OBJECT ORIENTED PROGRAMMING

Carl Erickson
Atomic Object, LLC

TABLE OF CONTENTS

1. MOTIVATION FOR OO PROGRAMMING

1.1 OO DIDN'T COME OUT OF THE BLUE…

OO has strong historical roots in other paradigms and practices. It came about to address problems commonly grouped together as the "software crisis."

Applied improperly, or by people without the skills, knowledge, and experience, it doesn't solve any problems, and might even make things worse. It can be an important piece of the solution, but isn't a guarantee or a silver bullet.

1.2 COMPLEXITY

Software is inherently complex because

- we attempt to solve problems in complex domains
- we are forced by the size of the problem to work in teams
- software is incredibly malleable building material
- discrete systems are prone to unpredictable behavior
- software systems consist of many pieces, many of which communicate

This complexity has led to many problems with large software projects. The term "software crisis" was first used at a Nato conference in 1968. Can we call something that old a crisis? The "software crisis" manifests itself in

- cost overruns
- user dissatisfaction with the final product
- buggy software
- brittle software

Some factors that impact on and reflect complexity in software:

- The number of names (variables, functions, etc) that are visible
- Constraints on the time-sequence of operations (real-time constraints)
- Memory management (garbage collection and address spaces)
- Concurrency
- Event driven user interfaces

How do humans cope with complexity in everyday life?

Abstraction

Humans deal with complexity by abstracting details away.

Examples:

- Driving a car doesn't require knowledge of internal combustion engine; sufficient to think of a car as simple transport.

- Stereotypes are negative examples of abstraction.

To be useful, an abstraction (model) must be smaller than what it represents. (road map vs photographs of terrain vs physical model).

Exercise 1

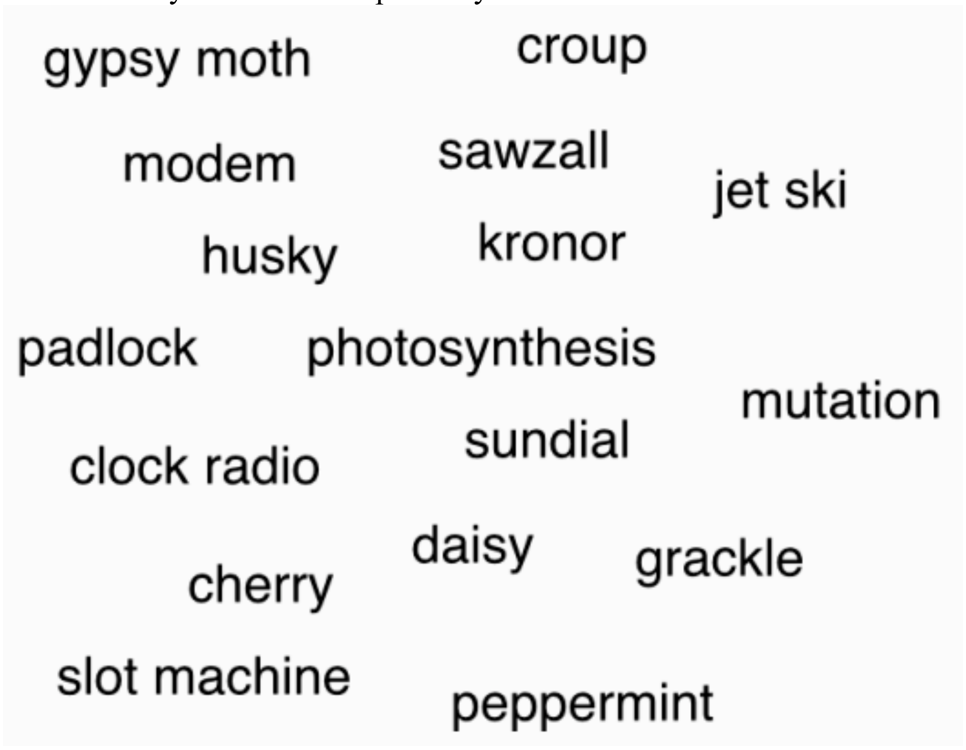
Memorize as many numbers from the following sequence as you can. I'll show them for 30 seconds. Now write them down.

1759376099873462875933451089427849701120765934

- How many did you remember?
- How many could you remember with unlimited amounts of time?

Exercise 2[1]

How many of these concepts can you memorize in 30 seconds?



gypsy moth
croup
modem
sawzall
jet ski
husky
kronor
padlock
photosynthesis
mutation
clock radio
sundial
cherry
daisy
grackle
slot machine
peppermint

Exercise 3

Write down as many of the following telephone numbers as you can…

- Home:
- Office:
- Boss:
- Co-worker:
- Parents:
- Spouse's office:
- Fax:

[1] Miller (Psychological Review, vol 63(2)) "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information"

- Friend1:
- Friend2:
- Local Pizza:

By abstracting the details of the numbers away and grouping them into a new concept (telephone number) we have increased our information handling capacity by nearly an order of magnitude. Working with abstractions lets us handle more information, but we're still limited by Miller's observation. What if you have more than 7 things to juggle in your head simultaneously?

Hierarchy

A common strategy: form a hierarchy to classify and order our abstractions.
Examples: military, large companies, Linaen classification system

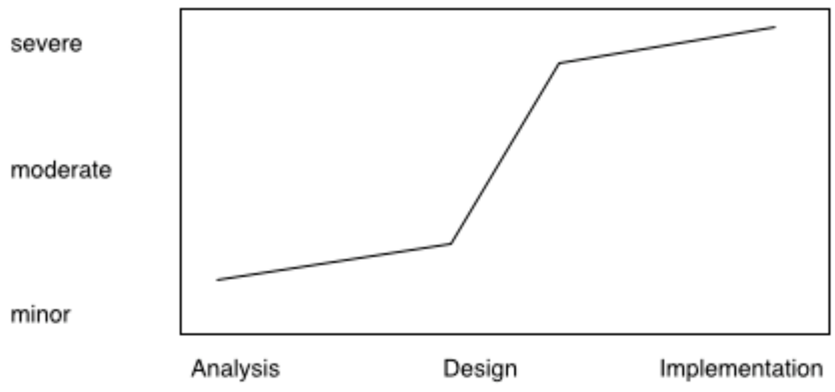
Decomposition

Divide and conquer is a handy skill for many thorny life problems. One reason we want to compose a system from small pieces, rather than build a large monolithic system, because the former can be made more reliable. Failure in one part, if properly designed, won't cause failure of the whole. This depends on the issue of coupling.
Tightly coupled systems of many parts are actually less reliable. If each part has a probability of being implemented correctly of p, and there are N parts, then the chance of of the system working is pN. When N is large, then only if p is very close to 1 will the system operate.
We can beat this grim view of a system composed of many parts by properly decomposing and decoupling.
Another reason is that we can divide up the work more easily.

Postpone obligation

Delaying decisions that bind you in the future for as long as possible preserves flexibility.

Deciding too quickly makes change more likely. The cost of change depends on when it occurs. Consider the traditional waterfall development model, and the cost of change during the lifecycle of a project.



1.3 OO TECHNOLOGY

Abstractions

Nothing unique about forming abstractions, but in OO this is a main focus of activity and organization.

We can take advantage of the natural human tendency to anthropomorphism.

We'll call our abstractions objects.

Hierarchy

We'll put our abstractions into a hierarchy to keep them organized and minimize redundancy.

This can be overrated in OO.

Decomposition

Whether we do our decomposition from a procedural, or algorithmic, point of view or from an OO point of view, the idea is the same: divide and conquer, avoid thinking of too much at once, use a hierarchy to focus our efforts.

In algorithmic decomp, we think in terms of breaking the process down into progressively finer steps. The steps, or the algorithm are the focus.

The problem with an algorithmic or top-down design, is that if we make the wrong top-level decisions, we end up having to do all sorts of ugly things down at the leaves of the decomposition tree to get the system to work. The killer is that it is hard to judge or test what are good decompositions at the topmost level when we know the least about the problem.

In OO decomp, we think in terms of identifying active, autonomous agents which are responsible for doing the work, and which communicate with each other to get the overall problem solved.

Advantages of OO decomposition

- smaller systems through re-use of existing components
- evolution from smaller, working models is inherent
- natural way to "divide and conquer" the large state spaces we face

Postponing commitment

In OO, our implementation decisions are easier to postpone since they aren't visible.

Analysis and Design: more work up-front pays off in the long run.

- Design for genericity – danger of taking too far.
- Problem of knowing enough of the requirements, tendency to change.

Revolution vs. evolution

Object-Oriented technology is both an evolution and a revolution

As evolution it is the logical descendant of HLL, procedures, libraries, structured programming, and abstract data types.

The revolution of OO technology is on a personal basis. People who are proficient in a structured programming world have learned to think in terms of algorithmic decomposition. This was hard to learn and is even harder to unlearn.

Computing as simulation

The primary difference between OT and structured HLLs is the fidelity of the abstraction to the real world. Fortran forces you into working with abstractions that are computer-language oriented (real, array, do/while). OT lets you describe your problem in terms of the problem space, in other words, OT is a modeling and simulation tool, whereas traditional HLL are simply descriptiors of algorithms.

Where does C sit in this regard? structures but no behavior.

The origins of OO programming are found in languages built for simulation.

Why OO technology now?

30 years old, why only now ubiquitous? More pressure on business to compete (globalization, need for greater productivity, flexibility, innovation, decentralization, empowered users)]

Best current means of dealing with complexity (there will be something else someday).

1.4 WORKING COMPLEX SYSTEMS

Booch identifies five common characteristics to all complex systems:

1. There is some hierarchy to the system. *A minute ago we viewed hierarchy as something we did to cope with complexity. This view says that a complex system is always hierarchic. Perhaps people can only see complex systems in this way.*
2. The primitive components of a system depend on your point of view. *One man's primitive is another complexity.*
3. Components are more tightly coupled internally than they are externally. *The components form clusters independent of other clusters.*
4. There are common patterns of simple components which give rise to complex behavior. *The complexity arises from the interaction or composition of components, not from the complexity of the components themselves.*
5. Complex systems which work evolved from simple systems which worked. *Or stated more strongly: complex systems built from scratch don't work, and can not be made to work.*

2. THE OBJECT ORIENTED PARADIGM

2.1 METAPHORS

- Hard boiled eggs
- Little black boxes
- Widget factories

All are necessarily incomplete, imperfect. The best might actually be a person. This makes anthropomorphizing very easy. This metaphor requires a multi-threaded OO model – interesting, but a bit more complicated to start with.

2.2 A SIMPLE EXAMPLE

(from Bruce Webster, Pitfalls of OO Development)

Suppose you bought a nifty device at Radio Shack, a Weather Clock. This device tells you the time, temp, pressure and humidity. Looking at your Radio Shack purchase you would see four gauges telling you the four pieces of data. You wouldn't know what's inside the device, or how the numbers being displayed were determined. The implementation of getting this data is encapsulated. All you see (and all you want/need to see) is the interface. Someday RS may change one of their gauges to be more accurate, or cheaper. They wouldn't have to change the interface, and the user would never know a change took place.

What if you want two weather clocks, one for the living room and one for the kitchen? Just buy another instance of the nifty RS device – own two of them. They are distinct, show different data, reside in different places, have different identities, but they are two of the same thing, a Weather Clock. Just because you own two of them doesn't mean that RS had to re-invent a new Weather Clock. They just ran another on off the factory for you. Since you already owned one of them, using the new one in the kitchen is easy – the interface is identical. *Is the implementation the same?*

What if you go back to RS and buy a Digital Watch. Suppose RS to save money uses the same time keeping mechanism and interface in both the watch and the weather clock. We can classify these devices by recognizing what makes them similar: they are both types of Time Pieces. The Digital Watch only keeps time. The Weather Clock does some more stuff, but it too keeps and displays time. Do you think that Radio Shack likes the idea of re-designing the time keeping mechanism and time-showing mechanism for these two products? No, so they re-use that part. We do this in OO by forming an inheritance hierarchy. You may have heard the term "isa" for this hierarchy, since we can say that WeatherClock isa TimePiece. (Note that in this case it might be better to say WeatherClock is-at-least-a TimePiece.)

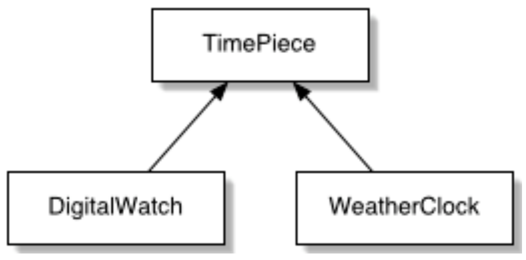


Figure: inheritance hierarchy for TimePiece, DigitalWatch, WeatherClock

What's in TimePiece?
What's in DigitalWatch?
What's in WeatherClock?
Would anybody buy a TimePiece product?

Suppose you have a serial interface to the devices. You want to get the current time from them. In OO speak you send a message. *Does the message need to be different?* No, why should it be, when it is asking for the same information?

Suppose RS sells a barometer and a humidity gauge. Wouldn't it be a drag to have to reinvent the wheel and have two barometers, one for standalone use and one to embed in the Weather Clock? *Can we avoid this with inheritance?* Nope, we need another means: composition. We can embed or include or compose one object in another.

2.3 OBJECT-ORIENTED PROGRAMMING

Programs are organized as collections of cooperative, dynamic ***objects****, each of which is an* ***instance*** *of some* ***class****. The classes may be organized into a* ***hierarchy*** *formed from* ***inheritance*** *relationships.*

To be OO, an implementation/language, must use/have these fundamental features.
OO programs may be written in non-OO languages, though it is usually very cumbersome to do so.

2. 4 OBJECT-ORIENTED DESIGN

A means and a notation to applying an object-oriented decomposition resulting in logical and physical, static and dynamic models of the system being created.
The logical model is comprised of the classes and objects.
The physical model is comprised of the set of modules and processes.

2. 5 OBJECT-ORIENTED ANALYSIS

A method of analysis which describes the problem domain in terms of classes and objects.
OOA feeds OOD which then can be implemented with OOP. In practice, these separate activities are usually done in an iterative fashion, with more analysis and less implementation up front, then more design, and finally more concentration on implementation. In other words, OO doesn't magically make waterfall work any better than it does in structured design.

3. VISUALIZING PROGRAM EXECUTION

This demonstration uses a very simple ATM system to contrast the execution of a structure and an OO program. The ATM only lets a validated user do a single withdrawl or deposit, then returns the card. The "code" is written in a sort of pseudo Java/C++. The "design" of the program leaves many things unexplained and weird. The intent is to give a more intuitive feeling for how an OO program looks in execution.

The topology of a structure program is inherently different than the topology of an OO program. Modules are nested into a calling tree, as shown below:

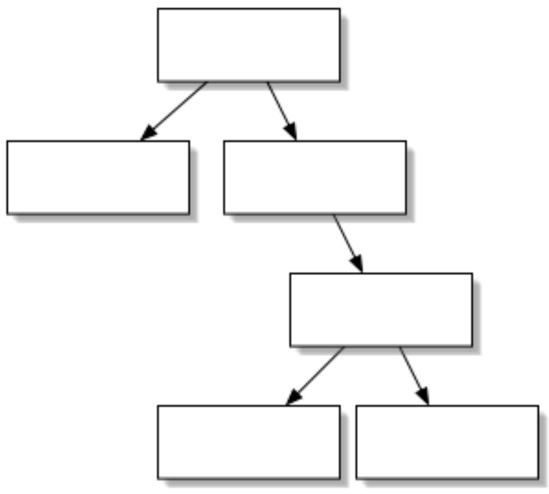


Figure: The topology of a structure program is a tree of modules.

By contrast, the topology of an OO program is inherently less ordered. Objects cluster together by those which communicate most frequently. Inter-cluster messages exist, but are less common than intra-cluster messages. Notice how much like a computer network this topology looks.

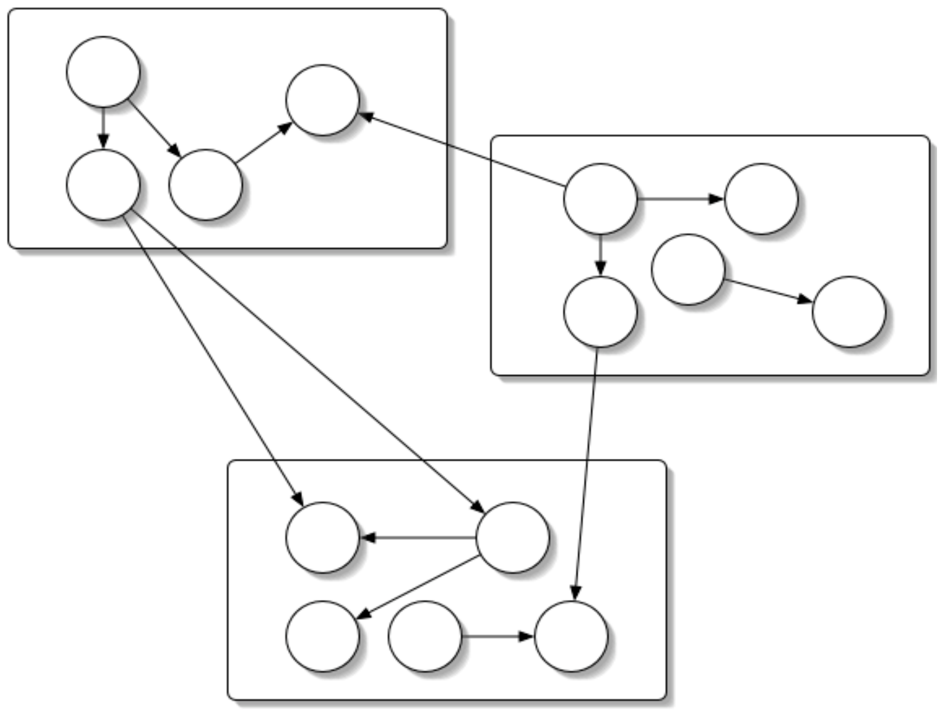


Figure: The topology of an OO program is communicating clusters of more tightly coupled objects.

Materials: blank paper, heavy marker pen

3.1 BASELINE PROCEDURAL VISUALIZATION FOR COMPARISON

Acting out

- global data structures – desks or chairs
- functions – students
- main function – instructor
- koosh ball – thread of execution (program counter)

1. Stand students in a line facing classroom (code layed out linearly in memory).
2. Label functions with pieces of paper students hold.
3. Stand yourself at the end of the line, with a whiteboard nearby.
4. Begin execution in main() showing the code for main() on the board.
5. Pass the koosh ball along to each function as they execute.

```
int main(){
  while( 1 ){
    display(GREETING_MSG);
      card = block_for_card_input();
      display(PROMPT_FOR_PIN);
      pin = block_for_pin_input();
      if( verify_account(card, pin) == OK ) {
        display(main_menu);
        action = get_activity_selection();
      switch( action ){
        case DEPOSIT:    deposit();
          break;
        case WITHDRAWL:    withdrawl();
          break;
      }
      return_card();
    else {
      display(AUTH_FAILED_MSG);
      eject_card();
    }
  }
}
```

Notice:

- how relatively important the main() method is; lots of intelligence
- how the main method has transaction logic in it.
- global data is used by functions to avoid unwieldy passing of arguments.
- the order of execution and use of machine/program is determined by the way the code is written.
- several functions (get_pin, get_activity_selection) use the same hardware device (keypad).
- functions are the units of modularity, organization, and encapsulation within the program.
- global data is used by functions to avoid unwieldy passing of arguments.

3.2 OBJECT ORIENTED

Acting out

- Put five empty chairs in a scattered, random sort of pattern. This is the object space.
- Start with empty chairs and main() method.
- Call students to chairs as objects are instantiated and give them their object name on a piece of paper.

```
int main(){
  ATM atm;
  atm.init();
  atm.run();
}

class ActiveDevice {
  public abstract void run();
}
class ATM subclasses ActiveDevice {
// ivars
  private CardReader cardReader;
  private Display display;
  private Printer printer;
  private KeyPad keyPad;
   private SecurityGuard securityGuard;
// constructor (initialization)
  public ATM() {
    display = new Display();
    keyPad = new KeyPad();
    securityGuard = new SecurityGuard();
    cardReader =
    new CardReader(display, keyPad, securityGuard);
    printer = new Printer();
  }
// methods
  public void init(){
    securityGuard.wakeup();
  }
    public void run(){
    Customer customer;
    Transaction transaction;
    while( 1 ){
      display.show(greetingMsg);
      customer = cardReader.waitForCustomer();
      if (customer.isValid()) {
        transaction = new Transaction(customer);
        transaction.start();
      } else {
        display.show(invalidCustomerMsg);
        cardReader.ejectCard();
      }
    }
  }
}

class CardReader {
  private Display display;
  private KeyPad keyPad;
  private SecurityGuard securityGuard;
// constructor
  public CardReader(Display d, KeyPad k, SecurityGuard g) {
    display = d;
    keyPad = k;
    securityGuard = g;
  }

  public Customer waitForCustomer() {
    PIN pin;
    Card card;
    Customer customer = new Customer();
    card = blockForCardInput();
    display.show(promptForPINMsg);
    pin = blockForPINInput();
    customer.setCredentials(card, pin);
    if (securityGuard.validateCustomer(customer)) {
      customer.setValid(true);
    }
    return customer;
  }
 }
```

Notice:

- how relatively unimportant the main() method is.
- how the main method doesn't have any transaction logic in it.
- how some devices are dumb (display) and some are smart (transaction).
- how the cardReader is notified of the display and keyPad device objects.

The main program is usually minimal in an OO design. Booch quotes Meyer as saying, "Real systems have no top", meaning that large complex systems are more naturally characterized as a collection of many services. As an example of that, here is the main program for a NEXTSTEP application (from the simplest to the most complex):

```
void main(int argc, char *argv[]){
  [Application new];
  if ([NXApp loadNibSection:"HRMan.nib"
  owner:NXApp withNames:NO])
  [NXApp run];
  [NXApp free];
  exit(0);
}
```

Booch claims that the OO paradigm is applicable to the widest variety of problems. A point of confusion when moving from a procedure-oriented to an object-oriented mindset is that the algorithms don't go away. This begs the question, "so what's the difference?"

Think about the difference as this: The algorithms still exist (after all, the answer has to be computed somewhere) but they are encapsulated into an object which represents the thing in the problem domain most closely associated with that algorithm.

In a way, you can look at the individual methods of an object as procedural (with the exception that they can/will message other objects) and the overall organization of the program as OO. Work happens in an OO system when objects ***message*** each other, rather than when procedures are invoked.

4. NAMING CONVENTIONS

A standard naming scheme for classes, objects, instance variables, and methods is important. Here are two alternatives:

Naming Scheme 1

Class names: concatenated words each starting with upper case.

```
Account, BankAccount, CashDispenser, SortedIntegerQueue
```

Objects, ivars, methods: concatenated words, first word all lower case, subsequent words starting with upper case.

- balance, shareBalance, count, quantityOfFives
- list, nodeList, account, newAcct
- deposit, balance, objectAt, dispenseMoney

Naming Scheme 2

Class names: concatenated words each starting with upper case.

```
Account, BankAccount, CashDispenser, SortedIntegerQueue
```

Objects: lower case separated by underscores.

```
list, node_list, account, new_acct
```

Ivars: lower case separated by underscores.

```
balance, share_balance, count, quantity_of_fives
```

Methods: concatenated words, first word all lower case, subsequent words starting with upper case,

```
deposit, balance, objectAt, dispenseMoney
```

Hungarian Notation

Everything has a tag that identifies it. This tag is appended to the name, so, for example, an ivar named **`height`** of type float might be called **`height_i_f`**. Similarly, every class ends in the capital letter “C”. This scheme forces you into the solution space (programming language) and distracts you from the problem space.

Names are vitally important, for the same reason they are important in non-OO languages, but also because of the anthropomorphic nature of OO programming.

It is common in OO languages to name things with the name of the class and a definite or indefinite article (**`Control, aController; View, theView`**). These names are fine when something more appropriate can’t be found (**`List, employees`**).

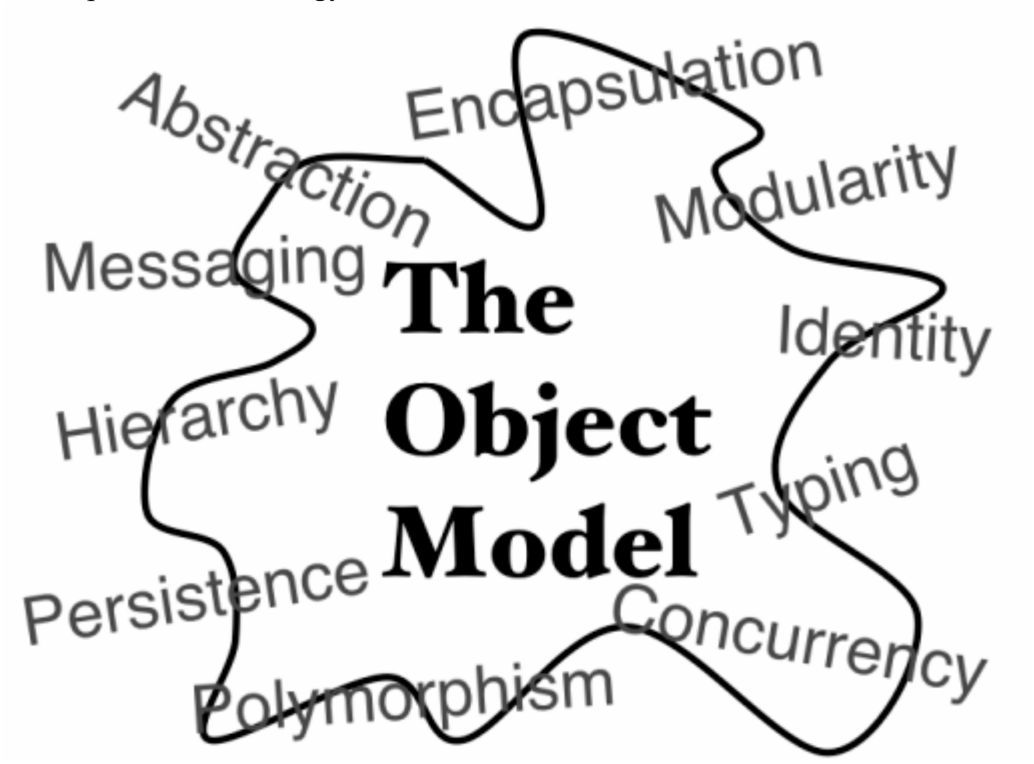
5. THE OBJECT MODEL

The terminology of OO is still evolving; each book and language has a slightly different set of terms. Start a terminology cheat sheet. I'll let you know what should be on it as we go.

The object model has many facets. Here are the ones you'll eventually have to understand to be mature OO developers. Note that these aren't "OO" concepts. There is however a way of thinking about and understanding them in an OO context.

> *Abstraction, Encapsulation, Identity, Modularity, Hierarchy, Typing, Concurrency, Persistence*

This is initially a scary, rather monstrous task. We'll take them iteratively, in keeping with our OO development methodology.



6. ABSTRACTION AND IDENTITY

6.1 ABSTRACTION

An abstraction is a simplified description of a system which captures the essential elements of that system (from the perspective of the needs of the modeler) while suppressing all other elements. The boundary of the abstraction must be well-defined so that other abstractions may rely on it. This cooperation between abstractions relies on the contract of responsibilities that an abstraction provides. This relationship is sometimes called *client* and *server.* The ***protocol*** of a server is the set of services it provides to clients and the order in which they may be invoked.

Classes

In OO, we represent our abstractions mostly as classes. Booch says,

"A class is a set of objects that share a common structure and a common behavior."

I don't like this definition, since it reverses the chicken/egg relationship between object and class. So, you have a bunch of similar objects? Then they are a class. But how did you get the objects to begin with? From a class definition, of course.

In OOA we will typically recognize the need for an active object from an analysis of the requirement specification. From that active object we can say immediately that we'll need a class. In practice you can't stop your brain from thinking of the two simultaneously. A class consists of two parts, an interface, and an implementation.

INTERFACE

This is the outside view of the class. The part visible by everybody (or at least any object who has the name of an object from this class).

Most OO languages also other interfaces to a class, for example a special one for subclasses, or for other classes in the package.

IMPLEMENTATION

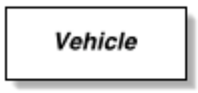
This is the actual code that implements the behavior of the class. Objects are asked to do things in messages which include the name of one of the member functions.

What happens in here can't effect clients of objects of this class, as long as the interface doesn't change. The instance variables that define what an object may know are part of the protected or private part of the interface. This encapsulates this information, allowing changes in a class without effect on its clients.

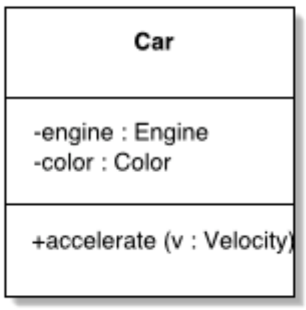
UML REPRESENTATION

Classes are usually found in class diagrams. They may have very little or a lot of detail in their representation.

Each class rectangle can have a name, attributes, and operations compartment.



Abstract class Vehicle.



Class Car has ivars engine, color (private -), and a method accelerate (public +), which takes a Velocity as parameter.

EXAMPLE – TEMPERATURE SENSOR

Booch uses the example of a hydroponics greenhouse to illustrate the elements of the object model. Plants grow in a nutrient solution; all aspects of the greenhouse (temperature, humidity, nutrient content, light, pH, etc) must be carefully controlled. All of these conditions must be monitored and controlled to maximize the yield of the farm.

Since we need to control the various factors, it is clearly important to know what they are. If we had a requirement specification for this hydroponic farm, we would see the word “sensor” in many places.

Consider a sensor that measures the temperature at some location. A temperature is a number with a certain range, and known to a certain precision, given in a certain system of units.

A location is an identifiable, unique place on the farm.

What must a temperature sensor do for us? The requirement spec might look something like this:

- “Sensors for temperature, pressure, and humidity will be located throughout the farm.”
- “Temperature sensors will have a unique location, be able to return the current temperature they are sensing, and be able to calibrate themselves to an actual temperature.”

What data does our TempSensor class need to know?

- the current temperature
- where it lives (unique location)

What behavior does our TempSensor class need?

- provide the temperature
- calibrate itself

These actions are the contract established by our temperature sensor class for its’ clients.

How does this look in C++? First cut:

```
class TempSensor {
  void calibrate(float actualTemp);
  float currentTemp();
  float temp;
  int location;
};
```

This simple class should illustrate what we’re studying in this aspect of the OO model, mainly the two aspects of implementation and interface that our abstractions support. Here’s a second cut:

```
class TempSensor {
public:
  void calibrate(float actualTemp);
  float currentTemp() const;
private:
  float temp;
  int location;
};
```

The **private** part of the class is the set of variables and methods that the TempSensor class uses to perform its responsibilities (i.e. to be and act like a temperature sensor). These are the implementation of the abstraction, and are a private matter for the class.

The `const` keyword means that the function `currentTemp` won't modify the variables of the object. They can be applied to `const` objects, whereas other functions (without the const) cannot.

Should I even be talking about the type of variables we use to store the data at this point? No, since that's a private implementation detail.

All we have here is a definition for an abstraction that is a temperature sensor.

What other things might we know about the temperature sensors we employ in the greenhouse?

name of the sensor manufacturer, model number

We abstracted this detail away, even though we probably will know it, since it isn't germane to the problem. *When might this be germane?*

So far we also only have a passive, abstract thing (the class definition). This doesn't do any work. To do some work, we need to ***instantiate*** an actual, active, dynamic, living, breathing, TempSensor object:

```
TempSensor gh1Sensor(1);
TempSensor gh3Sensor (2);
float t = gh1Sensor.currentTemp();
```

Now we have two TS that we can work with. The last line shows us asking the sensor in GreenHouse 1 what is the current temperature; it is an example of a message. *Note the four parts.*

How did our temperature sensor objects get their initial values? How did location get set? We passed an integer, but who consumed it?

Third cut:

```
class TempSensor {
  public:
    TempSensor(int);
    void calibrate(float actualTemp);
    float currentTemp() const;
  private:
    float temp;
    int location;
};
```

We added the definition of a special function (or method) to our class so that we can initialize objects of that class. This method is invoked automatically everytime an object of the TempSensor class is created.

What if we need to make our sensor smarter, more autonomous? Then it would be nice to have it be able to tell someone when things got too hot.

What's the classic, procedural means of doing this? invoke a call back function, generate an interrupt, poll the sensor, etc.

OO way: send a message. All work is done this way. To send a message our object obviously needs the identity of who it should notify, and maybe some arguments (i.e. the temperature, or the sensors location), since these describe the event that is happening.

```
class ActiveTempSensor {
  public:
    ActiveTempSensor(int, Alert* helper);
    void calibrate(float actualTemp);
    void establishSetpoint(float setpoint, float delta);
    float currentTemp() const;
  private:
    // internal details
};
```

Now our smart temp sensor can call someone when the temp set point is reached.

What exactly is `Alert* helper`? In C++ (and ANSI C), `Alert*` means "a pointer to type Alert". `helper` is the name of the object we call for help, at least in terms of this function parameter. In OO terms we should think of this as the identity of some object who will be messaged when our sensor goes over set point.

Aside: what do we know (care) about this helper object?

Suppose the helper we give to a sensor is a complicated object like a Controller. This class has lots of other responsabilities, but one of them is to know what to do when a temperature sensor sends an alert message.

We could have written the ActiveTempSensor class to use a Controller, like this:

```
ActiveTempSensor(int, Controller* helper);
```

but we don't really care what the class of the helper object is, so long as it satisfies the Alert interface:

```
class Alert {
  public:
    void alert(int location);
}
```

All we really care about it is that it will respond to our alert() message.
Consider testing the ActiveTempSensor class. If we had written ActiveTempSensor to use Controller directly, then we could not test an ActiveTempSensor without first constructing a Controller, which might be an expensive, complicated task.
By using the Alert interface, we can make a quick-and-dirty DummyController class just for testing purposes. Design for test often results in better design.
Notice that the setpoint is not included in the constructor method. This means that our sensor may never have its setpoint set. *What should it do? Should we allow this?*

EXAMPLE – PET CLASS

A Pet should know its name and age. Every Pet should know how to tell you their name and age, and set their name and age. Each Pet must keep track of its location and posture (sitting, standing, laying, etc). Every Pet should be able to "come", but it is expected that different types of Pets implement this behavior in their own way.

```
class Pet {
  public:
    Pet(String);
    void setName(String);
    void setAge(int);
    String getName() const;
    int getAge() const;
    void come();
  private:
    String name;
    int age;
    int location;
    int posture;
};
```

Now that you've got your Pet class definition, you put minimal, dummy code into the implementation, and you can begin instantiating and testing your classes.

```
#include "Pet.h"
#include <libc.h>
Pet::Pet(String n)
{name = n;}
void Pet::setName(String n)
{name = n;}
void Pet::setAge(int a)
{age = a;}
String Pet::getName() const
{return name;}
int Pet::getAge() const
{return age;}
void Pet::come()
{}
```

6.2 IDENTITY

Identity is the property of a thing which distinguishes it from all other objects. Humans have id numbers, fingerprints, DNA profiles. All these are representations of the fact that we are each unique and identifiable.

This element of the object model can be confused with ***state***. State is the set of values that an object encapsulates. Two objects may have identical state, and yet are still separate, distinct, identifiable objects.

Objects in an OO system have distinct identity. It is part and parcel of what makes them an object.

Objects

An analogy: an object is to its class, as a toaster is to the toaster factory.
Objects are living, breathing, dynamic entities. They get work done. We create them, often at runtime.

Classes are (mostly) static. They define the capabilities or behavior of an object. They (mostly) don't get work done.

The idea of being able to recognize objects and to think about them indepenent of seeing them develops in humans at a very early age.

Objects were first introduced in Simula.

Objects in an OOD may be abstract representations of real things, or useful abstractions for solving our problem which don't have a real-world analog. Booch defines them very generally as anything with a "crisply defined boundary".

There are three major things an object has that we'll learn to look for (discover) and, as needed, invent: State, Behavior, Identity.

What an object knows (state) and what it can do (behavior) are determined by its classification. Identity is required so that we may talk to an object without confusing it with another object, and so that we may have more than one object of a given class alive at the same time.

A synonym for object is ***instance***.

STATE

This is what an object knows. It is a set of properties, or variables (usually static) which can take different values at different times in the objects life (dynamic).

Since objects have state, their reaction to messages can vary depending on when the messages are sent, and what order they are sent in.

For example, an Employee object should refuse to generate a paycheck for itself if the Employee object has a state variable the value of which makes that Employee inactive.

State variables of an object are sometimes referred to as instance variables, or ivars.

The class definition in C++ defines the state of an object, much like a struct does, but allows for a finer degree of control (encapsulation).

BEHAVIOR

This is the set of actions that an object knows how to take. Your Pet object knows a set of tricks (behavior) that you can ask it (message it) to perform.

C++ calls the "tricks" that an object knows ***member functions***. A better name is ***methods***.
In general, we can group the methods of a class into five categories:

- Modifier Change the state of the object
- Selector Accesses, but does not change the state
- Iterator Operates across all parts of an object
- Constructor Creates an object and initializes its state
- Destructor Frees the state and destroys the object cleanly

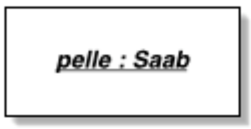
The responsibilities that an object has to its clients are key to defining the class that an object belongs to. When we do our OOA/D we'll often recognize a need for some action and that need will create some state and behavior in a class. Objects of that class can then fulfill this need for us.

Objects can be viewed as finite state automata, or simply, as machines.

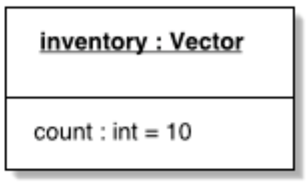
Objects can be either passive, or active. Passive objects do nothing on their own; they only execute when they are messaged by some other object. Active objects have their own thread of control and may take action while other objects are working. This rather complicated business is considered in the Concurrency aspect of the object model.

UML REPRESENTATION

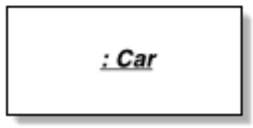
Objects are shown in rectangles, with their names underlined, and their type or class after their name. Anonymous objects have no name.



The object pelle is of type Saab.



The inventory object has a count of 10.



This anonymous object is of type Car.

EXAMPLE: C++ GUI FRAMEWORK

(modified from Booch, page 92)

Confusing the whole issue of identity is the use of pointers in high level languages. Pointers confuse the difference between an object itself and the name of the object.

Imagine some sort of GUI environment where objects that can be displayed on the screen must be described. One of the attributes of an object is obviously its location. A location is a point in a 2d Cartesian space. How do we represent points? We could use a class, so that we could have point objects, but a point really doesn't do all that much, so instead we use a simpler structure.

Booch Rule of Thumb Structure versus Class

If something is just a record of other primitive data types, and has no really interesting behavior of its own, make it a structure.

```
struct Point {
  int x;
  int y;
  Point() : x(0), y(0) {}
  Point(int xValue, int yValue) : x(xValue), y(yValue)
  {}
};
```

The Point structure lets us allocate points with initial values, or if not provided, will initialize the point to (0,0).

Now we need to represent a class for anything that can be displayed. We'll anticipate the need for many different types of things that are to be displayed, and make this class the top level of this particular part of the class hierarchy. DisplayItem is an abstract class designed to be subclassed to be useful.

```
class DisplayItem {
  public:
    DisplayItem();
    DisplayItem(const Point& location);

    void draw();
    void erase();

    void select();
    void unselect();

    void move(const Point& location);
    int isSelected() const;
    int isUnder(const Point& location) const;
    Point location() const;
};
```

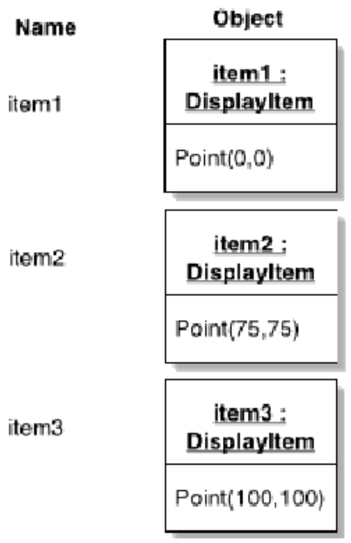
Reminders of what the C++ means…

Two constructors, one of which takes a Point which is the location of the item.
C++ lets us dynamically allocate classes and structures with the new operator. The new operator, together with a class constructor, is like malloc in C.

Now look carefully at these declarations:

```
DisplayItem item1;
DisplayItem* item2 = new DisplayItem(Point(75, 75));
DisplayItem* item3 = new DisplayItem(Point(100, 100));
DisplayItem* item4 = NULL;
```

Here is a picture representing the code fragment above.



The first line creates an instance (object) of class DisplayItem. The name of this object is item1. No matter what values this DislayItem takes on over the course of its lifetime, its name will always be item1. Do we know where it is stored? The initial location for item1 is determined by the default constructor (0,0). How much memory is allocated by the compiler?

The second and third lines allocate two things each. The first is a pointer. A pointer is simply a primitive data type that can be used to hold the address of another data element (structure, object, int, float, etc).

The second thing these lines of code do is to dynamically allocate (from the heap) enough memory to hold a DisplayItem object. Do we know the name of this object? We can’t know the name, because they are dynamically allocated (which is why in the picture the names are lines). The object doesn't exist until run-time. Clearly we can't refer, by name, in our program to an object that doesn't exist until run time. Instead we talk to or refer to these objects indirectly, by the names of the pointers that refer to them.

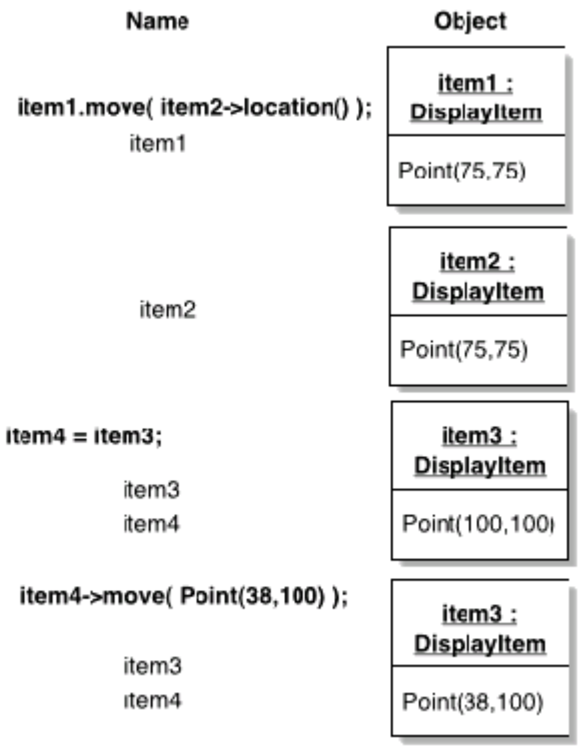
The fourth line creates a pointer, but doesn't allocate an object for it to point to.

What we did above was to create four names (1 object name and 3 pointer names), three DisplayItem objects, and three pointers.

Now we'll operate on our data space. We need to know a little more C++ syntax first. The operator "`.`" is the C structure access operator. In C++ it is used as the messaging syntax. The form is "`object.method()`". The operator "`->`" is the C structure access operator when you must refer to a structure indirectly, through its pointer. In C++, we use "`->`" to message an object through its pointer. The form is "`object_pointer->`**`method()`**".

Re-draw the above picture showing the changes the following code fragment makes.

```
item1.move( item2->location() );
item4 = item3;
item4->move( Point(38,100) );
```



The first thing to notice is that item1 and item2 (or rather, the object that item2 refers to) have the same state. Having the same state doesn't change their identity; they are still unique, singular things.

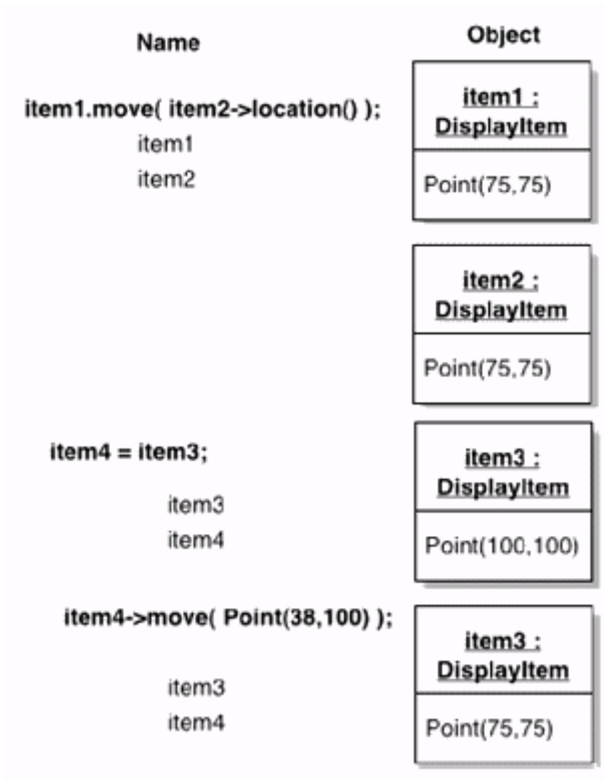
The next thing we've done is to have item4 refer to the same object as item3. We have created an alias for this third DisplayItem object. We can now operate on this object with either the name item3, or item4. The third line changed the state of the object using its new name.

Being able to create name aliases for the same object can cause many problems. One of them is the problem of a dangling pointer. Consider what happens if item3 is destroyed. Now item4 will point to nothing.

I asked you if we knew the address, or memory location of item1 when we declared it. We didn't need to know it, since we had its name. There is an operator in C/C++ that lets us find out the address of any data element. & does this.

Re-draw the picture showing the changes the following code fragment makes.

```
item1.move(Point(0,0));
item2 = &item1;
item4->move( item2->location());
```



First we put item1 back to the origin.

Then we changed the object that the pointer item2 refers to. Item1 now has a new name ("item2").

Finally, we move item4 to where item2 is. Is that the point (75,75), as the image might imply? No, item2 doesn't refer to that object any longer. The move results in item4 going to the origin.

We've also lost track of the DisplayItem object that item2 originally referred to. This object, and the memory it lives in, is permanently lost to us. There is no way to get it back. This is known as a memory leak.

7. OBJECT ORIENTED MESSAGING

Messaging is how work gets done in an OO system. Understanding messaging is a key part of being able to visualize how an OO program actually executes, and the relationship between the abstractions (objects) in an OO program.

A message has four parts:

- identity of the recipient object
- code to be executed by the recipient
- arguments for the code
- return value

The code to be executed by an object when it is sent a message is known variously as a ***method***, or a ***function.*** Method is preferable.

Messaging an object may cause it to change state. The Elevator object will move when it is otherwise idle, and a user presses a floor button. But note carefully that what an object does when messaged depends on the state it was in when messaged. Thus the state of an object is a consequence of the history of messages it has received.

Objects messaging each other are what gets work done in an OO system. Rumbaugh says that two objects which can message each other have a "link" between them.

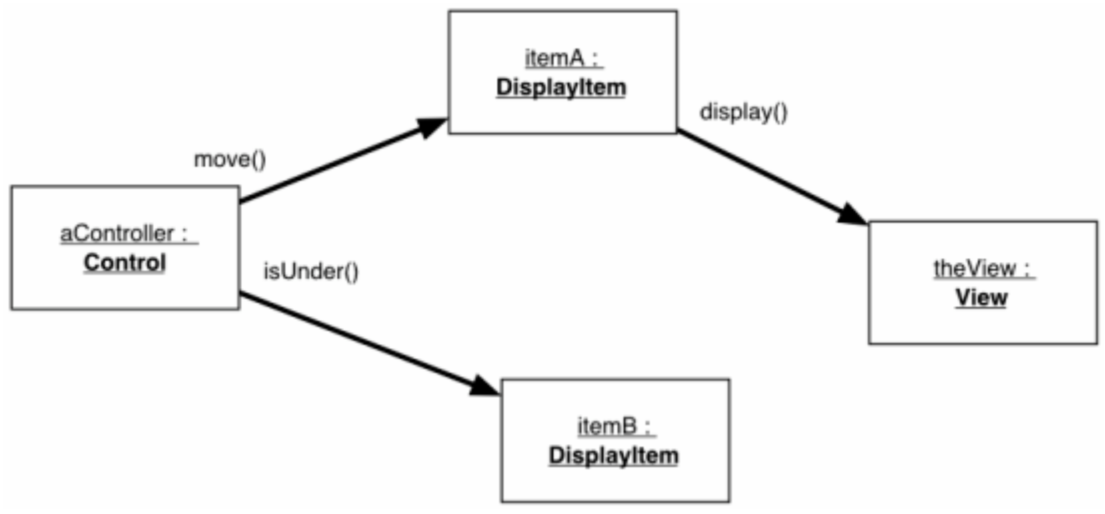
Remember, an object can't message another object without first having its name. So, how does an object find out the name of an object it wants to message?

Name is global
Name is part of the class of the messaging object
Name is passed as a parameter to some operation
Name is locally declared within object's method
Name is provided as return value from message

How objects relate to each other depends on how they collaborate.

Actor Operates on other objects, but is never operated upon.
Server Is operated on by other objects, but never operates on other objects.
Agent Both and Actor and a Server.

A simple example to illustrate these various object roles is shown below



- aController represents an actor object
  since it provides no service; is just asks other objects to do things
- theView represents a server object
  since it only takes message, doesn't generate them
- itemA represents represents an agent
  since it provides a service to aController, and messages aView

8. ENCAPSULATION AND MODULARITY

8.1 ENCAPSULATION

We encapsulate the details of a class inside its private part so that it is impossible (and not just suggested) for any of the class's clients to know about or depend upon these details.

The ability to change the representation of an abstraction (data structures, algorithms) without disturbing any of its clients is the essential benefit of encapsulation.

Aside – should I say the "class's clients" or the "object's clients"?

Why encapsulate and hide?

- You can delay the resolution of the details until after the design.
- You can change it later without a ripple effect.
- It keeps your code modular.

How we do this separation is with two separate pieces for each class, an ***implementation*** and an ***interface***.

8.2 MODULARITY

Modularity is closely tied with encapsulation; think of modularity as a way of mapping encapsulated abstractions into real, physical modules.

The C/C++ convention is to create two files for each class: a header file (.h suffix) for the class interface, and an implementation file (.c, .cp, .cpp, .C suffix) for the code of the class.

Booch gives two goals for defining modules. Make a module cohesive (shared data structures, similar classes) with an interface that allows for minimal inter-module coupling.

Other considerations: team work, security, documentation.

Important to remember that the decisions concerning modularity are more physical issues, whereas the encapsulation of abstractions are logical issues of design.

It is possible to "over modularize". This can increase documentation costs and make it hard to find information.

9. OBJECT ORIENTED HIERARCHY

We group our abstractions (classes) into a hierarchy to keep them organized (inheritance structure).

9.1 IS-A HIERARCHY

The "is a" hierarchy defines classes like this: "a dog is a mammal", "an apple growing plan is a fruit growing plan which is a growing plan". This is the test to apply if you are not sure whether there is a classical inheritance relationship between classes.

```
class FruitGrowingPlan isa GrowingPlan {…}

class AppleGrowingPlan isa FruitGrowingPlan {…}
```

As we form the class hierarchy we push the common state and behavior between lower level classes into the upper level classes. This allows for the lower level classes (which are usually "discovered" first in an OOA) to be built on top of the higher level classes, thus making them smaller and more readily understandable.

This activity (pushing commonality up the inheritance hierarchy) makes for a generalization/specialization hierarchy. The classes at the top are more general (or abstract) and the classes at the bottom are more specific (or concrete). It is usually the concrete classes that we instantiate objects from.

Abstract classes are simply that; abstract means of organizing shared information.

Forming the hierarchy eliminates redundant code and organizes our abstractions into something easier to understand.

The "isa" hierarchy is a compile-time hierarchy, in that it involves classes extending other classes.

9.2 PART-OF HIERARCHY

The other type of hierarchy is the "part of" hierarchy we find in our designs. This is hierarchy through aggregation.

Objects would be overly large if it was not possible to aggregate them. For example, a Graph object quite naturally contains a List of nodes in the graph. Without this ability, the List class would have to be duplicated in Graph.

The "hasa" hierarchy is another means of code organization.

The "hasa" hierarchy is a run-time hierarchy, in that it involves objects containing other objects.

10. OBJECT ORIENTED TYPING

The concept of type is more or less important in a language, depending on whether the language is strongly or weakly (or not at all) typed.

In strongly typed languages, the compiler prevents you from mixing different kinds of data together. This is limiting, but can be very helpful when you want to avoid putting a string, say, into the floating point value for the altitude of an airplane's automatic pilot.

In OO languuages, type is a synonym for class, at least as a first cut. In fact, class and type are quite distinct. A more nuanced understanding of type is one of the things that distinguishes a shallow from a deep understanding of OO design and programming.

Type defines the properties and behaviors shared by all objects of the same type (class).

OO languages can run the range of un-typed, weakly typed, or strongly typed.

The advantage of strongly typed languages is that the compiler can detect when an object is being sent a message to which it does not respond. This can prevent run-time errors. The other advantages of strong typing are:

- earlier detection of errors speeds development
- better optimized code from compiler
- no run-time penalty for determining type

The disadvantages of strong typing are:

- loss of some flexibility
- more difficult to define collections of heterogeneous objects

C++ maintains strong class typing. An object may only be sent a message containing a method that is defined in the object's class, or in any of the classes that this object inherits from. The compiler checks that you obey this.

Objective C support strong typing, but it also allows for weak typing. A special type, id, can be used for object handles. An id can be a handle to any class of object, thus it has no typing information, and the compiler can't check if you are sending proper messages. This is done instead at run-time.

10.1 TYPE VS CLASS

One of the best pieces of advice that can be given for OOD is to code to types, not classes. This follows from the general advice of postponing obligation.
By writing to an interface, the decision about which classes will actually be used is postponed. It becomes impossible to write in dependencies on any particular class. It becomes possible in the future to plug new and unanticipated objects into the same code structure. Flexibility is preserved.

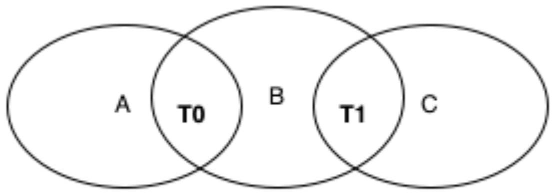
A class is two things: implementation and interface. The interface part is the only part that affects the type. The implementation part is just messy detail.

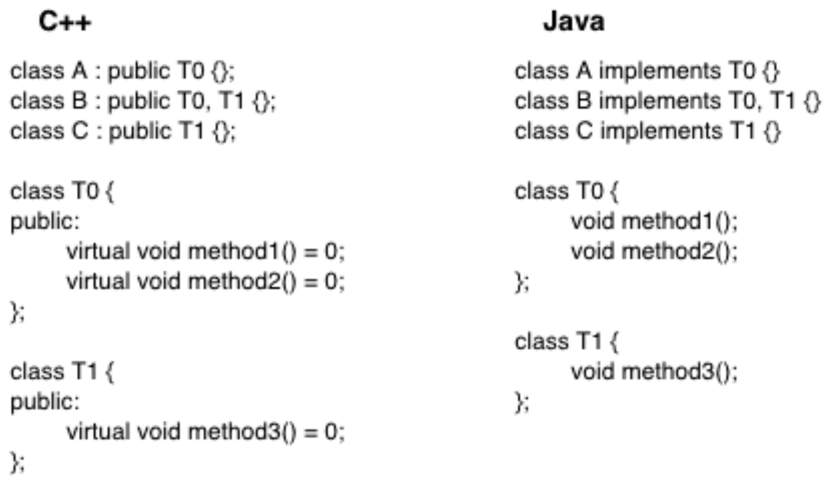
We can code carefully to a type using only the interface portion of a given class. We still have the problem that in the future we might need to add a new class of the same type, and our code is written with specific classes – time for rewriting. And if we don't code very carefully we might find that some class-specific things have crept into our code. In other words, we didn't stick strictly to the interface.

The better alternative is to be able to separately define an interface as its own full-fledged, supported, OO language abstraction. A class that had only interface (and no implementation) would be a type. It would be impossible for the code we write to this type to have any reliance on the specific implementation found in a class. And, as a bonanza, we have drastically decreased the amount of code that has to be re-written in the future when we want to add a new class, since we've coded to the interface/type, and we can simply plug objects of the new classes that are of the same type into our code.

The Venn diagram below shows three classes and two interfaces. It demonstrates the idea of substitutability as defined on type (interface), not class. A and B are both substitutable for type T0. B and C are substitutable for type T1.

Note that class A may be more than T0, in other words, it can have other aspects of its interface than what T0 specifies.





C++

```
class A : public T0 {};
class B : public T0, T1 {};
class C : public T1 {};

class T0 {
public:
        virtual void method1() = 0;
        virtual void method2() = 0;
};

class T1 {
public:
        virtual void method3() = 0;
};
```

Java

```
class A implements T0 {}
class B implements T0, T1 {}
class C implements T1 {}

class T0 {
        void method1();
        void method2();
};

class T1 {
        void method3();
};
```

OO languages have different ways of representing type.

Interface in Java

Java lets you define an interface. For example:

```
interface Device {
  boolean init();
  void start();
  void stop();
}
```

Interfaces don't get work done, objects do. Objects need to have a class definition. You can't have "interface objects". Instead, a Java class implements an interface, meaning it has implementations for the methods in the interface.

```
class Sensor implements Device {
  boolean init() { ... }
  void start() { ... }
  void stop() { ... }
}
```

Java lets us program to an interface by declaring variables of the type. Consider this fragment of client code:

```
Device dev = new Sensor();
dev.init();
dev.start();
```

Now consider what happens someday when we want to plug a new type of device into our client. The only line of code that changes is the line where we specifically instantiate the Sensor object. All the other code is written to use the Device interface variable `dev`.

Interface in C++

In C++, we don't have an interface keyword, but we can achieve the same thing of a type which is pure interface. Unfortunately we do this by using the class abstraction:

```
class Device {
  virtual boolean init() = 0;
  virtual void start() = 0;
  virtual void stop() = 0;
};
```

The odd looking syntax (initializing a method signature to zero?!) is what is known as a "pure virtual method". A pure virtual method has no implementation. A class with one or more pure virtual methods is abstract – you can't create objects from it. Subclasses of a class with a pure virtual method are required to implement that method, or be abstract themselves.

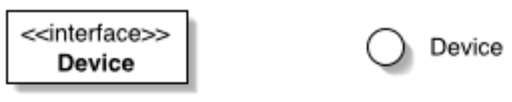
```
class Sensor : public Device {
  boolean init();
  void start();
  void stop();
};
boolean Sensor::init() {...}
void Sensor::start() {...}
void Sensor::stop() {...}
```

C++ lets us program to an interface by declaring variables of the abstract class type. Consider this fragment of client code:

```
Device dev = new Sensor();
dev.init();
dev.start();
```

UML Representation

There are two means of showing an interface in UML. The first is via a full stereotype (with all possible compartments and elaborations). The second is with a simple circle and the interface name.



Two alternatives for showing an interface.

11. OBJECT ORIENTED CONCURRENCY AND PERSISTENCE

11.1 CONCURRENCY

OO helps us with the complex issues of concurrency in the same way that is helps us with our "programming in the large" problems. Encapsulation, Abstraction, Inheritance all make it easier to isolate and re-use the solutions we come up with to our concurrency problems.

OO is certainly not a magic bullet for concurrency.

Booch distinguishes between *active* and *inactive* objects. Active objects are those with an independent thread of control that can execute in parallel with theads in other objects.

Look back at the ActiveTempSensor class. Walk through a messaging scenario:

> *Function Creates TS -> TS Construction -> Function Messages TS -> TS Method Execution*

We see how a TS can be messaged to do some work, but how can a temp sensor call anybody? *Isn't there just a single thread of control? How does control transfer to the temp sensor so that it can message someone? Is the temp sensor a process?*

The answer to these questions depends on the implementation. You might have a separate process for each temp sensor. You might have an event triggered by a hardware interrupt. You might have a multi-threaded app, with a separate thread for each sensor. You might have temp events come into the app with all the other events (mouse, keyboard, port events, etc). The answer depends on the OS environment.

The three sorts of solutions that are possible:

- a process abstraction from the OS
- a thread abstraction, within a single process
- interrupts

Synchronization is the big issue when you have concurrency.

Synchronization

Some terms help:

- Program: a collection of bytes on disk (passive, inactive)
- Process: living breathing entity which gets work done by executing a program
- Process space: where objects live; an address space

Processes with a single thread of control are easy to understand and follow. You can view their execution as follows:

```
while( process is running ){
  wait for stimuli(
    user input, timer expiring,
    network or kernel message, interrupt)
  execute message/method path to completion
}
```

The quiescent program starts execution within a method of the object receiving the "wakeup" event. This method in turn sends messages to other objects in the process space. They in turn may message other

objects. Each of these messages eventually returns (with our without a value). Finally the first method to execute returns and the process is quiescent again.
Suppose you have a process with multiple threads of control. Now it is possible that an object is messaged at more or less the same time by two different objects. We need to worry about this concurrency. What does the messaged object promise the messaging objects? Three possibilities for synchronization:

- sequential – semantics of object are guaranteed only for one thread at a time (aka not thread safe)
- guarded – semantics of the object are guaranteed only if the threads coordinate with each other (i.e. synchronization is outside the object)
- synchronous – object guarantees its semantics by doing its own guarding

11.2 PERSISTENCE

There is a range of time over which objects exist. From the most transitory (local variables within a method) to the most stable (objects that live forever). If an object is to have life beyond that of the process it runs in, some sort of db is needed.

The large investments in relational databases means that OO programmers often have to use an RDBMS to achieve a level of object persistence. This leads to the need for an object to know how to archive to and read itself from an RDBMS.

The up-and-coming alternative is an OO database. For storing/retrieving complex objects an OO db can be much faster than an RDB.

Some vendors of OODB: VERSANT, GemStone, ObjectStore, ONTOS. The RDB vendors (INGRES, Oracle) have announced intentions of extending or migrating to OO dbs.

The relational db model promotes a clean separation between the data and the logic or procedures for acting on the data. This helps you maintain stability and flexibility as requirements change. Since objects are all about wrapping data and functionality together, do we lose this advantage in an OODB? This trend can be seen in client/server database apps where triggers or stored procedures are part of the schema.

12. OBJECT ORIENTED DEVELOPMENT PROCESS

Booch identifies a macro process that organizes an entire team's activities over the course (weeks, months) of development. These are the same sort of good software engineering principles and management techniques that are already practiced.
The micro process is more interesting to us, since it it tailored to OO development. The micro process is iterative. It also emphasizes the interface and relationships between classes and objects over the implementation of specific methods in a class. We iterate over the four steps of the micro process as we proceed through the broader phases of Analysis, Design and Implementation.

12.1 THE MICRO PROCESS

Identify Classes & Objects

This is the OOA stage. We define the problem, identify what is and what is not part of our problem domain (abstraction).

The output of this stage is a data dictionary containing all the key elements of the problem. Data dictionaries have advantages over just maintaining a list of things that will are in the problem domain. These are:

- common and consistent vocabulary
- supports project browing (good for new team members)
- gives a global view of the project

Some sort of technique (CRC cards, drawing tool, CASE tool) is helpful here to represent each thing in the data dictionary.

Class & Object Semantics

We proceed with our list of things from the first step and develop the semantics (what our abstractions encapsulate and what they can do) of our classes and objects.

From this step we end up with interface files for our classes (.h files for C++).

This stage is where the "two heads are better than one" truism is seen most clearly. Debate sharpens the semantics of classes and improves them. The best ideas survive, the weak ones are reformulated.

Debates about class semantics and names can become heated. My theory is that since OO involves significant anthropomorphism, designers become more attached to their points-of-view.

Class & Object Relationships

We formalize in our diagrams the relationships between classes and objects.

Associations are between classes. We look at groups of related classes and work on the diagrams to represent how they associate with each other.

Collaborations are more the concern of the design phase and deal with object diagrams and how objects collaborate.

Implementing Classes & Objects

Initially (in analysis) we do just enough of this to help show gaps and lacks with the design. Eventually the implementation becomes more and more “real”. Iterations at this phase are consistent with the idea of working complex systems evolving from working simple systems.

13. OBJECT ORIENTED ANALYSIS TECHNIQUES

13.1 BEHAVIOR ANALYSIS

Focuses on behavior, versus properties.

Consider the responsibilities and functionality that must be represented. Form classes based on similar behavior.

13.2 DOMAIN ANALYSIS

Consider what has come before. Study existing systems, interview domain experts.

Users are often domain experts (particularly when they are customers).

One danger of domain analysis is that you must avoid limiting yourself to the goals or paradigms of existing systems. Users often know their job too well in the context of an existing tool. They can discourage innovation.

Example: you interview the main user of a human resources application program. You find out exactly how she uses the tool, where she spends the most time, how she thinks about the tasks she performs. But what you learn is often filtered through the user interface of the tool being used. The trick is to get behind the existing system and answer the fundamental questions about what needs to be done.

13.3 USE-CASE ANALYSIS (AKA STORYBOARDING)

Users and developers construct common scenarios of operation and “walk through” these scenarios looking for knowledge that must be represented and actions that must be performed. While doing this you answer the question, “who shall know/do this?”, which leads to the formation of classes and objects.

The problem with this technique is that scenarios overlap, so you have a lot of redundancy to wade through. It is also very difficult to know when you’ve captured all (or X%) of the needed classes/objects.

Use-case is applicable to most every technique, and a valuable way to get started on the iterative process of analysis, design and implementation.

13.4 PATTERN SCAVENGING

Looking for commonality among your class descriptions, or from previous projects. This is becoming more and more important, as experience with OO becomes more widespread, certain common patterns are emerging. Gamma, et al in Design Patterns, identify 23 common patterns in 3 broad areas (creational, structural, behavioral) which have been proven useful.

13. 5 CLASSICAL ANALYSIS

This approach follows the traditional approach to classification; recognize the entities that share common properties and form a class.
Look for the following in the problem description (three author’s suggestions):

- Things, Roles, Events, Interactions
- People, Places, Things, Organizations, Concepts, Events

- Structure, External Systems, Devices, Events remembered, Roles played, Locations, Organizational units

This is the most pragmatic of the techniques. It can help you avoid the “blank page syndrome”.

Starting from a written description of the system to be designed (a requirement specification) you analyze the English for verbs and nouns. Nouns are potential classes and objects, verbs are potential methods (to be assigned to classes).

The quality of the analysis depends a lot on the quality of the written document you start from. This technique suffers from a completeness problem.

The prime benefit of this technique is that is very algorithmic, and hence easy to apply when you are facing a “blank page.”

In reality a developer ends up using bits and pieces from all these techniques; you can readily identify fundamental classes based on commonality of knowledge (classical) and function (behavioral), you check out existing products and solutions, talk with experts in the field (domain), construct role-play scenarios (use-case), and pore over requirement specifications (English descriptions).

Tools can be helpful in this process. One such tool is the CRC card.

14. PITFALLS IN OBJECT ORIENTED ANALYSIS

Underestimating the need for and difficulty of analysis

Time and money pressures from managers, and the natural tendency of developers to "go to code" quickly, can be a bad combination. Analysis is expensive in time and money, and takes a lot of skill, both in the domain of the problem, and in the OO domain. There is much to consider about each abstraction: the static and dynamic models, the invariants, quality of abstraction, potential for re-use. There are design issues, driven partly by language choice: implementing the abstraction, life cycle issues, safe-use issues, documenting proper use.

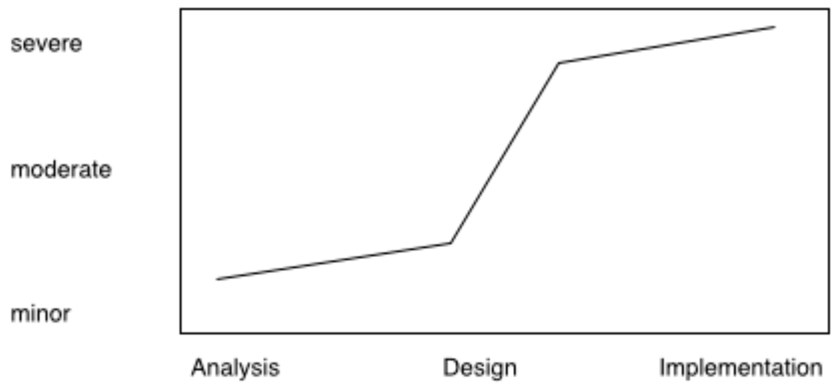
14.1 ACCURATE AND COMPLETE REQUIREMENT SPECIFICATIONS

Except for very constrained domains, re-implementations of existing functionality, or trivial applications, the process of defining requirements is almost guaranted to result in:

- incomplete specifications
- incorrect specifications

A study in 1999 by Elemer Magaziner of requirement specifications found that they are typically only 7% complete and 15% correct.

Another almost guaranteed phenomenon is that they will change over time as the project progresses. Given the cost of change over a project's lifecycle this is a huge problem and argues for alternatives to what the above pitfall seems to argue for (lots of upfront analysis).



14.2 BUILDING A TOO-GENERAL SOLUTION

Generality is good, but comes at a cost. Some balance must be met. Developers who really become OO are particularly prone to this problem. The elegant, complete, general solution to a class design becomes a goal unto itself. You've got to know when good is good enough.

14.3 THE WRONG NUMBER OF ARCHITECTS

Encapsulation and interface doesn't mean you can split your problem into pieces, give each piece to a team, then never talk across teams only to get back together and expect everything to "just work". There are aspects of class design that don't show up in the interface files, or in the documentation. Someone (or a small team of someones) has to have chief responsibility for the architecture of large projects.

14.4 RE-ARCHITECTING TOO OFTEN OR TOO LITTLE

Developers enamored with the beauty and elegance of a design may want to re-architect when something new throws their model for a loop. This is possible, and a big advantage of OO, if done properly, but can be taken too far (re-architecting for its own sake, or for the intellectual challenge). At the same time, you must occasionally face facts and re-think your architecture if you find it growing too complex, hard to understand, full of hacks, etc.

15. UML NOTATION

UML is a "method for specifying, visualizing, and documenting the artifacts of an object-oriented system under development."

UML is Booch, Objectory, and OMT combined, extended, simplified. Started in 1994.

Four goals for UML effort:

- To model systems (and not just software) using object-oriented concepts
- To establish an explicit coupling to conceptual as well as executable artifacts
- To address the issues of scale inherent in complex, mission-critical systems
- To create a method usable by both humans and machines

Targeted the modeling of concurrent, distributed systems.

Focuses on a standard language, not a standard process.

> Trying to get everyone to use the same methodology is a losing game. Different domains require different methodologies. Each developer and organization has different styles. UML started out as a methodology (the Unified Method) but evolved into something more abstract; a language for modeling.

The primary focus is developing a meta-model; second comes a standard notation to use it. The process isn't a goal, but Rational encourages an "architecture-driven, incremental, and iterative development process."

Developers using UMLdon't see much of the meta-model. Instead they create models of their designs using the notation. The meta-model is for developers of UML, and for developers of UML-aware tools. The artifacts created by a modeling effort are:

- Use case diagrams (from Objectory)
- Class diagrams (OMT)
- Behavior diagrams
    - State diagram (Booch &OMT)
    - Activity diagram
    - Sequence diagram (Booch interaction diagrams)
    - Collaboration diagram (Booch object-message diagrams)Process diagrams (new idea)
- Implementation diagrams
    - Component diagram (Booch Module diagrams)
    - Deployment diagram (Booch Network and Process diagrams)

A notation should reduce the cognitive load on the designer by being easy to use and consistent. This frees the designer for more important things, like concept development. A consistent notation also has the benefit that you can share your work with others. UML's notation is as widely agreed on as any.

There is a subset of UML notation that is useful in many applications; the details don't have to be mastered.

Large systems are too complicated for any single view or diagram to fully describe. UML uses the following diagrams to describe such systems:

- Use case diagrams
- Class diagrams

- Object diagrams
- Sequence diagram
- Statechart diagrams
- Collaboration diagrams
- Component diagrams
- Deployment diagrams

We use different diagrams at different stages of the project. In the analysis phase we use Object, Class and Statechart diagrams. In the design phase we supplement these with Component and Deployment diagrams, as well as the dynamic behavior diagrams, like Sequence and Statechart.

Since OO development tends to be iterative, the diagrams are not "write once and forget" drawings, they evolve with the development.

15.1 CLASS DIAGRAMS

Classes are shown as blobs with dashed line edges. They have a class name and the most important (usually not complete) set of attributes and behavior of the class.

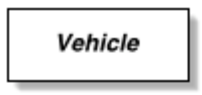
Attributes are described as follows:

- A atribute name only
- : C class name only
- A : C attribute of class
- A : C = E attribute of class with default value

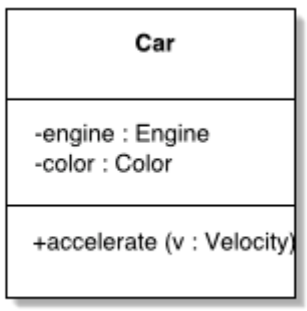
Operations are the behavior of the class and are show as:

- N() operation name only (preferred)
- R N(args) operation with return class type and arguments

A marker {abstract}after the name of the class means it is abstract.



Abstract class Vehicle.



Class Car has ivars engine, color (private -), and a method accelerate (public +), which takes a Velocity as parameter.

Class relationships

Associations are labeled with nouns, indicating the nature of the relationship. They can also be adorned with their cardinality at the target end:

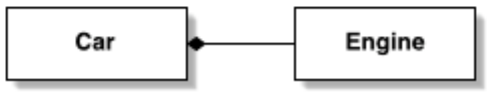
- 1 exactly one
- N unlimited
- 0..N zero or more
- 1..N one or more

• X..Y range from X to Y

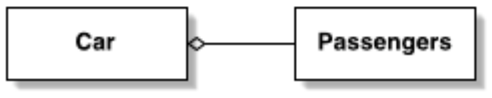
Inheritance relationships have arrows that point toward the superclass.

Hasa (composition) relationships are whole/part where the black diamond is on the whole or composite class.

Aggregation relationships use a clear diamond.



Composition: every car has an engine.



Aggregation: cars may have passengers, they come and go

Class categories

Logical groupings of classes. No semantic meaning. Not supported by most OO languages.

Advanced class notation

Parameterized classes, metaclasses, class utilities (free member functions in C++), nesting, export control, properties (static, virtual, friend in C++), physical containment, roles and keys, constraints, notes.

Specification

Textual representation of the graphical symbol.

Could just be the header files, with comments.

Each class has at least this much:

- Name identifier
- Definition textual
- Responsibilities textual
  Attributes list
- Operations list
- Constraints list

Specifications also include textual description of the advanced class definition features above.

15.2 STATECHART DIAGRAMS

Classes have Statecharts if they exhibit interesting and important state-dependent behavior.

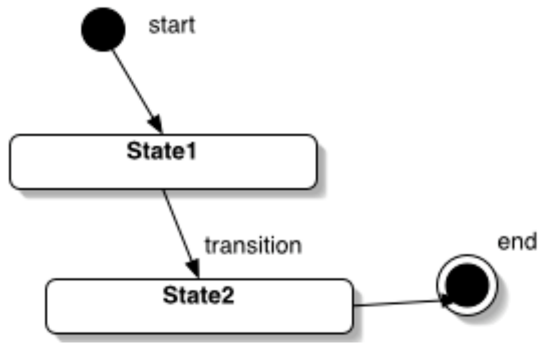
The state of an object is the collection of the attributes of the object with their current (usually dynamic) values. This state changes over time as the object sends and receives messages. The state of an object effects the way it reacts to messages.

When an object changes state it goes through a state transition. This is shown in STDs with an arrow. Transitions are unique, so that, given an object in a certain state, if a certain circumstance or event occurs, there is only one transition that can occur. There may however, be many transitions entering/leaving a single state.

Events can be viewed in several ways. Some options include looking at events as objects, messages, or the start/end of an activity. Consistency within a diagram is important.

STDs must have exactly one start state (with an unlabled transition from a filled circle to enter it).

STDs may have stop states, with dot within a circle, or they may not.



Advanced features

Actions and conditional transitions, nested states, history, orthogonal states.

`<example from Booch, page 311>`

15.3 OBJECT DIAGRAMS

Object diagrams are a snapshot in time showing the relationships between objects at some phase of the programs life. Object diagrams have a scope in which they live, since they represent actual, live objects, and not just abstract classes.

How are ojbect diagrams different than class diagrams? How many elements will they have, compared to class diagrams?

An object name can be of the form:

- A object name
- : C class name
- A : C object and class name

15.4 RELATIONSHIPS

A link can exist between two objects only if their corresponding classes have a relationship.
Links are shown by lines between objects.

15.5 SEQUENCE DIAGRAMS

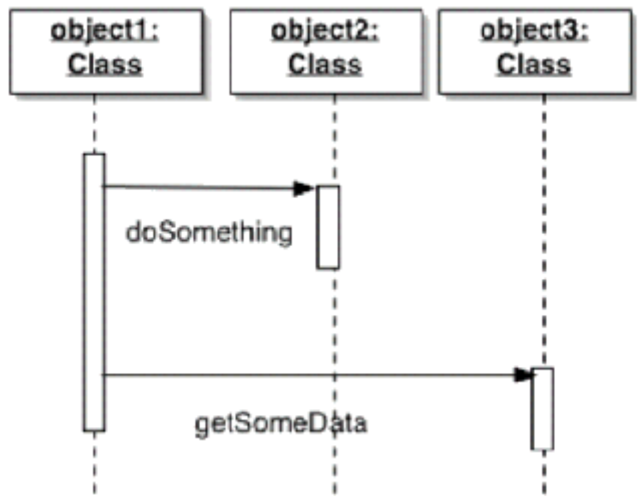
A sequence diagram shows how objects exchange messages over time to achieve some particular end. Time moves vertically down the diagram.

Objects are drawn horizontally across the diagram.

Dashed lines (lifelines) below objects show the boundary of the object for messaging, and the life time of the object.

Activations are rectangles showing when an object is active.

Horizontal arrows show messages starting at the client and ending at the supplier object.



Advanced concepts

Scripts can be written beside the diagram to enhance the description of what is taking place.

Loops can be represented.

Focus of control can be used to show which objects initiate a string of messages. These vertical boxes show what object control flow returns to.

`<example from Booch, page 451>`

15.6 COMPONENT DIAGRAMS

Mapping our class definitions to files is the concern of component diagrams.

We need to represent: interface files (.h), implementation files (.cpp), main program file (.cpp), and subsystems (collections of related classes).

The relationship we show between the file icons is a compilation dependency. In C and C++ this is done with the #include C pre-processor directive. An arrow from one file to another indicates the target of the arrow is included in the originator.

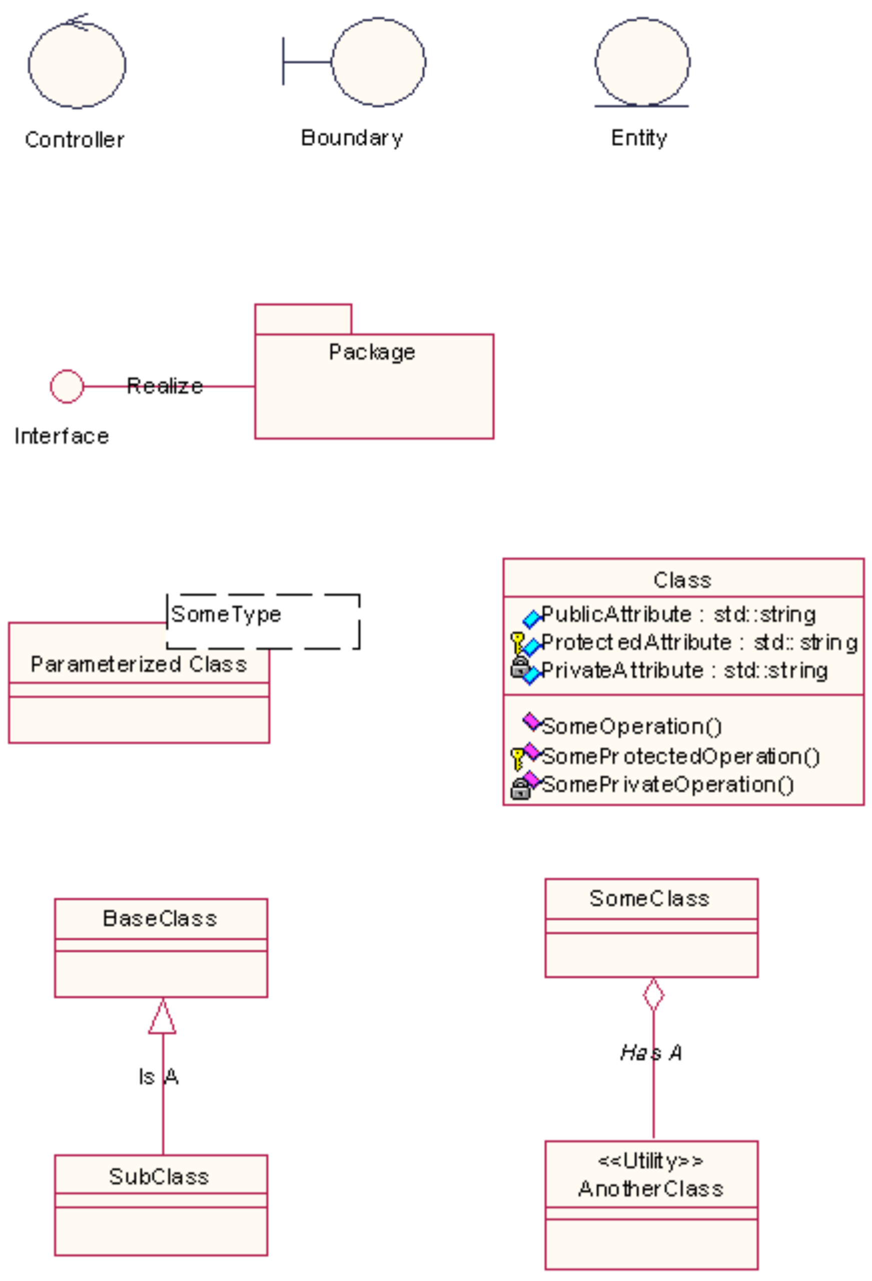
15.7 DEPLOYMENT DIAGRAMS

Mapping which shows processes and the processors they are assigned to. Also shows devices in the system and how devices communicate.

15.8 RATIONAL ROSE UML EXTENSIONS

These symbols are used by the Rational Rose modeling tool.

Rational UML Types



16. CRC CARDS

Class / Responsibilities / Collaborators

CRC cards are a simple representation of the classes in an OO system. They are often a good way to get started when facing a new project.
Physically the cards are 3×5 or 4×6 index cards.

You write the name of the class at the top of the card. On the left half of the card you write the responsibilities (which cover the class attributes via "knowing…") and on the rights side you write the collaborations (other classes) for this class.

Scenarios, use-cases, role playing, etc, can then be used to find new behavior, and make sure it is listed as a responsibility on some card.

The advantages of using low tech cards are:

- simple, cheap, editable
- spatial groupings for exploration of relationships: class hierarchy, object messaging
- very practical, good at playing "what if" games with class definitions.
- encourages seeing the separation and boundaries of abstractions.
- easy to role play with.

17. OBJECT ORIENTED CLASS RELATIONSHIPS

We're going to look at the way we define the relationships between classes so we know how objects can relate to each other.

Consider the differences and similarities between the classes of the following objects: pets, cats, siberian huskies, poodles, tails, owners.

We see the following relationships:

- a cat is a kind of pet
- a dog is a (different) kind of pet
- siberian huskies and poodles are both kinds of dogs
- a tail is a part of both dogs and cats
- owners feed pets, pets please owners
- owners use money to register pets

Depending on which book you read, or which tool you use, there are at least the following sorts of relationships between classes:

- aggregation
- inheritance
- using
- association
- instantiation

The first two are the most essential to OO design. Interestingly they are also sometimes equally valid design alternatives to solve a given problem.

Inheritance is the most distinctively OO form of relationship, but not necessarily the most important, and certainly not always the right choice.

Class relationships are of course visible in source code, but can also be defined graphically with a modeling language such as UML.

18. OBJECT ORIENTED AGGREGATION

18.1 AGGREGATION – A.K.A. COMPOSITION, AGGREGATION.

Different authors use different terms for the idea of containment. UML has a very specific use (and different symbols) of the terms aggregation and composition, though this is not necessarily consistent with OO authors. For the record, UML aggregation is “weak containment”, usually implemented with pointers, and the symbol is an open diamond. Composition is “strong containment”, implemented by value, and the symbol is a filled diamond.

An aggregate object is one which contains other objects. For example, an Airplane class would contain Engine, Wing, Tail, Crew objects. Sometimes the class aggregation corresponds to physical containment in the model (like the airplane). But sometimes it is more abstract (e.g. Club and Members).

Whereas the test for inheritance is “isa”, the test for aggregation is to see if there is a whole/part relationship between two classes (“hasa”). A synonym for this is “part-of”.

Being an aggregated object of some class means that objects of the containing class can message you to do work. Aggregation provides an easy way for two objects to know about each other (and hence message each other). *Can the contained object message the container object? Is the relationship symmetrical?*

Should the aggregation be done by value or by reference?

By value means that the lifetimes of the objects are exactly the same; they are born and die at the same time; they are inseparable during their lifetimes. Aggregation by value cannot be cyclic: one object is the whole, one the part.

By reference de-couples the lifetimes of the two objects. The whole may not have a part, or it may have a different part at different times. Different wholes may share the same part. De-allocating the whole won’t automatically de-allocate the part. Aggregation by reference allows for the part to be aware of its whole.

Example

A List class may look like this. This list stores integers.

```
class List {
  public:
    List();
    void addToFront(int);
    Int firstElement();
    int length();
    int includes(int);
    void remove(int);
    private:
    // some data structure for
    //  storing integer elements
};
```

The List class could be quite handy for creating a new Set class, since the behavior is similar. One way to do this is to use a List object to handle the data of the Set, via aggregation.

```
class Set {
  public:
    Set();
    void add(int);
    int size();
    int includes(int);
  private:
    List data;
};
```

Now we can pretty easily implement Set in terms of the functionality available in List:

```
  int Set::size() {
  return data.length();
}
int Set::includes(int newValue) {
  return data.includes(newValue);
}
void Set::add(int newValue){
  if( ! includes(newValue) )
    data.addToFront(newValue);
    // else, no-op, since sets can only have
    //  one copy of any given value
}
```

This shows composition/aggregation by value, since the List element ivar of Set is statically declared. Every Set object always has a List object. Delete a Set, the List goes away too, with no special action required.

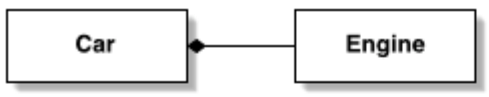
Note that you need to worry about the List ivar object being properly initialized, so the constructor to Set must do this:

```
Set::Set() : data()
  // other stuff for Set, as needed
}
```

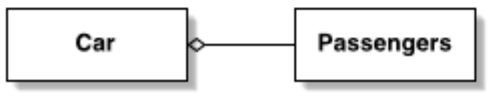
The section on Inheritance will consider a design alternative for our implementation of Set.

18.1 UML REPRESENTATION

UML distinguishes between composition and aggregation. (These are often synonymous terms.)



Composition: every car has an engine.



Aggregation: cars may have passengers, they come and go

19. OBJECT ORIENTED INHERITANCE

19.1 TERMINOLOGY

When we use inheritance, we have classes at the top of the hierarchy that are known variously as:

- Generalization
- Parent
- Superclass
- Base
- Abstract

And classes at the bottom of the hierarchy (or those which inherit from some other class) known variously as:

- Specialization
- Child
- Subclass
- Derived
- Concrete

The classes toward the top of an inheritance hierarchy tend to be abstract classes that won't have any objects instantiated from them. The bottom most classes of the hierarchy are the concrete classes from which we actually make objects that do work for us.

19.2 ADVANTAGES

- Avoid redundant code by sharing it (coupling?) between classes
  - Inheritance relationships let us avoid the redundancy of declaring the same instance variables and behavior in a group of related classes. For instance, Dogs and Cats don't both need a name variable; they can inherit it from Pet.
- Enforcing standards at a design level (interfaces)
- Programming-by-drag-and-drop (components)
- Avoid redundant code by sharing it between projects (reuse)
  - Shared code allows for rapid prototyping via reuse.
  - Shared code produces greater reliability via greater use (and hence discovery of errors).

Two views: expansion (additional state/behavior) versus contraction (more specialized)

19.3 SUBSTITUTABILITY

When objects of class X can be substituted for objects of class Y with no observable effects, we say class Y is substitutable for class X.

Why do we care about this?

Code now, for classes not yet written.

Graphics example:

1. Shape, Rectangle, Circle
2. Pointer to Shape
3. Container of Shape objects

4. Loop to send drawSelf() message
5. Later: add a new shape subclass

Why not define substitutability for type, rather than class?

- If type and class are the same (like in C++) then there is no difference.
- Separating the two has advantages, as we can program to the type and never even know the class. (remote objects example)

Statically and strongly typed languages tend to define substitutability in terms of class.

Dynamically and weakly typed languages tend to define substitutability in terms of type (or behavior).

19.4 EXAMPLE IN JAVA – PET CLASSES

Here's our Pet class in Java.

```
class Pet {
  private String name;
  private int age;
  private int location;
  private int posture;
  Pet();
  void setName(String n) { name = n; }
  void setAge(int a) { age = a; }
  String getName() { return name; }
  int getAge() { return age; }
  void come() { }
}
```

The Pet class we had from before was fine, as long as we didn't care much about the specifics of the pets we were making. This might be appropriate if the domain we're modeling doesn't require very pet-specific behavior to be captured.

Suppose now you care about behavior like "come" which is definitely pet species specific. We can still use the baseline functionality of our Pet class for dogs, since after all, pets are dogs. But our Dog class needs to do something that cats, anoles, and fish don't do, or do it in a special, dog-specific manner. And now the code for our simple Dog class:

```
class Dog extends Pet {
  Dog() { }
  void come() {
  System.out.println("I am coming, master");
}
  void rollOver() {
    System.out.println("rolling over");
  }
  void sitUp() {
    System.out.println("sitting up");
  }
}
```

Does Java separate interface and implementation as cleanly as C++? Write code to implement the sitUp() method of the Dog class. Assume you have these constants somewhere:

```
static final int STANDING = 0;
static final int SITTING = 1;
static final int LAYING = 2;

void sitUp() {
  posture = SITTING;
}
```

Eventually you write a complicated algorithm for a dog to situp based on whether it is already standing, where it is located, how attentive it is, whether it is in a good mood, etc (in other words, capturing the behavior of a real dog). When you teach your old Dog class this new trick, nothing changes in the interface.

Not all pets implement sitUp() and rollOver() – those are dog specific tricks.

Different pets, like cats, might implement come() quite differently.

These differences and the common state and behavior, are what determines how we can use hierarchy.

19.5 EXAMPLE IN C++ – ABSTRACT CLASS FOR GRAPHICS

Remember the `DisplayItem` class? We said initially that it was an "abstract" class, designed to subclassed. It alone is too generic to be of any use as living breathing objects (it'll have other uses as we'll see later).

In C++ we can say we mean a class to be abstract, and subclassed, by marking methods "virtual". This is a confusing but essential topic when learning C++, and one that needs treatment all by itself. Fow now, if you see "virtual" on a method, you should think of that method as one in which subclasses have their own implementation.

```
class DisplayItem {
public:
  DisplayItem();
  DisplayItem(const Point&amp; location);
  virtual void draw();
  virtual void erase();
  virtual void select();
  virtual void unselect();
  virtual void move(const Point&amp; location);
  virtual int isUnder(const Point&amp; location) const;
  int isSelected() const;
  Point location() const;
};
```

Reminders of what the C++ means…

- The methods draw, erase, ... are obviously going to need to be subclass defined, since each subclass would draw itself differently, so they are declared virtual and are meant to be overriden in a subclass definition.
- The selector methods are not declared virtual because they can be defined here for all subclasses. (Other OO languages (Java, Smalltalk, Objective C) don't need a keyword to say this same thing.)

19.6 FORMS OF INHERITANCE

1. For specialization
   a) The classic or purest form of inheritance: specialization versus generalization.
   b) The subclass is a special case of the superclass.

   c) Pushing the common stuff (state + behavior) up the hierarchy.
   d) The acid test for the purest form of inheritance (i.e. subclassing) is the "is a" question.
   e) Subclasses are subtypes as well. Everything that is true of Super is true of Sub.
2. For specification
   a) When you want to constrain how subclasses are designed. You insist that subclasses meet a certain interface. This lets you count on being able to message objects of these subclasses in a uniform manner.
   b) You want to enforce an interface of some sort, but you aren't providing an implementation for that functionality; that's up to the subclass.
   c) Subclasses are subtypes. This is a special case of specialization.
3. For extension
   a) Subclasses add additional, new functionality to the base class.
   b) Subclasses are subtypes, since they leave the parent class functionality/interface in place.
4. For construction
   a) Done when the subclass is implemented in terms of the super class. Aggregation is often a more desirable alternative.
   b) The super class has desired behavior we want, but we need to add to it.
   c) This form of inheritance is in direct comparison to aggregation (object relationship).
   d) Subclasses are not subtypes. No "isa" relationship.
5. For generalization
   a) Just the opposite of the form of inheritance as specialization (the "pure" form).
   b) Subclasses generalize functionality already found in the base class by overriding methods inherited from the parent.
   c) Example: monochrom windows versus colored windows.
   d) If you control the class hierarchy, it can be better to refactor the classes and modify the base classes to support the newly desired general functionality. Often times you don't have this control, so you are forced to use inheritance this way (upside down).
6. For limitation
   a) Subclasses decrease in some way the behavior or interface inherited from the base class.
   b) Methods are eliminated in the interface.
   c) Example: Professor, Programmer, Employee
      Professor -> Programmer -> Employee

      Programmer isa Employee, adds the additional functionality of produceRealCode() as a specialization of Employee (something that Secretary, for instance, wouldn't have).

      Professor isa Employee, but Professor "turns off" the produceRealCode() method found in Programmer.
   d) To be avoided, as subclasses are not subtypes, as the parents behavior is intentionally limited in the child. Most often done when the base classes are not controlled locally.
7. For variance
   a) When you have two unrelated classes that happen to have similar functionality, or a similar interface, then you can make one (arbitrarily) the superclass of the other to share the common implementation.
      Sometimes this can be the hint you need to recognize a new abstract superclass.
   b) Java solves some of these problems via separating class and interface. Interface may be common to unrelated classes (i.e. it is orthogonal to the class hierarchy).
   c) Subclasses are not subtypes.
8. For combination

a) Similar to construction, in that a new class is built from the elements of existing classes. Multiple inheritance allows this in some languages.
   May preserve typing.
b) The first three forms cover almost all the cases we want to use inheritance for. The others are special cases, or situations in which some other design is usually better.

19.7 EXAMPLE

This is an alternative design to the List/Set problem above, which was solved with composition.

Set shares many of the same behaviors as List. Can we say that Set isa List?

- Set redefines the behavior of the add() method. This looks like inheritance for specialization. Add still works, just in a way that makes sense for Set.
- Set adds new behavior in the form of the size() method. This looks like inheritance for extension.

```
class Set : public List {
public:
  Set();
  void add(int);
  int size();
};

Set::Set() : List(){}

void Set::add(int newValue){
  if( ! includes(newValue) )
    addToFront(newValue);
}

void Set::size(){
  return length();
}
```

Note that the construction of the Set objects is a bit different. The initializer list daisy chains the constructors together.

Note the use of the List methods without any class scope operation. Since Set inherits from List, it may use those methods without any specific naming. *Can we say that methods addToFront(), remove() and includes() are part of class Set? Is this a problem? (yes and yes!) (This design flaw is fatal in the case of Set inheriting from List.)*

19.7 DISADVANTAGES (IN GENERAL)

- Increased class/abstraction coupling
  - What is more coupled than super/sub classes? Coupling is bad. So inheritance is bad?
- Performance
  - The more general a body of code (class hierarchy) is, the less likely it is the optimal perfomance solution for any given specific problem.
  - Overblown.
  - Experts versus novices, and class maturity.
- Program size
  - Using large class hierarchies makes your project bigger.
  - Dealing with the run-time dispatch of messages to objects related in an inheritance hierarchy has memory overhead.

- Understanding a large system (the yo-yo problem, class documentation)
  - Inheritance means that it is not sufficient to study the header file (declaration) of a class to completely understand its behavior. You often must look up the hierarchy and consider the inherited behavior and state from its superclasses. On a practical note, this can be a pain for documentation, since you may not find the behavior you seek in the document for the class itself. Class browsers address this problem.

19.8 AGGREGATION VS INHERITANCE

Composition means less coupling between classes. This is good. Composition is simpler. This is good.

Composition lets you specify exactly an interface for Set, independent of the interface of List. This is good.

Inheritance gives Set an interface it may or may not wish to comply to completely.

To understand Set in the inheritance example, one must look at the List class header file.

The inheritance approach is smaller. Less "wrapper" code has to be written.

With inheritance, it is possible for users of Set to intentionally use the overridden, inherited methods in lieu of those redefined in Set. So for example a client could call addToFront directly, bypassing the check done in the Set add() method.

Coupling to the clients is strong with the List inheritance, since clients may come to rely on the fact that "Set isa List". This means it would be harder to change the implementation of the data structure (a list) someday if necessary. With composition the clients are unaware that a List is used, and so it could be changed easily.

We forgo the potential advantage of using polymorphic behavior with our List/Set objects at some future date if we use composition.

Having to understand the pre-conditions and proper use of List in order to use Set is a drawback to inheritance. With composition everything that needs to be known about Set is contained in Set.

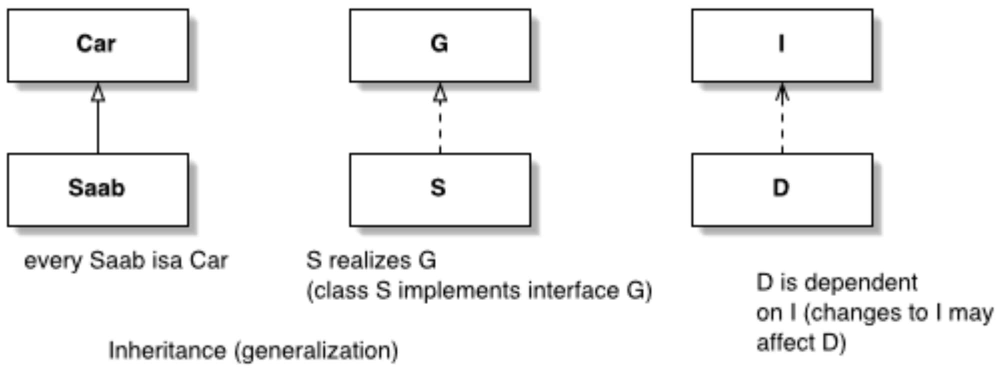
Inheritance is a little bit faster since the overhead of a message send to the aggregated List object is avoided.

Metric: use the question of substitutability to decide which technique is better.

Other authors come down quite forcefully on the side of favoring composition over inheritance.

UML REPRESENTATION

UML indicates a superclass/subclass relationship as shown below.


Car
Saab
every Saab isa Car
G
S
S realizes G
(class S implements interface G)
Inheritance (generalization)
I
D
D is dependent
on I (changes to I may
affect D)

20. OTHER OBJECT ORIENTED CLASS RELATIONSHIPS

20.1 CLASS RELATIONSHIP: USING

When a class definition uses another class as a parameter to a method, or declares an object of that class local to one of the member functions, then we say the first class is using the second.

Using is not the same as aggregation, since the class being used doesn't satisfy the whole/part question. The used class exists to help the first class.

For example, if a class method needed to keep track of a list of elements to do its works, then it could declare a List object as a local variable within the method. The List object will go away automatically when the method returns, just like any automatic (local) C variable. The class does not contain a list permanently, it simply has one method that uses a List.

Or another example: a class for calculating voltages, currents and power in an electrical circuit would need to deal with complex numbers a lot. A complex number class would be used to pass arguments to methods, hold temporary values, etc.

20.2 CLASS RELATIONSHIP: ASSOCIATION

Classes that cooperate with each other, or are associated with each other through the model of our problem, need each others names to message back and forth. This is usually implemented in C++ by putting pointers to objects in class definitions, just like Graph had a pointer to NodeList.
Association implies symmetry; both classes know each other and can message each other.

A Product class object may need to know the last sale that it was involved with. We could put a pointer to a Sale class object inside the private part of Product.

A Sale class object may need to know all of the Products involved with the Sale. We can put a pointer to a pointer (double pointer) in Sale that will let us store zero or more Products associated with a Sale.

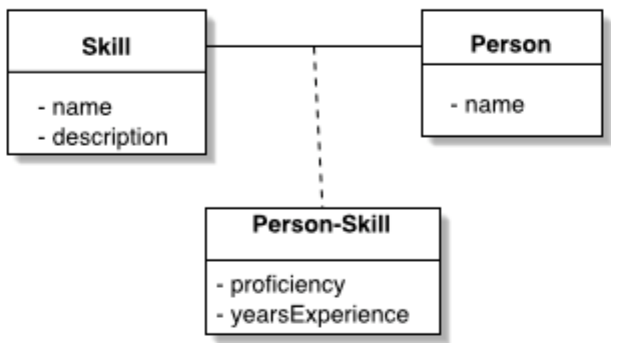
Associations have a cardinality. They can be one-to-one, one-to-many, or many-to-many. Example: Each Product is part of 1 Sale; each Sale has many Products.

Associations are the most generic of the relationships between classes.

A class may also represent an association between two other classes. For example, a Skill and a Person are associated with one another. A Person can have some number of Skills, and a given Skill may be associated with some set of Persons.

To define the relationship, a Person-Skill class can have state such as proficiency and yearsExperience which tell how good a particular Person is at a given Skill, and how many years experience they have with it.

UML Representation



21. OBJECT ORIENTED INSTANTIATION

This is the most subtle form of relationship between classes. The potential confusion for people new to OO is that instantiation is what happens to create an object, yet this is a class-to-class relationship – no objects are involved. A parameterized class cannot have objects instantiated from it unless it is first instantiated itself (hence the confusing name for this type of relationship).

Another term for this relationship is “generics”.

A generic class arises from a situation where the behavior of a class can be abstracted into something which is relevant to more than one type of state. The goal is to share the definition (in the class) of some behavior (i.e. the methods) across different data types.

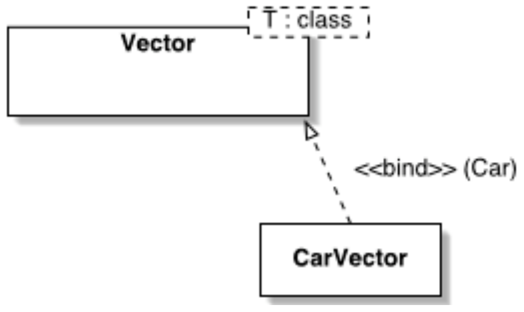
For example, a Stack class has the same fundamental behavior whether it is holding Dogs, ints, Strings, etc. We shouldn’t have to define a DogStack, IntStack, etc, as separate classes with nearly identical methods.

The common element across this “family” of potential stack classes is the data type being operated on. Let’s abstract that out and create a generic Stack class which is parameterized on the type of data it is used to hold.

One way of implementing a generic class is to write it in terms of generic pointers such as void* in C, or id type in Objective C, or Object references in Java. The drawback of this is that a Stack class would be happy to store a mixture of objects. You could put one int, a Dog, and a Person in the same stack. There is no type checking that can be done, and hence no type safety.

Some languages, such as C++, support generics in a type safe fashion.

UML Representation



22. OBJECT ORIENTED POLYMORPHISM

22.1 POLYMORPHIC: FROM THE GREEK FOR “MANY FORMS.”

When something (function argument, variable, etc) can be of more than one type.

Not limited to OO languages.

Seen in nearly every language in the form of operator overloading. For example in C, we have operators like +, -, *, etc, which operate for all primitive types. In other words we don’t have separate ”+ int”, ”+ float”, ”+ char” etc, operators for each type. The arguments to the + operator may be of a variety of types (the set of primitives). We can say that the + operator is polymorphic.

For OO languages polymorphism is tied up with substitutability. We design methods and we write client code that can operate on a set of types. What is common about these types is that they are substitutable for each other.

22.2 ADVANTAGES

In short, reusability of higher level abstractions. The higher we can push reuse, the better (more reliable) and cheaper (faster, greater reuse) our projects will be.

Traditional procedural languages and structured development allow for the reuse of lower level abstractions like data structures and algorithms for manipulating them. Budd uses the example of an algorithm for computing the length of a list. Written generically (at a high level, in other words) as a recursive algorithm in pseudo-C we have:

```
int length( list l) {
  if( l.link  NULL )
    return 1;
  else
    return 1 + length( l.link );
}
```

If you wanted to implement this in real C, you’d have to have a specific data structure. It would be something like this, for a list of integers:

```
struct listst {
  int value;
  struct listst *link;
};

int length( struct listst *lp) {
  if( lp->link  NULL)
    return 1;
  else
    return 1 + length(lp->link);
}
```

So the question is now, what can you use in your next project? The answer is:

- the high level recursive algorithm design
- the actual code, if you happen to be dealing with a linked list of integers

If you have a linked list of widgets in your next project, you’ll have to re-write the length() function.

The goal of a language that supports polymorphism is to let us specify and reuse our algorithms or design.

Polymorphism is necessary for the sort of higher level reuse which is the goal of design patterns and frameworks.

There are several flavors of polymorphism in OO. All of them require that variables can be polymorphic. How a language supports polymorphic variables depends on whether it is statically or dynamically bound.

22.3 BINDING

When is the method that is invoked by a message determined? If it is at compile-time (static binding) then you lose a lot of flexibility. If it is at run-time (dynamic binding) then you have polymorphism, or the ability for each object to react differently to the same message.

An object-oriented language may provide either form of typing (static or dynamic) and either form of binding (static or dynamic), which makes four possibilities to consider. Traditionally, non-OO high level languages use static typing and static binding. Pure OO languages use dynamic typing and dynamic binding.

C++ implements a limited form of polymorphism (dynamic binding) with virtual methods.

Statically bound languages differentiate between the static and the dynamic type of a variable. The compile-time checking is done against the static type. The dynamic type may be one of the subtypes (subclasses) of the static type.

Dynamically bound languages can be very loose, including a completely generic reference type that can refer to any sort of object. These languages push typing errors in to run-time.

Example:

Accumulate the total cost of items in inventory.

```
float cost = 0.0;
for( all objects obj in inventory ){
  typeOfObject = obj.type();
  switch( typeOfObject ){
    case bolts: cost += obj.boltCost();
      break;
    case nuts: cost += obj.nutCost();
      break;
    case screws: cost += obj.screwCost();
      break;
  }
}
```

Each object must be sent a message unique to its type, so we have one line of code for each object class or type. With polymorphism we don’t have the same restriction:

```
float cost = 0.0;
for( all objects obj in inventory ){
  cost += obj.ost();
}
```

The advantage is really obvious when you add washers to your inventory.

Example: Edit->Cut, Copy, Paste

Incorporating a single Edit menu item into your application lets the user perform the same operation (push a button) to cut a character, word, sound, graphic, etc, even though the button push results in very different objects being sent the “cut” message. At compile-time all you know is that the “cut” message will be sent to a selected object; you don’t know the class of that object. At run-time the actual code that is executed is determined by the type of the selected object.

Without dynamic binding, the alternative would be to have Cut_Sound, Cut_Word, Cut_Graphic… Or you would need a large case statement in your controller (where the code for the Edit menu is) that sent a message which depended on the type of object currently selected. In either case, you must change the controller’s code if you ever add another data type.

With dynamic binding you never need to change your controller since all it knows is that it will send a “cut” message to some object. The class of that object may not even exist when the controller is created.

There are several “flavors” of polymorphism if you look broadly at computer languages in general.

1. Pure polymorphism – when a single function (think of a piece of client code) can be applied to objects of many types.
2. Overriding – inheritance of specialization and extension
3. Deferred methods – inheritance of specification
4. Generics – parameterizing the data types of a class so that common behavior can be applied to various types
5. Overloading or ad-hoc polymorphism – when multiple functions have the same name but are distinguished by their parameters

Pure polymorphism

As used by Budd, and other authors, “pure” polymorphism refers to a function which can take parameters of many types.
This occurs as a natural result of the is a relationship. Subclasses which are subtypes are substitutable for their base classes. Clients can’t tell the difference and don’t care. A piece of code can be written in terms of what is common to all the objects (i.e. the interface of the base class).

An example would be the Edit menu of a GUI application. The functions, or behavior, that is being coded is to let users

select, cut, copy, and paste

The objects which these actions apply to may be of a variety of classes (text, audio, image, video, etc). Without polymorphism, you’d have to code this as follows:

```
void cut(selected_object) {
  switch( type of selected_object ) {
    case text:
      selected_object.cutText();
      break;
    case audio:
      break;
    // etc
}
```

Every time you added a data type you’d have to re-write the above function. With polymorphism we can write out cut function abstractly and have it work with a variety of types, including those not yet dreamed of.

Overriding

This sort of polymorphism supports the inheritance of specialization and extension. A subclass implements a different version, or extends the existing version, of a base class method. The method name is the same, but subtypes implement it differently.

This is distinct from overloading because of the type relationship between the overriding classes.

Deferred methods

This sort of polymorphism supports the inheritance of specification.
A method may have no implementation – it could be just an interface specification. Subclasses are required to implement such methods, but in the meantime, clients can be written in terms of the abstract method without worrying about how each subclass implements the method.

Generics

Container classes provide the best example of the use of generics, or parameterized classes.

The behavior common to container classes representing common data structures (queues, stacks, lists, etc) is completely independent (at the high level) of the particular data type being stored in the container.

One solution is to build a very generic container that will hold objects of any type. But what you lose with this approach is the safety of having the compiler type check your code.

What you want are type-safe containers, where the container is code is written in terms of a generic type, but the container class can be created with a specific type and the compiler will check with this specific type.

Overloading

The name of the function is polymorphic. The compiler uses the type (and possibly the number) of parameters to distinguish between multiple functions with the same name.

Basic operators for primitives support overloading.

An overloaded operator or function is not required to have any particular relationship to the class hierarchy. The function being overloaded may be in completely unrelated classes. So, for example, the draw() method works for Artists, Shapes, and CardDealers.

No semantic similarity is implied by overloading. Overloading of function names is likely if you're using good names, since there is redundancy and multiple meaning in human languages.

23. OBJECT ORIENTED CONCEPTS REVIEW

Objects

The active elements of an OO program. Objects are of a definite type (their class, and possibly some other interface) and have two parts: what they know (attributes) and what they can do (behavior). They occupy memory, they get work done, they have a unique id.

Classes

The templates from which objects can be instantiated. The definition of the class determines what objects of that class can know and do. Classes themselves are passive (compared to objects at least, and if you ignore class members).

Encapsulation

The idea that an object encapsulates knowledge and behavior. We control what outsiders may see with access control. Generally, the internal state of an object is hidden from other objects. The algorithms used in the definition of the class methods are hidden by the class.
Includes the idea of containment as an essential relationship between classes.

Inheritance

Classes may be related to each other in an inheritance hierarchy. Top level, abstract classes tend not to be instantiated into active, living, breathing objects. They serve to gather together the common attributes and behavior which their concrete subclasses inherit.

Messaging

Messages are what gets work done in an OO system. There are four parts to a message: a receiver object, a method name, optional method arguments, and an optional return value.

Identity

Objects must have a unique identity, knowledge of which is required to send an object a message. Pointers give us aliases for the unique names of objects, and confuse the issue of identity.

Polymorphism

The idea that the code that is executed as the result of a message being sent depends on the class of the object that receives the message. Objects of different classes can react differently to being sent the same method in a message.

Type

What determines the capabilities or behavior of a thing, such as an object. Distinct from, but often confused with, class. Type is equivalent to the interface of a class. Programming to types, not classes, maintains flexibility.

24. QUALITY OF CLASSES AND OBJECT ORIENTED DESIGN

24.1 THE QUALITY OF CLASSES AND OO DESIGN

Class quality

Object oriented analysis, design, and implementation are an iterative process. It is usually impossible to fully and correctly design the classes of an OO system at the outset of a project.

Booch proposes five metrics to measure the quality of classes:

- Coupling
- Cohesion
- Sufficiency
- Completeness
- Primitiveness

Coupling

How closely do classes rely on each other?
Inheritance makes for strong coupling (generally a bad thing) but takes advantage of the re-use of an abstraction (generally a good thing).

Cohesion

How tied together are the state and behavior of a class? Does the abstraction model a cohesive, related, integrated thing in the problem space?

Sufficiency

Does the class capture enough of the details of the thing being modeled to be useful?

Completeness

Does the class capture all of the useful behavior of the thing being modeled to be re-usable? What about future users (reusers) of the class? How much more time does it take to provide completeness? Is it worth it?

Primitive

Do all the behaviors of the class satisfy the condition that they can only be implemented by accessing the state of the class directly? If a method can be written strictly in terms of other methods in the class, it isn't primitive.

Primitive classes are smaller, easier to understand, with less coupling between methods, and are more likely to be reused. If you try to do too much in the class, then you're likely to make it difficult to use for other designers.

Sometimes issues of efficiency or interface ease-of-use will suggest you violate the general recommendation of making a class primitive. For example, you might provide a general method with many arguments to cover all possible uses, and a simplified method without arguments for the common case.

The truism "Make the common case fast" holds, but in this case ÒfastÓ means "easy". Easy may violate primitive.

24.2 RELATIONSHIPS BETWEEN CLASSES

The Law of Demeter provides a useful guideline:

The methods of a class should not depend on the structure of other classes.

A class should communicate with as few other classes as possible.

Applying this law results in loosely coupled classes.

Class inheritance hierarchy shape

The shape of the class hiearchy may be:

- Wide and shallow
  - Classes are loosely coupled; some commonality may be redundantly implemented.
- Narrow and deep
  - Classes are smaller, but are more tightly coupled to their parent classes.

There is no single "right" shape. This is very problem dependent. Don't expect to see one shape or another in your projects; you may be disappointed.

24.3 PITFALLS IN CLASS DESIGN

These are adapted from Bruce Webster's book.

Confusing class relationships

Isa, Hasa, Association are all different and must be applied properly
confusing interface inheritance with implementation inheritance

C++ gives you lots of control; you need to use it properly.

As a base class, do you want your derived classes to inherit:

- An interface and implementation (normal class function)
- An interface and default implementation (virtual class function)
- An interface only (pure virtual function)

Mis-using inheritance

Not everyone thinks inheritance is a necessary part of OO (small minority, but OLE 2.0 is an example).
Potential problem of coupling; lets a subclass get into the guts of a base class.

Some specific things to watch out for:

- Using MI to turn the isa hierarchy upside down by creating a general class out of two more specific classes. (e.g. Clock, Calendar, ClockCalendar)
- Using MI without restrictions, or a clear understanding of OO ideas
- Functionality of base classes – too much or too little
- Base classes which do to little may:
  - have no public or protected interface (i.e. they impose no protocol)
  - have no implementation (i.e. they are only a protocol)
  - have subclasses which do too much (duplicated code in subclasses)
- Base classes which do too much:
  - Offer an implementation that is mostly overridden by derived classes

Base class invariants

Base classes have assumptions built into them about their state, pre and post conditions, relationships between ivars, timing, invocation of methods, etc.

Since a derived class "isa" base class, the same invariants must hold true. But these invariants aren't necessarily visible from the header, or even sometimes from the implementation file, so it is easy for the designer of the derived class to mess up the invariants she inherits. Documentation and review are the keys.

Class bloat

A class becomes unwieldy at some point: too many ivars, too much functionality. This can happen over time, and may indicate the need to split the class via inheritance or aggregation, or simply other classes.

"Swiss Army Knife" classes are a special case of bloat. Too much functionality unrelated to the abstraction can creep into the class. When it does, people will rely on it, then you're stuck.

Finding classes/objects

All classes show up as nouns in a functional description of the project. This is too simple.

Some classes are obvious, have corresponding physical entities in the problem domain. Others are less obvious and are "discovered" or invented during analysis/design. Gamma says these sort are key to making designs flexible. Many of the objects in the pattern catalog are of this type. This is why it's important to know the catalog.

24.4 QUALITY OO DESIGN

Type versus class

A type is an interface. An interface is a collection of methods which an object responds to. Different kinds of objects may share the same type. An object can have many types.

An object's implementation is defined by its class. The class tells you about the state an object maintains. It also tells you how an object implements the methods in its interface.

In C++ the class and type of an object are the same, so you probably aren't used to separating these ideas. In C++ the class of an object also specifies its type.

Class inheritance is a means of sharing implementation (both state and behavior) between classes.

Interface inheritance (subtyping) only specifies when one object can be used in place of another.

The distinction is important because it is the type of an object that is important to flexible design. Interface is everything in good design; type is interface. Designing with an interface in mind (rather than an implementation) helps because:

- Clients remain unaware (hence decoupled) of the specific kinds of objects they use.
- Clients remain unaware of the classes of the objects they use. They just know the interfaces.

These points together describe what Gamma calls the first "principle of OO design"...

Program to an interface, not an implementation.

Variables should not be concrete classes. Rather, they should be to abstract classes that just specify an interface. Java gives us an even better means of doing this. There are creational patterns for instantiating objects of concrete classes (since you have to get work done somewhere) while remaining unaware of their class.

In C++ you can do pure interface inheritance via public inheritance of pure abstract classes. You can do pure implementation inheritance via private inheritance.

This distinction is important because the design patterns in Gamma often make it. Another interesting aspect of this issue is that it helps explain the features of other OOlanguages (e.g. the "interface"s of Java and "protocols" of Objective C).

Inheritance and composition

Class inheritance is great for implemenation re-use. But it goes against the general goal of decoupled classes. It's also a static, compile-time relationship. This makes it easier to understand and program to, but less flexible.

Composition (aka association) is another means of achieving re-use. Put an object inside another object and the first object can re-use the code behind the composed object. The difference is that the composed object can be determined at run-time. The only stipulation is that the composed object must be of the proper type (not class). If you view composition as an alternative to inheritance then you can in effect change the inheritance or class of an object at run-time.

In C++ we can make a useful distinction between aggregation and composition (aka association). Aggregation is done by value (life times the same between whole/part). Composition is done by reference (reference or pointer, lifetimes de-coupled) so that the object being referred to can change at run-time.

Gamma's second principle of good OOdesign is to:

Favor object composition over class inheritance.

Systems that follow this rule have fewer classes and more objects. Their power is in the ways that the objects can interact with each other. It would be important to have good dynamic models.

Delegation

Delegation is an important design pattern itself. It's used a lot in Objective C and NEXTSTEP programming as an alternative to mandatory inheritance from a complex framekwork and multiple inheritance.

If B is a subclass of A then sending B a message for something in A's interface means that B can re-use the implemtation found in A. Delegation uses composition to put re-use the implemenation from A. B's interface can be extended to have some of the same functionality of A. But when a B object is sent a message corresponding to a method in A's type, it forwards the message to an A object.

The aggregation design we saw earlier for implementing a Set in terms of a List is an example of the use of delegation.

EXAMPLE FROM NEXTSTEP APPKIT

Application object (one per application) knows about connection to window server, the program's icons, starting and stopping the app, power down, etc. Suppose you want to make an app that won't blow away unsaved work when the app is going to quit running due to logout. Subclass Application and add this functionality? Lots of work for minimal additional functionality. Think of all the X and X' classes you'd have in your app. Instead, Application has hooks for delegation built into it like this (translated to C++ like syntax):

```
Application::userLogout(){
  if( delegate != NULL )
    delegate->willShutdown();
}
```

EXAMPLE FROM BOOK: WINDOW AND RECTANGLE

`<C++ code example>`

Advantages: someday Windows may not be rectangular (or 2 dimensional, or …) Design for change now by settling only on an interface, not an implementation.

Disadvantage: Your Window class may be harder to read and understand. More messages means more time.

Static vs Dynamic Structure

Some things are apparent from the static structure and models of a system (e.g. inheritance, aggregation).

An executing OOprogram is a network of messaging objects. The objects come and go. The message patterns shift. The relationships vary over time. Little of this is apparent from the static models of the system. The dynamic models(object diagrams, etc) are crucial for understanding this behavior. If the dynamic behavior relationships have been created in the form of well known patterns then they will be that much easier to document and understand.

For the distinction between the static and dynamic elements of a system, consider an aquarium. What would you know about how an aquarium looks, behaves, captivates, rots, etc, by examining a description of the separate parts (tank, filter, rocks, fish, food, etc)? Now imagine how rich and complicated the interactions of the various components of an aquarium are. Think of what the pieces give rise to.

Design for Change

Here's how patterns address change that can otherwise force re-design:

1. Creating an object by specifying a class explicitly.
2. Specifying one particular operation. Example: NEXTSTEPAppKit has a responder chain which is used to determine who will handle a particular GUIinput event.
3. Limit the spread of platform dependencies. Helps in porting and maintenance to isolate particulars of the GUI or OS.
4. Knowing how an object is represented, stored, located or implemented. Some of this is naturally avoided by encapsulation. But you can take this further to de-couple classes, as in a Distributed Objects NXProxy. (Graph, Node, List examp.)
5. Algorithmic dependencies. Encapsulate algorithms that are likely to change so that change can't ripple. Could even be within the same class.
6. Avoid tight coupling. Classes that are tightly coupled can't be separated or changed independently. Also less likely to re-use.
7. Limiting re-use to inheritance. Composition and delegation provide alternatives which can be simpler and lead to more flexible systems.
8. Classes you can't alter. Patterns give you ways of using classes you can't touch which keeps your design flexible. NEXTSTEP AppKit and use of delegation is good example of this.