Introduction to Java

2. OOP and Java Code Structure

Object Oriented Fundamentals

- OO is based on three fundamental principles that have been evolving since the 1950s
- Iconicity
  - The idea that our programs are automating something “out there”
  - Our programs should look like what they are automating
  - The units of automation should make sense to a domain expert
- Recursive Design
  - The systems that we automate tend to be structured in layers or hierarchies
  - Our design process should be scale independent
  - As we can design the sub-systems of a system in the same way we design the system itself
- The Object Model
  - Our experience tells us that we perceive that systems are made up of objects in the real world
  - Then the most effective way to design an object is to have a similar sort of construct in our code

The Principle of Iconicity

- First proposed by Professors Ole-Johan Dahl and Kristen Nygaard at the Nordic Centre for Computing at the University of Oslo in the late 1950s
- The posed the question
  - *“Why should people have to learn how to interact with a computer? Why can we not design a computer program that resembles what it automates so that people can interact with it based only on what they already know?”*
- And developed the principle of iconicity
  - *Systems should look like what the automate so domain experts can use the system without training*
- This led to several corollaries
  - The separation of interface and implementation is essential
  - Designs should be based on domain modeling

Simula

- Dahl and Nygaard implemented these ideas in the Simula programming language in 1961-5
- The Simula code tanspiled to Algol which was then compiled
- The code ran on a UNIVAC 1107 which had
  - 256K of memory
  - Tape drive storage
  - 6MB of disk space
  - Accessed via punch cards and a line printer

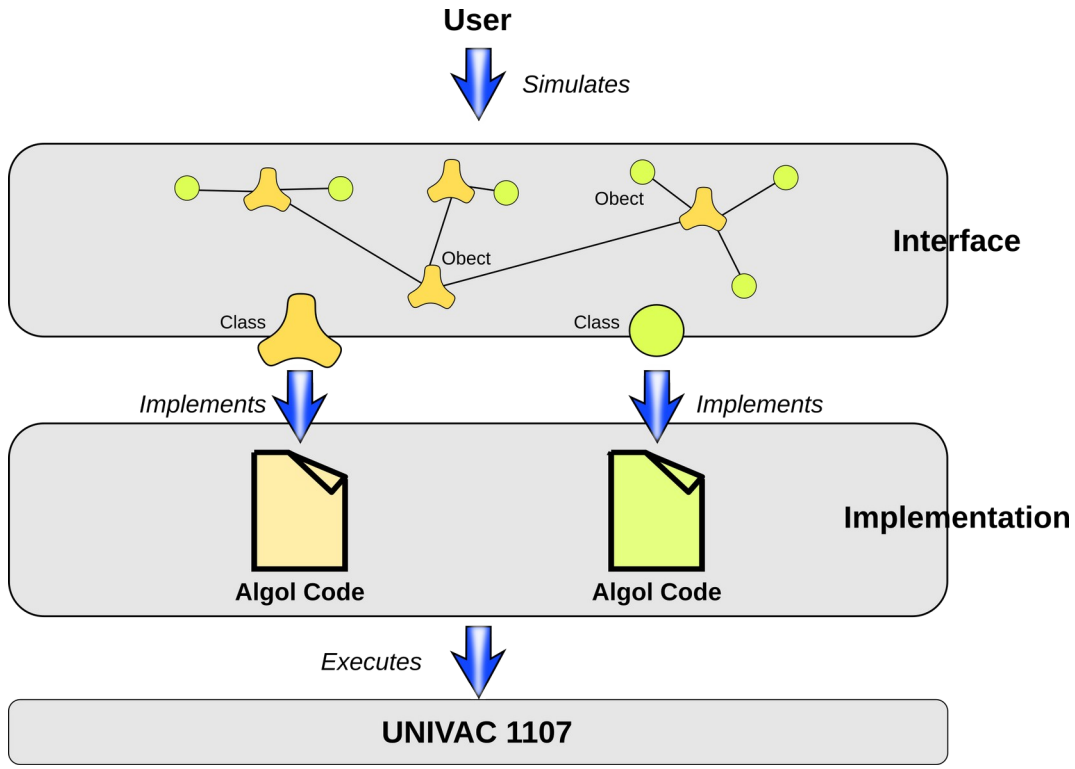
Simula Code

- The structure of a Java class definition is derived ultimately from the Simula definition
- This is an implementation of the object model
  - The class definition has a type, attributes and methods (called procedures)
  - We create objects by instantiating them from the class definition
- The code shows a class representing a geometrical point located at (X,Y)
  - ROTATED(P,N) Rotates this point about the point P by N degrees,
  - EQUALS(P) checks if this point and P are the same geometrical point,
  - DISTANCE(P) return the distance between this point and the point P.
  - Each time a point is created, R and THETA are calculated from the actual parameter values of X and Y.

```
CLASS POINT(X,Y); REAL X,Y;
    BEGIN REAL R, REAL THETA;
        REF(POINT) PROCEDURE ROTATED;
        BOOLEAN PROCEDURE EQUALS;
        REAL PROCEDURE DISTANCE;
        R := SQRT(X ↑ 2 + Y ↑ 2)
        THETA := ARCTAN2(X,Y)
    END
```

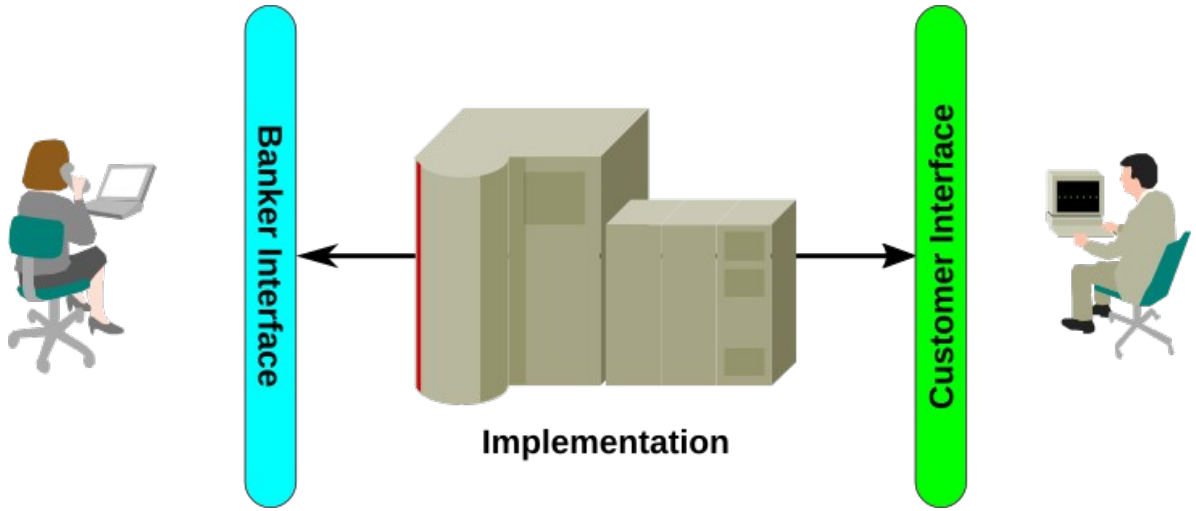
Interface and Implementation

- The class constructs in Simula provided an iconic interface that users could interact with
  - But the code could not be executed
- It had to be turned into executable code
  - This is the implementation
  - Implementation code is *not* iconic
- Keeping these two layers separate
  - Allows for different implementations
  - FORTRAN on and IBM/360 for example
  - The implementation is never iconic
  - It's a black box except to the developer
- This is called the decoupling of interface and implementation



Interface and Implementation

- This idea is fundamental to modern Java design
- Consider a bank application
  - There are two domain that conceptualize a bank account, and they are not the same
  - A banker's view of a bank account is not the same as the customer (bankers love debits and credits)
  - Neither view is the implementation
  - The bank account is a record in an SQL database
- There may be multiple interfaces
  - We can change interfaces without changing the implementation
  - We can change the implementation without changing the interfaces
  - The actual implementation is a black box to the banker and the customer (but not the DBA)



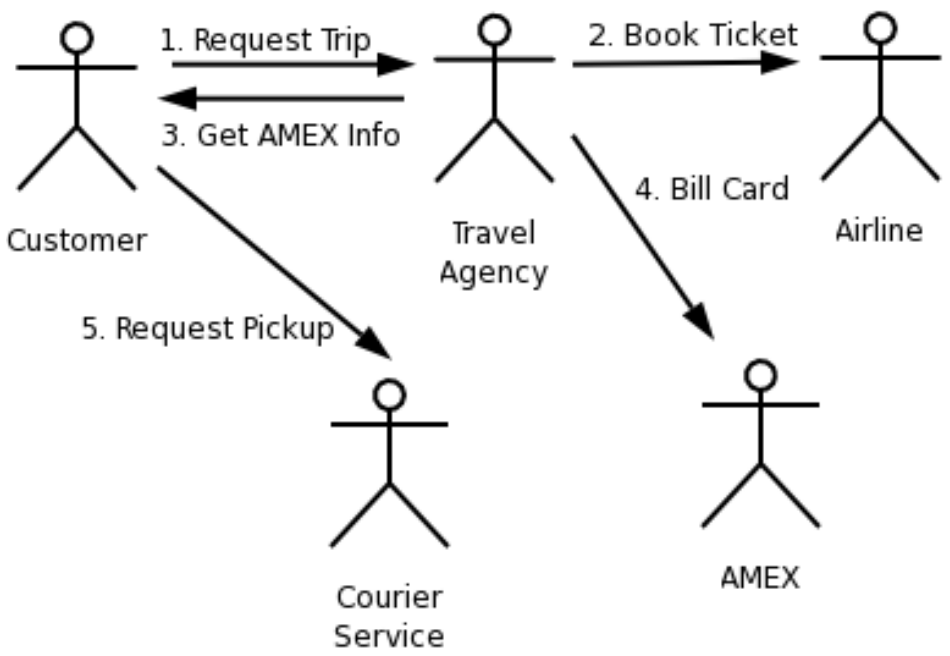
Interfaces as Views

- One useful way to think of interfaces is that they are views into the underlying implementation
  - A view shows only what is of interest to the viewer
  - A view may translate something in the implementation into a representation unique to that view
- Multiple interfaces provide different views into the same underlying implementation
  - In standard database development, these interfaces are called.. wait for it... views
- The challenge of developing interfaces that are both useful and realistic will be a topic we will touch on later in the course
  - *“We don't see things as they are, we see them as we are.”* Anias Nin
  - *“What you see is not reality but perspective, what you hear is not truth but opinion.”* Anonymous

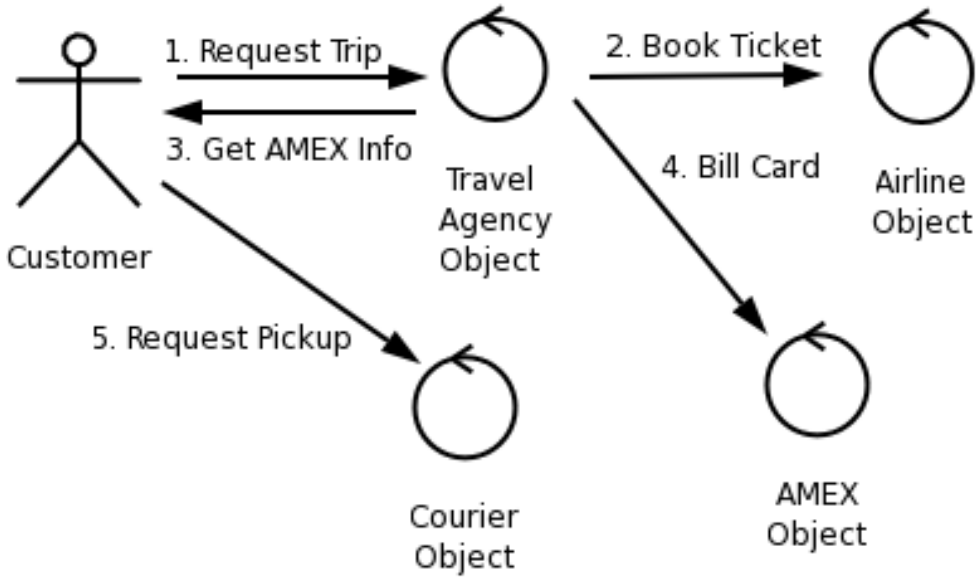
The Network Problem

- Structured programming does not have an easy solution to the problem of distributed agent systems which are characterized by specific properties:
  - They are distributed
    - *There is no central CPU or processing node*
    - *Each agent has its own processing capability.*
  - They are concurrent
    - *The processing done by each agent proceeds independently of the processing in any other agent.*
  - They collaborate
    - *Agents work together by collaborating to accomplish tasks*
    - *This collaboration takes place by the sending of messages back and forth.*
  - They are heterogeneous
    - *The agents that make up a system do not all have to be of the same kind. As long as they can send and receive the appropriate messages, they can be of any type*
- These sorts of systems describe many types of networks we may want to model
  - Human work teams, financial transactions between companies, biological processes, computer networks etc

The Network Problem

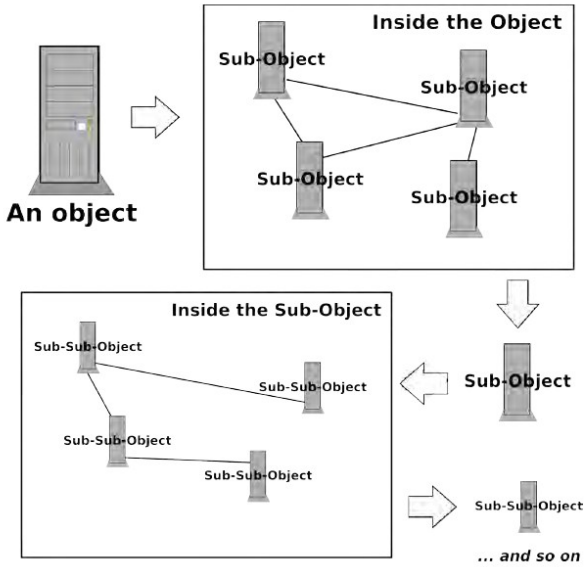


The manual system of people talking to each other – travel agents, airline representatives, courier dispatchers

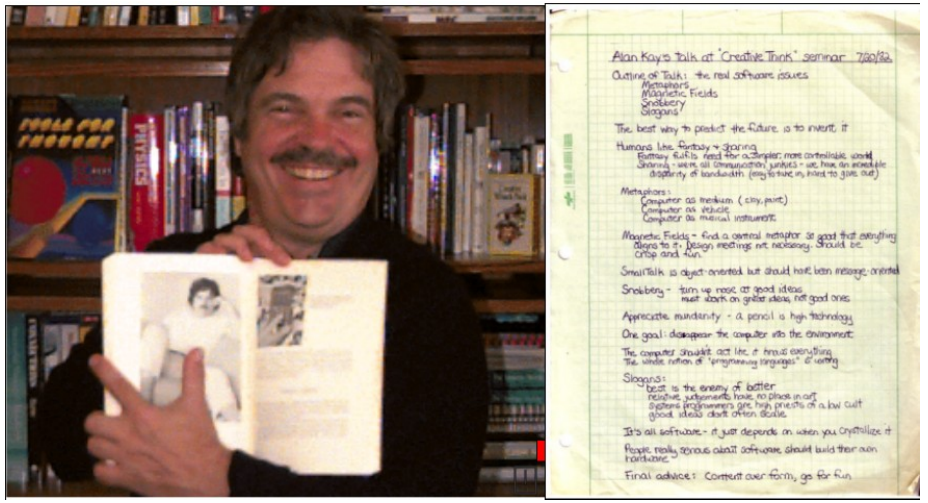


The system we want to build where the entities communicate directly instead of through human interfaces

Recursive Design



1. A system is built up in layers
2. Each layer is itself an object
3. Each layer is made up of a collection of peer objects which provide the functionality of that layer
4. There is no restriction to the types of objects that exist within a particular layer



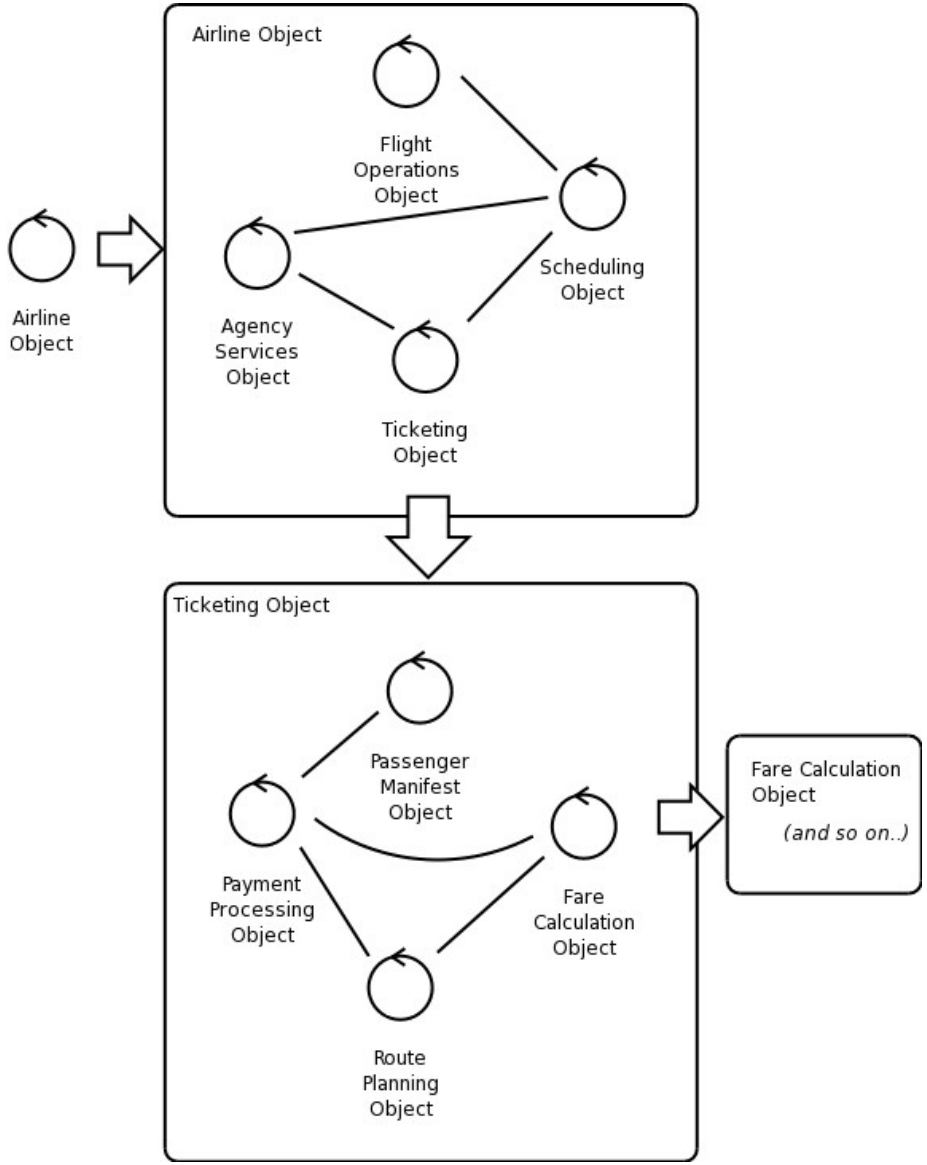
I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages (so messaging came at the very beginning – it took a while to see how to do messaging in a programming language efficiently enough to be useful)

Instead of dividing a system (or computer) up into functional subsystems, we think of it as a being made up of a collection of little processing engines, called objects. Each object will have similar computational power to the whole and processing happens when the objects work together. However, and this is the recursive part, each object can then be thought of in turn as a collection of sub-objects, each with similar computational power to the object, and so on.

Alan Kay

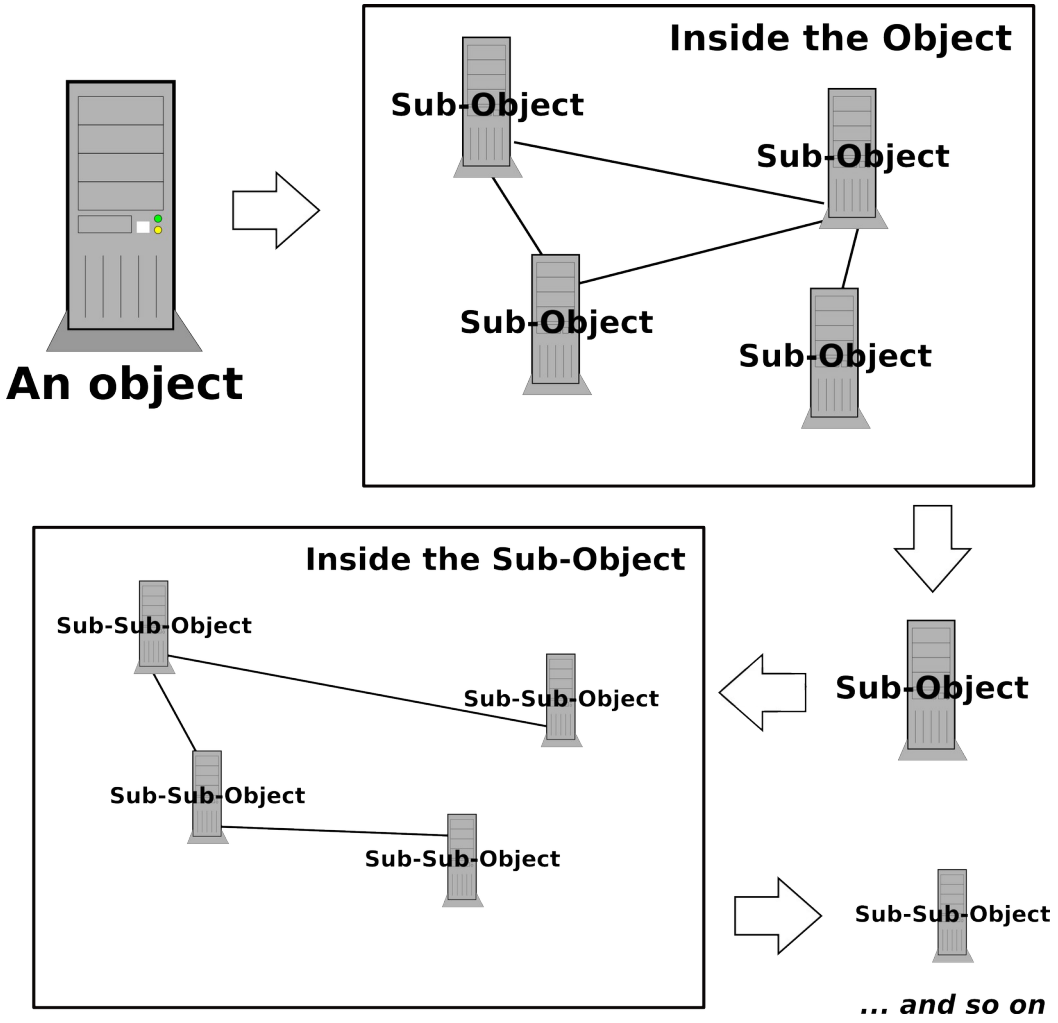
Recursive Design

- Consider the airline object from a previous slide
  - It actually is made up of a hierarchy of networks of smaller objects
- This is a complex system made up of layers of sub-systems of agents…
- This is characteristic of
  - Companies and organizations
  - Biological systems
  - Mechanical systems etc



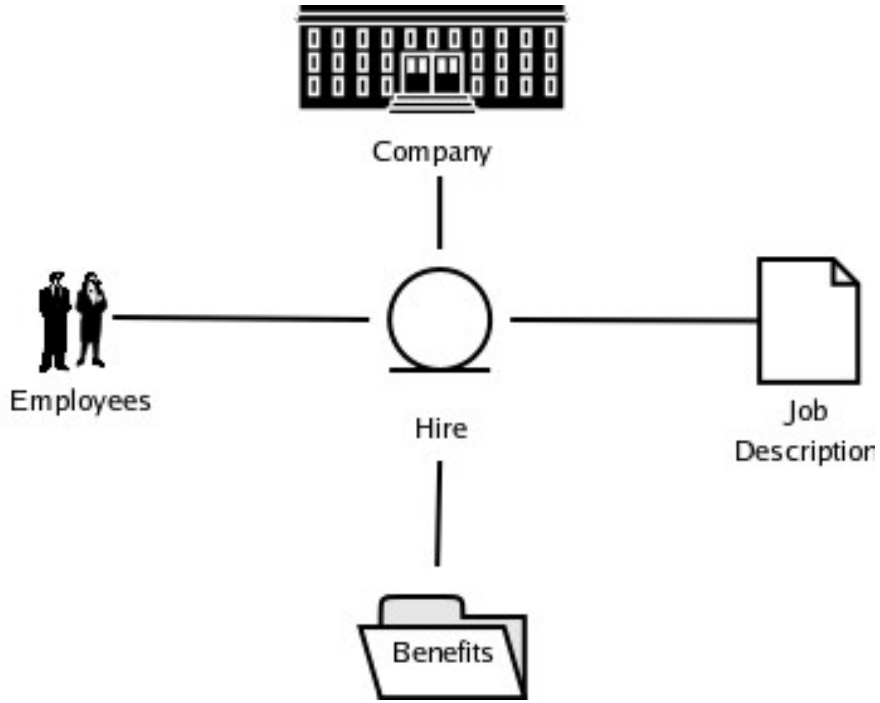
Recursive Design

- Alan Kay suggested that an object oriented approach is the best way to model and to design and develop these systems
  - Developed his principles of OOP
  - Java and other OO languages are designed to support these principles even though Kay thought they weren't OO enough
- The principles of OOP address the recursive nature of these systems



OOP #1

- **Everything is an object**
  - Everything can be represented as an object
- Actions and transactions are objects
  - Being born is a birth
  - Buying something is a purchase
  - Selling something is a sale
- Relationships between objects can be objects
  - Two married people is a marriage
  - A work relationship may be a job
- What this means is that the only basic construct we need in a OO language is a way to create objects

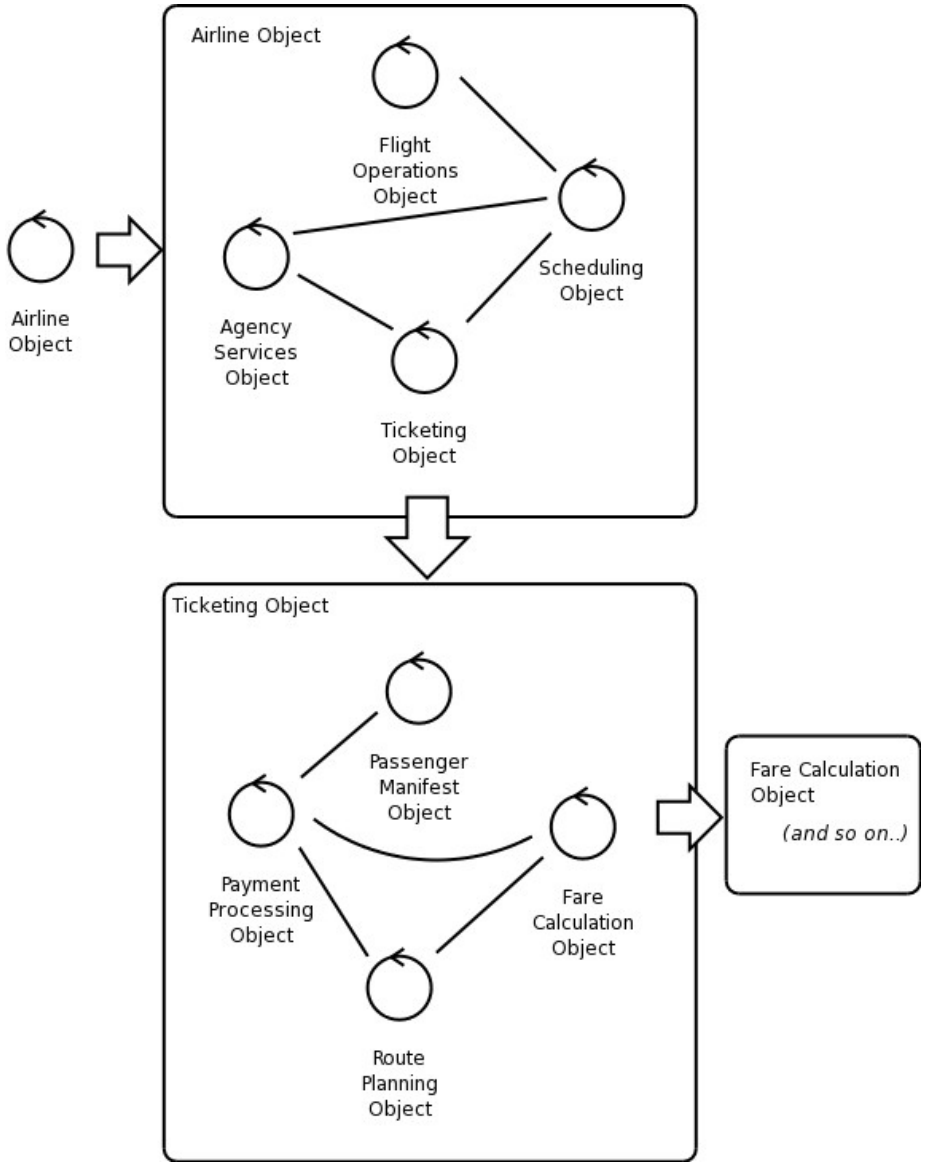


OOP #2

- **Systems are collections of objects collaborating for a purpose and coordinating their activities by sending messages to each other**
  - Systems tend to be gestalts – the system is more than the sum of the objects that make it up
    - *The “more” is the layer of organization that coordinates the objects*
    - *We often see a phenomenon called emergent behaviour in systems when the system displays behaviours that are not present in the individual objects that make up the system*
- However, by OOP#1, this system can be treated as an object
  - A programmer is an object
  - A group of programmers that work together on a project are a team object
  - A group of teams that work together on projects is an R&D department object

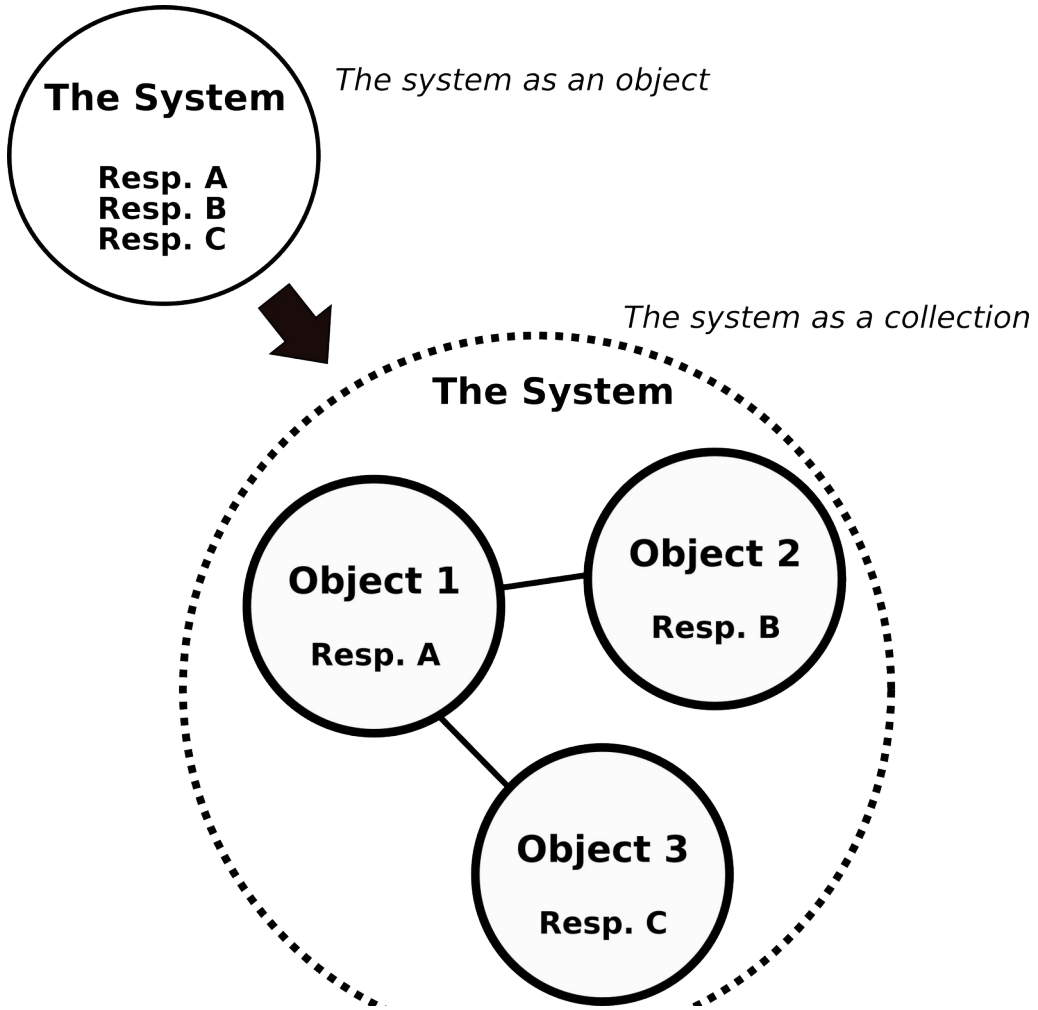
OOP #3

- **An object can have an internal structure composed of hierarchies of sub-objects.**
  - An object is a system (OOP #2)
  - The functionality of the object is delegated to sub-systems
  - The sub-systems themselves are objects (OOP #1)
- The first three axioms provide a framework for designing both an OO language and specific application designs that are exhibit this recursive property.



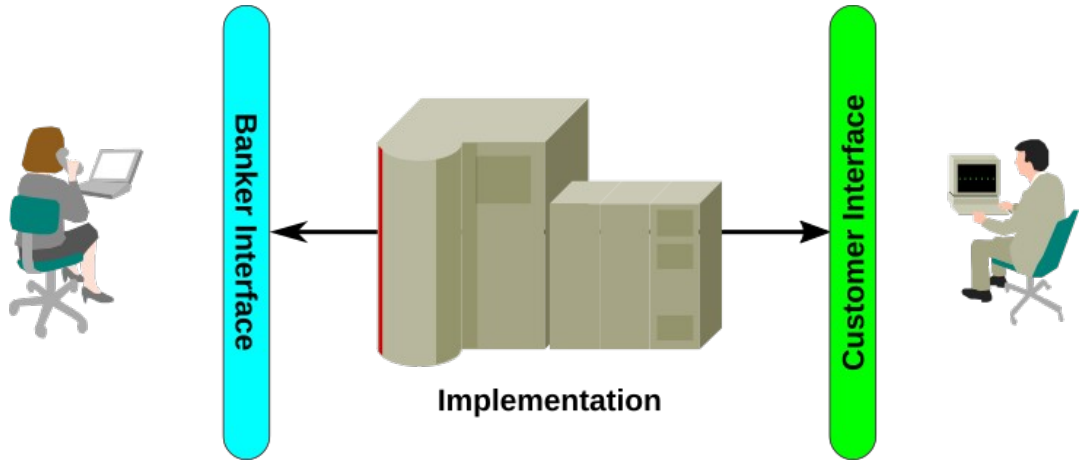
OOP #4

- **Objects within a system have individual responsibilities**
  - The responsibility of a system as a whole is distributed across the objects that make up the system by delegating specific responsibilities to individual objects
- This axiom can be summarized as
  - Each object has one responsibility or specialization
- This an OO version of several well known design principles in both software and engineering
- When a responsibility can be decomposed into sub-responsibilities
  - Each sub-responsibility can be assigned to a sub-object that specializes in that sub-responsibility
  - This is often called a functional decomposition



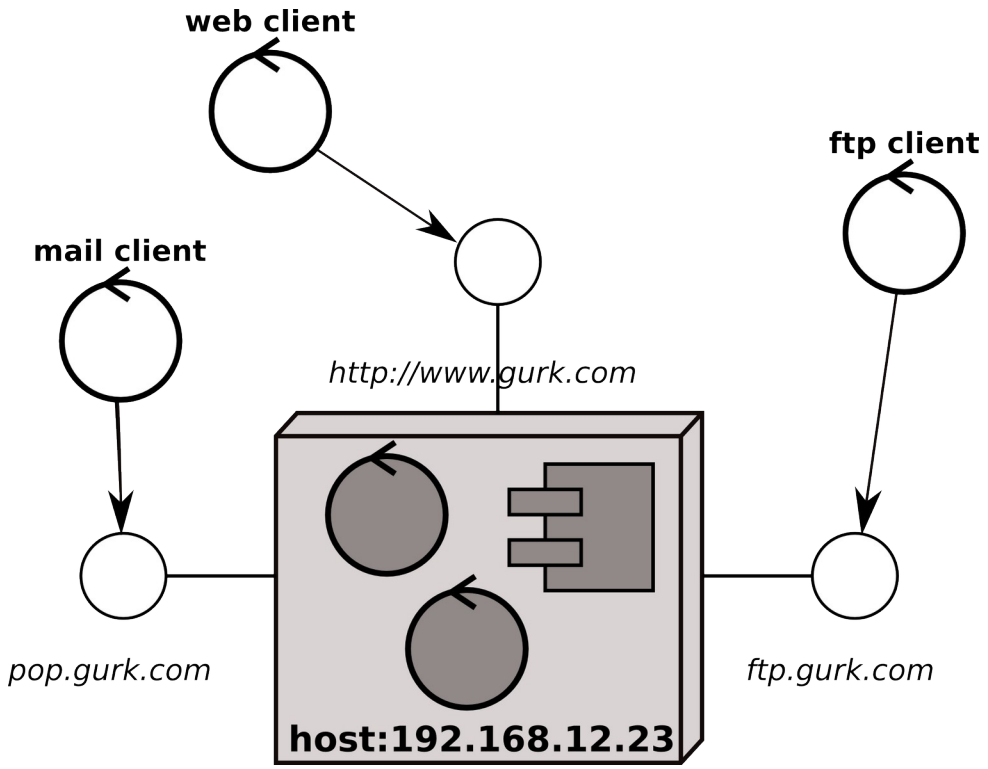
OOP #5

- **An object presents an interface that specifies which requests can be made of it and what the expected results of those requests are**
  - The interface is independent of the actual internal workings of the object
  - The interface presented by the object is often referred to as the object's "contract"
- An interface is what is visible to the other objects
  - It can be thought of as a list of the messages that the object understands
  - And a description of how the object will respond as a result of receiving a particular message



OOP #6

- **An object is of one or more types**
  - A type includes a role the object plays, a set of responsibilities and an interface
  - Each type may be derived from a hierarchy of types.
- This ties together the ideas of roles, interfaces and responsibilities under the concept of “type”
  - This axiom is unique because it does not talk about structure
- We don’t try to define what “type” means
  - Rather we note that it is a way to the ideas of the role objects play in a system, the interface associated with that role and the responsibilities of that role.
- An object that has more than one type is said to be *polymorphic*




Questions?

Java Packages

- Java packages combine two concepts
  - A directory like structure for organizing Java classes
  - Namespaces to avoid naming conflicts between classes in different package
- Packages as structure
  - A package can be thought of as a directory in a file system
  - It has a name
  - It can contain various Java constructs like class definition
  - But it can also contain other packages which enables a recursive organization
- Package *are* directories
  - Java is intended to be portable across file systems
  - Packages are an abstraction that is implemented into the local files system by the JRE
  - Dots are used as a directory structure
    - com.accounting.payroll → com/accounting/payroll or com\accounting\payroll

The package Statement

- There may be many files in a single package
- There is no analogue to a directory table in a package
  - This is very OS dependent
  - Trying to have the package index or keep track of its contents is not technically feasible
- Instead, each file remembers which package it should be in
  - This is the *package* statement which is the first line in every file
- There is a default unnamed package where files without package declarations go
  - This is only used for small quick and dirty code
  - Using it consistently is considered poor Java style

```
package com.mycorp;

public class Boot {

    public static void main(String[] args) {
        System.out.println("Welcome to MyCorp");

    }

}
```

The Bootstrap Problem

- All programs have to start somewhere, often called the bootstrap code
- Most compiled programming languages have a main() function that represents the entry point to the program execution
- The problem is that Java only allows methods (functions) inside classes
  - That means that we have to stick a main() method is some class
  - Java doesn’t care where it is, as long as it can find it
- This is problematic for several reasons
  - Putting the main() method in an arbitrary location makes it hard to find
  - Whatever class it is in now has an additional responsibility – starting the application
- Best practice
  - Create a special class with only one responsibility – to run the main() method
  - And put the class in a specific location, the topmost package for example, so that it does not pollute the rest of the cod

Demo

Working with Java Packages


Lab 2-1
Java Packages

Package as Namespace

- For this discussion we are going to only be referring to class definitions
  - This is just to illustrate the concept of visibility
  - We will refine this later when we work with classes in more detail
- Java uses *package visibility* by default
  - This means that every class in a package can refer to every other class in the same package
- Some classes can be declared to be *public*
  - This means that classes outside the package can also refer to it
- Java has a couple of special rules for public classes
  - There can only be one public class defined per file
  - The file has to have the same name as the public class
  - This makes it easier for Java to manage

Fully Qualified Class Names

- When code in one package refers to a public class in another package
  - Java has to find that definition.. somewhere
  - There is no index so just using the class name alone is pointless
- One way to help Java out is to prefix the class name with a full qualified name which is done by adding the package name as a prefix to the class name
  - This is like using a fully qualified path name in a file system

```
2  package com.mycorp;
3
4  public class Boot {
5
6      public static void main(String[] args) {
7          // can't find class
8          Coder kent = new Coder();
9          // Fully qualified class name
10         com.mycorp.dev.Coder anish = new com.mycorp.dev.Coder();
11
12     }
13
14 }
15
```

The import Statement

- Since it's a pain to use full qualified names, using the *import* statement makes easier
  - The full qualified class name is placed after the package statement in an *import* statement
  - This directs Java where to look to find the definition of the imported class
  - The import statement doesn't move anything, just allows Java to use the class name as a alias for the imported fully qualified name
- If there are a number of classes to be imported from a package the wildcard can be used instead of listing all the classes to be imported.

```
package com.mycorp;
import com.mycorp.dev.Coder;
// or import com.mycorp.dev.*;

public class Boot {

    public static void main(String[] args) {
        Coder kent = new Coder();
    }

}
```

Naming Conflicts

- It may may happen that an imported class name may conflict with another class
  - There may already be a class with that name in the package
  - There maybe two different classes with the same name being imported from two different packages
  - This is a name space collision
- In this case, to avoid ambiguity, one of the imported classes will have to use its full qualified name
- Notice in the example, the first import had to be removed
  - Java is only concerned with import statements when looking for collisions

```java
package com.mycorp;
import com.mycorp.dev.Coder;
import com.mycorp.dev.backend.Coder;

public class Boot {

    public static void main(String[] args) {
        Coder kent = new Coder(); // ??? Which coder??
        Coder bjarne = new Coder(); // again.. which"
    }
}
```

```java
package com.mycorp;
import com.mycorp.dev.backend.Coder;

public class Boot {

    public static void main(String[] args) {
        com.mycorp.dev.Coder kent = new com.mycorp.dev.Coder();
        Coder bjarne = new Coder(); // again.. which"
    }
}
```


Questions?

Demo

Namespaces and Importing

Lab 2-2

Namespaces and Importing


Java™