Lecture 11 – Object Oriented Programming

Prof. Brendan Kochunas

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

COLLEGE OF ENGINEERING
NUCLEAR ENGINEERING & RADIOLOGICAL SCIENCES
UNIVERSITY OF MICHIGAN

Outline

- Recap Software Lifecycles
- Testing
- Github example of software processes
- Object Oriented Programming and UML

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

- (Knowledge) list and define common types of testing categories
- (Skill) implement software engineering best practices in github
- (Skill) draw a UML Class diagram related to a computational science application
- (Knowledge) provide definitions of fundamental concepts in OO programming
  - (Skill) code simple examples of these concepts.

Software Lifecycles

Recap of Software Lifecycles

Overview

- The model *used to decide when* to perform particular development activities
- Enables
  - exploratory research to remain productive
  - reproducible research
  - overall development productivity
  - communication of maturity levels/expectations

Summary

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

Maturity of Software Quality Metrics (Ideal)

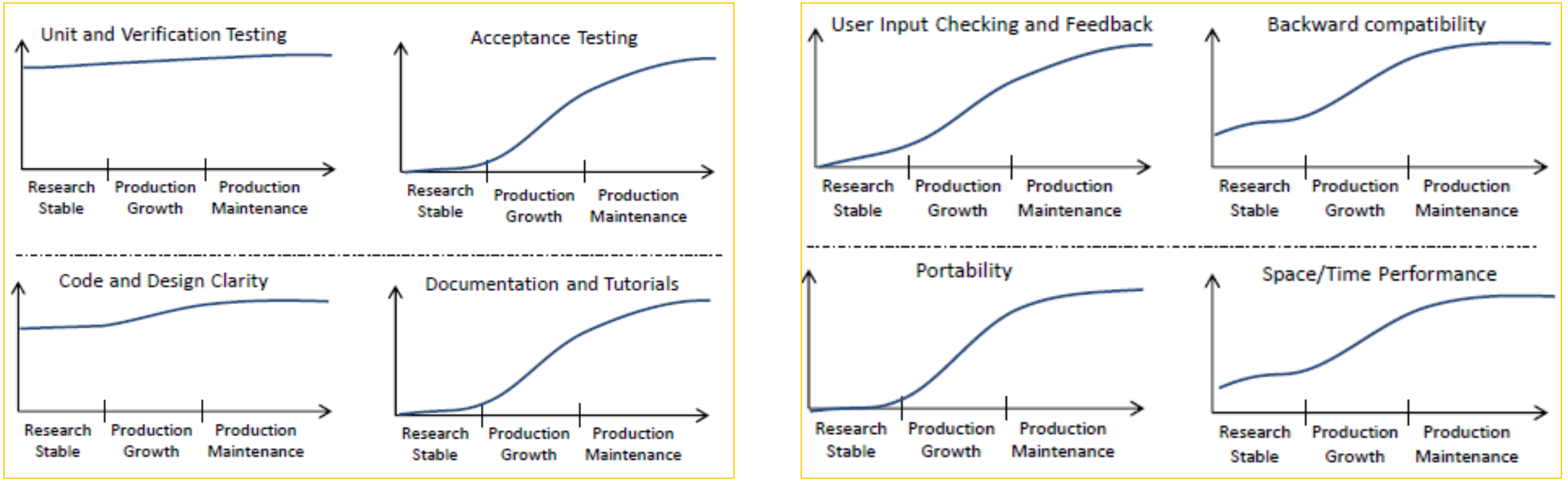


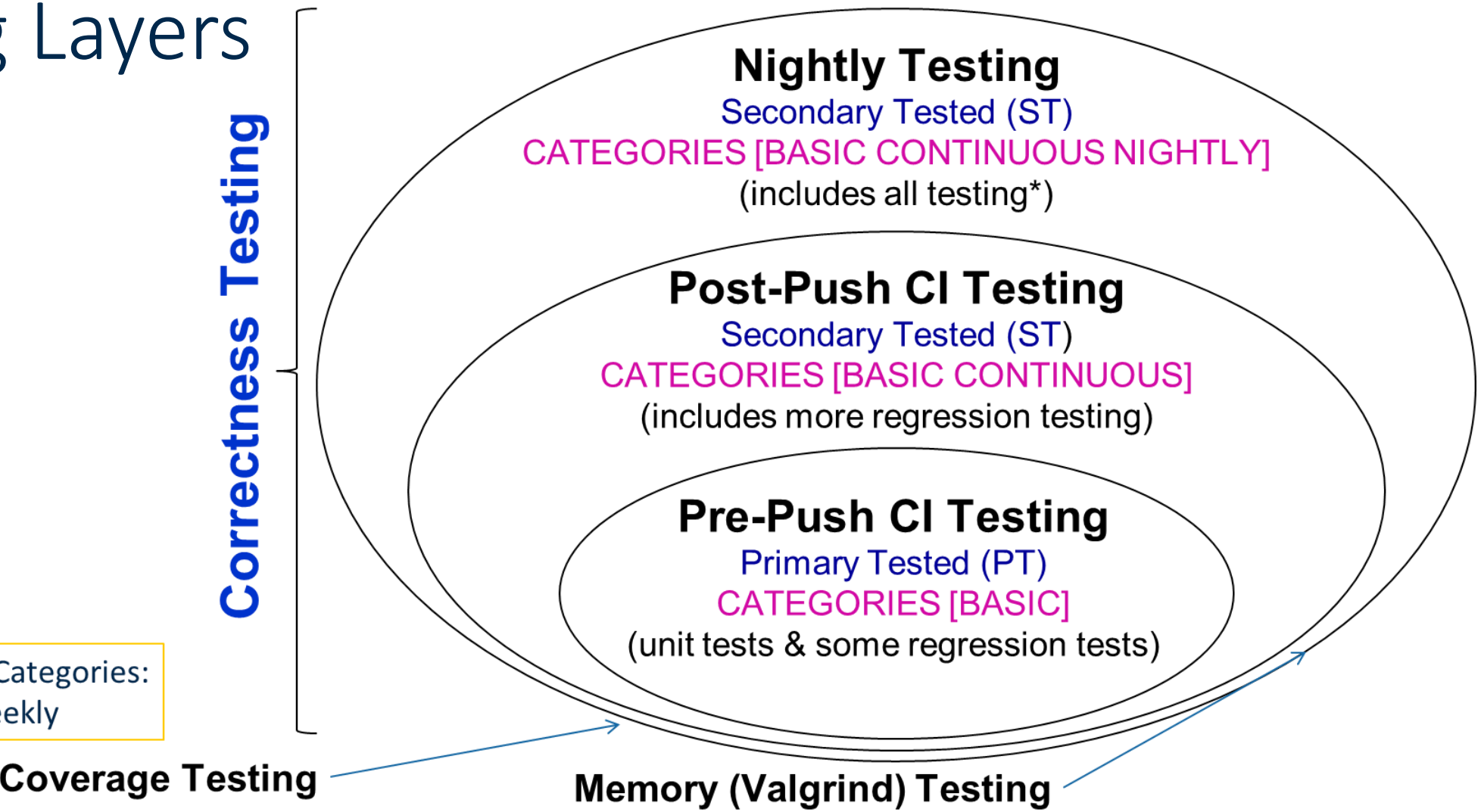
Figure 1. "Typical levels of various production quality metrics in the different phase of the proposed Lean/Agile-consistent TriBITS lifecycle model"
From R. Bartlett, et al., "TriBITS Lifecycle Model" Version 1.0," SAND2012-0561, (2012)

Ross's Taxonomy of Testing

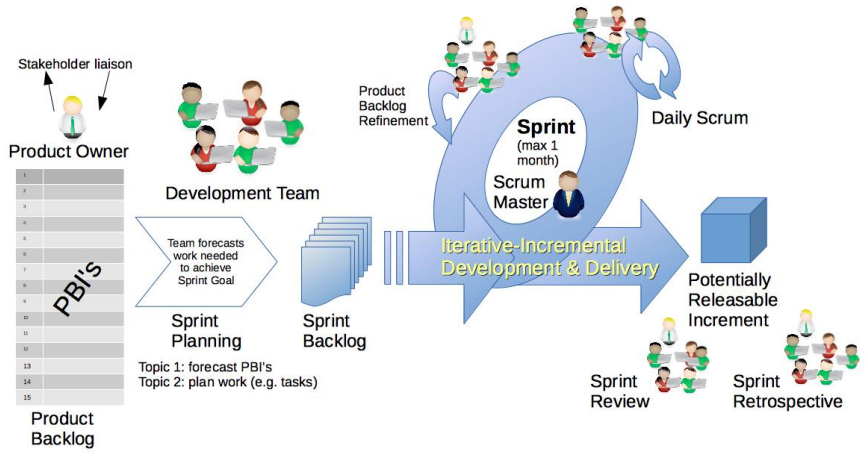
A Taxonomy of Testing

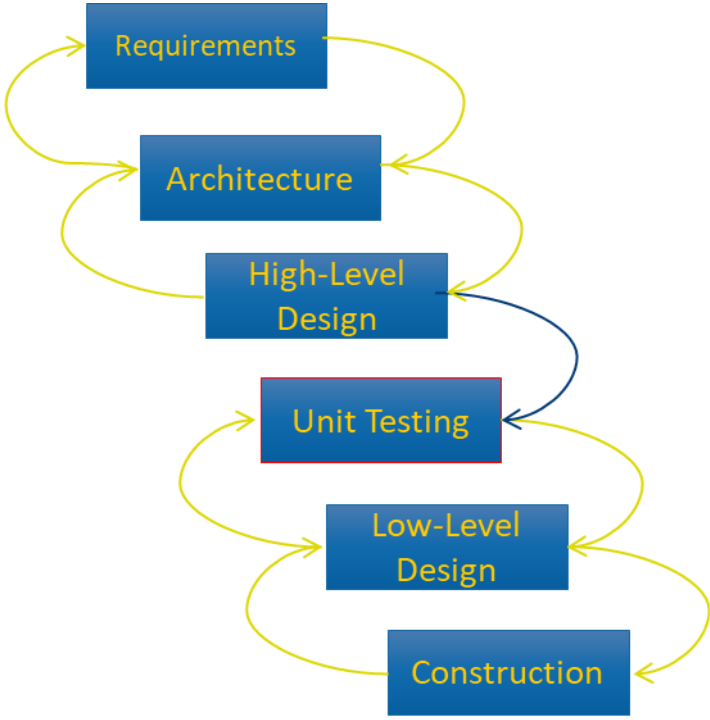
- Testing is the backbone of software quality assurance (SQA).
- Types of testing
  - *Unit Testing* – Test individual units of program *in isolation*
    - Should run very fast: < 1 second (a couple seconds is ok)
  - *Integral Testing* – Testing program components together
    - Should run fast: < 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!





Software Processes on GitHub

https://github.com/Reference-LAPACK/lapack

Object-Oriented Programming

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”?
- Lots of similar terminology that gets confusing.
  - Think like a philosopher--terms have very specific definitions within a specific context (project, discussion, paper, etc.

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

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.

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

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

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

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)

Abstract vs. Concrete Objects

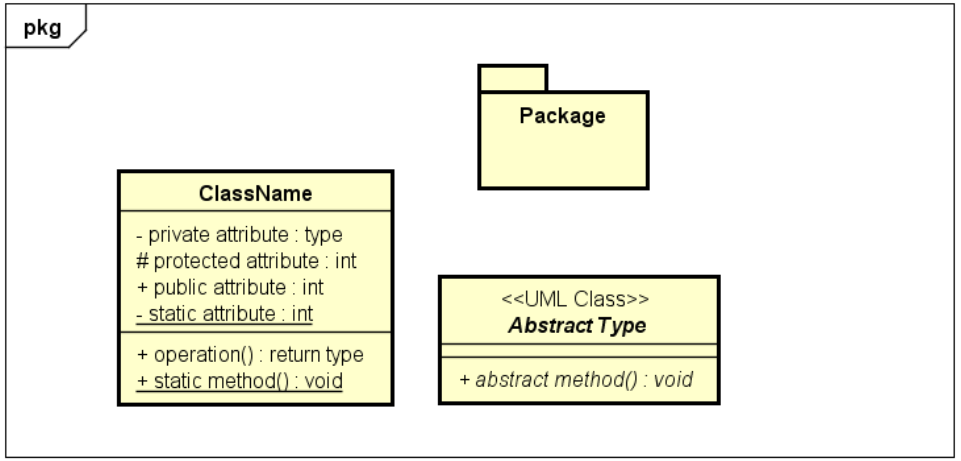
- Abstract objects may have incomplete definitions
- Abstract methods have no defined implementation (e.g. low-level design)
  - Just an interface definition
- Abstract methods can only be defined on abstract objects
- Abstract types can be used to define requirements of concrete types

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)

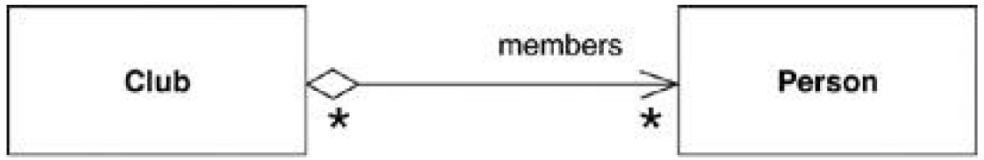
- Attributes `visibility name : type multiplicity = default`
  - variables
  - Format:
- Methods `visibility name (parameter-list) : return type`
  - procedures
  - Format:
- *Abstract*: no specific implementation
- Static: does not change
- Visibility: Public, Protected, Private
- Stereotyping



Associations (verbs)

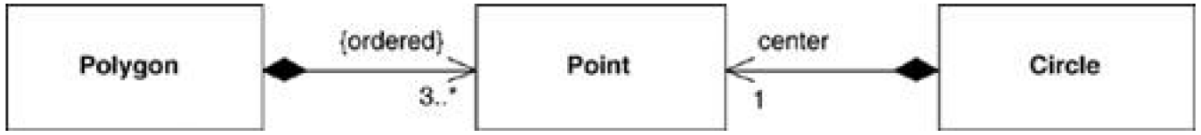
- Aggregation
  - Object A has Object B
  - Object B is a part of Object A

Figure 5.3. Aggregation



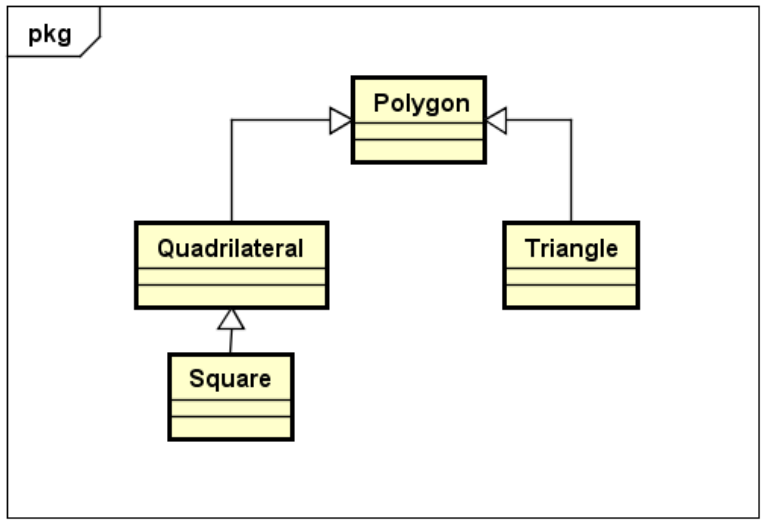
- Composition
  - Similar to aggregation, except
  - Implies exclusivity and coincident lifetimes

Figure 5.4. Composition



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

Dependency: "Client depends on supplier"



Generalizations represent inheritance

Example: Linear Solver Class

- Techniques for solving linear systems
  - Direct
    - LU
  - Iterative
    - Fixed point
      - Gauss Siedel
      - Jacobi
    - Krylov
      - GMRES

Sequence Diagram

- Sequence diagrams describe the behavior of a system.
  - Usually a single scenario
  - Illustrates order of operations
    - Notion of a “call stack”
  - Not great for showing loops or branching behavior
    - there are different diagrams for that

Time

Other types of UML Diagram

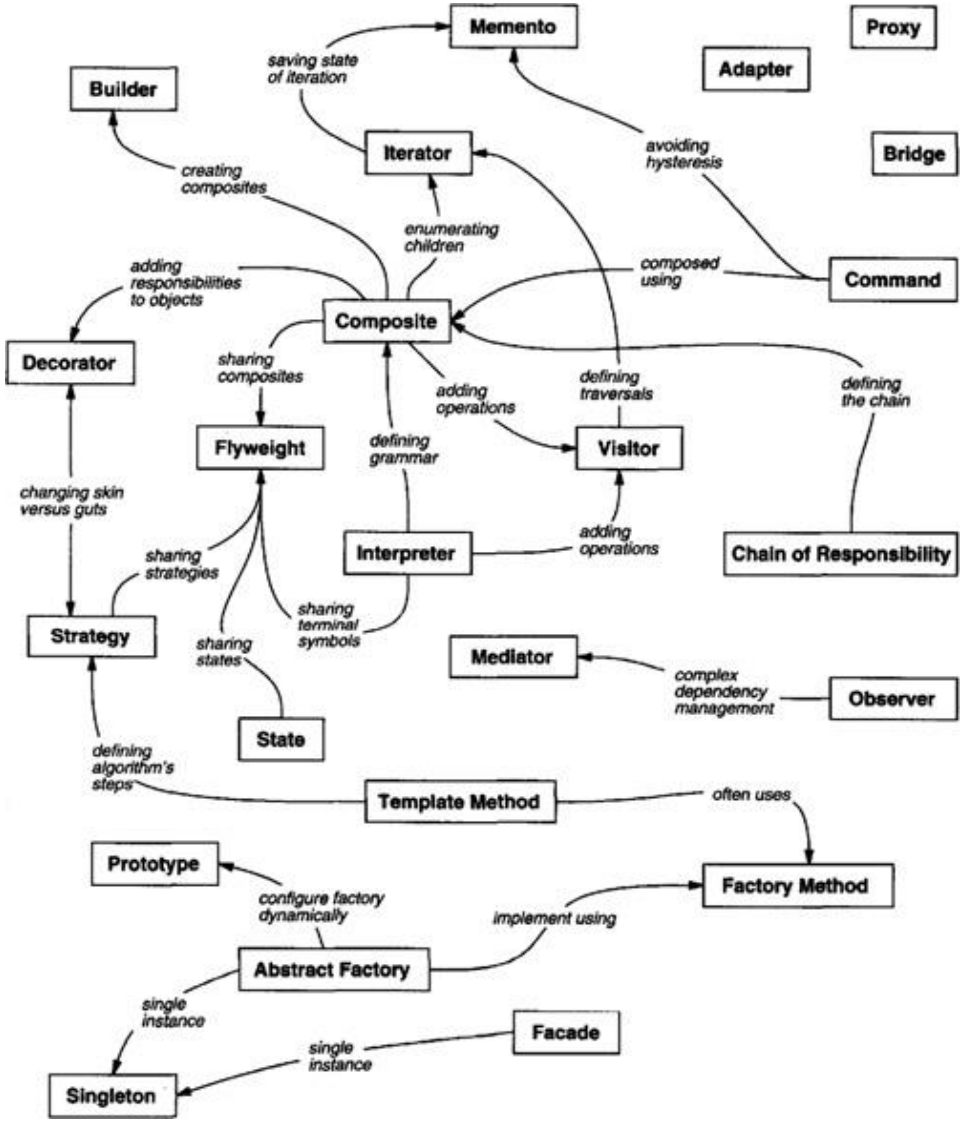
- *Object diagram*
  - Snapshot of the system at a point in time
  - Shows instances rather than classes
  - useful for supplementing class diagrams to illustrate specific examples
- *Package diagram*
  - For large scale systems, architecture, and illustrating dependencies
- *State diagram*
  - Used to illustrate behavior
  - Show possible states of an object and how objects transitions between states
- *Use case diagram*
  - Technique for capturing functional requirements
- *Activity diagrams*
  - Describe procedural logic or workflow
  - basically a flow-chart
- *Deployment diagrams*
  - Show a system's physical layout
  - Relates software to hardware
- *Communication diagrams*
  - Used to illustrate data dependencies between components
  - Similar to sequence diagrams, but emphasis is on what data is exchanged, not order of operations

Design Patterns

- Idea of design patterns is that many programming design problems are the same, and therefore have similar design solutions
  - *Common design solutions are called **design patterns**.*
- Key to using design patterns is understanding your design problem and being aware of patterns
- Different Types of Patterns

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Figure 1.1 and Table 1.1 from E. Gamma, et. al,
Design Patterns: Elements of Object-Oriented Software, Addison-Wesley, (1994).



Abstract Factory Pattern

- Factories create instances of objects
- Use abstract factory when
  - a system should be independent of how products are created
  - configured with one of multiple families of products
  - want client dealing with base classes
- Example: Automobiles

Make a door Car Part (door)

```
CarFactoryCreate(          )
```

Make the hood Car Part (hood)

```
CarFactoryCreate(          )
```

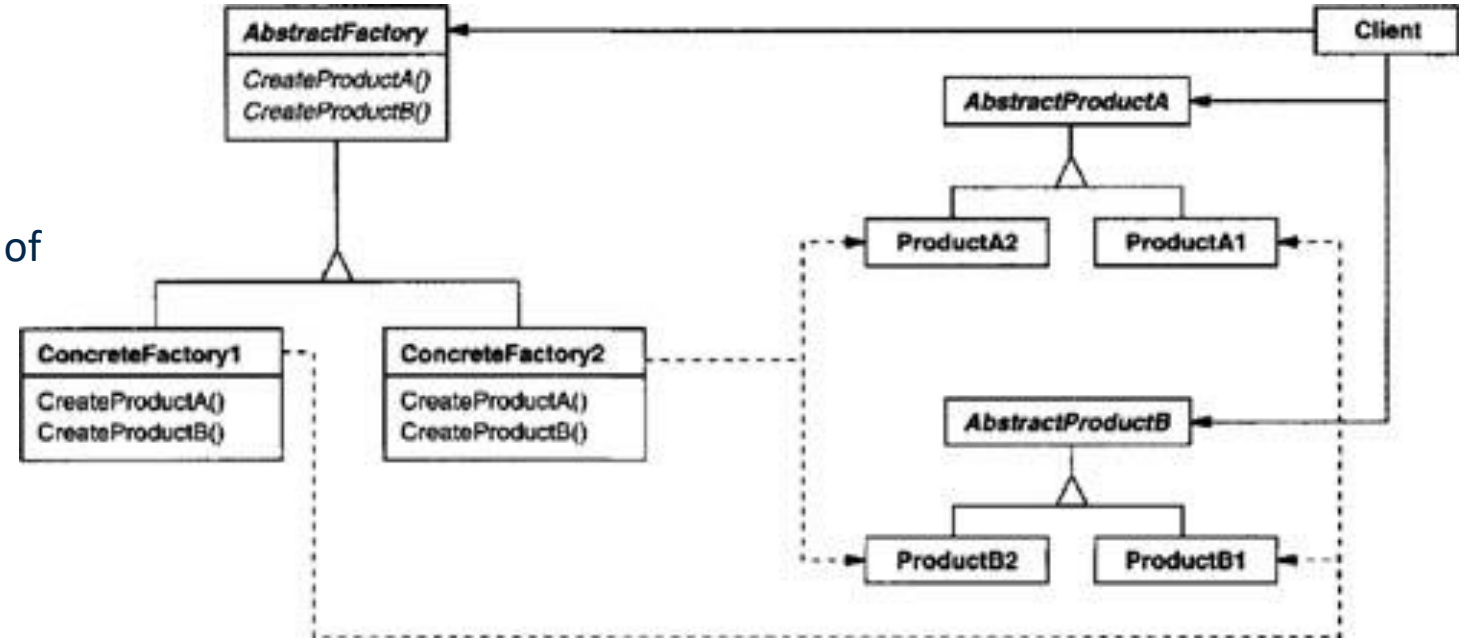


Figure from E. Gamma, et. al, *Design Patterns: Elements of Object-Oriented Software*, Addison-Wesley, (1994).

Adapter Pattern

- Also known as a “wrapper”
- Use it when you need to convert interface of one class to another
- Usually “lightweight” and can be scripted
- Example
  - Module to declare Fortran interfaces that bind to C interfaces
  - Wrapping multiple third party libraries

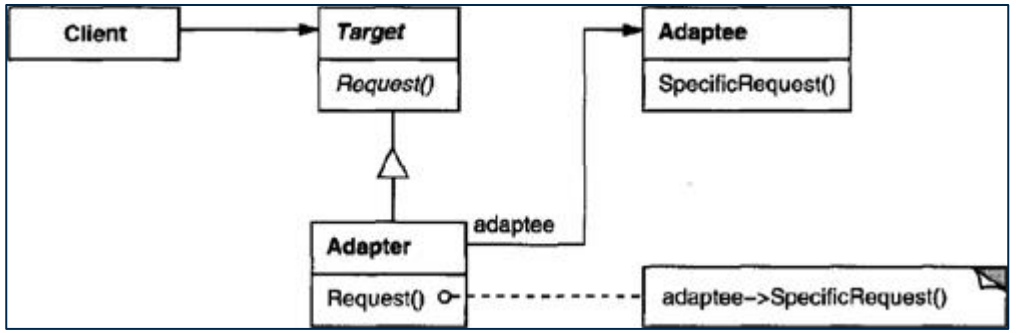


Figure from E. Gamma, et. al, *Design Patterns: Elements of Object-Oriented Software*, Addison-Wesley, (1994).

Composite Pattern

- In scientific computing
  - useful for representing problem geometry and mesh

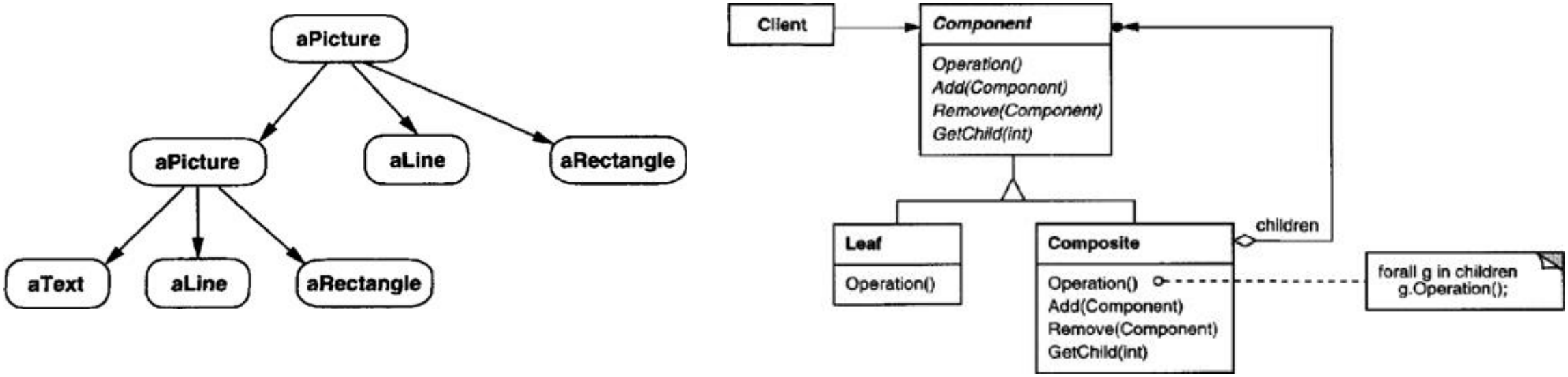
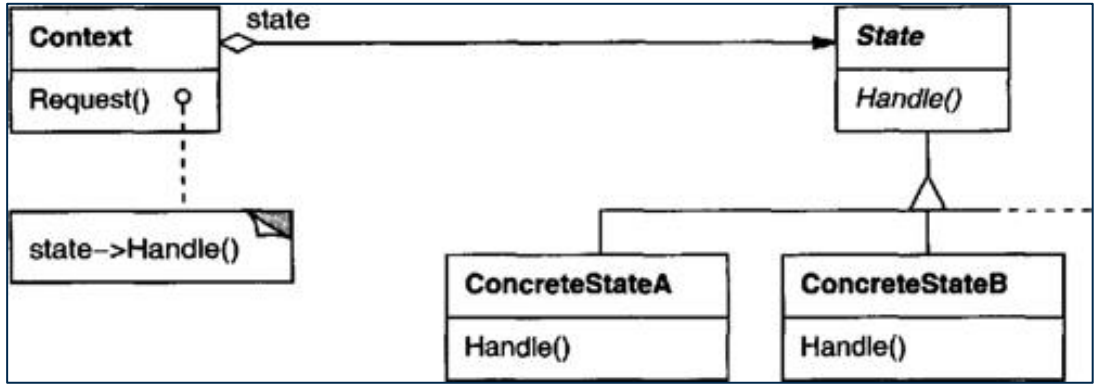


Figure from E. Gamma, et. al, *Design Patterns: Elements of Object-Oriented Software*, Addison-Wesley, (1994).

Behavioral Patterns

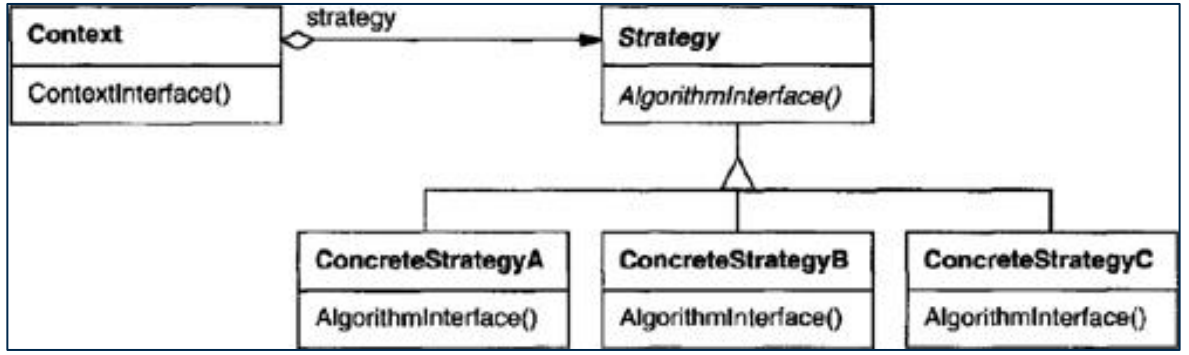
State Pattern

- Allow an object to alter its behavior when internal state changes
- Example: Output
  - control verbosity
  - edit to a file
  - edit to standard out



Strategy Pattern

- Define a family of algorithms that are encapsulated and interchangeable.
- Example: Linear solvers
  - if small, solve directly
  - if large, solve iteratively
  - if sparse use Krylov methods



Figures from E. Gamma, et. al, *Design Patterns: Elements of Object-Oriented Software*, Addison-Wesley, (1994).