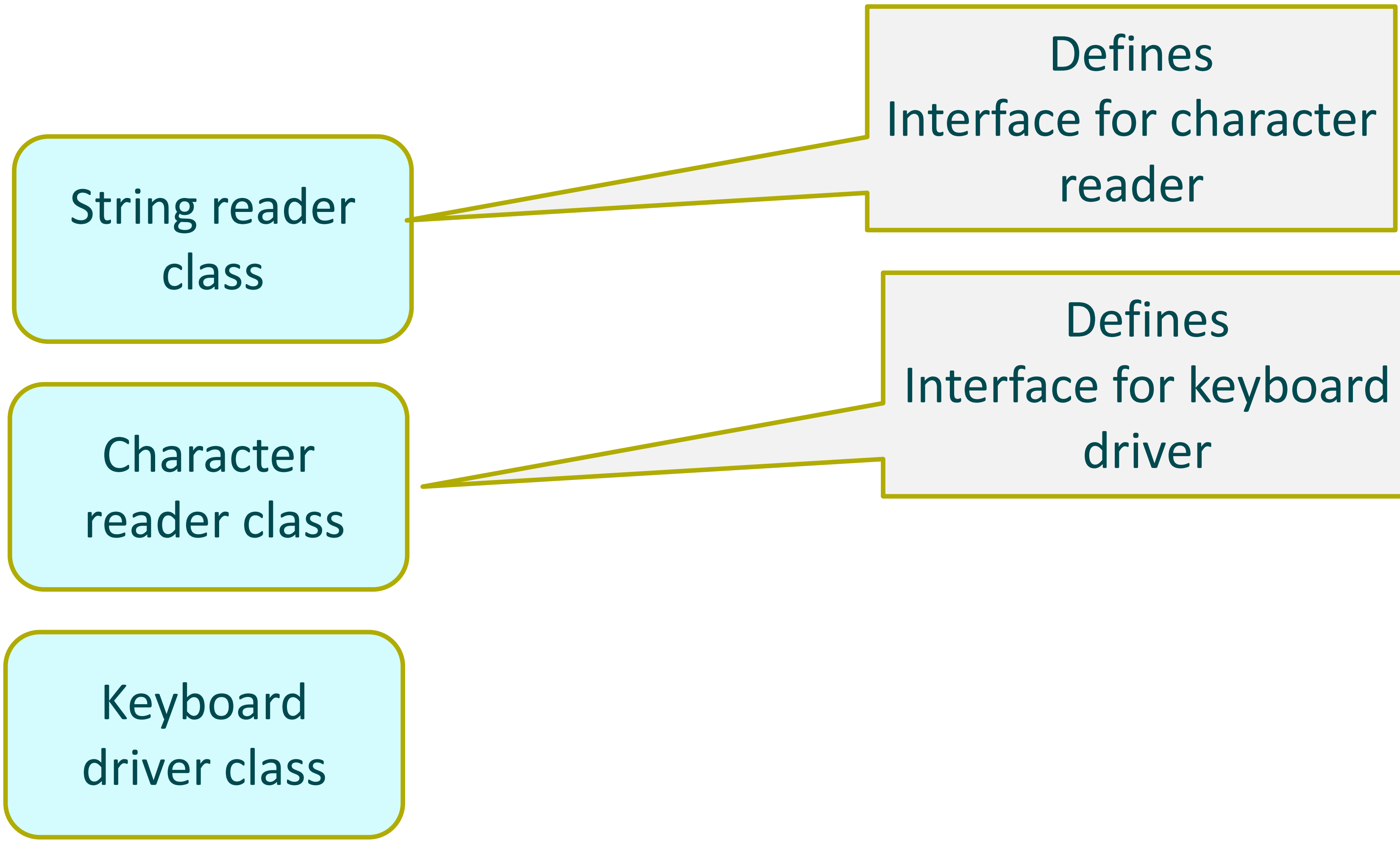
- Tradition dependency
  - High level (application level) methods rely on lower detailed functions
  - Unreliable
    - Low level concrete functions are liable to change (e.g. ascii -- > unicode handling)
    - Changes in low level can break high level code

© University of Liverpool

Conventional development…



OO design



© University of Liverpool

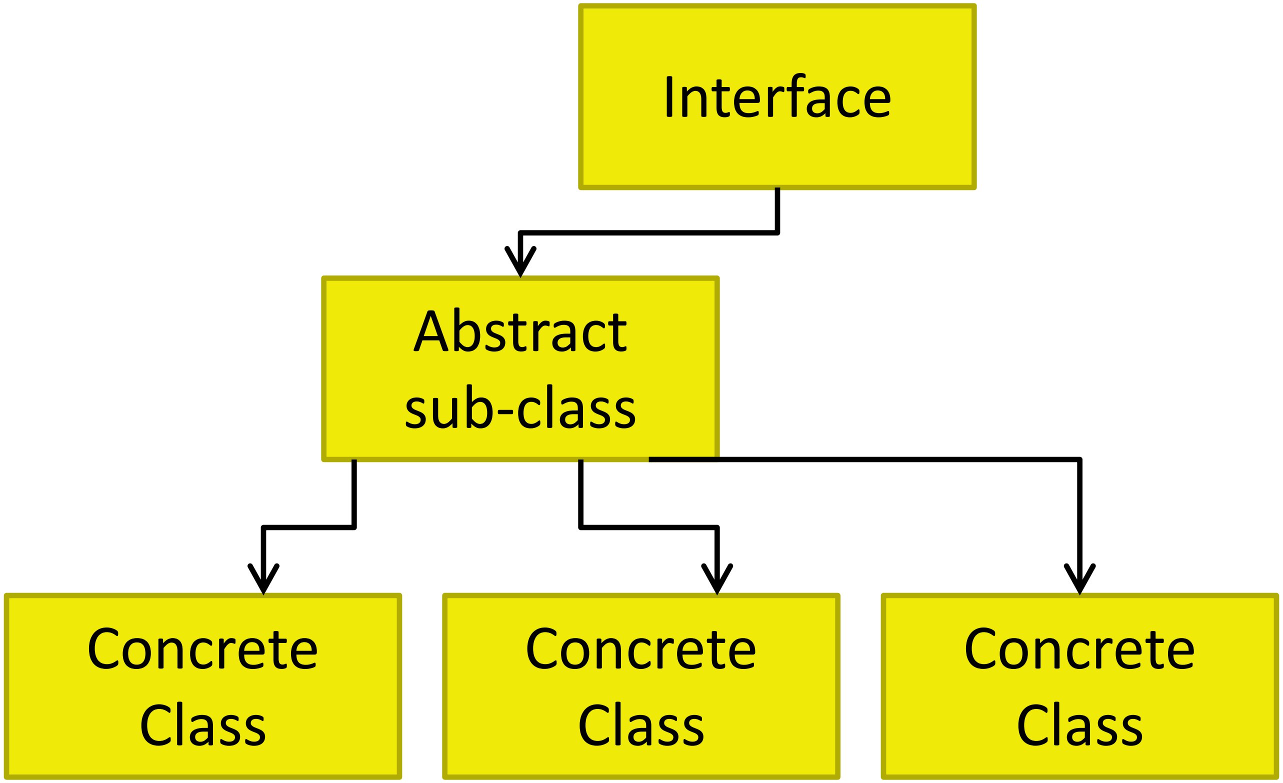
Dependency inversion

- Abstract service
  - Designed first
- Concrete implementation
  - Designed last
- Leads to
  - More portable design
  - Design driver by service required

© University of Liverpool

Dependency Inversion Principle

- With OO, concrete base classes depend on high level interfaces so…
- Depend upon Abstractions. Do not depend upon concretions.



© University of Liverpool

Object creation and DIP

- When creating class instance, you often need to be dependent on a concrete class
- Image p=new Image(“fred.png”);
- Our code is now directly coupled to the constructor for image….
- To de-couple this interface it is common to use some form of abstract factory

© University of Liverpool

Interface Segregation Principle

- If a class has a large interface to a number of different clients, break it down into
  - Different interfaces for each client type
- E.g. SQL execution helper
  - Insert interface
  - Select interface
  - Delete interface

© University of Liverpool

Class packaging principles

- Release Reuse Equivalency Principle
  - The granule of reuse is the granule of release. Only components that are released through a tracking system can effectively be reused. This granule is the package.
- Common Closure Principle
  - Classes that change together, belong together
- Common Reuse Principle
  - Classes that aren't reused together should not be grouped together
  - Consequence, keep packages as small as possible but which doesn't violate CCP

© University of Liverpool

Language levels

Object-oriented language (e.g. C++, Smalltalk, Java, C#)

High-level language (e.g. C, Pascal)

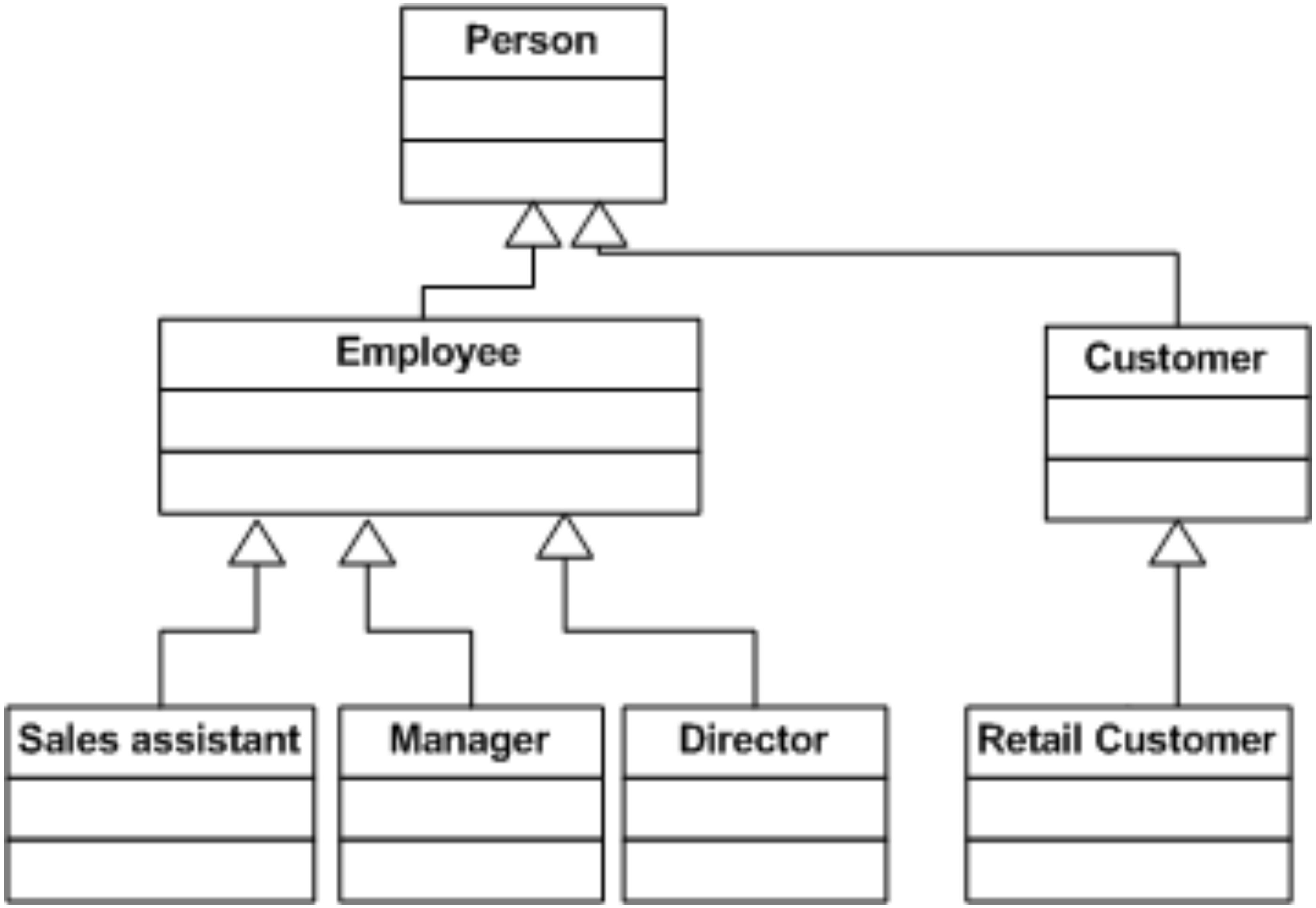
Assembly languge (e.g. IBM or Intel assembly language)

Machine language (ultimate output of compiler and/or assembler)

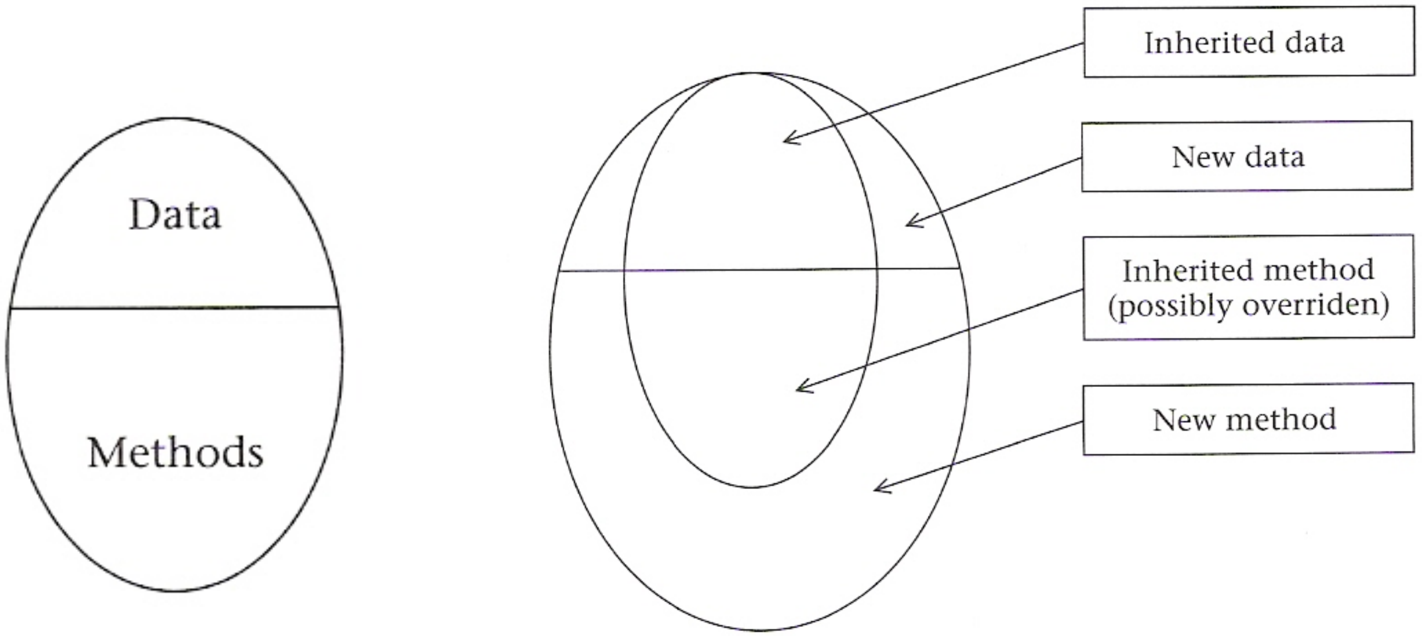
Microcode (tells the processor how to interpret machine language instructions)

© University of Liverpool

Classification



Encapsulation & Inheritance



 © University of Liverpool

Benefits of OO approach

- Inheritance - classes
- Encapsulation - classes + methods
- Polymorphism - function
- good Cohesion
- good Coupling

© University of Liverpool

OO Analysis (!= OO design)

" … is figuring out how to arrange a collection of classes that do a good job of representing your real-world problem in a format which a computer programmer finds easy to deal with."

- Input
  - Thinking effort
  - Pencil, paper and Notebook
  - Observations
- Output
  - Answer to "which classes to use?"
  - UML diagrams

© University of Liverpool

Object Orientated design

“ … is about what kinds of data and method go into your classes and about how the classes relate to each other in terms of inheritance, membership and function calls.”

- Input
  - Thinking effort + Pencil Paper, Notebook
  - OOA diagrams
- Output
  - UML diagrams
  - Header files (e.g. *.h files)

© University of Liverpool

Role of documentation

- Central communication
  - Cut down on communication overhead
- Control
  - If it’s not in the specification, it won’t be built
- Annotation
  - Particularly of code but also design
- Operational
  - User/system manuals

© University of Liverpool

Types of Documentation

- UML diagrams
- User Guides
- System Guides
- Management documents
- Requirement and Specification
- Schedule and Budget
- Organisation and Planning

© University of Liverpool

Design Patterns

Software Evolution → Patterns

Software design patterns

Object-oriented language (e.g. C++, Smalltalk, Java)

High-level language (e.g. C, Pascal)

Assembly language (e.g. IBM or Intel assembly language)

Machine language (ultimate output of compiler and/or assembler)

Microcode (tells the processor how to interpret machine language instructions)

Design patterns

- Repeatable approaches to problem solving in software design
- Not locked into any 1 language (but often use OO concepts)
- Speed up development
- Increase software flexibility
- Make software more readable
- Can be implemented as components which will move from reusable design to reusable code

 © University of Liverpool

Patterns and Components

- Patterns
  - Approaches to the problem
- Components
  - Re-usable code which solves approach

© University of Liverpool

Design Pattern types

- Architectural (approach to designing the whole system) example MVC
- Creational
- Structural (one class/method wrapping another)
- Behavioural
  - Example : call backs, persistence
- Concurrency
  - Controls multiple threads

Model View Controller

- Problem
  - Many different GUI APIs
  - GUI code can be very complex and messy
  - Porting GUI code between platforms is hardwork

© University of Liverpool

MVC Components

- Splits the code into
  - Model
    - Stores, retrieves and manipulates the data
  - View
    - Renders the data on the screen
    - View fetches data from model
  - Controller
    - Processes user input, passing events to model
    - Controller can instruct view to render

© University of Liverpool

Model

- Provides the following
  - business logic, rules (e.g. who can access a student's transcript)
  - validation (can also be in controller)
  - persistence
  - application state (session)
    - shopping cart for user
    - address book, contact list
    - logged in user id

© University of Liverpool

View

- Presents the information to the user
- Example View technologies
  - JSP   allows user to use Java to generate web pages
  - CSS web page presentation
  - HTML/XML
  - .aspx   Microsoft dynamic web technology

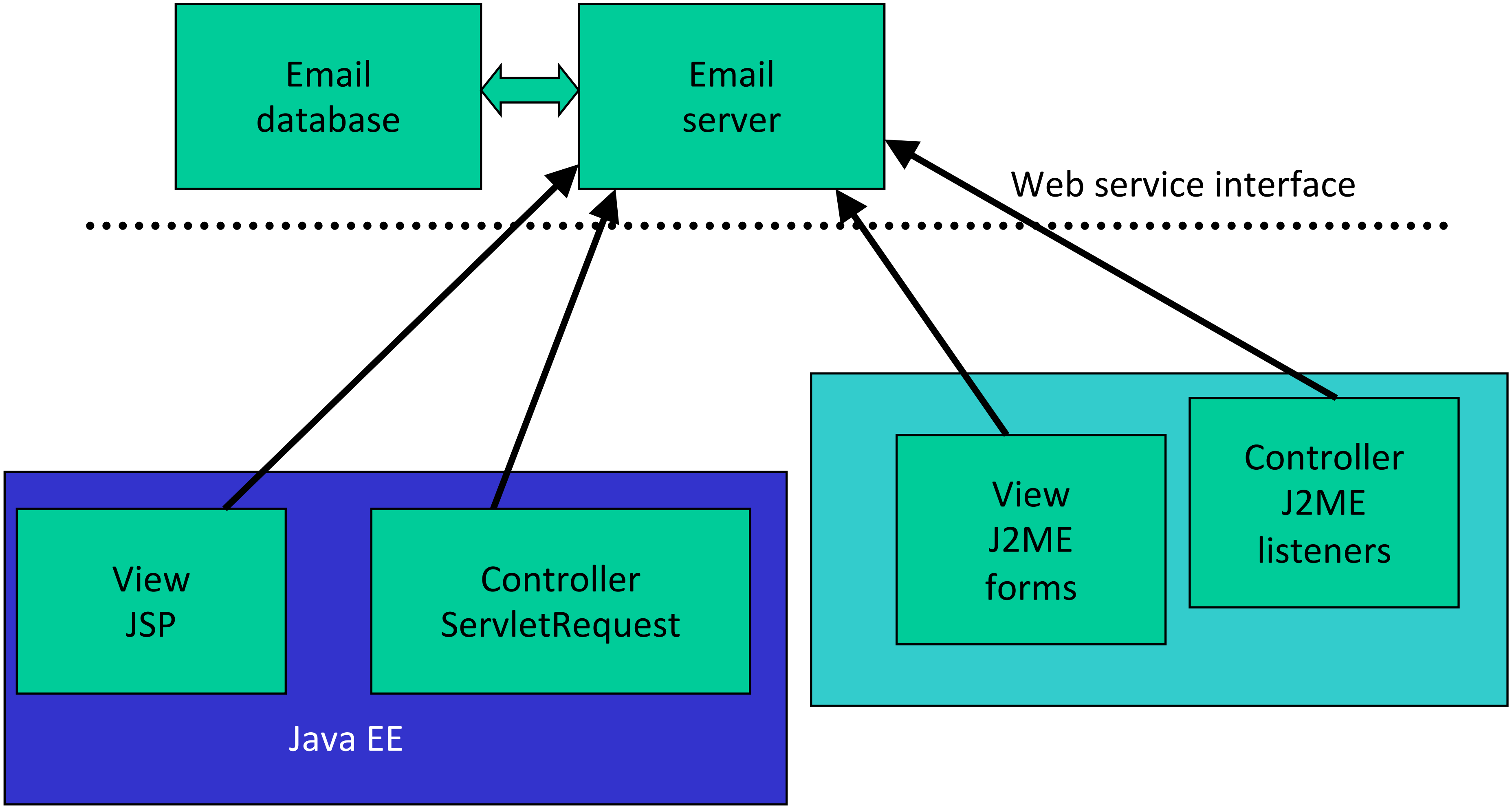
© University of Liverpool

View/Controller options

- Java servlets and JSP (browser client)
  - Java EE (Tomcat or Glassfish)
- .NET aspx pages (browser client)
  - Microsoft Server
- J2ME MIDP
  - Mobile Java
- Java AWT/Swing
  - Java SE

 © University of Liverpool

MVC Example



Email database
Email server
Web service interface
View JSP
Controller ServletRequest
Java EE
View J2ME forms
Controller J2ME listeners

Model code example

Plain old Java class

```
class Customer {
    private String surname;
    private String forenames;
    private Date dateOfBirth;
}
```

Note this class can be ported to any platform that supports Java

© University of Liverpool

View code example

```
Class CustomerForm extends Form {
    private TextField tfSurname; // text field input surname
    private TextField tfForenames; // forenames input
    private DateField dfDateOfBirth; // date of birth input
    private Command ok;
}
```

Controller Code (J2ME)

```
CustomerFormListener implements CommandListener {
  CustomerForm customerForm;
  public void commandAction(Command c, Displayable
    displayable) {
     if ( (c.getCommandType()==Command.OK)) {
                Customer customer=customerForm.getCustomer();
                customer.Save();
     }
}
```

MVC Model View Controller

- Benefits
  - Clear seperation of concerns
  - Easier to port software UI platform to UI platform
- VC code
  - Can be implemented by GUI specialist
- Team working
  - Web, Mobile App (iOS, Android), Mobile Web
  - Business logic

Command pattern

- Command
  - general abstraction for controller type interactions
  - allows controller API to change and keep business logic the same
- Code example

```
interface Command {
  void   OnExecute();
}
```

Command interface detail

```
public abstract class Command {
    private Hashtable <String,Object> callParameters=new Hashtable();
    private Hashtable <String,Object> returnParameters=new Hashtable();
    protected abstract void OnExecute();

    protected void setCallParameter(String name,Object object) {
        callParameters.put(name, object);
    }

    public void setCallParameters(Hashtable parms) {
        this.callParameters=parms;
    }

    protected Object getCallParameter(String name) throws ParameterNotFoundException {
        if (callParameters.containsKey(name)) {
            return(callParameters.get(name));
        }
        throw(new ParameterNotFoundException());
    }
}
```

CommandManager

```
public class CommandManager {
public void Execute(Hashtable parameters) throws
NoSuchCommandException,CommandNameMissingException {
      String packageName="patterns.commands";
       if (!parameters.containsKey("name")) {
         throw (new CommandNameMissingException());
      }
      String name=(String)parameters.get("name");
      String commandName=packageName+name;
      try {
        Class commandClass=Class.forName(commandName);
        Command commandObject=(Command)commandClass.newInstance();
        if (parameters!=null) {
          commandObject.setCallParameters(parameters);
        }
        commandObject.OnExecute();
      } catch (Exception exc1) {
         throw (new NoSuchCommandException(name));   // problem with command class
      }
   }
}
```

HttpCommandManager extends CommandManager

```
public void Execute(HttpServletRequest request) throws
NoSuchCommandException, CommandNameMissingException {
    Enumeration allNames=request.getParameterNames();
    Hashtable <String,Object> parameters=new Hashtable <String,Object> ();
    while (allNames.hasMoreElements()) {
        String pname=(String)allNames.nextElement();
        String parmValue=request.getParameter(pname);
        parameters.put(pname, parmValue);
    }
    Execute(parameters);
}
```

Factory class

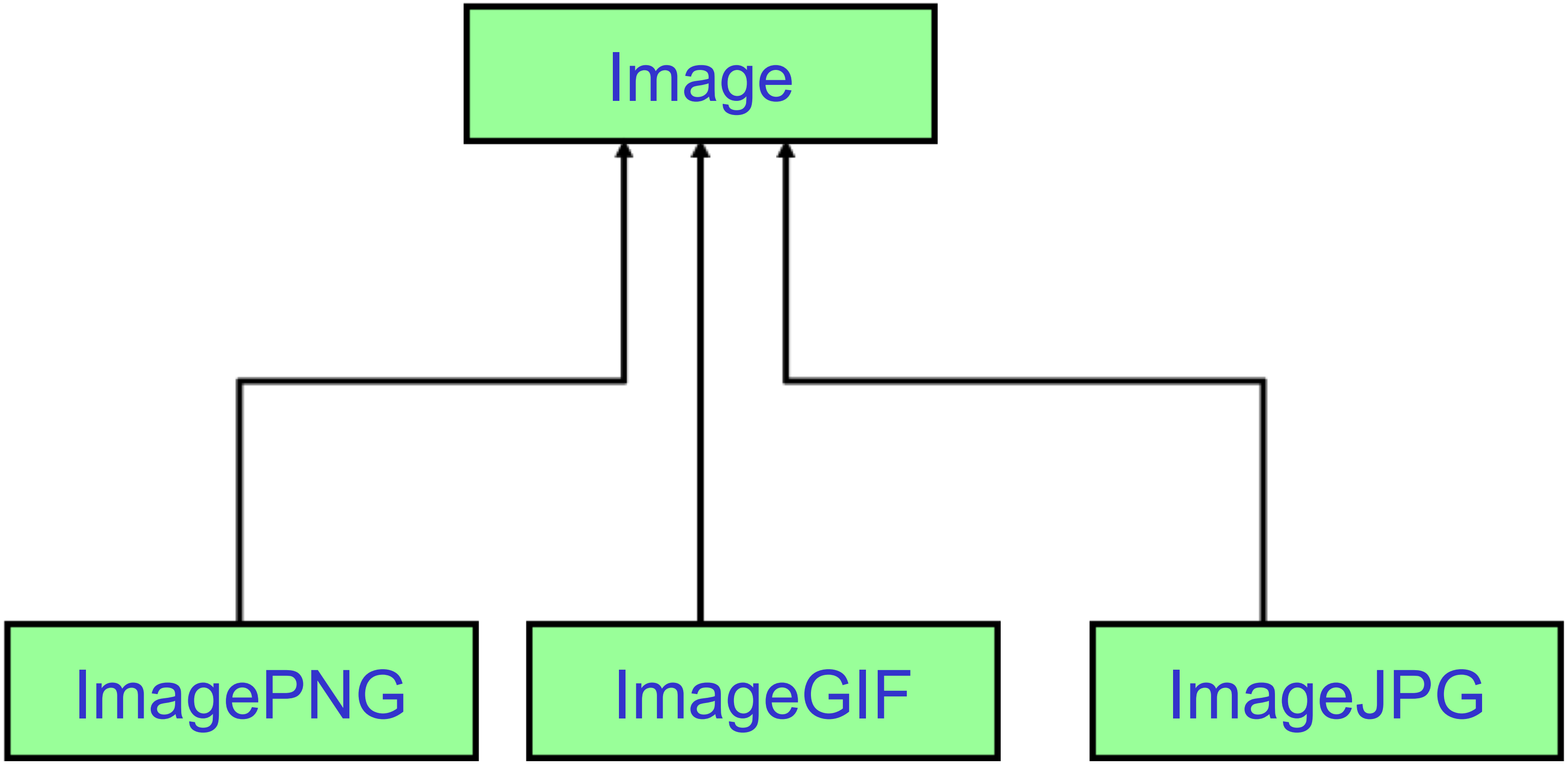
- Factory method constructs instances of a class
- Problem
- Constructing a Image class
  - Image format could be png, gif, jpg
  - Each format could have different image class

  - Calling code needs to use different class depending on image type
  - ImagePNG image=new ImagePNG("/picture.png");
  - Type may not be know till runtime

© University of Liverpool

Factory example

- Solution
  - Use inheritance from abstract class Image



```
public static  createImage(String fname) throws
Exception {
   if (fname.endsWith(“.gif”)) {
     return( (Image) new ImageGIF(fname) );
    }
    if (fname.endsWith(“.png”)) {
      return( (Image) new ImagePNG(fname) );
    }
    if (fname.endsWith(“.jpg”)) {
      return( (Image) new ImageJPG(fname) );
    }
    throw new Exception("Unknown image type
for file "+fname);
}
```

Singleton

- Single instance of class
- Constructor is private
- static final Class instance constructed when application loads
- or loaded only when need (lazy initialization)
- Examples of usage
  - to access database so that all threads go through one control point
  - Font class keeps memory load low

© University of Liverpool

Singleton Example in Java

```
public class DbaseConnector {
   private static final DbaseConnector instance=new
   DbaseConnector();
   private DbaseConnector() {
        // database construction code…..
   }

   public static DbaseConnector getInstance() {
      return(instance);
  }
}
```

Singleton Example (lazy initialization)

```
public class DbaseConnector {
    private static DbaseConnector instance;
    private DbaseConnector() {
            // database construction code…..
    }
    public static DbaseConnector synchronized getInstance() {
        if (instance==null) {
             instance=new DbaseConnector();
        }
        return(instance);
    }
}
```

 © University of Liverpool

Wrapper classes

- Problem
  - Different external technologies to connect to
  - Example for database connection
    - ODBC (Microsoft)
    - JDBC (Java standard)
  - Other examples
    - External Credit card payment
    - Network connection (Java and Microsoft)
    - Data structure libraries

© University of Liverpool

Wrapper classes

- **Problem with coding directly**
  - **Code will end up messy**
  - **Hard to port**
  - **Hard to understand**
- **Benefits of wrapping code**
  - **easier to swap modules (e.g. CC function)**
  - **easier to implement standard functions (e.g. accountancy, error logs)**

© University of Liverpool

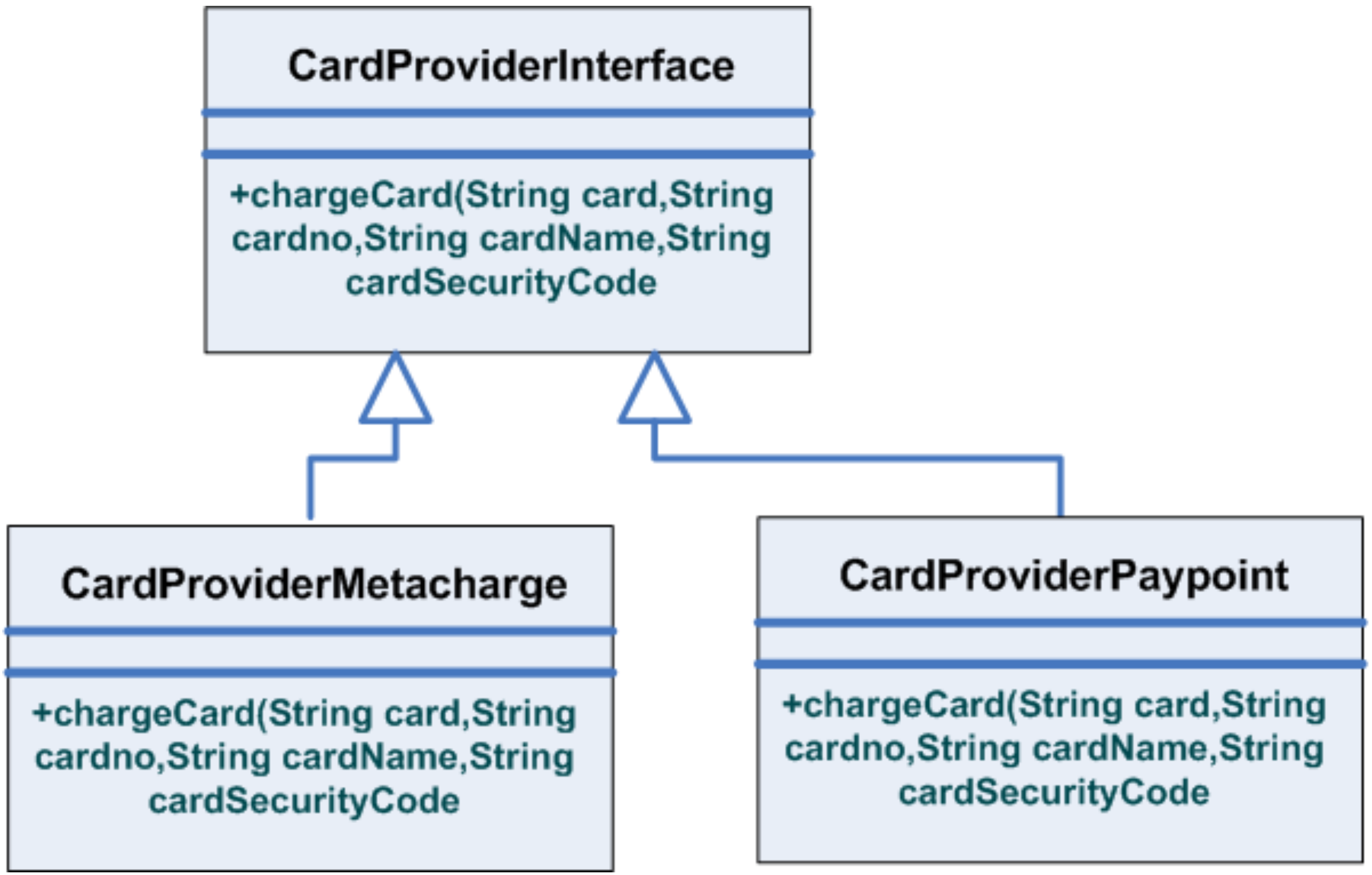
Wrapper example (unwrapped code)

```
String sql="select * from customers";
    try {
        java.sql.Statement      s=dbConnection.createStatement();
        int rows=s.executeUpdate(sql);

     } catch (Exception e) {
        status=sql+" "+e.toString();

     };
```

Wrapped code

```
public class SQLHelper {
  public void executeSQL(String sql) {
        try {
              java.sql.Statement
   s=dbConnection.createStatement();
              int rows=s.executeUpdate(sql);
        } catch (Exception e) {
          status=sql+" "+e.toString();
      };
}
}
```

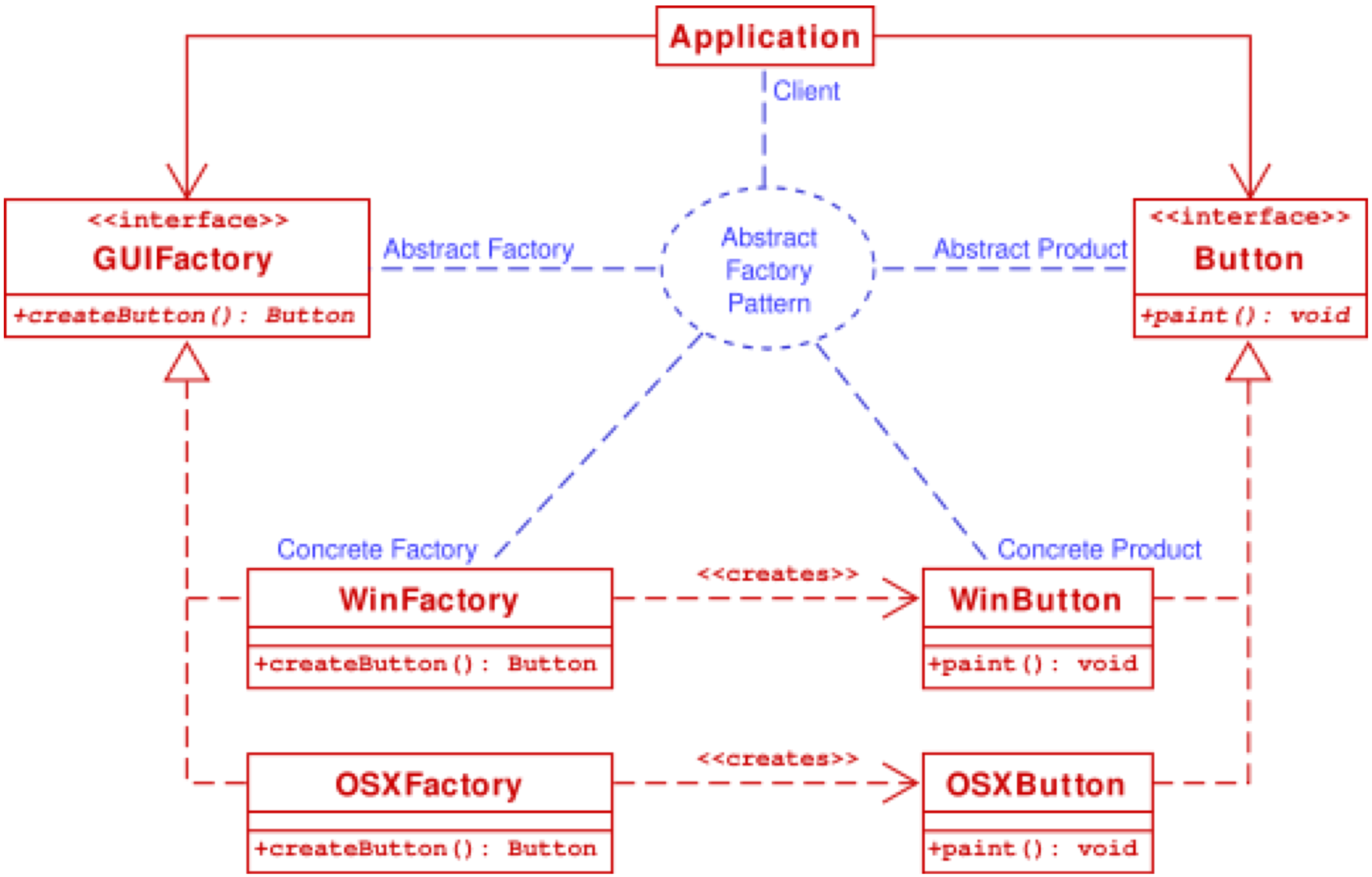
Adapter class diagram example



Abstract factory

- Used when you have an associated set of object types to create, but the actual class to create is decide at run time
- Example:
  - Sets of encryption algorithms from different providers
  - User interface components for different OS UI API

Abstract Factory class diagram



© University of Liverpool

Abstract factory code example

```
interface SecurityFactory {
  public Encryptor createEncryptor();
}
class LowSecurityFactory implement  SecurityFactory {
   public Encryptor createEncryptor() {
         return(new ShortKeyEncryptor());
   }
}
class  HighSecurityFactory implement SecurityFactory {
   public Encryptor createEncryptor() {
         return(LongKeyEncryptor());
   }
}
```


© University of Liverpool

Abstract factory example

```
class Application {
        private Encryptor encryptor;
         public Application(securityFactory sfactory) {
            encryptor=sfactory.createEncryptor();
         }
}

class Start {
   public static void main(String argsv[ ]) {
      Application application;
      if (professionalVersion) {
          application=new Application(new HighSecurityFactory());
      } else {
          application=new Application(new LowSecurityFactory());
      }
   }
}
```


© University of Liverpool

Builder

Separates abstract definition of an object from its representation

Example

Builder for SQL statements

Abstract interface

Defines interface with appropriate elements (table names, columns, indexes)

Concrete definition

SelectBuilder (for select statements)

Coding example

```
public interface ISQLBuilder {
    public void setTableName(String table);
    public String getTableName();
    public void setCommandName(String command);
    public String getCommandName();
    public void addColumnName(String columnName);
    public String toSQLString();
    public String getWhereClause();
    public void addWhereClause(String where);

}
```


© University of Liverpool

```
public abstract class SQLBuilderBase implements ISQLBuilder {
   private String tableName;
   private String commandName;
   private StringBuilder whereClause=new StringBuilder();
   private Vector <String> columnNames=new Vector <String>();

   public String getTableName() {
      return tableName;
   }
   public void setTableName(String tableName) {
      this.tableName = tableName;
   }

   public String getCommandName() {
      return commandName;
   }

public void setCommandName(String commandName) {
      this.commandName = commandName;
   }
```

```
public void addColumnName(String columnName) {
    columnNames.add(columnName);
}

public int getColumnCount() {
    return(columnNames.size());
}

public String getColumnName(int index) {
    return(columnNames.get(index));
}

public void addWhereClause(String whereStatement) {
    this.whereClause.append(whereStatement);
}

public String getWhereClause() {
    return(whereClause.toString());
}
```


© University of Liverpool

Coding example

```
public class SQLSelectBuilder extends
SQLBuilderBase {
    public SQLSelectBuilder(String tableName) {
        super.setTableName(tableName);
        super.setCommandName("SELECT");
    }
```

```
public String toSQLString() {
      StringBuilder sb=new StringBuilder();
      sb.append(this.getCommandName()+" ");
      if (getColumnCount()==0) {
         sb.append("(*)");
      } else {
         sb.append("(");
         for (int idx=0;idx<this.getColumnCount();idx++) {
            sb.append(getColumnName(idx));
            if (idx!=this.getColumnCount()-1) {
               sb.append(",");
            }
         }    sb.append(")");
      }
      sb.append(" from "+this.getTableName());
      sb.append(" where "+getWhereClause());
      return(sb.toString());
   }
}
```

Builder coding example

```
public class Main {
   public static void main(String argvs[]) {
      SQLSelectBuilder builder=new SQLSelectBuilder("customers");
      builder.addWhereClause("customerid=3");
      System.out.println("SQL string is "+builder.toSQLString());

   }

}
```

So why both with all this complexity?

```
SQLSelectBuilder builder=new SQLSelectBuilder("customers");
    builder.addWhereClause("customerid=3");
```

- Instead of
  - String sql="select (*) from customer where customerid=3"
- Reduces chance of syntax error
  - Fool proofing code
- Allows builder to generate code for other syntaxes (transparent translation) "limit v. top"
- Allows builder to introduce new layers in database complexity, for example sharding, security layer

Sharding

- Splitting data over more than 1 database or database table, based on a key index
- Data can be divided based on
  - Key index range
    - (0-99 table 1, 100-199 table 2 etc)
  - Hash function on key index

Sharding example

```
public String getTableName() {
    if (customer_id!=0) {
        return tableName+”_”+customer_id/1000;
    } else {
        return tableName;
    }
}
```

 © University of Liverpool

Table validation

- Can debug to make sure table names are valid for the application

```
public void setTableName(String tableName) throws
BadTableNameException {
        if (!tableNames.contains(tableName)) {
            throw new BadTableNameException();


            };
        }
        this.tableName = tableName;
    }
```

More useful than run time SQL error, since error is caught earlier, better application control

© University of Liverpool

Multiton

- Like a singleton, but
  - Produces a different singleton instance dependent on a key
  - Example
    - You want a singleton class instance for a CRM (Customer relationship manager) for each customer

© University of Liverpool

Multiton code example

```
public class CRMHandler {
   private int customer_id=0;
   private int staffid=0;
   private java.util.Date lastContactTime;
   private static  Hashtable <Integer,CRMHandler> allHandlers =new Hashtable
<Integer,CRMHandler>() ;

   private CRMHandler(int customer_id) {      // private constructor
      this.customer_id=customer_id;
      System.out.println("Making handler for customer id "+customer_id);
   }

   public static synchronized CRMHandler getCRMHandler(int customer_id) {
      if (!allHandlers.containsKey(new Integer(customer_id))) {
         CRMHandler handler=new CRMHandler(customer_id);
         allHandlers.put(new Integer(customer_id), handler);
      }
      return(allHandlers.get(new Integer(customer_id)));
   }
}
```

Flyweight pattern

- Used to share memory allocation between objects with similar properties
- Example
  - A word processor could have a different font definition for each character
- Related to Multiton
  - Heavyweight resource will be often expressed a multiton

© University of Liverpool

Flyweight example

```
public class FontDefinition {
   private static  Hashtable <String,FontDefinition> allFonts =new
Hashtable <String,FontDefinition>() ;
   private String name="";
   private java.awt.Font font;
   private FontDefinition(String name) {      // private constructor
      this.name=name;
      // TO DO
      // Code to create Font natively is here see AWT documentation for
Java
      //
   }
```

Flyweight example

```
public static synchronized FontDefinition getFont(String
name) {
      if (!allFonts.containsKey(name)) {
         FontDefinition definition=new FontDefinition(name);
         allFonts.put(name,definition);
      }
      return(allFonts.get(name));
   }
}
```

Flyweight example

```
public class WPCharacter {
    private FontDefinition fontDefinition;
    private char letter;
    public void setFontName(String fname) {
        // Font definition is fly weight…
        fontDefinition=FontDefinition.getFont(fname);
    }

    public WPCharacter(char letter) {
        this.letter=letter;
    }
}
```

 © University of Liverpool

Chain of responsibility

- A number of classes work together to handle a message (call)
- If the class doesn't want to handle the message, it passes the messages down to next class in the chain
- Example
  - System logging

```
abstract class Logger {
   public static int ERR = 3;  // highest priorty message
   public static int NOTICE = 5;
   public static int DEBUG = 7;
   protected int logger_level;
   private static Logger lastLogger;
   // The next element in the chain of responsibility
   protected Logger next;

   public Logger(int level) {
      this.logger_level=level;
      setNext();
   }

   private void setNext() {
      if (lastLogger!=null) {
         lastLogger.next=this; // add this into chain
      }
      lastLogger=this;
   }
```


© University of Liverpool

```
    public void message(String msg, int priority) {
        if (priority <= logger_level) {
            writeMessage(msg);
        }
        if (next != null) {
            next.message(msg, priority);
        }
    }

    abstract protected void writeMessage(String msg);
}
```

```
class StdoutLogger extends Logger {
    public StdoutLogger(int logger_level) {
        super(logger_level);
    }

    protected void writeMessage(String msg) {
        System.out.println("Writing to stdout: " + msg);
    }
}


class EmailLogger extends Logger {
    public EmailLogger(int logger_level) {
        super(logger_level);
    }

    protected void writeMessage(String msg) {
        System.out.println("Sending via email: " + msg);
    }
}
```

Memento

- Used to restore object to previous state
- Features
  - Stores complexity of objects state
  - Does not allow external classes to view state
  - State is passed back to original object
- Classes
  - Memento stores the state
  - Originator, where the state comes from
  - Caretaker, handles the state, redo
- Application examples
  - Word processing, version control, financial

Memento example (bank account)

- Memento defined as inner class of originator class
  - Allows private sharing of data with originator
- In practise
  - All mementos should be persistent (stored to dbase)

Memento example

```
class BankAccount {
    class Memento {  // memento defined as inner class
        private String state="";
        public Memento(String state) {
            this.state=state;
        }
        private String getSavedState() {
            return(state);
        }
    }
```

Memento example

```
private long lastTransactionID=0;
  private Vector <Transaction> allTransactions=new Vector  <Transaction> ();

  public Memento saveToMemento() {
     System.out.println("Originator: Saving to Memento.");
     return new Memento(""+lastTransactionID);
  }

  public void doTransaction(String description,int amount) {
     lastTransactionID++;
     Transaction transaction=new
Transaction(amount,description,lastTransactionID);
     allTransactions.add(transaction);
  }
}
```

Memento example

```
public synchronized void restoreFromMemento(Memento memento) {
      lastTransactionID =Long.parseLong(memento.getSavedState());
      for (int idx=0;idx<allTransactions.size();idx++) {
         Transaction transaction=this.allTransactions.elementAt(idx);
         if (transaction.getTransactionid()>lastTransactionID) { // remove
transactions after id
            allTransactions.remove(idx);
            idx--; // move back one position, }
      }
     String sql="delete from account_transaction where
transactionid>"+lastTransactionID; // remove all
      // TO DO
      // EXECUTE SQL
   }
```

Double-checked locking

```
class DbaseConnector {  // standard locking....
   private static DbaseConnector instance = null;
   public static synchronized Helper getConnector() {
      if (instance == null) {
           instance = new DbaseConnector();
      }
      return instance;
   }
}
```

Problem

Synchronizing a method can decrease performance by a factor of 100 or higher

© University of Liverpool

Double-checked locking

```
public class DbaseConnector2 {
   private static DbaseConnector2 instance = null;
   public static DbaseConnector2 getConnector() {
      if (instance == null) {
         synchronized (DbaseConnector2.class) {
            instance = new DbaseConnector2();
         }
      }
      return instance;
   }
} // Look’s good but is faulty?  Why?
```

Double-checked locking

```
public class DbaseConnector2 {

    private static DbaseConnector2 instance = null;
    /**
     * This method has a subtle bug
     * @return
     */
    public static DbaseConnector2 getConnector() {
        synchronized (DbaseConnector2.class) {
            if (instance == null) {
                instance = new DbaseConnector2();
            }
        }
        return instance;
    }
}
```


© University of Liverpool