COLLEGE OF ENGINEERING
NUCLEAR ENGINEERING & RADIOLOGICAL SCIENCES
UNIVERSITY OF MICHIGAN

# Lecture 9
# Object Oriented Programming and Design

Prof. Brendan Kochunas
10/2/2018

NERS 590-004

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

## Outline

- Quick Revisit of GMRES
- Software Lifecycles for Computational Science and Engineering
- Overview of Object Oriented Programming
    - OO Concepts & Nomenclature
- Example: Sparse Matrix Storage formats
- The Unified Modeling Language
- Design Patterns

## Learning Objectives

- Understand “maturity” and metrics for CSE software
- Know the basic object-oriented program concepts
- Learn the basics of UML and its relationship to OO Software design.
- Understand what design patterns are and how they may be used in scientific computing

## Revisiting GMRES

## Software Lifecycles

## Software Lifecycle Model

**What is it?**

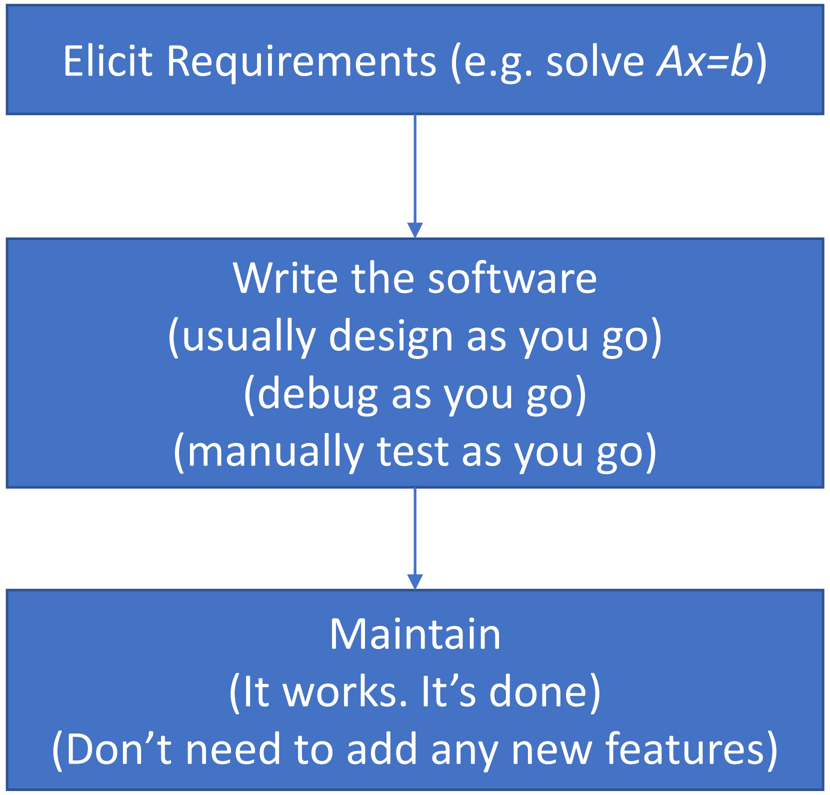
- The model *used to decide when* to perform particular development activities
- Implicit to all software projects
  - Not necessarily formally defined.
- Much better to have a formally defined lifecycle model.
  - Will define "maturity levels"
  - Also defines what activities to perform at each level

**What should a Lifecycle model do?**

- Allow exploratory research to remain productive
  - Don't require more work than necessary in early phases of basic research
- Enable reproducible research
  - Required for credible peer reviewed research
- Improve overall development productivity
  - Focus on right software engineering practices at the right time. Minimize overhead
- Improve production software quality
  - Focus on foundational issues first. Build on quality with quality
- Communicate maturity levels more clearly to customers
  - Manage user expectations

## Example of “Validation-Centric” Lifecycle Model (What you may be familiar with)

- Validation is “doing the right thing”
  - Software product is viewed as “black box” that is supposed to do the right thing.
  - Not generally concerned with the internal structure of the program
- Can be very efficient because it has little overhead initially.
- Usually more difficult to maintain long term
  - Software is poorly designed
  - Difficult to detect changes (no automated testing)
  - Little to no planning


Elicit Requirements (e.g. solve *Ax=b*)
Write the software (usually design as you go) (debug as you go) (manually test as you go)
Maintain (It works. It’s done) (Don’t need to add any new features)

As much as 75% or more of total cost in a software project can be maintenance!

# TriBITS Lifecycle Model: Maturity Levels

- **Exploratory/Experimental (EX)**
  - Primary purpose is to explore alternative approaches and prototypes
  - Little to no testing or documentation
  - Not to be included in a release
  - Very likely code will end up in recycle bin
- **Research Stable (RS)**
  - Strong unit and verification tests
    - Very good line coverage in testing
  - Has a clean design
  - May not be optimized
  - May lack “robustness” and complete documentation
- **Production Growth (PG)**
  - Includes all good qualities of RS code
  - Improved checking for bad inputs
  - More graceful error handling
  - Good documentation
  - Integral and regression testing
- **Production Maintenance (PM)**
  - Includes all good qualities of PG code
  - Primary development activities are bug fixes, performance tweaks, and portability.
- **Unspecified (UM)**
  - Provides no official indication of maturity

## Maturity of Software Quality Metrics (Ideal)

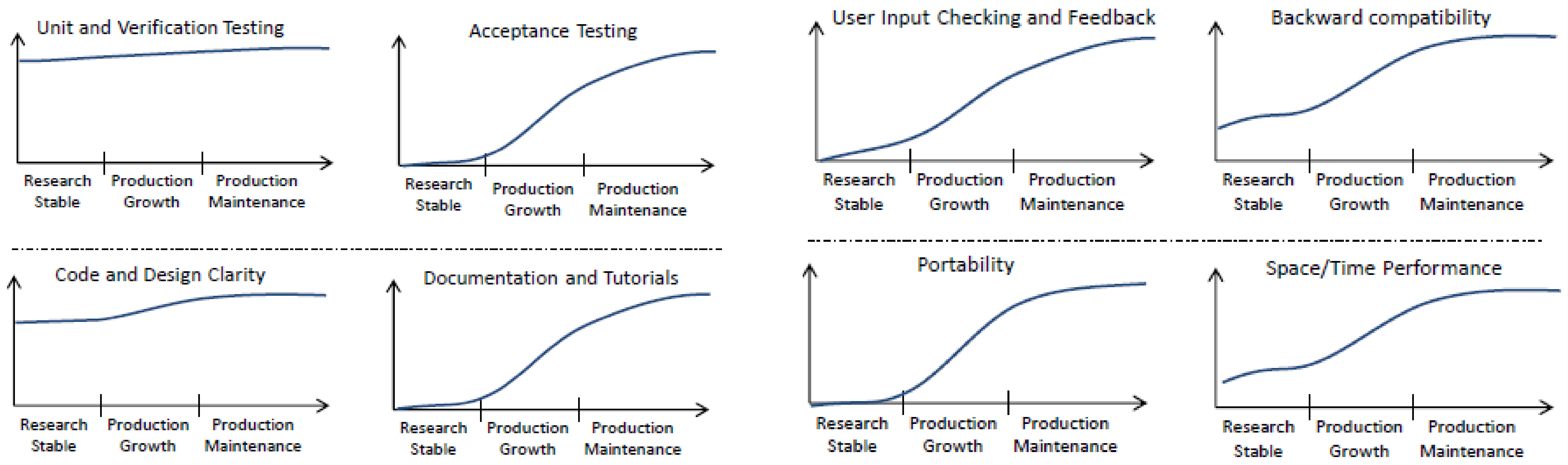
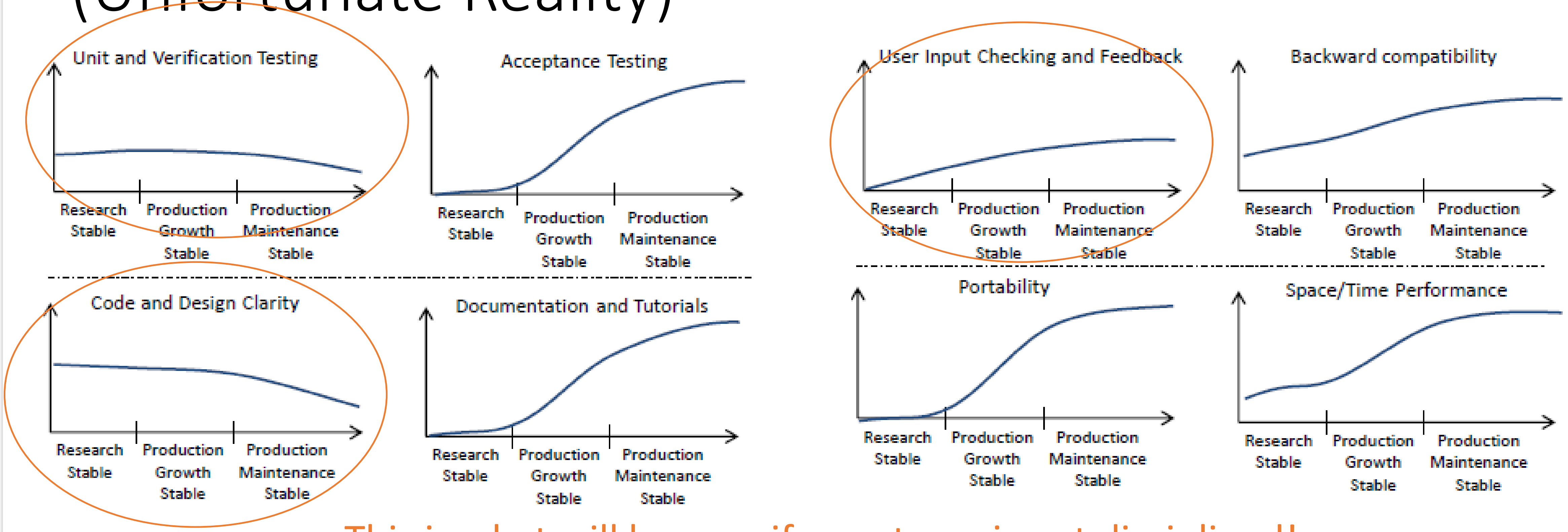


Figure 1. “Typical levels of various production quality metrics in the different phase of the proposed Lean/Agile-consistent TriBITS lifecycle model”
From R. Bartlett, et al., “TriBITS Lifecycle Model” Version 1.0,” SAND2012-0561, (2012)

## Maturity of Software Quality Metrics (Unfortunate Reality)



This is what will happen if your team is not disciplined!

Figure 6. "Example of the more typical variability in key quality metrics in a typical CSE software development process." From R. Bartlett, et al., "TriBITS Lifecycle Model" Version 1.0," SAND2012-0561, (2012)

## Summary

**Development Processes:**

- Add overhead, but will frequently save you time in the long term
- Require discipline
- provide software quality assurance
- may need to be tailored to your situation.

**Software Development consists of:**

- Problem Definition
- Requirements
- Architecture
- High-level design
- Low-level design
- Testing

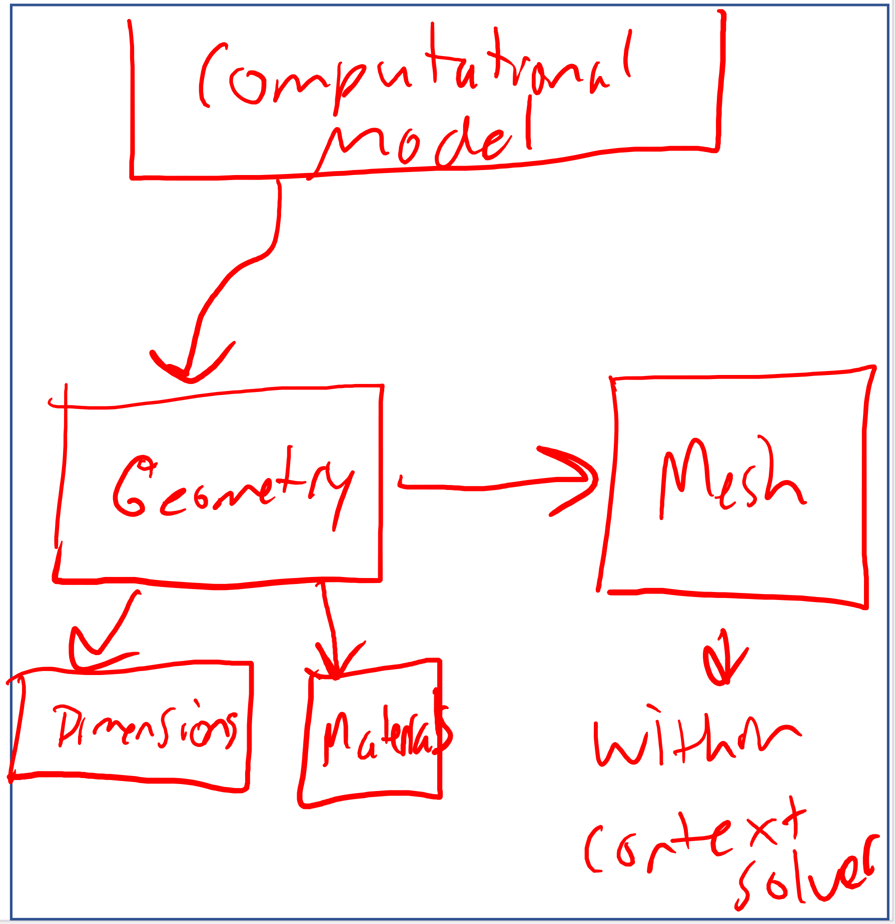
## Object Oriented Programming

- Basis for a lot of modern programming (wait... why?)
  - Improves code reusability
- Keep data and operations on that data “close together”
- In working through OO design, you may find yourself talking like someone who has “lost their marbles” or a philosopher.
  - e.g. What does it *mean* to be “a file”?

## Abstraction

- Abstraction is the idea of *simplifying a concept* in the problem *to its essentials* *within some context*
- Rely on the rule of least astonishment
  - Capture the essential attributes with no surprises and no definitions that go beyond the scope of the context
- This is the process by which you develop classes and their attributes.

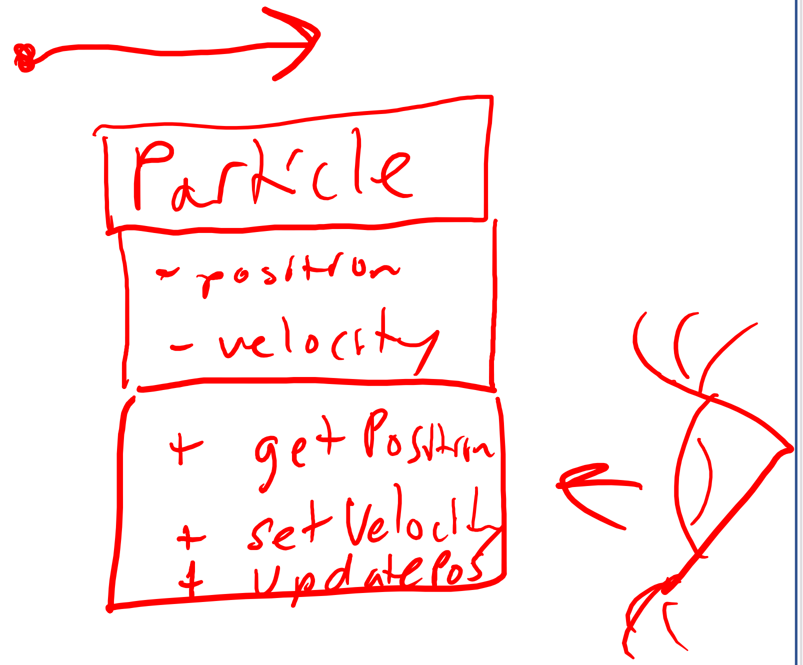
**Example**



## Encapsulation

- Encapsulation means to *hide data*
- If you are used to procedural programming this means eliminating global variables
- With strict encapsulation data is only passed through interfaces
- Advantage:
    - You can change the low-level design without having to update "client code"
- Disadvantage:
    - Argument lists for interfaces can become long, although this can be mitigated.
- Common practice: implement "accessor" functions
    - e.g. set & get

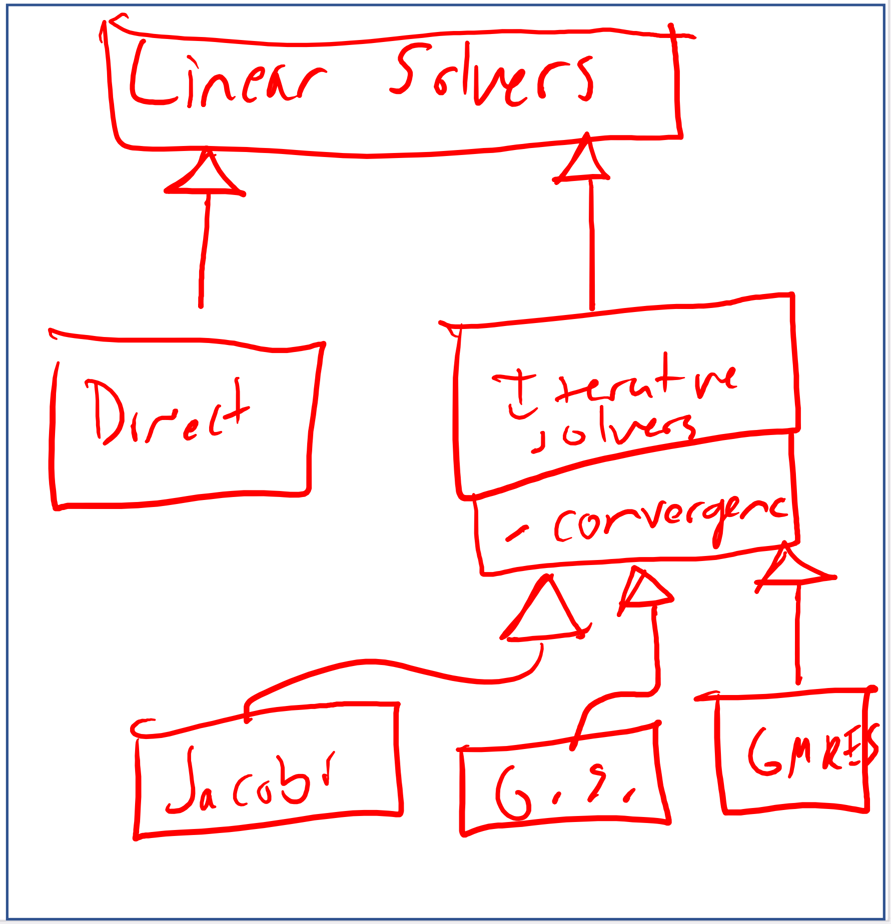
**Example**



## Inheritance

- Inheritance is *a type of relationship between classes*
- Defines a parent-child relationship
  - Child class has attributes and methods of parent class
- Facilitates code reuse
  - Key principle: do not repeat yourself
  - Results from *generalization* of concepts
- Can have single or multiple inheritance
  - Not all languages (e.g. Fortran) support multiple inheritance

**Example**



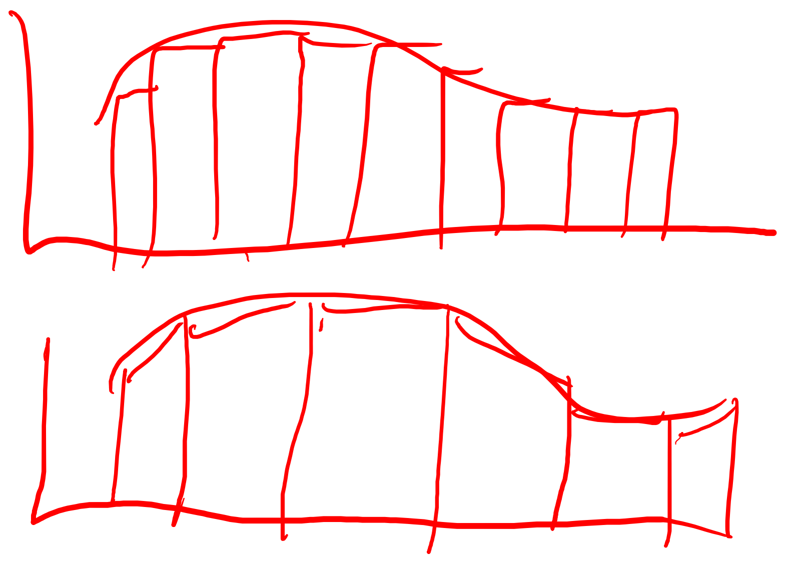
## Polymorphism

- Polymorphism means *to change behavior or representation*
- Facilitates extensibility for inheritance hierarchies.
- Lets clients make fewer assumptions about dependent objects.
  - Decouples objects lets them vary relationships at run time
- Several types of polymorphism
  - static – happens at compile-time
    - lower overhead
  - dynamic – happens at run-time
    - more flexibility

**Example**

Numerical Integration

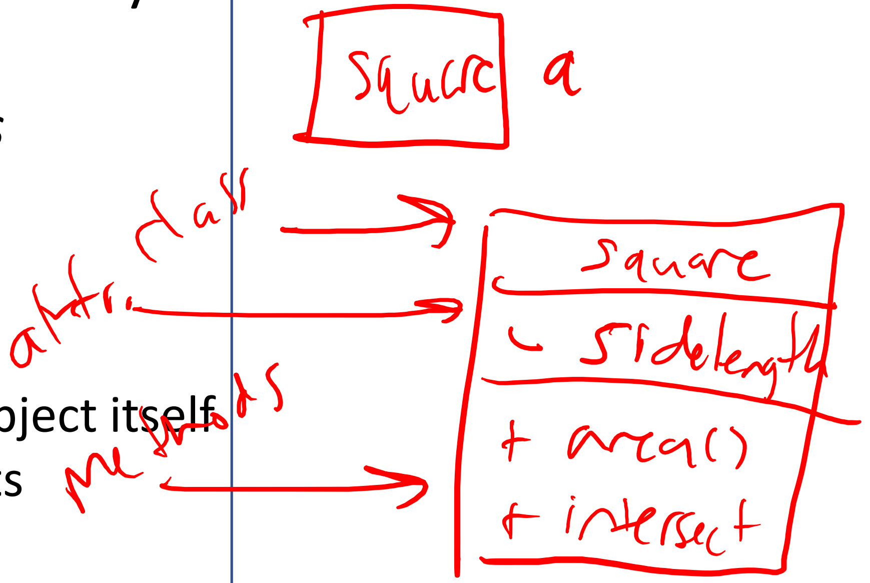
$f(x)$



# Classes, Attributes, and Methods

- In object oriented programming, objects are typically referred to or implemented as “classes”.
- A class is a collection of *attributes* and *methods*
  - attributes are data (or variables)
  - methods are operations (or procedures)
- In good object oriented design:
  - attributes are typically private or only used by the object itself
  - methods are typically public and other code interacts with the object through its methods
- What is it? (answer is attributes)
- What does it do? How is it used? (answer is methods)

**Example**



## Abstract vs. Concrete Objects

- Abstract objects may have incomplete definitions
- Abstract methods have no defined implementation (e.g. low-level design)
  - Just an interface definition
- Abstract methods can only be defined on abstract objects
- Abstract types can be used to define requirements of concrete types

**Example**

Fixed Point

$$x^{(\ell+1)} = F x^{(\ell)} + c$$

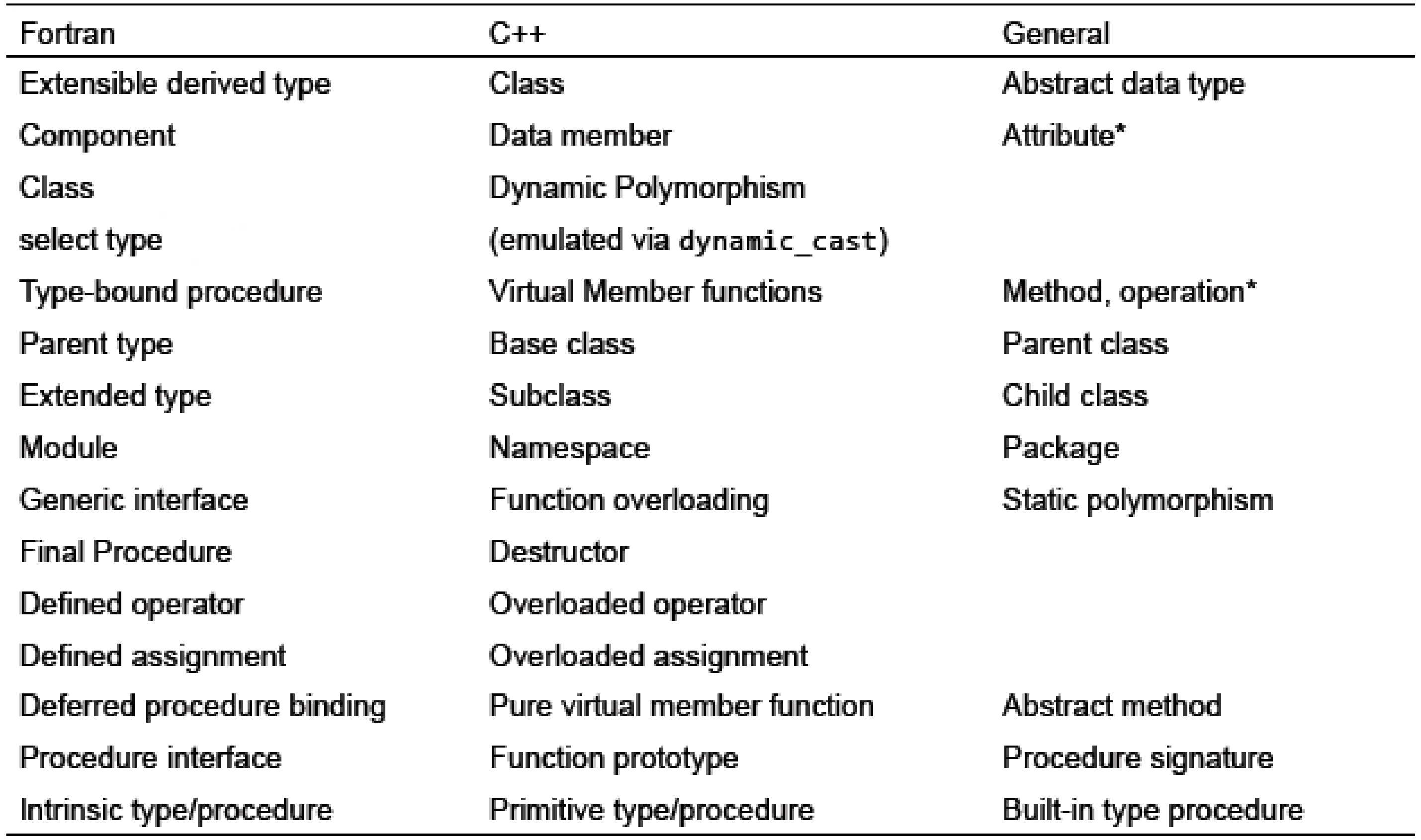
Jacobi

$$F = -D^{-1}(L+U)$$

Gauss-Seidel

$$F = -(D+L)^{-1} U$$

## OO Rosetta Stone for C++ and Fortran



| Fortran | C++ | General |
|---|---|---|
| Extensible derived type | Class | Abstract data type |
| Component | Data member | Attribute* |
| Class | Dynamic Polymorphism | |
| select type | (emulated via `dynamic_cast`) | |
| Type-bound procedure | Virtual Member functions | Method, operation* |
| Parent type | Base class | Parent class |
| Extended type | Subclass | Child class |
| Module | Namespace | Package |
| Generic interface | Function overloading | Static polymorphism |
| Final Procedure | Destructor | |
| Defined operator | Overloaded operator | |
| Defined assignment | Overloaded assignment | |
| Deferred procedure binding | Pure virtual member function | Abstract method |
| Procedure interface | Function prototype | Procedure signature |
| Intrinsic type/procedure | Primitive type/procedure | Built-in type procedure |

From "Scientific Software Design: The Object-Oriented Way", Damian Rouson, Jim Xia, and Xiaofeng Xu

## Implementation Examples

## Encapsulation & Polymorphism (C++)

### Operator Overloading

- Define extra operations for intrinsic operators (e.g. +, −, *, =, etc.)
  - Same idea as function overloading, but functions have special names

```
class Point
{
   public:
      Point operator+(const Point& p) {
        Point point;
        point.x = this->x + b.x;
        point.y = this->y + b.y;
      }
   private:
      double x;
      double y;
};

int main(void)
{
    Point p1,p2;
    p1 = p1 + p2;
}
```

### Function Overloading

- Use same function name, but with different arguments

```
#include <iostream>
using namespace std;
class printData
{
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;
      }
      void print(double  f) {
        cout << "Printing float: " << f << endl;
      }
      void print(char* c) {
        cout << "Printing character: " << c << endl;
      }
};
int main(void)
{
   printData pd;

   pd.print(5);           // Call print to print integer
   pd.print(500.263);     // Call print to print float
   pd.print("Hello C++"); // Call print to print character
   return 0;
}
```

# Encapsulation and Polymorphism (Fortran)

## Operator Overloading

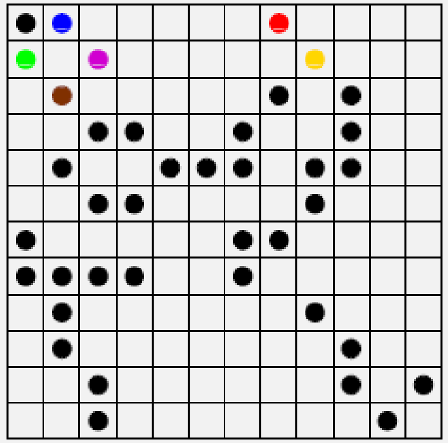
```
MODULE  Points

TYPE :: Point
  REAL(8) :: x,y
ENDTYPE

INTERFACE OPERATOR(+)
  MODULE PROCEDURE add_Point
ENDINTERFACE

CONTAINS

  FUNCTION add_Point(p1,p2) RESULT(p)
    TYPE(Point),INTENT(IN) :: p1,p2
    TYPE(Point) :: p
    p%x=p1%x+p2%x
    p%y=p1%y+p2%y
  ENDFUNCTION

ENDMODULE
```

## Function Overloading

```
MODULE printData

INTERFACE print
  MODULE PROCEDURE print_int
  MODULE PROCEDURE print_double
  MODULE PROCEDURE print_char
ENDINTERFACE

CONTAINS

  SUBROUTINE print_int(i)
    INTEGER,INTENT(IN) :: i
    WRITE(*,*) i
  ENDSUBROUTINE

  SUBROUTINE print_double(d)
    REAL(8),INTENT(IN) :: d
    WRITE(*,*) d
  ENDSUBROUTINE

  SUBROUTINE print_char(c)
    CHARACTER(LEN=*),INTENT(IN) :: c
    WRITE(*,*) c
  ENDSUBROUTINE

ENDMODULE
```

## Example: Sparse Matrix Storage

## Sparse Matrix Storage Formats: COOrdinate Storage

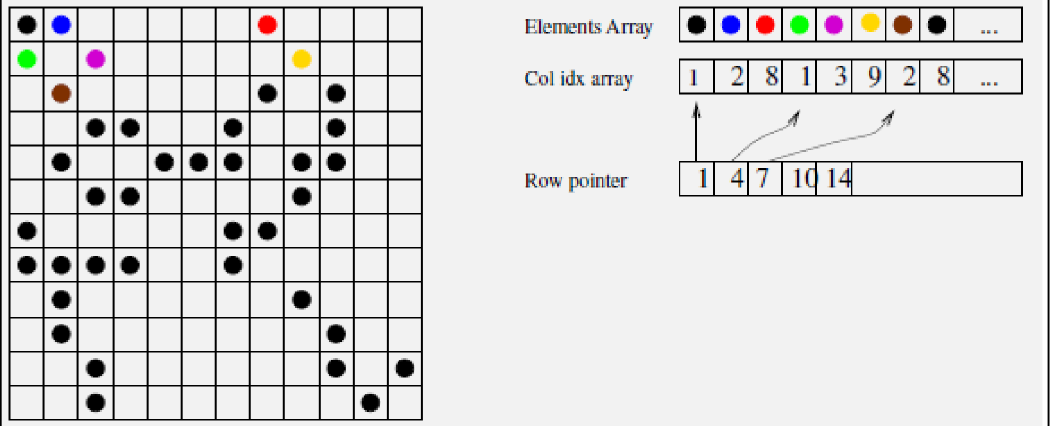


| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Elements Array | ● | 🔵 | 🔴 | 🟢 | 🟣 | 🟡 | 🟤 | ● | ... |
| Col idx array | 1 | 2 | 8 | 1 | 3 | 9 | 2 | 8 | ... |
| Row idx array | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | ... |

```
do i=1,nz
   ir = ia(i)
   jc = ja(i)
   y(ir) = y(ir) + as(i)*x(jc)
enddo
```

Cost: 5 memory reads, 1 write and 2 flops per iteration.

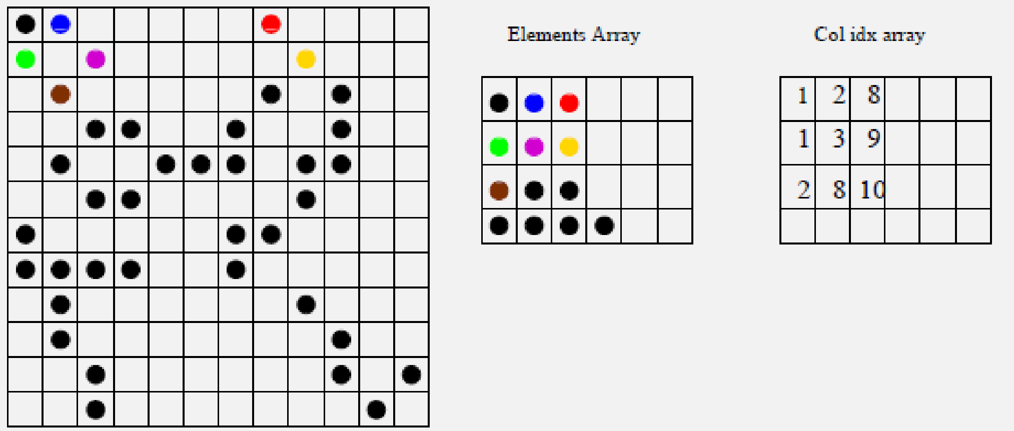
## Sparse Matrix Storage Formats: Compressed Sparse Row (CSR) Storage



```
do  i=1,m
   do  j=ia(i),  ia(i+1)-1
      y(i) = y(i) + as(j)*x(ja(j))
   enddo
enddo
```

Cost: 3 memory read and 1 write per outer iteration, 3 memory read and 2 flops per inner iteration.

# Sparse Matrix Storage Formats: ELLPACK Storage



```
do i=1,m
   do j=1, maxnz
      y(i) = y(i) + as(i,j)*x(ja(i,j))
   enddo
enddo
```

Cost: 1 memory read and 1 write per outer iteration, 3 memory read and 2 flops per inner iteration (also, regular access pattern).