

# OO miniFAQ v0.61

By William Paul Fiefer

## Contents and quick review

### Introduction

Read this FAQ before using this quick review.

### 1) Object

- What is it?

An object is the executable created from a class and is a package of methods and the variables they manipulate. You should use an object's methods to access its variables. Methods reflect an object's functionality; variables reflect its state.

- What good is it?

Your executing program is a collection of communicating objects. Objects communicate with other objects only by sending messages to methods. You create modular systems by using objects, messages, and methods.

### 2) Message

- What is it?

A message is the same as a function call and is the only way objects should request and receive services from other objects. These services are the outputs of an object's methods.

- What good is it?

Messaging channelizes the lines of communication between objects. This control reduces the assumptions you need to make about software behavior and promotes reuse and modular systems design.

### 3) Class

- What is it?

A class defines an object. Classes are the actual focus of object-oriented design methods, and serve as blueprints for objects. They are not executable. You must initialize a variable with a class to make an object. Classes can be trimmed, expanded, and modified to create other specific, desired classes or permutations of classes.

- What good is it?

Classes promote software reuse. Software built around classes is easier to modify and maintain than software built around functions alone.

#### 4) Inheritance

- What is it?

Inheritance is one of two ways to create new classes (the other way is through “aggregation”). The final versions of these new classes become objects in an executing system.

- What good is it?

Inheritance promotes reuse by making new classes from existing classes. Inheritance lets you easily incorporate old code in new designs.

#### 5) Finite state machine

- What is it?

A finite state machine depicts the permissible defined behaviors of a system. It is a simulation of the behavior of your system.

- What good is it?

Finite state machines expand your understanding of systems and help you debug them by depicting their permissible states. A finite state machine should describe all the states of the system and all the entry and exit points for those states.

#### 6) Methodology

- What is it?

Methodologies are sets of techniques for building software systems.

- What good is it?

Methodologies let you reuse the software development techniques that work, and tune or replace those that do not.

#### 7) Booch 94

- What is it?

Booch 94 is one of the most widely used object-oriented systems development methodologies available. Booch 94 consists of a lifecycle development process and a graphical systems modeling notation.

- What good is it?

Large corporations have successfully used Booch 94 to develop major software systems. Booch 94 is available as an automated CASE (computer-aided software engineering) product that consistency checks analysis and design, and generates executable source code.

#### Afterword: The economies of scale

## Introduction

### *Object-oriented software recipe*

*Encapsulate carefully, combining select variables with method signatures to make basic classes. Blend classes to make new classes (season judiciously with polymorphism). Arrange classes, garnishing with relationships for maximum effect. Prepare method implementations, storing these separately from their classes. Infuse with documentation. Assign choice classes to external variables to make objects. Heat. Allow calls between objects to rise. Test frequently for quality. Serves many.*

Object-oriented programming languages are now mainstream business development tools. Thanks to the popularization of C++, Smalltalk, object-oriented COBOL, Java, and certain corporate schools of systems thinking, the object-oriented concept has received solid media attention as well. The object orientation promises more comprehensible code, shorter development times, fewer logic defects, and greater reuse of existing source in new applications. These benefits, if realized, contribute strongly to any case for conducting new development with object-oriented rather than traditional techniques.

This *OO miniFAQ* is a brief survey of the central concepts of object-oriented (OO) development. Once you have read the *OO miniFAQ*, the quick review embedded in the table of contents will make sense. OO thinking is different, but not as most people assume. OO thinking raises the level of abstraction by one increment from procedural orientations, in the way COBOL boosted the abstraction an increment over assembler. The new emphasis is on classes, which supplant subroutines and variables.

That is all. There is no magic. But that is a big *all*, because it changes the way you will analyze systems and organize code.

Specifically, OO programming languages permit you to easily create complex new types (*classes*) either from scratch (through the *encapsulation* of variables and functions) or by merging one or more existing classes (single or multiple *inheritance*). These classes get used like any other types: by assigning their characteristics to a variable, at which point they are referred to as *objects* or *instances*. Objects, which are executable bundles of variables and functions, link with each other through function calls (*message passing*).

An OO function is called a *method*. OO Languages have several key characteristics. When you build a class, you stub in method names and parameters, and store their implementations elsewhere. You also fence off access to the variables within classes and their objects (*information hiding*), and so restrict their use to specified methods. Information hiding extends to the implementation of a method, so the user of a method deals with it through its name and parameters alone. Finally, you can wait until runtime to specify the class containing the implementation of a method you are calling. At runtime, the system will assign the correct implementation to your method call (*polymorphism*).

You do not need to use an OO language to create programs that offer these features, but because OO languages are designed from scratch to contain these features, using them to do so is more efficient. OO concepts scale smoothly, and so apply equally well to the subsystems of large software applications. When the OO ideas are scaled up, they become parts of OO analysis and design methodologies.

OO ideas are powerful tools for reducing the complexity of large software systems. Systems are large now, larger than they have ever been before. And they keep getting bigger. Big systems make for disproportionately big development, modification, and maintenance headaches because they offer more opportunities for poor organization and awkward construction. OO techniques offers an escape from this. OO lets you think clearly about big things.



# OO miniFAQ v0.61

## 1) Object

- What is it?

*Objects*, or collections of objects, are executable components of a software system. Inside an object are a set of related functions, their implementation, and the data those functions use. An object exists in live memory.

You should not access the function implementations or data in an object directly. You should use one of the object's functions to access that data. You should not construct methods inside an object that directly access the variables inside another. This violates the OO paradigm and most OO programming languages enforce these rules.

Objects have names, and this is how you locate a particular function to accept some parameters, do some work, or access a particular piece of data.

In OO terminology, a function is called a *method* or *operation*, and the data are called *variables* or *attributes*. The object should be regarded as a shell, encapsulating the variables, methods, and implementation of the methods.

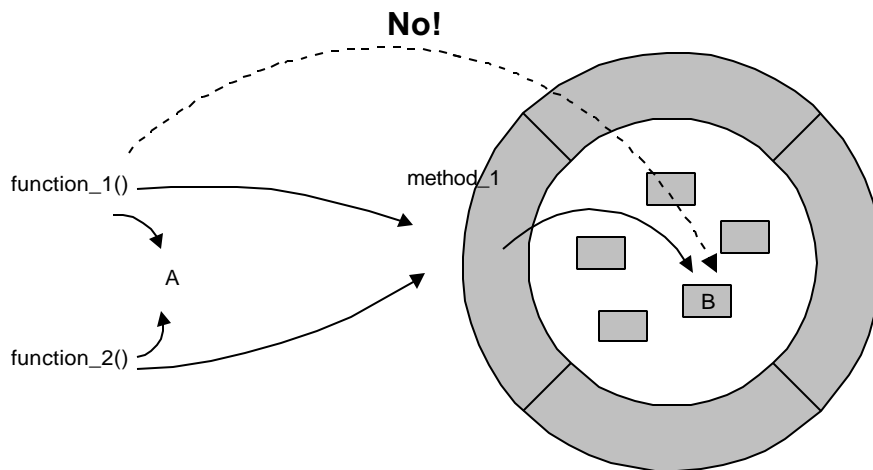
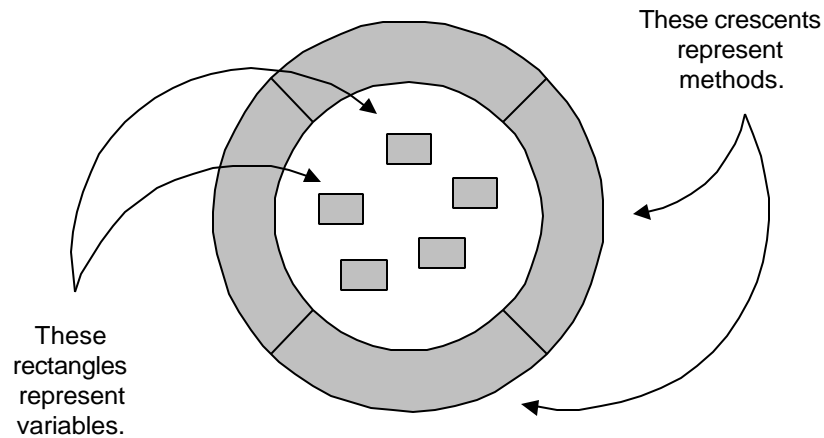
*Encapsulation* is a fundamental characteristic of objects. It is an aspect of their structure. The following code fragment depicts encapsulation:

```
01 STUDENT-RECORD.
   05 NAME.
       10 LASTNAME          PIC X(20).
       10 FIRSTNAME        PIC X(20).
   05 SS-NUMBER            PIC 9(9).
   05 DATE-OF-BIRTH       PIC 9(8).
```

Most people do not expect to see a COBOL record. Yet, "STUDENT-RECORD" encapsulates the name, social security number, and birthdate fields. You refer to this aggregate as "STUDENT-RECORD." And "NAME" further encapsulates last name and first name information. The record description is an abstraction until you populate it with data. Then it is an object.

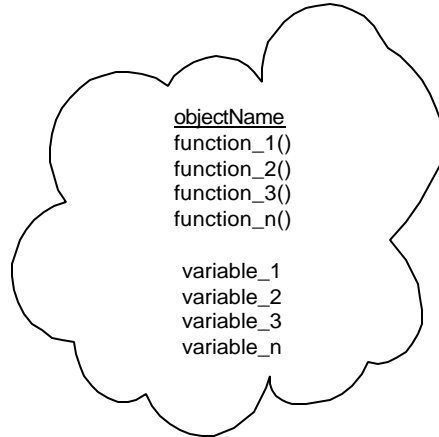
Encapsulation is means to support a condition known as *information hiding*. Information hiding permits you to view the inside of a software entity (its variables) only through an agreed upon access mechanism. You do not violate this abstraction to examine its contents even if you can. You use the entity's *interface*, which is shorthand for "an agreed upon access mechanism." Methods are the interface to an object. When you pass STUDENT-RECORD to a module and disregard STUDENT-RECORD subfields, STUDENT-RECORD is the interface. It is the tightest level of detail you think about. Information hiding is in effect.

The next two diagrams, expressed in what we call Taylor notation (after its inventor David Taylor), depict objects encapsulating methods and variables, and the methods mediating access to the variables:



Procedural languages do not tightly regulate access to variables. Any function, like functions 1() and 2() in the preceding diagram, can directly manipulate unencapsulated variable A. But to reach encapsulated variable B, functions 1() and 2() must use method\_1(), the interface.

Taylor notation is not widely used. More common is the Booch 94 notation, shown next, which we will use for the remainder of the *OO miniFAQ*:



Booch 94 depicts the object encapsulating methods and variables but does not clearly depict the interface relationship between the methods and variables.

Objects are real items that occupy space in memory. They generally serve to model an element of the real world and are named for that element. An object's methods mimic the actions of that element, and its variables store information that you believe the element keeps track of. Everything else is superfluous.

As an example, next is an OO description of a vending machine:

```

Name:
    VendingMachine
Purpose:
    To dispense canned beverages
    after receiving proper payment.
    This dispenser does not regulate
    the temperature of the beverages.
Methods:
    receiveChange()
    returnChange()
    acceptSelection()
    dispenseSelection()
Variables
    changeSubmitted
    changeDue
    beverageInventory

```

Now we can depict this using Booch 94:



This is a very simple model of a vending machine. It has methods that accept money, return change, accept a beverage selection, and dispense a beverage. During operation, its variables keep track of how much money the customer has deposited, how much change the customer is owed, and the quantities on hand for each beverage. The implementation of the methods and structure of the variables are hidden because they are irrelevant.

At any time, the contents of the variables reflect the object's current condition or properties – its *state*. The methods represent its permissible behavior. The object encapsulates methods and variables (shown in Taylor and Booch 94), and the methods channel access to the variables (shown clearly in Taylor).

- **What good is it?**

An object is an indivisible, encapsulated bundle of methods and variables. It is an island of data packaged with the functions using that data. You use the object as a black box (information hiding), knowing only the methods you can ask of and what parameters those methods take it (the interface). The object is similar to a specialized library of functions.

This promotes *modularity*, allowing objects to be used as components in other systems. You do not worry about the code inside an object or the format of an object's variables. The internals are irrelevant! All you need to think about is whether an object's methods do the job you want done. If they do, then you send a message to that object.

## 2) Message

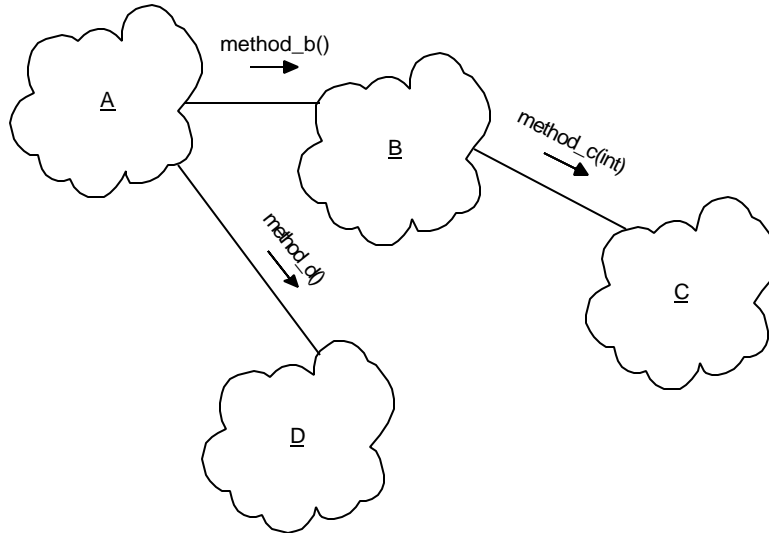
- **What is it?**

A *message* is a function call. The only way objects interact is by passing messages to other objects, asking them to perform methods for them. The receiving object can be on the same machine as the sending object or on another machine.

A message is built of three parts: the name of the receiving object, the method requested, and any parameters needed by the method:

**Message = ReceivingObjectName + method\_name() + parameters**

In the next diagram, object A is sending messages to objects B and D, and object B is sending a message to object C. Only method\_c takes a parameter (an integer argument). The connecting lines indicate two objects do business together; the arrows point from the sending to the receiving object.



If our vending machine is attached to a computer network, it can pass messages to its owner each night reporting how much money it contains, how many beverages remain, and whether it is time to empty it of change or replenish its stock.

- **What good is it?**

Without message passing, not much gets done. Messages support all of the interaction between objects and they let you distribute objects among many processes and machines with remote procedure calls. The messaging between objects plus the methods executed constitutes the behavior of a system.

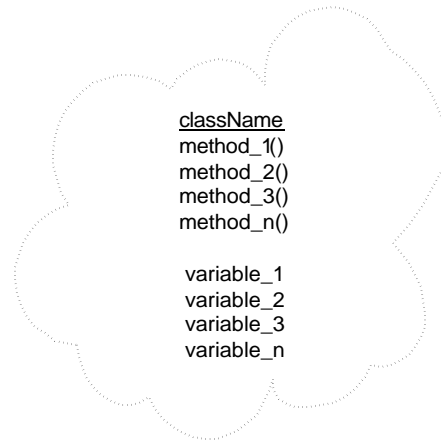
All access to an object should be through its methods via a message. Objects are connected by the messages they exchange rather than by the details of their implementation. This promotes modularity, reduces coupling, makes systems simpler to build, and easier to understand and maintain.

### 3) Class

- **What is it?**

A *class* is a template, blueprint, or prototype for objects or other classes. A class defines an object and is virtually synonymous with the concept of a type. Classes are

used to make objects and are not executable. The next diagram depicts a class. It resembles an object but with a dotted border to indicate that a class does not actually exist.



Since a class is a blueprint for an object, it too should be regarded as a shell, encapsulating variables, their defaults, and the interface methods. Encapsulation is fundamental to forming classes, and objects exhibit encapsulation because of this. But a class is not the object any more than the blueprint of a house is a house. A class is the name of something and the specification of its contents, meaning its methods, attributes, and, *sometimes*, its implementation.

(Most OO programming languages let you place the implementation of a method outside the class, leaving only the method name, arguments, and return type – its *signature* – inside the class. This separation is an OO linchpin because it serves to firmly segregate a software specification from its implementation. When the implementation of a class is mentioned in the *OO miniFAQ*, this physical separation is assumed.)

The act of creating an object from a class is called *instantiation*. If you have a class called `SoftDrinkVendingMachine`, you can stamp out all the `SoftDrinkVendingMachine` objects you desire or need your application to track.

- **What good is it?**

Classes promote *reuse*. Rather than writing the implementation details of an object from scratch, classes let you use code from an existing class. This saves time and effort.

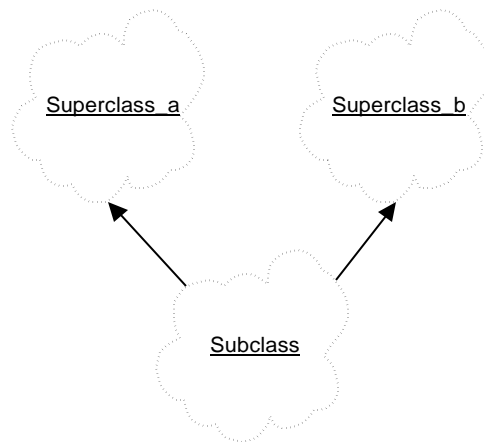
Do not confuse class reuse with copybooks or the like. Unlike copybooks, changes made to the original class source file propagate to every class reusing it. Class source is broadcast by the system from one central, authoritative image. The advantage to you is that you edit and debug only one file. The disadvantage is that your changes require a recompilation of all code touching the changes since its source might be broken.

As an aside, reuse runs along a broad continuum. You can copy a few lines of code (initializing an array), put a handy utility subroutine into a library (create an English date), build a class (SoftDrinkVendingMachine), generalize a conceptual mechanism across many programs (balanced line master file update algorithm), reuse an entire program (an Internet Web browser), or reuse several interdependent programs within an application or business domain (office suites).

## 4) Inheritance

- What is it?

*Inheritance* is the procedure used to build a class from one or more existing classes. The original classes are called *superclasses* or *base classes*. The resulting class is called a *subclass* or *derived class*. The next diagram depicts inheritance:

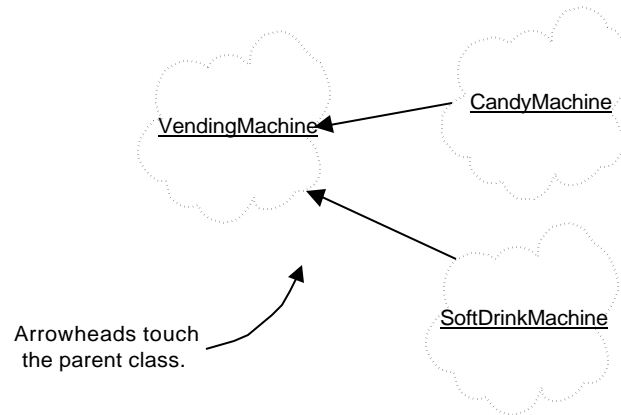


Booch 94 notation is paternal, and the arrows point to the parent superclasses. In the preceding diagram, the subclass is defined as some specific merger of the methods and variables of superclasses “a” and “b.” The arrowheads touch the superclasses. When the subclass is instantiated, the object created will exhibit a blend of the properties of the superclasses “a” and “b.” You control the specifics of that blend through inheritance, and can eliminate, modify, or add parts to the subclass, *overriding* those of the original.

When a subclass is the product of one superclass, the process is called *single inheritance*. When the subclass is the composite of two or more superclasses, the process is called *multiple inheritance*. The subclass cuts, maintains, or adds to the superclass methods and variables, and can override superclass implementation details.

Inheritance reflects a clear *relationship* between two or more classes in which the subclass is a variation of the superclass. This is called an *is-a*, or *generalization-specialization* (gen-spec), relationship. A candy machine, for example, is a type of vending machine, and typical of is-a relationships.

The next diagram depicts an is-a relationship:

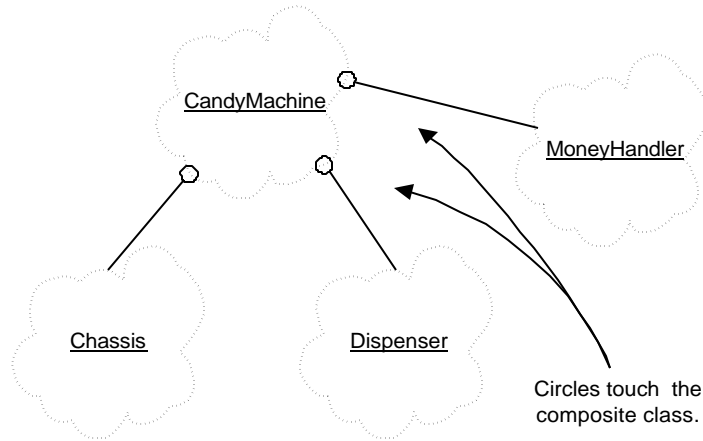


Candy dispensers and drink dispensers are each types of vending machine. This is an is-a, or generalization-specialization (gen-spec), relationship. *Arrowheads touch the parent.*

Inheritance is one kind of relationship among classes. Another kind of relationship is the *part-of* or *aggregate* relationship. Part-of relationships are *not* inheritance relationships.

A candy machine, for example, is composed of a dispenser mechanism, a money-handler mechanism, and a chassis to hold these components. Classes that are aggregates of several classes are typical of part-of relationships. Composite classes either totally contain their constituent classes or have system pointers (address references) to them. Both relationships can exist simultaneously for a given class. A candy machine, for example, is simultaneously a descendent of an archetypal vending machine (inheritance) as well as a composite class, built of many components (part-of).

The following diagram depicts a part-of relationship:



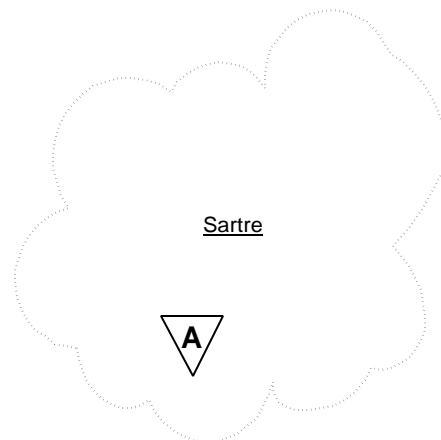
Candy machines are built of dispensing mechanisms, money-handling mechanisms, and a chassis to hold these parts together. This is a part-of relationship. Booch 94 notation uses open circles rather than arrowheads for composite relations, and the *circles touch the composite class*.

Inheritance and part-of diagrams take the form of trees or lattices, which can get very dense.

- **What good is it?**

The subclasses created with inheritance specialize the variables and methods of their superclass. This promotes reuse because the superclass serves as a template for subclasses.

Inheritance can get sophisticated. You can, for example, create generic superclasses (*abstract classes*) consisting of partially implemented or totally unimplemented methods that get filled in during later inheritance procedures. In Java these are called *interfaces*. An abstract class is depicted next.



The apex-down triangle contains an “A” indicating the class is abstract. The class name is underscored.

You cannot create objects from abstract classes because of they are incomplete, but you have the flexibility of later merging them with classes that supply the missing implementation. These new classes can be instantiated. Classes capable of instantiation are called *concrete classes*. The classes depicted elsewhere in the *OO miniFAQ* are concrete classes.

Here is another interesting twist on inheritance, which is best explained by an example. First, you start by writing multiple implementations for a superclass method, one for each separate subclass of that superclass. Next, you send a message to that method without specifying the containing subclass. Instead, you replace the name of the containing subclass with a variable that gets defined at runtime. Recall that

**Message = ReceivingObjectName + method\_name() + parameters**

You are leaving “ReceivingObjectName” a variable.

When your program runs and the message is dispatched, the runtime system will execute the correct implementation. Variable, type, and value get bound on the fly. The OO system runs the right method implementation when given a dynamically defined containing subclass. This is called *polymorphism* and it lends expressive power to OO systems.

## 5) Finite state machine diagram

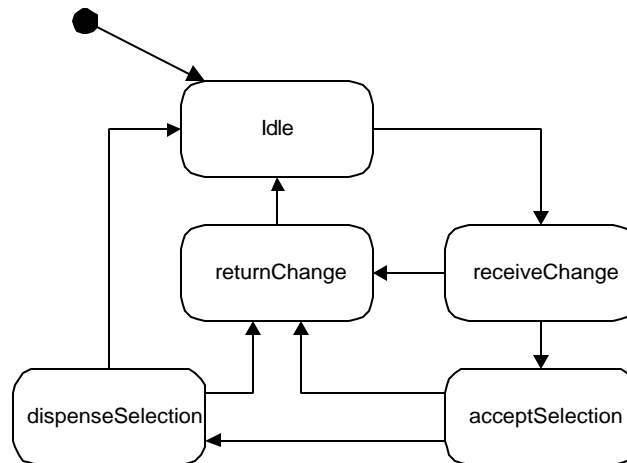
- What is it?

A *finite state machine diagram* depicts all possible conditions (*states*) your program can reach and how your program gets to and from these states. The list of all these states is called the *state space*. A program is considered a finite state machine because it enters only a few (a finite number of) states. A finite state machine is a paper simulation of the behavior of your system.

State space occupies a range of values over one or more variables. For example, your thermal state space fluctuates several degrees above and below 98.6 degrees, cooling at night and rising later in the day. Body temperature is the variable over which thermal state space roams. A specific state must be clearly defined (three thermal states are “base” and “peak” temperatures, and “serious fever,” a state of maintaining a temperature exceeding 104 degrees Fahrenheit for more than four hours). Your financial state space, as measured by your checking account balance, rises and falls in a specific pattern through most months (and you have many state possibilities here, including “below-minimum” and “interest-earning”). A multivariant state space

composed of your thermal and financial states is more complex and potentially more interesting.

The next diagram depicts the state space for our vending machine and the entry/exit paths among these states.



The large arrow with the marked tail is a clerical device, signifying the entry point to the machine. State flow moves as directed by the arrows (for example, you cannot go from “returnChange” to “dispenseSelection”). Once you enter the machine, it is assumed to run forever (that is, maintain a *steady state*).

- **What good is it?**

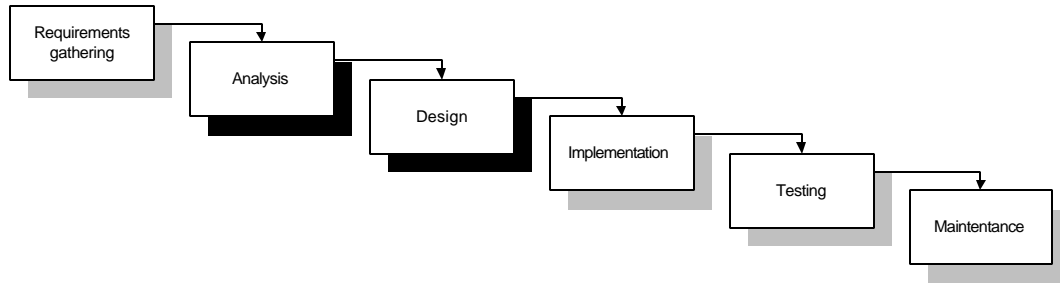
A finite state machine lets you track the behavior of your system. This is valuable for developing test suites and for finding out whether you have any states that can be entered but not exited or exited but not entered. Bugs often live in the states you never knew your program had.

## 6) Methodology

- **What is it?**

A *methodology* is a recipe for building software systems. Methodologies usually consist of a notation for modeling the planned system and a process for building the system. In a sense, notation is the input data to process, which is the software for running a human/social machine that builds systems. You must emphasize the word “recipe” and realize that any methodology needs to be tailored for specific systems and specific development environments.

Different methodologies can cover different phases of the development *lifecycle* which by convention spans requirements gathering, analysis, design, implementation, testing, and maintenance. The heart of most methodologies, however, is analysis and design. The next diagram depicts a standard lifecycle model:



Each development phase has a specific purpose:

**Requirements gathering** collects the needs and expectations of your user community.

**Analysis** formalizes those needs and expectations into a requirements specification document describing what the system will do.

**Design** converts the requirements specification document into a set of models that describe how the system will work.

**Implementation** transforms the design models into a source code description of the system.

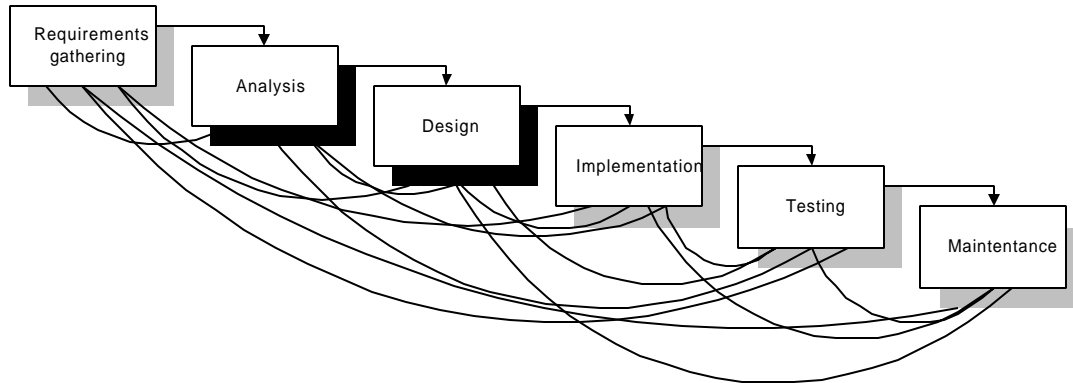
**Testing** determines whether the coded system does what it is supposed to do.

**Maintenance** corrects and perfects the system.

The diagram reflects a sequential form of software development, in which a phase is entered, completed, exited, and a new phase begins. This is known as the **waterfall model** and it *does not work* for most software development and for OO development in particular.

The model does not work because it does not include entry points and procedures for reversing direction or running in two phases at once. The model also has no provisions for building prototypes. The complexity of software systems means development teams must cycle backward or operate parallel phases to get everything done properly. Developers build many prototypes to figure out which direction to move next, which phase to enter next, and which features to add, drop, or change. The waterfall makes you wait until the end of a project to see a working copy of the code.

Consequently, the lifecycle model should really look more like the following diagram:



The process is wrapped in layers not shown here. Project management is a wrapper, as are technical reviews, software quality assurance, version control, and the human dynamics of the physical environment where everything takes place. This last wrapper is another reason to emphasize the recipe aspect of your chosen methodology.

OO methodologies use vocabularies slightly different from the one used here. Here is a conversion table:

| <b><i>OO miniFAQ</i></b> | <b>Synonyms</b>                                      |
|--------------------------|--|
| Abstract class           | Interface  |
| Class                    | Object   |
| Concrete class           | *  |
| Encapsulation            | *  |
| Implementation           | Algorithm, Representation, Source code               |
| Inheritance              | *  |
| Instantiation            | Declaration, Template expansion                      |
| Is-a                     | Gen-spec, Generalization-specialization, Inheritance |
| Message                  | Function, Operation                                  |
| Message passing          | Communication, Function call                         |
| Method                   | Message, Operation, Function                         |
| Methodology              | Heuristic, Method, Technique                         |
| Object                   | Instance   |
| Part-of                  | Aggregate, Composite object, Compound object, Uses   |
| Signature                | Interface  |
| Subclass                 | Derived class  |
| Superclass               | Base class   |
| Variable                 | Attribute, Data                                      |
| Waterfall model          | Linear model, Linear-sequential model                |

- **What good is it?**

Systems are too large and complex to build without the governance provided by a methodology. You should use a methodology for a variety of reasons. A methodology

*Lets you simplify the problem by partitioning and decomposing it:* Methodologies usually contain toolkits that include procedures for dividing problems into more easily solved components.

*Gives you heuristics for software development:* Methodologies consist of sequences of activities that increase the prospects of successfully producing or modifying a software system and successfully repeating those steps every time you produce or modify a software system.

*Links developers through a common vocabulary and graphic notation:* Methodologies establish consistent languages of expression that allow developers to communicate software development concepts precisely and efficiently.

*Gives you multiple perspectives of the software system:* Methodologies prescribe constructing several orthogonal models of the system under development (similar to the orthogonal top-, side-, and angled-view models of a building constructed by architects).

*Enforces consistency among and within the software system models:* Methodologies prescribe procedures to prevent different developers from naming the same data or functions differently, or from assigning the same function different signatures.

*Reduces redundancy among and within the software system models:* Methodologies establish central repositories that contain the master, authoritative versions of models, source code, and documentation.

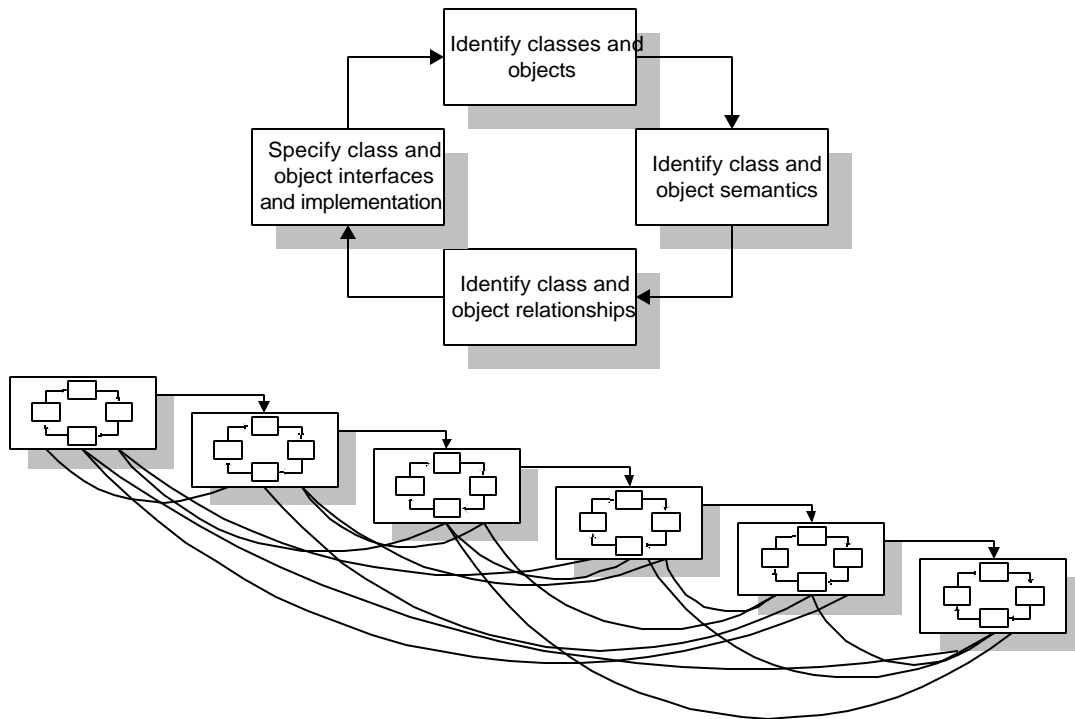
*Formalizes roles and responsibilities in the development process:* Methodologies define who will accomplish particular tasks and what they will be expected to do.

## 7) **Booch 94**

- **What is it?**

**Booch 94**, developed by Grady Booch of the Rational Software Corporation, has been one of the most widely used OO software systems development methodologies in the United States. Booch 94 is a methodology that covers analysis and design.

Central to Booch 94 is the *micro development process*, an iterative sequence of activities nested within each phase of the standard software process. This process and its nesting are illustrated next.



The micro process centers around identifying and building classes. Each major phase of development consists of at least one trip (or *spin*) through the micro process. The steps within each spin remain unchanged, but the nature of the steps changes as development proceeds.

Spins in early phases (requirements gathering and analysis) emphasize finding classes and recording their general features. Late phase (design and implementation) spins specify and code class details. Successive spins *inside a particular phase* serve to sharpen or expose the details of their target classes. When you think of the effect of spins on classes, think of watching pieces of film develop.

The micro process accomplishes two things: First, it defines a set of activities common to every development phases. Second, it dissolves boundaries between development phases, freeing developers to return to earlier phases, gather more information, make improvements, issue corrections, and evolve their system.

James Rumbaugh's OMT (Object Modeling Technique) is regarded as easier to learn for persons with a process-oriented background. Booch 94 is pure OO, focused on the behavior of objects. OMT requires developers to consider data models in developing systems and thus is regarded as a bridge methodology, spanning OO and structured techniques.

Rumbaugh is with Rational, too, and is working with Booch and Ivar Jacobson, another well-regarded methodologist, to promote the UML (Unified Modeling Language) notation. (Booch, Rumbaugh, and Jacobson are colloquially known as the

three amigos.) The UML can express most constructs a software developer can imagine. But detractors say UML is devoid of new approaches and so does not advance the art of development. UML has replaced Booch 94 notation as the standard, but the detractors have a point: there is nothing fresh from Rational or anyone else.

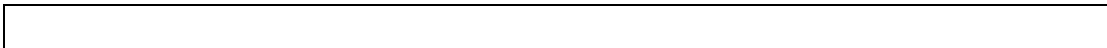
- What good is it?

Rational is the CASE tool market leader, according to Ed Yourdon). Rational says their object techniques have been heavily tested on large applications by major U.S. and European firms, including Smith Barney, Ericsson, and British Airways. Booch says he has personally performed work for Apple, AT&T, Boeing, IBM, Lockheed, Motorola, Rockwell International, Shell International, and many others, all of which has given them an edge in building and refining a solid methodology.

Rational says it continues improving the support tools associated with its methodology. These tools are called *CASE* (computer-aided software engineering) tools, and they take the products of analysis and design and generate executable source code in several languages including C++, Smalltalk, and Java.

Rational has also joined with Microsoft. Microsoft packages subsets of Rational CASE tools within Microsoft's applications development products, and Rational packages subsets of Microsoft's development tools in its CASE tools. Yourdon has said Rational and Microsoft will divide the market into code development (Microsoft's "Visual" series of products) and lifecycle development (Rational's CASE tools).

Rational is obviously positioning itself as the Microsoft of the methodology industry, hiring and contracting the methodologists, populating the industry committees, proposing the standards, and producing the software that automates much of the labor involved in developing systems. Rational is pushing the agenda and the three amigos are major players. Given Rational's position in the marketplace, they should be able to provide continuing support and upgrades for their products for the functional lifetime of those products.



## Afterword: The economy of scale

A mechanical engineer works with fewer than ten fundamental conceptual building blocks when constructing a machine. The screw, lever, wedge, gear, and pulley are some of these prototypical concepts – simple machines – from which all machines are built. Engineers do not directly use screws, levers, wedges, or any fundamental component (“Put a screw here”). They use specific instantiations of them (“Put a 12mm 24-pitch chrome-moly hex-head metal screw here”). Engineering is all about being specific.

Programmers work with three fundamental components. These are the sequential, branching, and looping statements. In 1966, Corrado Böhm and Guiseppi Jacopini proved these formation rules were sufficient to produce all possible source code. Programmers do not directly use these concepts (“Put a sequence here”). They use specific instantiations of them (“Declare two integer variables, assign 0 to each, then use the first as a tally and the second to maintain a running total”). Programming is all about being specific, too.

If the world ran on simple, tiny machines and programs, these abstractions would be enough. They are not. Machines and programs (for example, cars and the software they run) use hundreds of thousands and millions of fundamental parts. The only way designers can mentally control this complexity is to elevate the abstraction to an aggregate. Encapsulation and information hiding shield you from getting swamped by the details.

Big systems such as cars are built from big components, which themselves are built from smaller components, and so forth, down to a specific (instantiated!) screw. Systems at once contain and are contained, like Russian dolls. The fundamentals of our car example become engine blocks, fuel pumps, and generators as well as *their* aggregates, such as power trains and electrical subsystems. Figuring out how to cleanly break them into reusable subsystems is a central chore of design.

Physical manufacturing is implicitly OO. General Motors runs an economy of scale by reusing one engine and its minor variants in many vehicles. Engineers focus on the performance specifications of the engine when using or modifying it. They do not consider how it is screwed together. Those details are below their engineering level of abstraction. (“Put a 32-valve 4-liter, 290 horsepower, aluminum-block engine in the power train” or “Put the high-end power train in the SC400 coupe”). The details are encapsulated and subjected to information hiding.

Software development has yet to reach this stage, but it apparently must pass through the OO door to get there. No other current framework for abstracting software scales so smoothly from simple, single, small components to large aggregate concepts. No other current framework works so well to hide unnecessary detail. Sure, you can achieve encapsulation and information hiding by walling off variables through a naming convention and accessing them via a preordained set of functions. You do that now and you call it a file. But this is a time-consuming way of building a class.

Why not let an OO language handle the details and an OO methodology reflect this in notation and process?

Ideally, you will build a set of fundamental classes, general, specific to your business, and specific to your industry if possible. From these classes, which are analogous to subsystems in heavy manufacturing, you will blend, derive, trim, and supplement methods and variables to produce the classes required by your applications. This class inventory is your source for the objects you use to build current and future applications. They are the screws, levers, wedges, and gears of the software world.

At a higher level, you will use messages to stitch together sets of these classes into the conceptual equivalents of engines and compressors, and from there create power trains, cooling systems, and electrical systems. Your designs will become as reusable as your code. Higher levels of abstraction are possible. You can model vehicle fleets and transit systems, instituting reuse along wide dimensions. When you begin moving toward these goals, the economies of scale beckon. That is the promise of OO.

*The newbie asked, "Master, what is the true nature of objects?" The master flamed the newbie, shouting, "Go away" or words to that effect. As the novice climbed out of a pool of coffee the novice experienced satori and went off to form a new school of methodology.*

Richard J. Botting, Department of Computer Science, California State University