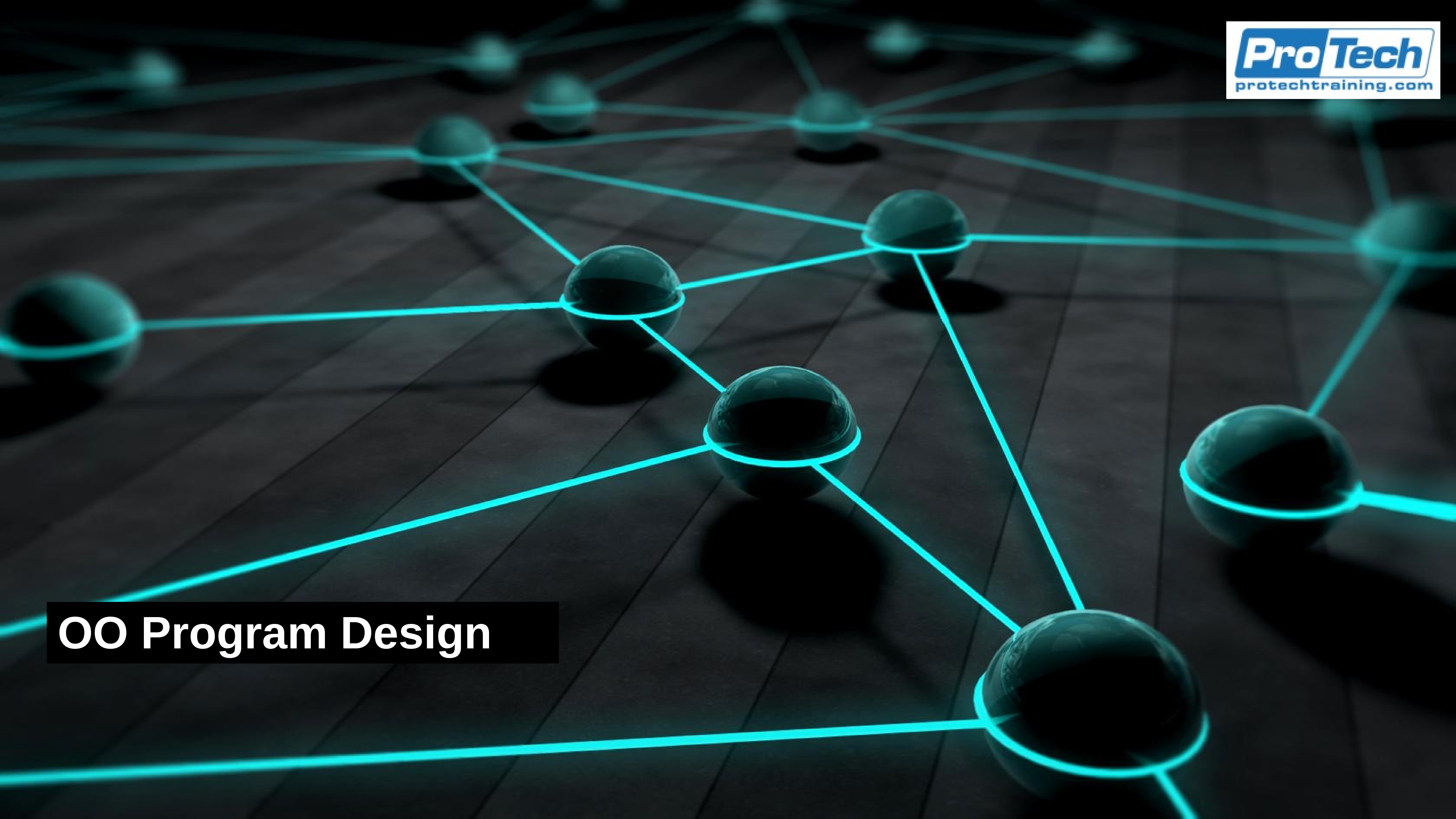


OO Program Design

Object Oriented Fundamentals

- Originated in the 1960s
  - First OO programming language was Simula in 1961
- Emphasized usability of interfaces
  - Decoupled from underlying design
  - Based on domain objects
- Software should have iconic user interfaces
  - Requires an in depth understanding of what users know and do in the domain

"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?"

Professors Ole-Johan Dahl and Kristen Nygaard

Object Oriented Fundamentals

- Based on three basic ideas
- Iconicity
  - Users should be able to use computer applications without training
  - They should be able to use an application based only on their domain knowledge
  - Applications should resemble what they automate
  - For example, accountants should see ledgers, journals, transactions etc.
  - This was revolutionary in the 1960s, but is mainstream today
  - The goal was to make computer applications usable for users in a domain

Object Oriented Fundamentals

- The Principle of Recursive Design:
  - Until the 1980s, almost all computing was done in monolithic mainframe environments
  - The structured approach was ideally suited to development in the sort of hardware and computing environments that existed through the 1960s to the early 1990s
  - However, as distributed computing became increasingly common, the structured paradigm was found to be difficult to use in distributed and networked environments
  - These new complex systems were built up in layers and were made up of independent nodes of computation
  - The principle of recursive design became an alternative approach for designing and coding applications in this new environment

Object Oriented Fundamentals

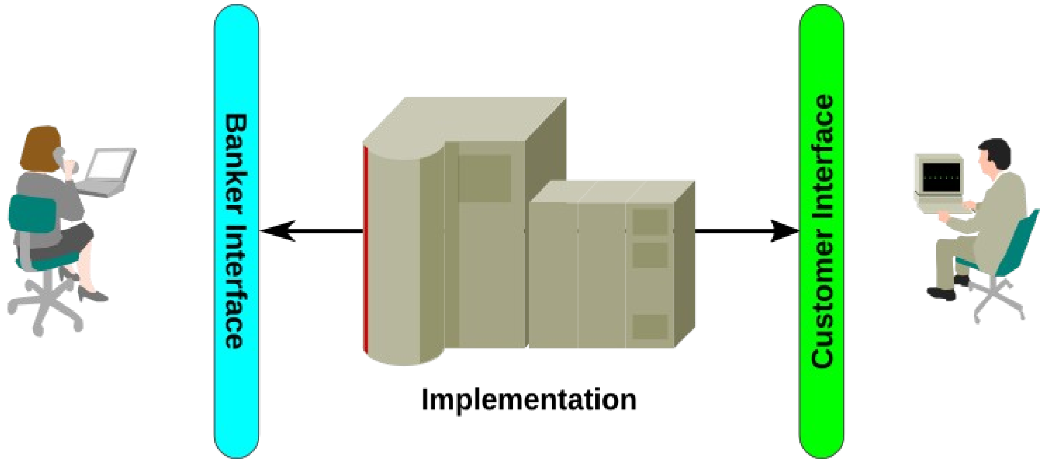
- The Object Model
  - The object model are modeling concepts about the fundamental units that make up a system
  - Models a system as a collection of objects working together
  - Sets out a template for objects and types, their behavior and properties and how they interact

Domain Modeling

- To have iconicity we need to show the user an iconic representation of their domain
  - That means we have to understand how their domain appears to them
  - People tend to model their domain cognitively in terms of types and objects
  - This forced OO to develop the object model.
- It turns out that the object model is a good enough approximation to how people think about their domain to be workable
- This also means having two layers to our code
  - The interface where users interact with their iconic domain objects
  - The implementation, which is the translation of the interface activities into executable modules
  - There is no requirement for the implementation to follow the object model

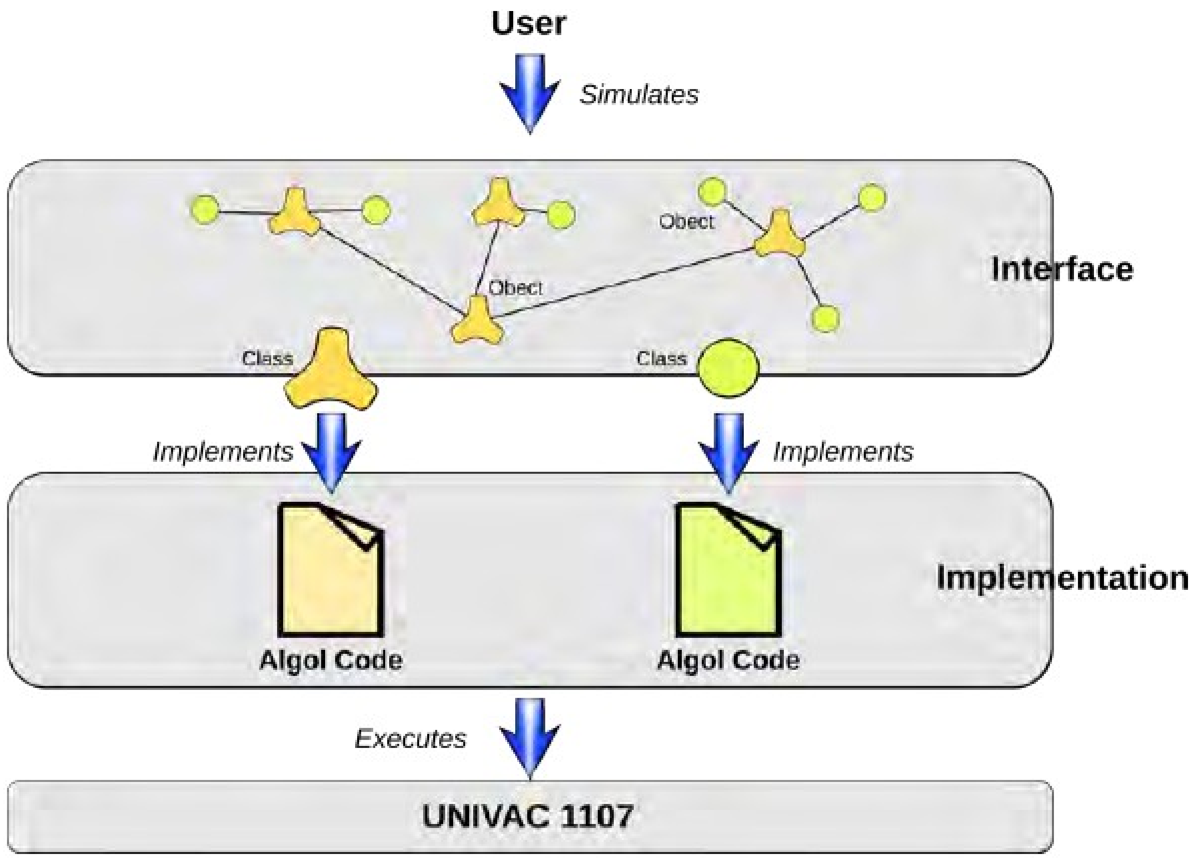
Interface and Implementation

- Consider a banking application
  - The users need to be able to interact with their account objects, which are depicted for them in the interface
  - However, their accounts are just records in a relational database, the implementation
  - The system translates the interface operations into database queries so users need to know nothing about the implementation
  - This also allows for changing either the interface or implementation independently (loosely coupled)



Interface and Implementation

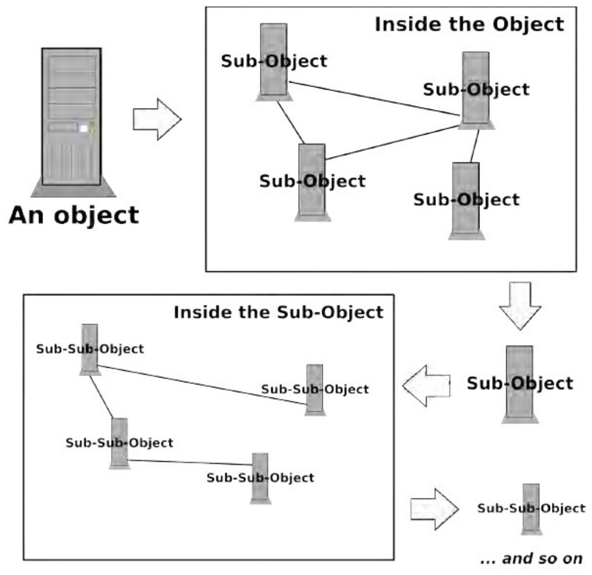
- This is the original design of Simula
  - Users interacted with the interface by writing Simula code
  - For example, working with geometric shapes in a drafting package
  - The Simula code is then compiled into ALGOL code (a structured language)
  - Then the ALGOL code is run on the mainframe



Principle of Recursive Design

- Developed by Alan Kay
  - Liked the OO model but it provided no way to automate layered complex systems
  - He looked at the way natural systems were organized
  - Similar to what we saw in the section on engineering design principles
  - Organizing a system into layers and each layer delivering its functionality by objects
  - Each object can have an internal structure of sub-objects
- Notice that there is a lot of similarity with other ideas
  - For example, structured programming functional decomposition and modularization
  - But OO reworked these ideas and added more to manage complex systems of independent agents

Principle of Recursive Design

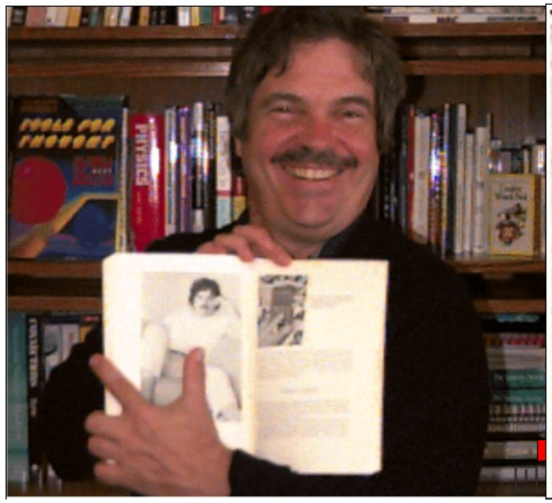


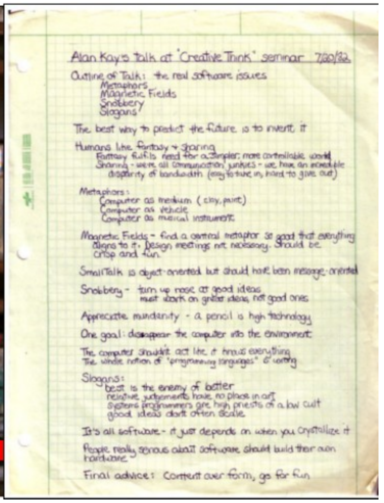
1. A system is built up in layers

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





Alan Kay's talk at "Creative Think" seminar 7/20/82

Outline of Talk: the real software issues
Metaphors
Magnetic Fields
Snobbery
Slogans

The best way to predict the future is to invent it

Humans like fantasy + sharing
Fantasy fulfils need for a simpler, more controllable world
Sharing - we're all communication junkies - we have an incredible disparity of bandwidth (easy to take in, hard to give out)

Metaphors:
Computer as medium (clay, paint)
Computer as vehicle
Computer as musical instrument

Magnetic Fields - find a central metaphor so good that everything aligns to it. Design meetings not necessary. Should be crisp and fun.

SmallTalk is object-oriented but should have been message-oriented

Snobbery - turn up nose at good ideas
must work on great ideas, not good ones

Appreciate mundanity - a pencil is high technology

One goal: disappear the computer into the environment

The computer shouldn't act like it knows everything
The whole notion of "programming languages" is wrong

Slogans:
best is the enemy of better
relative judgements have no place in art
systems programmers are high priests of a low cult
good ideas don't often scale

It's all software - it just depends on when you crystallize it

People really serious about software should build their own hardware

Final advice: Content over form, go for fun

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.

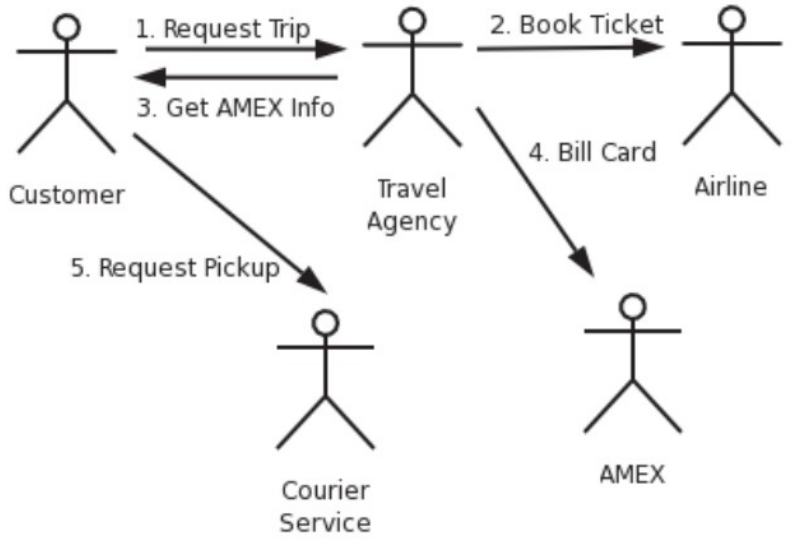
Alan Kay

Agent Systems

- Agent systems have several basic properties:
- They are distributed
  - There is no central CPU or processing node, rather each agent has its own processing capability
- They are concurrent
  - The internal processing done by each agent proceeds independently of the internal 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

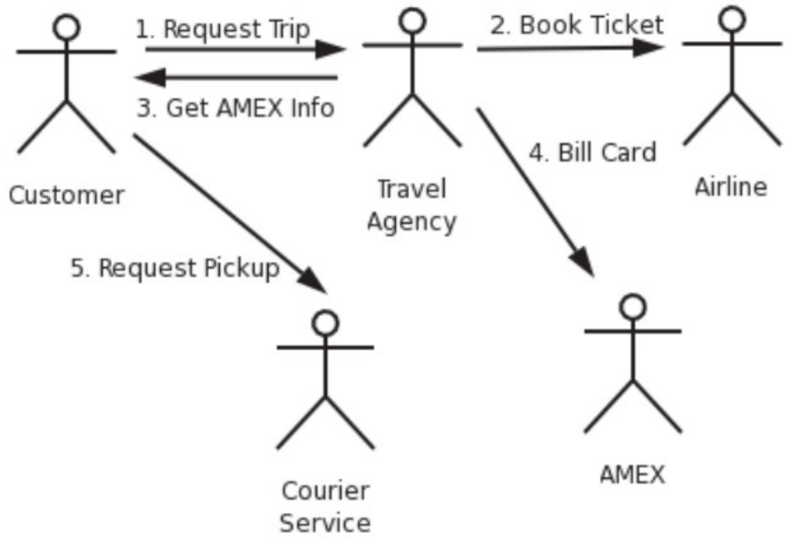
Agent Systems

- Buying a plane ticket from a travel agent
- The agents are:
  - You, the customer
  - the travel agent
  - the credit card company,
  - the airline
  - the courier service



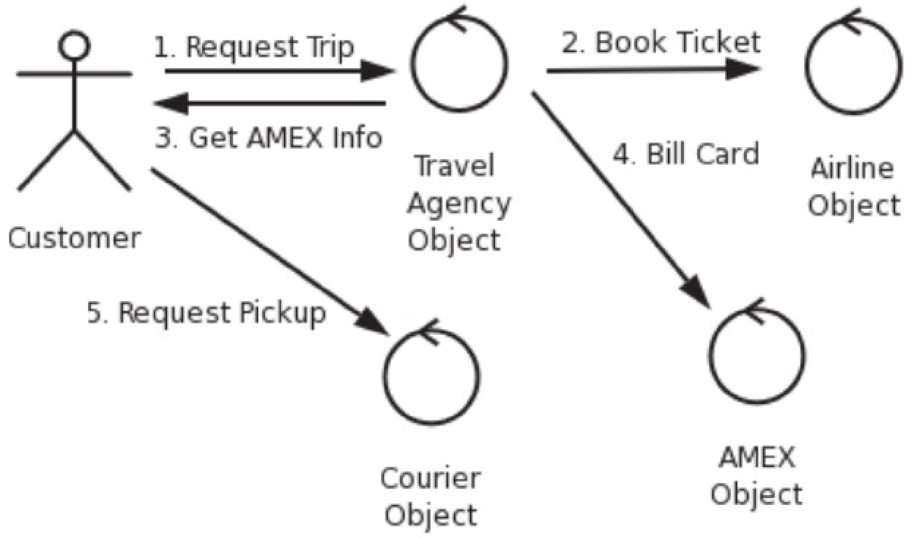
Agent Systems

- The agents collaborate in the following way:
  - You call your travel agent and ask for a ticket to London
  - The travel agent calls the airline and books the ticket
  - The travel agent calls you back and gets your credit card number
  - The travel agent calls American Express and gets the transaction approved
  - You call a courier to pick up the ticket
- You do not need to know how any of these agents process the messages they receive
  - Only that they can receive them and provide an appropriate response



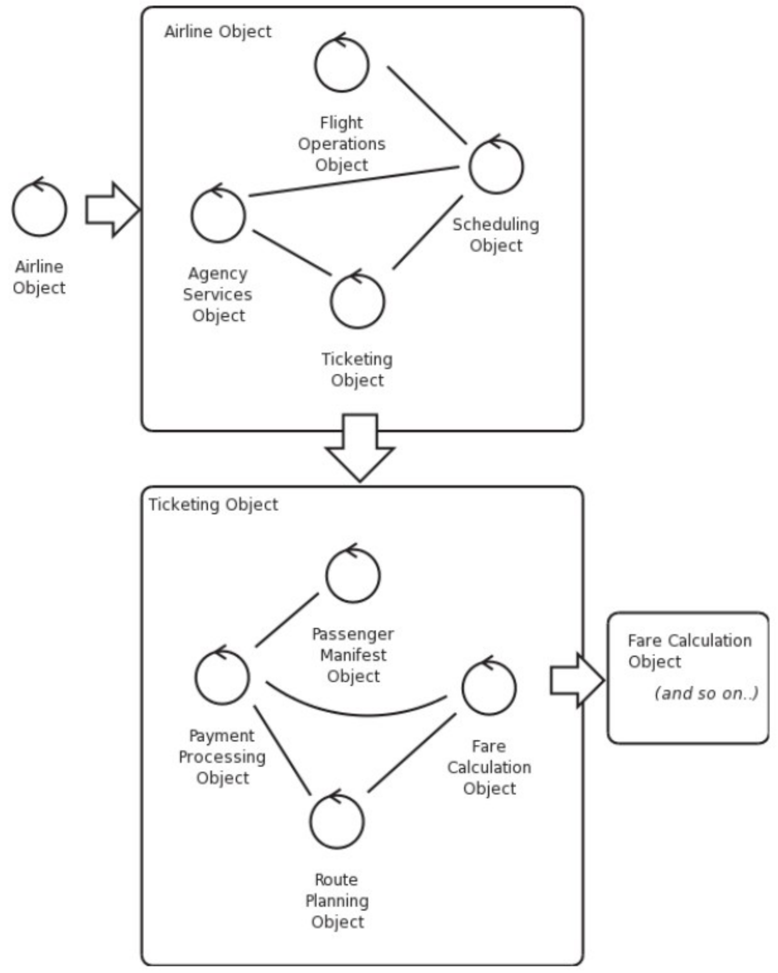
Agent Systems Automation

- In this example, we replace the humans with automation
  - The same messages still flow
  - This is often important because the type and content of the message has critical business importance
- The objects now receive the message
  - Then route it to the correct internal objects for processing.
  - We model the interfaces after the messages in the original system
    - This will be covered in more detail in the API section



Agent Systems

- The airline object is built up in layers
  - An example of recursive design
- When receiving a reservation
  - The public facing interface has to route the request to the correct layer
  - The layer will then delegate it to an object to be processed
  - Generated responses will make their way back to the interface and back to the external client
- These concepts will recur in later sections on micro-services

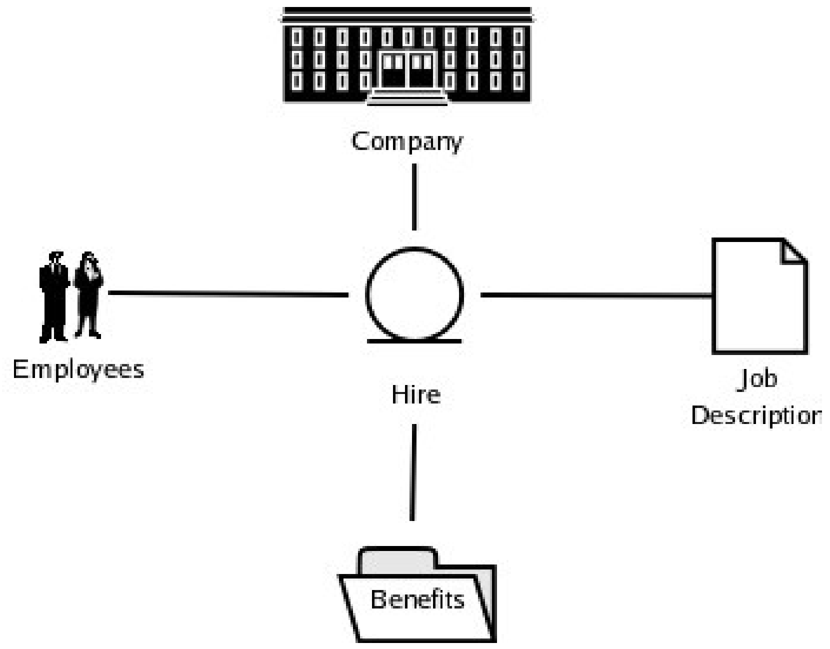


Axioms of OO Development

- Derived from Alan Kay's six principles of object-oriented programming.
  - Kay's original principles were very tied to SmallTalk, but these are more generic
  - They provide guidelines for organizing the data and functionality into coherent packages and classes
  - They provide guidelines for deciding how what is needed in terms of the interfaces and interrelationships, both delegation and inheritance, between classes
  - They provide a basis for developing a program or application architecture
  - They are the motivation for specific language features in many object-oriented programming languages
  - These are consistent with all of the best practices we have been studying

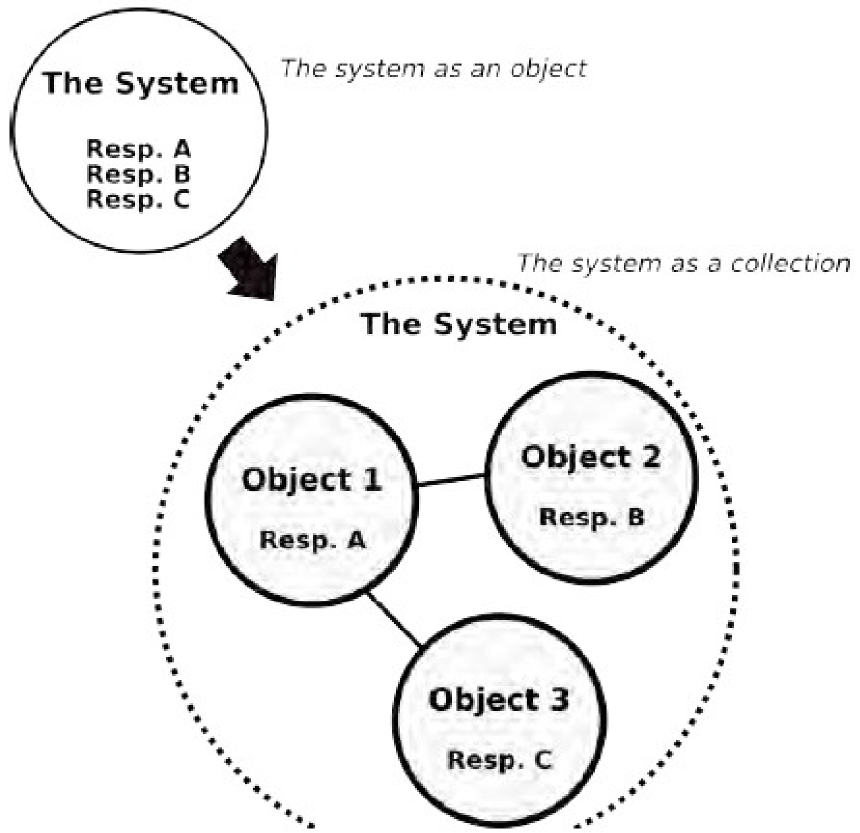
OO Programming Design Principles

- One: Everything can be modeled as an object
  - This is basically what people do
  - Not saying everything is an object, but rather we can represent anything as an object in our design
  - For example, events or relationships that are important are modeled as objects
    - My work. My marriage.
- An important type of object is a transaction object
  - Created when some event happens that links together other objects
  - A hire is an event that links together a company, a job description, a benefits package and a person



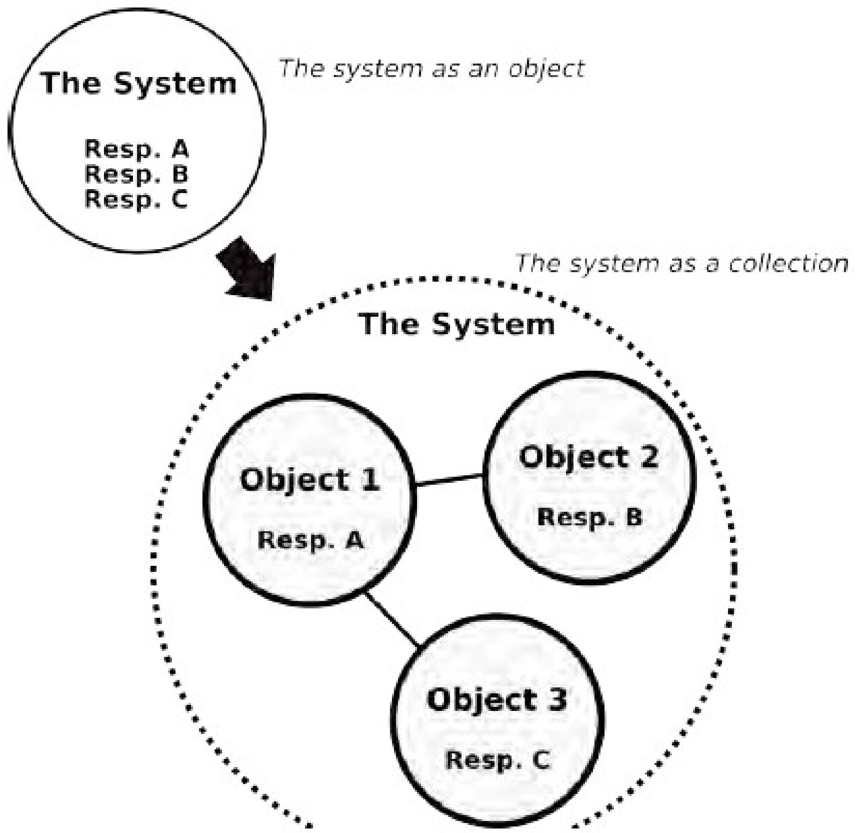
OO Programming Design Principles

- Two: Systems are collections of objects collaborating for a purpose and coordinating their activities by sending messages to each other
  - Coupled with our first axiom this also means we can consider any system to be an object
  - This is derived from the principle of recursive design



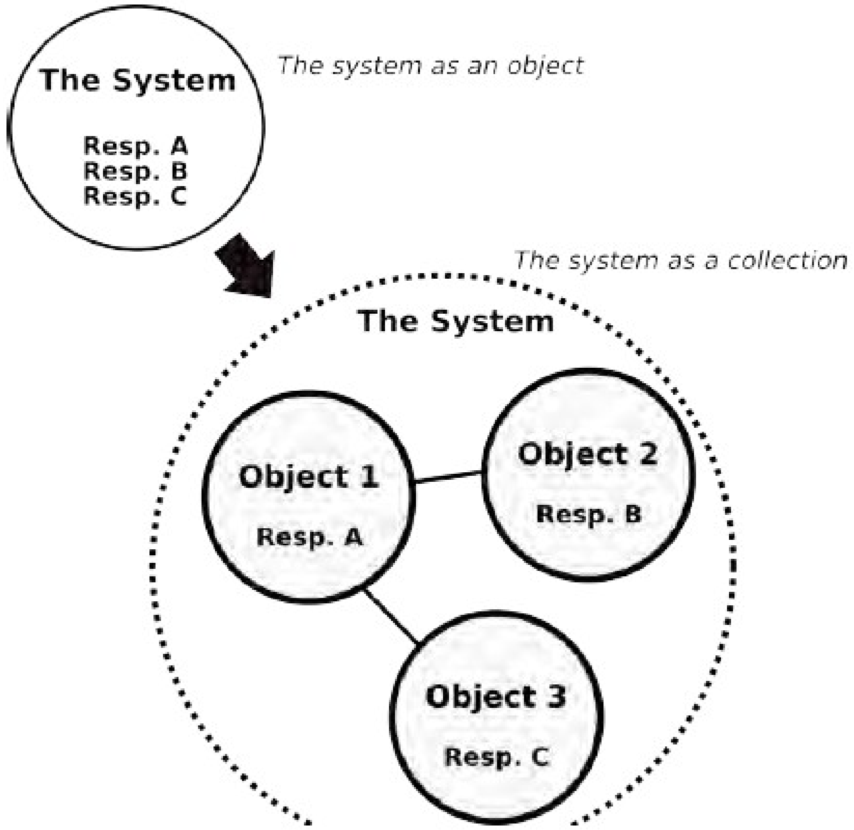
OO Programming Design Principles

- Three: An object can have an internal structure composed of hierarchies of sub-objects
  - This is the converse of the last axiom
  - This means that we can essentially zoom in and out when looking at a layered system
  - The idea of a system and object are interchangeable – it just depends on the scale we are looking at



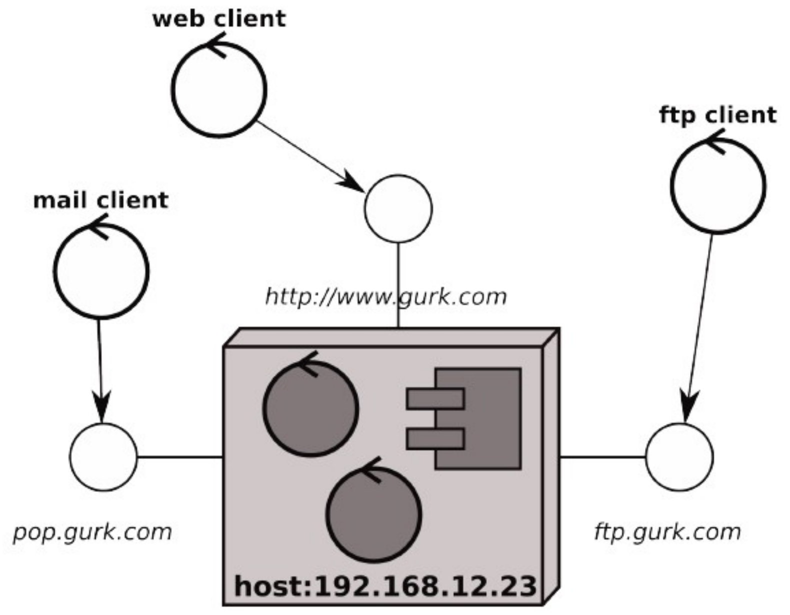
OO Programming Design Principles

- Four: 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 is also called a functional decomposition
  - It is the same concept we saw in the structured programming section
  - But it is reformulated to fit into the OO paradigm



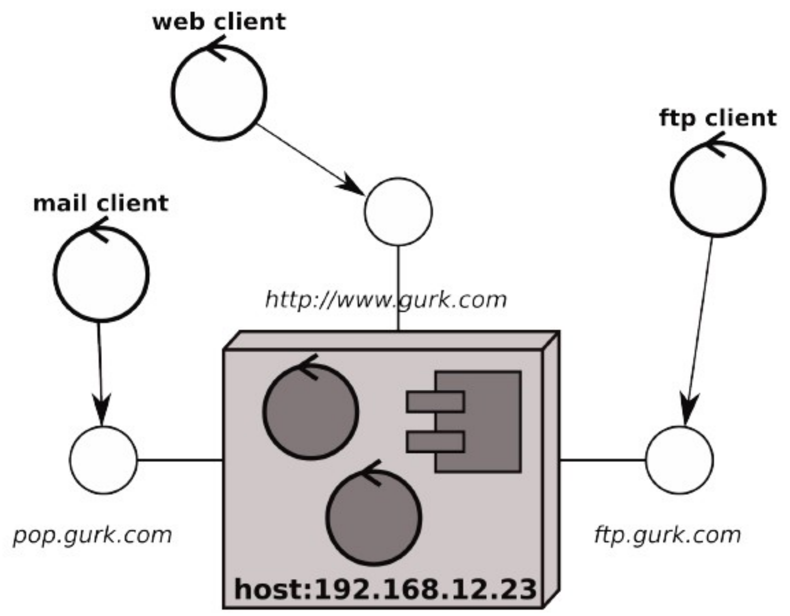
OO Programming Design Principles

- Five: An object presents an interface that specifies which requests can be made of it and what the expected results of those requests are
  - This interface is independent of the actual internal workings of the object – remember the separation of interface and implementation from before
  - However, this is also a restatement of the idea of modularization and suppleness
  - The internal workings of an object are hidden
  - Messages pass through the interface
  - An object can have more than one interface



OO Programming Design Principles

- Six: 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
- In the diagram we see an object of type “host” which is actually three “types” of host;
  - A web server, and ftp host and a pop mail host
  - Each one of these types is associated with a set of responsibilities (render web page, list directory, route mail to user for example)
  - And an interface that clients talk to while the object is acting as (or the role of) a web server or an ftp host or a pop mail host



The Object Model

- There is no definitive definition of what an object is
  - It is just a concept we all understand at a basic level
  - Grady Booch gave what most consider to be the most practical definition
    - An object is something you can do things to
- However, the object model describes the characteristics of the objects that work in OO systems development
  - Classes / Objects
    - Also can be called types / instances
    - A class defines attributes and operations that objects of that type should have
    - Classes don’t physically exist, they are specifications for how to build objects of a specific type
  - Attributes / State
    - The data held by objects: their fields, properties. E.g. for a User class: name, email, phone number
    - The collection of values of an object’s attributes at a specific moment is called its state

The Object Model

- Operations / Methods / Behavior
  - What each object (class) can do: its methods
  - Generally, a message received by an object is responded to by some method
- Identity
  - Each object has identity
  - Even if two objects have identical state, they are distinct entities because they occupy different spatial or temporal locations

- These are the basic properties of the model
  - There a number of other secondary properties
  - But these are considered essential for any programming language to support OO

OO Programming Strategies

- These are standardized ways to design OO code
  - It makes our designs fit with the OO principles
  - Supported directly by OO languages
  - These are often confused with OO principles
  - These are OO design techniques derived from the principles
- Encapsulation
  - This means that the internal workings or implementation of an object are not visible outside the object
  - All interactions with the outside world must go through the interface
  - Exposing the implementation directly to the external world is called breaking encapsulation
    - This is considered a serious breach of OO program design

OO Programming Strategies

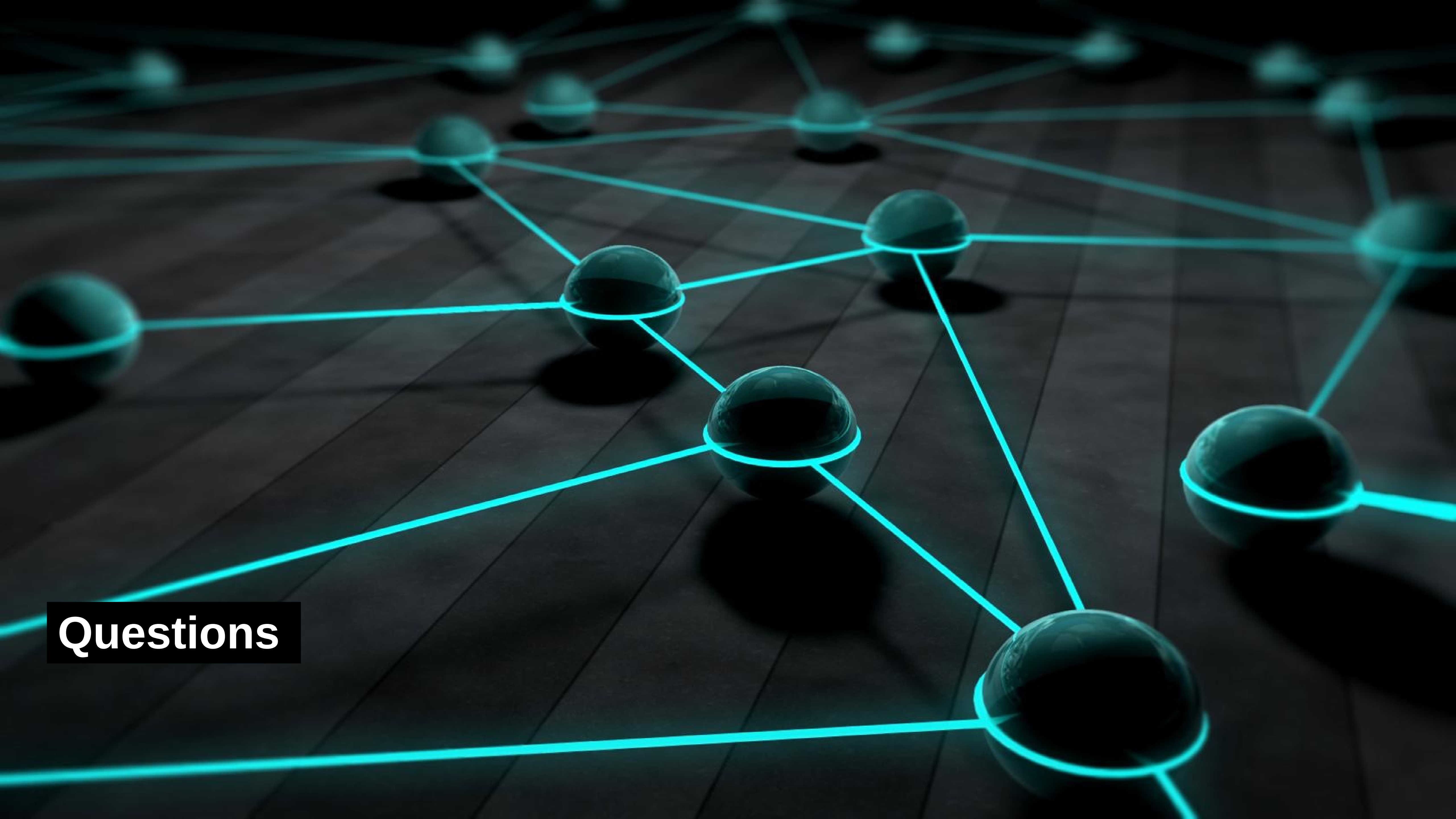
- Abstraction
  - We can abstract away or remove specific details when we create a more general type out of a more specific type
    - For example, if we want to create a more general type “pet” from “dog”, “cat” and “hamster”
    - We would only keep the common characteristics (size, diet) and abstract out (ignore) details that are unique to each of the more specific types of pet
    - The result is often called an abstract type
  - Generally we can’t create an object from an abstract type because we are missing critical info
    - For example we can’t create a generic pet until we know more about what specific type of pet it is
  - However, these abstractions are very helpful in code design
    - They are reusable types that provide bundles of attributes and functionality

OO Programming Strategies

- Inheritance
  - The converse of abstraction
  - It is creating a new type from an existing type by adding to the original type
  - For example
    - A “service dog” is a type of dog, so it inherits all of the properties and functionality of dogs
    - But it also has additional functionality that generic dogs don’t have
  - Inheritance is often expressed in a hierarchy with an “is a” relationship
    - A service dog is a dog which is a mammal
    - Therefore a service dog inherits all of the properties of dog and mammal

OO Programming Strategies

- Polymorphism
  - This is when different classes provide their own unique implementation of a method that is declared in an interface
    - In short: same message, different behaviors
    - It depends on what sort of object is receiving the message
    - Allows different code to be executed by different objects that implement the same interface
  - For example: If I yell “FIRE!”
    - I will get different responses or results depending on who receives the message
    - A firefighter will have a different response than an artillery officer



Questions