

SOFTWARE ARCHITECTURES

prof. Luca Dan Serbanati

2020

Software Design Fundamentals

Design is a key concept for future progress in software technology

Complexity, conformity, changeability, intangibility

- No two software parts are alike
 - If they are, they are usually abstracted away into one
- **Complexity** grows non-linearly with size
 - Except perhaps “toy” programs, it is impossible to enumerate all states of a program
- **Conformity**. Software is required to conform to its:
 - Operating environment
 - Hardware

(Often the last is left because it is perceived as most conformable)
- **Changeability**. Change originates with :
 - New applications, users, machines, standards, laws
 - Hardware problems

(Software is viewed as infinitely malleable)
- **Intangibility**. Software is not embedded in space
 - Often no constraining physical laws
- No obvious representation.

What is Design?

- *Design = Management of Constraints*
 - *Negotiable constraints*
 - *Non-negotiable constraints*
- The duty of the designer would be identify those constraints and then try to find the delicate balance between them by satisfying all the non negotiable constrains and optimize the negotiable ones.
- *“A pile of rocks ceases to be a rock when somebody contemplates it with the idea of a cathedral in mind.”*
- *“A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away.”*

Design Techniques

- Basic conceptual tools
 - Separation of concerns
 - Abstraction:
 - (1) look at details, and abstract “up” to concepts,
 - (2) choose concepts, then add detailed substructure, and move “down”
 - Modularity
- Object-oriented design is a widely adapted strategy
 - Objects:
 - Main abstraction entity in Object-Oriented Design (OOD)
 - Encapsulations of state as attributes with functions for accessing and manipulating that state
 - Pros:
 - UML modeling notation
 - Design patterns
 - Cons:
 - Provides only
 - One level of encapsulation (the object)
 - One notion of interface
 - One type of explicit connector (procedure call)
 - Even message passing is realized via procedure calls
 - OO programming language might dictate important design decisions.

Separation of Concerns

- Separation of concerns is the subdivision of a problem into (hopefully) independent parts.
- The difficulties arise when the issues are either actually or apparently intertwined.
- Separations of concerns frequently involves many tradeoffs
- Total independence of concepts may not be possible.
- Key example from software architecture: separation of components (computation) from connectors (interaction)

Abstraction and its relatives

- **Abstraction**: “The act or process of separating in thought, of considering a thing independently of its associations; or a substance independently of its attributes; or an attribute or quality independently of the substance to which it belongs.”
- **Reification**: “The mental conversion of ... [an] abstract concept into a thing.”
- **Deduction**: “The process of drawing a conclusion from a principle already known or assumed; spec. in Logic, inference by reasoning from generals to particulars; opposed to INDUCTION.”
- **Induction**: “The process of inferring a general law or principle from the observation of particular instances (opposed to DEDUCTION, q.v.).”
- What concepts should be chosen at the outset of a design task?
- One technique:
 - Search for a “simple machine” that serves as an abstraction of a potential system that will perform the required task.
 - For instance, what kind of simple machine makes a software system embedded in a fax machine?
 - At core, it is basically just a little state machine.
 - Simple machines provide a plausible first conception of how an application might be built.
 - Every application domain has its common simple machines.

Design vs. Architecture

- Design is an activity that pervades software development. It is an activity that creates part of a system's architecture and many objects in the system.
- Typically, in the traditional Design Phase decisions concern :
 - The system's structure
 - Identification of its primary components
 - Their interconnections.
- Architecture denotes the set of **principal** design decisions about a system
 - It implies that not all design decisions are architectural
 - How one defines "principal" will depend on what the stakeholders define as the system goals.
 - That is, they do not necessarily impact a system's architecture
 - **That is more than just structure**

Traditional design phase suggests translating the requirements into algorithms, so a programmer can implement them.

Architecture-centric design

- stakeholder issues
- decision about use of Commercial-off-the-shelf (COTS) components
- overarching style and structure
- package and primary class structure
- deployment issues
- post implementation/deployment issues.

Primacy of Design

- Software engineers collect requirements, code, test, integrate, configure, etc.
- An architecture-centric approach to software engineering places an emphasis on design.
 - Design pervades the engineering activity from the very beginning
- But how do we approach the task of architectural design?

Software Architect

- A distinctive role and character in a project
- Very broad training
- Amasses and leverages extensive experience
- A keen sense of aesthetics
- Deep understanding of the domain:
 - Properties of structures, materials, and environments
 - Customers' needs .
- Even first-rate programming skills are insufficient for the creation of complex software applications
 - But are they even necessary?

Levels of Design

Enterprise

Application

Macro Level

Micro Level

Objects

Enterprise-wide Architecture

Architecture styles

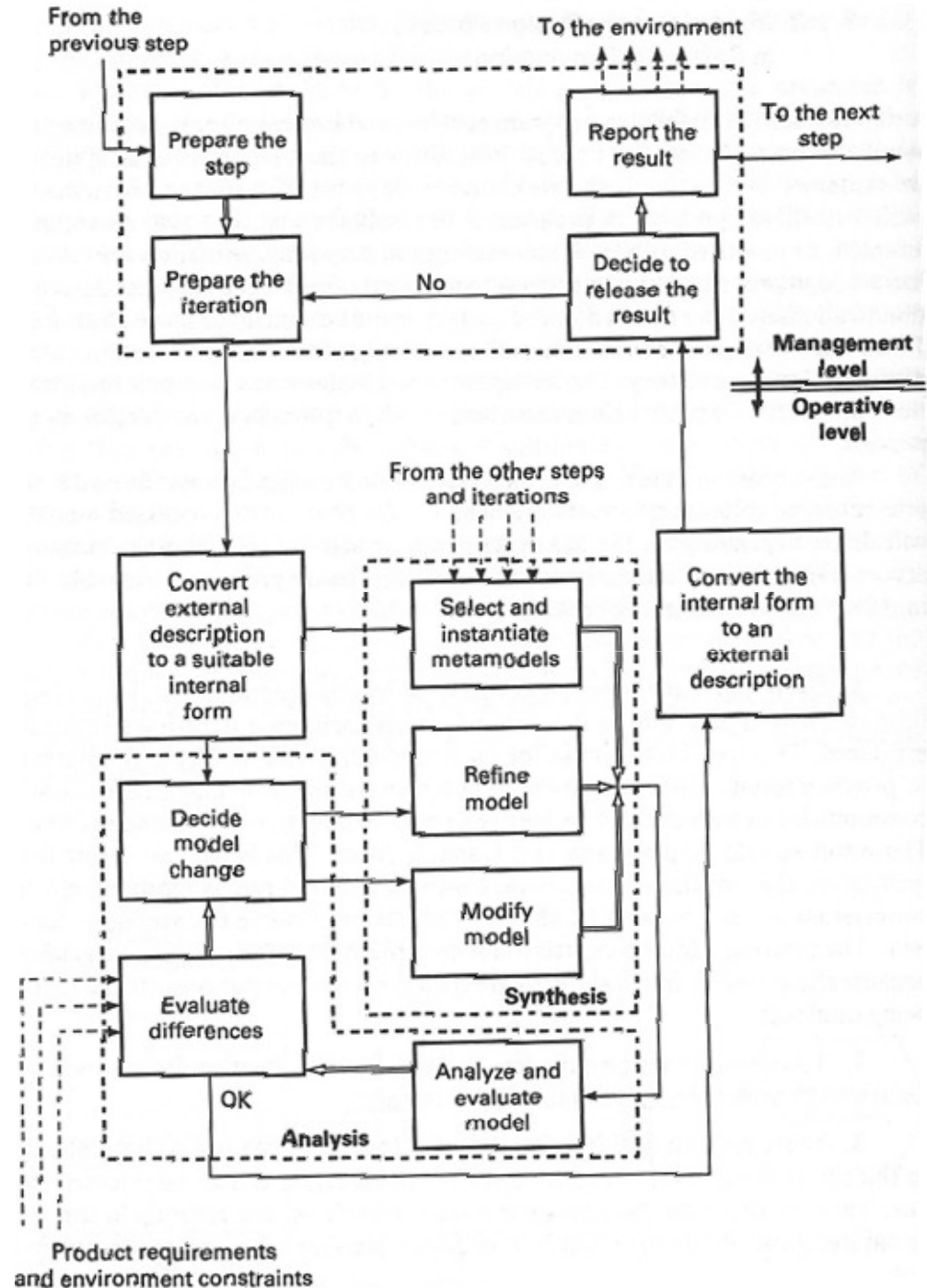
Frameworks

Design-Patterns

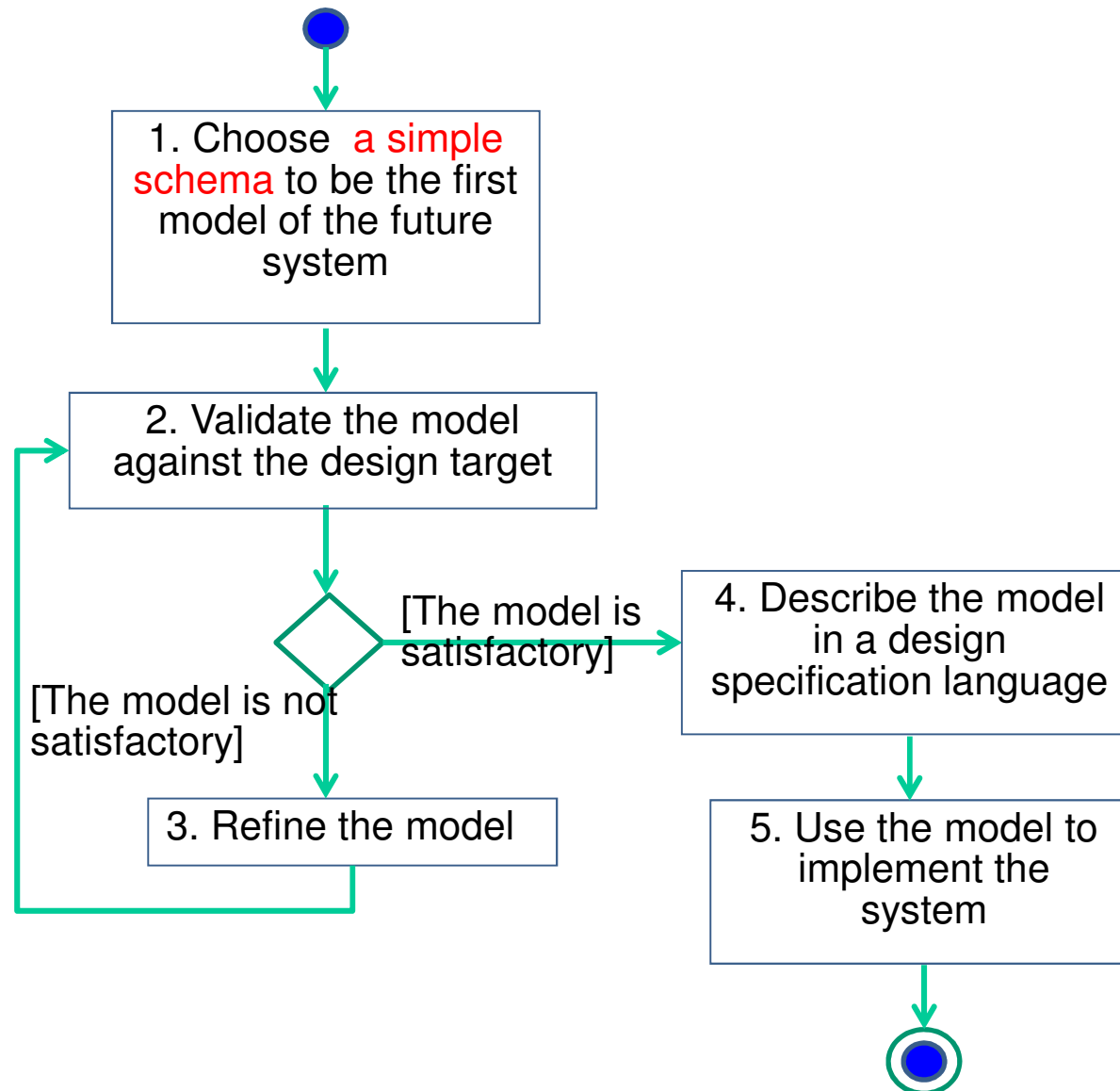
OO Programming

The Design Process

1. Each step in the program design is iteratively accomplished.
2. To choose or refine a model, we may need information from previous iterations or even other steps in the design process.
3. There are two decision points in the design schema for each iteration.



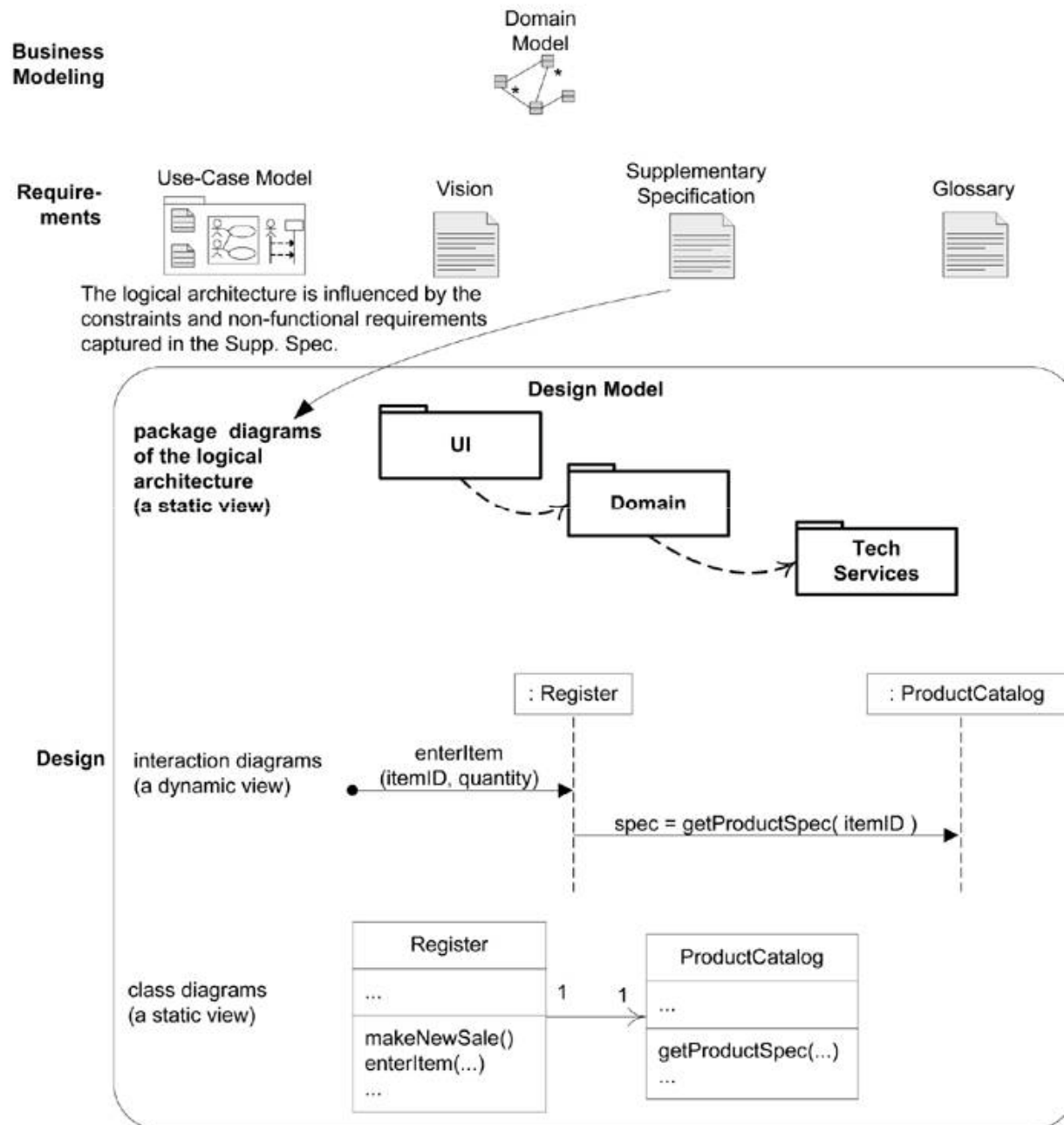
Model-Driven Design Process



Modeling Software Design

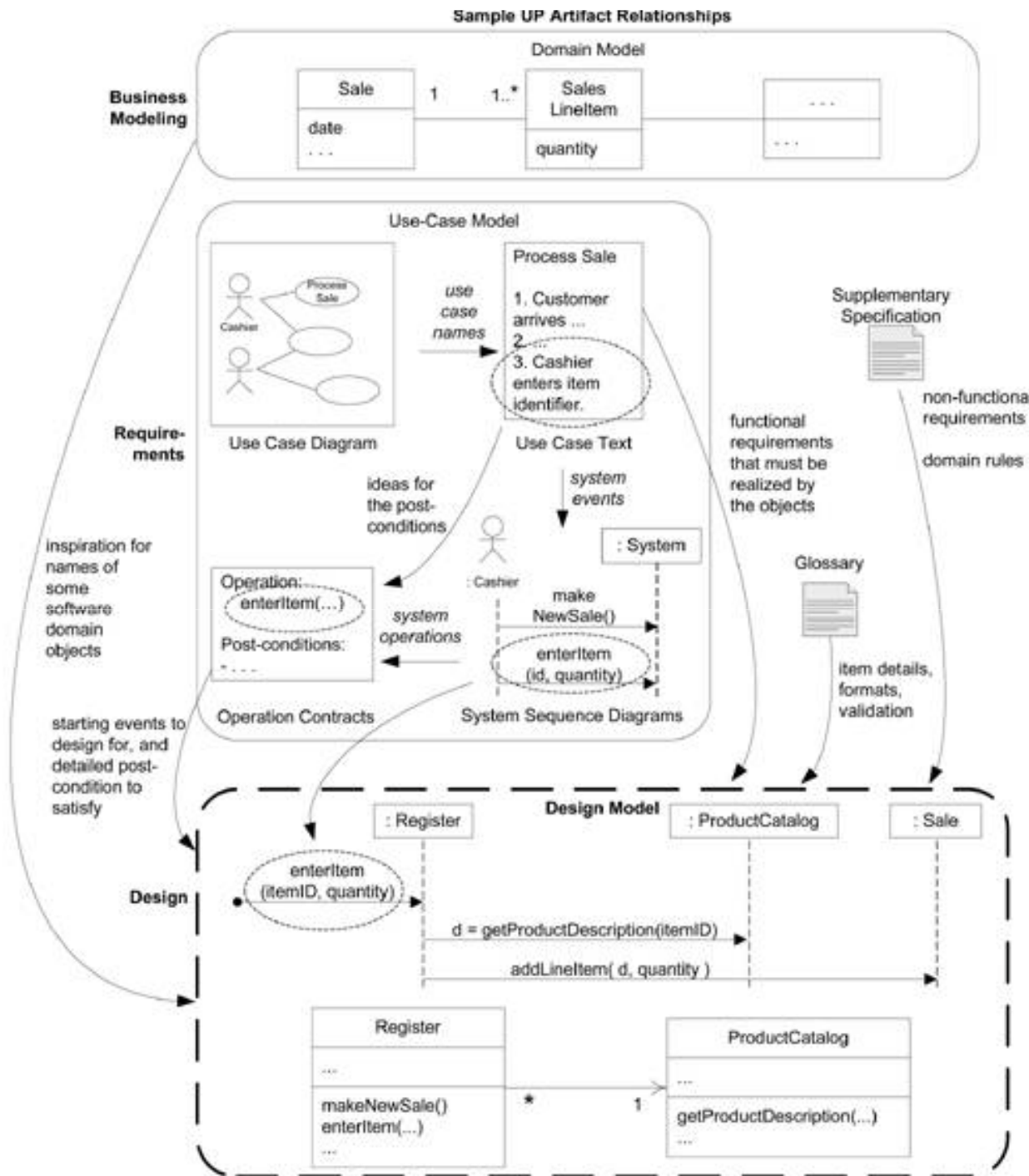
- From early concept verbalization to a detailed description of the product, ready for manufacturing, from the first product version through more and more improved versions, adapted to a given application, *the design is a continuous process* that gradually changes the emphasis from the concept, through the architecture, to the detailed description of a new product or a change in an existing one.
- The design process includes the following characteristics:
 1. A service: It is a subtask for another, more general process.
 2. Goal-oriented.
 3. Constructive: Over some existing assumptions it builds new assumptions strongly related to the previous ones.
 4. Inherently iterative: the designer repeatedly goes back to refine and improve the future product description until it satisfies the requirements.
 5. Creative: It is the result of a mental synthesis process.
 6. Closely controlled by successive verifications of the results against the requirements.
 7. Continuously refinement of the model.
 8. Multi-view on the product.
 9. Paradigmatic: It uses some existing solution patterns.
 10. Empirical: Originating in or based on observation or experience .
 11. Multi-solution.
 12. Approximate: The initial requirements are obviously fuzzy and incomplete.

Sample UP Artifact Relationships



UP: Design Models

UP: Deriving the Design Model

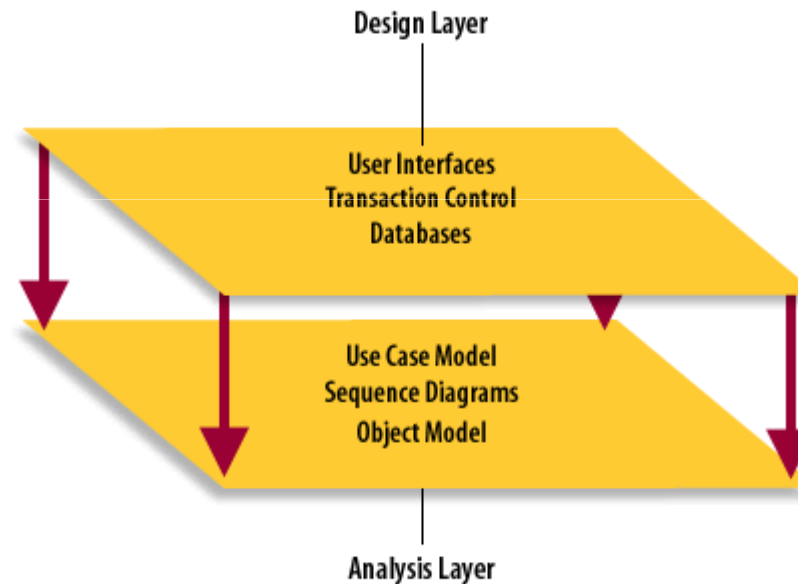


From Analysis to Design

- By the end of **analysis**, we had created and tested complete models of the problem domain. We defined all of the goals and resources that the future system has to support. These work products represent our target during analysis. The functionality (*use case model*), the resources (*object model*), and the interaction of the resources *to support the functionality* (*sequence and/or collaboration diagrams*) would exist whether or not we ever provided automation.
Everything we defined in analysis must remain intact as we move into design. In fact, the analysis level object model will be the basis for our database design.
- **Design** is about planning how to accomplish the goals defined in the analysis work products. The planning process identifies the desired solution, not the completed solution. Design addresses functionality, as well as performance, flexibility, and maintainability.
- Our design retains the picture of the desired outcome, while the **implementation** must conform to the limitations of the current technology and environment. Technologies and environments change rapidly, presenting new opportunities to improve the implementation. The design provides a framework against which to measure these new opportunities and plan for their introduction into the implementation.

Software Design (I)

- Design adds a “layer” of functionality on top of the analysis models. This layer is the software that facilitates the use of the problem domain resources using interfaces, databases, transaction control, and communication that conforms to the use case model. This layer of technology will likely change often, but the underlying problem domain will remain relatively stable.

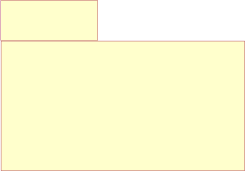
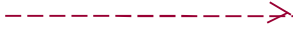
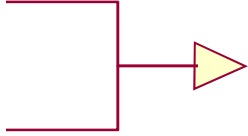
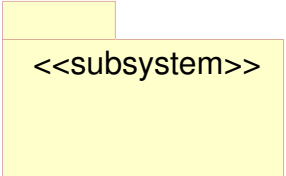
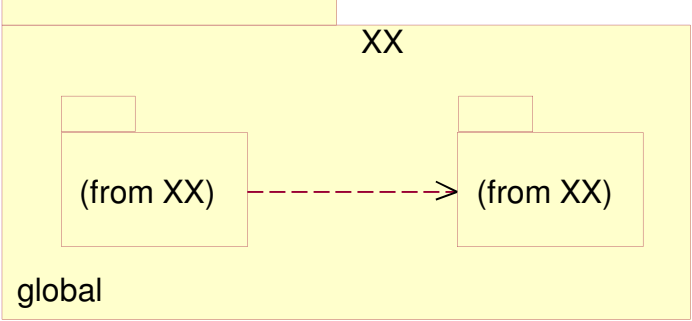


Software Design (II)

- Design is divided into two steps:
 - architectural design and
 - object design.
- Object design requires a context. Different architectures dictate different low-level designs. For example, the differences between local and distributed applications are significant. The challenges of latency, memory access, partial failures, and concurrency, require significantly different designs for local solutions than for distributed solutions. Consequently, architectural choices provide the context for low-level design.
- An architecture dictates where each piece of software will reside, how responsibilities are divided among the software components, and how the pieces of the architecture communicate.
- The architecture divides the solution according to functionality and technology. Not every function requires the same technological distribution of responsibilities. The job of the architect is to map functional requirements and distribute their responsibilities to the technologies best suited to support goals of the design, for example, performance, maintenance cost, ease of use, and the like.

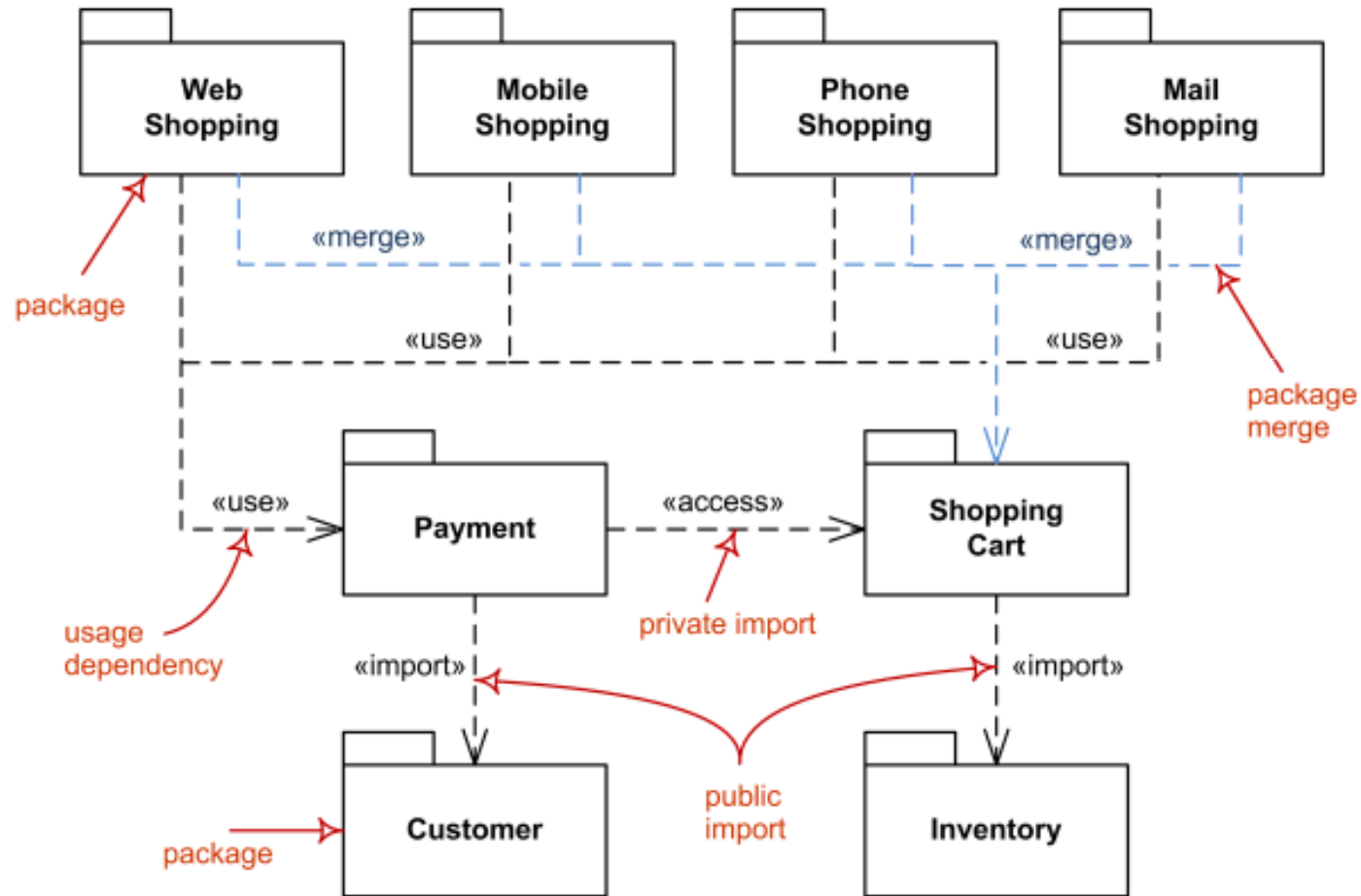
Package Diagram

- UML package diagrams are often used to illustrate the logical architecture of a system, the layers, subsystems, Java packages, etc.
- A UML package diagram provides a way to group elements: classes, other packages, use cases, anything. Nesting packages is very common.

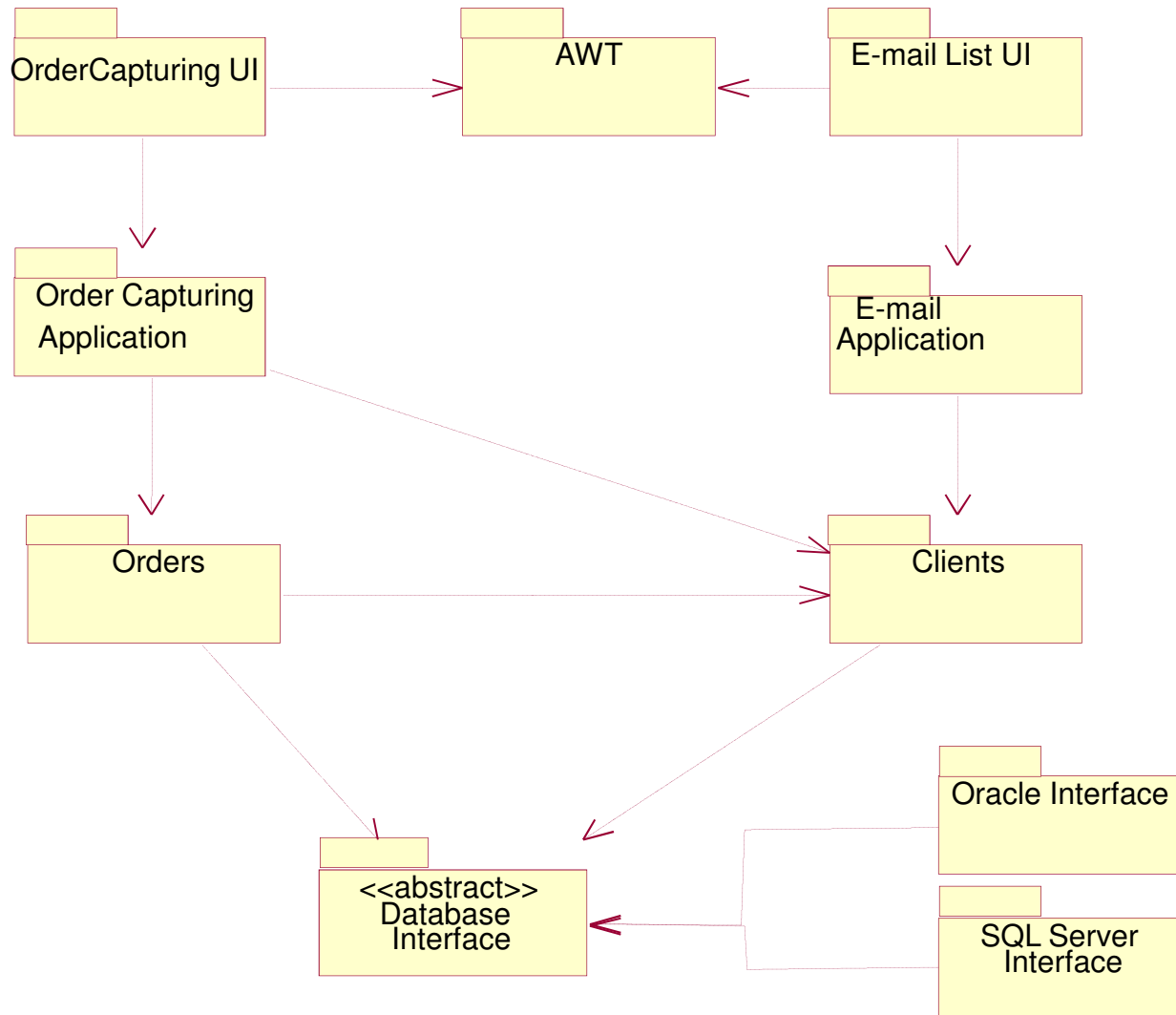
Package	Dependency	Generalization
		
Stereotype	"Contains" and "uses" relationships	
		

- UML standard stereotypes for packages: **façade**, **framework**, **stub**, **subsystem**, **system**.
- **merge**, **use**, **import**, **access** are stereotypes for dependencies.
- A UML package represents a *namespace*.

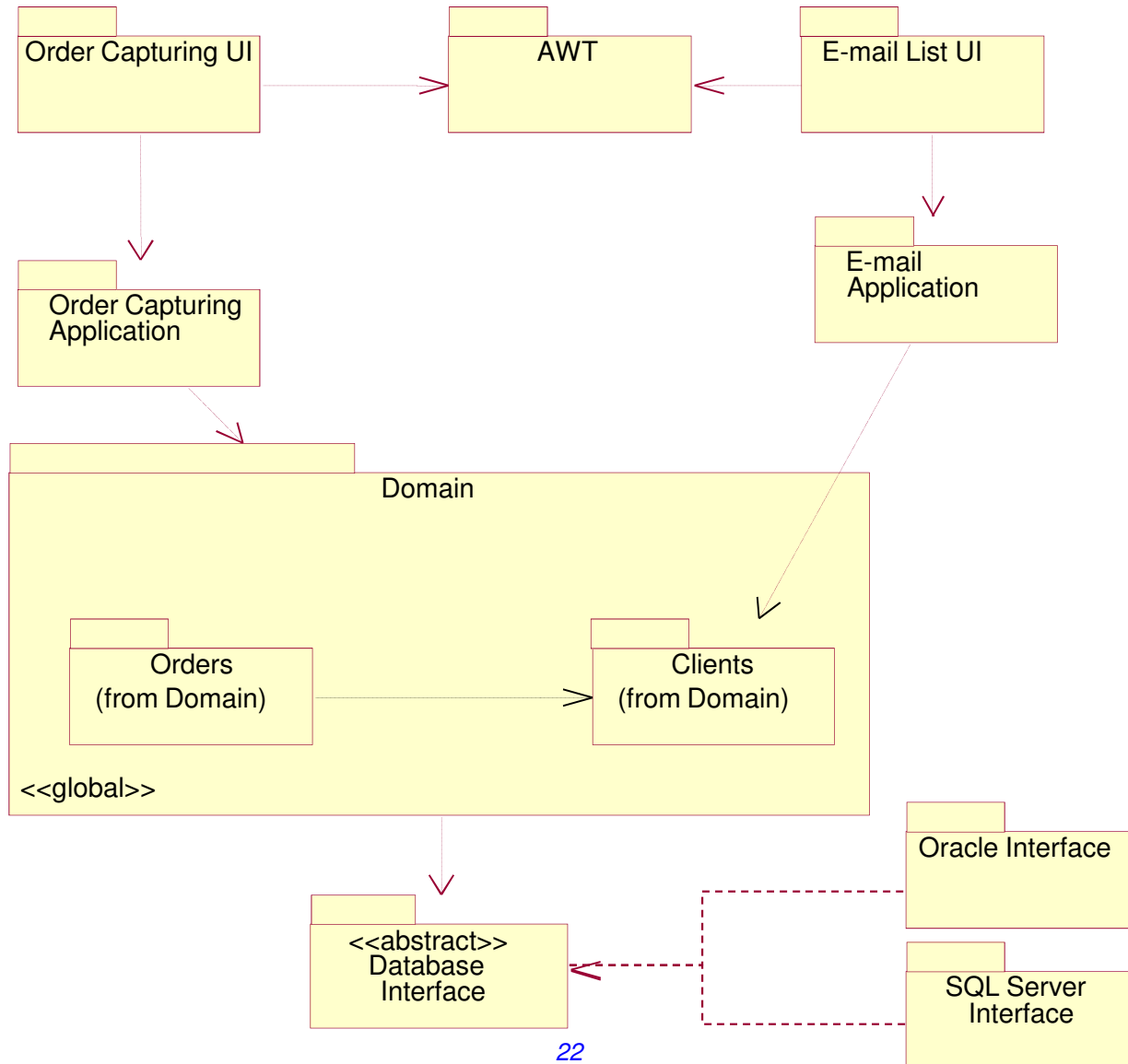
Stereotypes for Dependencies



Package Diagram - Exemple

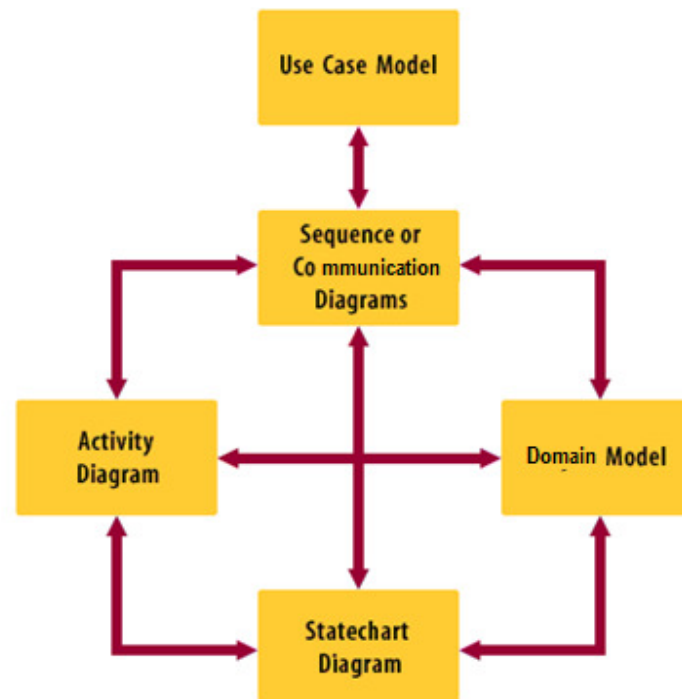


Nested Package Diagram

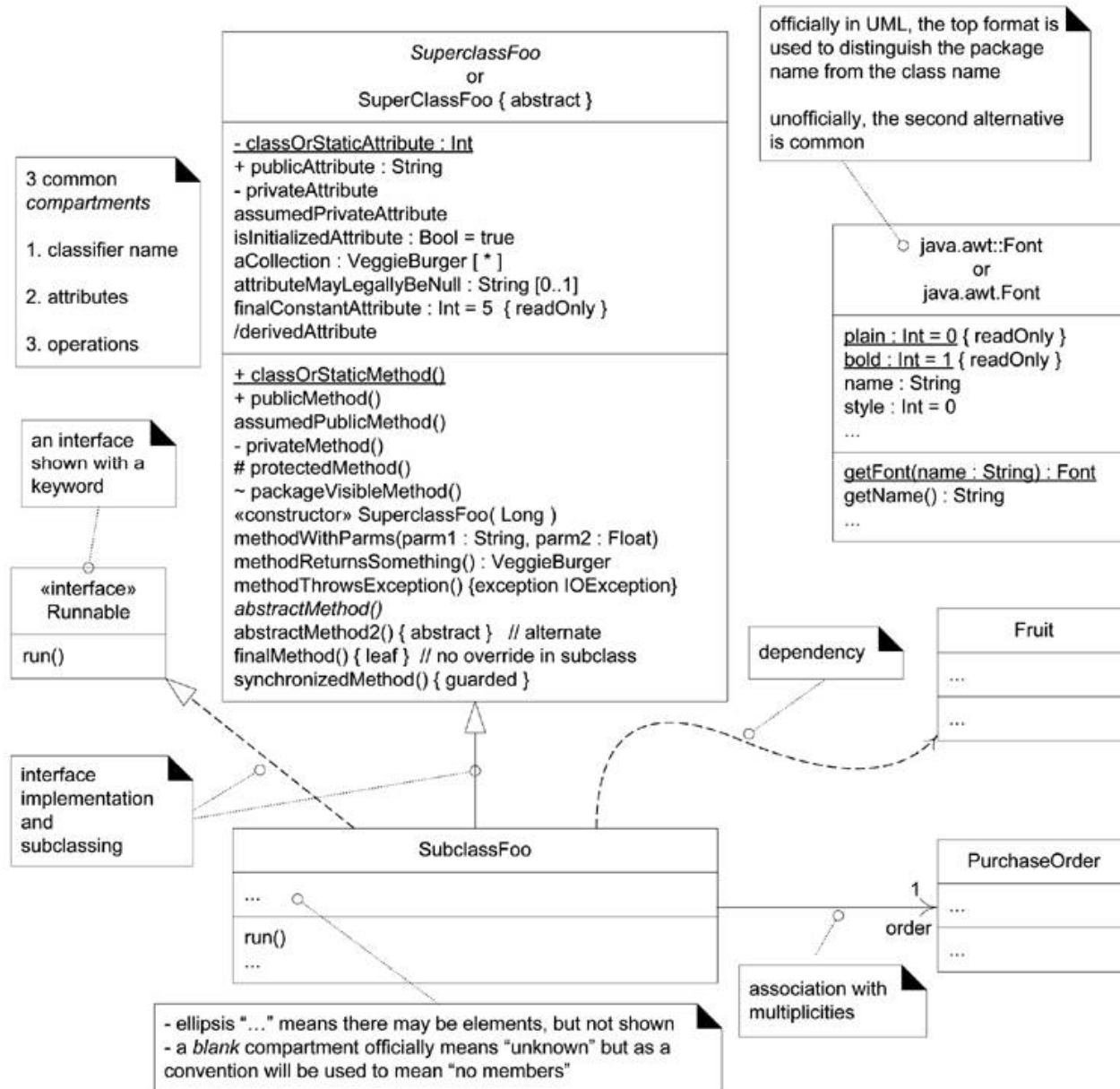


Object Design

- The context for object level design is the architecture in which the design will reside. During object design each partition presents a different type of design challenge. For example, the user interface partition addresses a very different set of problems from the data access partition. A transaction server partition is very different from a client application partition.
- Object design will use the *statechart diagram* in addition to all of the tools that you have used in analysis. Together, these tools provide working models of every aspect of the software design.



Design Class Diagram



Pieces of Advice for Software Architects

1. In software projects you can count on one thing that is constant: CHANGE.
Solution:
 - Deal with it.
 - Make CHANGE part of your design.
 - Identify what vary and separate from the rest.
2. Encapsulate what varies.
3. Program to an interface not to an implementation.
4. Favor Composition over Inheritance.
5. Classes should be open for extension but closed for modification.
6. Don't call us, we'll call you.
7. The single responsibility principle: A class should have only one reason to change.
8. Dependency Inversion Principle: Depend upon abstractions. Do not depend upon concrete classes.
9. Separate the construction of a complex object from its representation so that the same construction process can create different representations.
10. Strive for loosely coupled designs between objects that interact.
11. Principle of Least Knowledge: Talk only to your immediate friends.

END OF FIRST PART