Object-Oriented Software Engineering
Using UML, Patterns, and Java

Chapter 1 & Chapter 2
Introduction & Modeling principles

Course instructor : TA.Nada Alamoudi

Objectives of the Course

- Appreciate the Fundamentals of Software Engineering:
  - Methodologies
  - Process models
  - Description and modeling techniques
  - System analysis - Requirements engineering
  - System design
  - Implementation: Principles of system development

Focus: Acquire Technical Knowledge

- Different methodologies ("philosophies") to model and develop software systems
- Different modeling notations
- Different modeling methods
- Different software lifecycle models (empirical control models, defined control models)
- Different testing techniques (e.g. vertical testing, horizontal testing)
- Rationale Management
- Release and Configuration Management

Acquire Managerial Knowledge

- Learn the basics of software project management
- Understand how to manage with a software lifecycle
- Be able to capture software development knowledge (Rationale Management)
- Manage change: Configuration Management
- Learn the basic methodologies
  - Traditional software development
  - Agile methods

Techniques, Methodologies and Tools

- **Techniques:**
  - Formal procedures for producing results using some well-defined notation
- **Methodologies:**
  - Collection of techniques applied across software development and unified by a philosophical approach
- **Tools:**
  - Instruments or automated systems to accomplish a technique
  - CASE = Computer Aided Software Engineering

Software Engineering: A Working Definition

Software Engineering is a collection of techniques, methodologies and tools that help with the production of

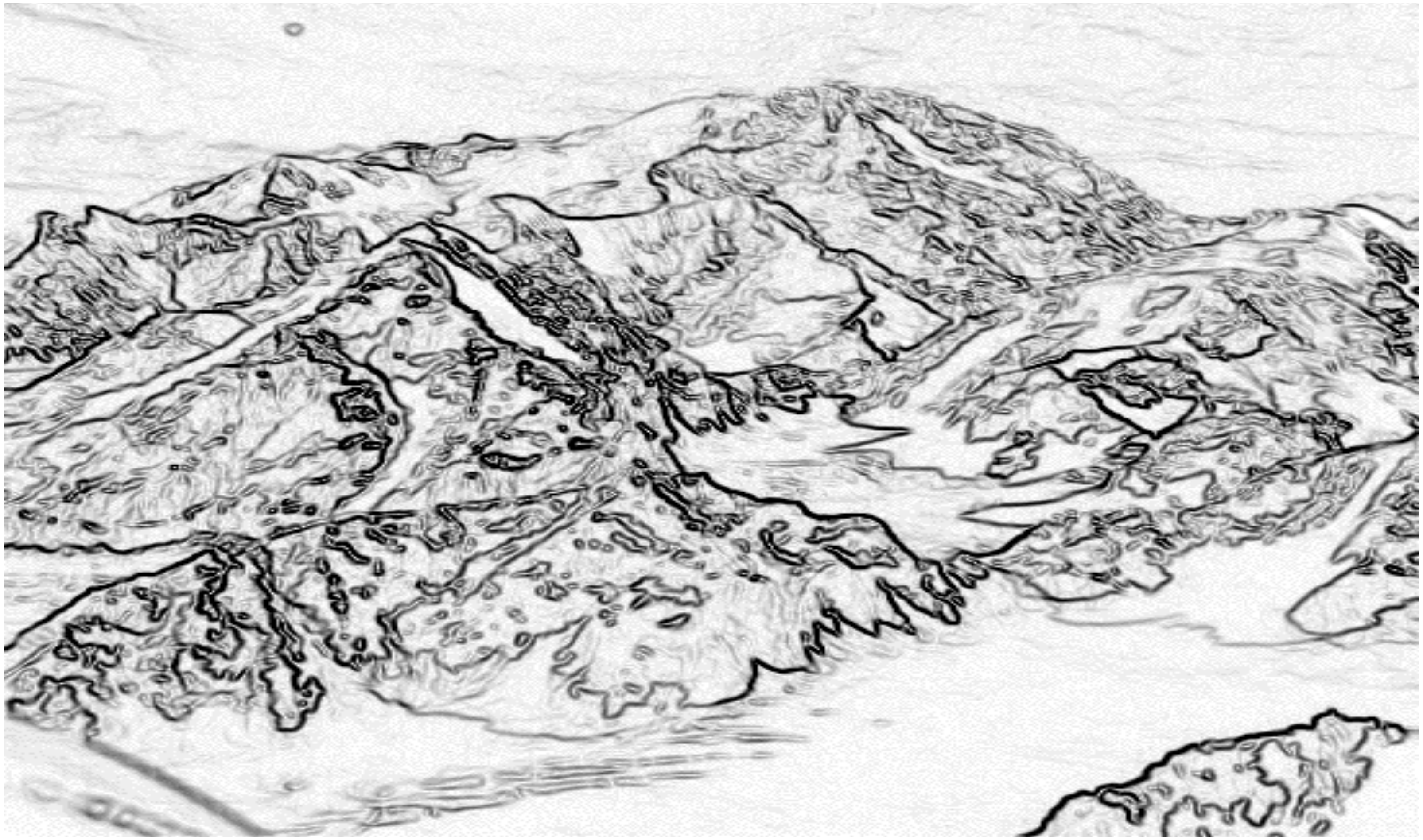
A high quality software system developed with a given *budget* before a given *deadline* while *change* occurs

Challenge: Dealing with complexity and change

Object-Oriented Software Engineering
Using UML, Patterns, and Java

Chapter 2, Modeling principles

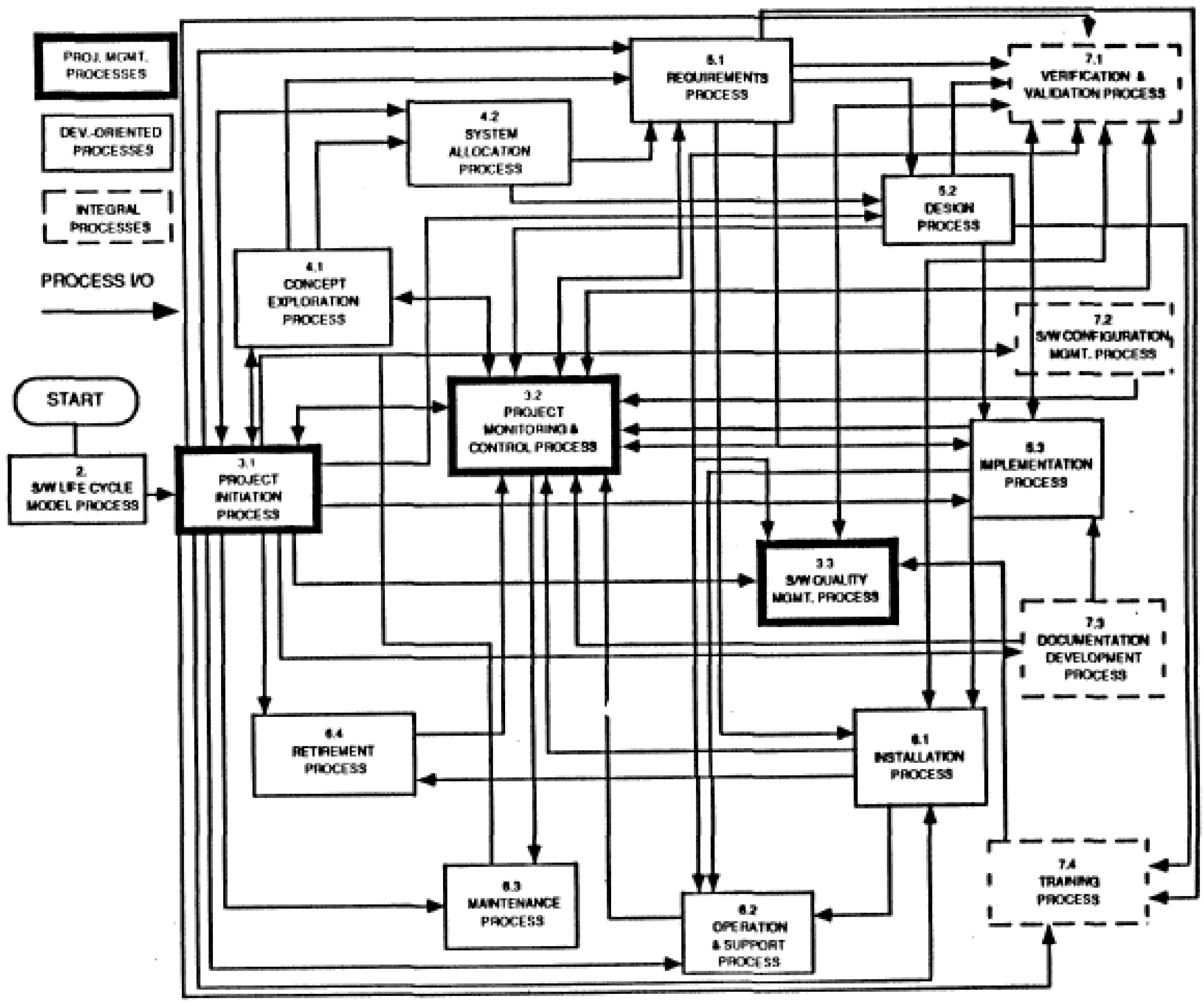
(Textbook Chapter 2)



Overview for the Lecture

- Three ways to deal with complexity
  - Abstraction and Modeling
  - Decomposition
  - Hierarchy
- Introduction into the UML notation

What is the problem with this Drawing?



Abstraction

- Abstraction allows us to ignore unessential details
- Two definitions for abstraction:
  - Abstraction is a *thought* فكر *process* where ideas are distanced from objects
    - **Abstraction as activity**
  - Abstraction is the *resulting idea* of a thought process where an idea has been distanced from an object
    - **Abstraction as entity**
- Ideas can be expressed by models

Models

- A model is an abstraction of a system
  - A system that no longer exists
  - An existing system
  - A future system to be built.

We use Models to describe Software Systems

- Object model: What is the structure of the system?
- Functional model: What are the functions of the system?
- Dynamic model: How does the system react to external events?

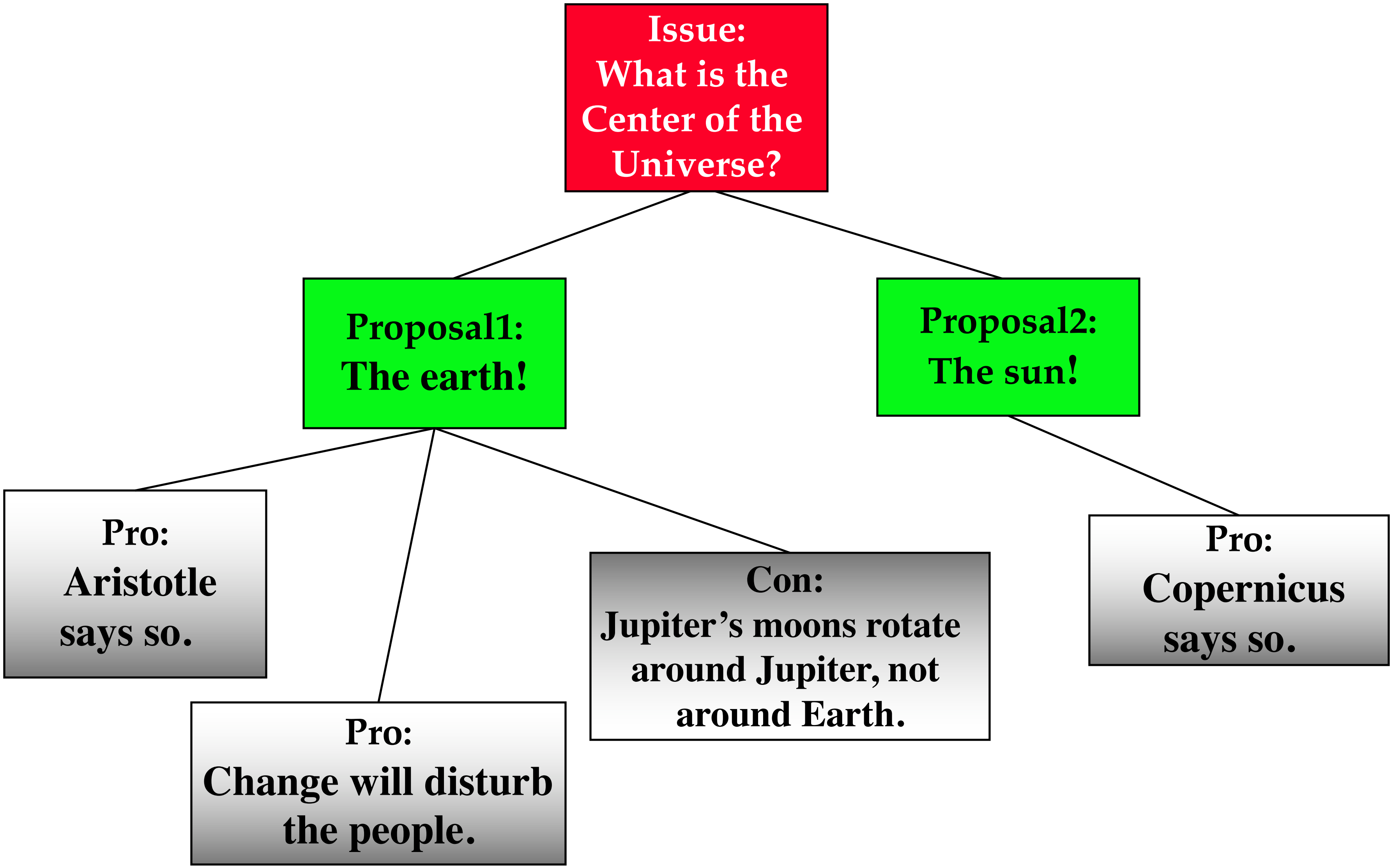
- System Model: Object model + functional model + dynamic model

Other models used to describe Software System Development

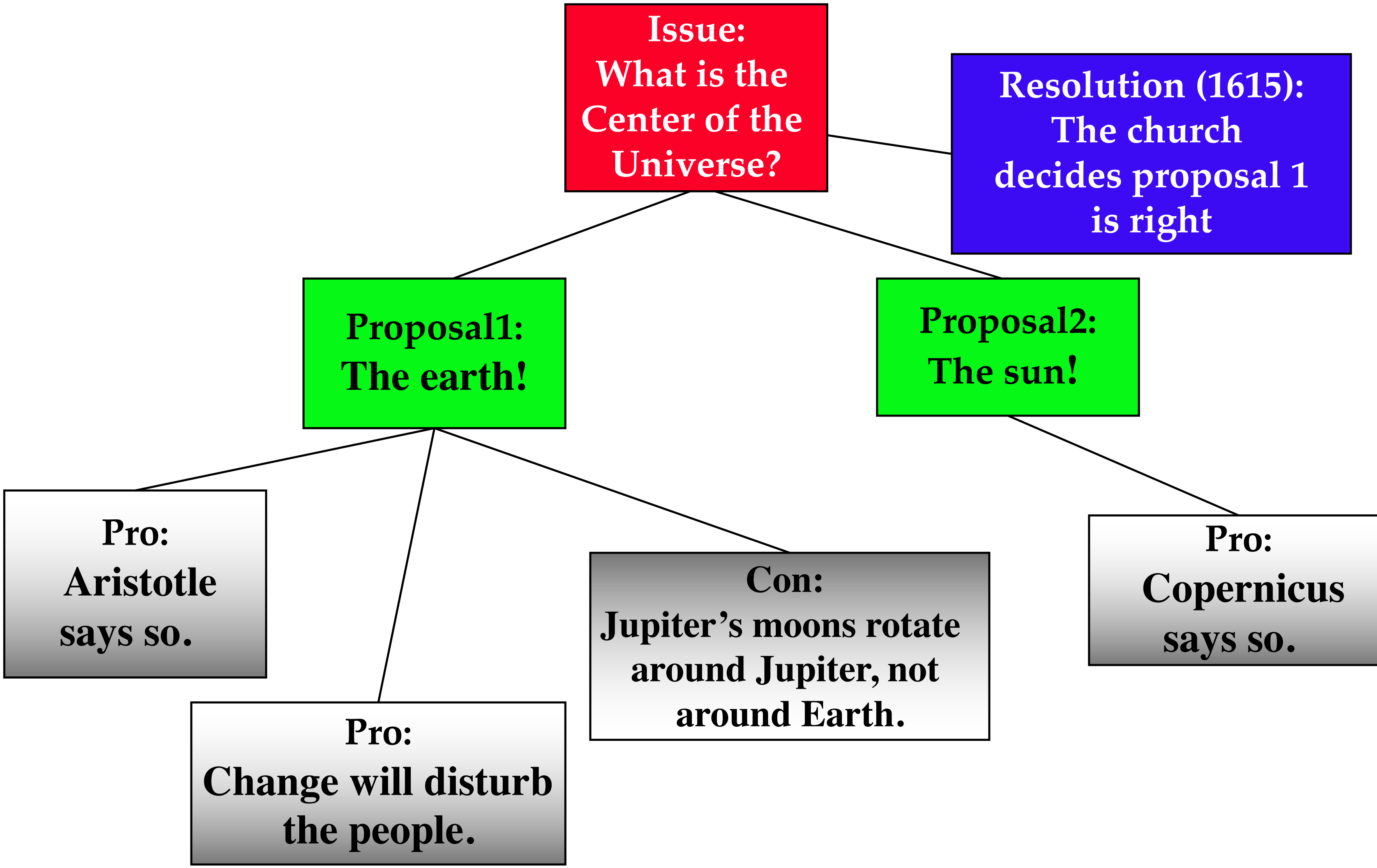
- Task Model:
  - PERT Chart: What are the dependencies between tasks?
  - Schedule: How can this be done within the time limit?
  - Organization Chart: What are the roles in the project?

- Issues Model:
  - What are the open and closed issues?
    - What blocks me from continuing?
  - What constraints were imposed by the client?
  - What resolutions were made?
    - These lead to action items

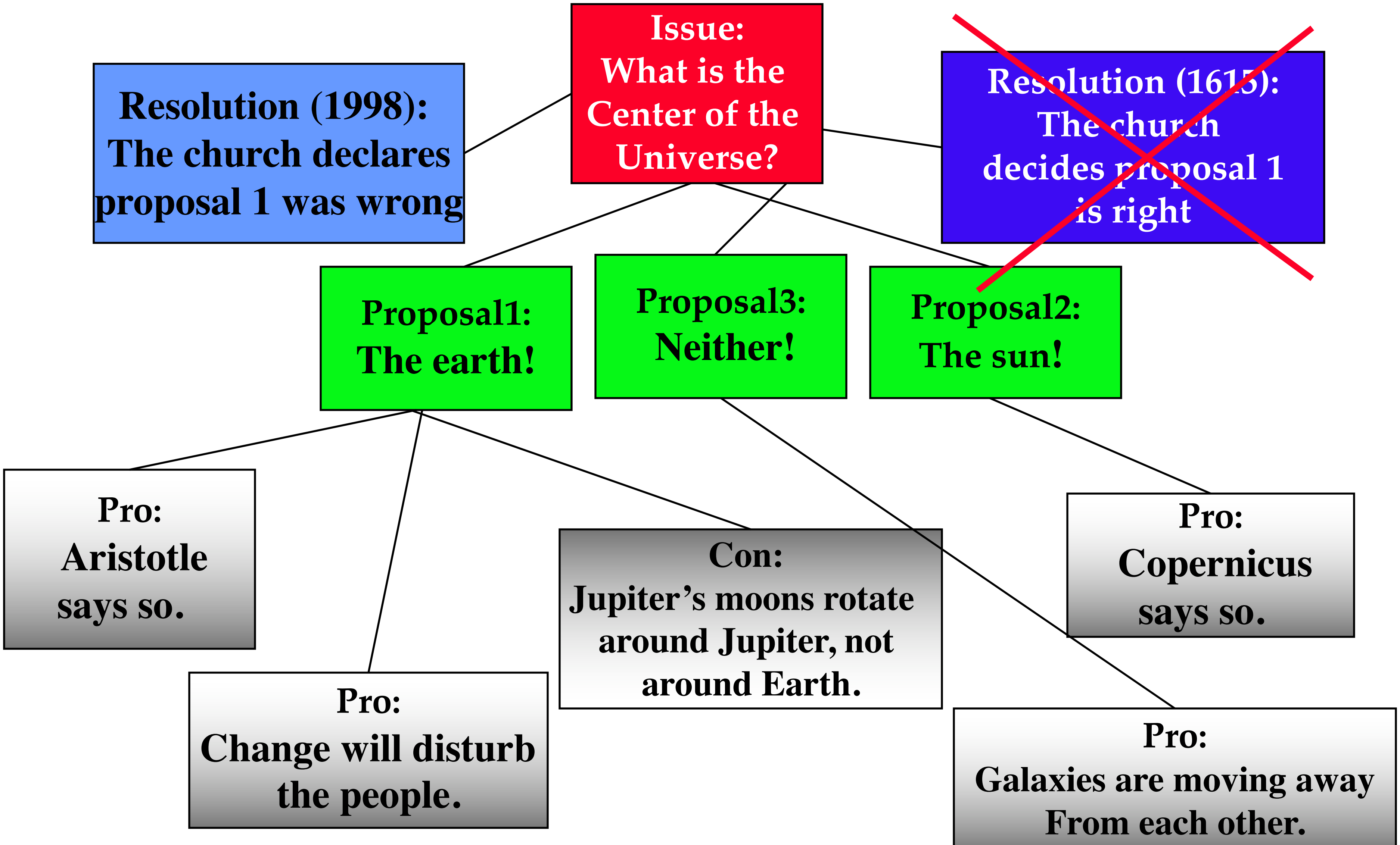
Issue-Modeling



Issue-Modeling

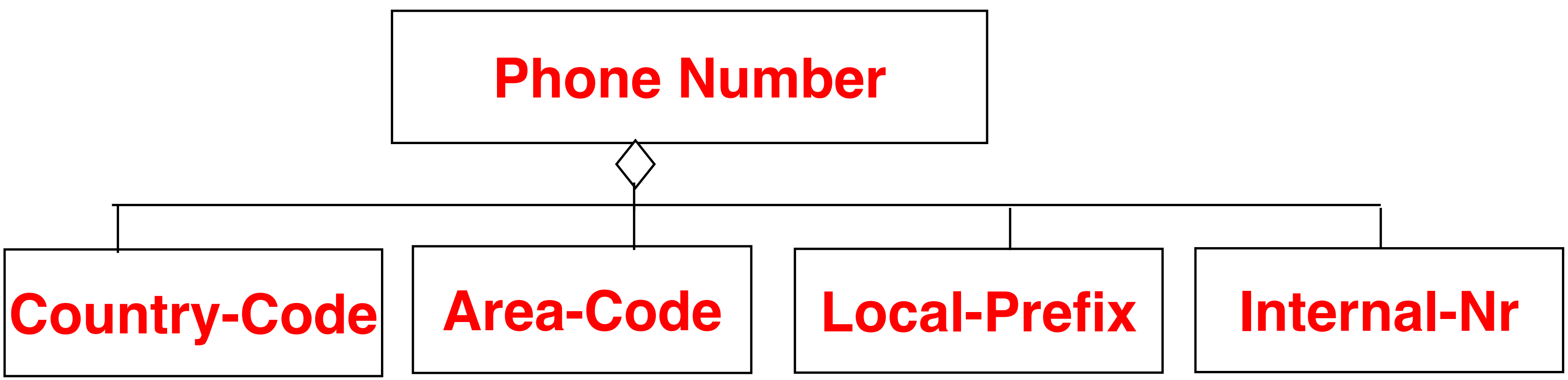


Issue-Modeling



2. Decomposition

- Complex systems are hard to understand
  - The 7 +- 2 phenomena
    - Our short term memory cannot store more than 7+-2 pieces at the same time -> limitation of the brain
    - My Phone Number: 498928918204
- Chunking:
    - Group collection of objects to reduce complexity
    - State-code, Area-code, Local Prefix, Internal-Nr



Decomposition (cont'd)

- A technique used to master complexity ("divide and conquer").

- Two major types of decomposition
  - Functional decomposition
  - Object-oriented decomposition

- Functional decomposition
  - The system is decomposed into modules
  - Each module is a major function in the application domain
  - Modules can be decomposed into smaller modules.

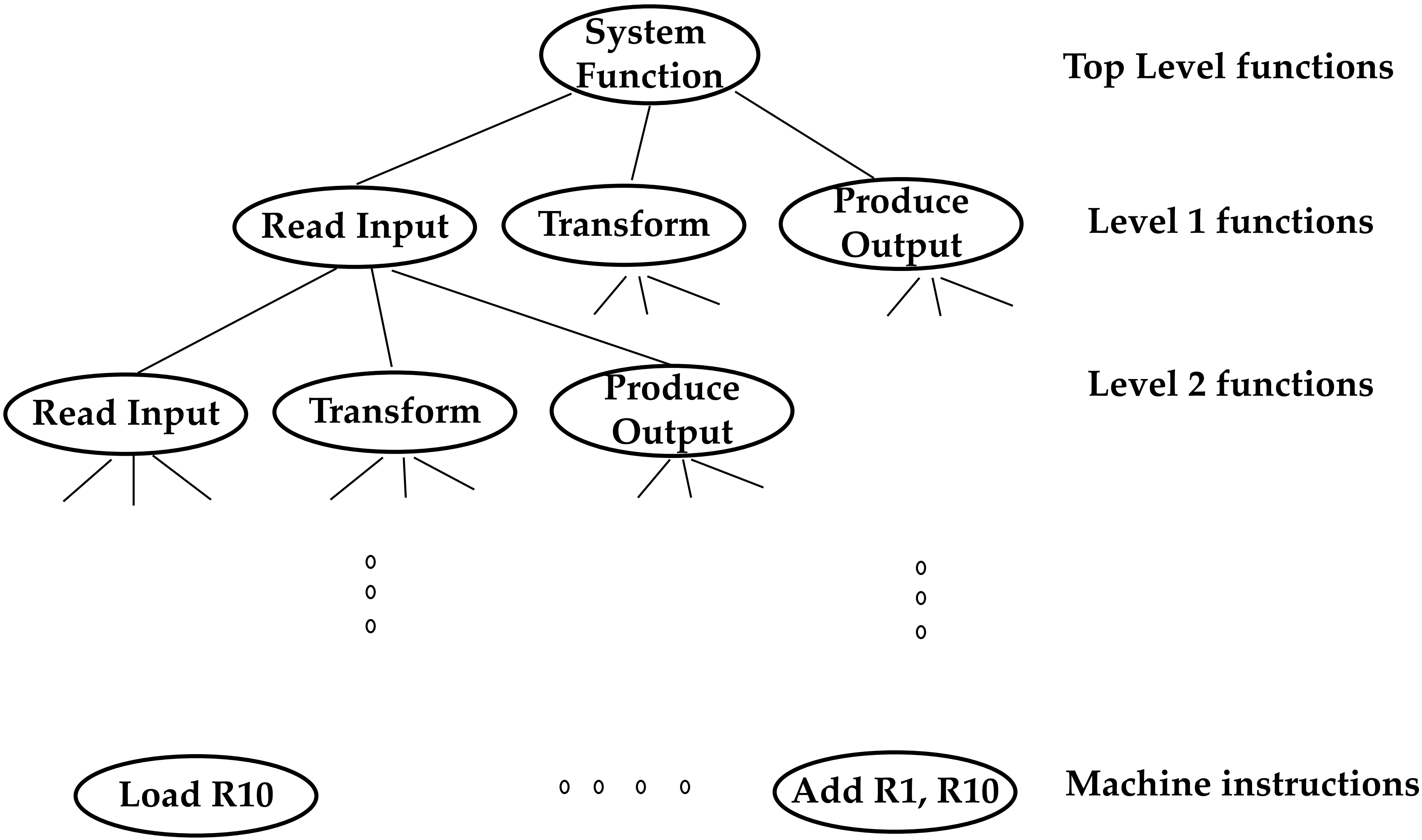
Decomposition (cont'd)

- Object-oriented decomposition
  - The system is decomposed into classes ("objects")
  - Each class is a major entity in the application domain
  - Classes can be decomposed into smaller classes

- Object-oriented vs. functional decomposition

Which decomposition is the right one?

Functional Decomposition



Functional Decomposition

- The functionality is spread all over the system
- Maintainer must understand the whole system to make a single change to the system
- Consequence:
  - Source code is hard to understand
  - Source code is complex and impossible to maintain
  - User interface is often awkward and unintuitive.

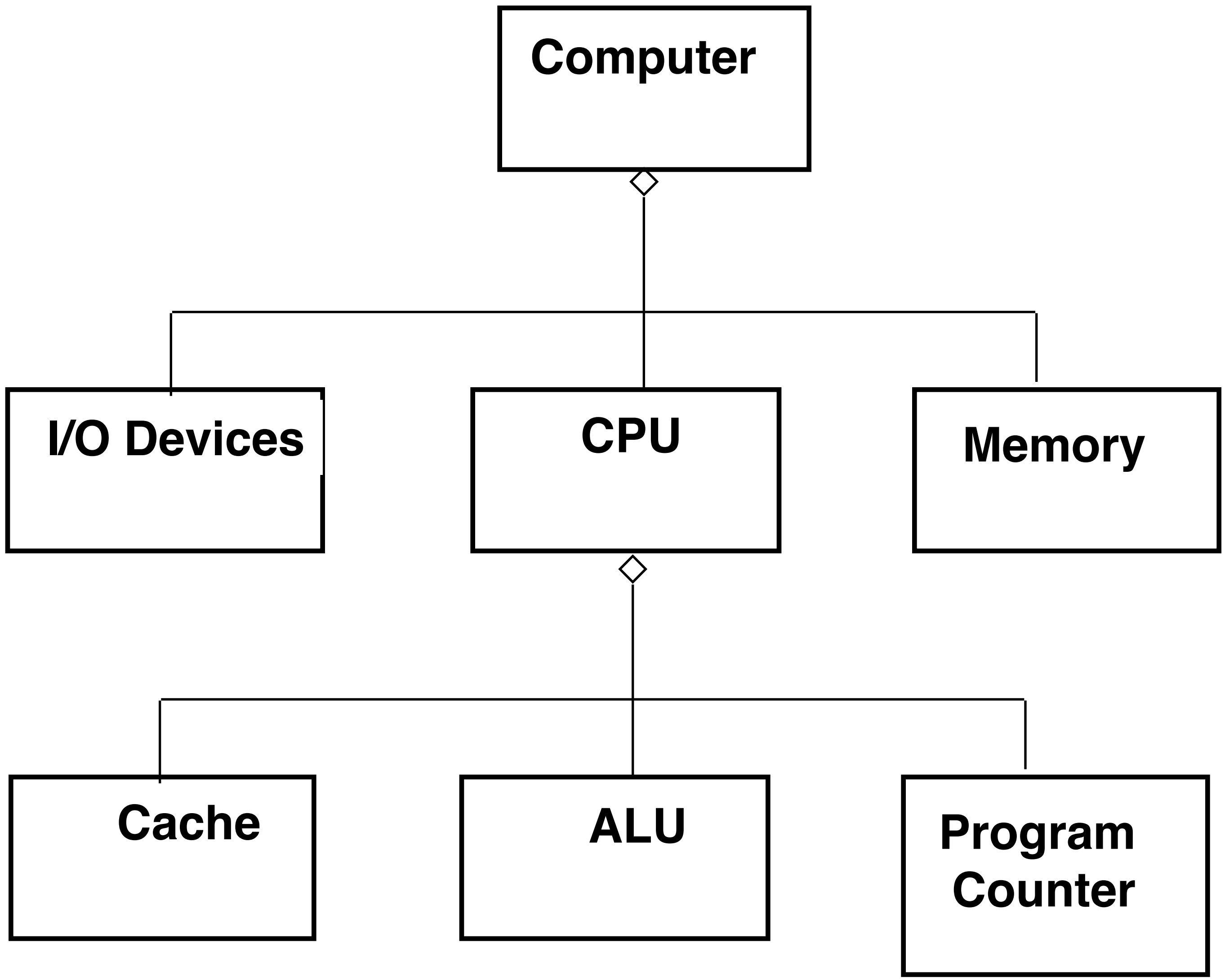
Class Identification

- **Basic assumptions:**
  - We can find the *classes for a new software system:* Greenfield Engineering
  - We can identify the *classes in an existing system*: Reengineering
  - We can create a *class-based interface to an existing system:* Interface Engineering.

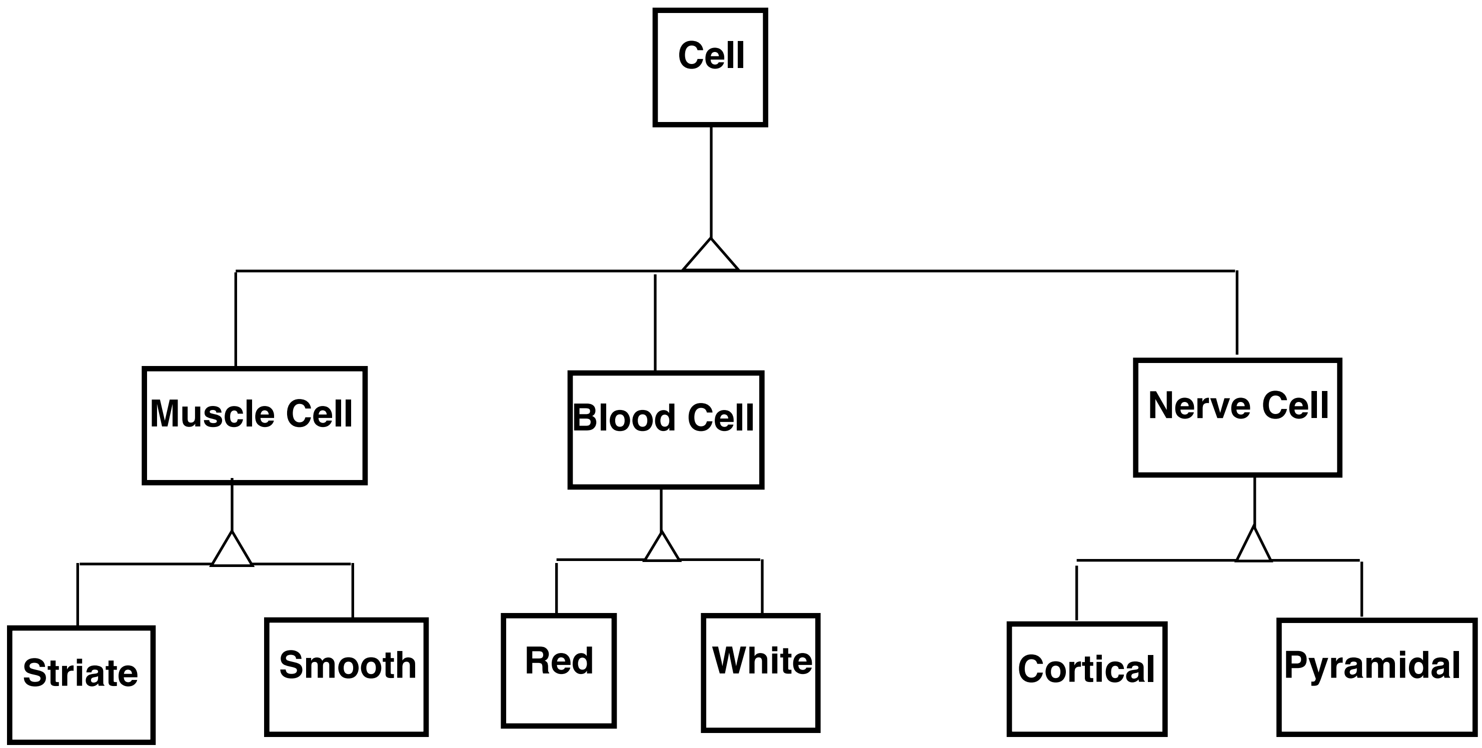
3. Hierarchy

- So far we got abstractions
  - This leads us to classes and objects
  - “Chunks”
- Another way to deal with complexity is to provide relationships between these chunks
- One of the most important relationships is hierarchy
- 2 special hierarchies
  - "Part-of" hierarchy
  - "Is-kind-of" hierarchy.

Part-of Hierarchy (Aggregation)



Is-Kind-of Hierarchy (Taxonomy)



Where are we?

- Three ways to deal with complexity:
  - Abstraction, Decomposition, Hierarchy
- Object-oriented decomposition is good
  - Unfortunately, depending on the purpose of the system, different objects can be found
- How can we do it right?
  - Start with a description of the functionality of a system
  - Then proceed to a description of its structure
- Ordering of development activities
  - Software lifecycle

Models must be falsifiable

- Karl Popper ("Objective Knowledge):
  - There is no absolute truth when trying to understand reality
  - One can only build theories, that are "true" until somebody finds a counter example
- Falsification: The act of disproving a theory or hypothesis

- In software engineering any model is a theory:
  - We build models and try to find counter examples by:
    - Requirements validation, user interface testing, review of the design, source code testing, system testing, etc.
- Testing: The act of disproving a model.

Concepts and Phenomena

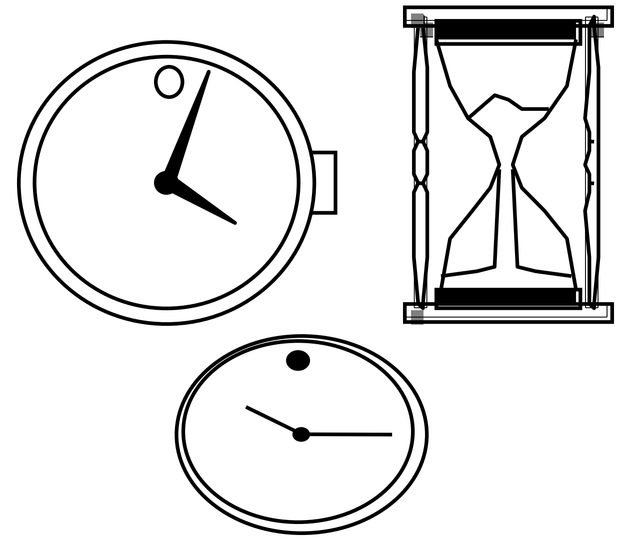
- Phenomenon
  - An object in the world of a domain as you perceive it
    - Examples: This lecture at 9:30, my black watch
- Concept
  - Describes the common properties of phenomena
    - Example: All lectures on software engineering
    - Example: All black watches
- A Concept is a 3-tuple:
  - **Name:** The name distinguishes the concept from other concepts
  - **Purpose:** Properties that determine if a phenomenon is a member of a concept
  - **Members:** The set of phenomena which are part of the concept.

Concepts, Phenomena, Abstraction and Modeling

Name | **Purpose** | **Members**

`Watch`

`A device that measures time.`



Definition of Abstraction:

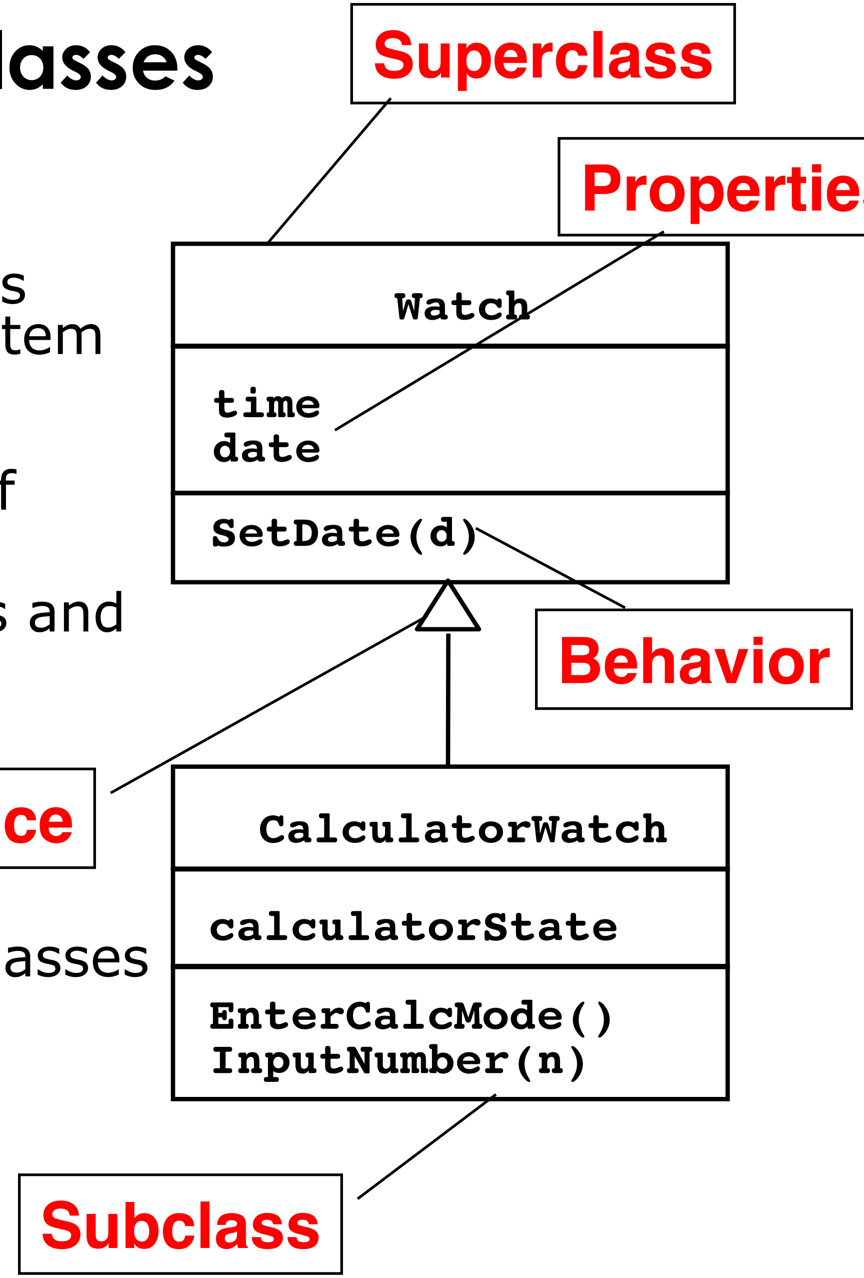
- Classification of phenomena into concepts

Definition of Modeling:

- Development of abstractions to answer specific questions about a set of phenomena while ignoring irrelevant details.

Abstract Data Types & Classes

- **Abstract data type**
  - A type whose implementation is hidden from the rest of the system
- **Class:**
  - An abstraction in the context of object-oriented languages
  - A class encapsulates properties and behavior
    - Example: Watch



Unlike abstract data types, subclasses can be defined in terms of other classes using inheritance

- Example: CalculatorWatch

Type and Instance

- Type:
  - A concept in the context of programming languages
  - Name: int
  - Purpose: integral number
  - Members: `0, -1, 1, 2, -2,…`
- Instance:
  - Member of a specific type

- The type of a variable represents all possible instances of the variable

The following relationships are similar:

Type <-> Variable
Concept <-> Phenomenon
Class <-> Object

Systems

- A *system* is an organized set of communicating parts
  - Natural system: A system whose ultimate purpose is not known
  - Engineered system: A system which is designed and built by engineers for a specific purpose
- The parts of the system can be considered as systems again
  - In this case we call them *subsystems*

Examples of natural systems:

- Universe, earth, ocean

Examples of engineered systems:

- Airplane, watch, GPS

Examples of subsystems:

- Jet engine, battery, satellite.

Systems, Models and Views

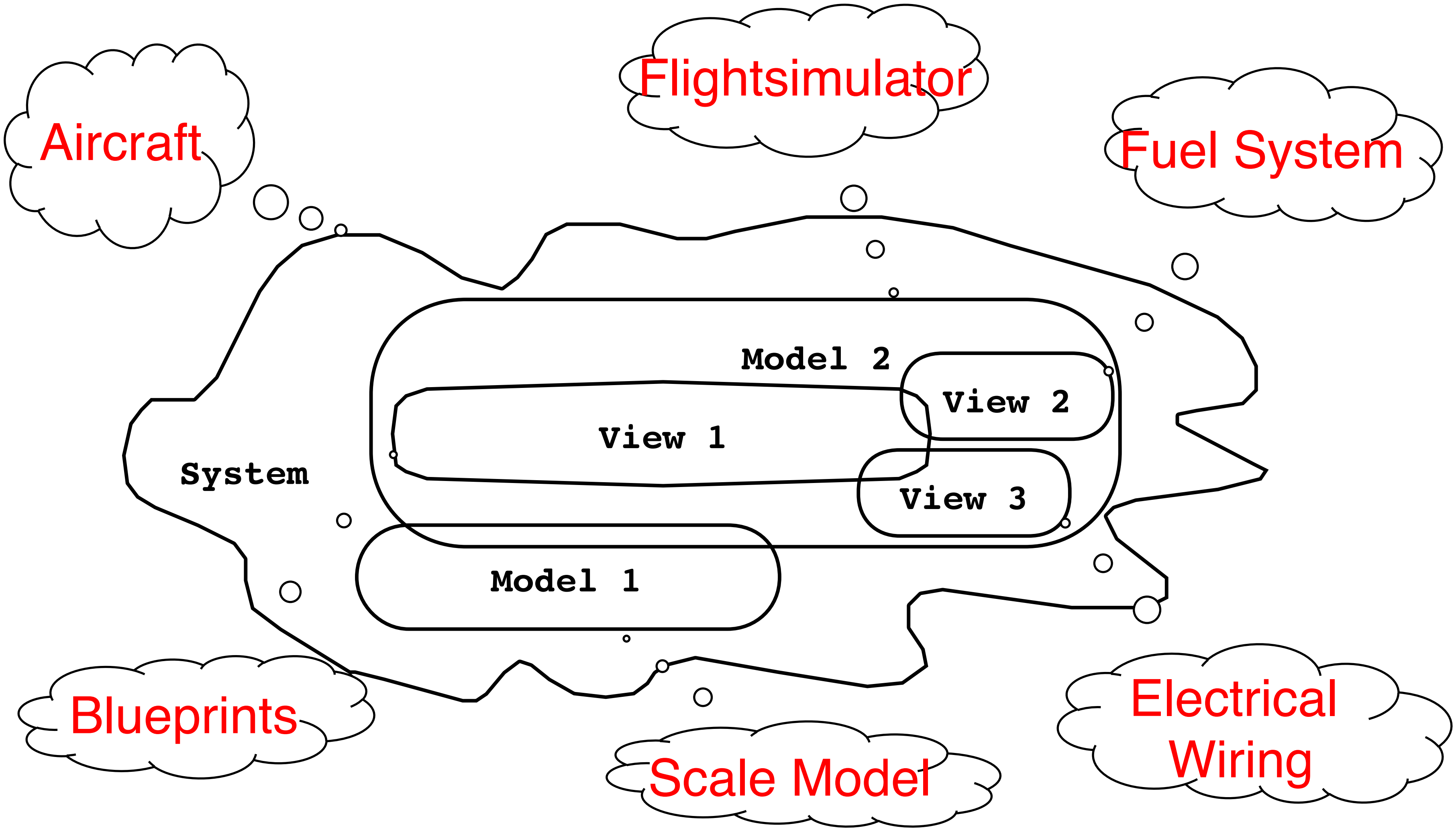
- A ***model*** is an abstraction describing a system or a subsystem
- A ***view*** depicts selected aspects of a model
- A ***notation*** is a set of graphical or textual rules for depicting models and views:
  - formal notations, "napkin designs"

System: Airplane

Models:
Flight simulator
Scale model

Views:
Blueprint of the airplane components
Electrical wiring diagram, Fuel system
Sound wave created by airplane

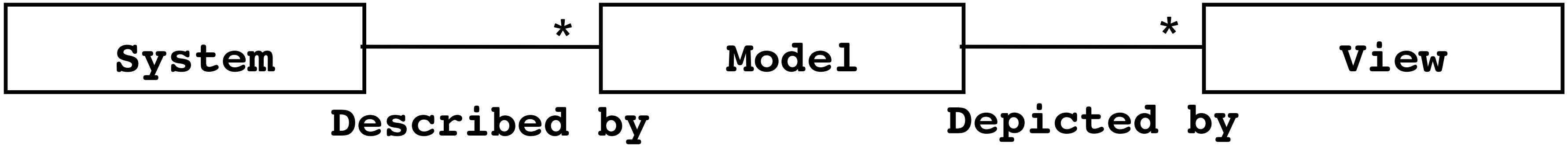
Systems, Models and Views ("Napkin" Notation)



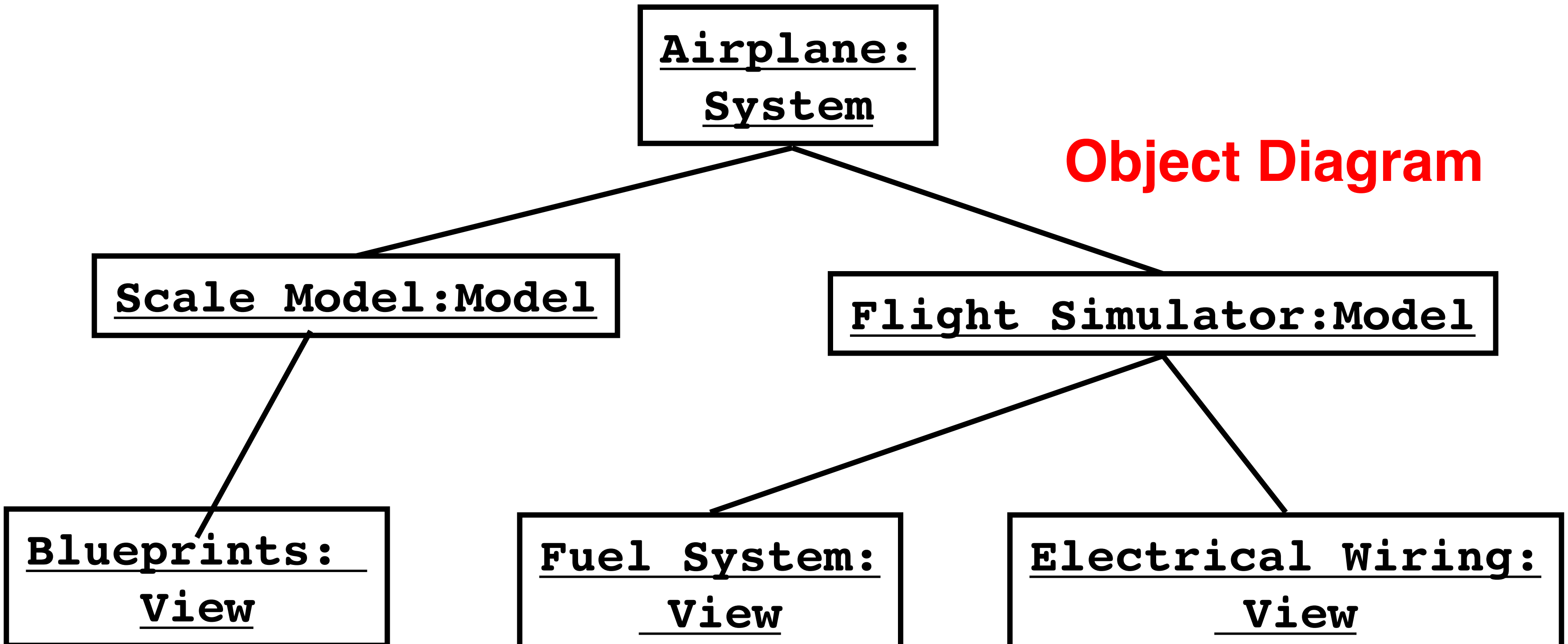
Views and models of a complex system usually overlap

Systems, Models and Views (UML Notation)

Class Diagram



Object Diagram



Model-Driven Development

1. Build a platform-independent model of an application functionality and behavior
   a) Describe model in modeling notation (UML)
   b) Convert model into platform-specific model

2. Generate executable from platform-specific model

Advantages:

- Code is generated from model ("mostly")
- Portability and interoperability

- Model Driven Architecture effort:
  - http://www.omg.org/mda/
- OMG: Object Management Group

Model-driven Software Development

Reality: A stock exchange lists many companies. Each company is identified by a ticker symbol

Analysis+ design result in analysis/design object model (Example: UML Class Diagram):

Implementation results in source code (Java):

```
public class StockExchange {
    public m_Company = new Vector();
 };
public class Company  {
  public int m_tickerSymbol;
  public Vector m_StockExchange = new Vector();
};
```

Application vs Solution Domain

- Application Domain (Analysis):
  - The environment in which the system is operating
- Solution Domain (Design, Implementation):
  - The technologies used to build the system
- Both domains contain abstractions that we can use for the construction of the system model.