Lecture 10 – Software Engineering and Workflows

Prof. Brendan Kochunas

NERS/ENGR 570 - Methods and Practice of Scientific Computing (F20)

COLLEGE OF ENGINEERING
NUCLEAR ENGINEERING & RADIOLOGICAL SCIENCES
UNIVERSITY OF MICHIGAN

Outline

- Software Construction
  - Short Tangent on UML
- Development Workflows
- Software Lifecycles
- Testing (Time permitting)

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

- (Knowledge) define the phases of software development/construction
- (Skill) read (and write) some simple UML diagrams
- (Knowledge) identify when to perform certain activities in software development
- (Skill) implement software engineering best practices in github

Overview of Software Engineering

Source material: Steve McConnell, *Code Complete 2nd Edition*, Microsoft Press, 2004.

Software Engineering Practices

- Software Engineering Practices relate to the question: "How do you write your software?"
- Includes topics such as
  - Version control
  - Testing
  - Lifecycle
  - Release schedule
  - Development process (software construction)
  - Coding standards
- Its all about your PROCESS!

Metaphors in Writing Software

Various metaphors

- Historically writing software has been called:
  - “a science”
  - “an art”
  - “a process”
  - “a game”
  - “farming, hunting werewolves, or drowning with dinosaurs in a tar pit”
- Presently:
  - Writing software is like “construction” (e.g. building a house)
    - implies planning, preparation, and execution

Learning from the Metaphor

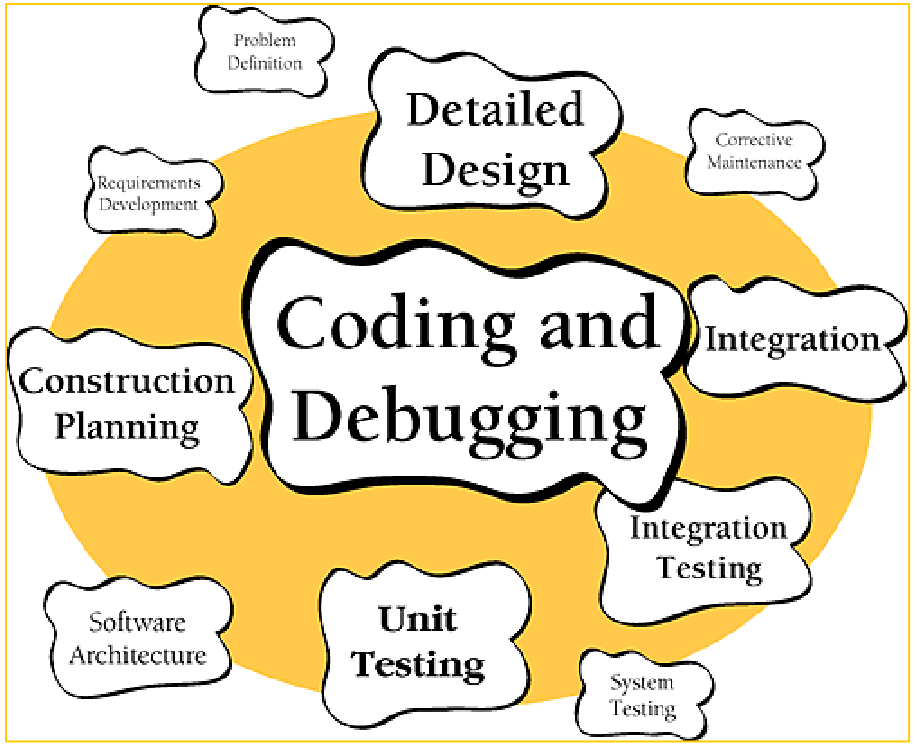
- Things needed for simple structures may not work for large structures
  - “Building a 4’ tower requires a steady hand, a level surface, and about 10 undamaged coke cans”
    - Building a tower that’s 400’ can’t use 400 coke cans. You’ll need something else...
- Simple projects may not need a lot of planning

- Don’t build things you don’t have to
  - You’re not going to build a dishwasher from scratch for your house
- Plan appropriately: e.g. how much complexity is there?

Software "Construction"

What is Software Construction

- Software construction is software development. It includes:

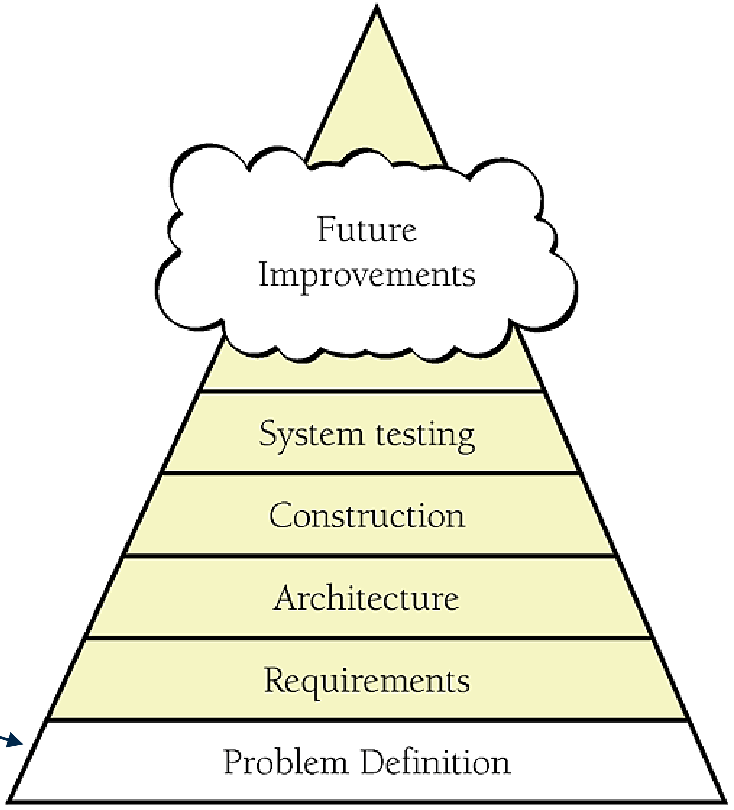


Why is it important?

- Construction is the central part of any software project.
  - The point is you write code.
  - It is the only guaranteed activity in software development
- Like actual construction it can be done well or poorly.
  - As computational scientists we want to do it well.
  - Good software construction leads to users and leads to a career.

Parts of Software Construction: Problem-Definition

- The problem definition defines the problem to be solved WITHOUT any reference to possible solutions.
  - Kind of like a vision statement or mission statement.
  - Should be in terms of the “user” not the “developer”
- Example:
  - “We need to make sure a nuclear reactor will shutdown safely in the event of an accident.”
- The problem statement is the foundation of everything that follows in software construction.
- The penalty for failing to get the right problem definition is you can waste *a lot of time* solving the wrong problem.



Parts of Software Construction: Requirements (1)

- Requirements describe in detail what a software system is supposed to do.
  - It is the "contract" between the customer/user and the developer.
  - Useful to define for each individual part of the program or feature
- Usually describe functional requirements
  - In CSE this is usually solving a mathematical equation
  - Define inputs (coefficients)
  - Define outputs (solution) and output format
  - *Should* define error handling behavior
  - *May* define performance goals
  - *May* define solution algorithm
- Requirements always change
  - "Requirements are like water. They're easier to build on when they're frozen"
- **Requirements always change**
- **REQUIREMENTS ALWAYS CHANGE**
  - Changing requirements when desigining ~3x overhead
  - Changing requirements during construction 5x-10x overhead
  - Changing requirements after release 10x-100x overhead

Parts of Software Construction: Requirements (2)

More Analogies

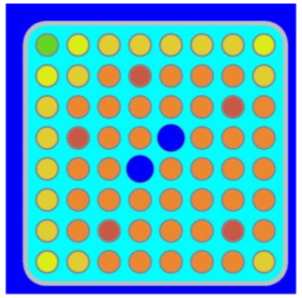
- Importance of planning:
  - You don’t just start hammering boards together with nails when you want to build a house
  - You figure out what a house needs to do by familiarizing yourself with building codes.
- Affect of changing requirements
  - After the construction crew has put the dry-wall up and installed the windows the homeowner decides they want to move some windows.
  - When the homeowner is meeting with the architect they decide to move some windows.

Example

- Obtain a solution to the following equation

$$-\vec{\Omega}\cdot\nabla\varphi\left(\vec{r},\vec{\Omega},E\right)+\Sigma_t\left(\vec{r},E\right)\varphi\left(\vec{r},\vec{\Omega},E\right)=\frac{\chi(E)}{4\pi k_{eff}}\int_0^\infty \nu\Sigma_f\left(\vec{r},E'\right)\phi\left(\vec{r},E'\right)dE'$$
$$+\int_0^\infty\int_0^{4\pi}\Sigma_s\left(\vec{r},\vec{\Omega}'\cdot\vec{\Omega},E'\rightarrow E\right)\varphi\left(\vec{r},\vec{\Omega}',E'\right)d\vec{\Omega}'dE'$$

- For the following model



Boiling Water Reactor Fuel Lattice

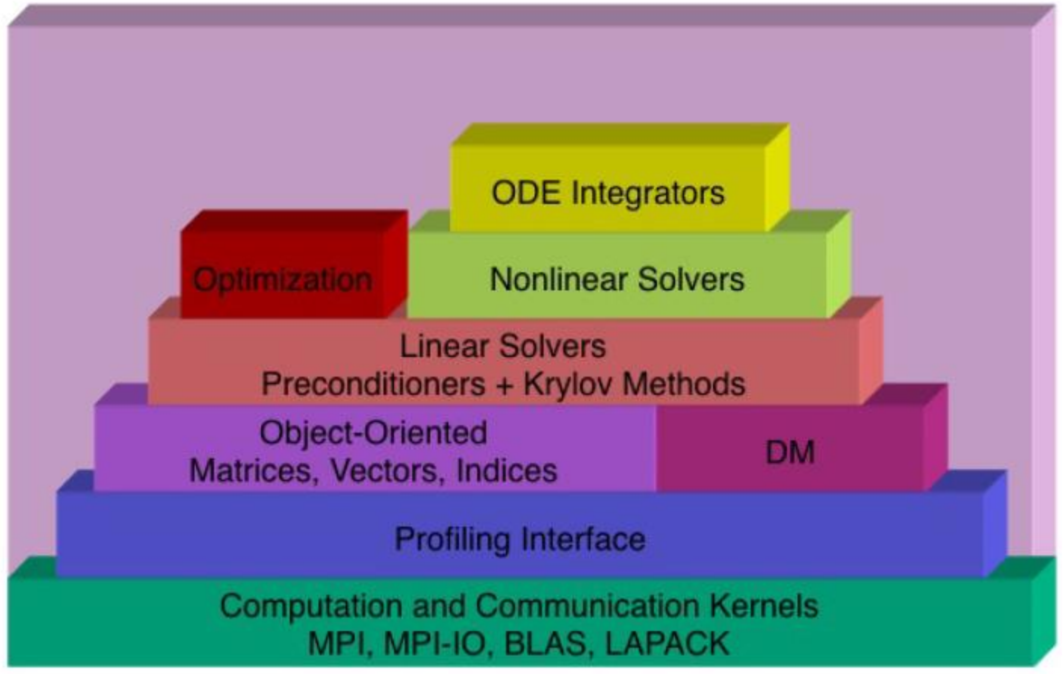
- Pitfalls:
  - Requirements need not state the solution methodology (e.g. algorithm), although they may
  - Requirements should not be given in terms of the program entitites (variables, classes, libraries, interfaces)

Parts of Software Construction: Architecture

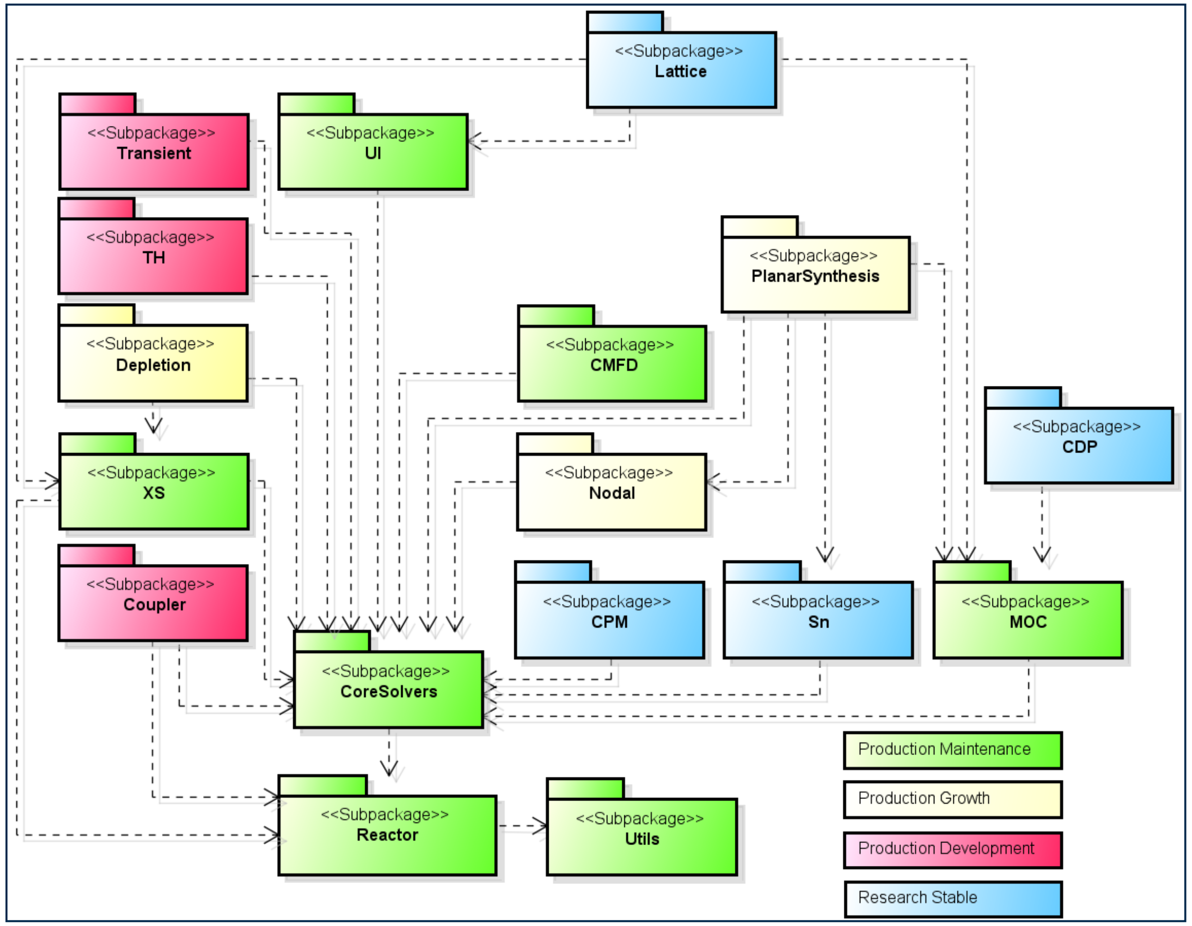
- Software Architecture is the highest level of software design.
  - The frame that holds the detailed parts.
    - Should be easily understandable
  - Work here generally overlaps with the "high-level" design.
- Key practice: Consider multiple designs.
  - In doing architecture design force yourself to develop more than one solution and compare them.
- Pitfall:
  - Architecture should be agnostic of programming language.

Examples:

PETSc Structure


ODE Integrators
Optimization
Nonlinear Solvers
Linear Solvers
Preconditioners + Krylov Methods
Object-Oriented
Matrices, Vectors, Indices
DM
Profiling Interface
Computation and Communication Kernels
MPI, MPI-IO, BLAS, LAPACK

Parts of Software Construction: Another Architecture Example

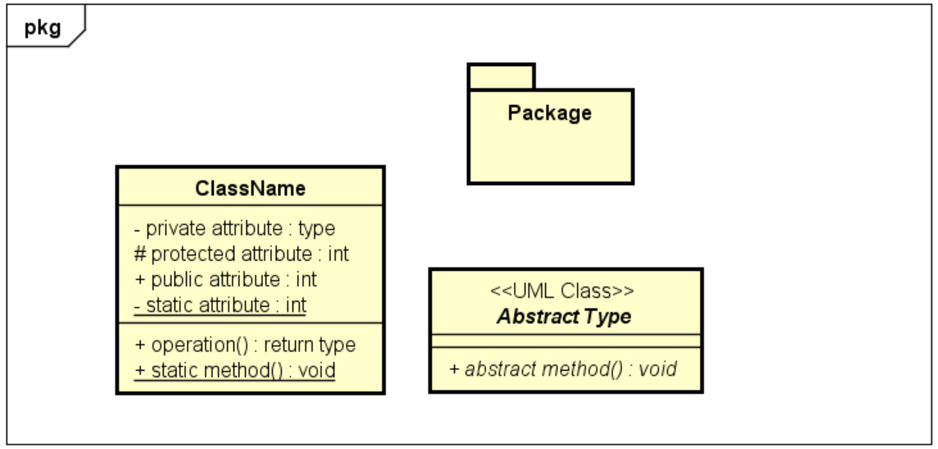


UML Overview

- The Unified Modeling Language (UML) is a family of graphical notations
  - Help in describing software systems, particularly object oriented system
  - A “meta”-language: a language that can describe languages (including itself!)
- Defined by a standard (OMGUML) http://www.omg.org/spec/UML/2.5/PDF
- Several programs allow you to create UML diagrams
  - Recommend astah: http://astah.net/ (free and professional versions)
  - PlantUML is another
- Can be used to quickly sketch ideas in discussion
- Can be used as a blue-print tool (design then code)
- Can be used to reverse engineer (code then document design)
- Accepted form of “technical documentation” for software design
  - Rather than conceptual

Classes (nouns)

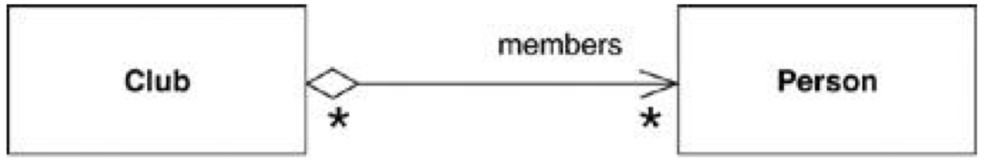
- Attributes `visibility name : type multiplicity = default`
  - variables
  - Format:
- Methods `visibility name (parameter-list) : return type`
  - procedures
  - Format:
- *Abstract*: no specific implementation
- Static: does not change
- Visibility: Public, Protected, Private
- Stereotyping



Associations (verbs)

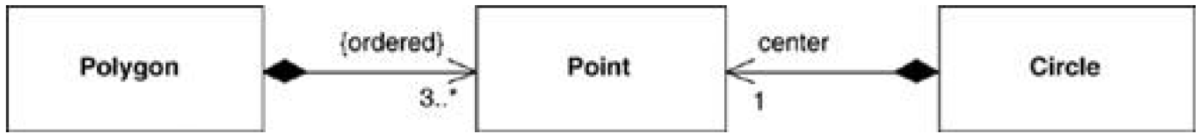
- Aggregation
  - Object A has Object B
  - Object B is a part of Object A

Figure 5.3. Aggregation

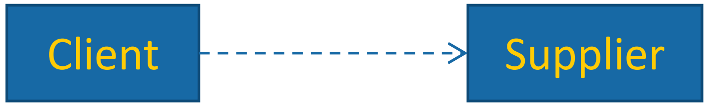


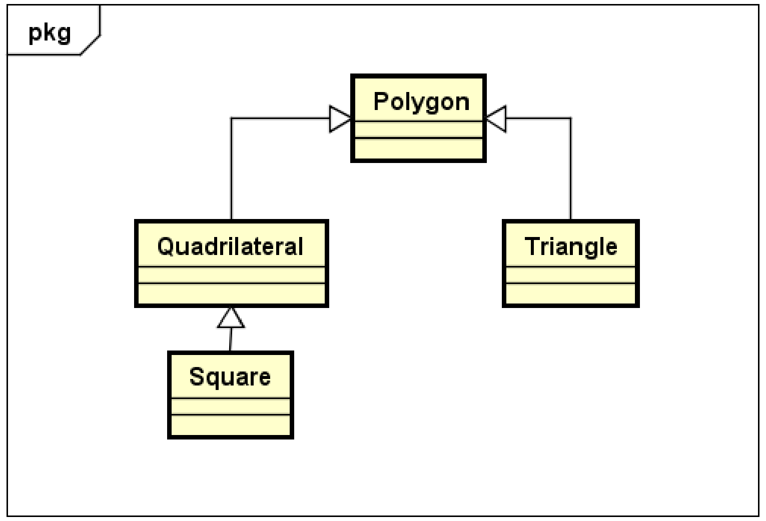
- Composition
  - Similar to aggregation, except
  - Implies exclusivity and coincident lifetimes

Figure 5.4. Composition



Dependency: "Client depends on supplier"





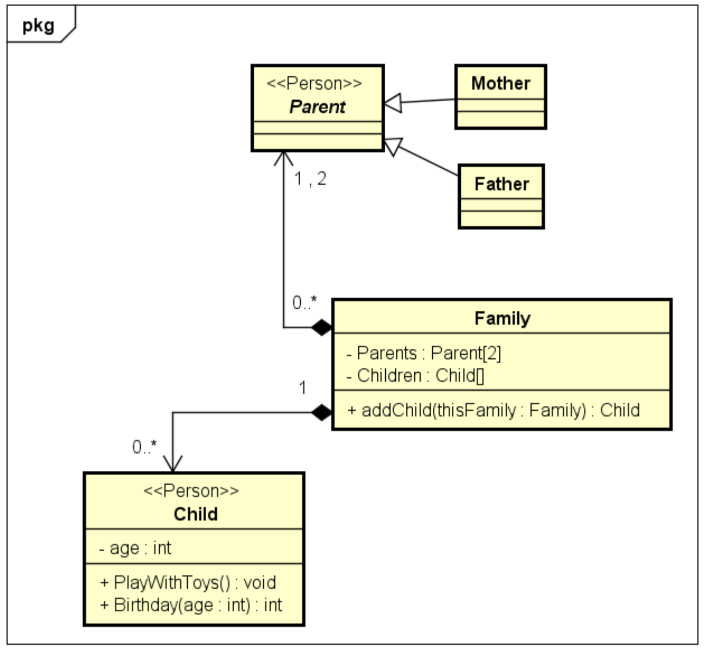
Generalizations represent inheritance

Figures from Martin Fowler, *UML Distilled*, 3rd Ed., Addison-Wesley, (2004).

Parts of Software Construction: High Level Design

- The high level design includes detailed design documentation that would describe
  - An Application Program Interface (API)
  - Class definitions
    - methods, attributes, inheritance
  - Dependencies within different parts of the program
  - State diagrams, sequence diagrams, activity diagrams (flow-charts)
- Key practices:
  - Consider multiple designs. force yourself to develop more than one solution and compare them.
  - The unified modeling language (UML) is an excellent tool for doing HLD.
- Pitfalls:
  - Difficult to keep up to date.
  - Creating unnecessary dependencies.

UML Example



High Level Design Defines Relationships
(between parts of program)

Parts of Software Construction: Low Level Design

- Low level design is the design that takes place "behind" an interface
  - e.g. "under the hood"
  - Often some choice of algorithm as well.
    - e.g. sort a data set
- Key Practice
  - Get very detailed requirements when you can.
  - Defensive programming / Design by Contract
- Pitfalls:
  - Poor performance
  - Memory leaks
  - Unnecessary dependencies
  - Bugs

Example: Homework 1

- Computation of the Z-curve index
  - Code I provided
  - Extra-credit using bit manipulation

Ok, I understand everything that's a part of Software Development

But, how do I use these parts effectively?

Development Workflows

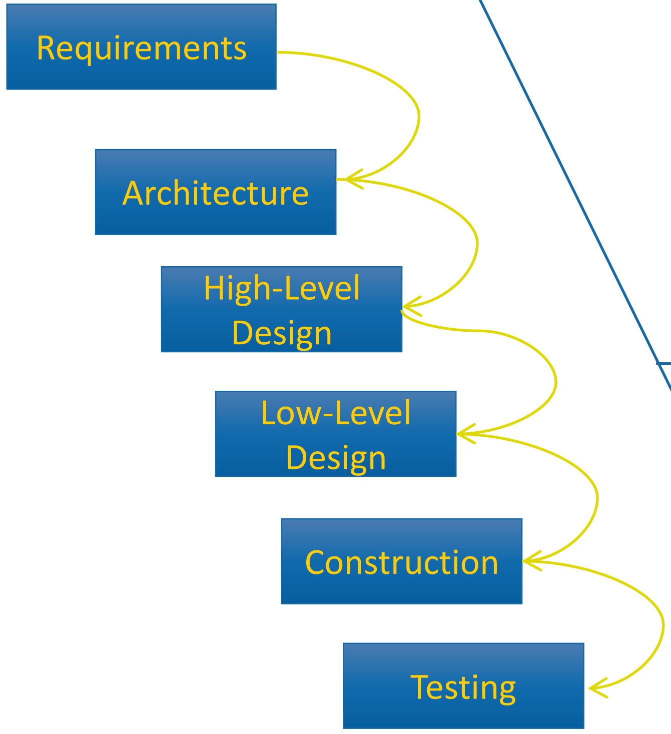
Development Workflows

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

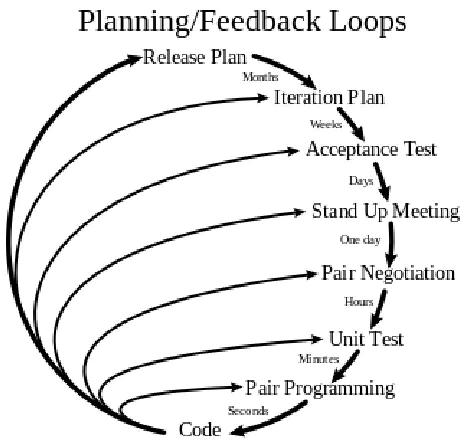
- Workflows are *based on a particular philosophy* or approach to software development.
  - Waterfall: Once through and your done
  - Incremental: Perform the same tasks in cycles
    - Spiral: Focused on risk management
  - Iterative:
    - Agile: Work in a way that lets you adapt quickly
      - Scrum, 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.

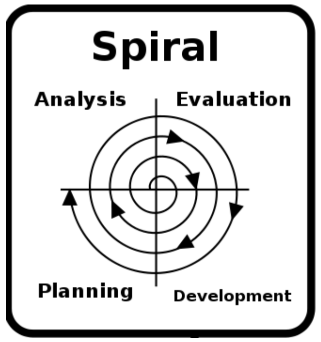
Workflow Illustrations

Waterfall

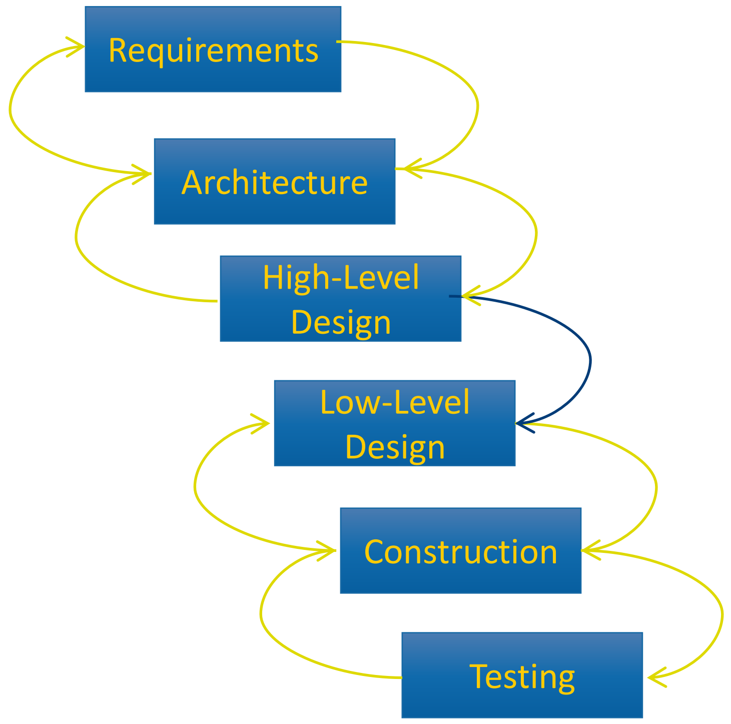


Extreme Programming





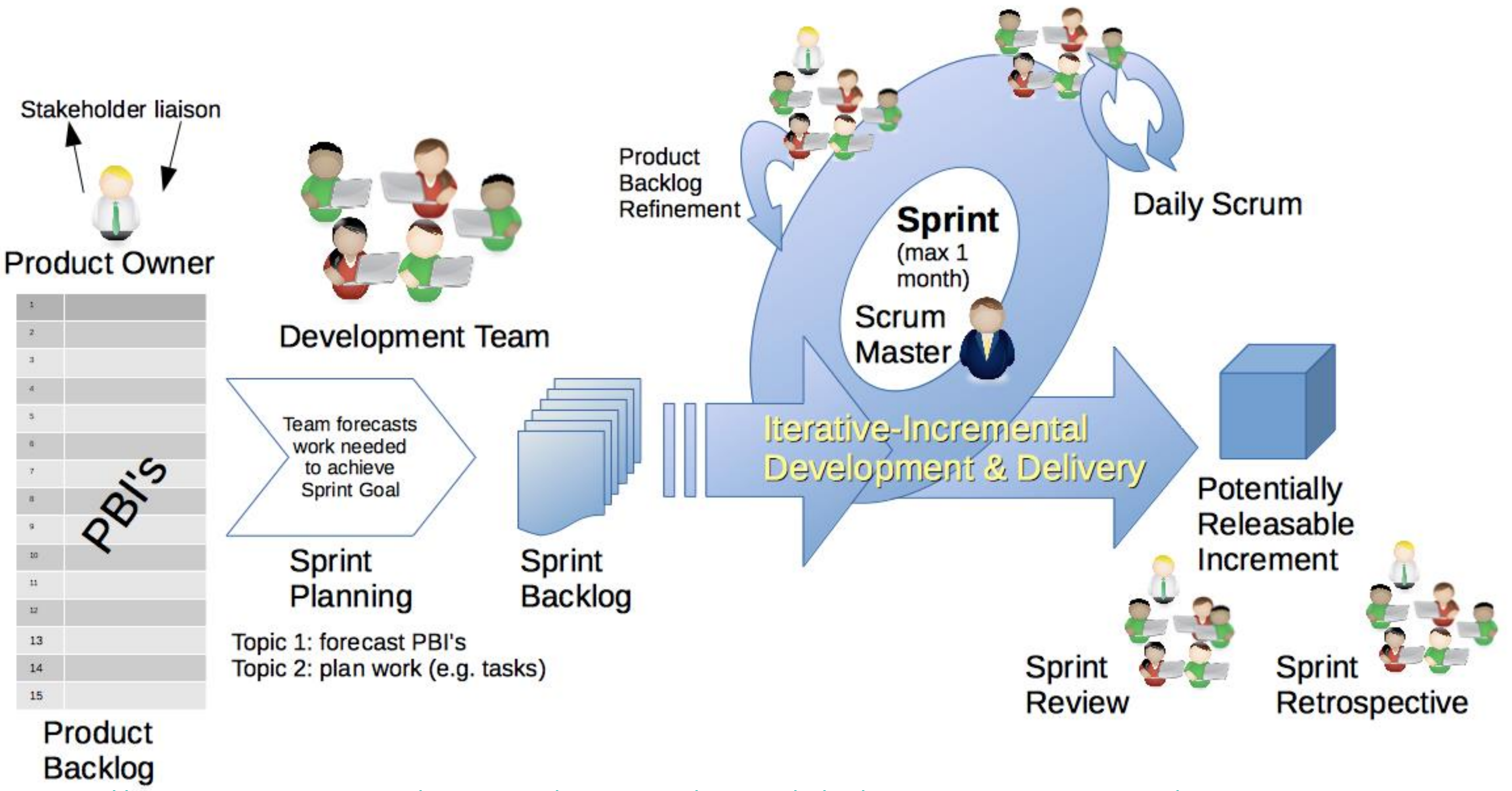
Agile Iterative



Other Workflow Concepts: Scrum

Concepts

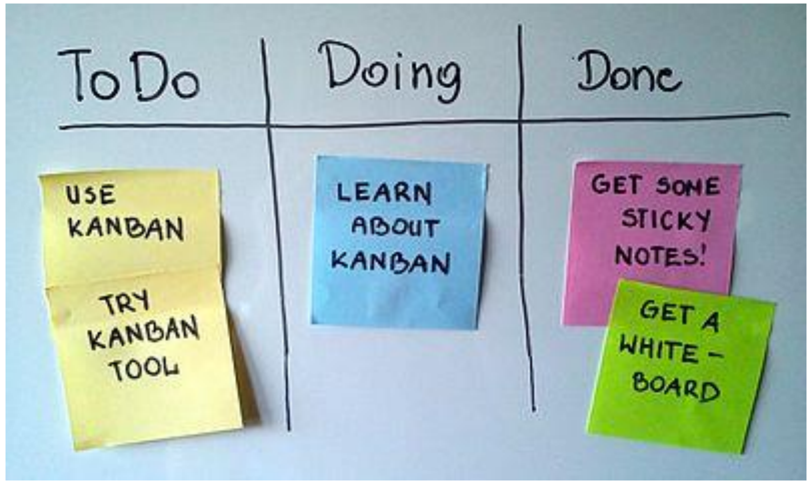
- Project roles
- Timeboxing
- Backlog
- Stand-ups
- Review and Retrospective



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

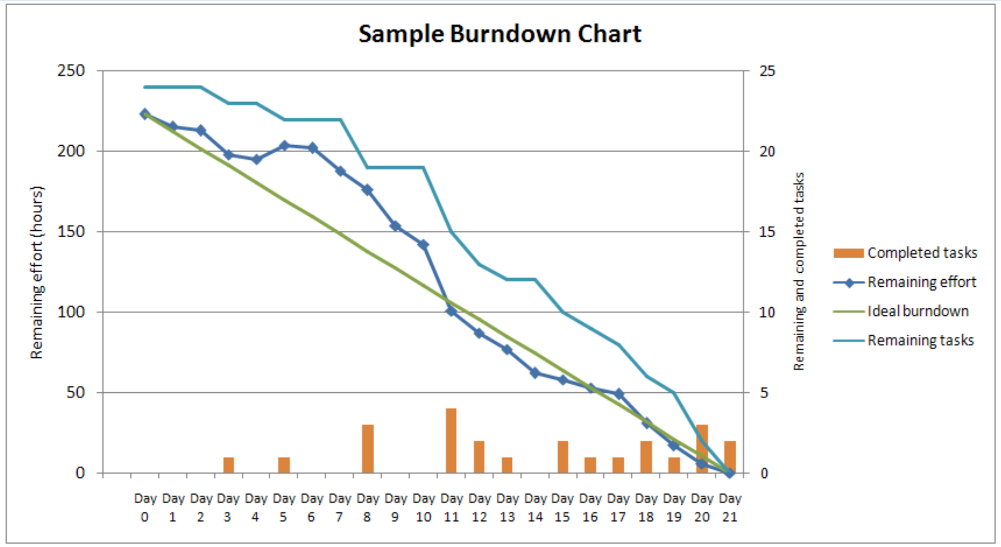
Other Workflow Concepts: Kanban & Burndown

Kanban Board



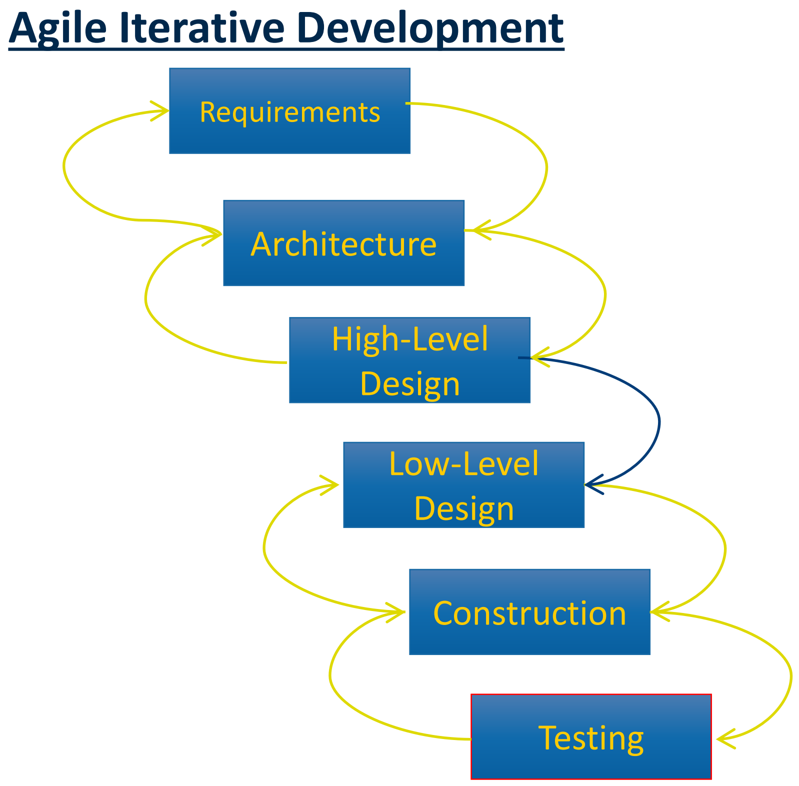
Concepts

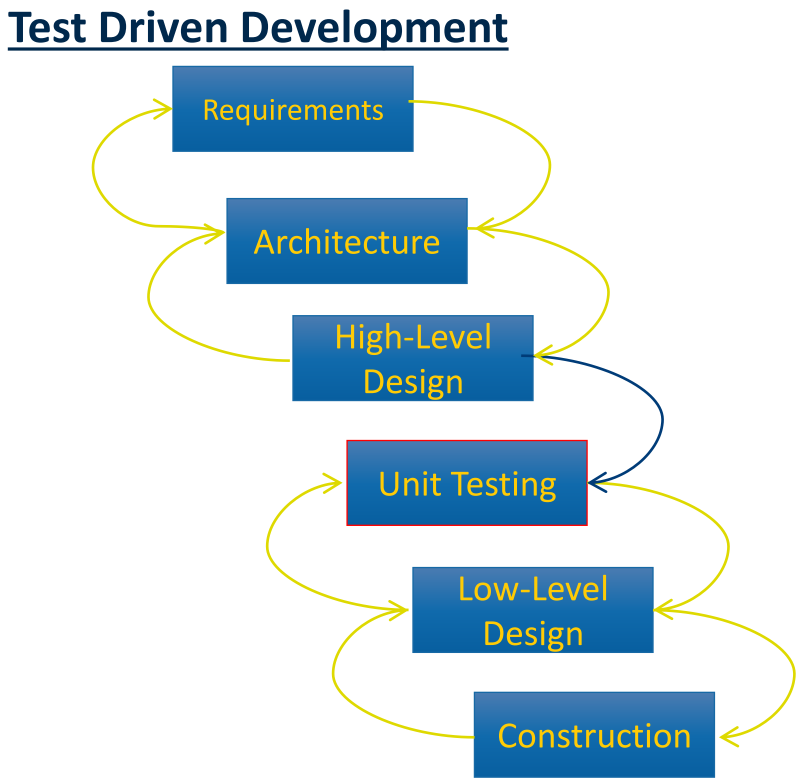
- Visualize Workflow.
- Limit work in progress.
- Evolve policies.
- Burndown Charts.



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

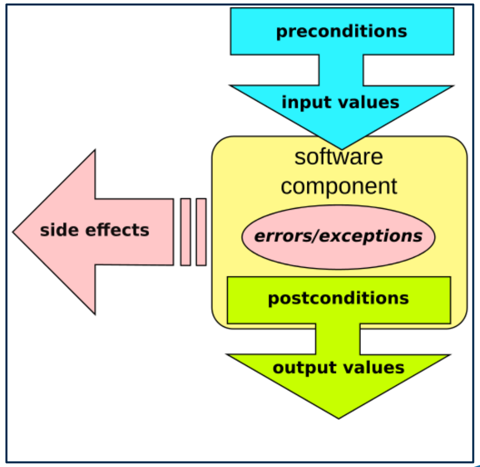
Test Driven Development





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
    - 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
```

https://en.wikipedia.org/wiki/Design_by_contract#/media/File:Design_by_contract.svg

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

Software Lifecycle Model

What is it?

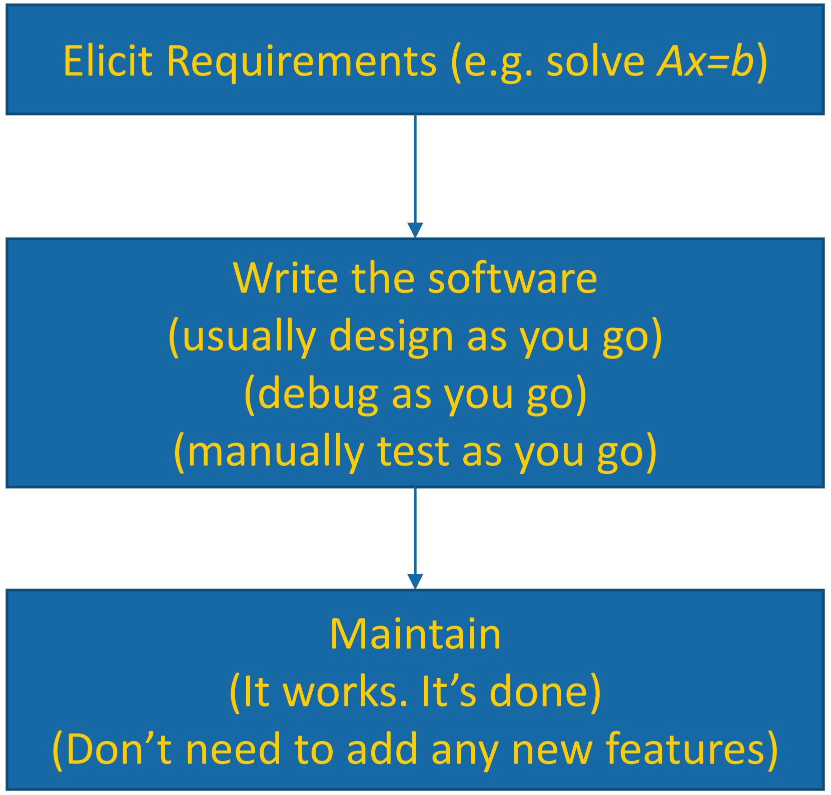
- The model *used to decide when* to perform particular development activities
- Implicit to all software projects
  - Not necessarily formally defined.
- Much better to have a formally defined lifecycle model.
  - Will define “maturity levels”
  - Also defines what activities to perform at each level

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

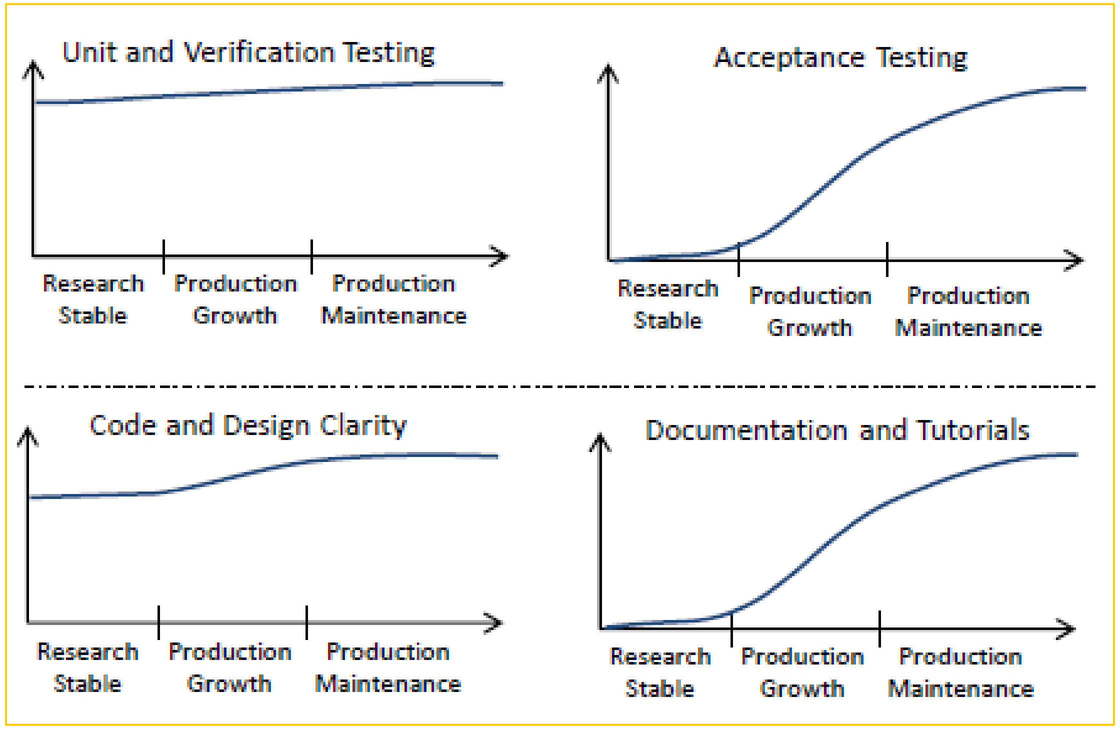

Elicit Requirements (e.g. solve *Ax=b*)
Write the software (usually design as you go) (debug as you go) (manually test as you go)
Maintain (It works. It's done) (Don't need to add any new features)

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)



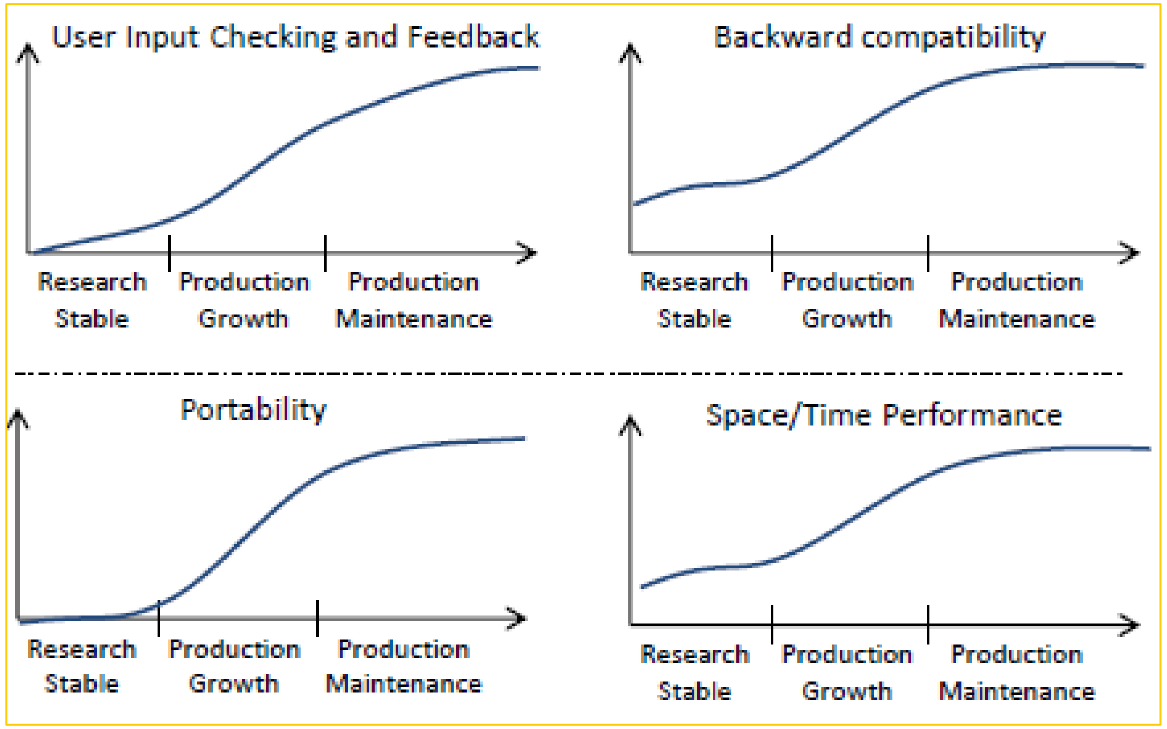
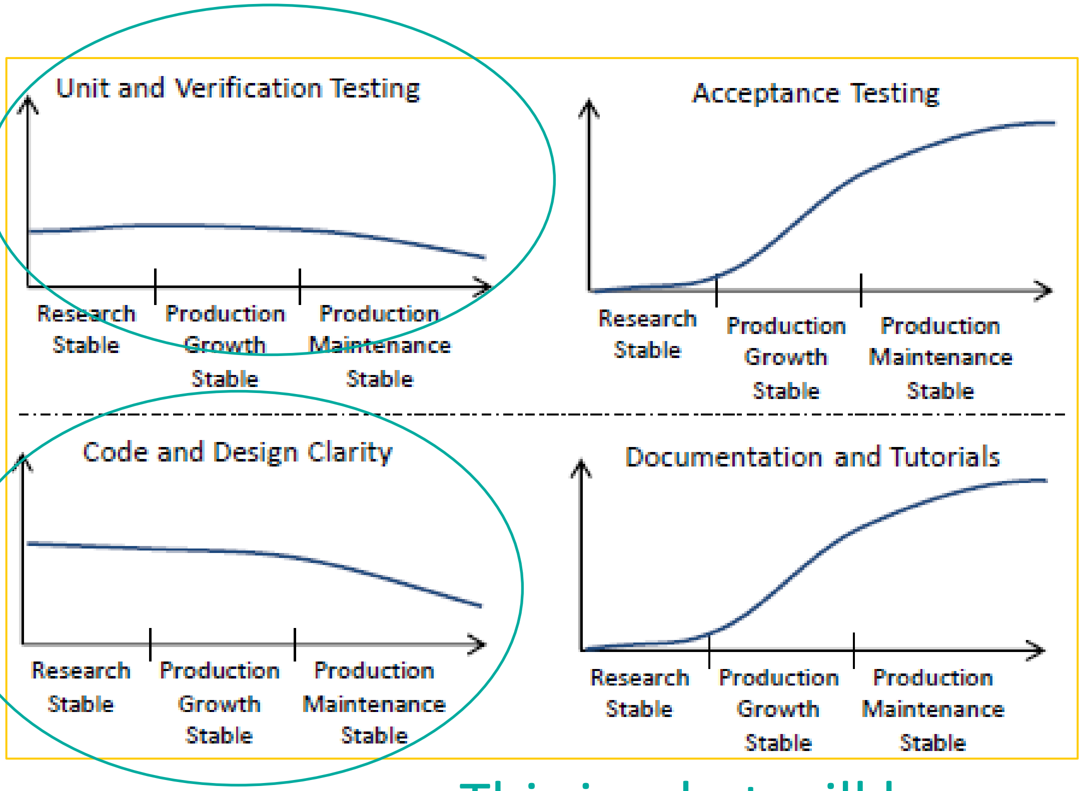
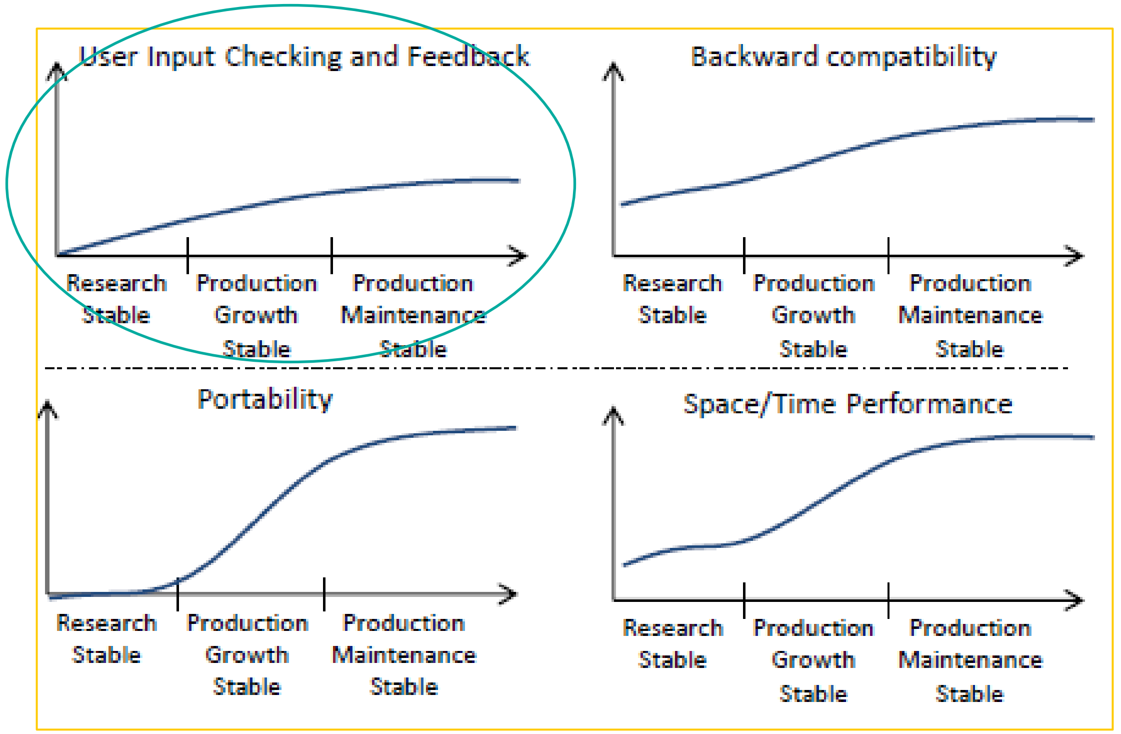


Figure 1. “Typical levels of various production quality metrics in the different phase of the proposed Lean/Agile-consistent TriBITS lifecycle model”
From R. Bartlett, et al., “TriBITS Lifecycle Model” Version 1.0,” SAND2012-0561, (2012)

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 of Lifecycle

- 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

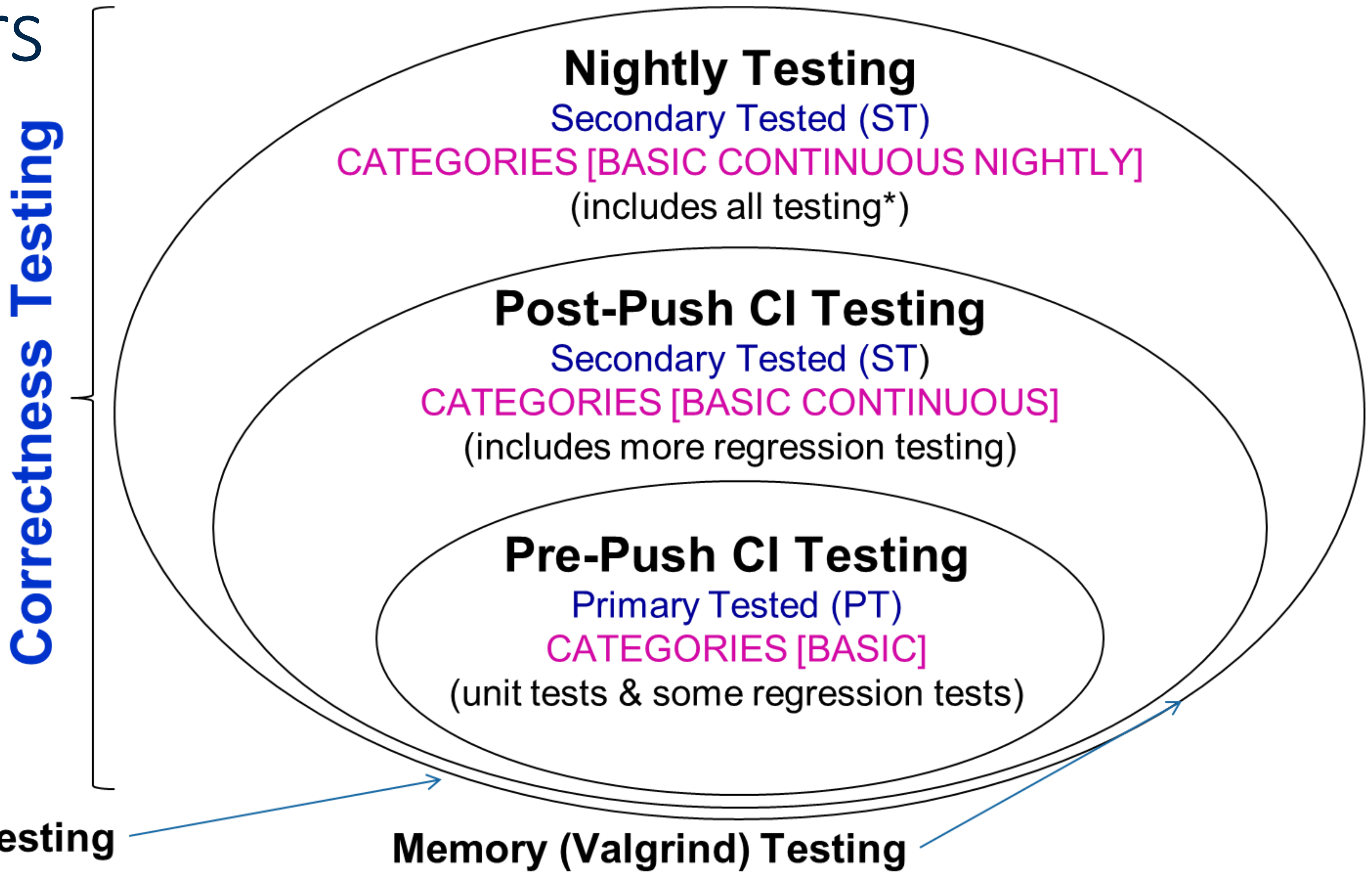
Software Processes on GitHub

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



*Additional Categories:
Heavy or Weekly

AUTOMATE TESTING AS MUCH AS YOU CAN!