# OO Analysis & Design: Modeling, Integration, Abstractions

CS577b

Spring 2001

# Modeling and the People Technology Gap

# Why Model?

- What makes computers useful?

Can faithfully represent a conceptual system in a particular context outside of real time/space.

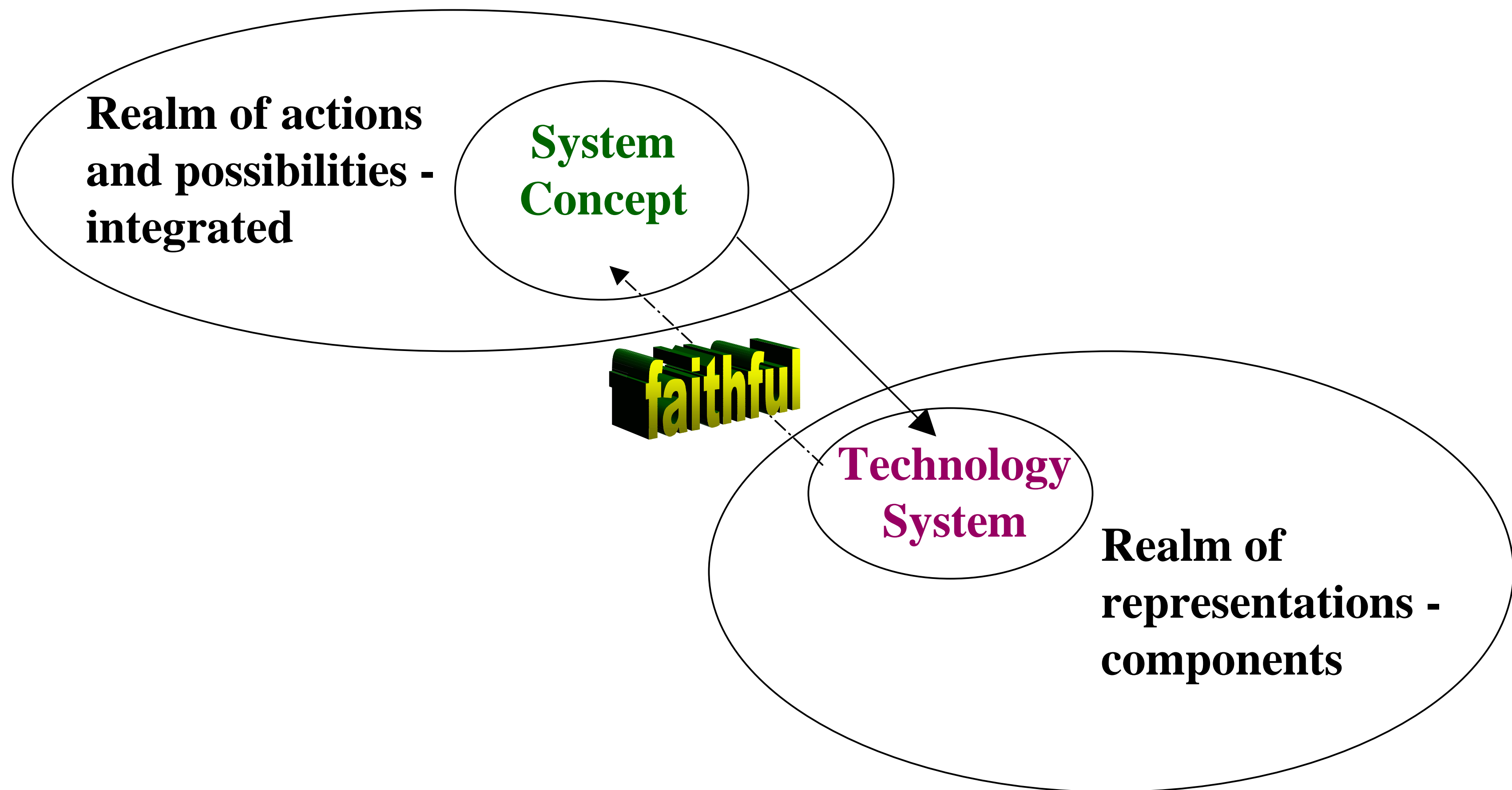
That is….

Computers support software models.
Software implementations are representations (models) of real-world conceptual systems.

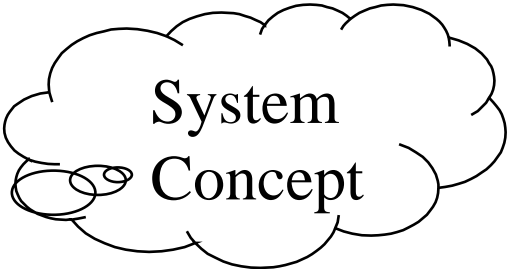
# Realms

- Systems start as people conceived ideas
- The task of development is to represent concepts with technology
- Development engineering processes move a concept from the *Realm of Actions* (concepts) to the *Realm of Representations)* (technology)

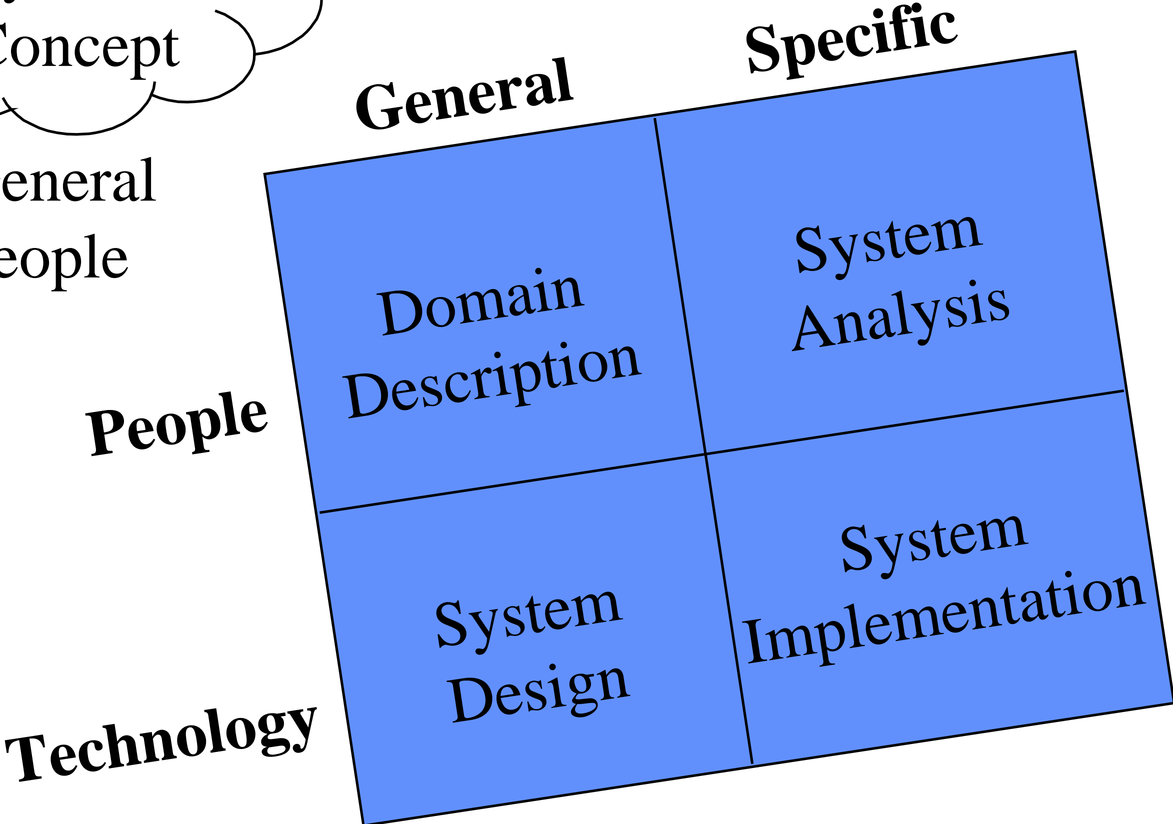
# Realm of Actions to Realm of Representations

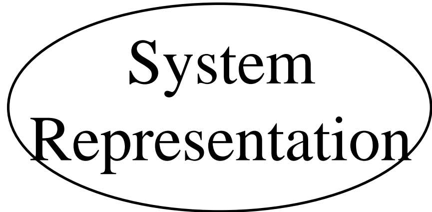


# Concept to representation gap (product models)




- General
- People




- Technology
- Specific

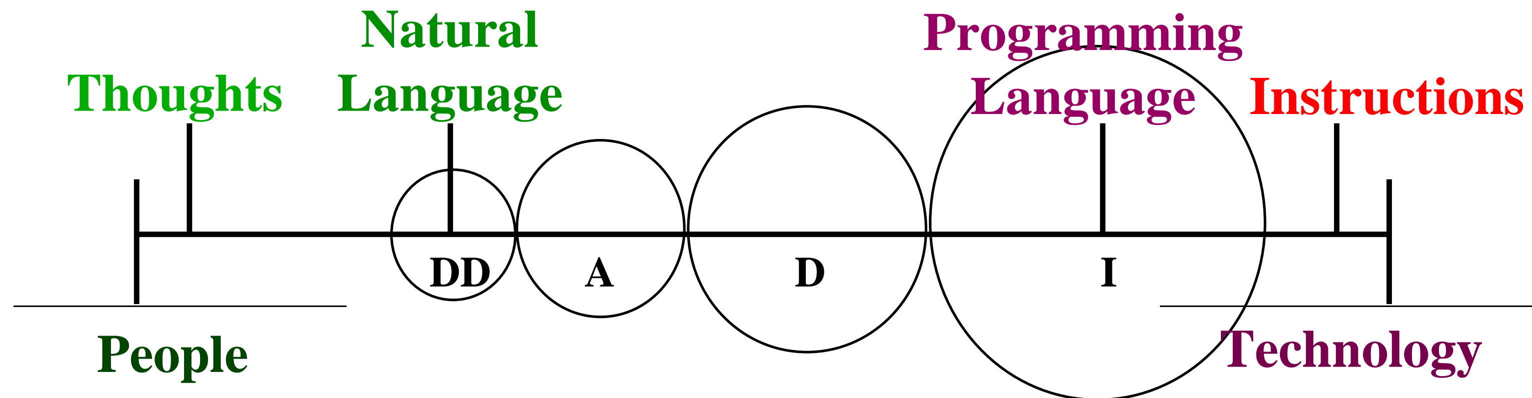
# Computer Based Models

- Computers can be used to build models
- Computer models can
  - faithfully represent a conceptual system in a particular context outside of real time/space.
  - support software models.
  - support software implementations that model (real-world) conceptual systems.

# People Based Models

- People
  - also build models
  - use language to abstract everyday experiences
  - model systems differently than technology
- A good methodology tries to resolve these differences between people and computers

# People-Technology Gap




- Development must build bridges from people to technology
- These layers translate between various audiences and languages
- Subsequent stages increase information content and complexity

# People vs. Computers

| **People** | **Computers** |
| --- | --- |
| •Non-linear | •Linear |
| •Abstract | •Concrete |
| •Continuous | •Discrete |
| •Context Sensitive | •Context Free |
| •Active | •Passive |
| •Creative | •Logical |
| •Inconsistent | •Consistent |
| •Need Donuts | •Need Batteries |

# People vs. Computers (cont.)

| **People** | **Computers** |
| --- | --- |
| •Semantic | •Syntactic |
| •Highly Parallel (small tasks) | •Sequential |
| •Approximate (PAC, PACE) | •Exact |
| •Learn | •Represent |
| •Do as they want | •Do as they are told |
| •Make Choices | •Compute |
| •Flexible (usually) | •Rigid |
| •Desires power | •Needs power |
| •Informal | •Formal |

# Model Audiences

- Different project audiences view a system concept in different ways (non-technical vs. technical)
- The final project's constituents usually do not want disruptive revolution
- The software engineer should try to evolve new structures from the existing successes within the project domain and users

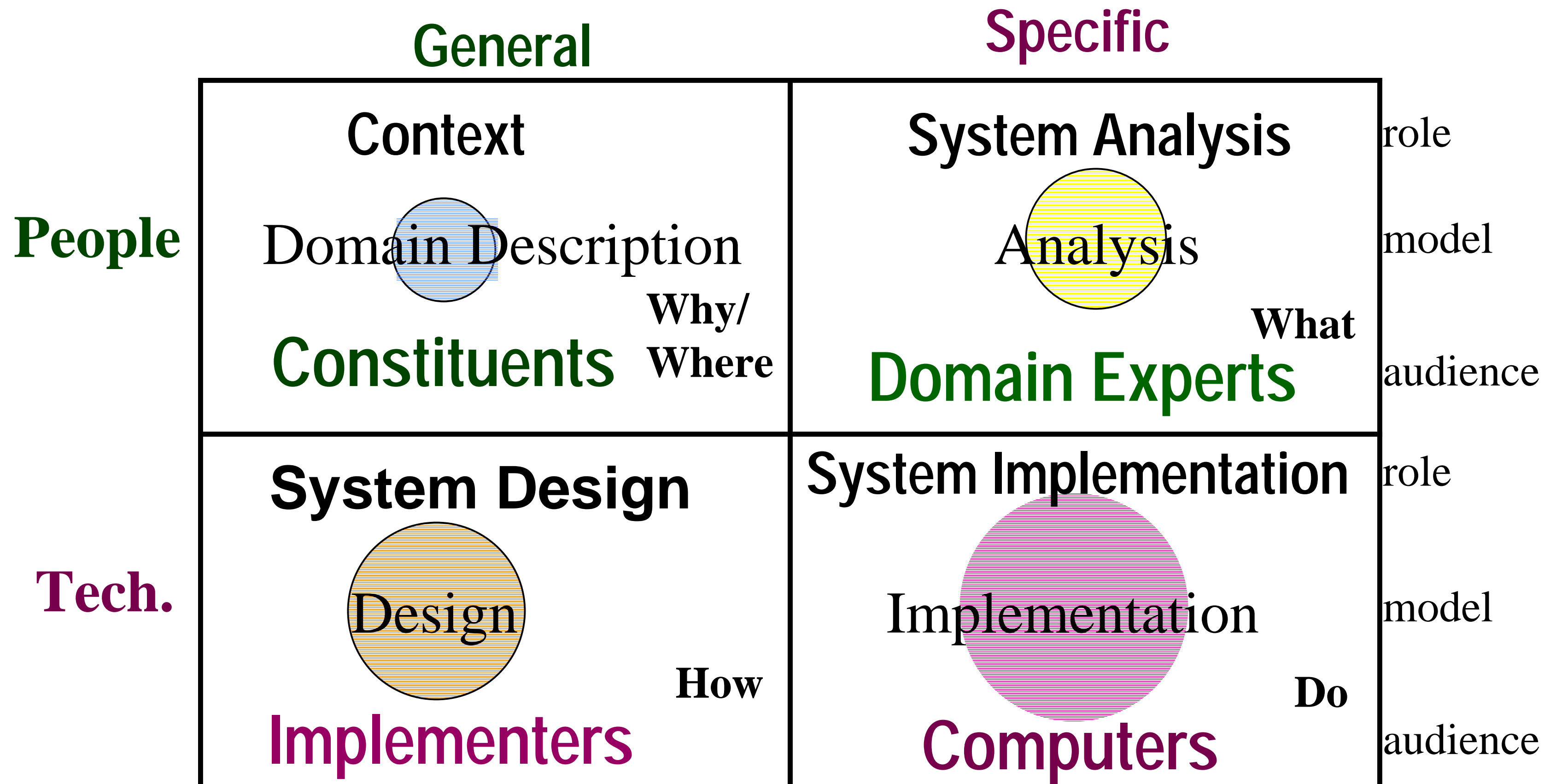
# Project Roles

- 4 different types of people/audiences involved in a project

| | Conceptual | Practical |
|---|---|---|
| User | Customers and Domain Experts | Analysts |
| Creator | Designers | Implementers |

# Audience Matrix

Each role works with a particular model and audience

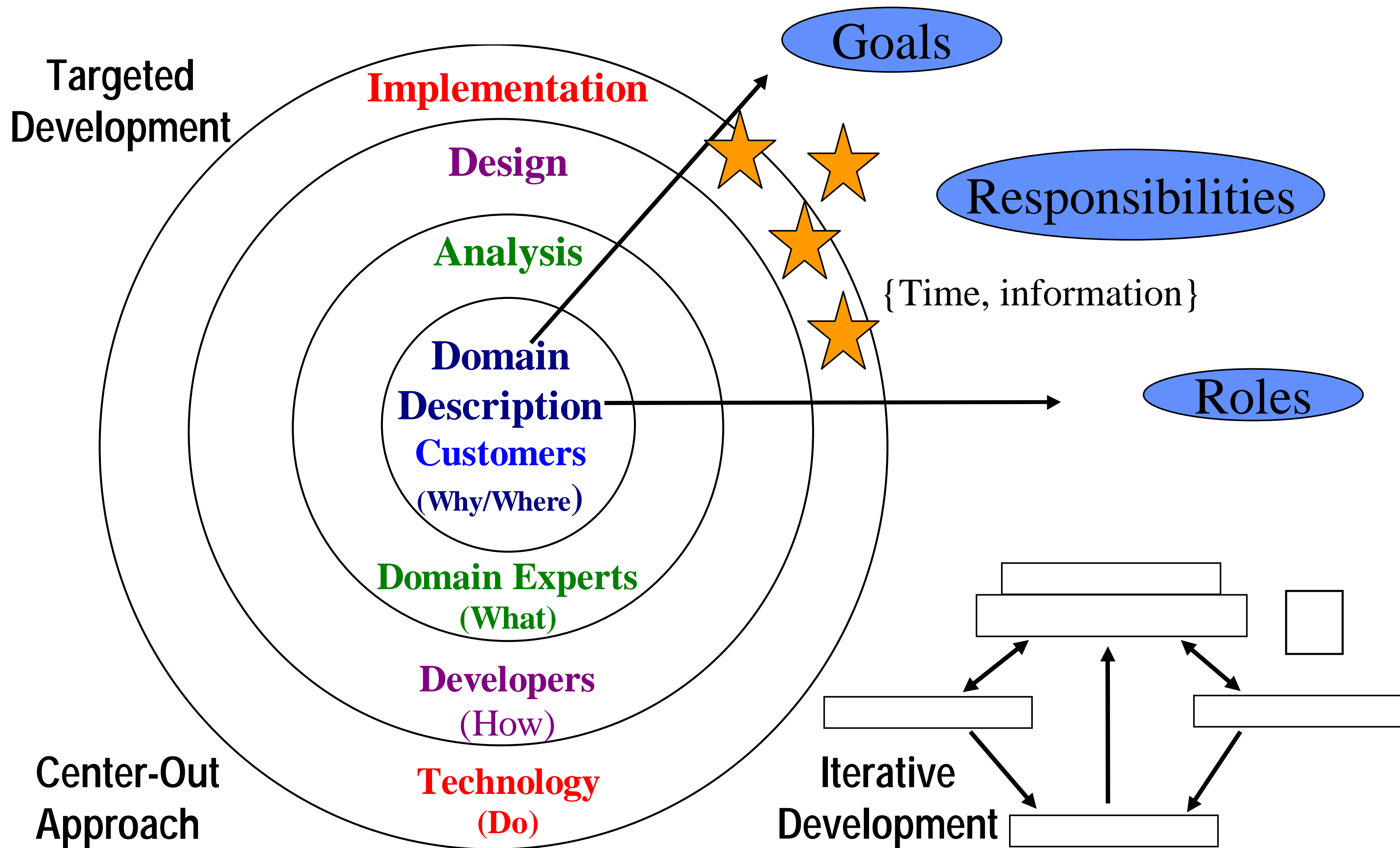


# Model Layers

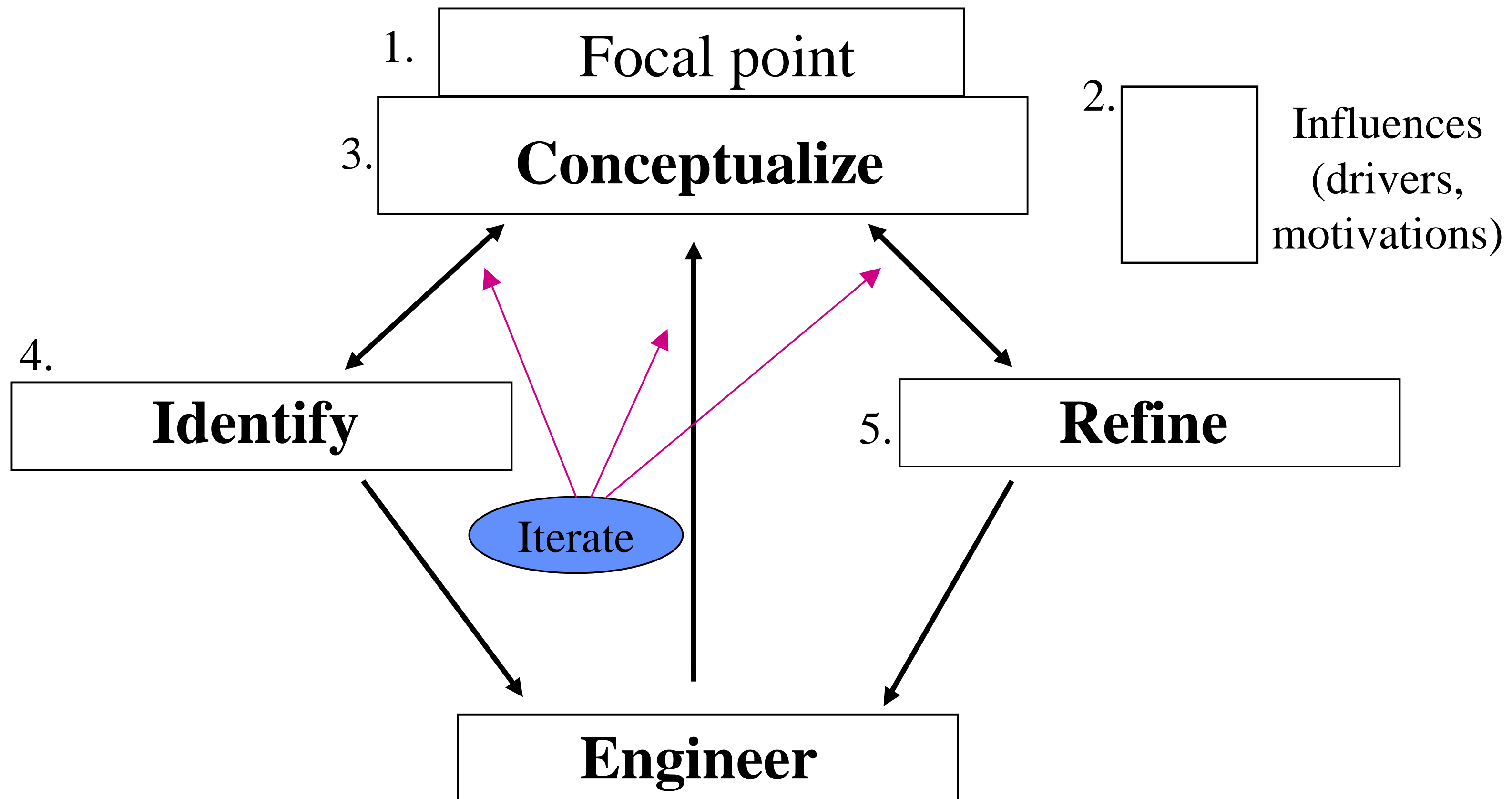
# Product Model Layers

- MBASE builds product models in stages
- Each stage refines models from previous stages
- This approach provides openings for a project to change details with out driving the project off-track
- Models allow us to hide complexity and focus on key aspects

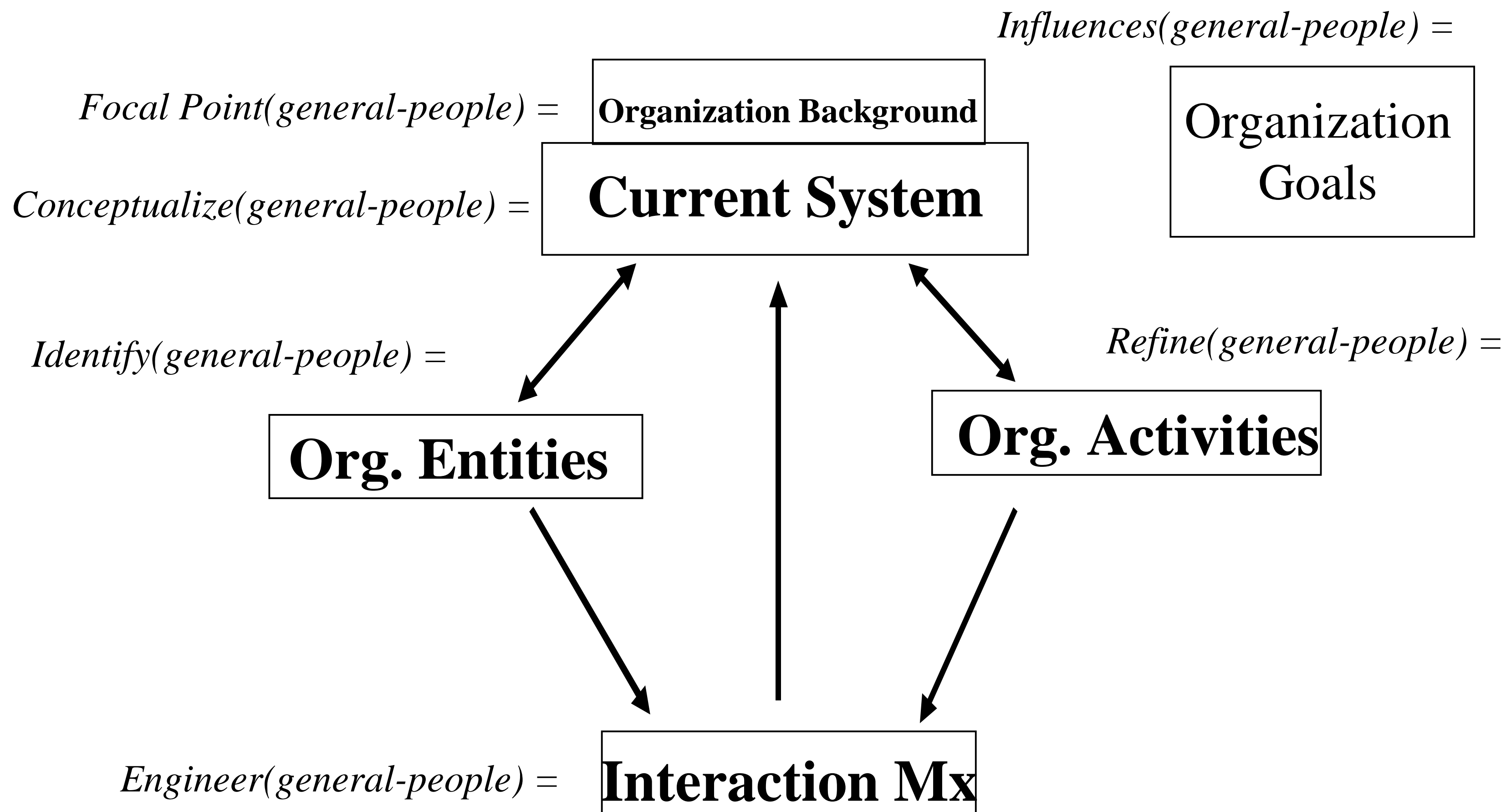
# General Center-Out Development



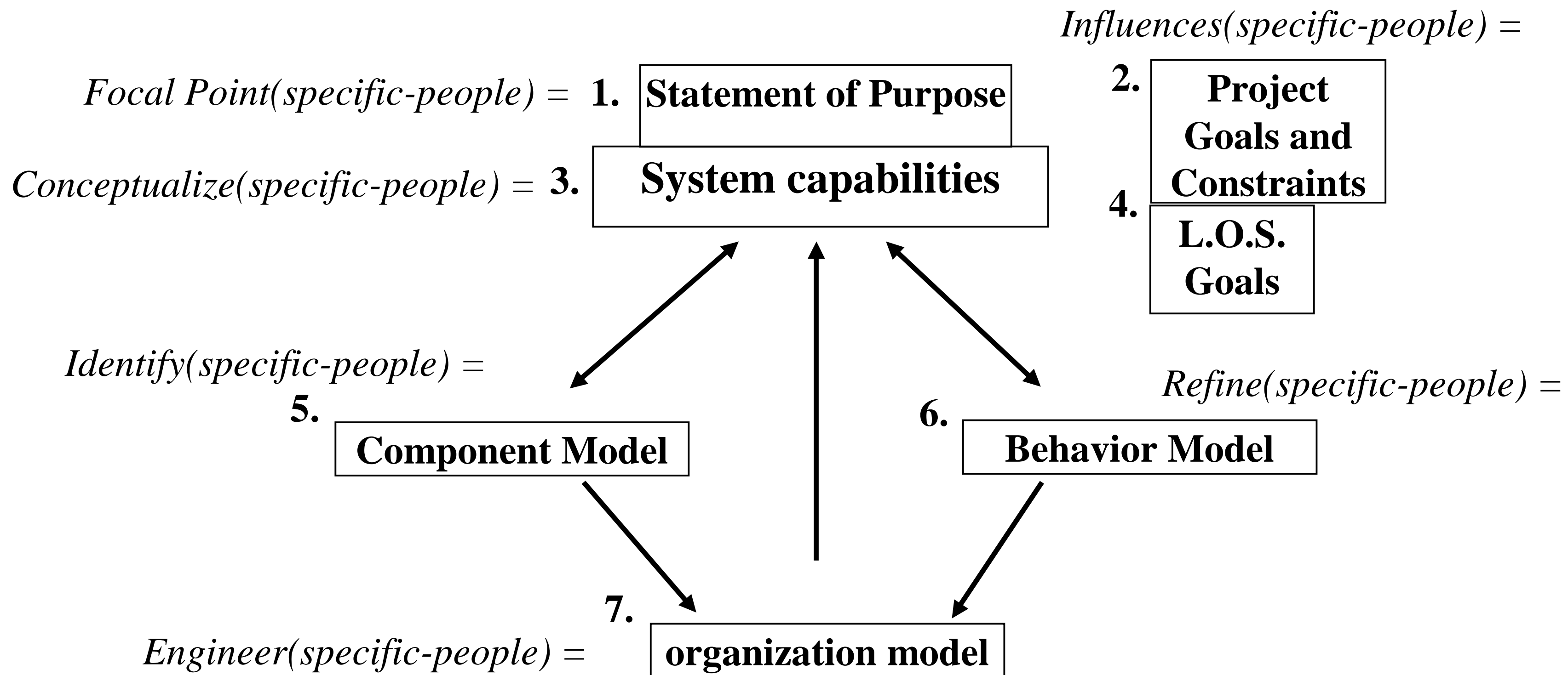
# Iterative Development



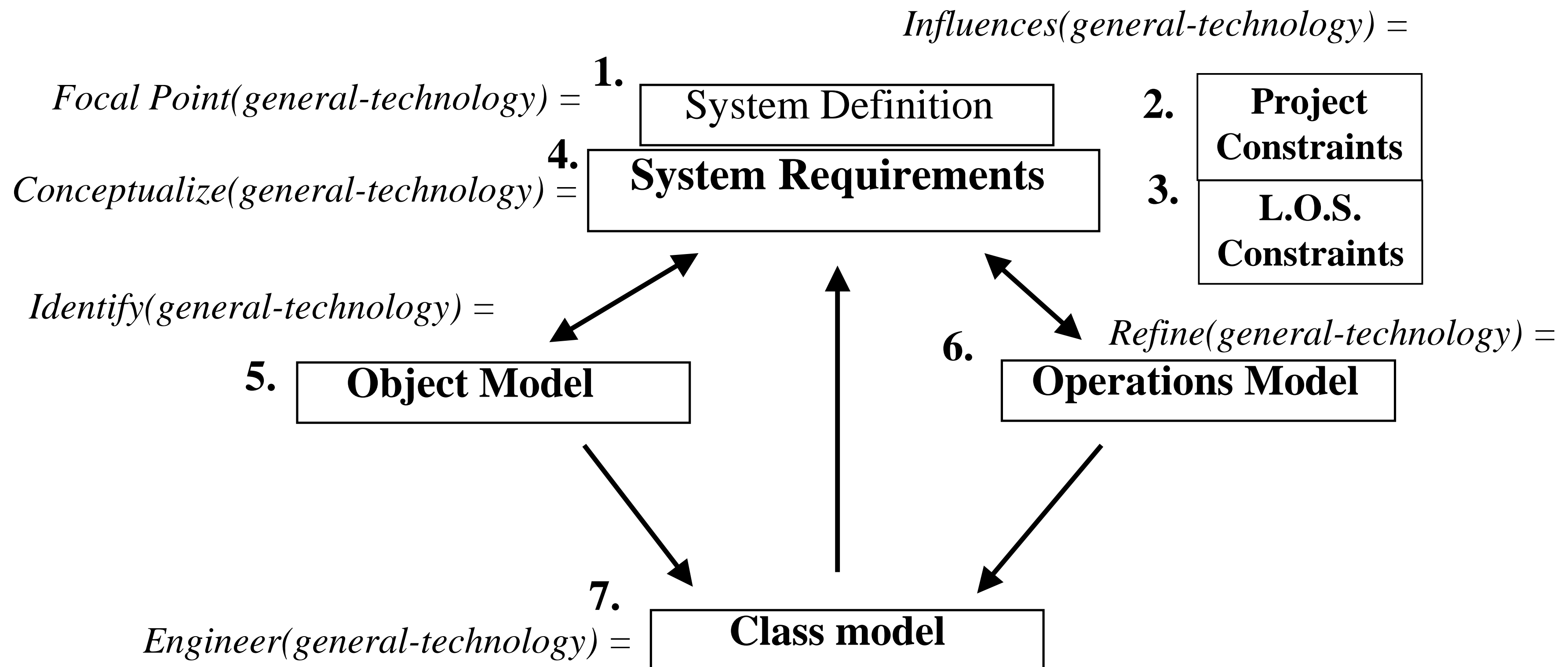
# Domain Description Models



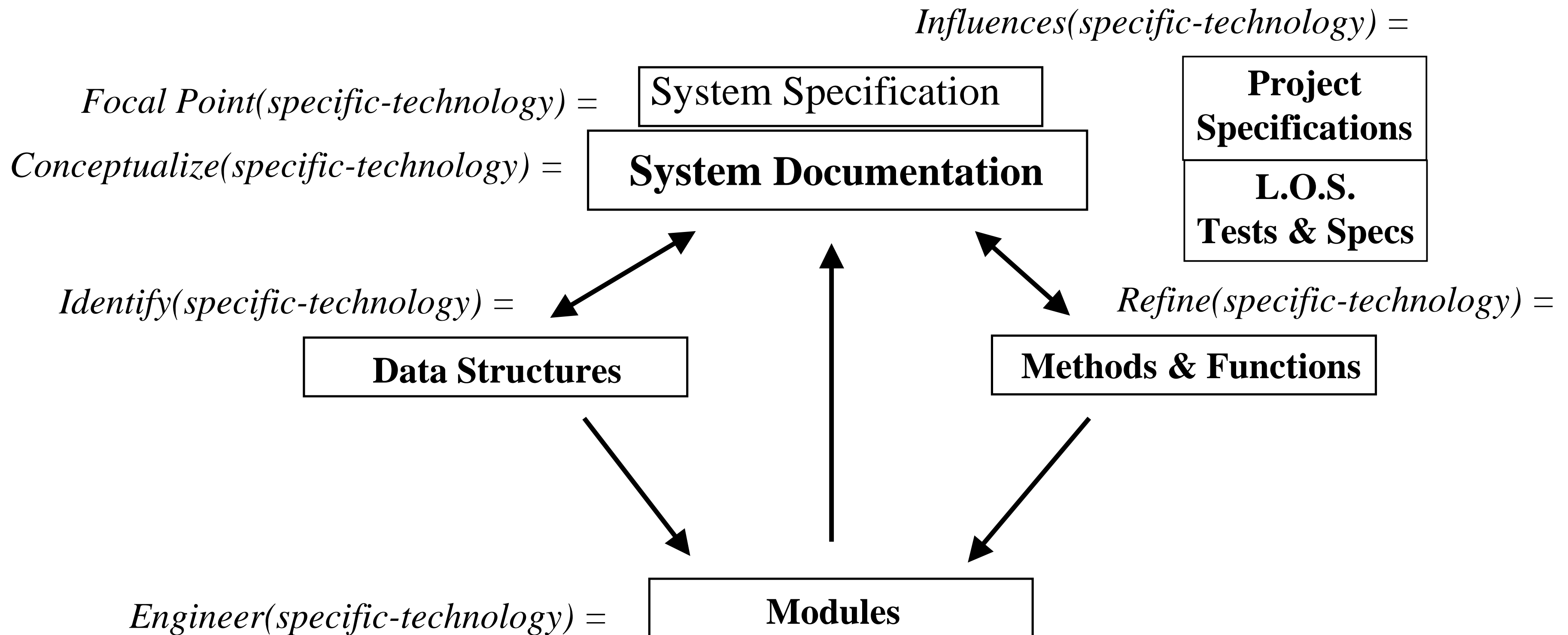
# Analysis Models



# Design Models



# Implementation Models



# Faithful Models and OOAD

# Integrated Software Models

- Software models needs to integrate faithfully
- Ultimately software interactions should map back faithfully to conceptual associations within the domain
- Model clashes occur when the underlying assumptions of a model (or models) are incompatible or insufficient

# Model Clashes: Faithfulness

- ***Consistency***:
  (I) A constraint set forth by one model element does not violate a constraint elsewhere (e.g. inter-model clashes such Waterfall and IKIWISI)
  (II) The same name is never used to denote two different model elements (e.g. domain-entities and system-components, [internal and external] increments)
  (III) Two different names are never used to refer to the same model element (e.g increment and build)
- ***Soundness***: all elements (including the product itself) deduced from a model satisfy <u>all</u> models
  - **Ex. All requirements must be consistently represented in the software**
  - **Ex. Requirements are eventually functionally decomposed into operations (operations can only be carried out by objects)**
  - **Ex. Gantt/Pert charts must represent the process strategy**
- ***Coverage***: all model elements map to other model or non-model elements
  - **Ex. All operations must eventually be mapped to objects**
  - **Ex. All level of service requirements must have a verification activity**

# Why Faithful Systems

- Faithfulness - A good software system should reflect the systems original intent and the context in which the system was conceived
- No matter how complex, an unfaithful system can destabilize a project
- Faithful systems are intuitive, predictable, and may be more robust and scalable

# Faithfulness

- Is the key to building good, reliable systems
- Acknowledges that computer structures are different than the concepts they represent
- Defines the semantics for the final implemented structure via the original concept it represents
- May not always be achievable, but MBASE tries to achieve faithfulness at least locally

# Modeling Framework

- Models help manage complexity and reflect the real world
- Without a proper framework, model clashes can cripple a development effort
- Many modeling techniques exist, but to be effective, it is essential to manage communication between various modeling stages of large software development efforts

# OOAD

- Object Oriented Analysis and Design (OOAD) provides a modeling framework as a set of techniques to
  - Build faithful models
  - Build natural, tangible structures.
  - Manages complexity of sophisticated systems
  - Help guarantee the end users get an effective, usable system implementation

# OOAD Goals

- Use success-based practices
- Manage complexity
- Provide metrics
- Build Effective documentation
- Give Assurance, by defining and realizing desired properties and achieving measurable goals.

# OOAD Goals (cont.)

- Raise awareness of what we do as developers
- Motivate and Discover
  - "**What**" - works and does not work?
  - "**Why**" - (the key to reproducibility and scalability)
  - "**How**" - something can be done, and how can we be more effective.
  - ”**Do**" - Carry out implementation, with useful documentation and effective communication

# OOAD Goals (cont.)

- Provide a framework to answer difficult questions, such as:
  - What are the entities/components in our domain?
  - When is a component/object “too big?”
  - When is one class/structure “better” than another?
  - When should we subclass?
- Raised awareness equates to conscious choices, resulting in better systems

# OOAD Model Layers

- OOAD naturally falls into four development strata (for reasons stated previously)
  - **Domain Description**
  - **System Analysis**
  - **System Design**
  - **Implementation**
- Each project decides what emphasis to place on each strata

# Why Analysis and Design

- Analysis and Design help us to engineer large software projects
- **Analysis** deals with "high level" concepts in a flexible way
- **Design** quantifies analysis concepts into detailed models without regard to implementation details

# Why OOAD

- Objects combine entities and behaviors into named quantities
- Objects promote
  - organized, intuitive projects
  - re-use of structures
  - metrics to measure progress
- OOAD uses object concepts to organize and drive phases of a project life-cycle

# OOAD Methodology Framework

- Methodologies combines decades of heuristic experiences of successful and problematic software projects
- Many methodologies and approaches exist, and your choices may be partially based on taste
- A framework for OOAD assists the project life-cycle to realize it's goals and deliver a usable, intuitive system
- The MBASE framework supports many methodologies

# Objects

- Represent the way people usually abstract everyday ideas and structures
- Let software developers concentrate more on real world processes rather than computer processes
- Help hide technical complexities
- Support reuse of intuitive collection of code and software systems

# Review of Model Layers

*Domain Description*

*System Analysis*

*System Design*

# Domain Description

# Domain Descriptions

- Specify and justify:
  - **Why** the system is being built
  - **What** overall organization goals and activities the project will support and be responsible for when deployed
  - **Where** the project is starting from
    - What is there already available to build from, and what else is missing and needed.
    - What context do we build in and for

# Context Setting

- Methodologies help a project meet a list of pre-formed desires
- A domain description helps keep these desires in harmony with actual activities and goals of the organization
- The domain description captures subtle "context sensitive" notions that are invisible to "context free" technology

# Domain Description

- Serves as the starting point of a project
- Uses the organization's goals to help describe the entities and operations within the current organization relevant to the project
- Sets the boundary for what is relevant and what is not relevant
- Helps with later decision making as more information and details emerge

# Motivating a Project

- A domain description provides a natural starting point for a project, but specifying agreed terminology to describe the existing organizations architecture
- A strong domain description takes extra time, but helps guide a project
- Without a solid domain description, projects can become less intuitive, unmotivated and drift off-track

# Domain Description Purpose

- Create a concise, but not precise, description of the basic organization for which the project will be built
- Establish the context within which the project will both be developed and will operate
- Determine what is or is not relevant to the project
- Provide a familiar starting point

# System Analysis

# 4. Proposed System (Analysis of)

- System Analysis involves several steps
  - Components - models, attributes, relationships, constraints, roles, and states
  - Behavior models
  - Engineering - Abstraction, enterprise class engineering
- This section is an overview of Analysis for OCD 4.0
- Details follow in later sections for SSAD

# System Analysis

- The creation of precise, consistent description of a conceptual system in terms of its high-level components
- Description is within the organization domain, independent of implementation
- Analysis goes beyond simple checklists and pictures
- Analysis ties the domain description to the system design and implementation

# Analysis Defined

- A separation of a whole into its component parts
- An examination of a complex system, its elements, and their relations
- A statement of such an analysis
- A method in philosophy of resolving complex expressions into simpler or more basic ones

# Analysis Goals

- Quantify *what* we want to represent, not *how* it is done
- Formalize and refine the specific parts of the organizations capabilities, entities, activities, and interactions described in the domain description that are to be automated
- Capture the high level architectural information that will represent (I.e. model) the conceptual system

# Analysis Audience

- The Domain Description is for all constituents of the project
- Analysis is for Domain Experts - the high level leaders who understand the domain, know what they want, and have the authority to make decisions
- Not for implementers, who prefer design and implementation details ("hows")

# Touring OOA concepts

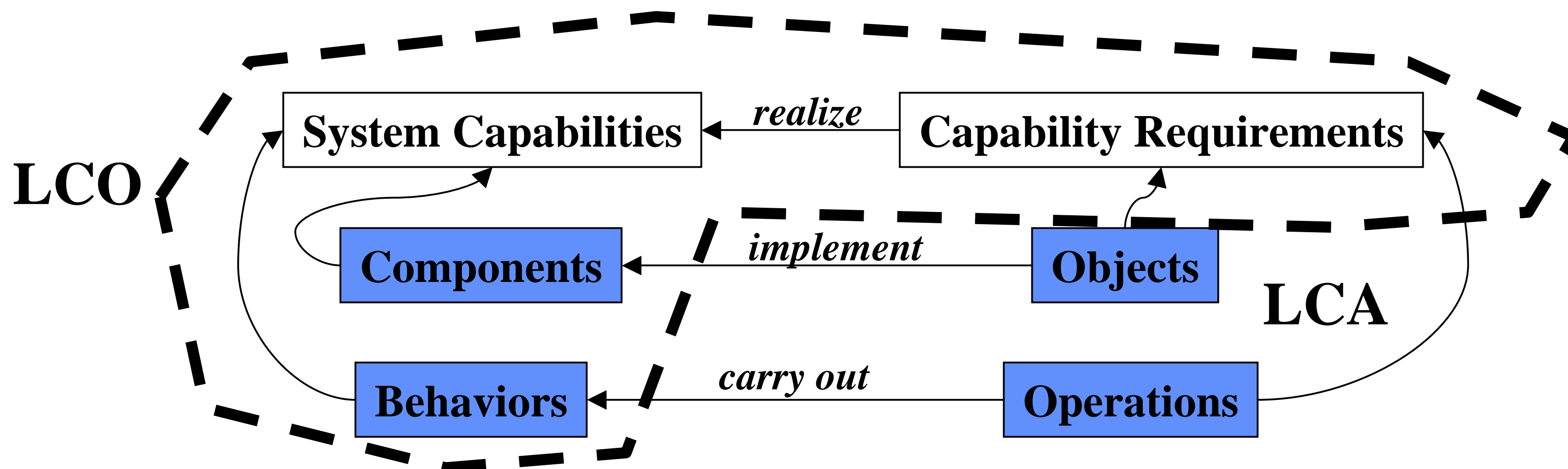
- This section introduces OOA concepts via extended examples
- The example will illustrate a few issues that can arise and how they are handled
- Notation, terminology, etc. not as important as awareness of issues
- Further study will be required

# 2. Architectural Analysis

- Main intent is to describe precisely "what" the domain experts vision of the system without implementation details
- As described via Component/Object Oriented Analysis (C/OOA)
  - Component, Behavior, Enterprise models
- Defines the “high-level” architecture of the system in a more precise way than the OCD

# 2.1 Component Model

- High-level architectural decomposition of the target system into functional partitions
- Should be consistent with System Capabilities(OCD 4.3)
  - Requirements in SSRD 3.0 will be based on what is mandated to be built in order to realize these capabilities
  - Object models in SSAD 3.0 will be derived from components in such as way as to realize the requirements in SSRD 3.0
- Use ISDM Component modeling techniques



# Component Modeling

- Output is a collection of *diagrams* and *specifications*
- Provides details on important component qualities required for a *faithful* implementation
- Serves as the overall view of the system "parts" and their relationships (a system *partition*)

# Component Modeling

- Why model components?
  - A language is needed to explain why a component exists or not, and to get a handle on the "things" which comprise the system
  - Systems are not often built entirely from scratch anymore, identification needed for integration
- How the components can or will be implemented is a design issue (will discuss in OOD)
- All components should be faithful as determined by Domain Experts (i.e. ask them!)

# System Design

# 3. System Design

- Details on <u>how</u> the system can be represented in software
- Describes specific technology solutions that meet requirements (both project and system)
  - high-level: resolves Analysis issues
    - how will roles and states be handled, expand bi-directional relationships, break multi-way relationships, handle global and relational attributes, decompose Components into objects, complex dependencies and other constraint
  - low-level: direct implementation considerations
    - use of databases, web-servers, hardware, critical algorithms, sequence, significant events, GUI's, etc.

# What is OOD?

- Mapping Analysis models to software
- Focus on finding software representations and solutions to implementation problems
- May target particular technologies
- Addresses:
  - UI's
  - DB descriptions
  - Design constructs (e.g. custom value objects, layers, ...)

# Purpose of Design

Describes how the system components are to be realized within specific software structures

Answers: “What are the fundamental objects (and their relationships) that can faithfully represent the system components?

# OOD Main Tools

- Formalized conventions
  - Address many implementation details
- Analysis models
  - Traceable evolution from what is wanted
- Architectural views
  - Indicate how to implement Components in software
- Design Patterns
  - Can help with common complex implementations
- Frameworks and mechanisms
  - Provide simple solutions to common problems
- Documentation (models) for implementers

# Design resolves Analysis issues

- Resolve policies by finding “algorithms” to carry out
- Resolves complex relationships between components
  - bi-directional, multi-way
  - memberships
  - multiplicity (containers, selectors, etc.)
  - relational attributes
- Enforces constraints (especially dependencies)
- Resolves roles, complex states, sub-types, modes

# Design models may include COTS

- COTS solutions
  - mechanisms
  - frameworks
  - API's
  - sub-systems
  - entire applications
  - Software libraries
  - Repositories

# Design Approaches

- Not an exact science
  - many many approaches, variations, choices, solutions
  - an “art” but has basic principles to follow
  - often no single “right” answer, usually “will work”, “is feasible”, or “best we can do for now”

# Components vs. Objects

- **Design Objects** are the smallest (most refined) entity we consider in our models prior to implementation
- **Components** are compositions (membership relationships) of objects with a high degree of *cohesion* within the domain
- Components are what you need to describe the system to domain experts at a higher level of abstraction (less detail)

# Objects

- The Design phase may decompose components into objects.
- Objects are used to represent the system in software.
- An object is a specialization of a component.
- An object is an atomic unit for systems analysis purposes.

# Design Preview: Java Composites and Components

- Design Composites:
  - collection: Vectors
  - aggregation: HashTable
  - grouping: Array
- Java Components:
  - “Component”
  - Canvas
  - Window

# Implementation

- Implementation is the final stage of a project
- OOAD constrains a project to create *faithful* implementations
- Involves language choices - Java, C++, HTML, Perl
- Java features support technology integration (JDBC, RMI, Media Frameworks, JNI)

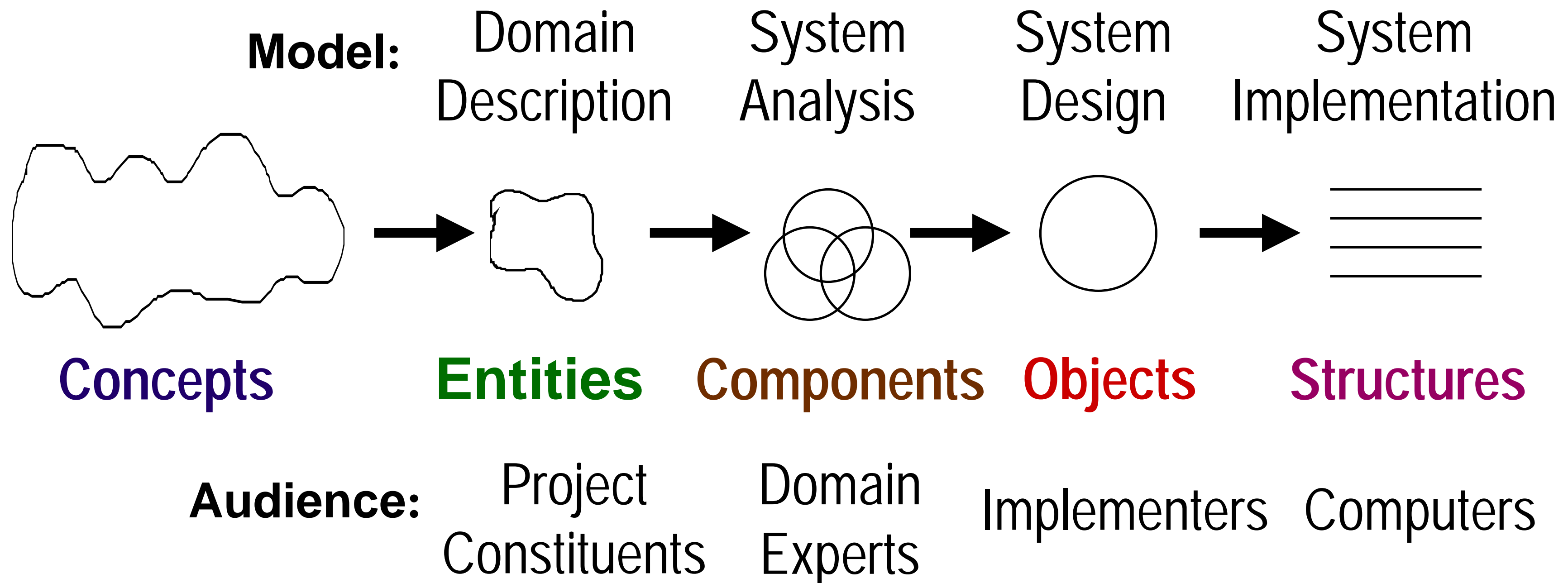
# Evolution

*About the homework…*

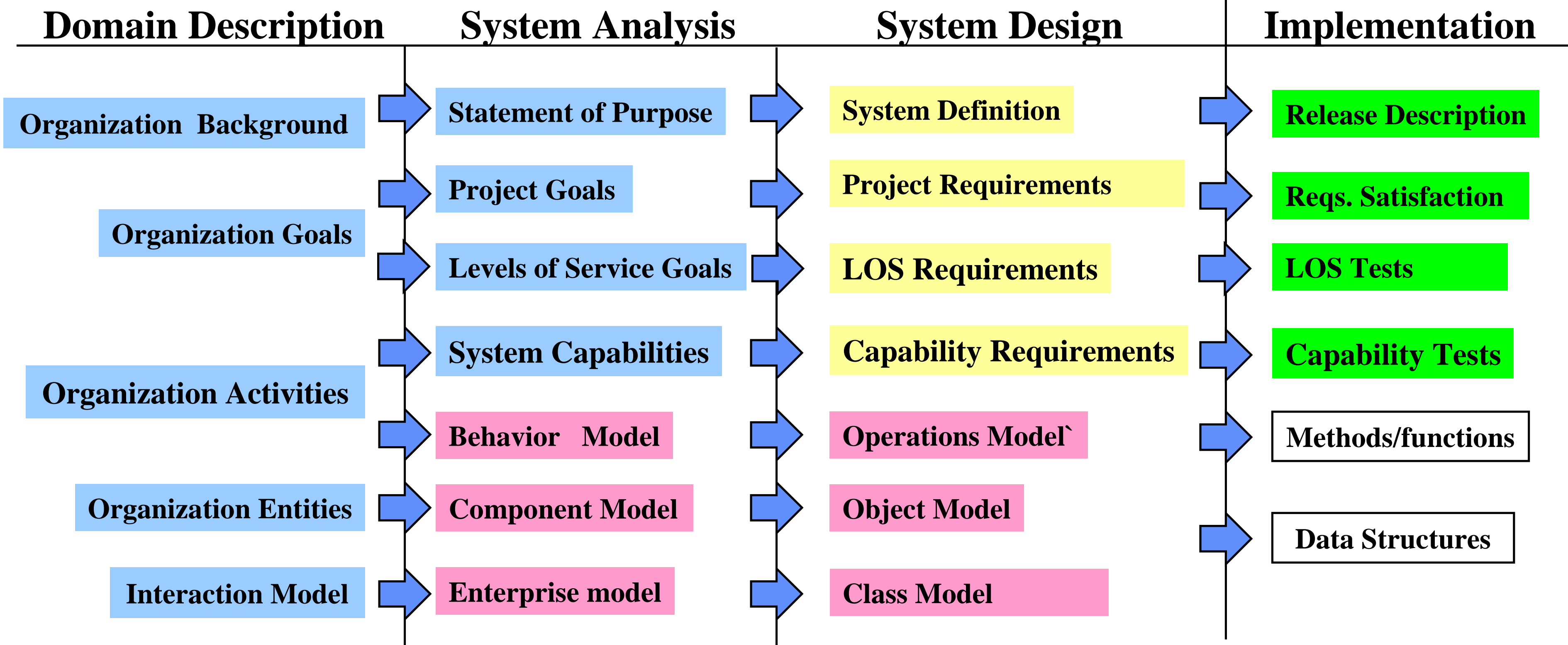
# Evolutionary Modeling

- Different types of audiences move a software project from conception to implementation (customer to software)
- Breaking the evolution into natural phases helps manage complexity
- Avoiding jumps to implementation can free up a project to have a more effective overall architecture

# OOAD Evolution



# Coverage/Traceability of MBASE Product Models*




Operational Concept Description (OCD)

System and Software Requirements Definition (SSRD)

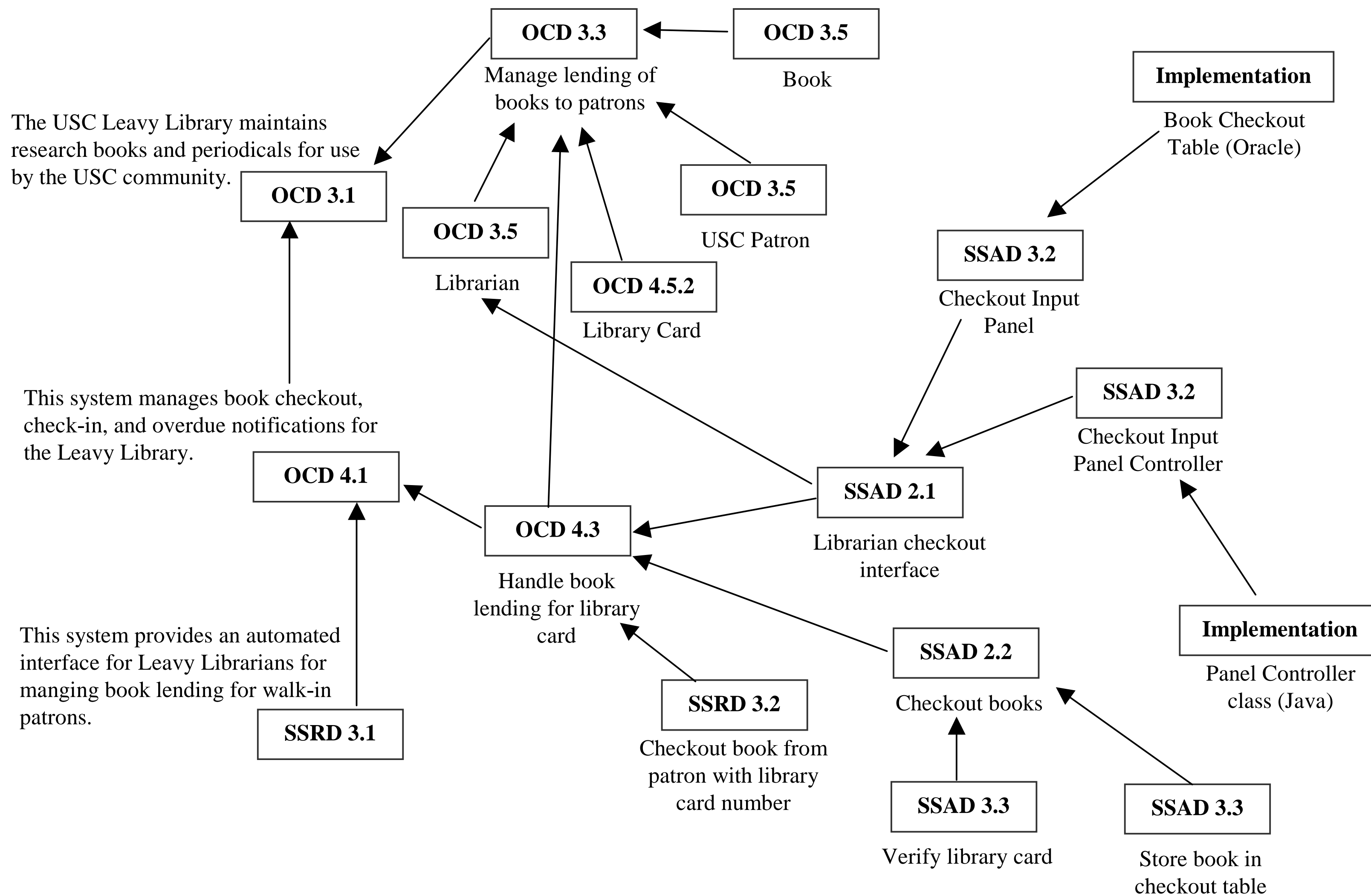
System and Software Architecture Description (SSAD)

Construction,Transition,Support (CTS)

External to MBASE

** Does not include all MBASE models*

# Example Trace Map



# The Language of Abstractions

*The foundation of all models and modeling*

# Abstractions

- Are representations of things/information
- Simple interfaces to complex information
- Form the basic elements that constitute our models
- Are the language we use throughout the modeling process
- Are created within a context, with defining qualities within the context

# Example: Abstractions

- Window in the context of parts of a building
  - quality: “An opening you can look through”
- Airplane in the context of things that fly
  - quality: “Human built machine that carries people through the air quickly”
- Airplane in the context of things you can reserve a seat on
  - quality: “All carry-on luggage must be stowed”

# Language of Abstractions

- Abstractions are
  - used to build models
  - closer to natural language, but can progressively become more technical
  - specified by a <u>defining quality</u>
  - defined within a particular context

# Abstractions

- Abstractions let you
  - focus on the “big picture” and problem domain
  - specify the problem before the solution
  - identify relationships that span project boundaries
  - allow for polymorphisms: two abstractions can have the same defining quality
  - directly engineer the defining qualities to increase faithfulness

# OO models and abstractions

- Models provide a tangible way to work with groups of complex concepts through abstractions
- Enable management of complex sets of abstractions
- Offer multiple views of abstractions for a variety of purposes

** All models use the language of abstractions **

# Concepts and Representations

- Concepts and representations of concepts
- Concepts & Representations
  - Concept - more refined thought
  - Representation - proxy for a concept
- Example:
  - Concept: Customer, Products
  - Representations: Credit Account, Inventory DB

# Data vs. Information

- Data - collection of facts or results
  - Ex. Raw scores on an exam
- Information - processed data (significance is assigned by an audience)
  - Has an intended meaning with context and relationships directed toward a particular audience
  - Ex. Average score on an exam

# Example: Data and Information

- Data
  - Object name and fields
  - Event objects
  - Interface implementation
  - Argument to a method
  - Attribute value
  - Exception Objects
- Information
  - Result of some computations
  - Typing an object with an interface
  - Caller of a method
  - Attribute owner
  - Exception name used in a catch block

# Information vs. Encapsulation

- Putting bits in a bucket gives you a bucket of bits (easier to carry, but still data)
- Encapsulation hides complexity such as implementation details
- Encapsulation itself does not necessarily add information
- Information always has intended meaning

# Example: Information and Encapsulation

- A Clock:
  - Information - The current time relative to a place on earth
  - Abstraction - Represents time and it's defining qualities (year, day, hour, minute, second, alarm)
  - Encapsulation - The display is an interface to hide the complexity of the mechanics (chips, gears, atomic vibrations, net time, etc.)

# Encapsulation: Java

- Encapsulation in Java is performed through new class definitions
- Encapsulation abstracts data and behaviors into related groupings
- Good OOAD can make the actual Java encapsulation stage natural and automatic
- Class names can provide an intuitive context for naming attributes and methods

# Interfaces and Information

- Abstractions provide interfaces to their data and behaviors
- Represented information is provided by interacting with an abstraction’s interfaces, not with the abstraction instance itself
- Interfaces must be smaller than the information they represent, and can set the context for presenting information

# Example: Interfaces

- A clock has a display, knobs, etc.
- You don’t tell time by looking at the gears or reading the chip impulses directly

# Abstraction Qualities

- Abstractions are specified by their qualities, defined within a particular context
- A *quality* is an atomic unit of representation within an abstraction
  - Some qualities may themselves be abstractions
- Example: “Employee” has qualities of “name”, “education”, and “job title

# Quality Resolution

- **Quality Resolution** - the specific information the quality represents when applied to a particular instance of an abstraction (object)
- **Constraints** - are restrictions on the resolution of qualities
- Example: “color” of a “Fire Engine” often resolves to “red”, but not always

# Defining Quality

- Is a parsimonious collection of qualities that give an abstraction it's identity
- Does not create an objective definition, since they are *defining* to a specific context and audience
- Context and defining qualities relate and influence each other
- Choosing a good defining quality is tough

# Precision and Coverage

- Are used to invoke the precise representation of the abstraction in the minds of the audience
- A *precise* quality excludes everything that is not relevant
- A quality provides *coverage* if it includes everything it should

# Inclusion-Exclusion Algorithm

- Technique for balancing precision and coverage in a quality
  - 1. List qualities. If all done, goto 3.
  - 2. Adjust defining qualities to include 1.
  - 3. List qualities the abstraction shouldn't have. If no more, you are done!
  - 4. Adjust defining qualities to exclude all of 3.
  - 5. Go to 1.

# Name of an Abstraction

- The purpose of the name is to invoke the defining quality in the minds of the audience
- The name should always reflect the defining quality
- **WARNING:** As tempting as it is, do not name an abstraction before specifying its defining quality

# Section : Elegance

# Abstraction Elegance

- Abstractions are representations of information
- Encapsulation is an interface to an abstraction
- The interface may in fact hide information
- *Elegance* deals with maximizing the information delivered through the simplest possible interface

# Measuring Elegance

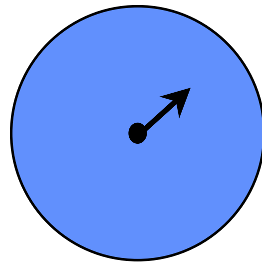
- Elegance is not always quantifiable, but is an important consideration
- Working measure of elegance: *Information to interface ratio*
  - A qualitative measure, not quantitative
  - Uses relative, indirect (not absolute) comparisons
  - A very useful heuristic

# Factors That Affect Elegance

- Well specified, relevant defining quality
- Name - should always reflect the defining qualitiy
- CCSC - completeness, cohesion, sufficiency and coherence

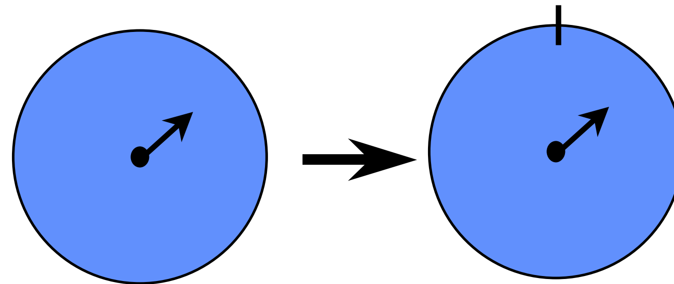
# Completeness

Is all the information represented in the interface?



# Completeness

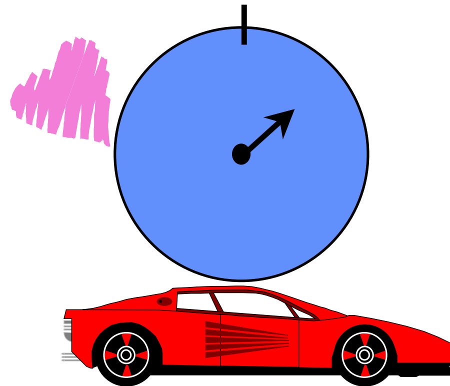
Is all the information represented in the interface?



**Incomplete interfaces are common in OO**

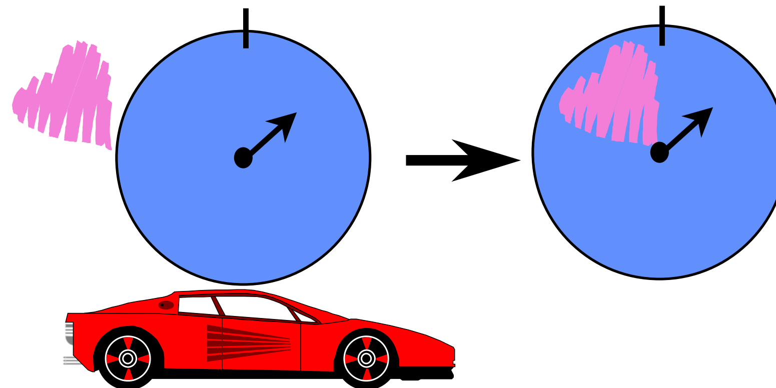
# Cohesion

How well do the qualities fit together?



# Cohesion

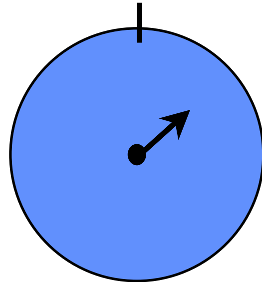
How well do the qualities fit together?



**Increase information-to-interface ratio by combining qualities in a way the audience anticipates or expects**

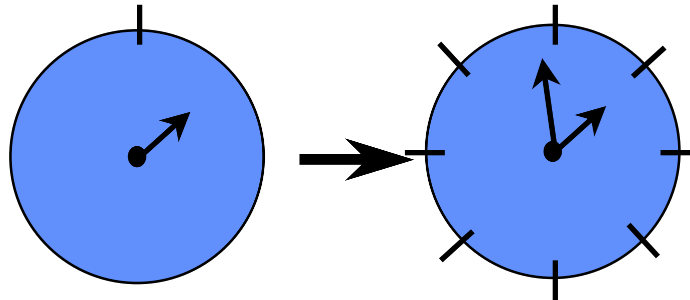
# Sufficiency

How easy is it to access the information?



# Sufficiency

How easy is it to access the information?

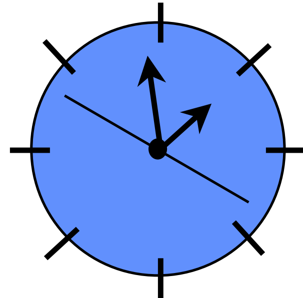


**An increase in access time decreases the amount of information that can be conveyed in practice.**

**Abstractions are often complete but overly terse.**

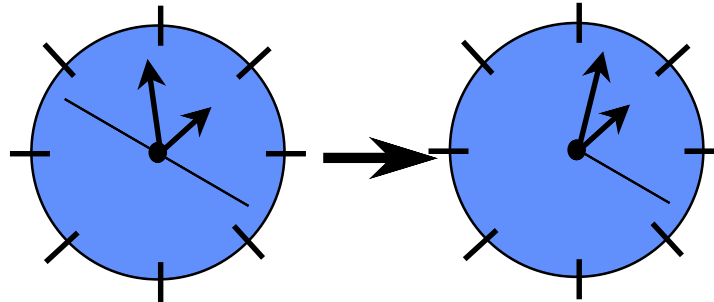
# Coherence

Do the qualities resolve correctly and consistently?



# Coherence

Do the qualities resolve correctly and consistently?



**Applies not to the qualities, but *how* they resolve.**
**Sometimes this is very subtle!**

# Example: Software Elegance

- Issues of elegance in software are reflected in
  - Robustness
  - Scalability
  - Flexibility
  - Usability

# Summary: Elegance

- Elegance is a subtle but important factor
- Elegant abstractions maximize information with simple interfaces
- Defining qualities and names influence elegance
- Completeness, cohesion, sufficiency, and coherence support elegant models