# Lecture 11 – Dev Process and Object Oriented Programming and Design

Prof. Brendan Kochunas

NERS/ENGR 570 - Methods and Practice of Scientific Computing (F22)

COLLEGE OF ENGINEERING
NUCLEAR ENGINEERING & RADIOLOGICAL SCIENCES
UNIVERSITY OF MICHIGAN

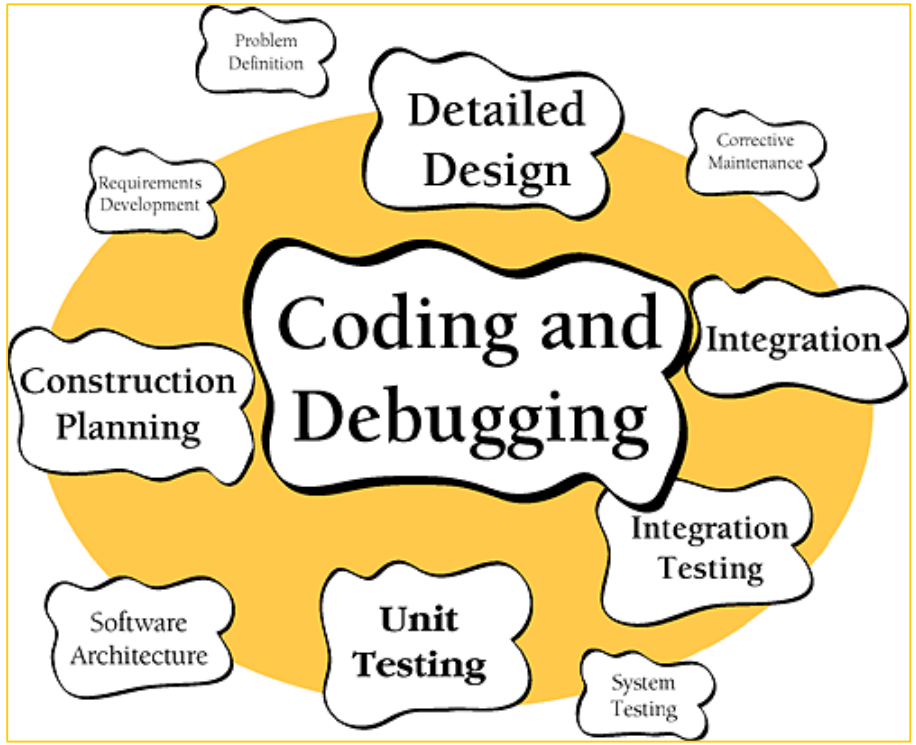
# Outline

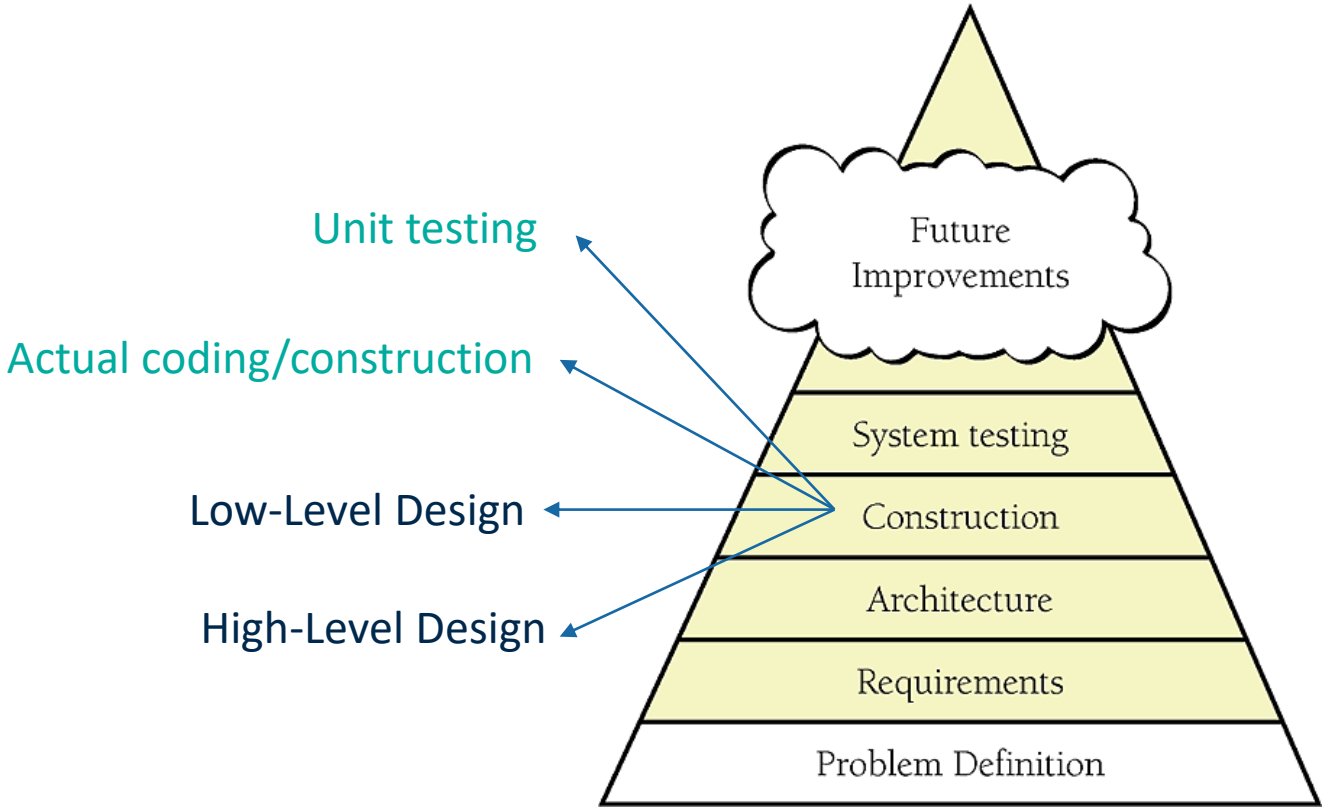
- Recap (5 min)
- Finish Lecture 10 (30 min)
  - Development Workflows
  - Software Lifecycles
- Brief Overview of Testing (5  min)
- Object Oriented Programming (30 min)
  - Interactive application to SpMV
- Preview of Lab 06

# Learning Objectives: By the end of Today's Lecture you should be able to

- (*Knowledge*) identify when to perform certain activities in software development
- (*Knowledge*) list and define common types of testing categories
- (*Skill*) draw a simple UML Class diagram related to a computational science application
- (*Knowledge*) provide definitions of fundamental concepts in OO programming

# Software Construction



# Development Workflows

Managing the Chaos and putting the routine in subroutine

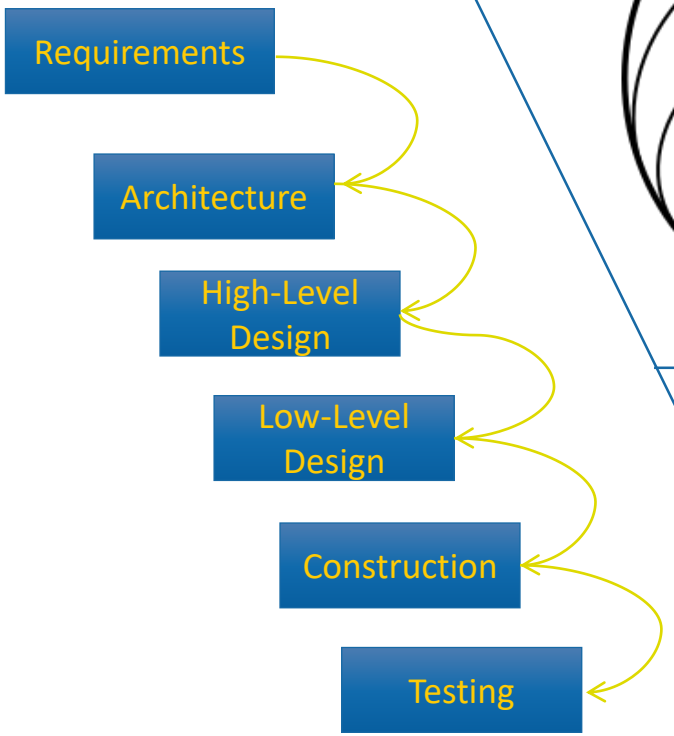
# Development Workflows

Further Reading: https://en.wikipedia.org/wiki/Software_development_process

- Workflows are *based on a particular philosophy* or approach to software development.
  - Waterfall: Once through and your done
  - Incremental: Perform the same tasks in cycles
    - Spiral: Focused on risk management
  - Iterative:
    - Agile: Work in a way that lets you adapt quickly
      - Scrum, Extreme programming
  - Lean: Don't do more than you need to. Minimize "waste".
    - Kanban
  - ...and many more
- There is no right or wrong workflow, but for certain projects in certain situations some workflows will be more productive than others.
  - Often times you need to tailor something specific for your project & team.

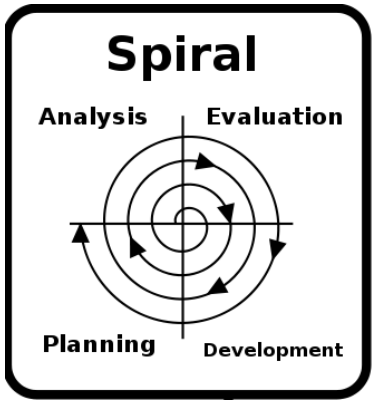
# Workflow Illustrations

## Waterfall

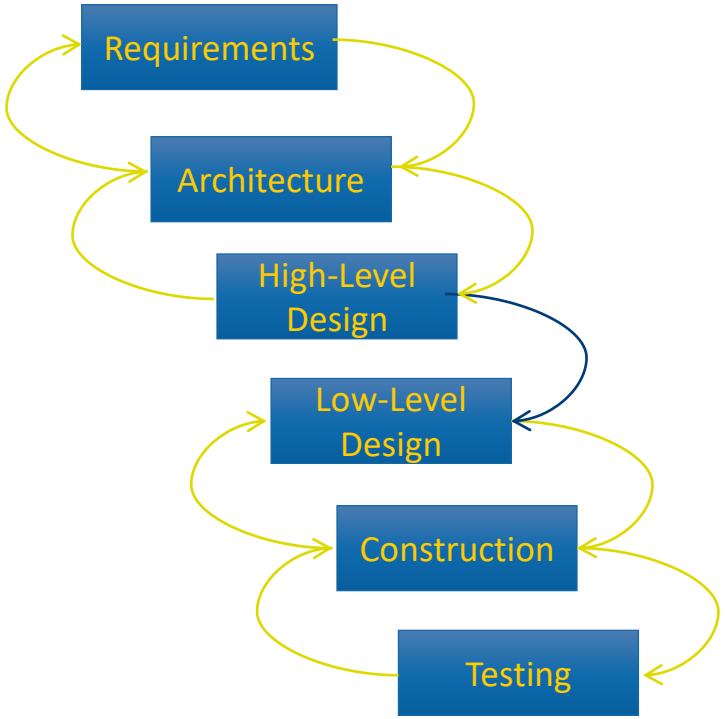



## Extreme Programming


## Spiral



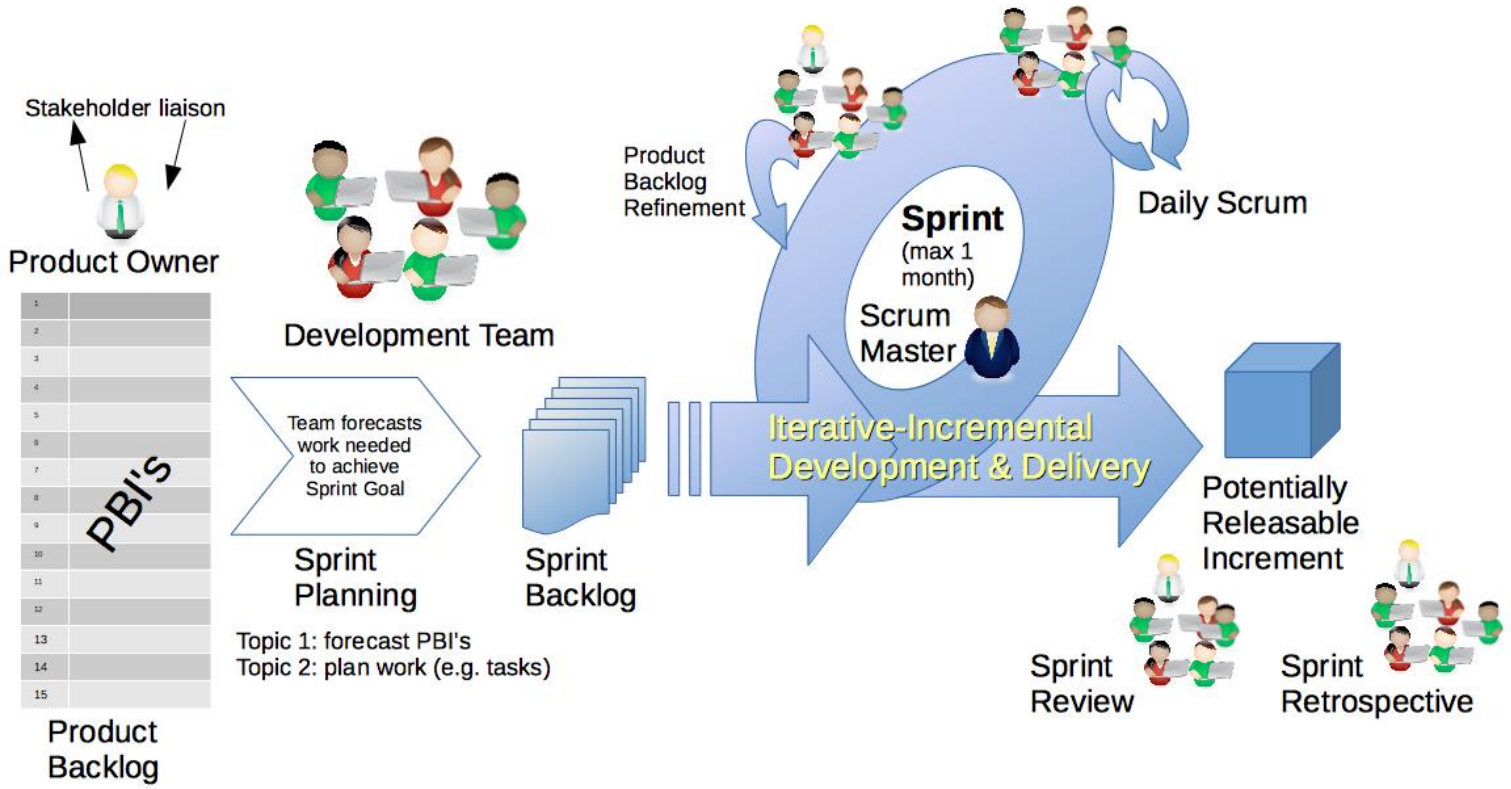

## Agile Iterative



# Other Workflow Concepts: Scrum (software development in groups)

**Concepts**

- Project roles
- Time-boxing
- Backlog
- Stand-ups
- Review and Retrospective



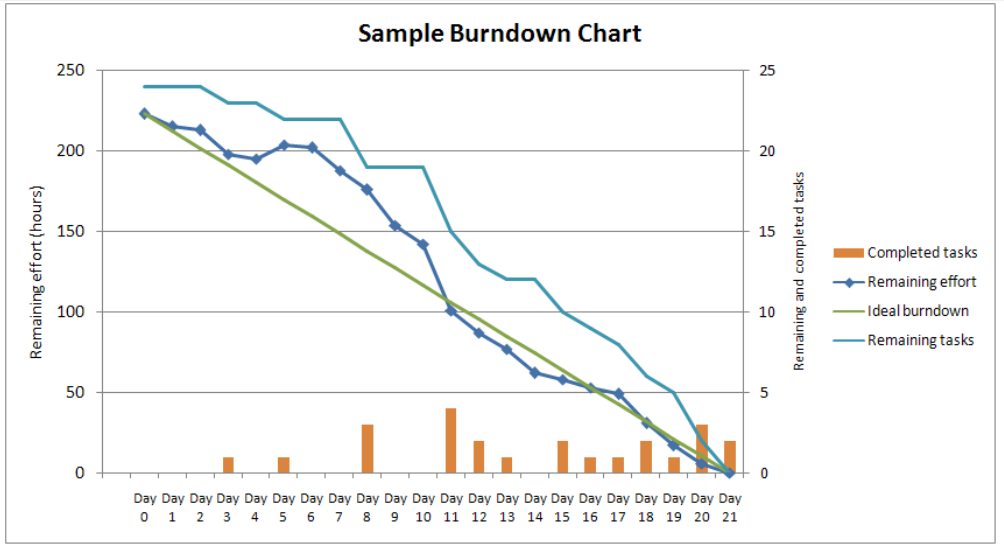

https://upload.wikimedia.org/wikipedia/commons/thumb/d/df/Scrum_Framework.png/220px-Scrum_Framework.png

# Other Workflow Concepts: Kanban & Burndown (resource management)

Kanban Board


## Concepts

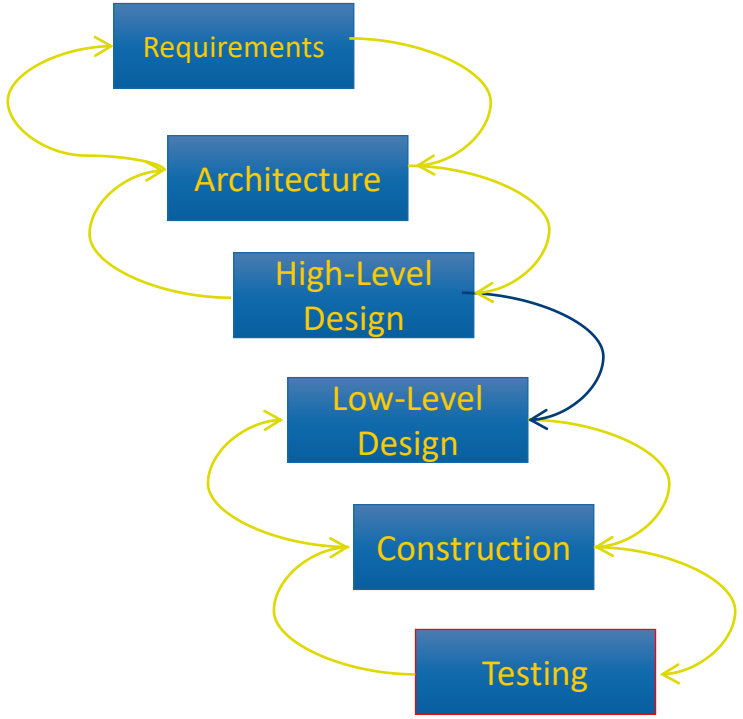
- Visualize Workflow.
- Limit work in progress.
- Evolve policies.
- Burndown Charts.



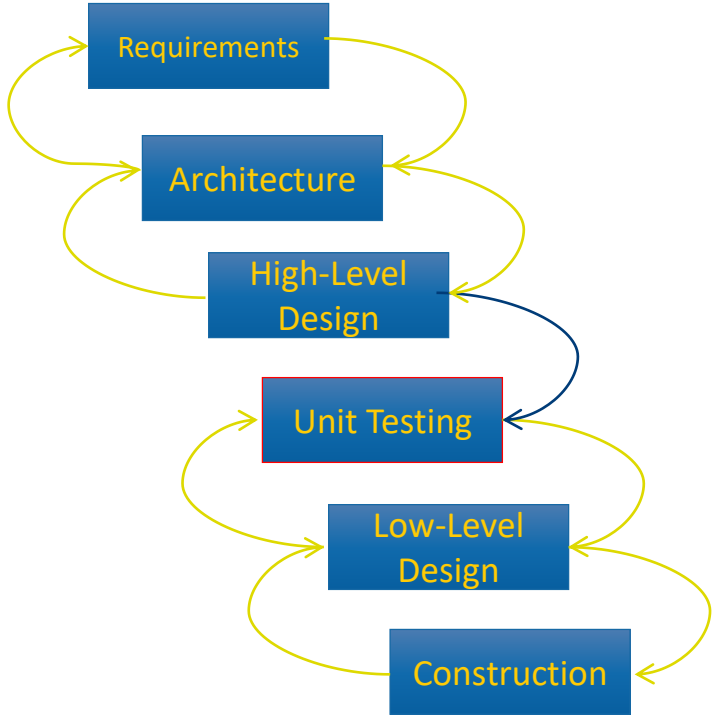

https://upload.wikimedia.org/wikipedia/commons/0/05/SampleBurndownChart.png
https://upload.wikimedia.org/wikipedia/commons/thumb/d/d3/Simple-kanban-board-.jpg/1280px-Simple-kanban-board-.jpg

# Test Driven Development

**Agile Iterative Development**

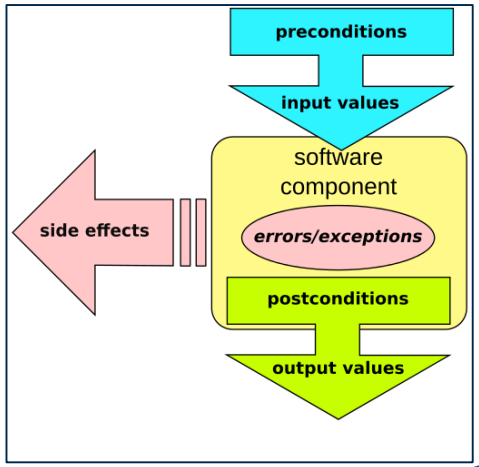



**Test Driven Development**



# Design by Contract

- Articulated from "business contracts" for a client and supplier
  - A form of "defensive programming" where overhead can be eliminated
  - *Check assumptions made by code*
- Not natively supported in most languages
  - Available as a 3RD party feature
    - For C/C++ use C preprocessor or GNU Nana (assert function)
    - For Fortran use C-preprocessor
    - Python has PyContracts or PyDBC




```
SUBROUTINE MatVec(A,x,y)
  REQUIRE(SIZE(A,DIM=2) == SIZE(x))
  REQUIRE(SIZE(A,DIM=1) == SIZE(y))

  !Perform y <- A*x
  ENSURE(.NOT.(y /= y)) !test for NaN
ENDSUBROUTINE
```

Debug

```
SUBROUTINE MatVec(A,x,y)
  IF(.NOT.(SIZE(A,DIM=2) == SIZE(x))) &
    WRITE(0,*) "DBC REQUIRE FAIL!", &
    __FILE__,__LINE__
  IF(.NOT.(SIZE(A,DIM=1) == SIZE(y)) &
    WRITE(0,*) "DBC REQUIRE FAIL!", &
    __FILE__,__LINE__

  !Perform y <- A*x

 IF(.NOT.(.NOT.ANY((y /= y))) &
   WRITE(0,*) "DBC ENSURE FAIL!", &
   __FILE__,__LINE__
 ENDSUBROUTINE
```

Release

```
SUBROUTINE MatVec(A,x,y)

  !Perform y <- A*x
ENDSUBROUTINE
```

# Final Disclaimer on Workflows

- Using workflows effectively is like using version control (or a lab notebook) effectively.
- Workflows require self-discipline.
- They do not help you if you do not adhere to their rules.
- Typically requires active effort on the part of someone to “enforce” workflow practices
- They can be a lot of overhead at times.

# Software Lifecycles

Communicating the quality of code and *when* to do what

# Software Lifecycle Model

## What is it?

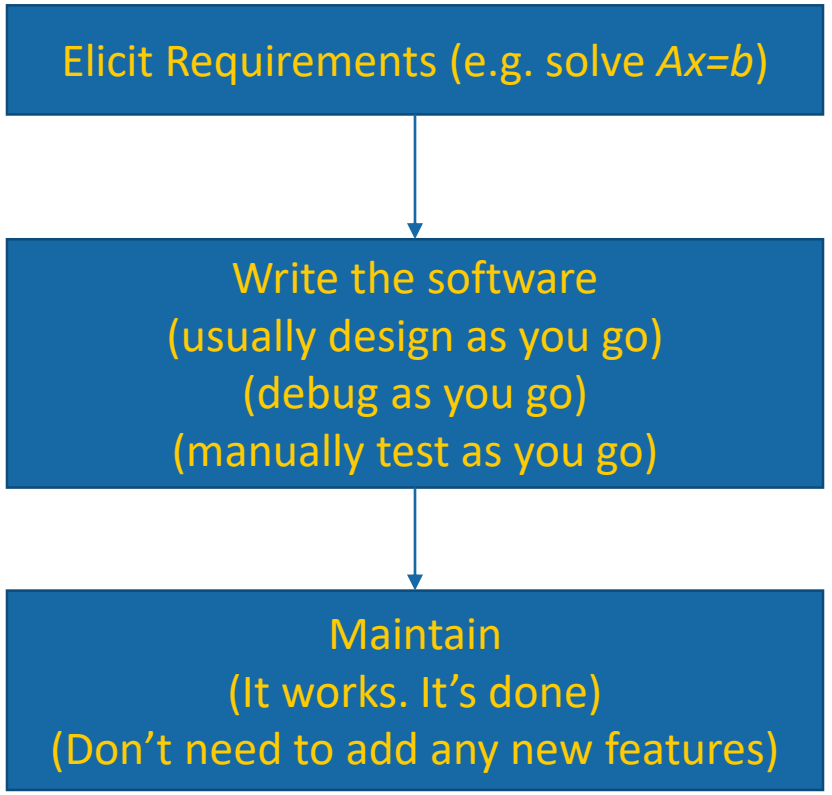
- The model *used to decide when* to perform particular development activities
- Implicit to all software projects
  - Not necessarily formally defined.
- Much better to have a formally defined lifecycle model.
  - Will define “maturity levels”
  - Also defines what activities to perform at each level

## 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




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)

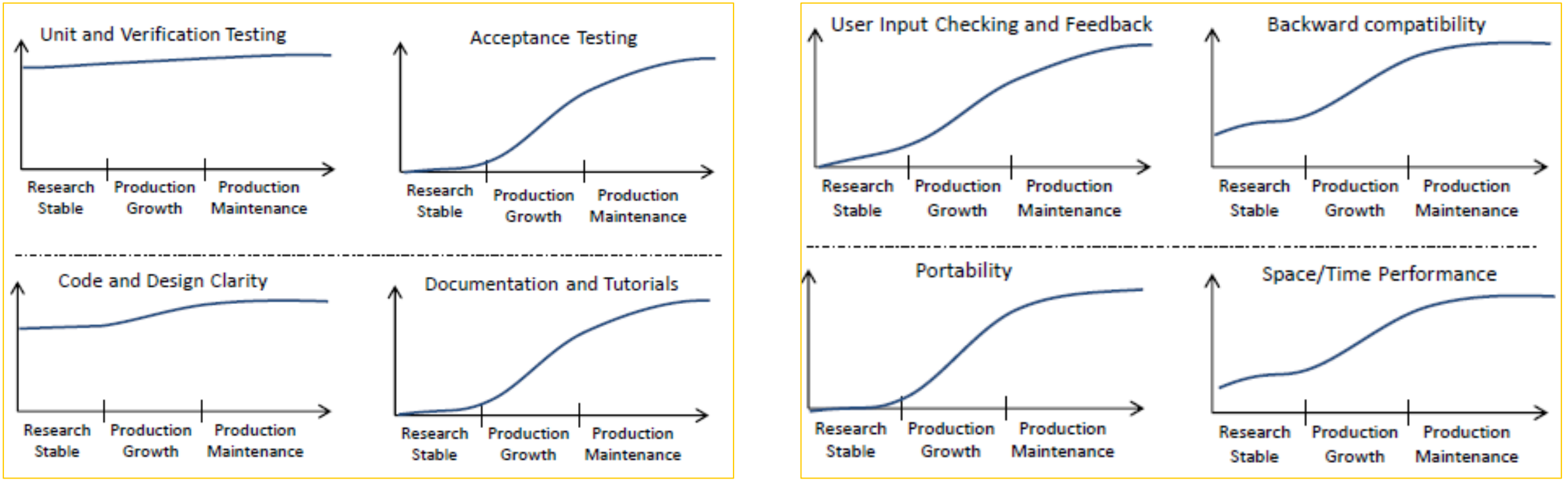
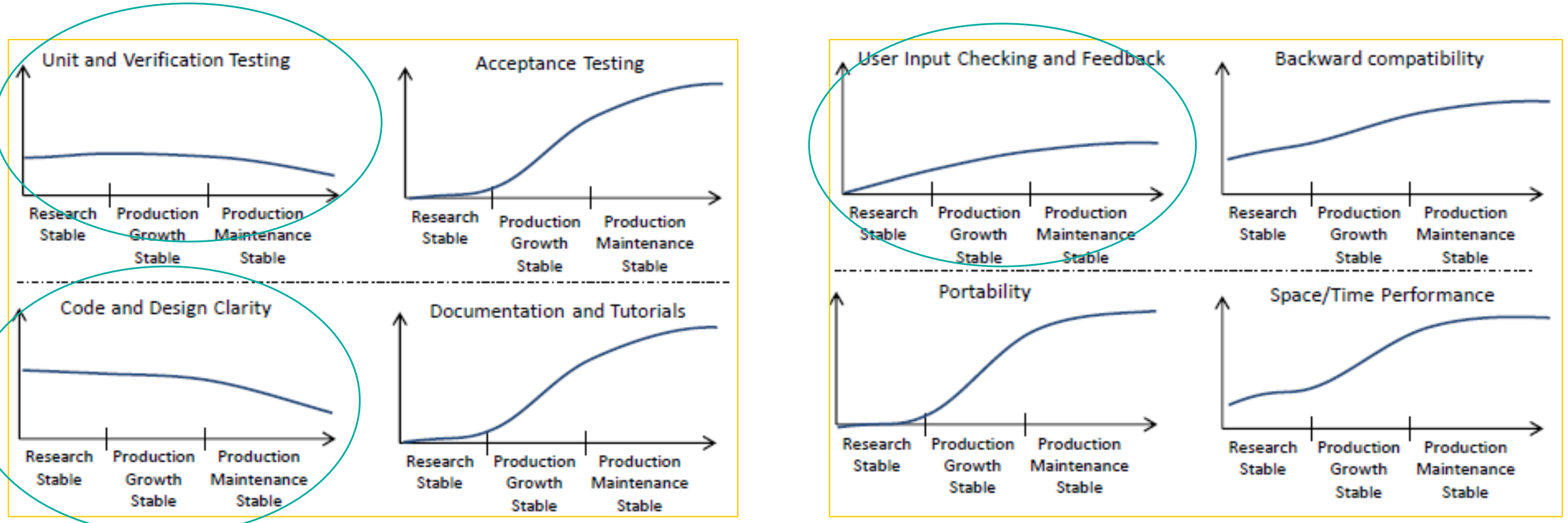



Figure 1. "Typical levels of various production quality metrics in the different phase of the proposed Lean/Agile-consistent TriBITS lifecycle model"
From R. Bartlett, et al., "TriBITS Lifecycle Model" Version 1.0," SAND2012-0561, (2012)

# 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 of Lifecycle for Scientific Computing

- Make it work
- Make it correct
- Make it robust
- Make it fast
- Make it easy to use

# Overall 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:**

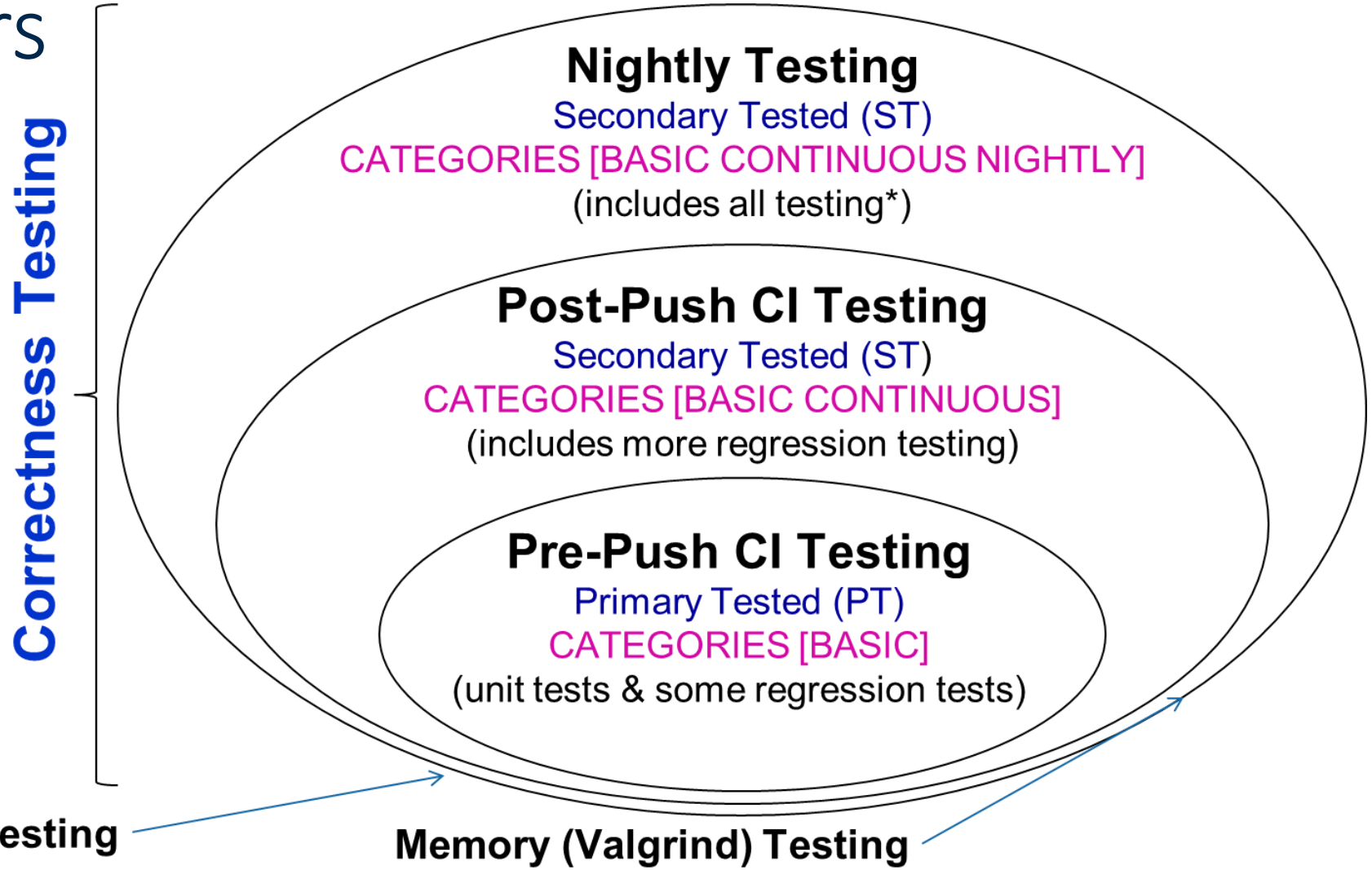
- Problem Definition
- Requirements
- Architecture
- High-level design
- Low-level design
- Testing

# Ross's Taxonomy of Testing

# A Taxonomy of Testing

- Testing is the backbone of software quality assurance (SQA).
- Types of testing
  - *Unit Testing* – Test individual units of program *in isolation*
    - Should run very fast: < 1 second (a couple seconds is ok)
  - *Integral Testing* – Testing program components together
    - Should run fast: < 1 minute (a couple minutes is ok)
  - *Regression Testing* – Test whole program for changes in program output
    - Should run fast: < 1 minute (a couple minutes is ok)
  - *Verification Testing* – Test that you are "doing things right"
    - Can happen at unit or integral or regression level. Comparison analytic solutions or manufactured solutions.
  - *Validation Testing* – Whole program testing "doing the right thing"; simulating reality, comparison to experiment.
    - May be long running: minutes to hours
  - *Memory Testing* – Expensive testing that does detailed memory simulations to detect errors (valgrind)
  - *Coverage Testing* – Figure out how much of your source code is actually covered by testing
  - *Portability Testing* – test on different platforms and with different compilers
- Other types of testing exist

# Testing Layers



# AUTOMATE TESTING AS MUCH AS YOU CAN!

# Object-Oriented Programming

# Object Oriented Programming

- Basis for a lot of modern programming (wait... why?)
  - Improves code reusability
- Fundamental idea: 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 matrix”?
- Lots of similar terminology that gets confusing.
  - Think like a philosopher—terms have very specific definitions within a specific context (project, discussion, paper, etc.)

# 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)

# 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.

# 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

# 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

# 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

# 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

# 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

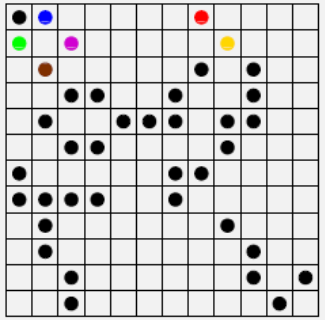
# Lab 06 - Workflow

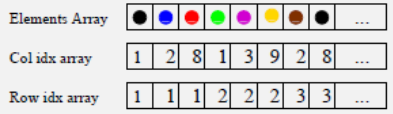
C++ Library supporting various Sparse Matrix Storage formats to perform SpMV

Make use of OO Design and Design Patterns

# Matrix Storage Formats

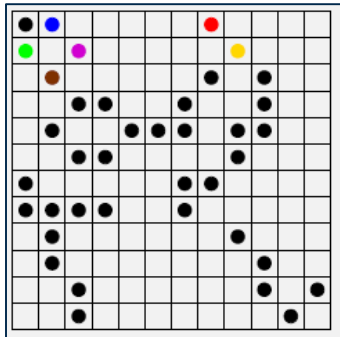
## COO

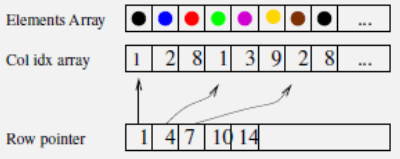





```
do i=1,nz
   ir = ia(i)
   jc = ja(i)
   y(ir) = y(ir) + as(i)*x(jc)
enddo
```

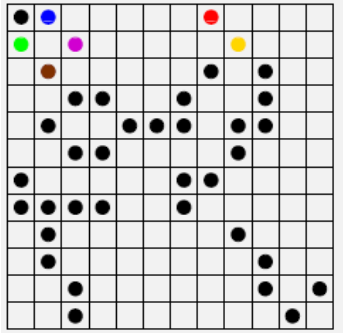
## CRS

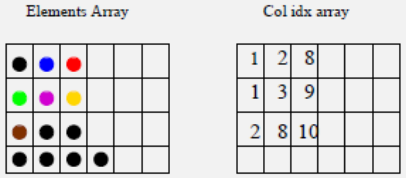





```
do i=1,m
   do j=ia(i), ia(i+1)-1
      y(i) = y(i) + as(j)*x(ja(j))
   enddo
enddo
```

## ELLPACK



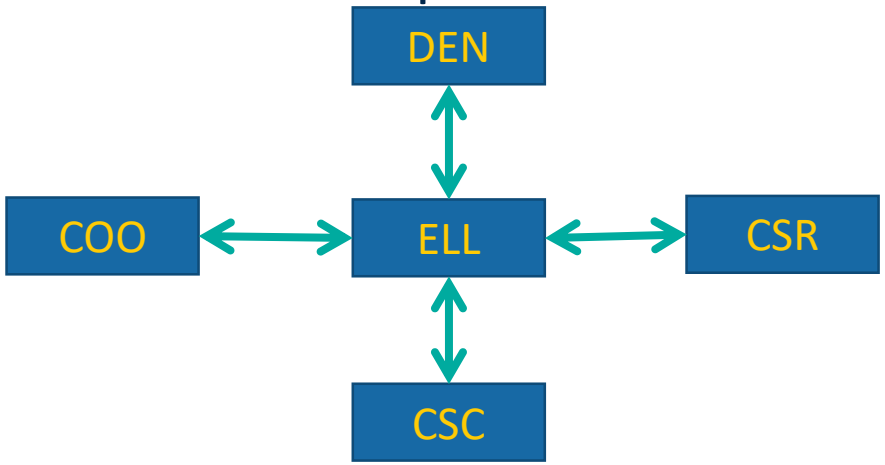


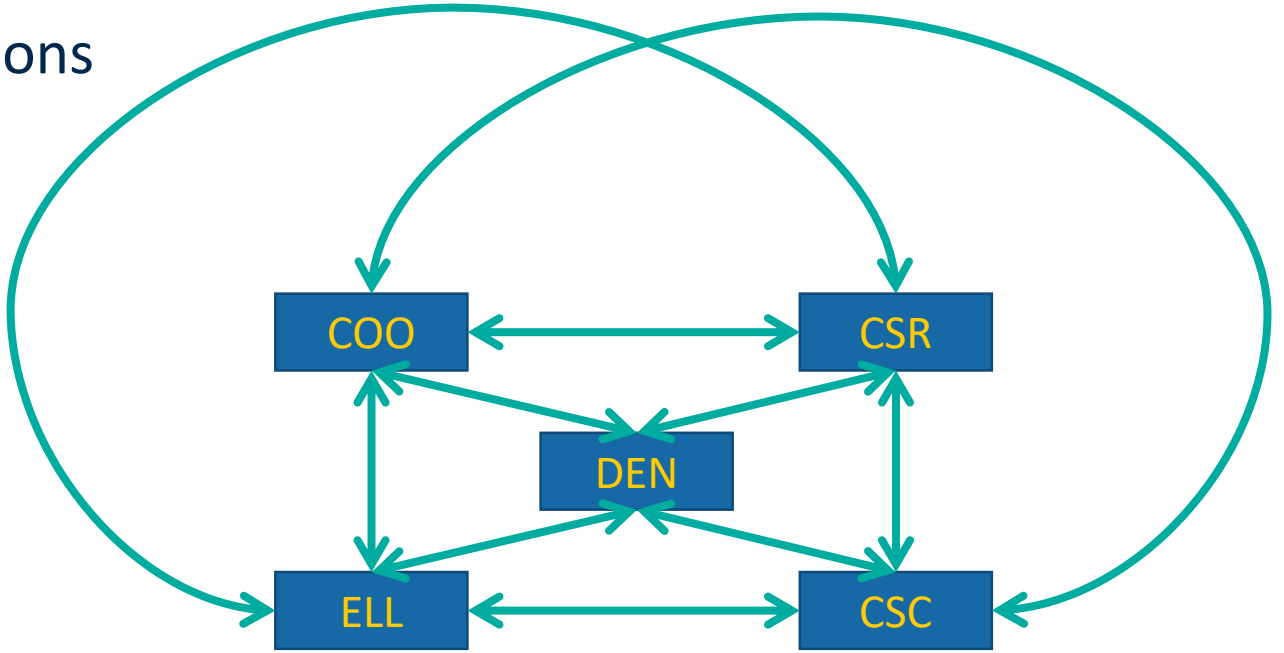

```
do i=1,m
   do j=1, maxnz
      y(i) = y(i) +  as(i,j)*x(ja(i,j))
   enddo
enddo
```

Other formats: Block CRS, Jagged Diagonal, Dense

# Example of Design Pattern Application

- Support *N* matrix formats
  - That is $N^2$ different types of conversions
  - Don't implement them all!




- Use Mediator!
  - Move from fully connected graph to "star" graph