Neal Goldstein

President
Neal Goldstein Design, Inc.

Tools

Design Elements of Object Programming

A Synthesis of Natural Perception and Software Development

Copyright

- © Copyright 1990 Neal Goldstein Design Inc. All rights reserved.
- This material is copyrighted by Neal Goldstein Design, Inc. and may be used only with written permission of Neal Goldstein Design, Inc.

Introduction

The National Object Programming Test

- Can you answer true to the following:
  - Users love me
  - I am on time and within budget
  - I look forward to good enhancement and extension requests

Why Object Programming?

- It is more natural
  - The computer representation more closely parallels the real world
  - We naturally think in terms of objects
- Easier extension and enhancement
- Efficiency

It Is More Natural

- Capitalizes on our natural ability to
  - Make*distinctions*
    - We perceive the world as objects
  - Make*distinctions within distinctions*
    - Wholes and parts
  - Make*distinctions about distinctions*
    - Classification and types

Easier Extension and Enhancement

- Implementation can be changed with minimal side effects
  - Enhance
    - Improve existing functionality
  - Extend
    - Add new functionality

Efficiency

- Reuse existing code through
  - Inheritance
  - Using other objects

Software

- Good software must be responsive to *current and future* needs and requirements
- Good software must meet real world time and budget constraints
- Software as solution
- Software as simulation

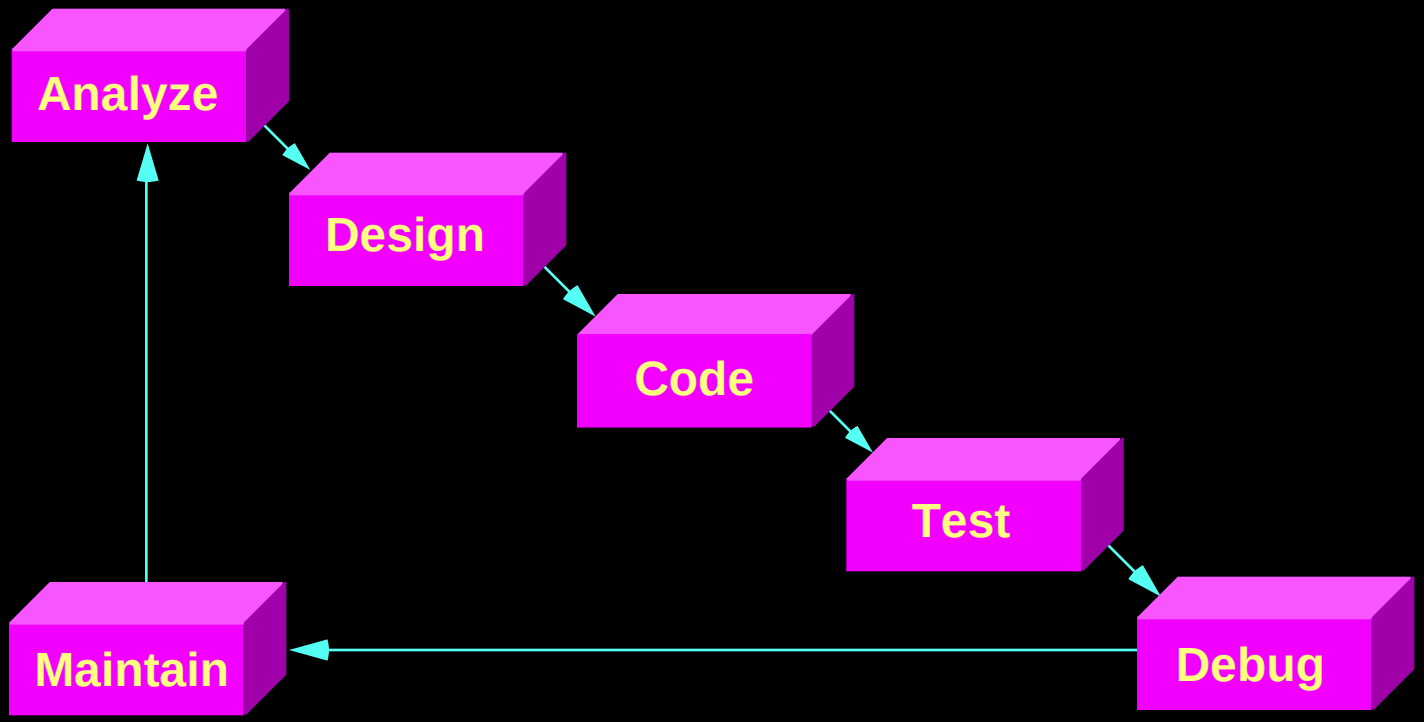
Doing This Requires

- Software that is designed to be extensible, enhanceable, and repairable
- A supporting software architecture and development methodology

We Need

- Problem analysis <=> program architecture tightly coupled
- A development process that supports this
  - Interactive
  - Iterative
  - Incremental

What is Wrong with this Picture?

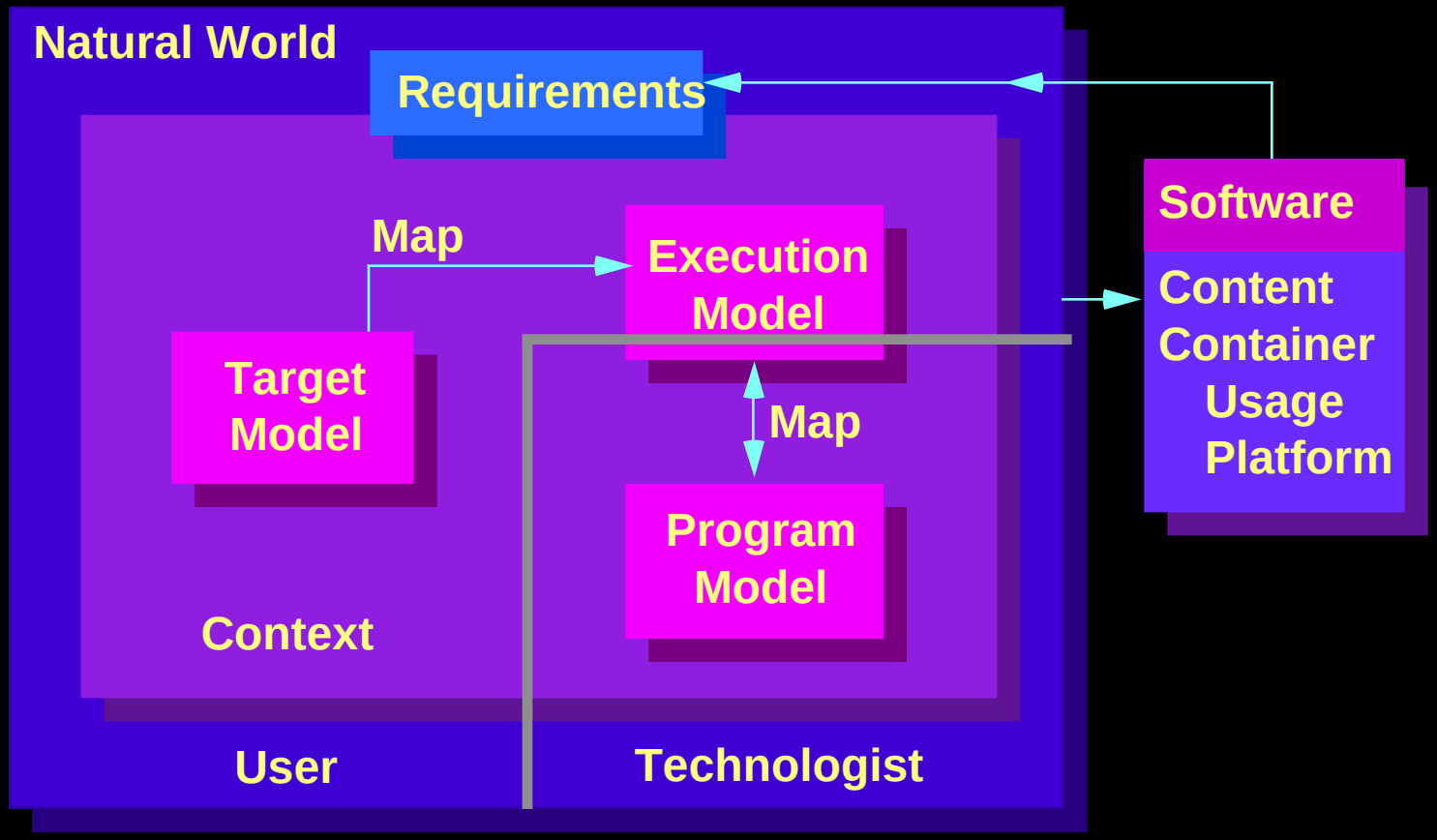


Leave the waterfalls in Yosemite

Object-Based Design

- Does not make the usual distinctions of analysis, design, and implementation
- Instead
  - A model of a software architecture
  - A development methodology to build an instance of that model in a specific context

Object-Based Design



Object-Based Architecture

- Natural world model
- Mapped onto a computer
  - Execution model
  - World of objects
- Abstracted into a program
  - Program model
  - World of classes and inheritance

Object-Based Development

- A new way to develop software
  - Approach
  - Implementation

The Development Approach

- An iterative and interactive process
  - Designs understandable by users
  - Rapid prototyping
  - Constant refinement
- Development and use of class libraries
- You can’t get it right the first time
  - Or the first time right the first time

The Development Implementation

- Development is not linear
  - Modeling the natural world
  - Mapping of the model onto an implementation
    - Coding and program and class design
- Design directing implementation
- Implementation validating design

Modeling

The Parts

Models Explain

- A distinction
- Behavior
  - Goal directed
  - Sum of the parts

A Model

- A set of structural elements
- Rules about the relationship of elements to each other, and to the whole of which they are a part
- Has boundaries
- Has constraints
- Explains a set of behaviors
- Can be abstract or concrete

Models Explain

- Externally
  - What the system does
  - What you want the system to do
- Internally
  - How it does it

The Reference Frame

- The context
  - The bounding of the explanation
- The behavior set to understand
- The constraints
  - Goals are a kind of constraint

Distinctions – the Elements

- The world is filled with objects
  - We perceive the world around us as made up of different types of separate things
- A natural world object is anything we distinguish from something else
  - Tangible
  - Intangible

Natural World Objects

- Have attributes – that can be given values
  - A person has a name, hair color …
- Have behaviors – the things that objects do
  - People breathe, talk, work …
- Communicate with each other
  - Tell stories, smile, hug …
- Can have parts
- Are categorized

Stability

- What tends to change
  - Functions
  - Sequencing of functions
  - User interface
  - Data
  - Interfaces between system components
- Least likely to change

  The actual objects in the problem space

A Model Based on Objects

- Results in a model that is likely to be as stable as possible over time

Relationships

- Intersecting planes
  - Parts and wholes
  - Categories and members
- Collaborator

Distinctions Within Distinctions

- The world is full of things
  - Things are made of other things
  - Things are parts of things

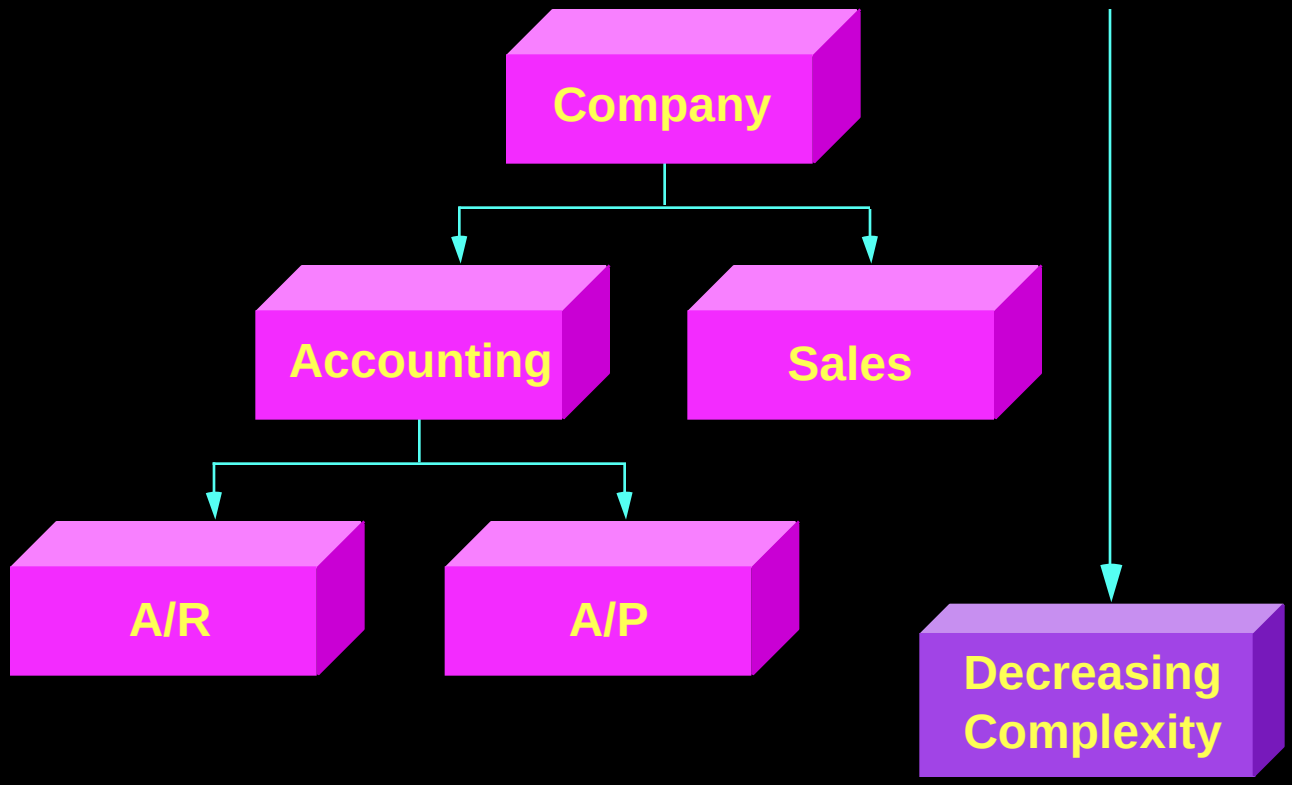
Wholes and Parts

- Divide into parts (model)
  - Wheels, engine, transmission
- Assemble into wholes
  - Drive train, car
  - Are more than the sum of their parts
- Not necessarily a single tree structure

Wholes and Containers

- Wholes
  - Are greater than the sum of their parts
- Containers
  - Contain wholes

Whole/Part Relationship



Distinctions About Distinctions

- The world is full of similar units
  - The same broad features recurring
  - Differing in detail
- People group together elements
  - As members of a category
  - Distinguish that category from others
- Classes can be hierarchical
  - Flower is a plant

Class Membership

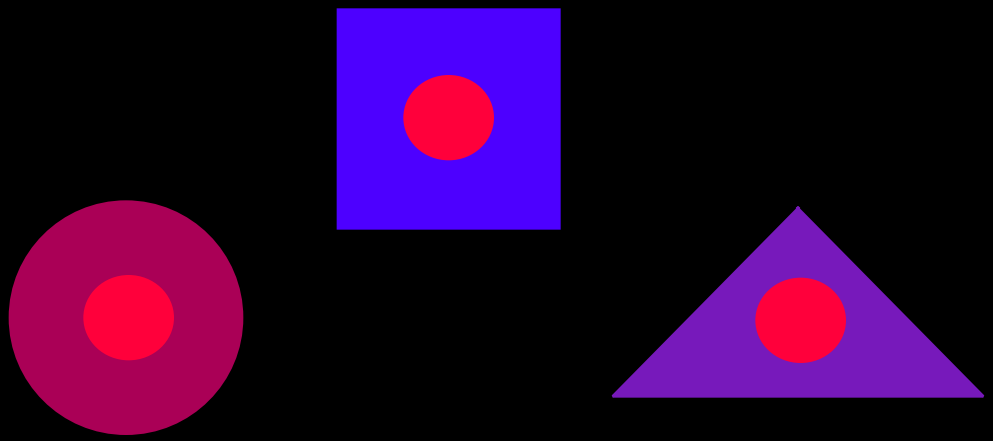
- Something is part of a class
  - If it shares key features of that class
    - We classify a rose as a flower
- A member is an instance of the class

Categorization

- Derived
- Associate
- Prototype

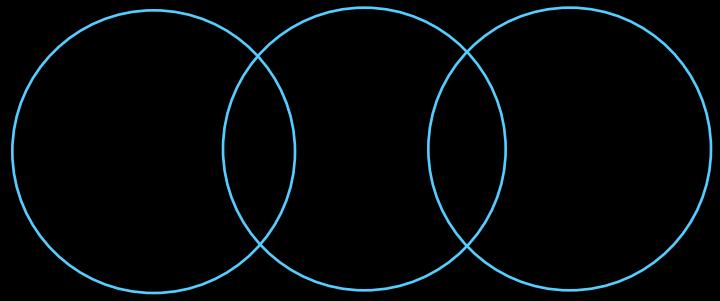
Derived

- Each derived type completely contains its base



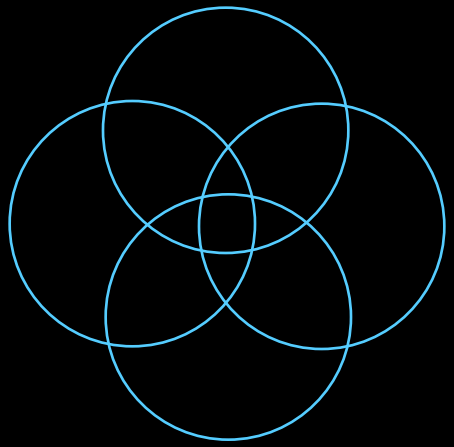
Associate

- If A is related to B, and B is related to C; A, B, and C are all members of the same class



Prototype

- All members share a set of key elements



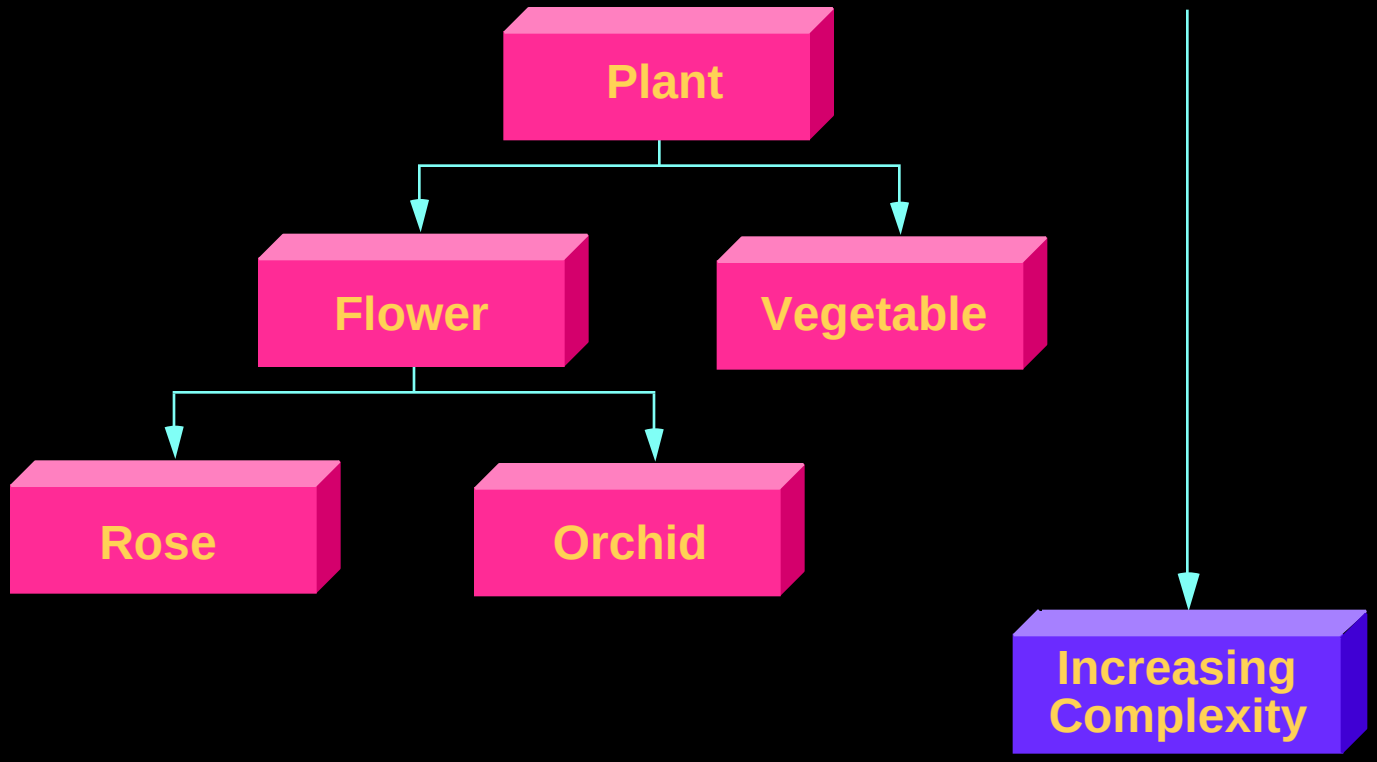
Abstraction

- We use abstraction to hide detail
  - Treat similar things as the same
  - Hourly or salaried are employees
- Abstract based on commonality
  - Roses and orchids are flowers

Derivation

- We use derivation to add detail
  - Treat unique things as different
  - Employees are salaried or hourly
- Derive specific instances based on type
  - Flowers can be roses or orchids or…

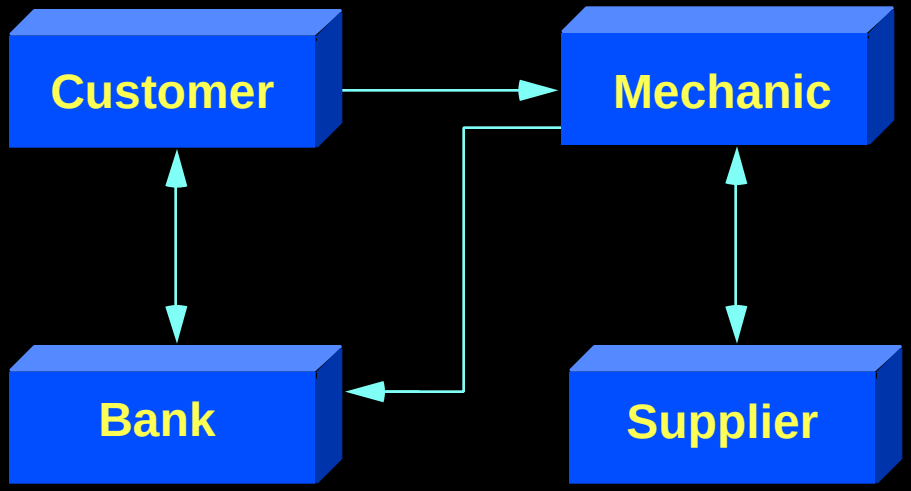
Class/Member Relationship



Collaborator

- There are, and needs to be, connections between elements
- Elements request other elements to do something
  - Ask a mechanic to repair your car
- Elements respond to a request
  - In response to a bill you send a check

Collaborator Relationship



Wholes, Members, Collaborators

- A whole “has-a”
  - A house has a door
- A member “is-a”
  - A bungalow is a house
- A collaborator “does”
  - The mechanic does the requested repair

Modeling

Construction

Building a Model

- Define the reference frame
- Select the objects
- Determine abstraction levels

The Reference Frame

- The context
  - The bounding of the explanation
- The behavior set to understand
- The constraints
  - Goals are a kind of constraint
- Different slices of the same pie

Building Models

- We perceive the world as objects
  - There are multiple overlapping sets
- A model defines a decomposition set
- Models are built
  - Top-down
  - Bottom-up

Submodels

- To build a model we may group together related parts (abstractions) as submodels
  - The “same” object may appear in multiple submodels
    - Usually, a different “type”
- With behaviors (and goals)
  - Processes in a business
  - Consumer/supplier

Select the Objects

- How?

Natural World Objects

- Have attributes that can be given values
  - A person has a name, hair color…
- Have behaviors – the things that objects do
  - People breathe, talk, work…
- Communicate with each other
  - Tell stories, smile, hug…
- Can have parts
- Are categorized

Defining the Elements

- Name
  - Noun
- Behaviors
  - Requested to do
    - Supporting behaviors
  - Requests to have done
    - Connections to other objects

Top-Down

- Decompose from a “system”
  - The parts (at an abstraction level)
    - “Concrete”
      - Managers, programmers, clerks
    - “Abstract”
      - Employee
  - The collaborations
    - Who requires what from whom

Results In

- A system
  - Finance
- Elements
  - At an abstraction level
- Collaborations

Bottom-Up

- Compose from “parts”
  - Parts, abstractions, and collaborations
    - Objects, classes, polymorphism
  - Understand abstractions
    - Inheritance is an implementation strategy
  - Build wholes
    - Knowledge limitation strategies
  - Refine collaborations

Results In

- A system
  - Object-Based Design
- Elements
  - At an abstraction level
- Collaborations

Define Collaborator Relationships

- What this object needs to do
- What this object needs to have done
- What objects this object creates/destroys

(Sub)Model (Decomposition) Rules

- All subtypes must be non-related
  - Don’t include an element and its base (single logical level rule)
- A type can appear only once in a recomposition
- At a given level, each type must be the same kind of type
- A physical instance can appear only once

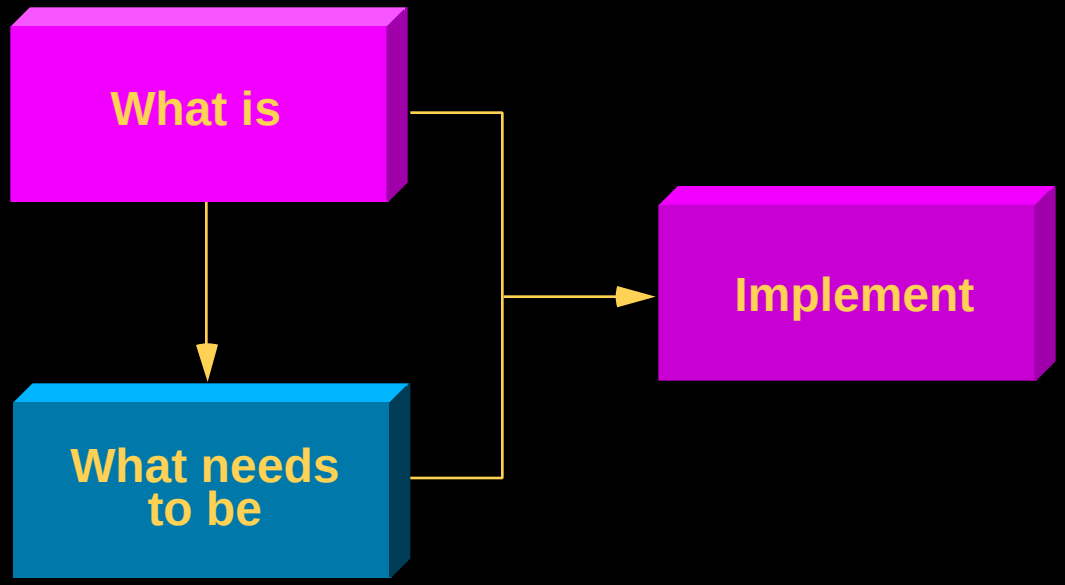
Simple or Complex Models

- Simple models have a single behavior set or are a single decomposition
  - Payroll
- Complex models have multiple goals or contain multiple models
  - Finance–A/R, A/P, Payroll

(Sub)Models May Share Elements

- But the elements can be of different types
  - Employee object in payroll
  - Employee object in personnel planning

Two Models May Be Required



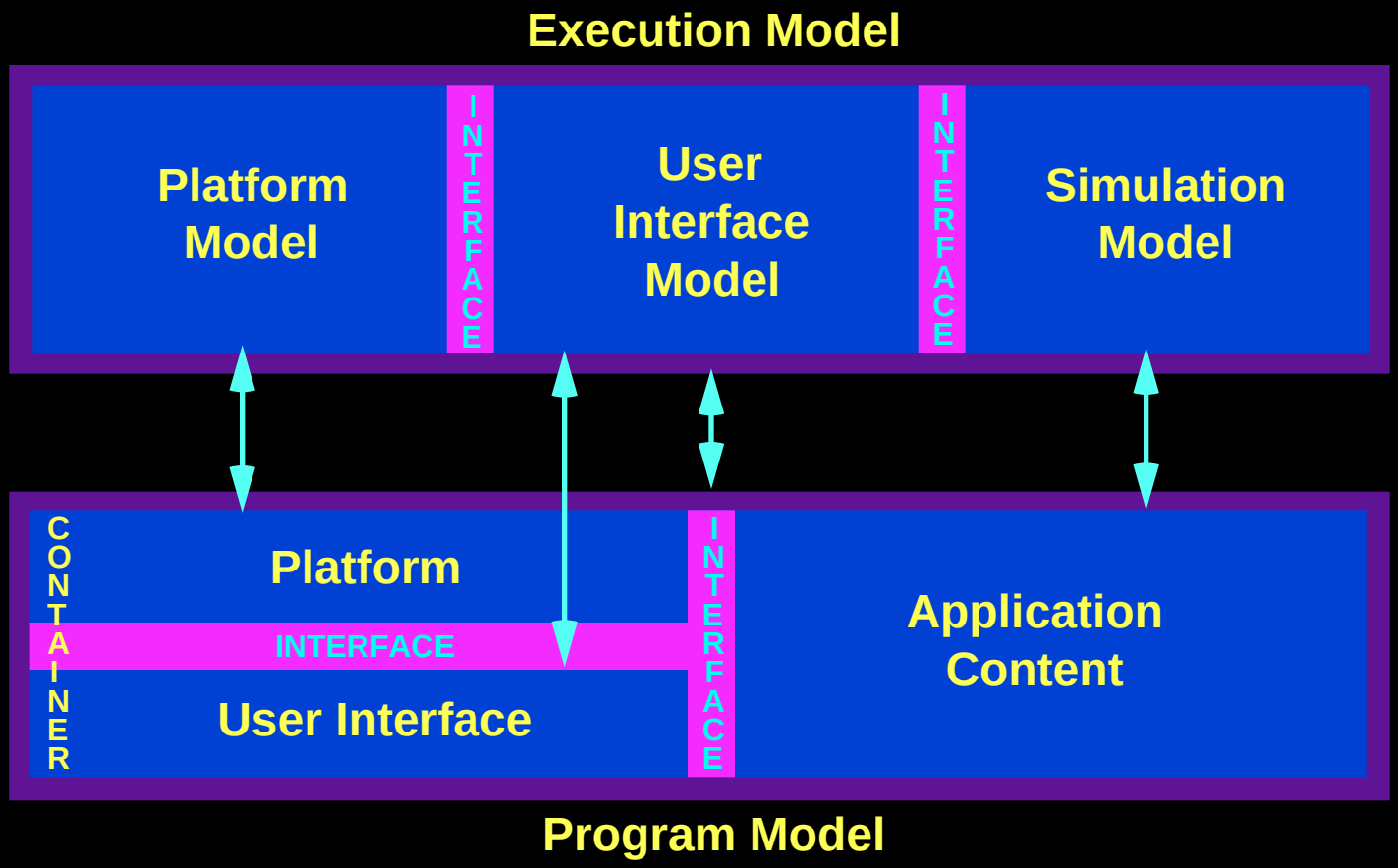
“Good” Models

- Complete and correct
  - Explains all that you know to be true
  - Contains all three sets of relationships
- Consistent
  - Correct distinction between part, type, and collaboration

Object-Based Software Architecture

An Introduction

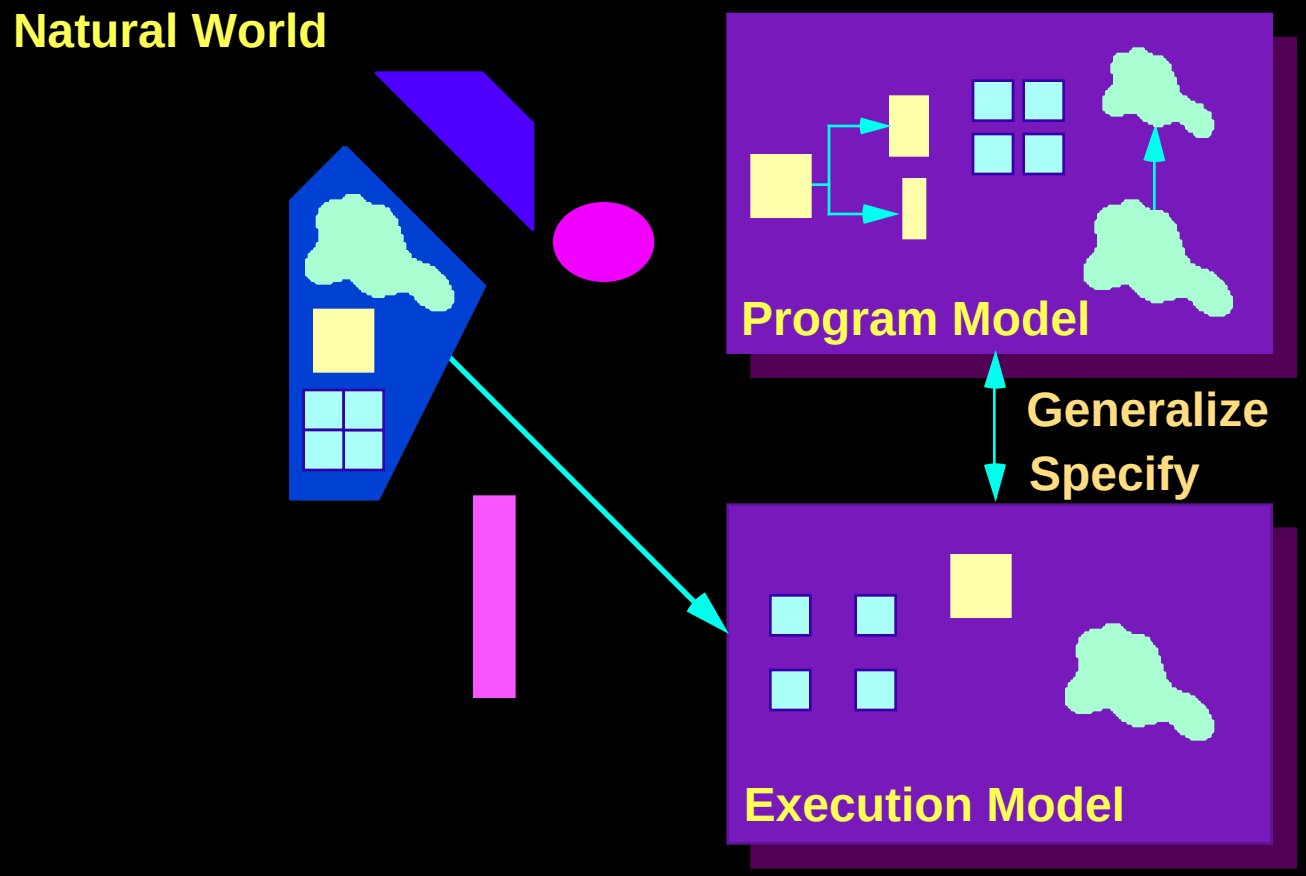
Object-Based Application Architecture



Object-Based Program Architecture

- Execution model
  - The world of objects
  - Wholes and parts
- Program model
  - The world of classes
  - Types and subtypes
- Structure
  - The world of messages
  - Collaborators

Development Process



The Execution Model

The Execution Model

- Based on the natural world
  - Minimizes natural world abstractions
  - Software changes isomorphic with real world changes
- Based on things that stay more constant
- Parts of wholes

The Executing Program

- Is the implementation of a solution model
- Program objects
  - To represent the natural world model
    - Objects necessary to take into account collaborator relationships
  - Needed for automation
  - Required for platform implementation

Program Objects

- Objects behave to "solve a problem" by
  - Carrying out their responsibilities
  - Processing their own data
  - Sending messages to other objects
- An object is known to its system
  - By its (public) protocol or messages

Defining Objects

- Start with the system model objects
  - Reorganize
    - Connect and isolate
  - Generalize and specify
  - Decompose and compose
    - Logically decompose into the smallest parts

Single Purpose Decomposition

- Provide and require specifically defined services
- Highly specialized classes and methods/member functions
- Small method/member function sizes
  - Small numbers of arguments
- Use collaborators
  - To perform common tasks

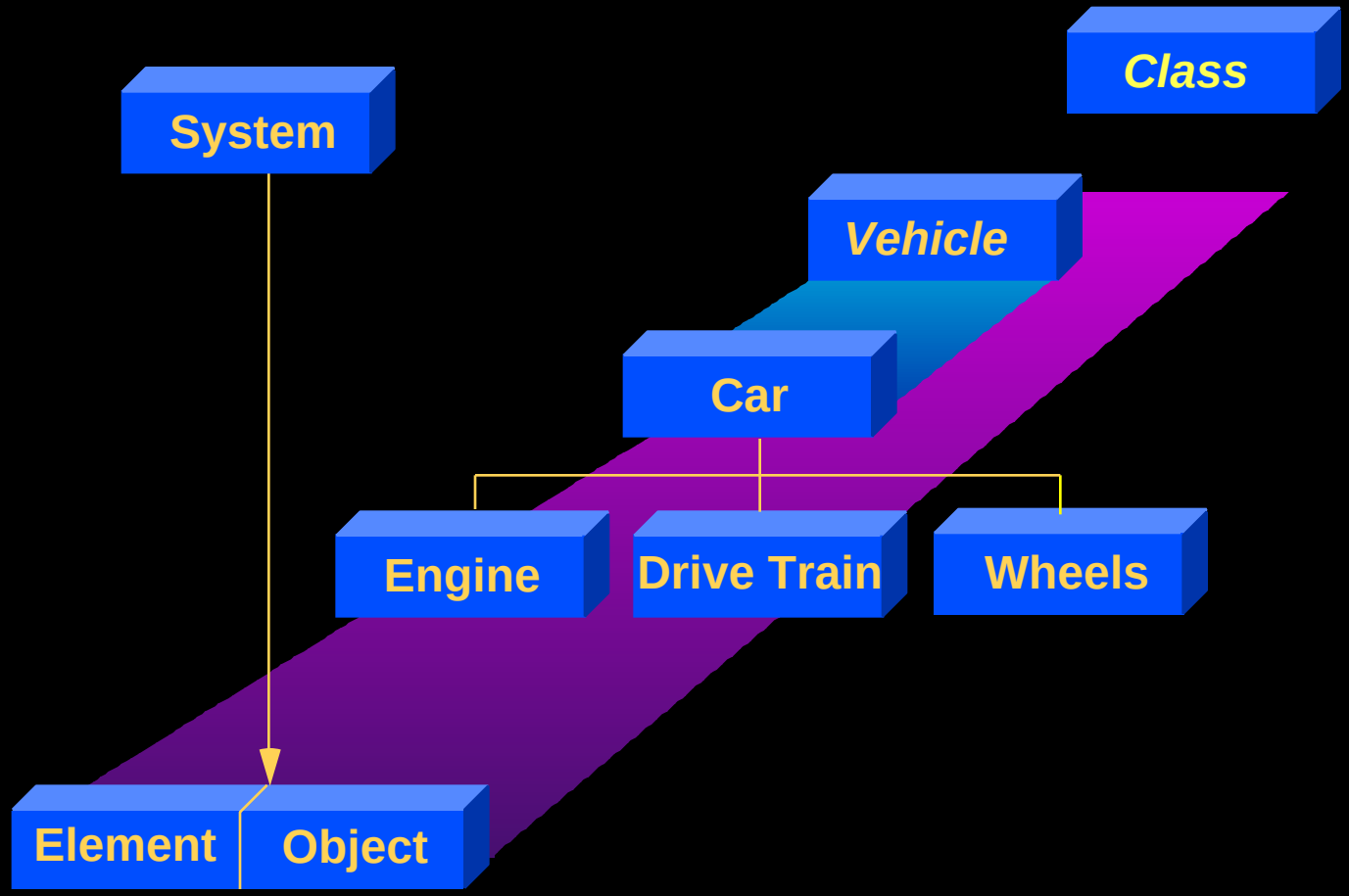
Natural World Structure

- Accountability and structure stay the same
- Performance pushed down to data owner
- It is an ideal world
  - The payroll department is accountable
  - The EmployeeObject generates its own check
  - The CheckObject will write itself

We Code

- Build objects

Intersecting Planes



The Program Model

The Program Model

- A transformation of the natural world
- A hierarchy of derived types
- Takes advantage of object-based software engineering

Distinctions About Distinctions

- The world is full of similar units
  - The same broad features reoccurring
  - Differing in detail
- People group together elements
  - As members of a category
  - Distinguish that category from others
- Classes can be hierarchical
  - Flower is a plant

Class Membership

- Something is part of a class
  - If it shares key features of that class
    - We classify a rose as a flower
- Categorization
  - Derived
  - Associate
  - Prototype
- A member is an instance of the class

In a Company

- Classify the people as employees
  - Subclassify them as
    - Exempt and non-exempt
      - Further subclassify them as
        - Full time and part time

Derivation

- A class can be derived from an existing one
  - Inheriting
    - Data and functions
  - Adding
    - Additional data and functions
  - Overriding
    - Existing functions

Subclasses or Derived Types are Based on Differences

- Only new data items and new or different behavior need be defined
- The higher you go in the hierarchy
  - The more abstract (it should be)
    - Plant is more abstract than rose
- Derived from one or more bases

Creating a Class/Type/Category

- Based on commonalities of behavior and data
  - Separate what is common to a class
    - SalariedEmployee and HourlyEmployee share the idea of Employee

Creating a Class/Type (Category)

- The world is full of similar units
  - The same broad features reoccur
- Start with some instances (or classes)
  - Circle, square, rectangle
- What do they have in common?
  - Can be drawn, dragged, resized
- End with the notion of a class
  - Shape

Deriving a Type or Instance

- The world is full of similar units
  - But differing in detail
- Start with a class
  - Employee
- Derive instances (specific kinds) of Employees
  - HourlyEmployee, SalariedEmployee
- Merging with the world of objects
  - Concrete classes become/are objects

Further Decompose and Compose

- Divide into parts (model)
  - Wheels, engine, transmission
- Assemble into wholes or composites
  - Drive train, car
- Components as convenience in class
- Components are new models

Classes in the Program Model

- A way to share member functions/methods
  - Same function code is used for each instance
- A way to factor code
  - All appointments are a kind of event
- A way to abstract type
  - Implement polymorphism

Class or Object

- People understand there is a difference between
  - The class of something
  - The thing itself (an instance)
- Child is a subclass of person
  - Sarah and Evan are my children

Classes Are Templates for Objects

- Define the variables
  - State or representation
- Define the messages (public interface)
  - The object will respond to
- Define the messages (non-public interface)
  - Supporting functionality
- Define accessibility
  - Member functions and data members

Define Requestor-Responder Relationships

- Address appropriate level of abstraction

Program Development

The Program Design Process

- Start with the natural world/execution objects
- Design classes
  - Build types from instances
  - Derive instances from types
- Build new classes or use existing ones
- Recursively reapply the process

The Program Architecture

- Made up of class hierarchies
  - That add specific behavior
- Made up of class hierarchies
  - With concrete classes that become objects
- A structure
  - Collaborator relationships
- Transformed at execution
  - Into a reflection of the natural world

Defining a Class or Object

- Define the public interface
  - The object's responsibilities
  - The required information
- Define the non-public interface
  - Supporting functionality, data into information
- There is a class/object architecture

Class Design

- Recursively
  - Generalizing and specifying
  - Decomposing and composing
- The “rules” of objects apply

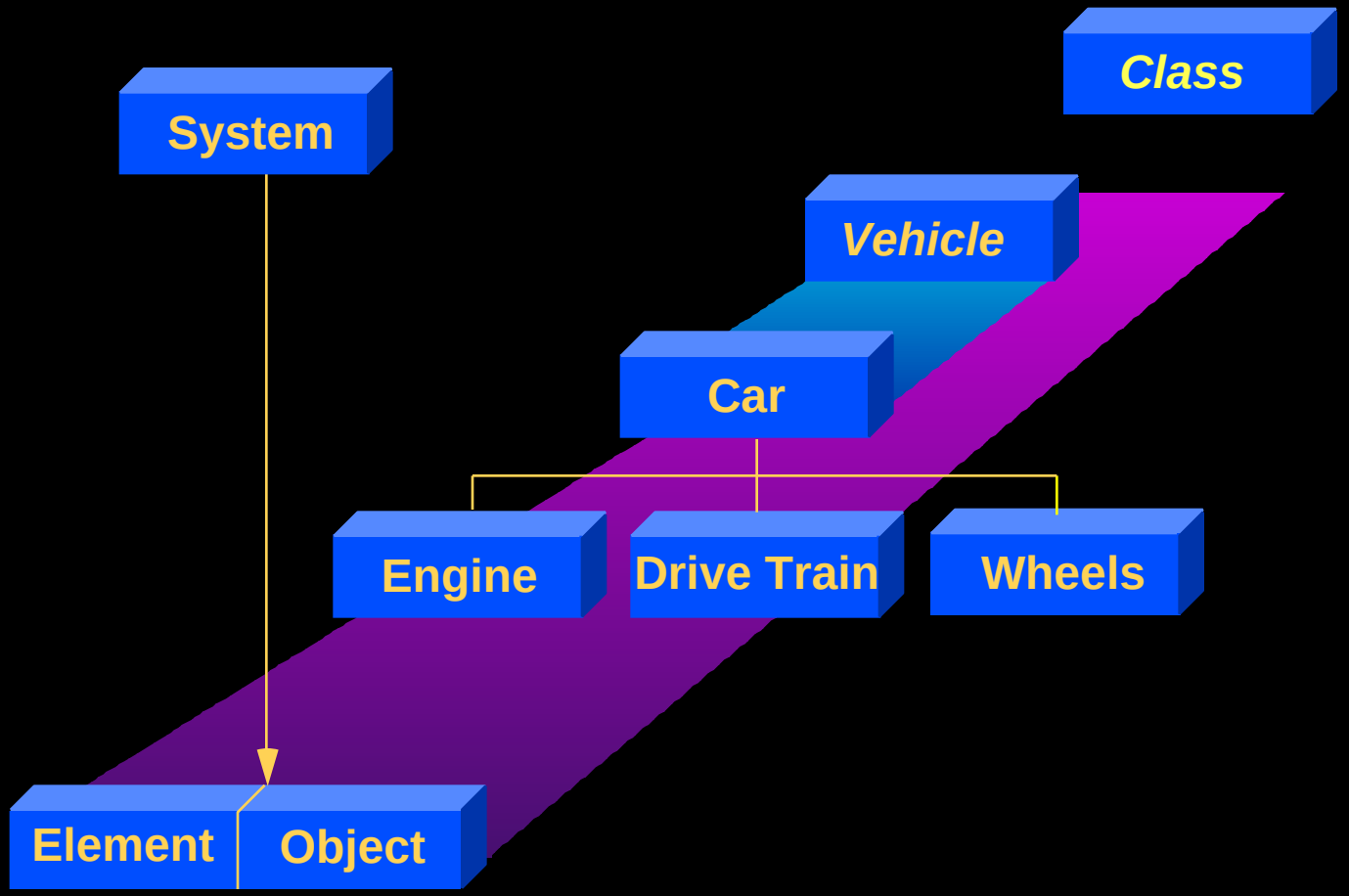
Building Types, Deriving Subtypes

- Commonality between objects and classes
  - Create an abstract classes
  - Maintain the relationship rules
- Uniqueness
  - Create a derived class
  - Maintain the relationship rules

Decomposing and Composing

- Reducing to smallest functional units
- Making wholes

Intersecting Planes



Object Architecture

Program Objects

- People often liken them to a highly specialized program
  - *They are not!*
- There is an architecture

Object Architecture

- Three concepts of access
  - Public – the interface
    - Can be accessed by anyone
  - Protected – the implementation
    - Is public to a subclass
  - Private – the implementation
    - Can be accessed only by a member of the class

Ignorance Is Bliss

- Data representation
- Implementation
- Relationships

Object Architecture

- Requested Behavior
- Support for Requested Behavior
- Data
- Requesting Behavior
- Existence Management

Requested Behavior

- The public interface
- Requests from other objects
- Member functions/methods
- Part of the rules about the relationship of elements to each other, and to the whole of which they are a part

Support for Requested Behavior

- Non-public
- Carry out auxiliary tasks
- Member functions/methods
- Needed to carry out the object’s responsibilities

Data

- Accessed functionality and transformed into information
  - Dismiss the idea of data
- Member functions/methods
  - Accessing data members/variables
- Owned by the object

Requesting Behavior

- Requests to other objects
- Member functions/methods
  - Accessing object reference variables
  - Using references passed as argument
  - Creating new objects
- Part of the rules about the relationship of elements to each other, and to the whole of which they are a part

Existence Management

- Necessitated by computer implementation
- Member functions/methods
  - Accessing existence dependent data

Responder-Requestor Architecture

- Objects communicate through messages
  - Separates requestor from implementor
- An object does something in response to a message
  - Sender requests
  - Receiver implements that request
- Messages as verbs

The Control Structure

- Defined by messages – not a formal control structure
- Responder-Requestor architecture
- Close to the natural world relationships
  - Accountability stays the same
  - Performance pushed down to data owner
- No one is “in charge”
  - Control is immanent in the system

Good Objects

- Small, highly specialized, well defined
- Small and highly specialized member functions/methods
- Use other objects
- Independent

Minimizing Dependencies

There Are Connections

- Between
  - Requestors and responders
  - Wholes and parts
  - Superclasses and subclasses

Yet Independence Is Critical

- Limit data knowledge
- Limit implementation knowledge
- Limit relationship knowledge
- Limit responsibilities

Limit Data Knowledge

- Functionally access data as information
- Unfreeze the representation of data
  - Data members/variables
  - Computed data
- Make changes in representation transparent
- Define data once
- Don’t treat data members/variables as global

Limit Implementation Knowledge

- Encapsulate policies and procedures
- Concentrate on incremental improvement
- Control access to lower level functions
- Identify responsibility
- Implement algorithms once

Limit Relationship Knowledge

- Limit type knowledge
  - Type at the highest level of abstraction
- Limit instance knowledge
  - Use as few other objects as possible

Limit Responsibilities

- Decompose to single purpose objects
- Small, specialized methods/member functions
  - Treat member functions/methods as single function/responsibility objects
- Minimize the public interface

How To Minimize Dependencies

Components and Collaborators

- Let you hide code as well as data
  - Composites and components
    - Objects decomposed into parts
  - Requestor-Responder
    - Objects provide services

Classes

- Limit code knowledge through inheritance
- Limit object knowledge through polymorphism

Optimizing Class Use

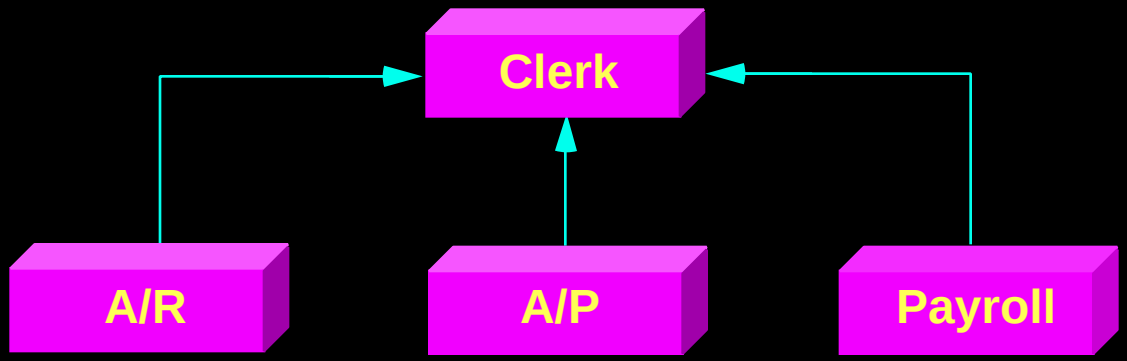
- The design context affects reuse
- Follow the rules of good objects

Optimize Hierarchies

- Subclass abstract classes or special case concrete classes
- Class hierarchies should be appropriately
  - Deep or shallow
  - Narrow or wide

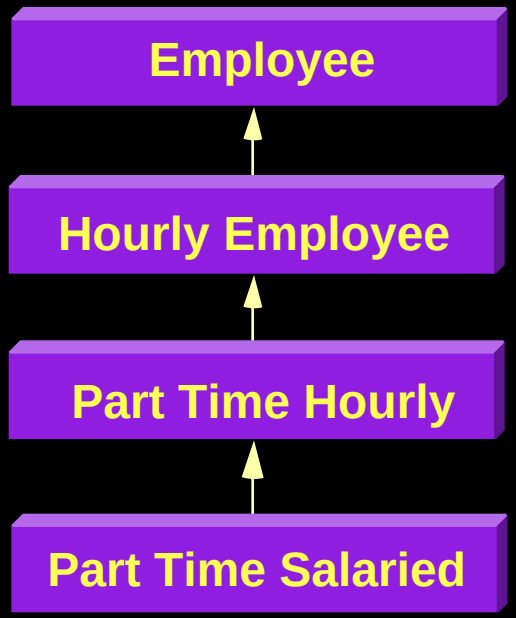
Wide Hierarchies

- May indicate we confuse values of variables with types
- In personnel planning – a department variable



Deep Hierarchies

- May indicate confused abstraction
- Excessive overriding?



Class Membership

- Something is part of a class
  - If it shares key features of that class
    - We classify a rose as a flower
- Categorization
  - Derived
  - Associate
  - Prototype

Inheritance

- Derived
  - Type extension - design
  - To implement details and encapsulate
- Prototype
  - Reuse objects across models - implementation
  - To build a new base

Overriding In Type Extension

- Derived
  - Limit overrides
    - Common method holders necessary for polymorphism
    - Use the inherited function/method and add functionality
  - Subclasses should be able to be used in place of superclasses

Overriding In New Bases

- Prototype
  - More complete overriding may be necessary

Multiple Base Classes

- Used to build new types (base classes) from primitives
  - You are not building a derived type
  - Private inheritance
- Uses the multiple inheritance mechanism, but is not “inheritance”

Multiple Inheritance

- Four criteria for deriving from a second class
  - There is an “is-a” relationship
  - It crosses branches in the hierarchy
    - Collectable cars
  - Behavior must be added or modified
  - Polymorphism is desired
    - Treat Packard as car or collectable

Multiple Inheritance

- Can result in a “write only” class structures
- Limit additional base classes to “multi-type” functionality
  - The idea of mixin classes
    - Optional functionality

Object-Based Software Architecture

Building Applications

There Are Types of Objects

- We group objects together to achieve the same benefits we get from objects
  - Maintainability
  - Enhanceability
  - Extensibility

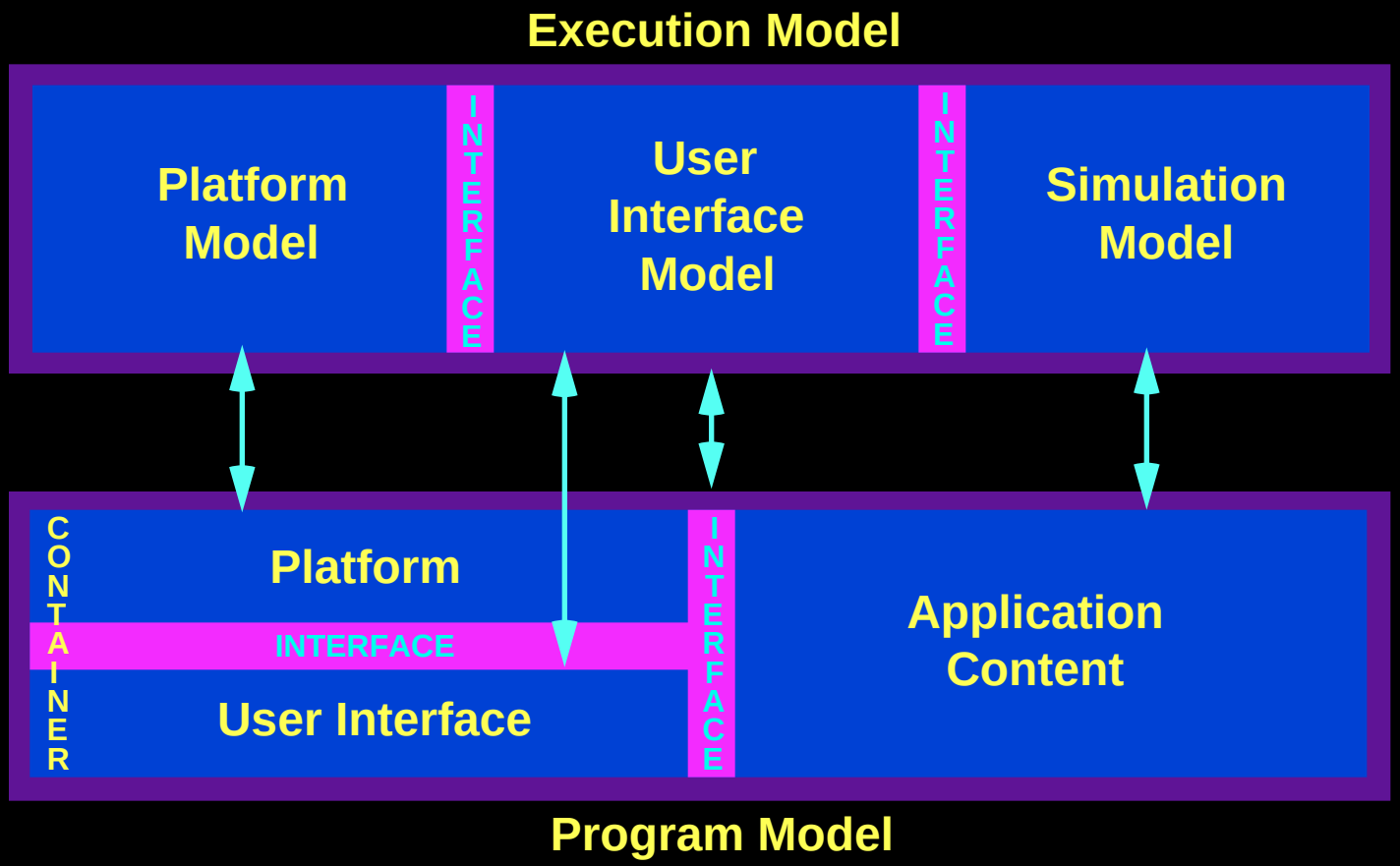
Program Objects Have Rules

- Their relationship to each other
- Their relationship to the whole of which they are a part
- The idea of independence
  - Limit data knowledge
  - Limit implementation knowledge
  - Limit relationship knowledge
  - Limit responsibilities

Composite Relationships

- Composites “inherit” these rules
- The relationship between
  - Content
  - User interface
  - Platform

Object-Based Application Architecture



The power to be your best