LA-UR- 96-3020

Title:	Notes on Object-Orientation
Author(s):	Brian W. Bush
Submitted to:	World Wide Web

MASTER

Los Alamos
NATIONAL LABORATORY

Form No. 836 R5
ST 2629 10/91

DISCLAIMER

Notes on Object-Orientation

B. W. Bush
Los Alamos National Laboratory
7 May 1996

I. Overview of concepts

A. *Definitions*

- object-oriented
  - "Something is **object-oriented** if it can be extended by composition of existing parts or by refinement of behaviors. Changes in the original parts propagate, so that compositions and refinements that reuse these parts change appropriately." [GR 95]
- object
  - An **object** has state, behavior, and identity. [CS 94]
  - "An **object** is characterized by a number of operations and a state which remembers the effect of these operations." [JCJ 92]
- class
  - "A **class** represents a template for several objects and describes how these objects are structured internally. Objects of the same class have the same definition both for their operations and for their information structures." [JCJ 92]
- instance
  - "An **instance** is an object created from a class. The class describes the (behavior and information) structure of the instance, while the current state of the instance is defined by the operations performed on the instance." [JCJ 92]

B. *Key features*

- abstraction
  - "An **abstraction** denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer." [Bo 94]
  - allows building models which map to the real world
- encapsulation
  - "**Encapsulation** is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation." [Bo 94]
  - hides implementation details
- inheritance
  - "If class B **inherits** class A, then both the operations and the information structure described in class A will become part of class B." [JCJ 92]
  - enables and organizes code reuse
- polymorphism
  - "**Polymorphism** means that the sender of a stimulus does not need to know the receiving instance's class. The receiving instance can belong to an arbitrary class." [JCJ 92]
  - reduces software maintenance and increases extensibility.

C. *Issues addressed*

- scheduling: meeting delivery dates
- complexity: modeling complex applications
- size: managing interdependencies in large systems
- compatibility: making different chunks of code inter-operate

D. *Benefits*

- reuse of code
- reduced code size
- increased productivity
- lower defect rate

II. Software development

A. *Process*

1. Overview

- "The five habits of a successful object-oriented project include:
  - "A ruthless focus on the development of a system that provides a well-understood collection of essential minimal characteristics.
  - "The existence of a culture that is centered on results, encourages communication, and yet is not afraid to fail.
  - "The effective use of object-oriented modeling.
  - "The existence of a strong architectural vision.
  - "The application of a well-manages iterative and incremental development life cycle." [Bo 96]
- "Why do certain object-oriented projects succeed? Most often, it is because:
  - "An object-oriented model of the problem and its solution encourages the creation of a common vocabulary between the end users of a system and its developers, thus creating a shared understanding of the problem being solved.
  - "The use of continuous integration creates opportunities to recognize risk early and make incremental corrections without destabilizing the entire development effort.
  - "An object-oriented architecture provides a clear separation of concerns among the disparate elements of a system, creating firewalls that prevent a change in one part of the system from rending the fabric of the entire architecture." [Bo 96]
- Booch's macro process
  - "Establish core requirements (conceptualization).
  - "Develop a model of the desired behavior (analysis).
  - "Create an architecture (design).
  - "Evolve the implementation (evolution).

- "Manage postdelivery evolution (maintenance)." [Bo 94]

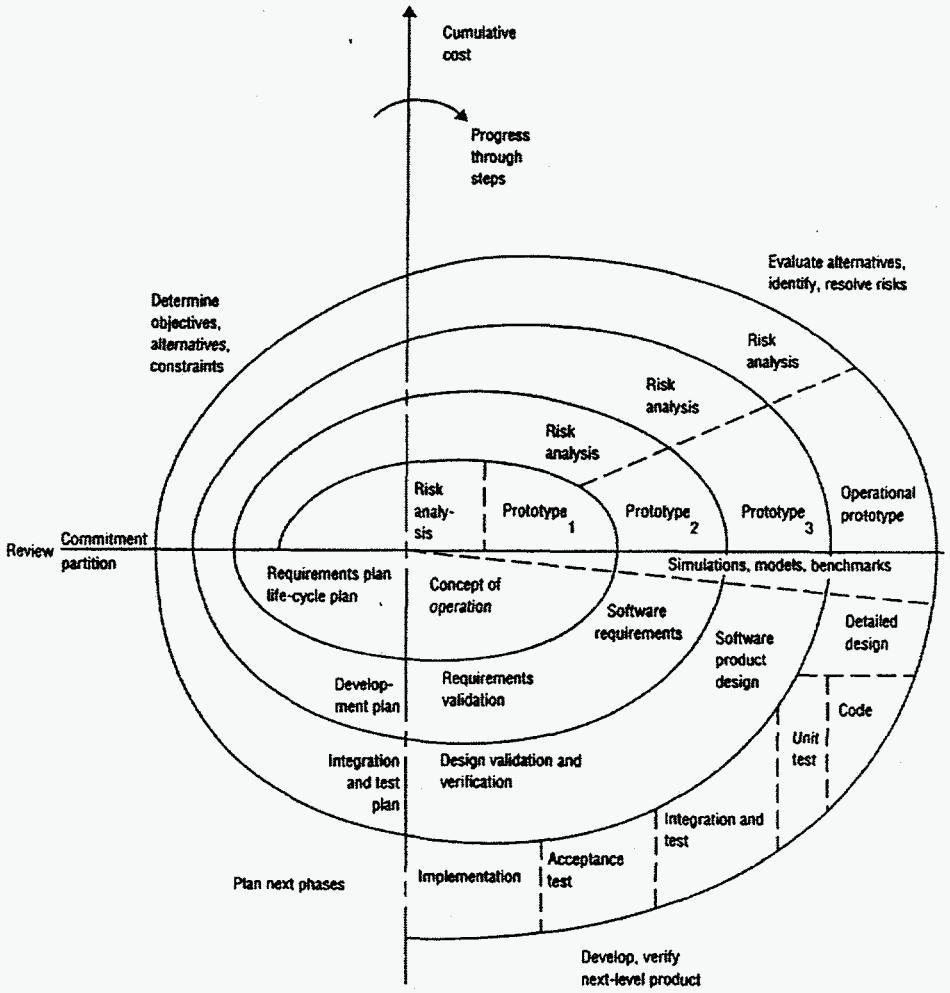
- "A common fallacy in the software industry is that there is one process model that will work for projects that use object-oriented technology. . . . But one process model does not fit all software development situations. . . . We identified and classified five types of projects labeled as:
  - "First-of-Its-Kind
  - "Variation-on-a-Theme
  - "Legacy Rewrite
  - "Creating Reusable Assets, and
  - "System Enhancement or Maintenance." [GR 95]
- SEI Process Capability Maturity Model (CMM)
  - "*Level 1: Initial*
    "Characteristics: chaotic; unpredictable cost, schedule, and quality performance.
  - "*Level 2: Repeatable*
    "Characteristics: Intuitive; cost and quality highly variable, reasonable control of schedules, informal and *ad hoc* methods and procedures. The key elements to achieve level 2 maturity follow:
    + "Requirements management
    + "Software project planning and oversight
    + "Software subcontract management
    + "Software quality assurance
    + "Software configuration management.
  - "*Level 3: Defined*
    "Characteristics: qualitative; reliable costs and schedules, improving but unpredictable quality performance. The key elements to achieve this level of maturity follow:
    + "Organizational process improvement
    + "Organizational process definition
    + "Training program
    + "Integrated software management
    + "Software product engineering
    + "Intergroup coordination
    + "Peer reviews.
  - "*Level 4: Managed*
    "Characteristics: quantitative; reasonable statistical control over product quality. The key elements to achieve this level of maturity follow:
    + "Process measurement and analysis
    + "Quality management.
  - "*Level 5: Optimizing*
    "Characteristics: quantitative basis for continued capital investment in process

automation and improvement. The key elements to achieve this highest level of maturity follow:

+ "Defect prevention
+ "Technology innovation
+ "Process change management." [Ka 95]

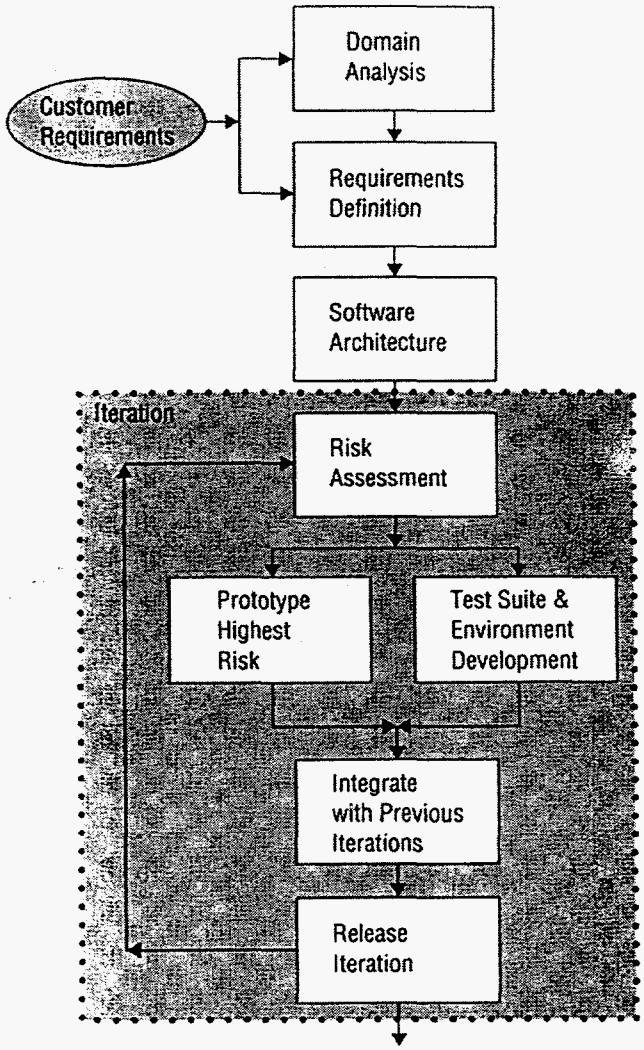
2. Spiral model

- "Maxims underlying the Spiral Model state:
  - "An activity starts with an understanding of the objectives and risks involved.
  - "Based on evaluating alternative solutions, use the tool(s) that best reduce(s) the risks.
  - "All related personnel should be involved in a review that terminates each activity, and plans for and commits to the next activities.
  - "Development can proceed in increments at each stage." [GR 95]
- spiral model diagram [Ka 95]

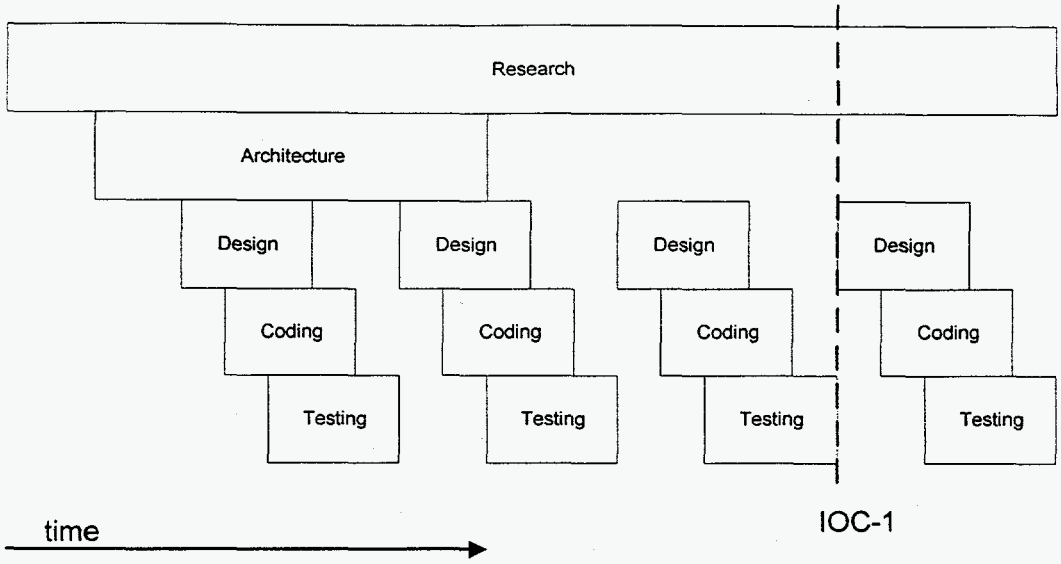


3. Iterative and incremental development

- "The act of reviewing a prior result for possible change is referred to as **iteration**. What people seem to mean is: We do not have to get it all done the first time around—we can come back later. Just get done what we know, and we'll add more later, or we'll fix the coding of it later, or we'll replace this function later, and so on. . . . The real intent of **iterative development** is to allow for the controlled reworking of part of a system to remove mistakes or to make improvements based on user feedback." [GR 95]
- "**Incremental development** is a strategy for making progress in small steps to get early tangible results. . . . The incremental approach requires that a problem be partitioned into several subproblems so that each can be developed in turn. As each partition is completed, it is tested and integrated with the other completed partitions of the system." [GR 95]
- iterative process flow chart [Ka 95]



- TRANSIMS iterative process



4. Parallel development

- "[T]he basic approach is to decompose a problem systematically into independent components . . . [T]he components must be as independent of one another as possible . . .
  - Development should focus on working in parallel on independent components.
  - Systems should be designed as compositions of independently created components." [GR 95]

5. Prototyping

- "The purpose of **prototyping** is to seek the information needed to make decisions. Prototyping can help reduce the risk of making mistakes in setting requirements or in designing the system architecture. . . . A **prototype** is a preliminary, or intentionally incomplete or scaled-down, version of a system. . . . The term **rapid prototyping** refers to the process of quickly building and evaluating one or more prototypes." [GR 95]

6. Rapid development

- **Rapid development** generally involves the following:
  - short iterative/incremental cycles
  - extensive user involvement and feedback
  - evolving prototypes into products

B. Analysis

- "Object-oriented analysis is a method of analysis that examines requirements from the perspective of classes and objects found in the vocabulary of the problem domain." [Bo 94]

- **Use cases** are a tool for the **definition of requirements** and the **analysis of the problem domain**. Requirements are generated both from the customer's definition of the problem and results of analyzing the abstractions in the problem's domain.
- analysis process [Lo 93]

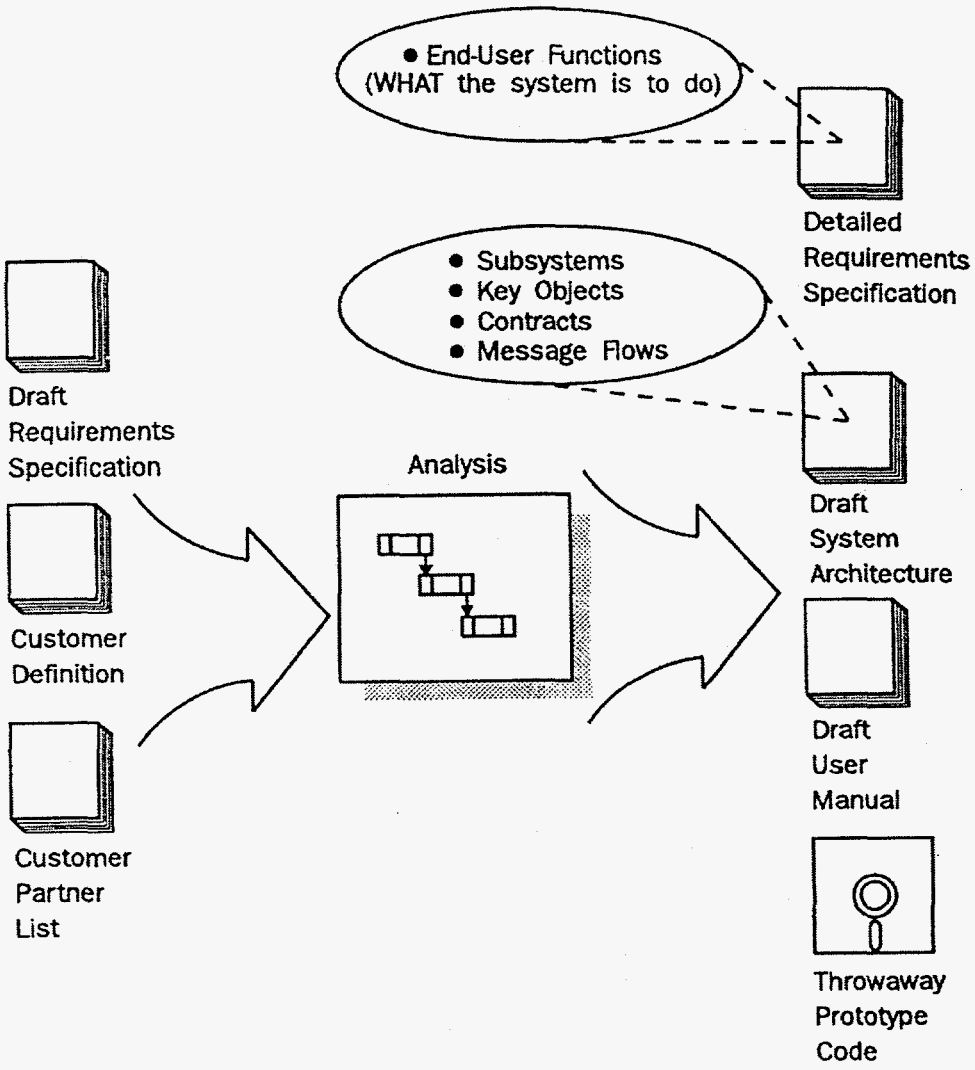


Figure 4.4 Analysis-Phase Prerequisites and Deliverables

1. Requirements definition

- function of requirements
  - "Creates a clear, precise and complete definition of the system
  - "Limits the scope of the system
  - "Defines the functionality of the system
  - "Forms a contract between the developer and the buyer of the system" [CS 94]
- "It is important to keep control of an iterative process. This control is effective by driving the effort expended and decisions of completion by documented requirements. . . . Strong control over the requirements can be maintained by regularly:

- "*Asking customers:*
  "Is this necessary to satisfy our documented requirements? Does our competition have this feature? If not, does it provide real value to our customers? Have we asked them?
- "*Showing customers:*
  "Demonstrations of the latest product under development, getting feedback such as features ranked by importance." [Lo 95]
- Requirements document: "This is a clear, concise statement of the end users' needs for a product. It contains functions to be provided and no design." [Lo 93]

- Requirements specification contents [Lo 93]
  - general
    - purpose
    - terms
  - system summary
    - background
    - objectives
  - system and subsystem definition
    - system diagrams
      - major system software functions
      - major system software functional relationships
    - interface definition
    - assumptions and constraints
  - detailed characteristics and requirements
    - specific performance requirements
    - computer program functional requirements
    - failure contingencies
    - design requirements
    - human performance requirements
  - environment
  - quality assurance

2. Domain analysis

- "**Domain analysis** attempts to understand the basic abstractions in a discipline. The goal of domain analysis is to determine a general domain model from which it is possible to develop multiple applications. . . . The outcome of a domain analysis is the identification of reuse opportunities across applications in a domain." [GR 95]
- information sources
  - domain experts
  - the customer
  - other products

- in-house prototypes

3. Use cases

- "The **use case model** uses **actors** and **use cases**. These concepts are simply an aid to defining what exists outside the system (actors) and what should be performed by the system (use cases)." [JCJ 92]
- "Sequences of actions are captured as **use cases** (a specific kind of scenario), and the objects that reside outside the system boundaries are called **actors**. Use cases identify three types of analysis objects: interface, entity, and control.
  - "**Interface objects** translate an actor's actions into events, and the system events into results an actor can understand.
  - "**Entity objects** model system information.
  - "And **control objects** are a catchall for behaviors not easily allocated to the other two types, essentially providing the glue that unites the other system objects." [GR 95]
- a use case from TRANSIMS

Producing a Fundamental Diagram

Actor: Analyst

Basic Course: A fundamental diagram presents traffic flow vs. density over a path—one that has no entry or exit nodes other that the beginning and end nodes.

The analyst identifies a set of simulations for which to produce fundamental diagrams. In addition the analyst specifies (by graphical interaction with a map of the simulated road network) the path on which to compute the diagram and the time step to use. Then the toolbox computes and plots vehicle flow vs. density at specified times.

Alternatives: The analyst can request printed output of the images currently seen. Alternate methods for selecting a road segment are possible.

C. *Design*

- design procedure
  - "*Model the essential system:* The essential system describes those aspects of the system required to make it achieve its purpose, regardless of the target hardware and software environment. It is composed of essential activities and essential data. This step has five substeps:
    + "Creating the user view.
    + "Modeling essential activities.
    + "Defining solution data.
    + "Refining the essential model.
    + Constructing a detailed analysis.
  - "*Derive candidate essential classes:* This step uses a technique known as 'carving' to identify candidate essential classes and methods from the essential

model of the system. A complete set of data-flow diagrams, along with supporting process specifications and data dictionary entries, is the basis for class and method selection. Candidate classes and methods are found in external entities, data stores, input flows, and process specifications.

- "*Constrain the essential model:* The essential model is modified to work within the constraints of the target implementation environment. Essential activities and essential data are allocated to the various processors and containers (data repositories). Additional activities are added to the system as needed, based on the target implementation environment limitations. The essential model, when augmented with the additional activities needed to support the target environment is referred to as the incarnation model.
- "*Derive additional classes:* Additional candidate classes and methods specific to the implementation environment are selected based on the additional activities added while constraining the essential model. These classes supply interfaces to the essential classes at a consistent level.
- "*Synthesize classes:* The candidate-essential classes and the candidate-additional classes are refined and organized into a class hierarchy. Common attributes and operations are extracted to produce superclasses and subclasses. Final classes are selected to maximize reuse through inheritance and importation.
- "*Define interfaces:* The interfaces, object-type declarations, or class definitions are written based on the documented synthesized classes.
- "*Complete design:* The design of the implementation module is completed. The implementation module comprises methods where each provides a single cohesive function. Logic, system interaction, and method invocations to other classes are used to accomplish the complete design for each method in a class. Referential integrity constraints that are specified in the essential model (using the data model diagrams and data dictionary) are now reflected in the class design." [Ka 95]

- design process [Lo 93]

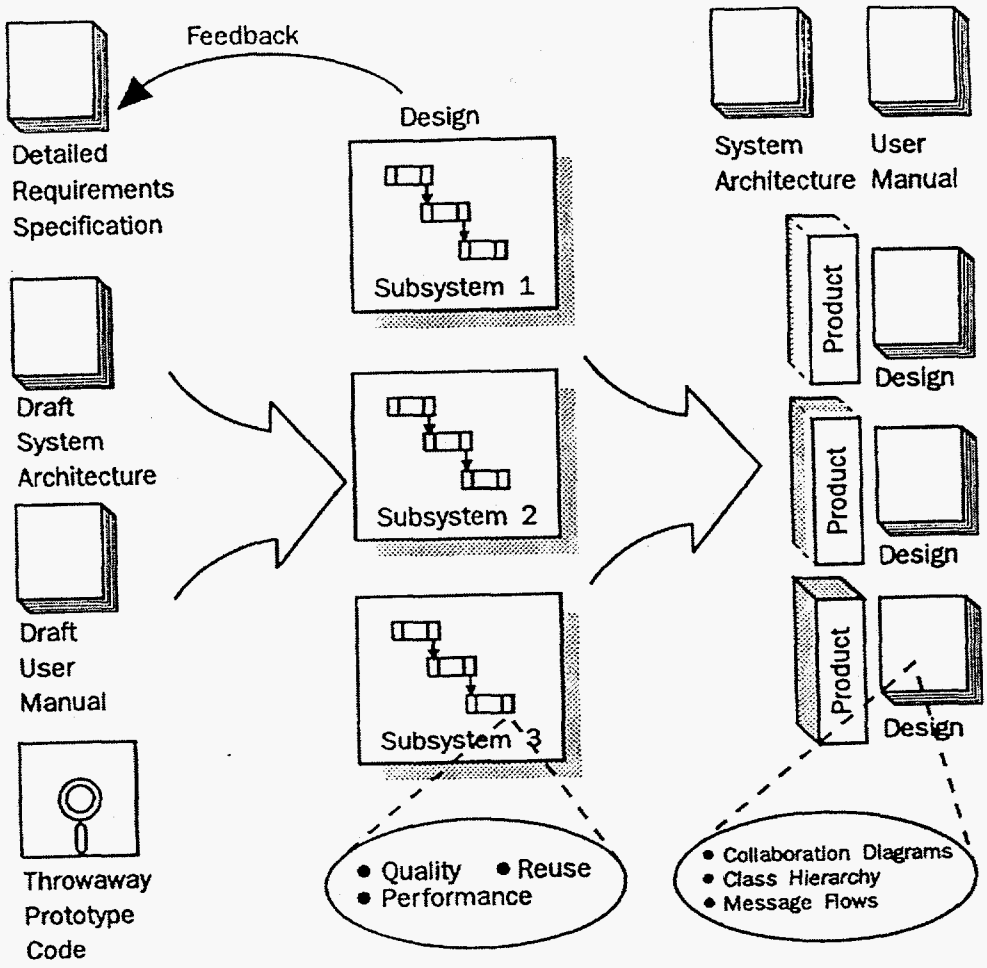


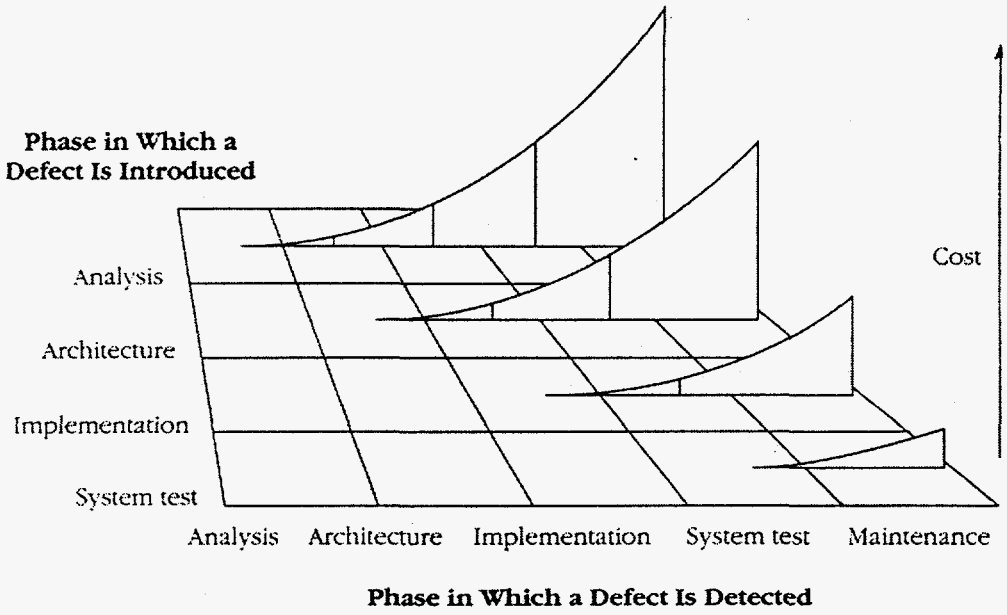
Figure 4.5 Design-Phase Prerequisites and Deliverables

1. Architectural design

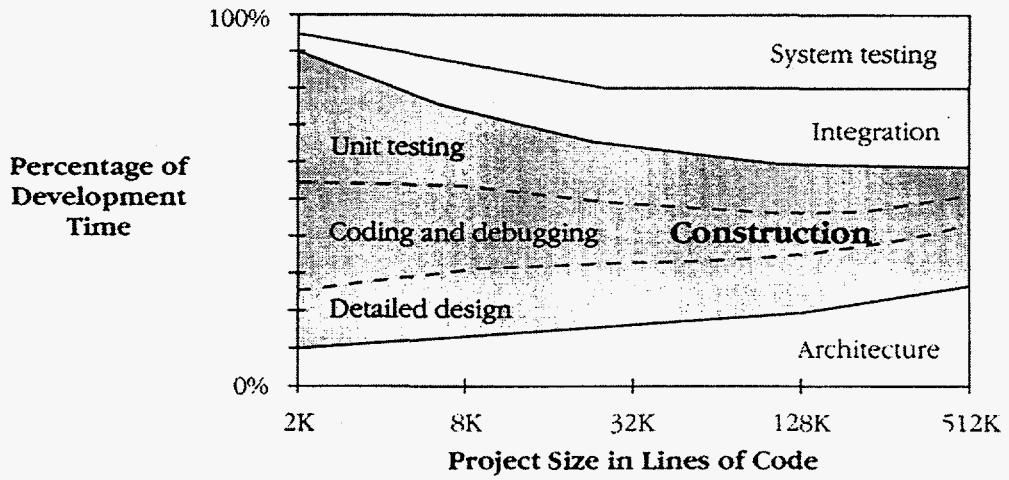
- "Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamics models of the system under design." [Bo 94]
- Architectural considerations become more important the larger a software development project becomes
  - "On a small project, construction is the most prominent activity by far, taking as much as 80 percent of the total development time.
  - "On a medium-size project, construction is still the dominant activity but its share of effort falls to about 50 percent.
  - "On very large projects, architecture, integration, and system testing each take up about as much time as construction." [Mc 93]
- It also pays to do things right the first time.
  - "Data from TRW shows that a change in the early stages of a project, in requirements or architecture, costs 50 to 200 times less than the same change later, in construction or maintenance.

- "Researchers at IBM found that purging an error by the beginning of design, code, or unit test allows rework to be done 10 to 100 times less expensively than when it's done in the last part of the process." [Mc 93]

- Defects introduced in the architecture are expensive to correct: [Mc 93]



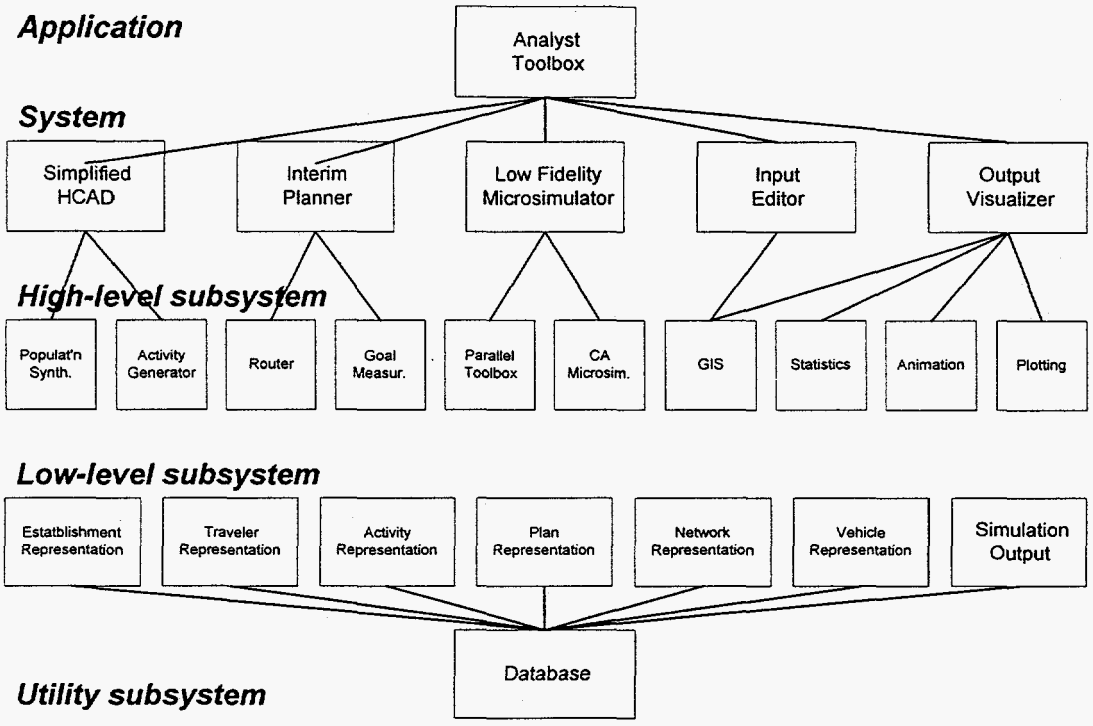
- Architecture consumes larger fractions of the development time for larger projects. [Mc 93]



- "A well-structured object-oriented architecture consists of:
  - A set of classes, typically organized into multiple hierarchies
  - A set of collaborations that specify how those classes cooperate to provide various system functions" [Bo 96]

a) *Layering*

- **Layering** separates the software components into a hierarchy with the application at the top, the domain in the middle, and the technology at the bottom.
- Each layer uses the layer below it, but not vice versa.
- Layering encourages the reuse of software components in different parts of the application.
- Layering provides an integrated framework for the software development.
- TRANSIMS architecture



b) *Modularity*

- "**Modularity** is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules." [Bo 94]
- Each software component/module has responsibilities and provides services.
- The actual implementation of the module is separate from its public interface.
- Modularity reduces the coupling between software components that can make maintenance, reuse, portability, and extension difficult.

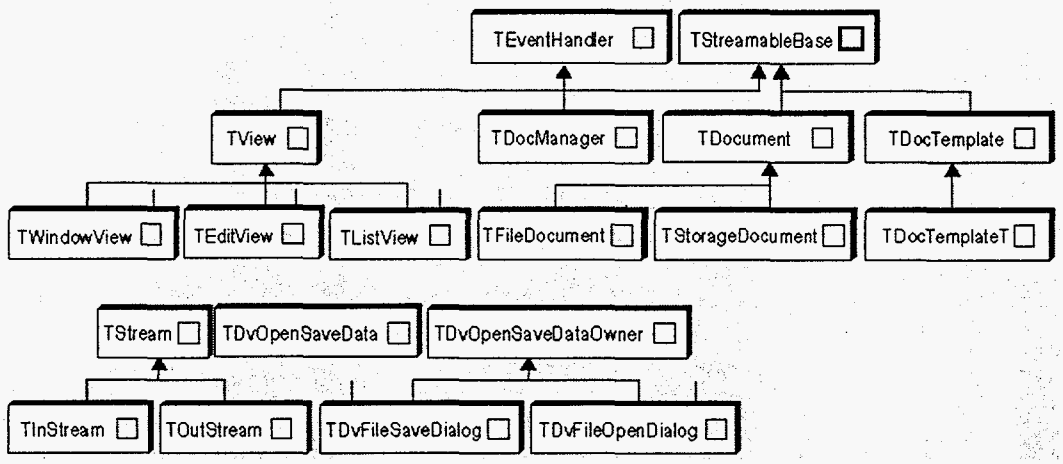
c) *Frameworks*

- "We can define an application in sufficiently general or abstract terms, so that we can change the objects that provide the expected behavior without changing the implementation of the basic application itself. This extends the usefulness of the application. Applications written in this style are referred to as application frameworks. An **application framework** is a set of objects that interact to form the basic structure and processing of applications within a given domain. The objects that

provide the general behaviors required by the application framework are called **components** of the framework." [GR 95]

- Frameworks need not be limited to abstracting application behavior: there are also **database frameworks, OLE frameworks**, etc.
- Borland's ObjectWindows Library

Doc/View Classes



2. Class design

- Booch's micro process
  - "Identify the classes and objects at a given level of abstraction.
  - "Identify the semantics of these classes and objects.
  - "Identify the relationship among these classes and objects.
  - "Specify the interface and then the implementation of these classes and objects." [Bo 94]
- **CRC cards, software diagrams,** and **design patterns** are tools for designing classes.
- Nouns appearing in functional specifications and use cases are candidates for classes.

a) CRC cards

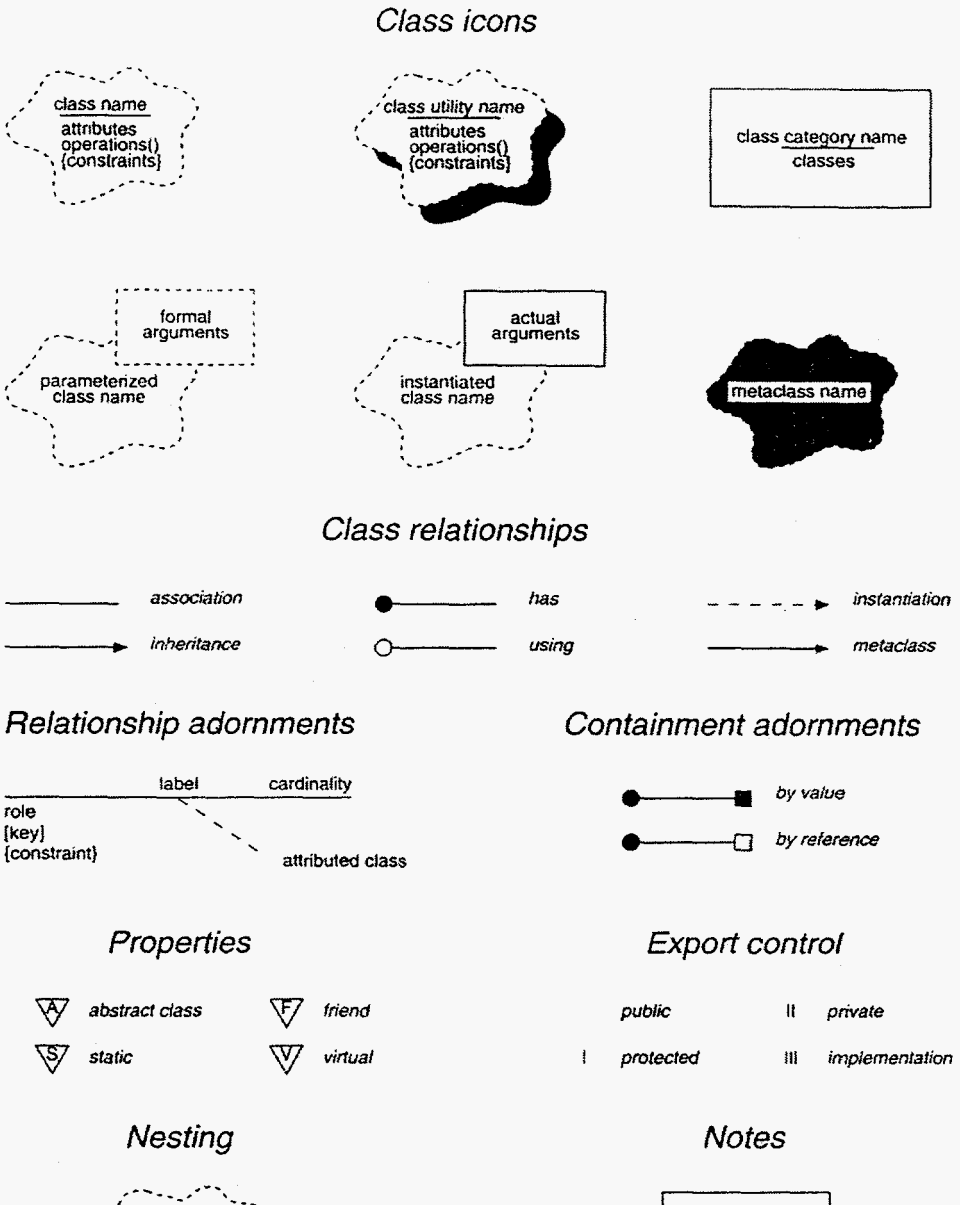
- A **CRC card** is an index card for a **class**, listing its **responsibilities** and **collaborators** (other classes).
- "The CRC card **session** is a fast-paced, creative exchange. Classes are discovered, converted into cards, and then annotated with responsibilities and collaborators through the **physical simulation** of the system. Each person is responsible for holding, moving, and annotating one or more cards as messages fly around the system in support of a particular need or activity." [Wi 95]
  - "The **brainstorming** process consists of one person standing at a board writing names of classes as they are suggested by the session participants." [Wi 95]
  - "As the number of classes being suggested decreases, the session gradually enters a **filtering** stage, wherein classes are examined more closely. During this stage, we use the requirements plus the knowledge of the people in the room to eliminate redundancies, identify missing abstractions, and recognize related classes." [Wi 95]

- "The classes, each of which becomes a CRC card, can be **assigned** as they are suggested (usually to the person who suggests them) or later after most classes have been discovered." [Wi 95]
- "Once a reasonable set of classes are suggested, filtered, and well understood by everyone in the room, it is time to start assigning them the **behaviors** that will combine to provide the functions of the application. Single **responsibilities**, which are derived from the requirements, or responsibilities that are 'obvious' from the name of the class can be listed even before the scenario execution is begun." [Wi 95]
- "**Attributes** may also be identified by reading the requirements. Often nouns that are not classes but characteristics of classes are best represented as attributes." [Wi 95]
- "**Scenarios** to be explored are the 'what happens when's that illustrate the expected use of the system. . . . The simulation of the scenarios should be dynamic and anthropomorphic. The people who own particular cards should hold the card in the air and 'become' the object when a scenario causes control to pass to them." [Wi 95]

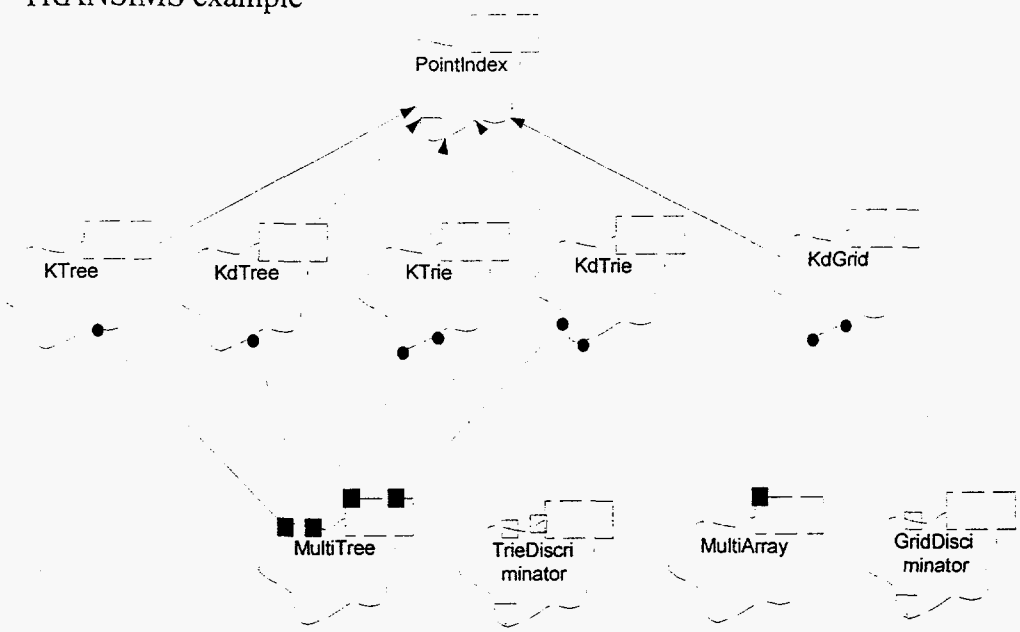
b) Software diagrams

- A **class diagram** shows "the existence of classes and their relationships in the logical view of a system." [Bo 94]

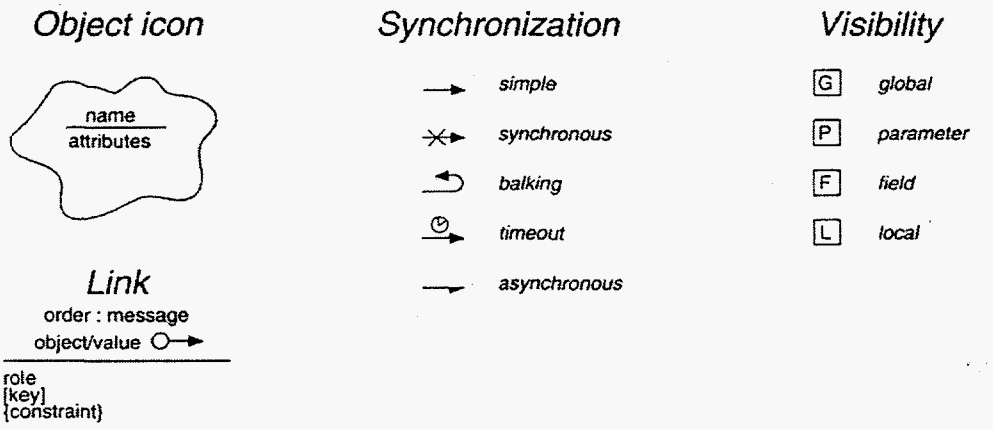
– Booch notation [Bo 94]



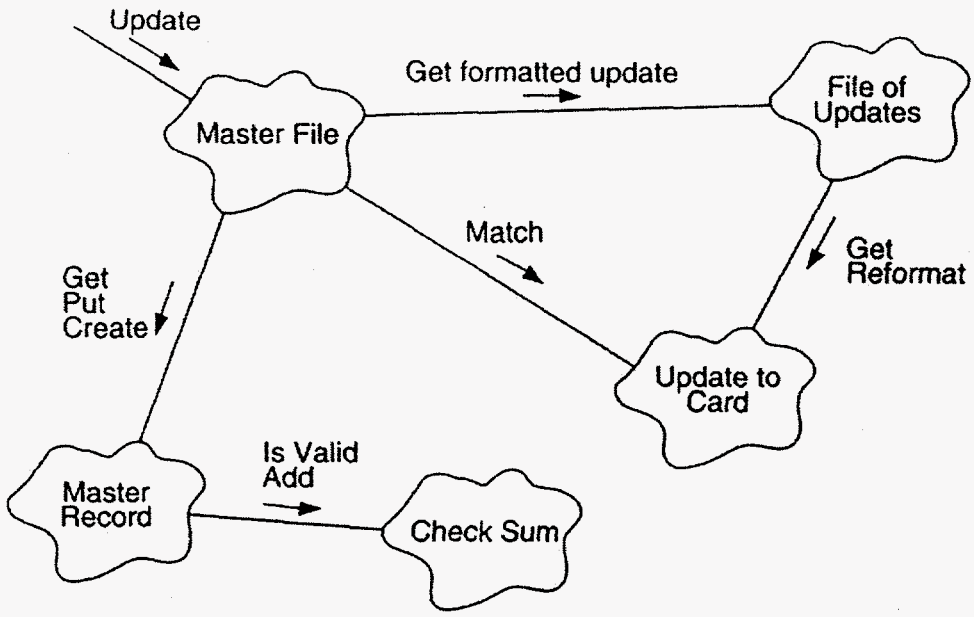
– TRANSIMS example



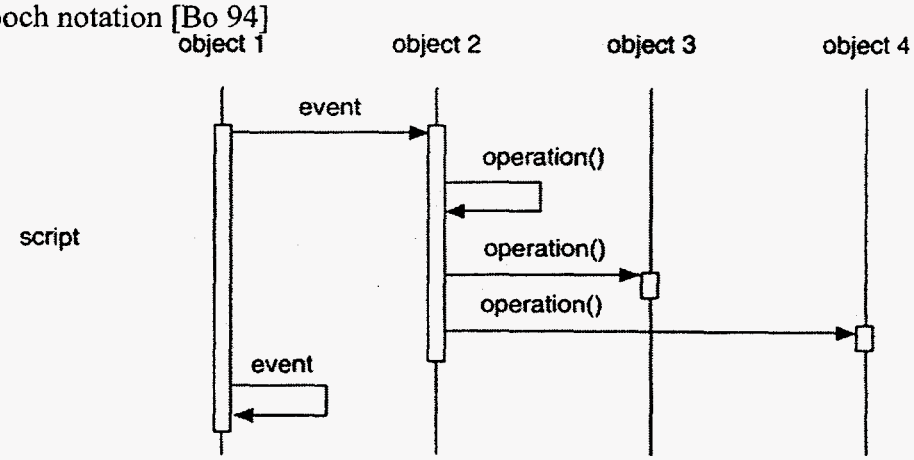
- An **object diagram** shows "the existence of objects and their relationships in the logical view of a system." [Bo 94]
  - Booch notation [Bo 94]



– example [Bo 94]



- An **interaction diagram** traces "the execution of a scenario." [Bo 94]
  – Booch notation [Bo 94]

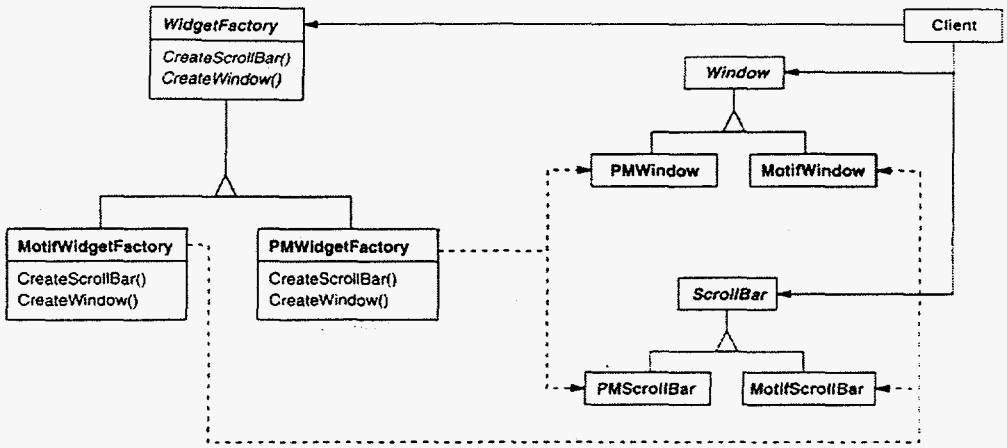


---

c) Design patterns

- **Design patterns** are "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context." [GHJ 95]
- "A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue." [GHJ 95]

- example [GHJ 95]



D. Coding

- "**Object-oriented programming** is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships." [Bo 94]

- OOP language comparison [RBP 91]

	C++ 2.0	Smalltalk 80	CLOS	Eiffel	Objective C
Integration of classes with primitive types	hybrid	pure	integrated	integrated	hybrid
Strong type checking	Y	N	N	Y	Y
Ability to restrict access to attributes:					
Control of access from clients	Y	Y	N	Y	Y
Control of access from subclasses	Y	N	N	Y	N
Standard class library	N	Y	N	Y	Y
Parameterized classes	F	—	—	Y	N
Multiple inheritance	Y	N	Y	Y	N
Scoping of class names (packages)	N	N	Y	N	N
Messaging model:					
Single target object	Y	Y	N	Y	Y
Dynamic binding on multiple args	N	N	Y	N	N
Method combination features:					
SUPER concept	N	Y	Y	Y	Y
Before & after methods	N	N	Y	N	N
Assertions and constraints	N	N	N	Y	N
Metadata at run-time	N	Y	Y	N	Y
Garbage collection	N	Y	Y	Y	N
Efficiency:					
Static binding when possible	Y	N	N	Y	Y

Key to table entries:
- Y = Yes, the feature is present.
- N = No, the feature is not present in common current implementations.
- F = Planned in a future release
- — = Not Applicable: Parameterized classes are not needed in languages with weak typing.

Figure 15.2 Comparison of commercially available object-oriented [illegible]

- OOP language comparison [GR 95]

Concept/ Mechanism	*Benefits*	*Drawbacks*	*Languages*
Object Abstraction			
Classes or templates	Capture similarity among like objects	Overhead for applications that have many one-of-a-kind objects	C++, CLOS, Eiffel, Objective C, Smalltalk
Encapsulation			
Multiple levels	Flexibility in controlling visibility	Reduces potential for reuse	C++, CLOS
Circumvention	Potential performance boost by avoiding message-passing as a way of accessing data	Violates an object's encapsulation and introduces tight coupling between objects	C++, CLOS, Objective C
Polymorphism			
Unbounded polymorphism	Flexibility in prototyping and maintenance to replace an object with another object that supports the required interface	Inhibits static type checking	CLOS, Objective C, Smalltalk
Bounded polymorphism	Provides additional information for type checking and optimization	Reduces the flexibility of object references	C++, Eiffel
Inheritance			
Of interface specification without implementation	Promotes behavior reuse and object substitution	In isolation no drawback, but if there is no implementation inheritance, then forces redundant coding	C++, Eiffel
Of implementation	Promotes code reuse	Inheritance hierarchies may not reflect object type specializations	C++, CLOS, Eiffel, Objective C, Smalltalk
Multiple	Useful when a class is viewed as a combination of two or more different superclasses	Can lead to exceedingly complex inheritance patterns, difficult to understand and maintain	C++, CLOS, Eiffel[1]

[1] A number of add-on packages provide multiple inheritance for Smalltalk programmers, although these are not widely used.

continued

Concept/ Mechanism	*Benefits*	*Drawbacks*	*Languages*
Typing			
No declarations	Less work for the developer	Omits important information that could improve implementation understandability	CLOS, Smalltalk
Formal declarations	Makes implementations easier to understand and provides necessary information for static type checking	More work for the developer	C++, Eiffel
Static type checking	Detects type errors before execution	May impede prototyping by rejecting implementations that could run	C++, Eiffel, Objective C
Dynamic type checking	Allows flexible construction and testing of implementations	Detects type errors only at runtime	CLOS, Objective C, Smalltalk
Binding			
Static	Avoids runtime lookup, or use of large amounts of memory to store compiled code for alternative execution pathways	Requires unique names for all system operations, and may require multiple code changes when requirements change	(C and Pascal)
Dynamic	Creates very flexible code that is resilient to the addition and removal of types	Incurs the overhead of binding at execution time, or the creation of extra code for alternative execution pathways	CLOS, Smalltalk
Both	Can choose the appropriate form of binding for the situation	Requires the developer to know the difference and to specify the information needed to support both	C++, Eiffel, Objective C
Object Lifetime			
Classes are objects available at runtime	Additional abstraction capability and runtime flexibility to modify and add classes	Overhead for maintaining the class information in the runtime environment	CLOS, Objective C, Smalltalk
Manual runtime storage reclamation	Allows the developer to control reclamation in special situations	Is error prone and forces the developer to deal with a low-level systems issue	C++, Objective C
Automatic runtime storage reclamation	Frees the developer from determining when space is to be reclaimed	Imposes an overhead on the runtime system to do the reclamation	CLOS, Eiffel, Smalltalk

1. Smalltalk

- language description [RBP 91]

Smalltalk was the first popular object-oriented language, developed at Xerox PARC, and its success engendered many other object-oriented languages. Smalltalk is not only a language but also a development environment incorporating some functions of an operating system. For single-user development, it offers arguably the best features of both language and environment. Smalltalk falls short in areas where it was not intended to be used—in multiple person projects and in its weak or unspecified ability to interface with external software or hardware devices. Smalltalk elegantly articulates the goals of extensibility and reusability.

All aspects of the Smalltalk language system are available through an on-line interpreter and class browser. The language syntax is simple. Variables and attributes are untyped. Everything is an object, including classes. Classes can be added, extended, tested, and debugged interactively. A garbage collector frees the programmer of the burden of memory management.

What does Smalltalk provide the implementor? Perhaps the most important contribution is the highly interactive development environment, which avoids the edit-compile-link cycle delays of the traditional compiler-based language. The Smalltalk environment permits rapid development of programs. Another strength is the class library, which was designed to be extended and adapted by adding subclasses to meet the needs of the application. Because Smalltalk is an untyped language, library components can be combined to rapidly prototype an application.

The Model/View/Controller (MVC) architecture for user interface design is another important contribution of Smalltalk. A user interface is divided into an underlying application-defined model, any number of different views of the model, and controllers that synchronize changes to the model and the views. MVC makes it possible to concentrate on the essentials of an application (the Model) and add the user-interface (the Views and Controllers) independently. The class library provides standard versions of each of these components, which can be subclassed and extended incrementally. There can be many different view/controller pairs for each model, and the views and controllers can be modified extensively with little or no change in the model. However, the MVC is a complex system that is not easy to learn.

Smalltalk is a pure object-oriented system with extensive metadata available and modifiable at run-time. Implementation of the language as an interpreter, tightly integrated with other parts of the its self-contained environment, gives ideal support for rapid incremental development and debugging.

2. C++

- language description [RBP 91]

C++ is a hybrid language, in which some entities are objects and some are not. C++ is an extension of the C language, implemented not only to add object-oriented capabilities but also to redress some of the weaknesses of the C language. Many added features are orthogonal to object-oriented programming, such as inline expansion of subroutines, overloading of functions, and function prototypes. Because of its origin as an extension of C, its backing by major computer vendors, the perception of it as a nonproprietary language, and the availability of free compilers, C++ seems likely to become the dominant object-oriented language for general use.

C++ is a strongly-typed language developed by Bjarne Stroustrup at AT&T Bell Laboratories. It was originally implemented as a preprocessor that translates C++ into standard C. As a preprocessor, C++ introduced problems for symbolic debuggers, but direct compilers are now available, and symbolic debuggers that support objects with inheritance and dynamic binding are now available. Commercial vendors offer C++ implementations for a variety of operating systems. A C++ compiler with debugger and library are available for no charge from the Free Software Foundation (with restrictions on commercial use).

Unlike several other OO languages, C++ does not contain a standard class library as part of its environment, although the standard AT&T release includes libraries for I/O, coroutine tasking, and complex arithmetic. Class libraries have been implemented by various developers, including a class library developed by the USA National Institutes of Health (NIH), which is achieving wide usage [Gorlen-90]. Class libraries for object-oriented windowing systems include Interviews [Vlissides-88] and ET++ [Weinand-88]. Unfortunately, because C++ provides no guidelines for library organization, different libraries may be incompatible. The emergence of a consensus in favor of a standard foundation class library would be an important asset to C++.

C++ contains facilities for inheritance and run-time method resolution, but a C++ data structure is not automatically object-oriented. Method resolution and the ability to override an operation in a subclass are only available if the operation is declared *virtual* in the superclass. Thus, the need to override a method must be anticipated and written into the origin class definition. Unfortunately, the writer of a class may not expect the need to define specialized subclasses or may not know what operations will have to be redefined by a subclass. This means that the superclass often must be modified when a subclass is defined and places a serious restriction on the ability to reuse library classes by creating subclasses, especially if source code for the library is not available. (Of course, you could declare *all* operations as *virtual*, at a slight cost in memory and function-calling overhead.)

The implementation of run-time method resolution is efficient. For each class, a predefined *struct* is initialized with pointers to each method available to the class. Each object contains a pointer to the method structure for its class. At run-time, a virtual operation is resolved by retrieving the method structure from the object and selecting a member to find the method address. C++ does not support run-time class descriptor objects other than the method pointer structure. C++ 2.0 supports multiple inheritance.

C++ contains good facilities for specifying access to attributes and operations of a class. Access may be permitted by methods of any class (*public*), restricted to methods of subclasses of the class (*protected*), or restricted to direct methods of the class (*private*). In addition, "spot" access can be given to a particular class or function using the *friend* declaration.

As with C, the declaration syntax of C++ is awkward and its grammar is difficult to parse. C++ supports overloaded operators: several methods that share the same name but whose arguments vary in number or type. C++ supports several memory allocation strategies for objects—statically allocated by the compiler, stack-based, and allocated at run-time from a heap. The programmer must avoid mixing objects of different memory types or dangling references may cause run-time failures. Each class can have several *constructor* and *conversion* functions, which initialize new objects and convert between types for assignment and argument passing; these are semantically sound but perhaps somewhat confusing for normal use.

In summary, C++ is a complex, malleable language characterized by a concern for the early detection of errors, various implementation choices, and run-time efficiency at the expense of some design flexibility and simplicity.

3. Eiffel

- language description [RBP 91]

Eiffel is a strongly typed object-oriented language written by Bertrand Meyer. Programs consist of collections of class declarations that include methods. Multiple inheritance, parameterized classes (*generics*), memory management, and assertions are supported. A modest class library is provided, including lists, trees, stacks, queues, files, strings, hash tables, and binary trees. For portability, the Eiffel compiler translates source programs into C. Eiffel has good software engineering facilities for encapsulation, access control, renaming, and scope. Eiffel is arguably the best commercial OO language in terms of its technical capabilities.

The focal point of Eiffel is the class declaration, which lists attributes and operations. Eiffel provides uniform access to both attributes and operations by abstracting them into a single concept called a feature. An Eiffel class declaration may include a list of exported features, a list of ancestor classes, and a list of feature declarations. Eiffel does not treat either classes or associations as first class objects.

Eiffel supports memory management through a coroutine which detects objects that are no longer referenced and releases the memory allocated to them. The Eiffel run-time system executes the coroutine whenever the available memory space is low. Several mechanisms are provided to control memory management. Automatic execution of the coroutine may be suppressed through a compiler switch or turned on or off at run-time. For operating systems that

do not support virtual memory, there is a compiler switch to arrange for Eiffel's run-time system to provide automatic paging.

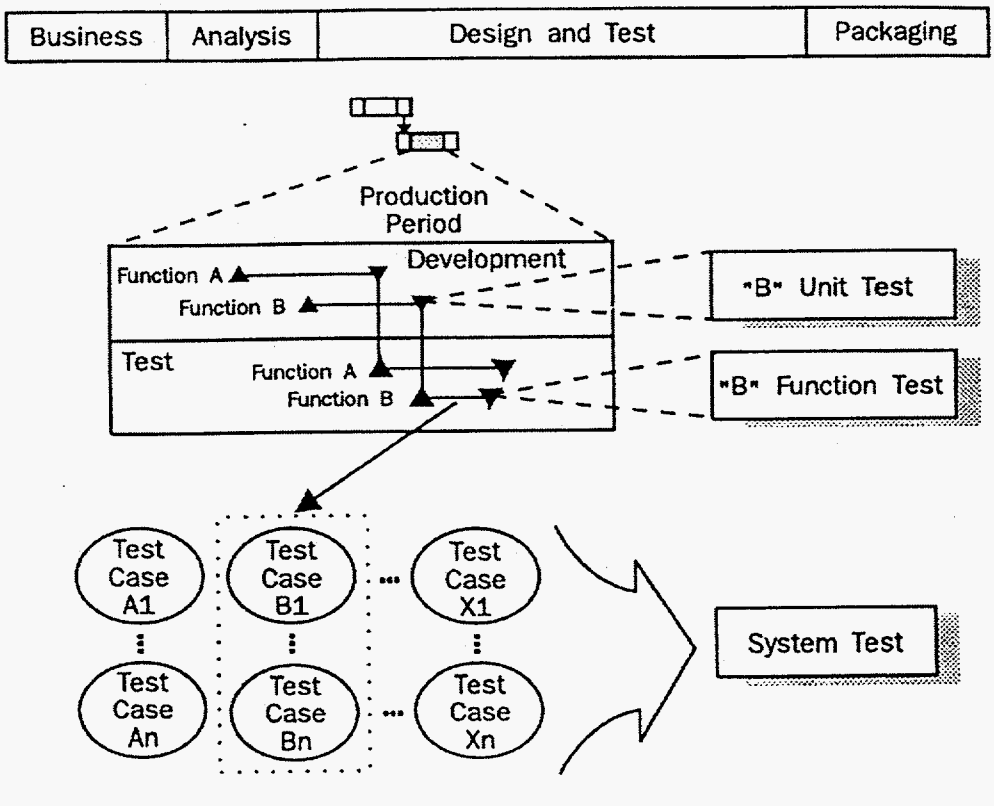
A contractual model of programming is supported by preconditions, postconditions, invariants, and exceptions. A *precondition* is a condition that the caller of an operation agrees to satisfy. A *postcondition* is one the operation itself agrees to achieve. An *invariant* is a condition that a class must satisfy at all stable times. Conditions and invariants are a part of the class declaration and must also be obeyed by all descendent classes. If they are violated at run-time, an exception occurs, which either causes the faulty operation to fail or executes an exception handler for the class if the programmer provides one. Compiler switches provide several levels of error checking. Once an application is debugged you can turn off assertion checking.

4. Java

- Java resembles C++ in syntax but simplifies many of its complexities and avoids the hybridization present in C++.

E. Quality assurance

- testing process [Lo 93]



- **Tests, reviews, inspections,** and **metrics** are complementary techniques for assuring software quality.

1. Tests

- testing levels

- unit
- integration
- system

- testing approaches [JCJ 92]
  - regression test
  - operation test
  - full-scale test
  - performance/capacity test
  - overload test
  - negative test
  - requirement test
  - ergonomic test
  - user documentation test
  - acceptance test

2. Inspections and reviews

- An **inspection** is a formal and detailed group examination of a design or implementation.
  - "Checklists focus the reviewers' attention on areas that have been problems in the past.
  - "The emphasis is on defect detection, not correction.
  - "Reviewers prepare for the inspection meeting beforehand and arrive with a list of the problems they've discovered.
  - "Distinct roles are assigned to all participants.
  - "The moderator of the inspection isn't the author of the work product under inspection.
  - "The moderator has received specific training in moderating inspections.
  - "Data is collected at each inspection and is fed into future inspections to improve them.
  - "General management doesn't attend the inspection meeting. Technical leaders might." [Mc 93]
- A **review** is an informal group examination of a design or implementation.

3. Metrics

- **Metrics** (measurements) can be used to assess the quality of design .and implementation.
- "System quality
  - "Stability
  - "Defect density
  - "Defect-discovery rate" [Bo 94]

- "Size/complexity
  - "Number of classes
  - "Classes per category" [Bo 94]
- "Class quality
  - "Number of operations
  - "Shape of inheritance lattice
  - "Number of children
  - "Coupling/cohesion
  - "Response
  - "Primitive/sufficient/complete" [Bo 94]

III. Other topics

A. *User interfaces*

- characteristics
  - "*Users see objects and choices displayed graphically*, and choose them by pointing with the mouse cursor. . . .
  - "*The syntax of commands is 'object-action'*; the user selects an object (by pointing and clicking) and then specifies the action on it. . . .
  - "*Users get immediate feedback from actions.* This is part of providing the feeling of **direct manipulation** . . .
  - "*The interface is modeless.* Modes are global states of the interface that affect the meaning of user actions. . . . . Modes can be useful, but if they are not visible, they can be disturbing because they interfere with the user's ability to predict the results of actions. . . .
  - "*The interface displays objects in WYSIWG form ('What You See Is What You Get').*
  - "*Objects and actions are consistent* both within an application, and across different applications. . . ." [Co 95]

B. *Databases*

- "The idea of an Object DBMS (**ODBMS**) is to store the objects as such, and thus bridge the semantic gap all the way to the database." [GR 95]
- criteria for ODBMS in addition to DBMS criteria
  - "*Complex objects.* It should support the notion of complex objects.
  - "*Object identity.* Each object must have an identity independent of its internal values.
  - "*Encapsulation.* It must support encapsulation of data and behavior in objects.
  - "*Types or classes.* It should support a structuring mechanism, in the form of either types of classes.
  - "*Hierarchies.* It should support the notion of inheritance.

- "*Late binding.* It should support overriding and late binding.
- "*Completeness.* The manipulation language should be able to express every computable function.
- "*Extensibility.* It should be possible to add new types." [GR 95]

C. *Distributed objects*

- **Distributed objects** is a style of computing where objects communicate with one another over a network.
- An Object Request Broker (**ORB**) "acts as an intermediary between requests sent from clients to servers." [OPR 96]
- **CORBA** and **COM** are the two major specifications for distributing objects.

IV. References

[Bo 94] G. Booch, *Object-Oriented Analysis and Design with Applications*, (Redwood City, California: Benjamin/Cummings Publishing Company, 1994).

[Bo 96] G. Booch, *Object Solutions: Managing the Object-Oriented Project*, (Menlo Park, California: Addison-Wesley Publishing Company, 1996).

[Ca 94] R. G. G. Cattel, *Object Data Management*, (Reading, Massachusetts: Addison-Wesley Publishing Company, 1994).

[Co 95] D. Collins, *Designing Object-Oriented User Interfaces*, (Redwood City, California: Benjamin/Cummings Publishing Company, 1995).

[CS 94] Catalyst Solutions, *Object-Oriented Analysis and Design with C++*, (n.c.: n.p., 1994).

[GG 93] T. Gilb and D. Graham, *Software Inspection*, (Wolkingham, England: Addison-Wesley Publishing Company, 1993).

[GHJ 95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, (Reading, Massachusetts: Addison-Wesley Publishing Company, 1995).

[GR 95] A. Goldberg and K. S. Rubin, *Succeeding with Objects: Decision Frameworks for Project Management*, (Reading, Massachusetts: Addison-Wesley Publishing Company, 1995).

[JCJ 92] I. Jacobson, M. Christerson, P. Johsson, and G. Övergaard, Object-Oriented Software Engineering: A Use Case Driven Approach, (Wokingham, England: Addison-Wesley Publishing Company, 1992).

[Kh 93] S. Khoshafian, *Object-Oriented Databases*, (New York: John Wiley & Sons, 1993).

[La 94] W. LaLonde, *Discovering Smalltalk*, (Redwood City, California: Benjamin/Cummings Publishing Company, 1994).

[Lo 93] M. Lorenz, *Object-Oriented Software Development*, (Englewood Cliffs, New Jersey: Prentice Hall, 1993).

[Lo 95] M. Lorenz, *Rapid Software Development with Smalltalk*, (New York: SIGS Books, 1995).

[Ka 95] S. H. Kan, *Metrics and Models in Software Quality Engineering*, (Reading, Massachusetts: Addison-Wesley Publishing Company, 1995).

[Ma 94] S. Maguire, *Debugging the Development Process*, (Redmond, Washington: Microsoft Press, 1994).

[Ma 95] R. C. Martin, *Designing Object-Oriented C++ Applications Using the Booch Method*, (Englewood Cliffs, New Jersey: Prentice-Hall, 1995).

[Mc 93] S. McConnell, *Code Complete*, (Redmond, Washington: Microsoft Press, 1993).

[OPR 96] R. Otte, P. Patrick, and M. Roy, *Understanding CORBA*, (Upper Saddle River, New Jersey: Prentice-Hall, 1996).

[RBP 91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, (Englewood Cliffs, New Jersey: Prentice Hall, 1991).

[Wi 95] N. Wilkinson, *Using CRC Cards*, (New York: SIGS Books, 1995).