DETAILED DESIGN

Dr. Tony D. Barber

Fall, 2024

TODAY'S CLASS OBJECTIVES AND OUTCOME

- Explain the *process* of developing the detailed design
- Use appropriate *UML notations* for documenting detailed design
- Apply *design principles* for developing understandable, maintainable, and reusable design
- Understand *design patterns* and their applicability, advantages and drawbacks
- List the content of the Software Design Document

OUTLINE

- Software detailed design
  - Process
  - Models and their representation
  - UML notations used to represent detailed design
  - Construction techniques and principles
  - Design Patterns
  - Operations and data specification
  - Software design document
- Assignments

RESOURCES

- **Books**
  - Textbook – Chapters 11-19
  - *Design Patterns*, by E. Gamma, R. Helm, R. Johnson, J. Vlisides
  - *Head First Design Patterns*, by Eric Freeman and Elizabeth Robson
  - *Software Engineering: Theory and Practice*, by Pfleeger and Atlee
  - Clean Architecture, by Robert C. Martin (Uncle Bob)
  - Practical Object-Oriented Design: An Agile Primer Using Ruby, by Sandi Metz
  - Code Complete - 2nd edition, by Steve McConnell
  - Clean Code, by Robert C. Martin (Uncle Bob)

RESOURCES

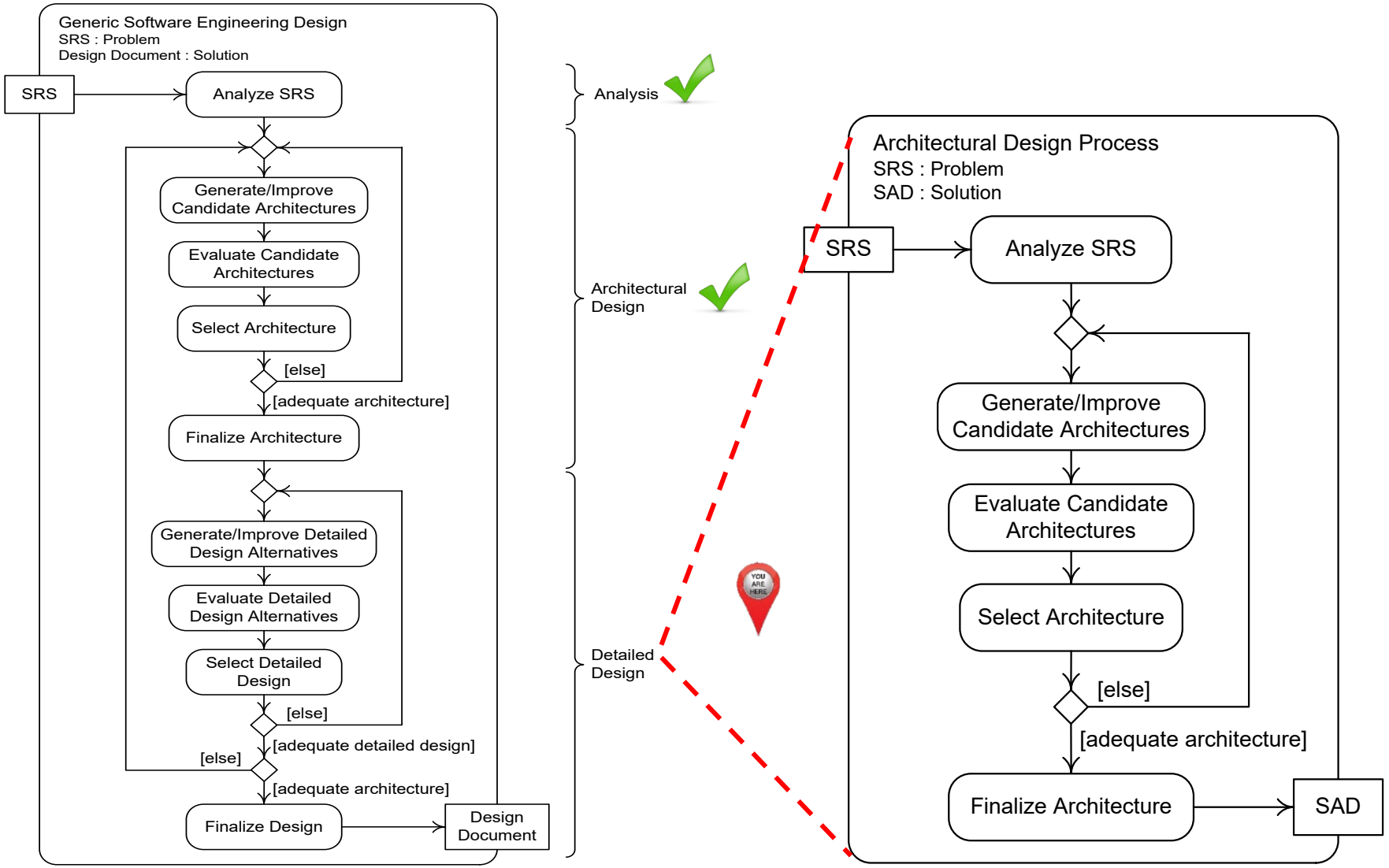
- Design Principles:
  - http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
  - http://www.blackwasp.co.uk/SOLID.aspx
  - https://www.codeproject.com/Articles/703634/SOLID-architecture-principles-using-simple-Csharp
  - http://www.oodesign.com/design-principles.html
  - Watch (**start at min 12**): Bob Martin SOLID Principles of Object Oriented and Agile Design
  - Watch Core Design Principles for Software Developers (Venkat Subramaniam)

RESOURCES

- Design patterns with Java code
  - https://newcircle.com/bookshelf/java_fundamentals_tutorial/design_patterns
  - https://www.tutorialspoint.com/design_pattern/index.htm
  - http://www.journaldev.com/1827/java-design-patterns-example-tutorial
  - http://www.oodesign.com
  - http://www.javatpoint.com/core-java-design-patterns_
  - https://sourcemaking.com/design_patterns

- YouTube design patterns tutorials:
  - https://www.youtube.com/watch?v=vNHpsC5ng_E&list=PLF206E906175C7E07
  - https://www.youtube.com/watch?v=v9ejT8FO-7I&list=PLrhzvIcii6GNjpARdnO4ueTUAVR9eMBpc
  - https://www.youtube.com/watch?v=NU_1StN5Tkk

SOFTWARE DESIGN PROCESS



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

OUTLINE

- Lecture
  - We are here
  - Improve architecture solution
  - Evaluate architecture
    - Architecture analysis/evaluation (scenario based)
    - Prototyping
  - Select architecture
  - Finalize architecture
    - Software Architecture Document (SAD)
  - Software design in maintenance and evolution
  - Architecture reuse and product lines
- Assignments

WHAT ARE DESIGN PATTERNS

- Design Patterns
  - Reusable solutions to common problems that arise in software design, such as object creation, communication, behavior, and structure.
  - Design patterns are not specific to any language or framework, but rather describe general concepts and best practices that can be applied in different contexts.
  - Some examples of design patterns are singleton, factory, observer, strategy, and decorator.
  - Descriptors:
    - Address specific issues and challenges within the system.
    - More focused on the functional requirements, such as functionality, logic, and behavior.
    - Can be applied and refined throughout the development process.
    - Not rules or recipes that you must follow blindly.
    - Guidelines and suggestions that you can adapt and modify to fit your specific situation and needs
    - Should not force a design pattern, where it doesn't make sense or add value.
    - Use design patterns to solve real problems and improve the quality and readability of your code.

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

NOTE ON TEXTBOOK TERMINOLOGY

- Detailed design consists of two activities
  - Mid-level design
    - The activity of specifying software at the level of medium-sized components, such as compilation units or classes, and their properties, relationships, and interactions
  - Low-level design
    - The activity of filling in the small details at the lowest level of abstraction

- We will not use *mid-level design* and *low-level design* terminology, just *detailed design* that encompasses both

DETAILED DESIGN CONCEPTS

- Detailed design is the activity of ***specifying*** for all major program parts (architectural components):
  - Their **internal elements**
  - Their **structure**, **relationships**, and **processing**
  - Often, their **algorithms** and **data structures**

- Detailed design
  - Levels of abstraction: from architecture to coding
  - The boundaries can be fuzzy between
    - Architectural design and detailed design
    - Detailed design and programming

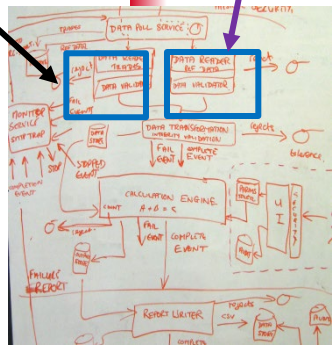
ARCHITECTURE AND DETAILED DESIGN

Architectural design

- The activity of specifying
  - A program's **major parts (elements)**
  - Their responsibilities, properties, and interfaces, and
  - The **relationships** and **interactions** among them
- "High-level" design
- Global, upfront, major design decisions (e.g., technologies, configurations, OS, reuse, styles)
- More abstract

Detailed design

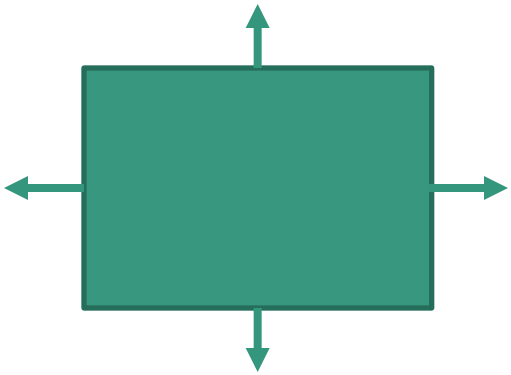
- The activity of specifying
  - The ***internal elements*** of **all major program parts (elements)**
  - Their structure, relationships, and processing, and
  - Often, their algorithms and data structures
- "Mid-level" and "low-level" design
  - Low level detailed design "shades" into coding
- Local decisions, made after architecture decisions
- More detailed



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

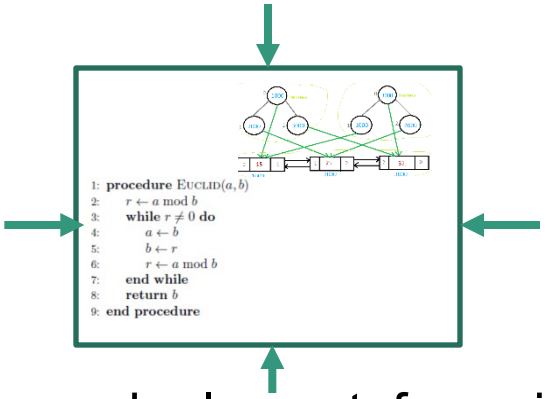
ARCHITECTURE AND DETAILED DESIGN (CONTINUED)

Architectural design



- For each element, focus outwardly:
  - Assigned responsibilities
  - Systemic properties
  - Relations and interfaces
  - Behavior (within the system)

Detailed design



- For each element, focus inwardly:
  - Element properties (data and behavior)
    - Data structures and algorithms
  - Details closer to implementation

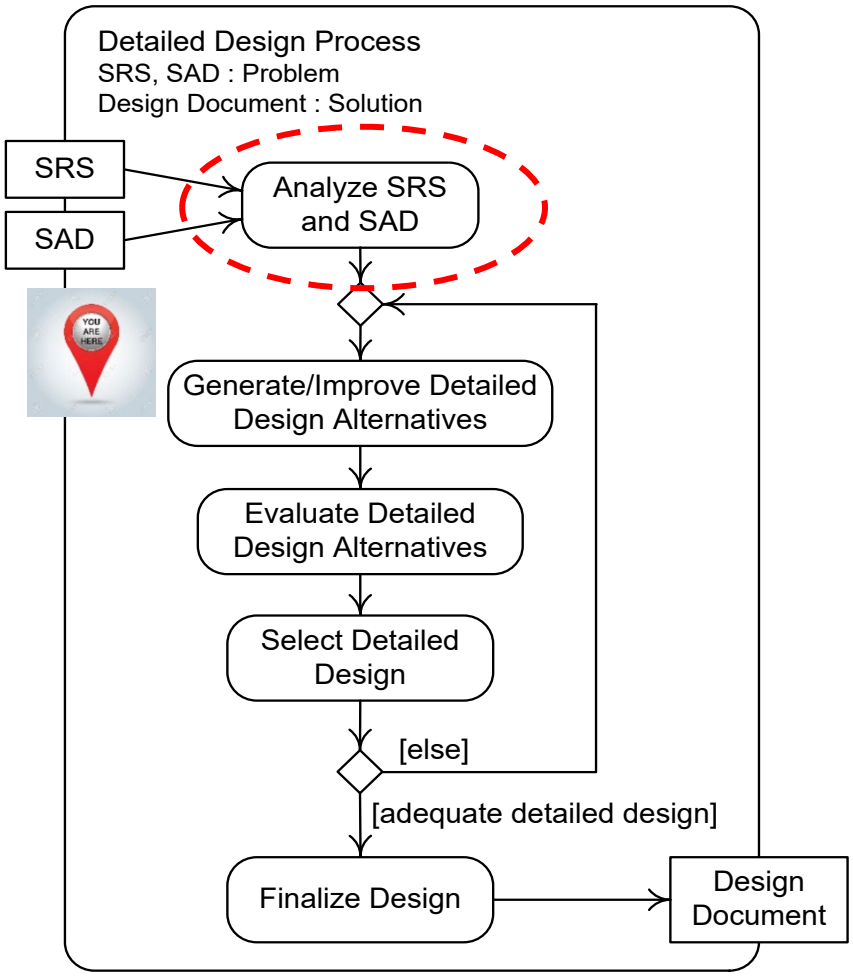
MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

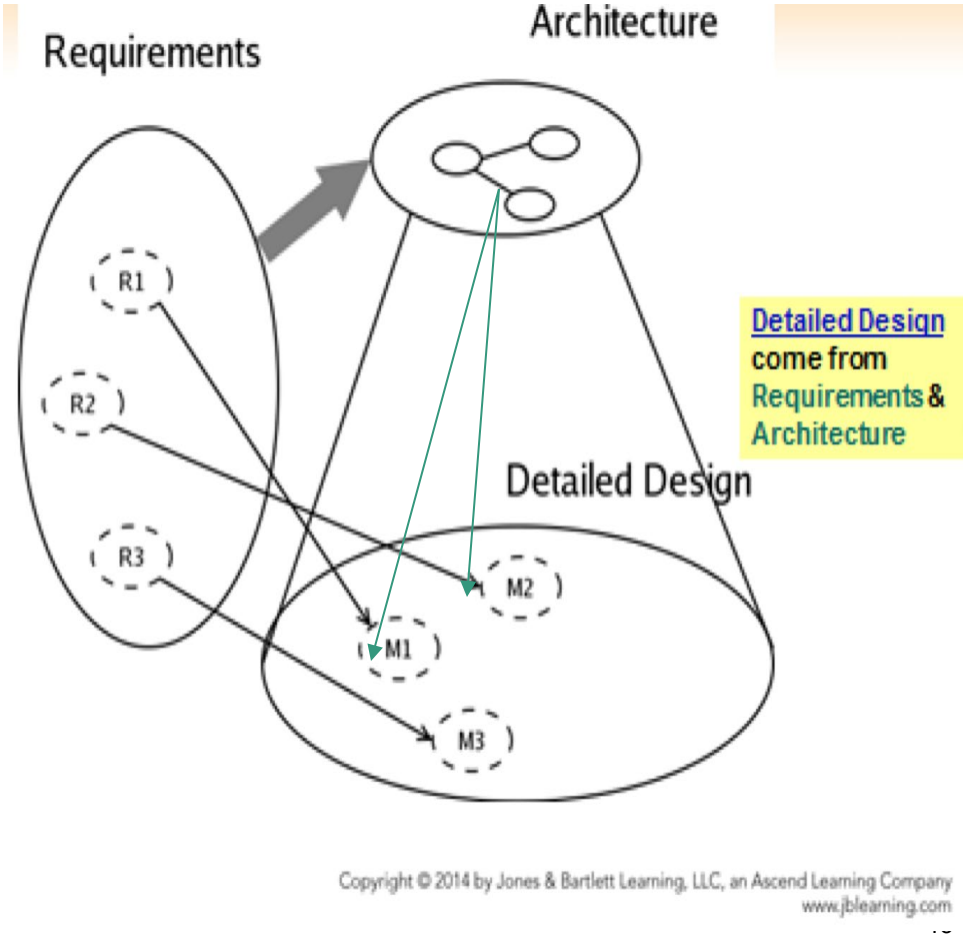
OUTLINE

- Software detailed design
  - Process
  - Models and their representation
  - UML notations used to represent detailed design
  - Construction techniques and principles
  - Design Patterns
  - Operations and data
  - Software design document
- Assignments

We are here

DETAILED DESIGN PROCESS – ANALYZE SRS AND SAD

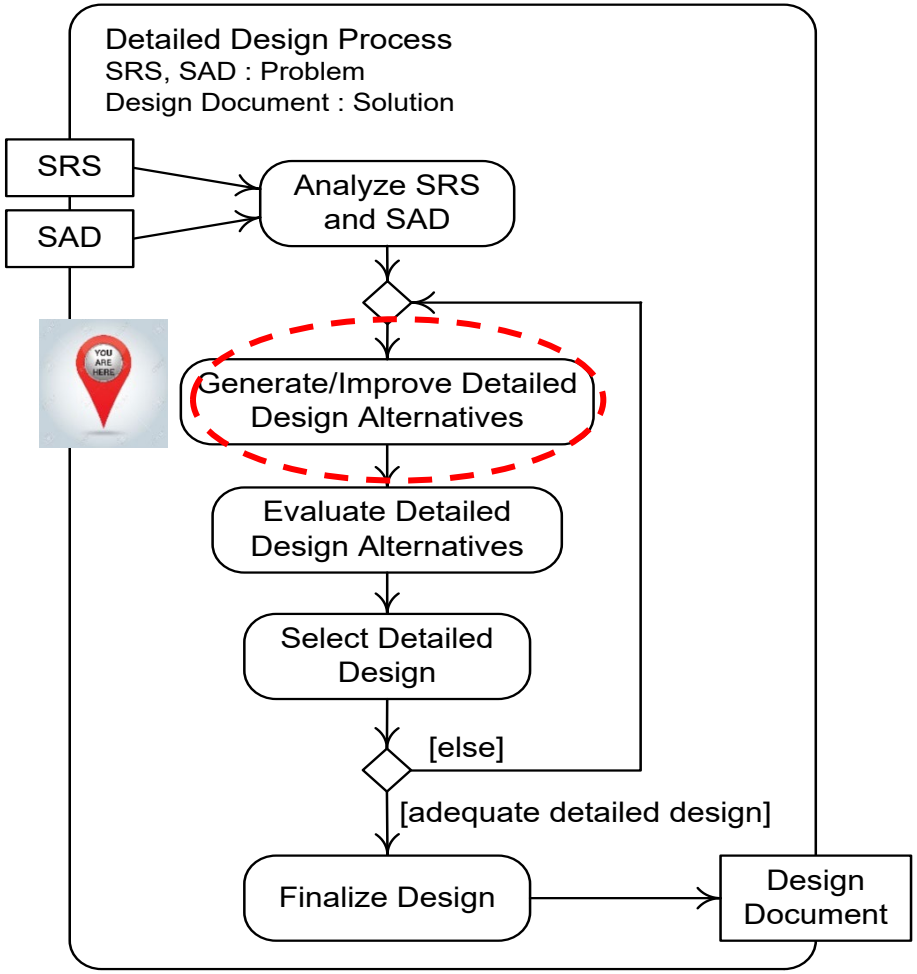




Copyright © 2014 by Jones & Bartlett Learning, LLC, an Ascend Learning Company
www.jblearning.com

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

DETAILED DESIGN PROCESS – GENERATE DETAILED DESIGN



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

DETAILED DESIGN SPECIFICATION

What information does detailed design represent?

- **Decomposition**—Parts or units (detailed design *elements/components*)
- **Responsibilities**— Data and behavior
- **Interfaces**—Public features
- **Collaborations**—Who does what, and when
- **Relationships**—Inheritance, associations, etc.
- **Properties**—Performance, reliability, etc.
- **States and transitions**—Externally visible
- **Packaging and Implementation**—Scope, visibility, etc.
- **Algorithms, Data Structures and Types**

OUTLINE

- Software detailed design
  - Process
  - Models and their representation
  - UML notations used to represent detailed design
  - Construction techniques and principles
  - Design Patterns
  - Operations and data
  - Software design document
- Assignments

We are here

DETAILED DESIGN MODELS

- **Static** models – document software **structure**
  - Detailed design static model: detailed design components and their relations
    - **When using an OO design approach, we call the detailed design components "classes"**
    - Use **UML class diagram** for this model representation
  - Operations specification (algorithms), data structures
- **Dynamic** models – document **behavior**
  - Interactions between detailed design components (objects)
    - Typically use **UML sequence diagrams** for this model representation
  - Process aspects, e.g., communication, timing, concurrency
    - Typically use UML sequence diagram and timing diagram for this model representation
  - States and transitions
    - Use **UML state diagram** for this model representation

Static and dynamic models construction should go hand-in-hand

OUTLINE

- Software detailed design
  - Process
  - Models and their representation
  - UML notations used to represent detailed design
  - Construction techniques and principles
  - Design Patterns
  - Operations and data
  - Software design document
- Assignments

We are here

DETAILED DESIGN STATIC/STRUCTURAL MODEL REPRESENTATION

- Using **UML class diagram**
  - Represent a *detailed design component (a.k.a. class in OO)* with a *UML class*
  - Represent *relations* between detailed design components with *relations between UML classes*
    - E.g., generalization, dependency (uses)

Revisit advanced features of UML class diagram (textbook Chapter 11)

DETAILED DESIGN DYNAMIC/ BEHAVIOR MODEL REPRESENTATION

- Using **UML sequence diagram**
  - Represent each *detailed design component* (class/object) on its own *lifeline*
  - Represent *communication* of detailed design components (classes/objects) with *messages* between lifelines
  - Use combined fragments to show logic/control flow
    - E.g., alternatives, loops, branches

- ***Revisit advanced features of UML sequence diagram (textbook Chapter 12)***

DETAILED DESIGN DYNAMIC/BEHAVIOR MODEL REPRESENTATION (CONTINUED)

State and transitions - Using **UML state machine diagram**

- Purpose/Usage: To better understand complex entities (classes/objects), particularly those that act in different manners depending on their state; how individual objects change their state in response to events
- Frequently used for real time software

- Heuristic: draw one or more state machine diagrams for a class (object) that exhibits *different behavior depending on its state*
  - Usually we don't need a state diagram for all the objects in the system

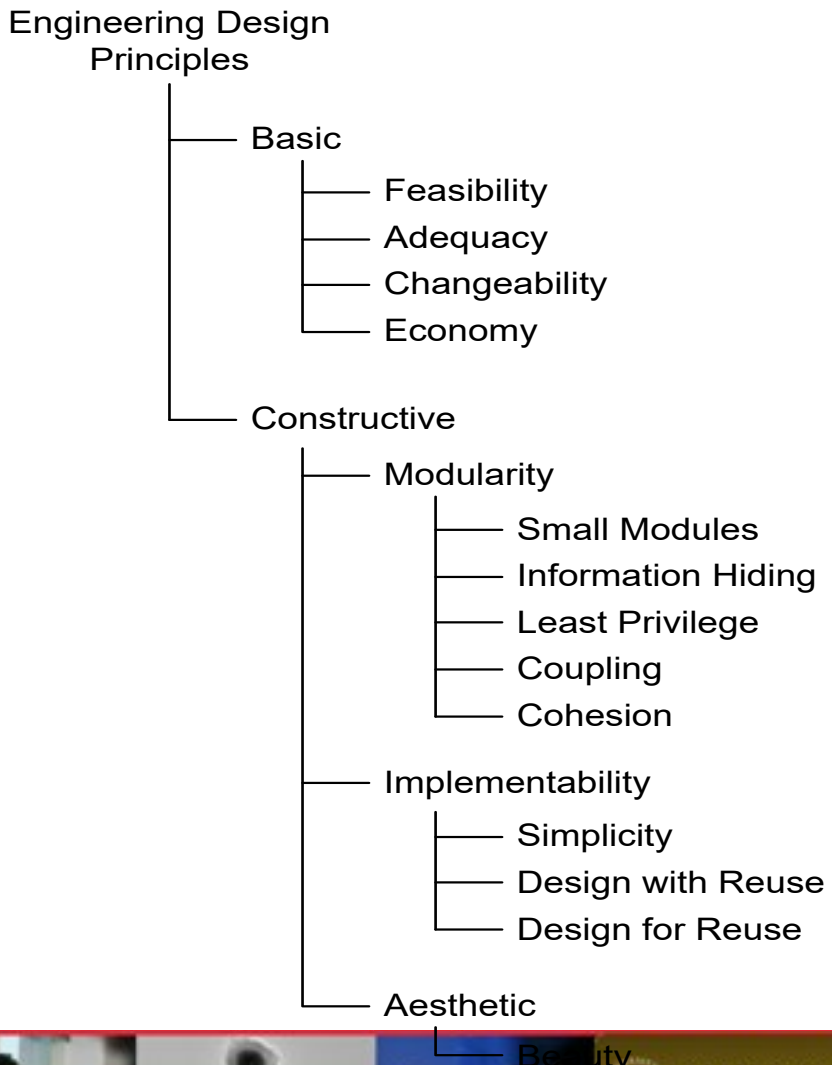
- ***Revisit advanced features of UML state machine diagram (textbook Chapter 13)***

OUTLINE

- Software detailed design
  - Process
  - Models and their representation
  - UML notations used to represent detailed design
  - Construction techniques and principles
  - Design Patterns
  - Operations and data
  - Software design document
- Assignments

We are here

ENGINEERING DESIGN PRINCIPLES



- These design principles apply to **both** architectural design <u>as well as</u> detailed design

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

SOFTWARE DETAILED DESIGN CONSTRUCTION

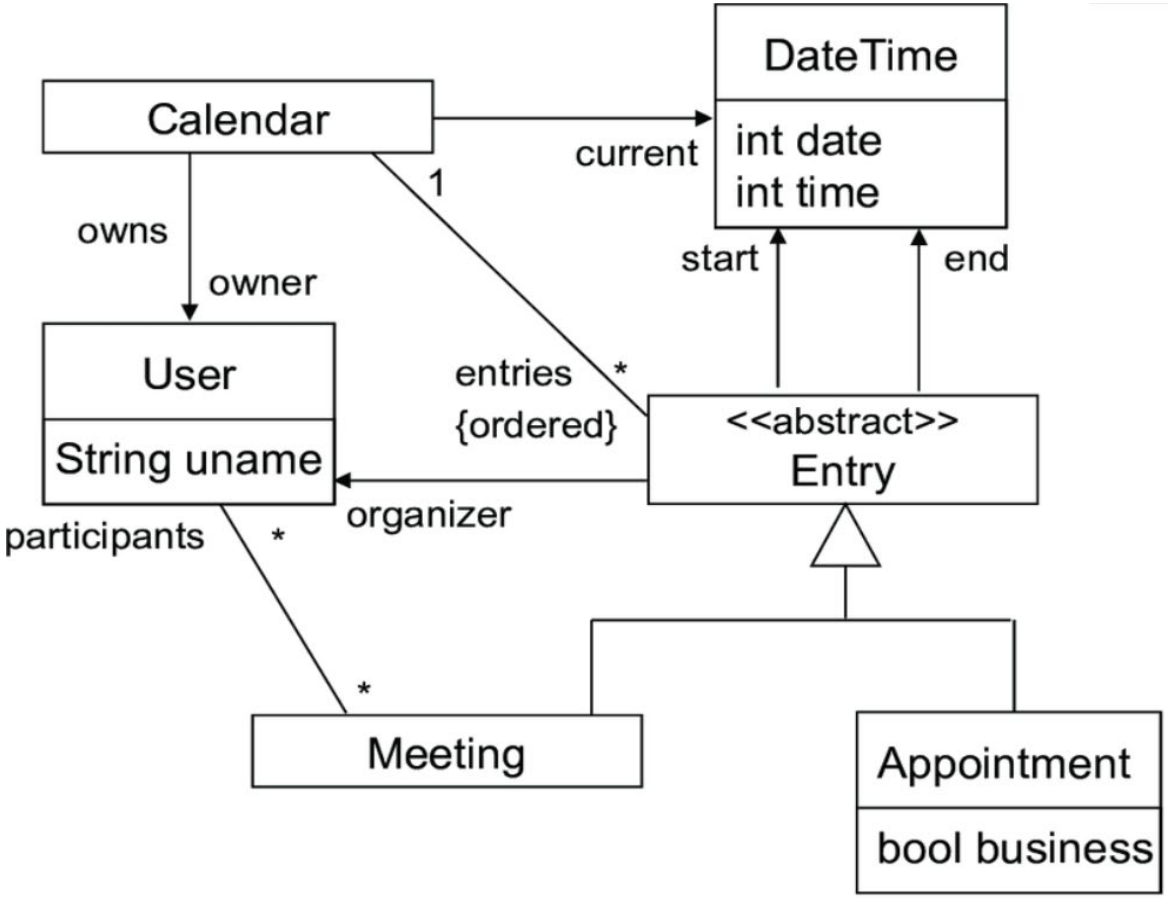
- Categories of construction techniques :
  - *Creational*—Make a detailed design class model from scratch
    - Functional decomposition
    - Quality attribute decomposition
    - Design themes
  - *Transformational*—Change another model into a detailed design class model
    - Similar system
    - Patterns or **architecture**
    - Analysis model

EXAMPLE OF *TRANSFORMATIONAL* TECHNIQUES

- Transform a *Conceptual (Domain) Model* directly into a *Detailed Design Static Model*
  - Typically for small software
  - Design entities resemble real world entities
  - Heuristics:
    - Change external domain entities (actors) to interface classes
    - Add actor-related/specific domain classes
    - Add a startup class
    - Convert or add controllers and coordinators
    - Add classes for data types
    - Convert or add container classes
    - Convert or add engineering design associations
- Transform *Architectural Models* into *Detailed Design Models*
  - Typically for large software
  - Refine architecture models

EXAMPLE OF *TRANSFORMATIONAL* APPROACH

From a *domain model* to *detailed design*



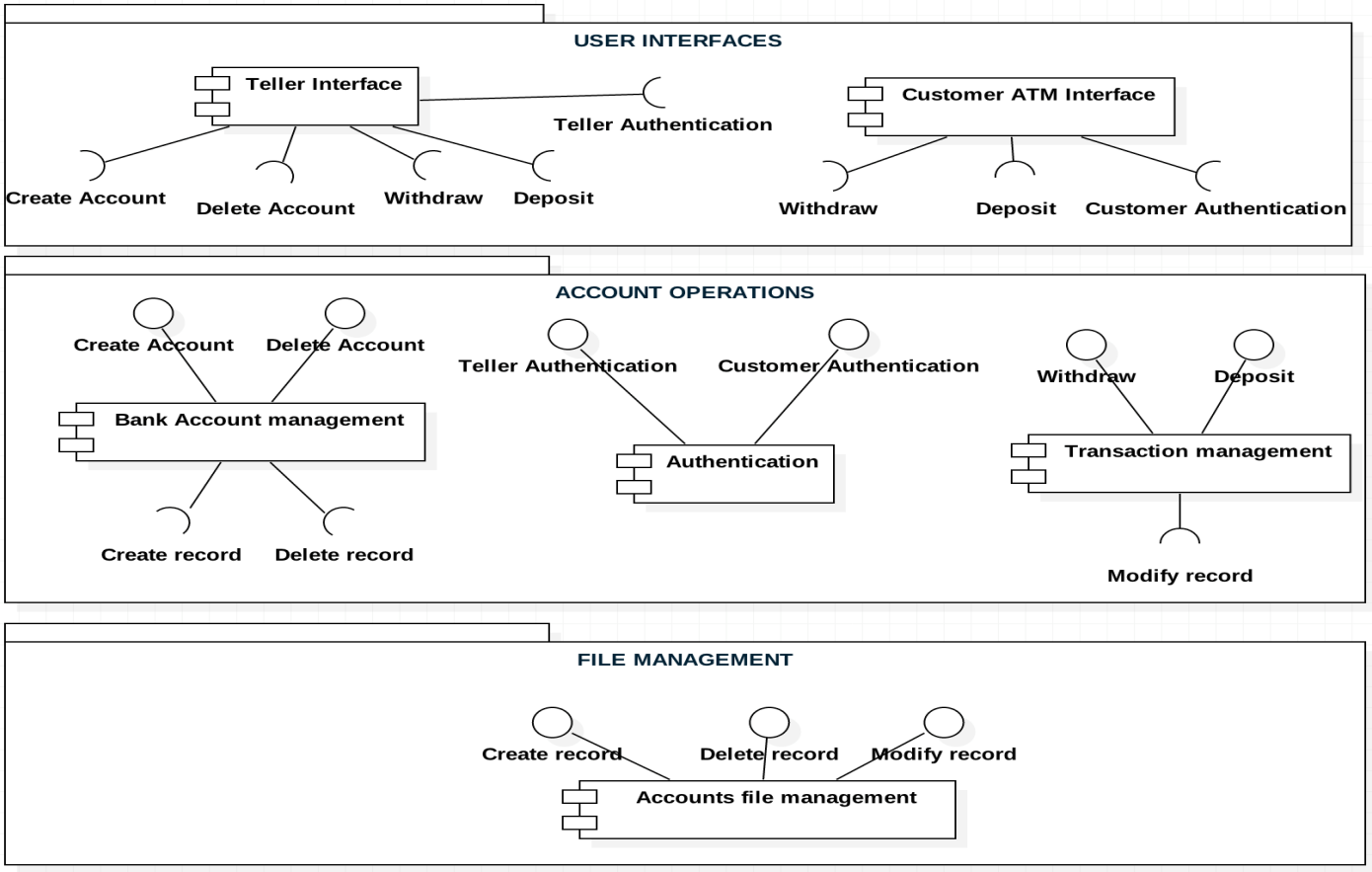
Works well for small software (e.g., a calendar app) where domain entities correspond to detailed design elements

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

EXAMPLE OF *TRANSFORMATIONAL* APPROACH

From a *software architecture model* to a *detailed design model*

The *Bank Account System* Architecture

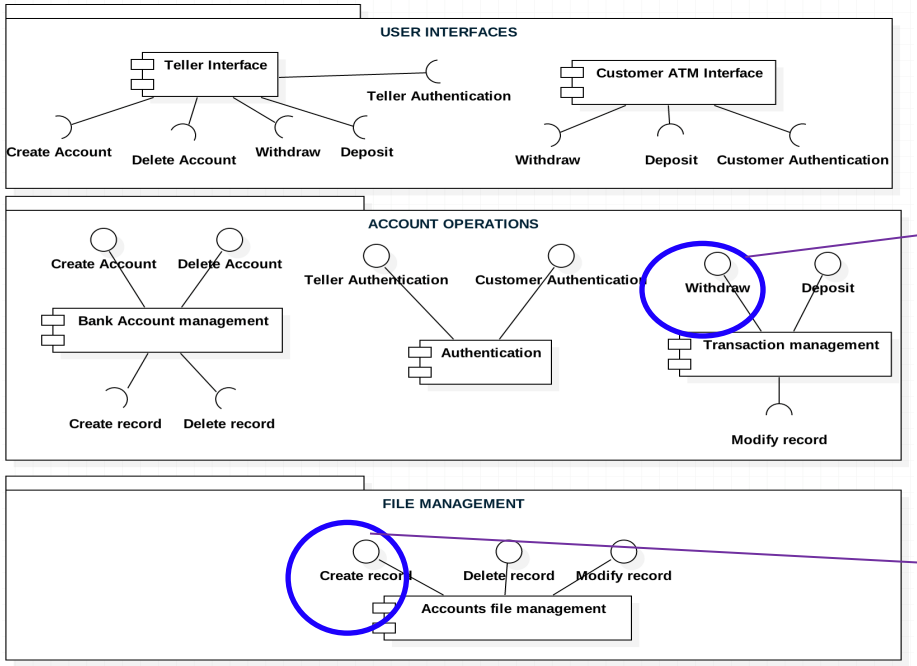


MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

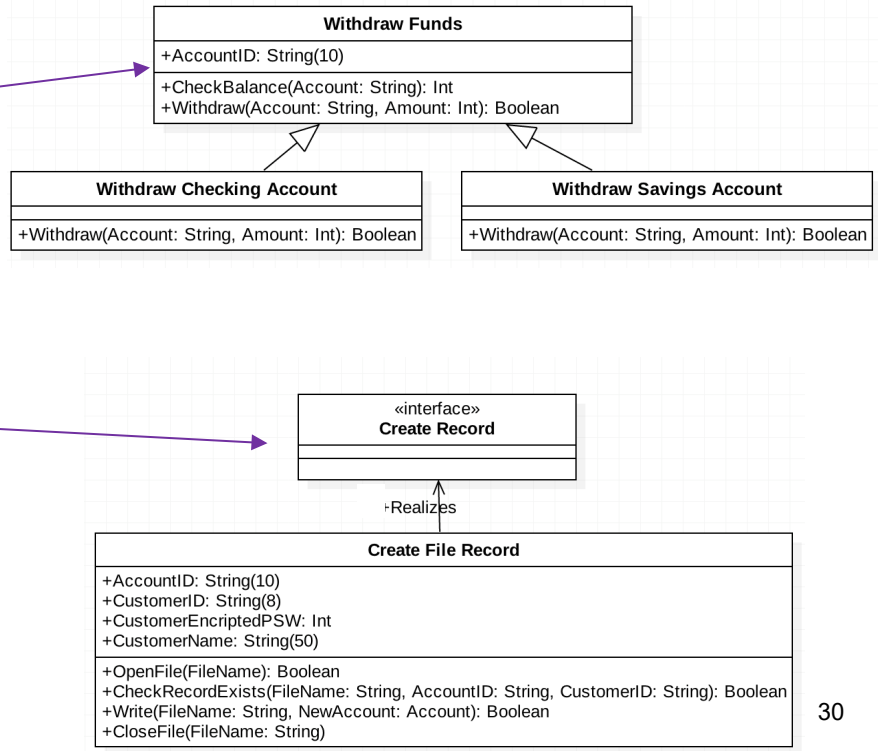
EXAMPLE OF *TRANSFORMATIONAL* APPROACH

From a *software architecture model* to a *detailed design model*

Bank Account System Architecture



Bank Account System Detailed Design



Legend: ⟶

means “Decompose and/or refine architectural element into design element(s)”

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

RESPONSIBILITY-DRIVEN DECOMPOSITION

- A *responsibility* is an *obligation* to
  - Perform a task (*operational responsibility*) or
  - Maintain some data (*data responsibility*)

- *Operational responsibilities* are usually fulfilled by detailed design components' (classes') **operations**
- *Data responsibilities* are usually fulfilled by detailed design components' (classes') **attributes**

- Class collaborations may be involved

RESPONSIBILITY-DRIVEN DECOMPOSITION (CONTINUED)

- *Responsibilities* may be stated at different levels of abstraction and can be decomposed
  - High-level responsibilities can be assigned to top-level components
  - Responsibility decomposition can be the basis for decomposing components
- Responsibilities reflect both operational and data obligations, so responsibility-driven decomposition could be different from functional decomposition

RESPONSIBILITY DECOMPOSITION HEURISTICS

High Cohesion, Low Coupling

- Assigning responsibilities well, helps achieve *high cohesion* and *low coupling*
  - State both operational and data responsibilities
  - Assign modules *at most* one operational and one data responsibility
  - Assign complementary data and operational responsibilities
  - Make sure module responsibilities do not overlap
  - Place operations and data in a module *only* if they help fulfill the module's responsibilities
  - Place *all* operations and data needed to fulfill a module responsibility in that module

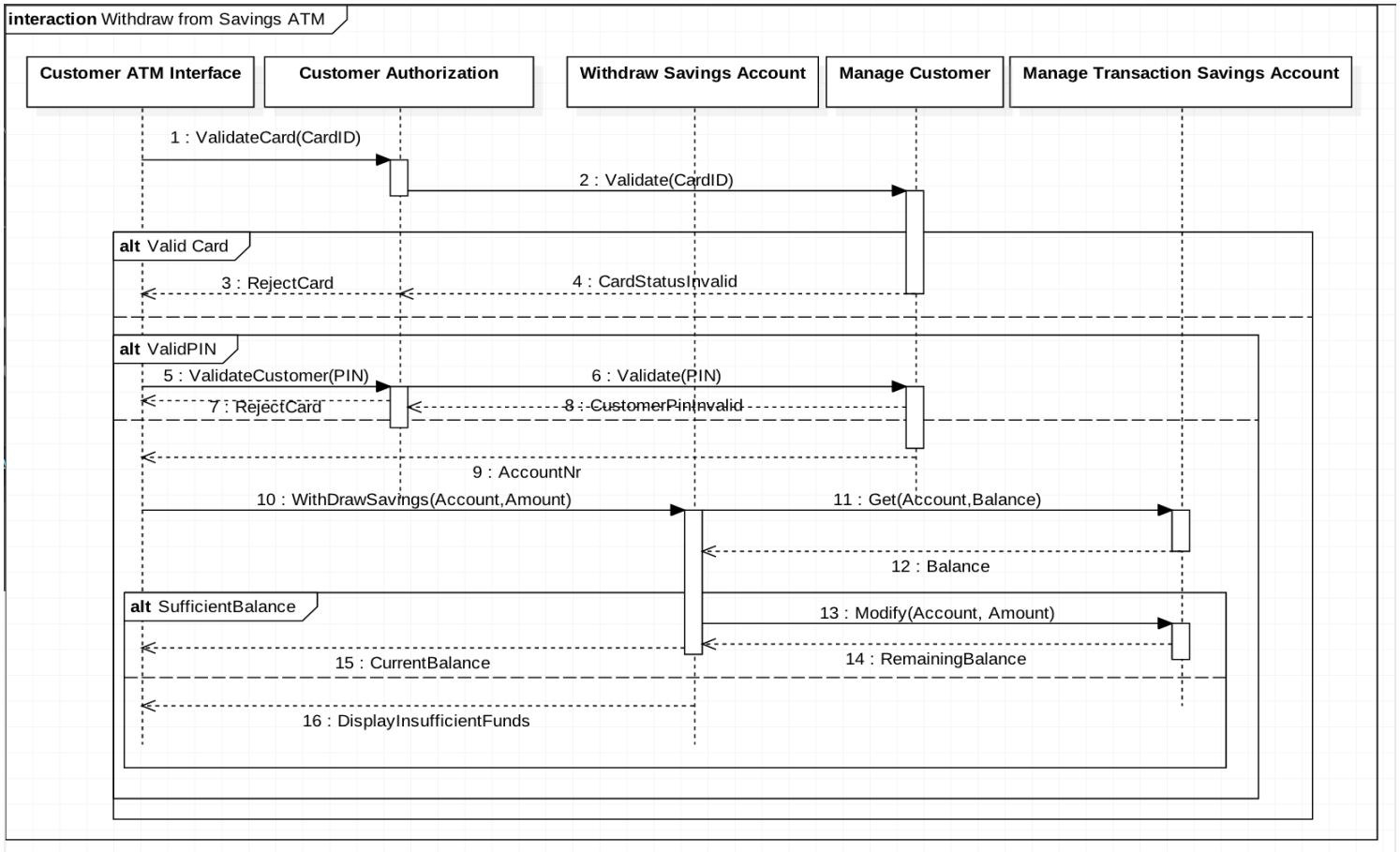
Modularity **design principle**

DETAILED DESIGN BEHAVIORAL MODELS

- Are *dynamic* models
- Represent interactions between detailed design components (classes or their instances - objects) to realize functional or quality scenarios
  - Similar to architecture behavioral models
- Typically **UML sequence diagrams** are used for this model representation
  - A life line represents a detailed design component and its behavior (according to its identified/assigned *responsibility*)
    - Including construction/destruction (start/end of life)
  - Data exchange, calls and returns between detailed design components are represented as messages
  - Interaction fragments show control logic, e.g. condition/branching, loops

EXAMPLE OF DETAILED DESIGN BEHAVIOR MODEL USING UML SEQUENCE DIAGRAM

Bank Account System – Withdraw from Savings Account functional scenario



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

OBJECT ORIENTED DESIGN

- A design is **object oriented** if it decomposes a system into a collection of components called ***objects*** that encapsulate data and functionality
  - Objects are uniquely **identifiable** runtime entities that can be designated as the target of a message or request
  - Objects can be **composed**, in that an object's data variables may themselves be objects, thereby encapsulating the implementations of the object's internal variables
  - The implementation of an object can be reused and extended via **inheritance**, to define the implementation of other objects
  - OO code can be **polymorphic:** written in generic code that works with objects of different but related types

OBJECT ORIENTED DESIGN (CONTINUED)

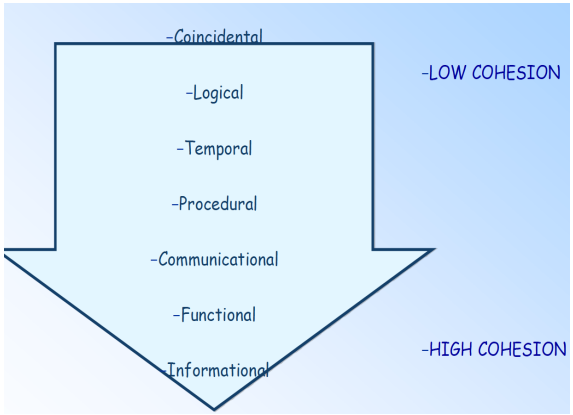
- Principles:
  - Encapsulation – of data and behavior
  - Inheritance
  - Polymorphism

- Benefits:
  - Facilitate reuse
  - Enable modifiability
  - Prevent code rigidity and fragility

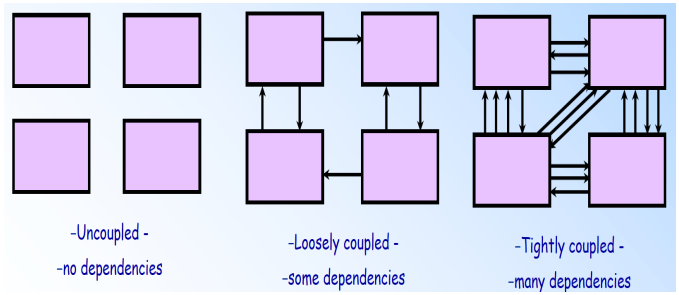
MODULARITY DESIGN PRINCIPLES – COHESION AND COUPLING

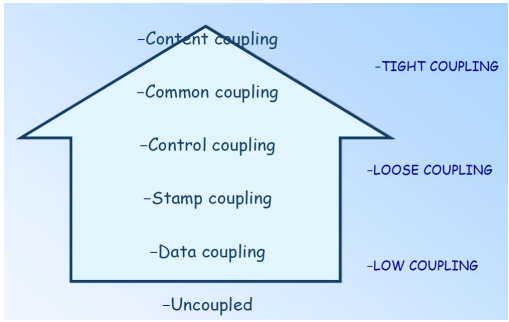
- Goal: **High** *cohesion* and **low** *coupling*

Types and levels of Cohesion



Types and levels of Coupling

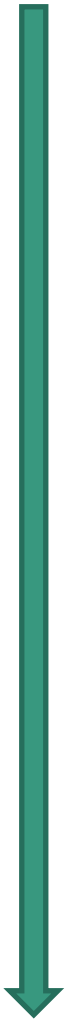




From: *Software Engineering: Theory and Practice*, by Pfleeger and Atlee

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

TYPES OF COHESION



- Coincidental (lowest degree)
  - Parts are unrelated to one another
- Logical
  - Parts are related only by the logic structure of code
- Temporal
  - Module's data and functions related because they are used at the same time in an execution
- Procedural
  - Similar to temporal, and functions pertain to some related action or purpose
- Communication
  - Operates on the same data set
- Functional (ideal degree)
  - **All elements essential to a single function are contained in one module, and all of the elements are essential to the performance of the function**
- Informational (ideal degree)
  - Adaption of functional cohesion to data abstraction and object-based design

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

TYPES OF COUPLING

- Content coupling
  - Occurs when one component modifies an internal data item in another component, or when one component branches into the middle of another component
- Common coupling
  - Making a change to the common data means tracing back to all components that access those data to evaluate the effect of the change
- Control coupling
  - When one module passes parameters or a return code to control the behavior of another module
  - It is impossible for the controlled module to function without some direction from the controlling module
- Stamp coupling
  - Stamp coupling occurs when complex data structures are passed between modules
- Data coupling
  - If only data values, and not structured data, are passed between modules

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

MODULARITY DESIGN PRINCIPLE - INFORMATION HIDING

- Prevent certain aspects of a software component from being accessible to its clients
- Protect information from any inadvertent change
- Do not reveal unnecessary details
- Segregation of *design decisions* that are most likely to change
- Provide a stable interface which protects the remainder of the software from the implementation, i.e., the details that are most likely to change
- Key technique for hiding information - restrict access to program entities as much as possible
  - Limiting *visibility*—Use scope and visibility markers to restrict visibility
  - Not extending access—Avoid using references and aliases to extend visibility

INFORMATION VISIBILITY AND ACCESSIBILITY

- Definitions:
  - A program *entity* is anything in a program that is treated as a unit
  - A *name* is an identifier bound to a program entity
  - A program entity is *visible* at a point in a program text if it can be referred to by name at that point; the portion of software over which an entity is visible is its *visibility*
  - An entity is *exported* from the module where it is defined if it is visible outside that module
  - A program entity is *accessible* at a point in a program text if it can be used at that point
- Types of visibility
  - *Local*—Visible only within the module where it is defined
  - *Non-local*—Visible outside the module where it is defined, but not visible everywhere in a program
  - *Global*—Visible everywhere in a program

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

OBJECT ORIENTED VISIBILITY

- *Private*—Visible only within the class where it is defined
  - A form of ***local*** visibility

~ *Package*—Visible in the class where it is defined as well as classes in the same package or namespace
  - A form of ***non-local*** visibility

Protected—Visible in the class where it is defined and all its sub-classes
  - A form of ***non-local*** visibility

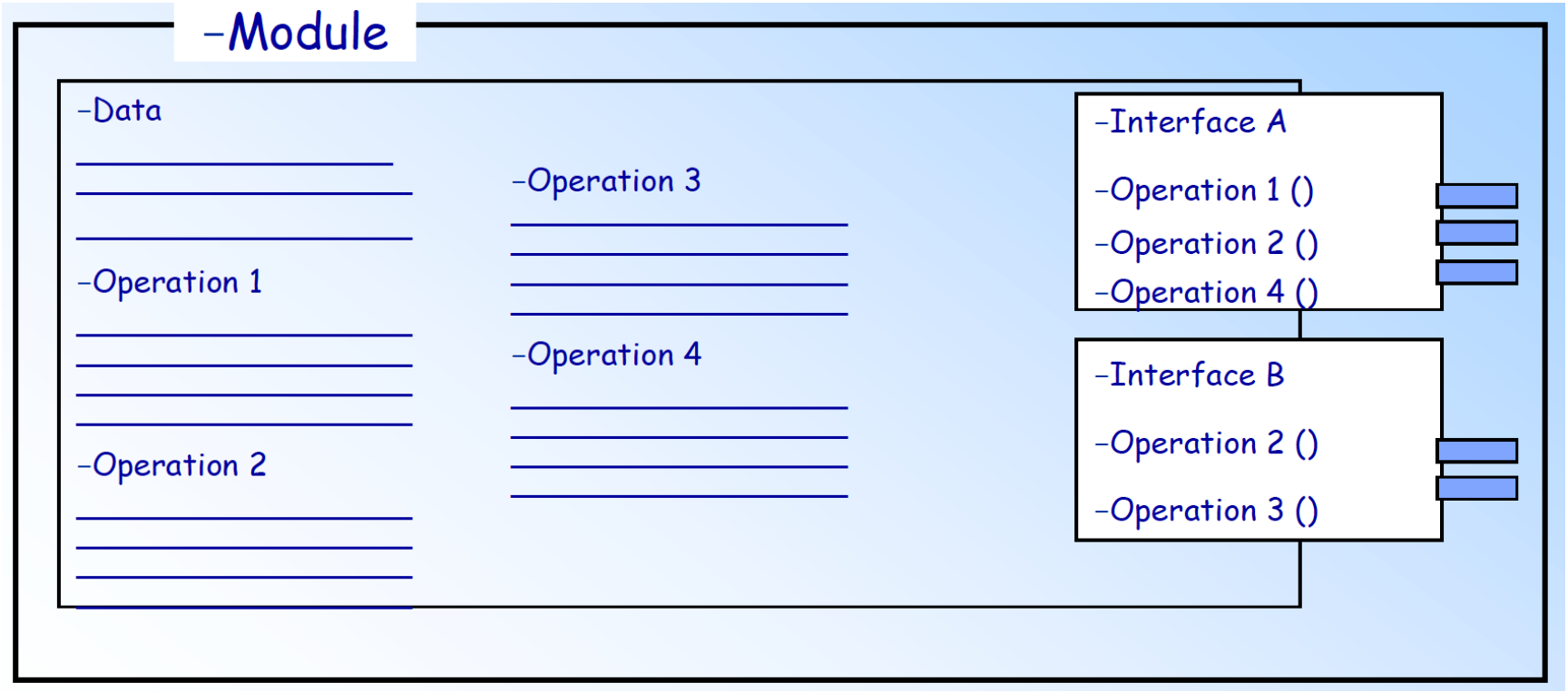
\+ *Public*—Visible anywhere the class is visible
  - A form of ***global*** visibility

INTERFACE VS. IMPLEMENTATION (INTEGRATION)

- *Clear distinction between interface and Implementation* is a key design principle
- Supports separation of concerns
  - Client components/classes/modules (consumers) care about services/ resources exported from servers
  - Server components/classes/modules (providers) care about implementation of those services/resources
- Interface acts as a *contract* between a module (*server*) and its *clients*
  - A ***server***/*used* module *exports* services through its *interface* and implements the exported resources
    - Implementation is *hidden* to clients
  - The ***client*** *imports (uses)* the resources that are exported by its servers

Note the use of terms *server* and *client*

INTERFACE CONCEPT

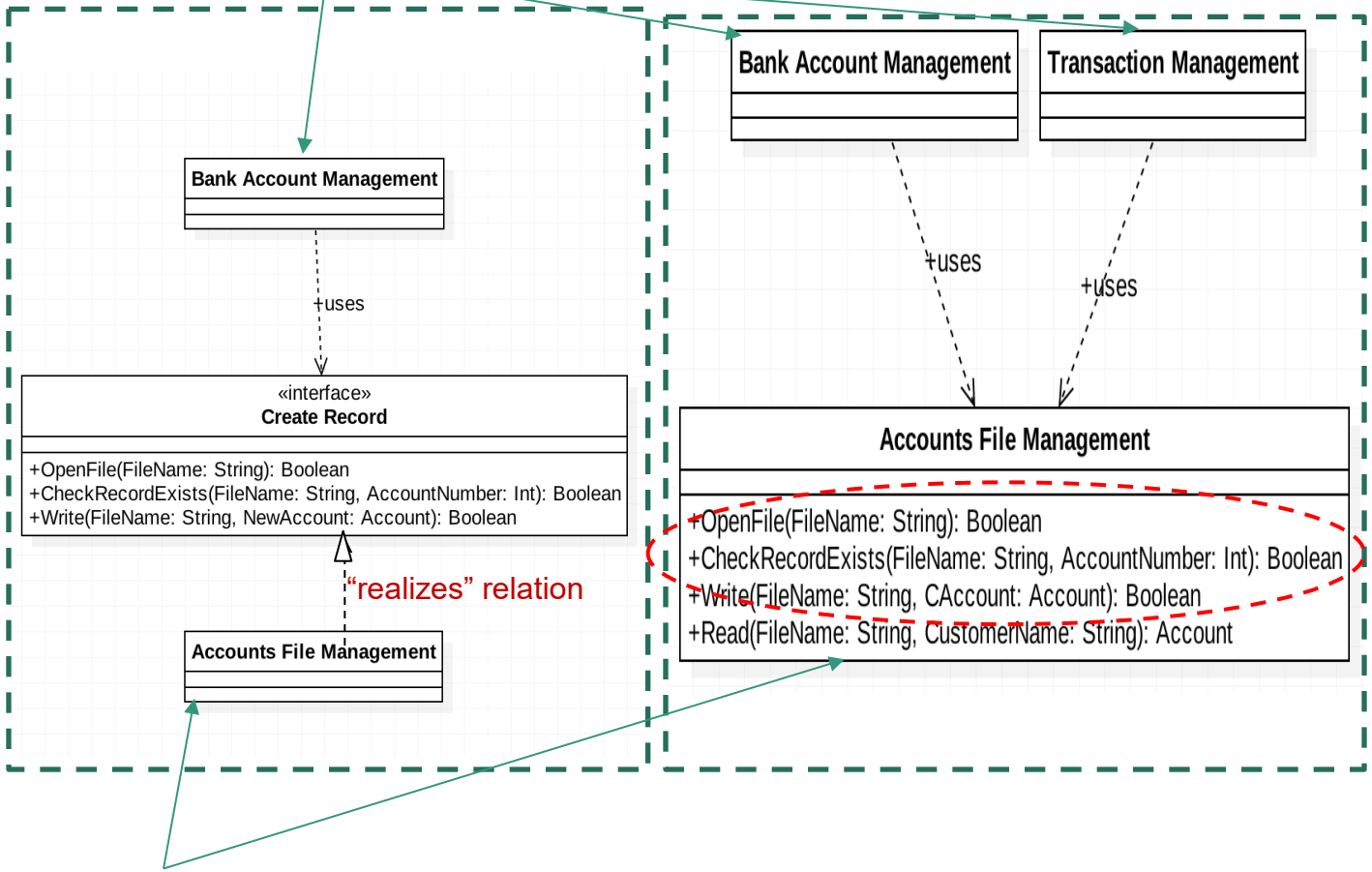


From: *Software Engineering: Theory and Practice*, by Pfleeger and Atlee

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

INTERFACE AND IMPLEMENTATION EXAMPLE AND UML CLASS NOTATION

Client class *imports* the resources/services that are *exported* by the *server* that the *client uses*



Server class (*provides/exports* resources/services through its (provided) interface

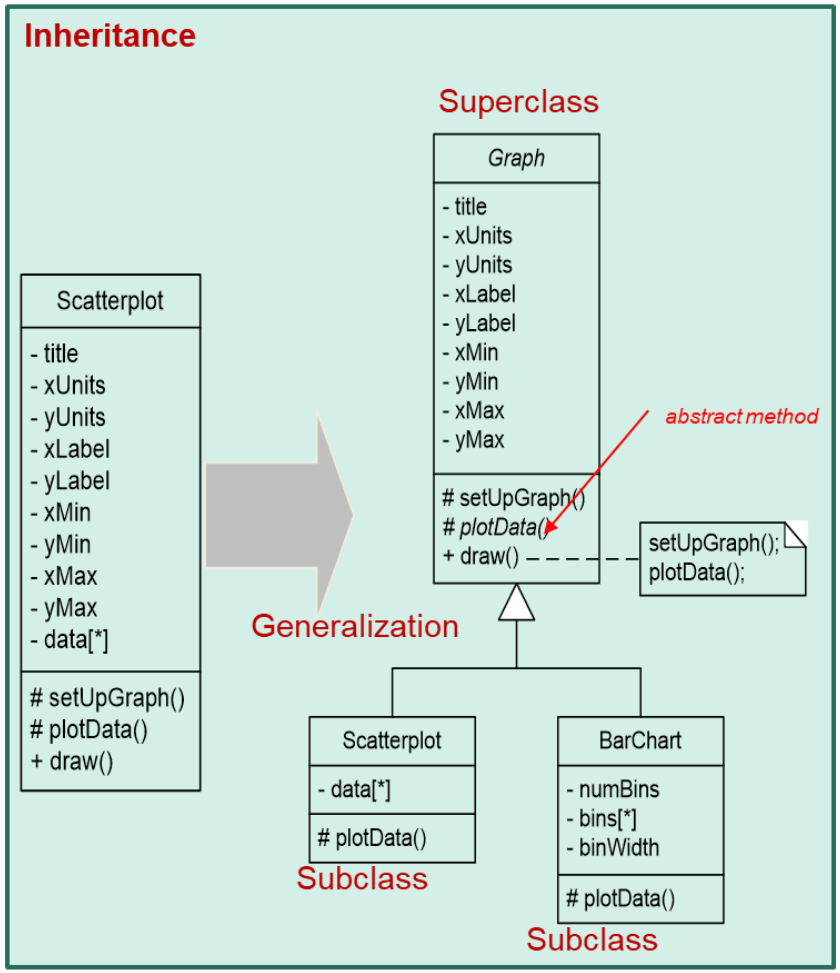
MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

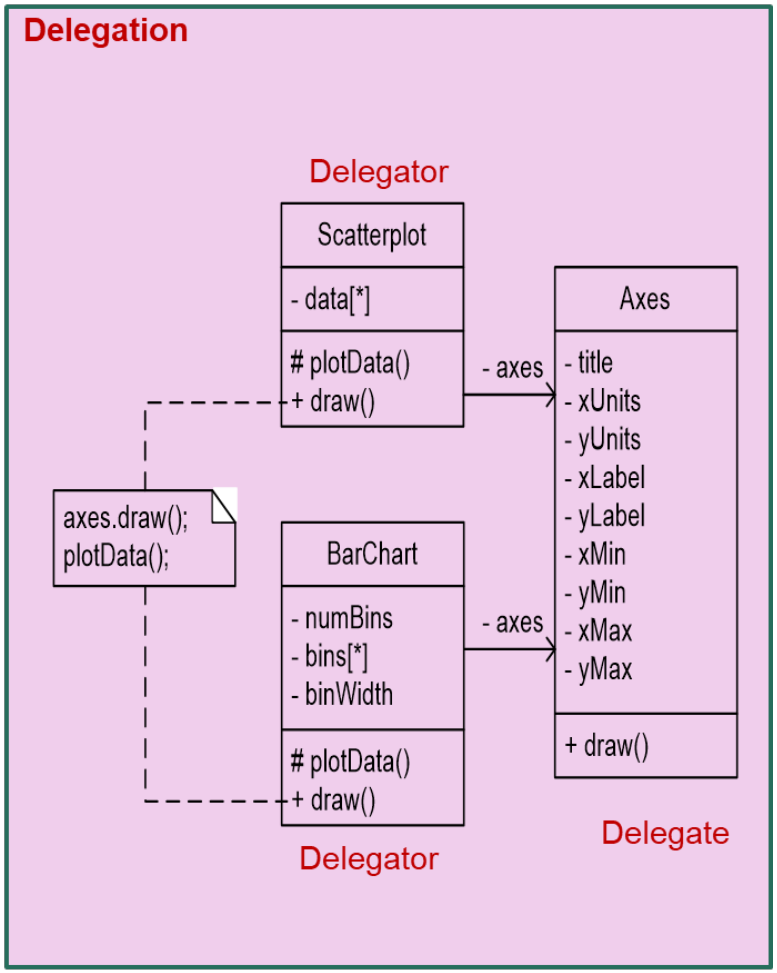
INHERITANCE AND DELEGATION AS REUSE MECHANISMS

- Inheritance
  - A declared relation between a class and one (or more) super-class(es) that causes the sub-class to have every attribute and operation of the super-class(es).
    - Captures a *generalization relation* between classes
    - Allows ***reuse*** *of attributes and operation* from super-classes in sub-classes

- Delegation
  - A tactic wherein one module (the *delegator*) entrusts another module (the *delegate*) with a responsibility.
    - Allows **reuse** without violating inheritance constraints
    - Makes software more reusable and configurable

INHERITANCE AND DELEGATION EXAMPLES





MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

INHERITANCE AND DELEGATION HEURISTICS

- Use *inheritance* only when there is a generalization relationship between the sub-class and its super-class(es)
  - Combine common attributes and operations in similar classes into a common super-class
  - Use inheritance when you want *substitutability*
- Use *delegation* to increase reuse, flexibility, and configurability

OO "SOLID" DESIGN PRINCIPLES

- **Problem**: change happens in software development and maintenance
  - If class dependency management is badly handled => code can become rigid, fragile, and immobile, hence difficult to maintain and reuse (*spaghetti* code)
    - Rigid - every change affects many other parts
    - Fragile – things break in unrelated places
    - Immobile – cannot be reused outside its original context

- **Solution** – apply **"SOLID" principles** to module (class) **dependency management**
  - => software maintainability and reusability
- Design software not just to satisfy immediate needs, but with future *reuse* and *change* in mind
  - Anticipated change is based on input from stakeholders

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

OO SOLID DESIGN PRINCIPLES (CONTINUED)

SRP	The Single Responsibility Principle	*A class should have one, and only one, reason to change*
OCP	The Open Closed Principle	*You should be able to extend a class's behavior, without modifying the class*
LSP	The Liskov Substitution Principle	*Derived classes must be substitutable for their base classes*
ISP	The Interface Segregation Principle	*Make fine grained interfaces that are client specific*
DIP	The Dependency Inversion Principle	*Depend on abstractions, not on concretions*

From: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

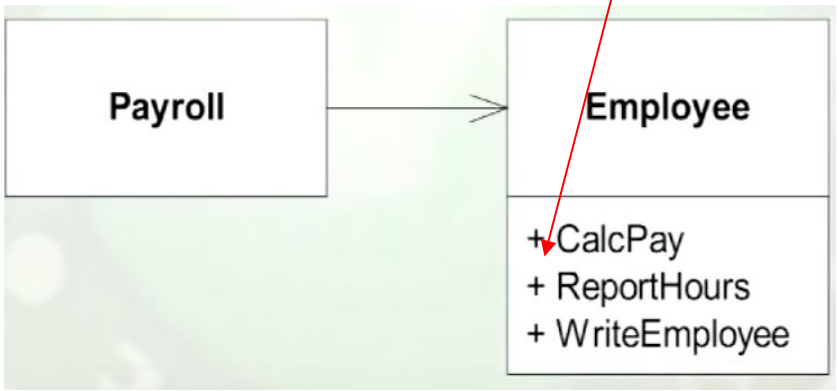
SINGLE RESPONSIBILITY PRINCIPLE (SRP)

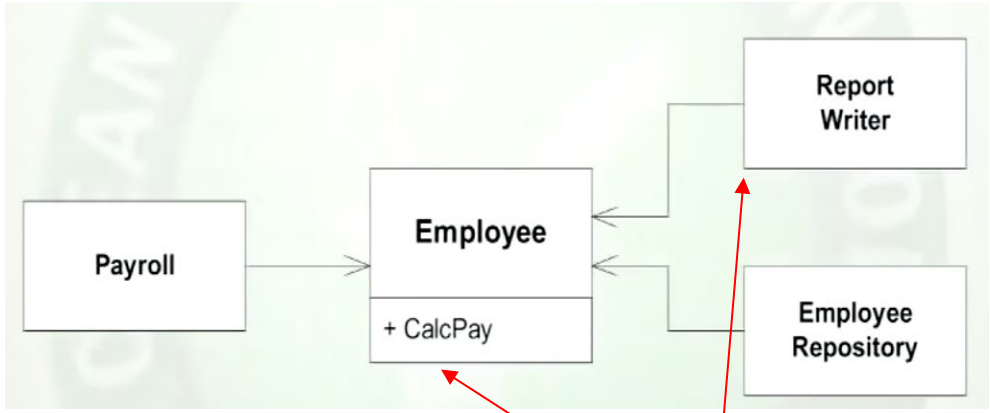
- *A software entity (module, class) should have one and only one reason to change*
  - Don't mix multiple concerns in classes
- *Class responsibility* is related to *change*
- Every class should have *only one job to do*
  - Everything in the class should be related to that single purpose
  - It does **not** mean that classes should only contain one method or property
    - There may be many members *as long as they relate to the single responsibility*
  - It may be that when the one reason to change occurs, multiple members of the class may need modification or multiple classes will require updates
- Applying SRP leads to *smaller*, *more cohesive classes* => code is simpler to understand and maintain
  - The number of classes in a solution may increase accordingly -> organize them well using namespaces and project folders

SINGLE RESPONSIBILITY PRINCIPLE EXAMPLE

Multiple responsibilities
(changing one operation may inadvertently chan another one

A class should have one, and only one, reason to change





CalcPay implements the algorithms that determine how much a particular employee should be paid, based on that employee's contract, status, hours worked, etc.
WrtieEmployee stores the data managed by the Employee object onto the enterprise database.
ReportHours returns a string which is appended to a report that auditors use to ensure that employees are working the appropriate number of hours and are being paid the appropriate compensation.

From where does change come?
CalcPay – from CEO
ReportHours – from COO
WriteEmployee – from CTO

Single responsibility

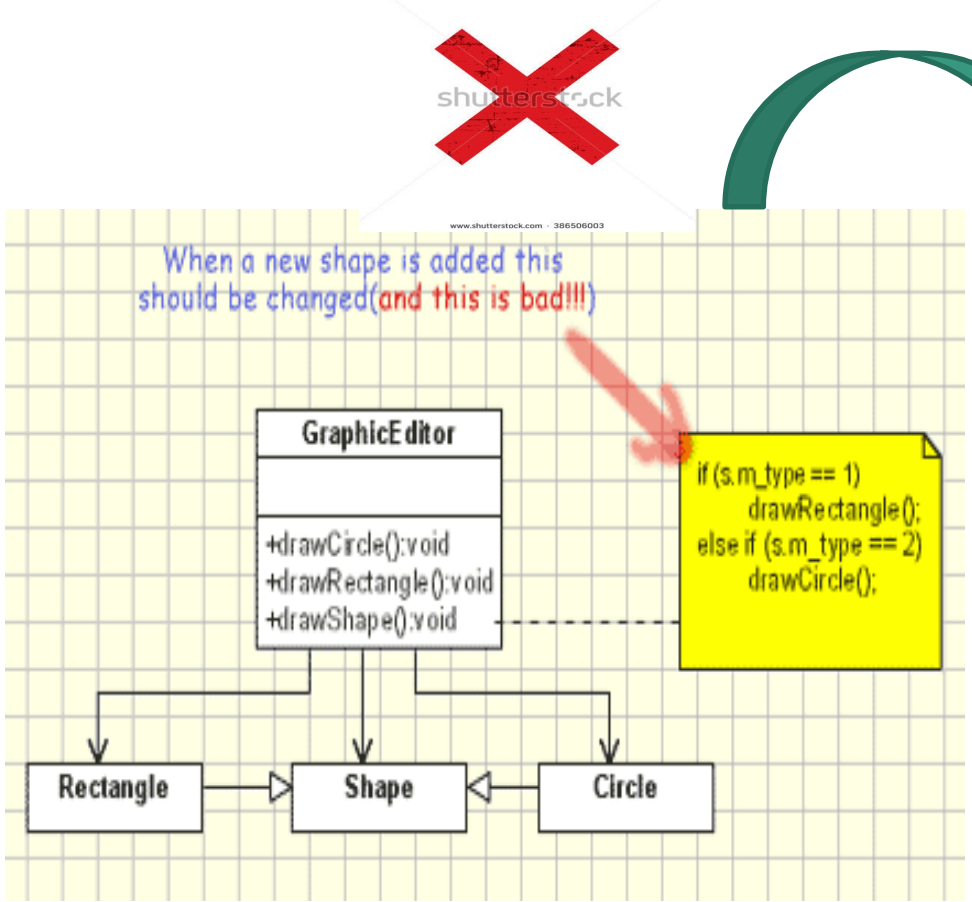
From: Bob Martin SOLID Principles of Object Oriented and Agile Design
https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html

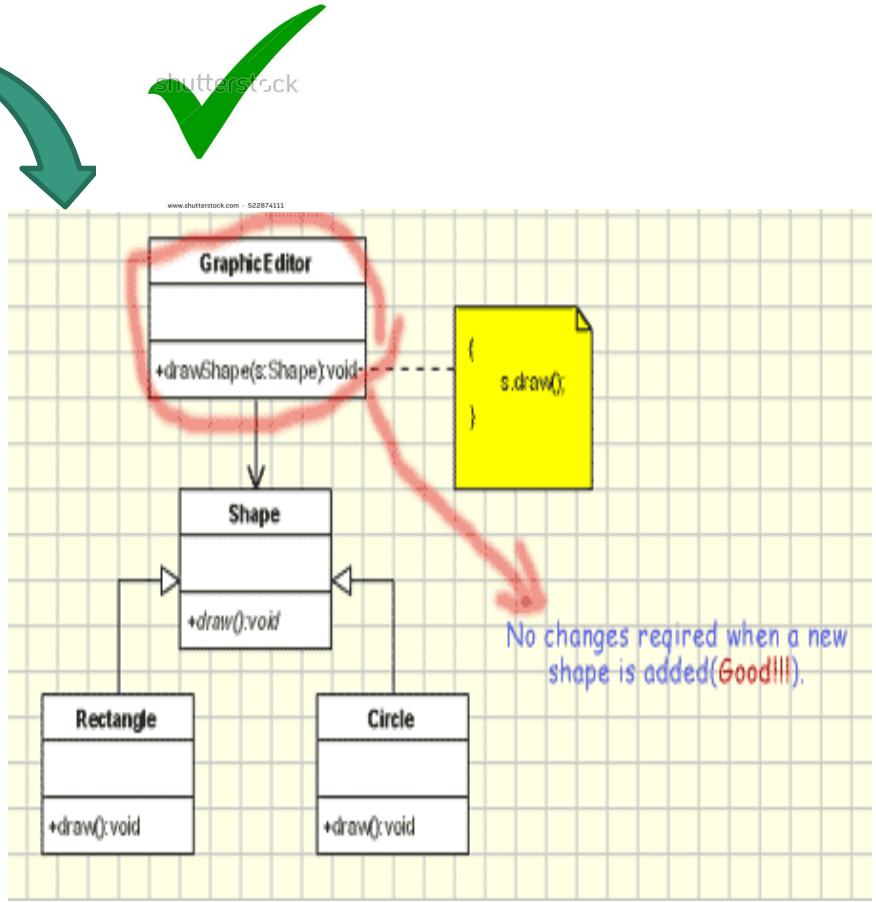
OPEN/CLOSED PRINCIPLE (OCP)

- *Software entities (classes, modules, etc.) should be open for extension but closed for modification*
- You should be able to extend a class's behavior, without modifying the class
- The "closed" part of the rule states that once a module has been developed and tested, the code should only be adjusted to correct bugs
- The "open" part says that you should be able to extend existing code in order to introduce new functionality/ behavior
  - Write new code, but don't change the existing one
- As with the SRP, this principle reduces the risk of new errors being introduced by *limiting changes to existing code*

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

OPEN CLOSE PRINCIPLE EXAMPLE





"Plug-in" design solution

"Extend a Class Behavior without Modifying the Class"

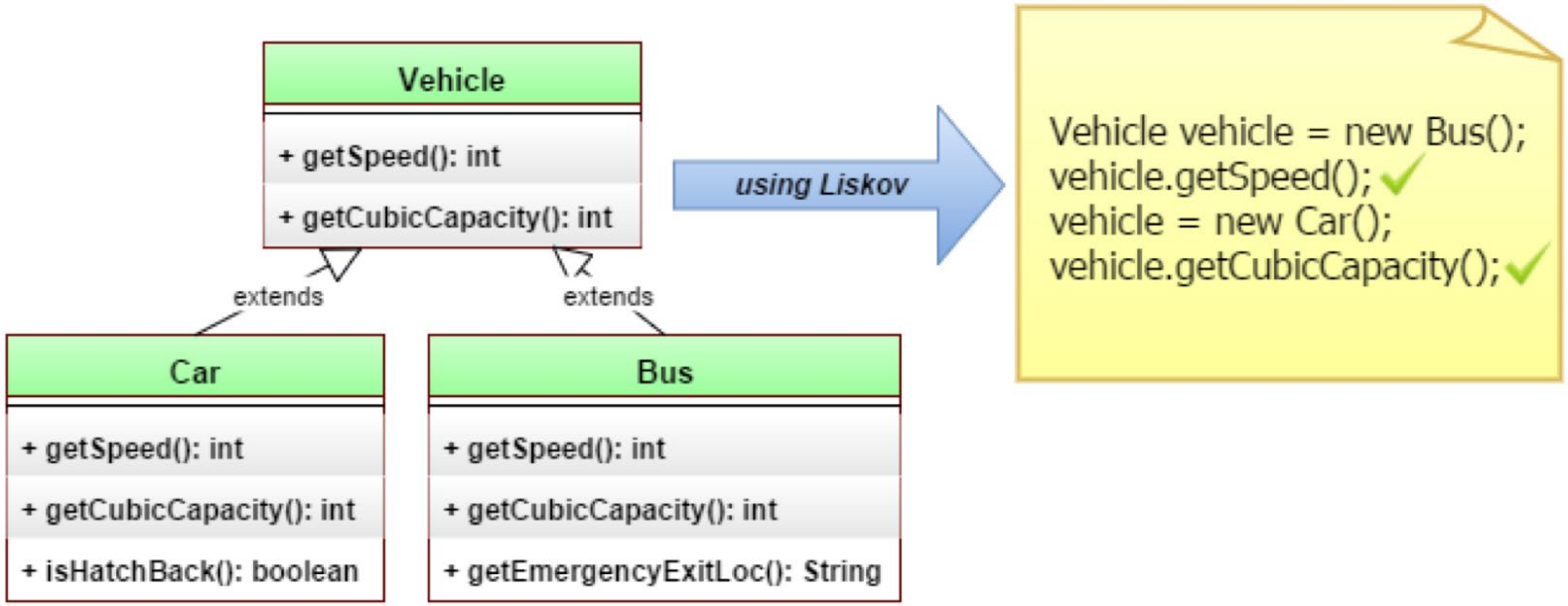
MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

LISKOV SUBSTITUTION PRINCIPLE (LSP)

- *Derived classes must be substitutable for their base classes*
- Code that uses a base class must be able to substitute a subclass without knowing it
- If you create a class with a dependency of a given type, you should be able to provide an object of that type or any of its subclasses without introducing unexpected results and without the dependent class knowing the actual type of the provided dependency
  - If the type of the dependency must be checked so that behavior can be modified according to type, or if subtypes generated unexpected rules or side effects, the code may become more complex, rigid and fragile
- "Subtypes must be substitutable for their base types".
  - In other words, given a specific base class, any class that inherits from it, can be a **substitute** for the base class.

LISKOV SUBSTITUTION PRINCIPLE EXAMPLE

"Derived classes must be substitutable for their base classes"



Can assign an object of type Car or that of type Bus to a reference of type Vehicle. All the functionality which is inherent in base class Vehicle, and is acquired by Bus and Car via inheritance, can be invoked on a reference of type Vehicle

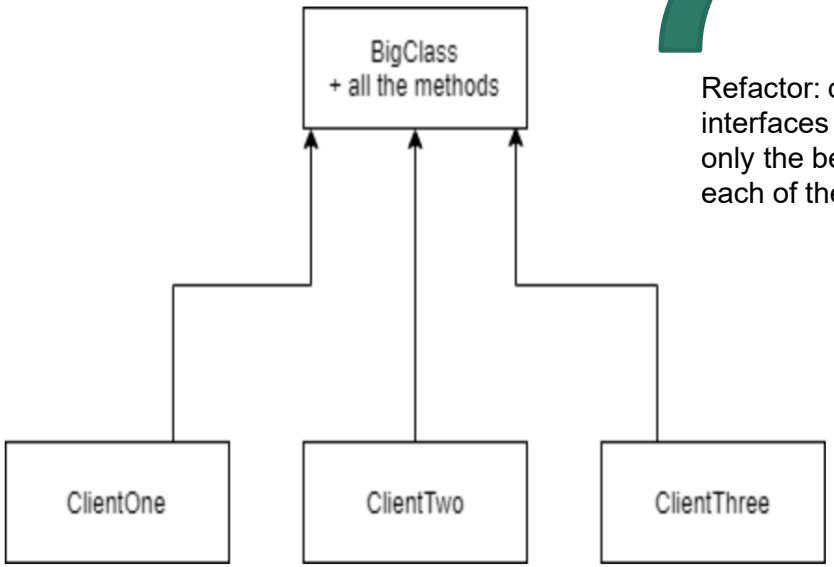
From: https://www.javabrahman.com/programming-principles/liskov-substitution-principal-java-example/

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

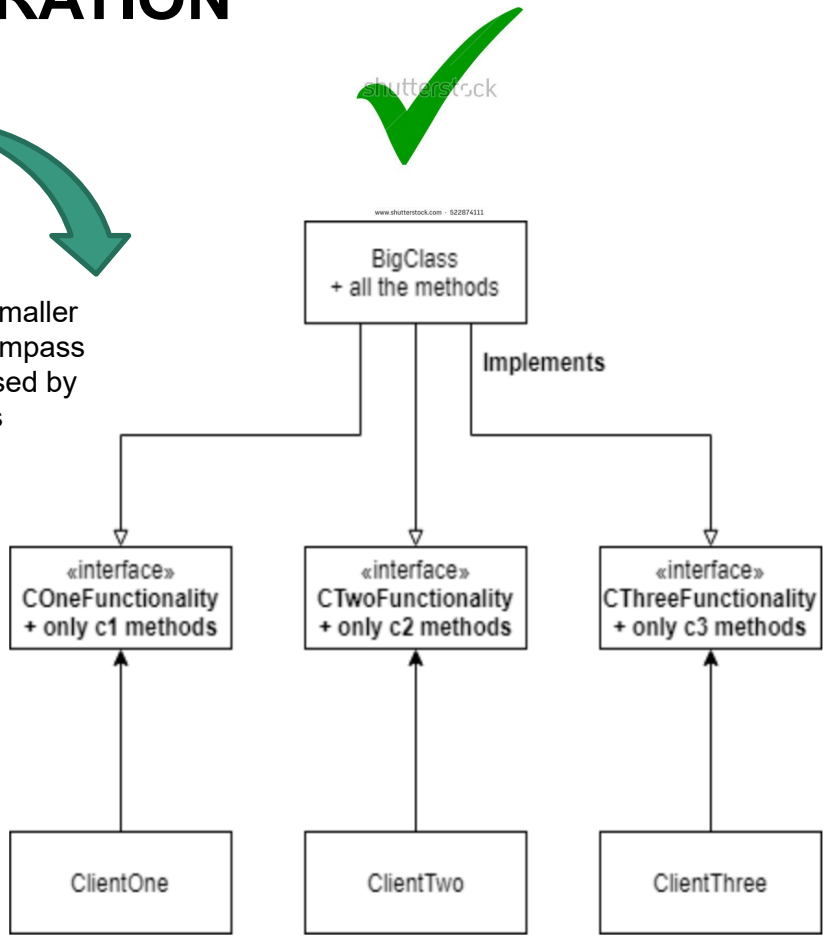
INTERFACE SEGREGATION PRINCIPLE (ISP)

- *Make fine grained interfaces that are client specific*
- Many client-specific interfaces are better than one general-purpose interface
- Clients should not be forced to depend upon interfaces that they do not use
  - An object should not depend on behavior it doesn't need
  - When one class depends upon another, the number of members in the interface that is visible to the dependent class should be minimized
- When you follow the ISP, large classes implement multiple smaller interfaces that group functions according to their usage. The dependents are linked to these for looser coupling, increasing robustness, flexibility and the possibility of reuse

ISP ILLUSTRATION



Refactor: create 3 smaller interfaces that encompass only the behavior used by each of the 3 clients



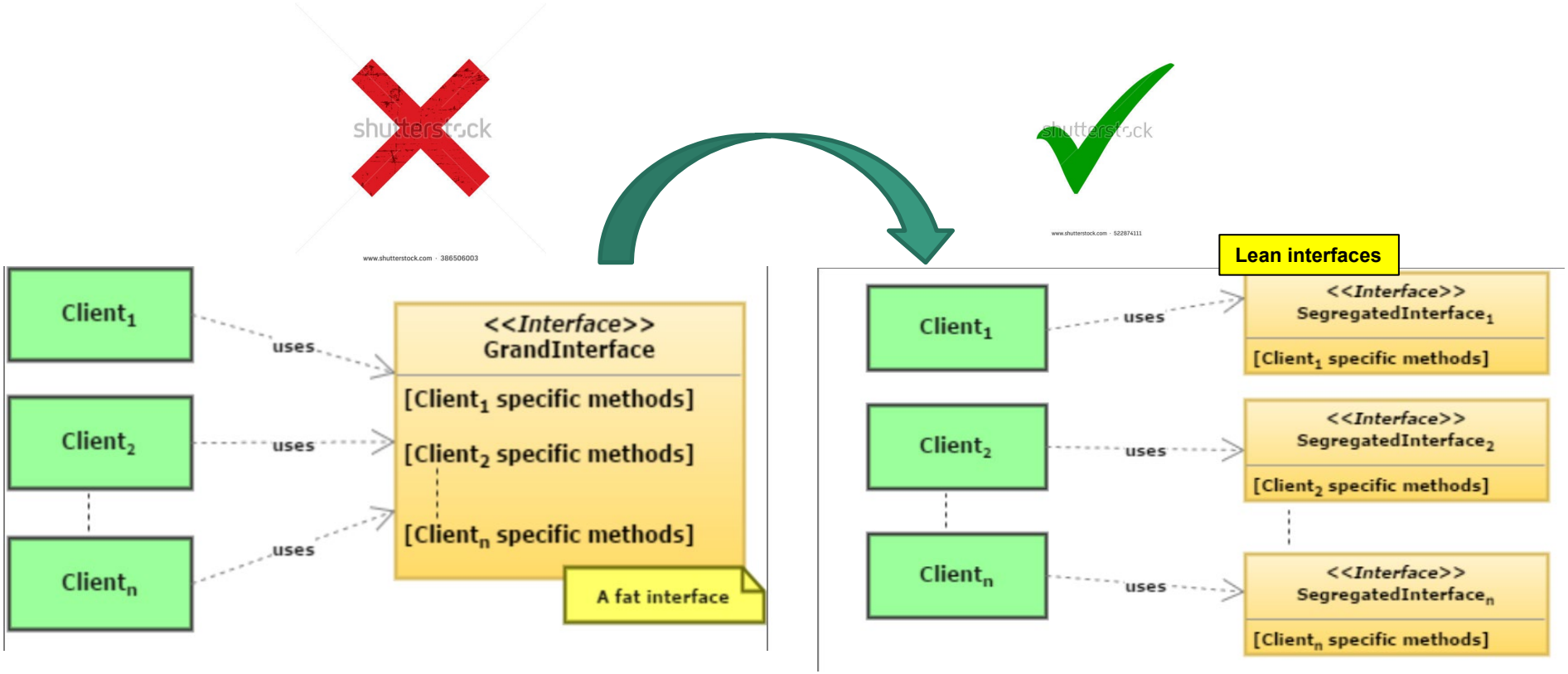
- *ClientOne*, *ClientTwo* and *ClientThree* use only a portion of the functionality from *BigClass*
- But because they reference *BigClass*, the three Client classes implicitly depend on functionality they don't use

- Each client depends only on the functionality they need
- It is even better if instead of the *BigClass* we had smaller classes that implement each of the 3 new interfaces. In reality, this is not always possible, and sometimes a big class can be extremely hard to “chop” into smaller ones

From : https://www.brainstobytes.com/interface-segregation-principle/

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

ISP ILLUSTRATION (CONTINUED)



Interface Segregation Principle avoids the design drawbacks associated with a fat interface by refactoring each *fat interface* into multiple *lean interfaces* (only contain methods which are required for a specific client)

From: https://www.javabrahman.com/programming-principles/interface-segregation-principle-explained-examples-java/

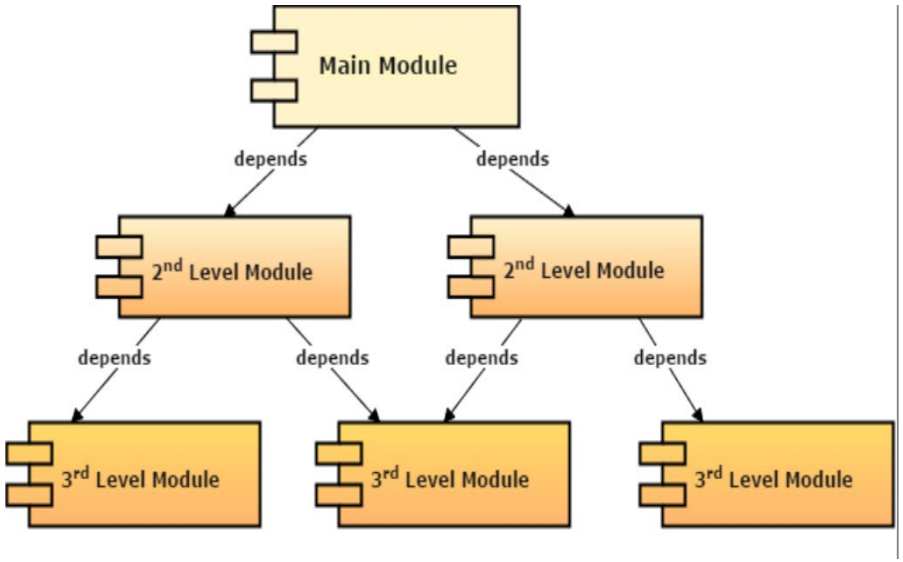
MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

DEPENDENCY INVERSION PRINCIPLE (DIP)

- *Depend on abstractions, not on concretions*
- **High-level modules should not depend on low-level modules. Both should depend on abstractions**
- Abstractions should not depend upon details. Details should depend upon abstractions
- The DIP primarily relates to the concept of layering within applications
  - Lower level modules deal with very detailed functions
  - Higher level modules use lower level classes to achieve larger tasks
  - **Where dependencies exist between classes, they should be defined using abstractions, such as interfaces, rather than by referencing classes directly**
  - This reduces fragility caused by changes in low level modules introducing bugs in the higher layers

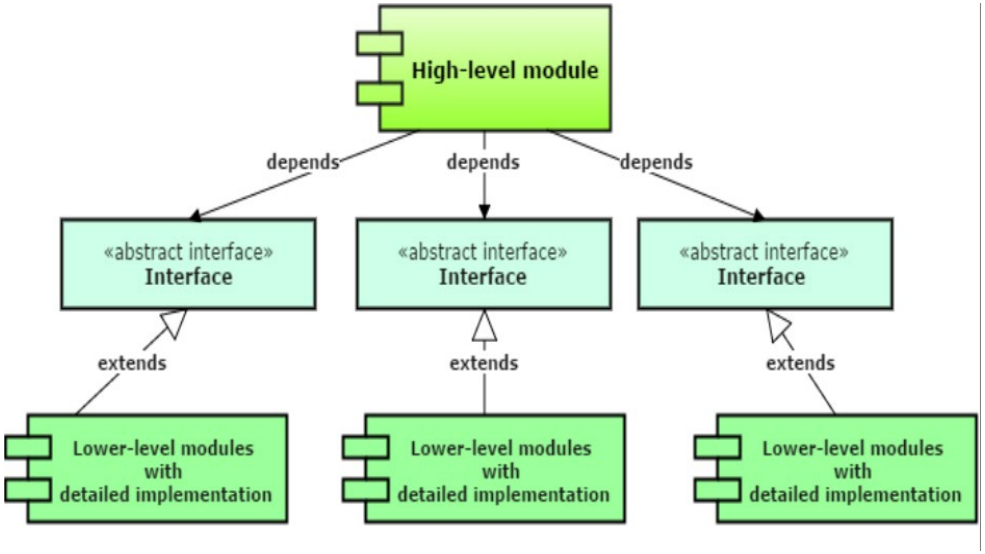
DIP EXAMPLE

Top-down dependency



Main module depends on *2nd Level Modules* which in turn depend on *3rd Level Modules*

Dependency inversion

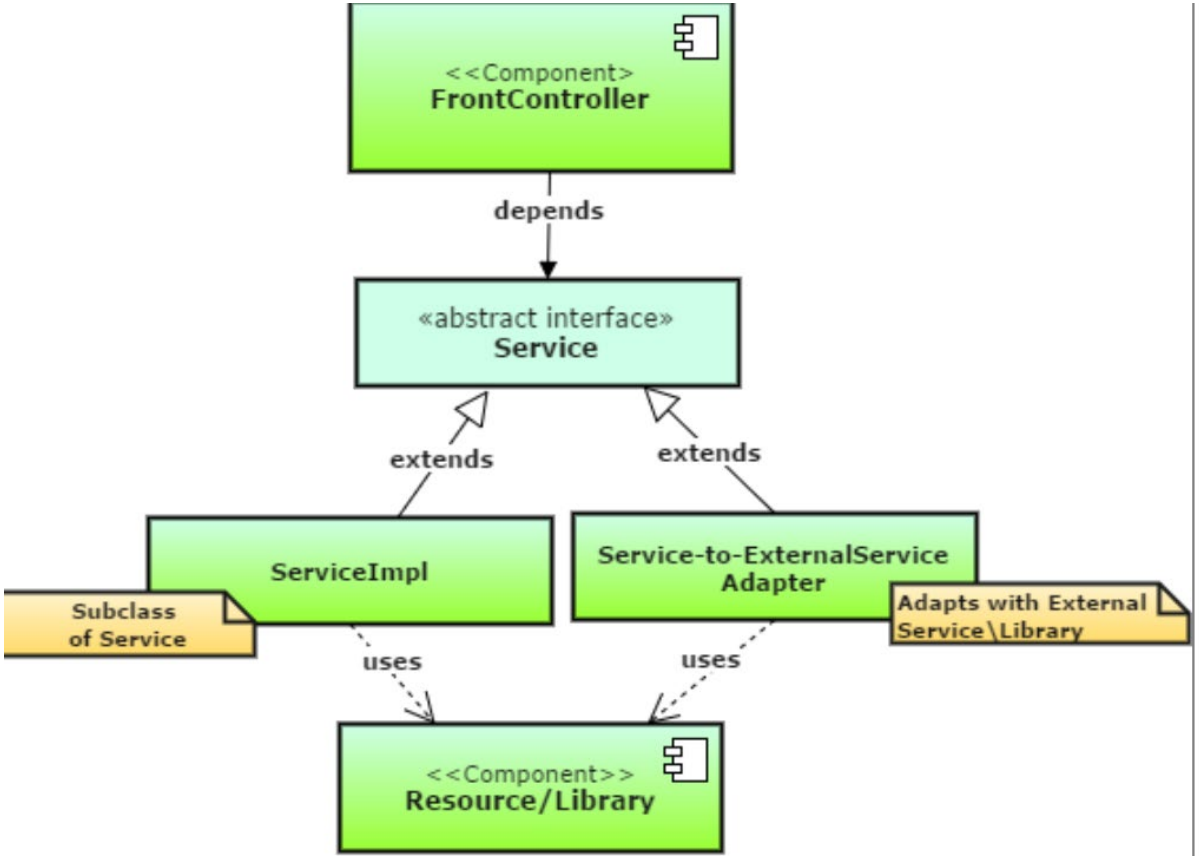


High-level module depends on abstract interfaces which are defined based on the services that the *High-level module* requires from the *Lower-level* modules

From: https://www.javabrahman.com/programming-principles/liskov-substitution-principal-java-example/

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

DIP EXAMPLE: SERVICE LAYER IN THE JAVA SPRING FRAMEWORK



From: https://www.javabrahman.com/programming-principles/liskov-substitution-principal-java-example/

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

EXAMPLES OF SOLID PRINCIPLES VIOLATIONS AND REFACTORING

- https://medium.com/@amit.aggarwal/uml-solid-in-practice-f2c4fc851580
- https://codedesignetc.com/2017/03/09/solid-principles-in-oops/

OUTLINE

- Software detailed design
  - Process
  - Models and their representation
  - UML notations used to represent detailed design
  - Construction techniques and principles
  - Design Patterns
  - Operations and data
  - Software design document
- Assignments

We are here

SOFTWARE DESIGN PATTERNS

- *"*Each pattern describes a *problem* which *occurs over and over again* in our environment, and then describes the *core of the solution* to that problem, in such a way that you *can use this solution a million times over, without ever doing the same thing twice*. " - Christopher Alexander

- A pattern is a *generic template* for a *solution* to a *frequent problem*
- **A pattern must be instantiated each particular use**
  - Can be modified and adapted

- Different granularity of patterns:
  - *Architectural styles or patterns* are for entire systems and sub-systems
  - ***Design patterns* involve several interacting functions or classes**
  - *Data structures & algorithms* are low-level patterns
  - *Idioms* are ways of doing things in particular programming languages

A SOFTWARE DESIGN PATTERNS CLASSIFICATION

- **Creational** Patterns
  - Factory
  - Factory Method
  - Abstract Factory
  - Singleton
  - Builder
  - Prototype
  - Object Pool
- **Structural** Patterns
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Flyweight
  - Proxy

From: http://www.oodesign.com

- **Behavioral** Patterns
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Observer
  - Strategy
  - Template Method
  - Visitor
  - Null Object

https://www.dre.vanderbilt.edu/~schmidt/POSA/POSA2/conc-patterns.html

- **Concurrency** patterns
  - Active Object
  - Monitor Object
  - Half-Sync/Half-Async, Leader/Followers
  - Thread-Specific Storage

DESIGN PATTERNS CLASSIFICATIONS

- There are hundreds of design patterns and multiple patterns classifications (catalogs)
- To make patterns easy to learn, the course textbook presents a subset of design patterns organized into three ***categories*** (broker, generator, reactor), based on:
  - Patterns' *form* – helps grouping them
    - The patterns included here involve a few (usually three) classes, one of which is always a client
  - *Problems* they solve – helps remembering them
- This patterns subset and classification is a starting point for learning patterns
- There are many more additional patterns that may or may not fit into this classification
  - E.g., the "Gang of Four" (GoF) patterns - *Design Patterns*, by E. Gamma, R. Helm, R. Johnson, J. Vlisides
    - http://w3sdesign.com/index0100.php - detailed explanation for (some of the) patterns intent, problem, solution, motivation, applicability, collaboration (behavior), consequences, implementation, and sample code

DESIGN PATTERNS CLASSIFICATION IN TEXTBOOK

From: *Introduction to Software Engineering Design*, by C. Fox (textbook)

- **Broker patterns**
  - A *client* needs a service from a *supplier*, and a *broker* mediates the interaction between client and supplier (e.g., **Façade, Mediator, Adapter, Proxy**)
  - Analogy: Brokers are like stock brokers who mediate interactions between an investor (client) and the stock market (supplier)
- **Generator patterns**
  - A *client* needs a new instance of a product, and a *generator* class supplies the instance (e.g., **Factory, Singleton, Prototype**)
  - Analogy: Generators are like interior designers who obtain material from manufacturers (products) on behalf of their clients
- **Reactor patterns**
  - A *client* needs to respond to an event in a target. The client delegates this responsibility to a *reactor* (e.g., **Command and Observer**)
  - Analogy: Reactors are like lawn service companies that respond to conditions in a lawn (target) on behalf of a homeowner (client)

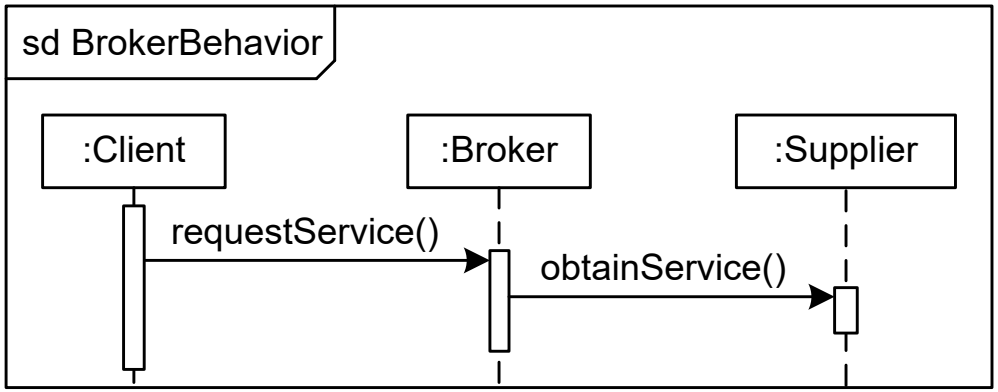
BROKER PATTERNS

- The *Client* must access the *Broker* and the *Broker* must access the *Supplier*
  - Most Broker patterns elaborate this basic structure

Legend:
Components
Rules

Structure

Behavior



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

BROKER PATTERNS BENEFITS

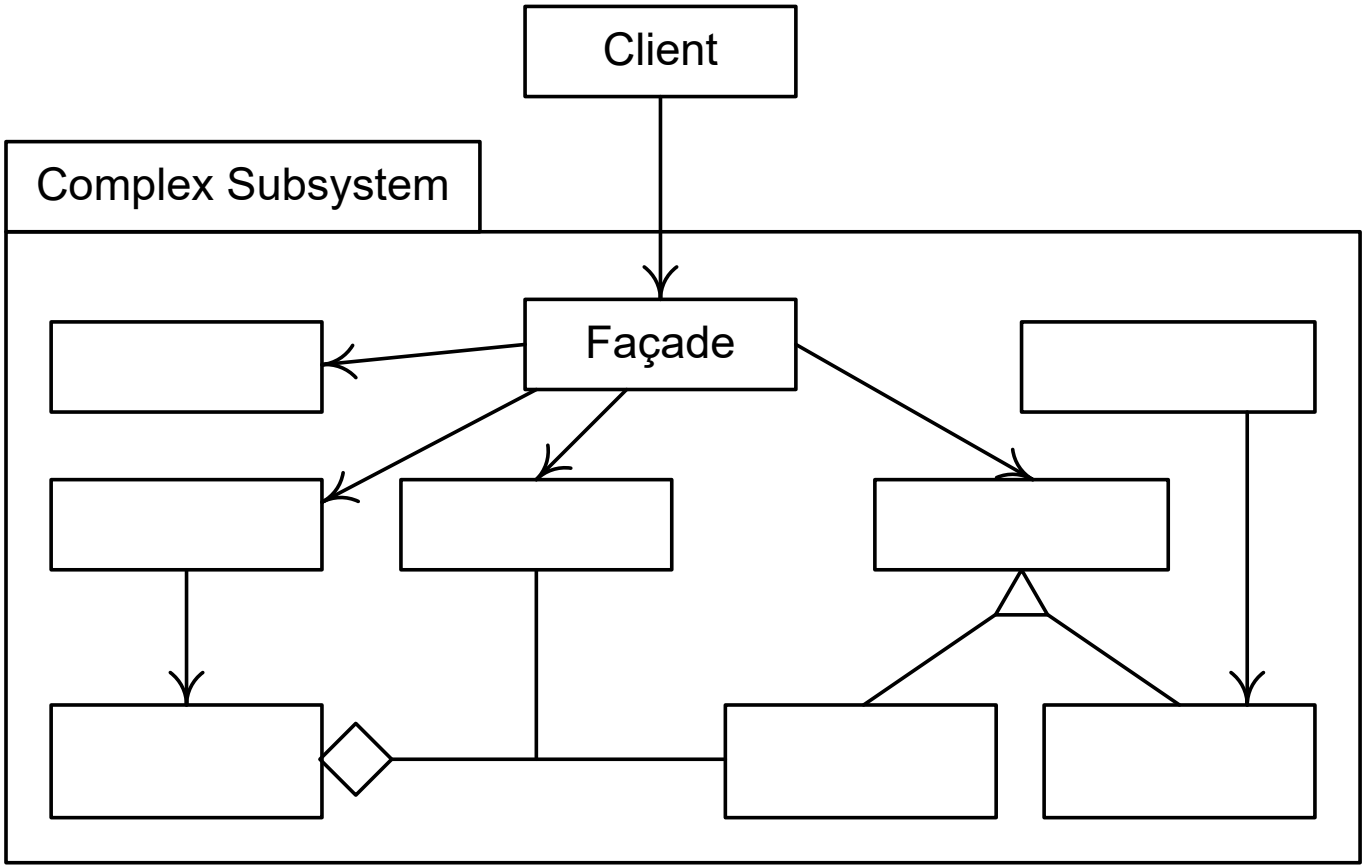
- *Simplify the Supplier*
  - A Broker can augment the Supplier's services
- *Decompose the Supplier*
  - A complex Supplier can offload some of its responsibilities to a Broker
- *Facilitate Client/Supplier Interaction*
  - A Broker may present a different interface, handle interaction details, etc.

FAÇADE PATTERN

- Façade pattern eases interaction between a client and a **sub-system of suppliers** by **providing a simpler interface to the sub-system**
- The broker class is a façade that provides simplified sub-system services to clients
  - Analogy: a travel agent

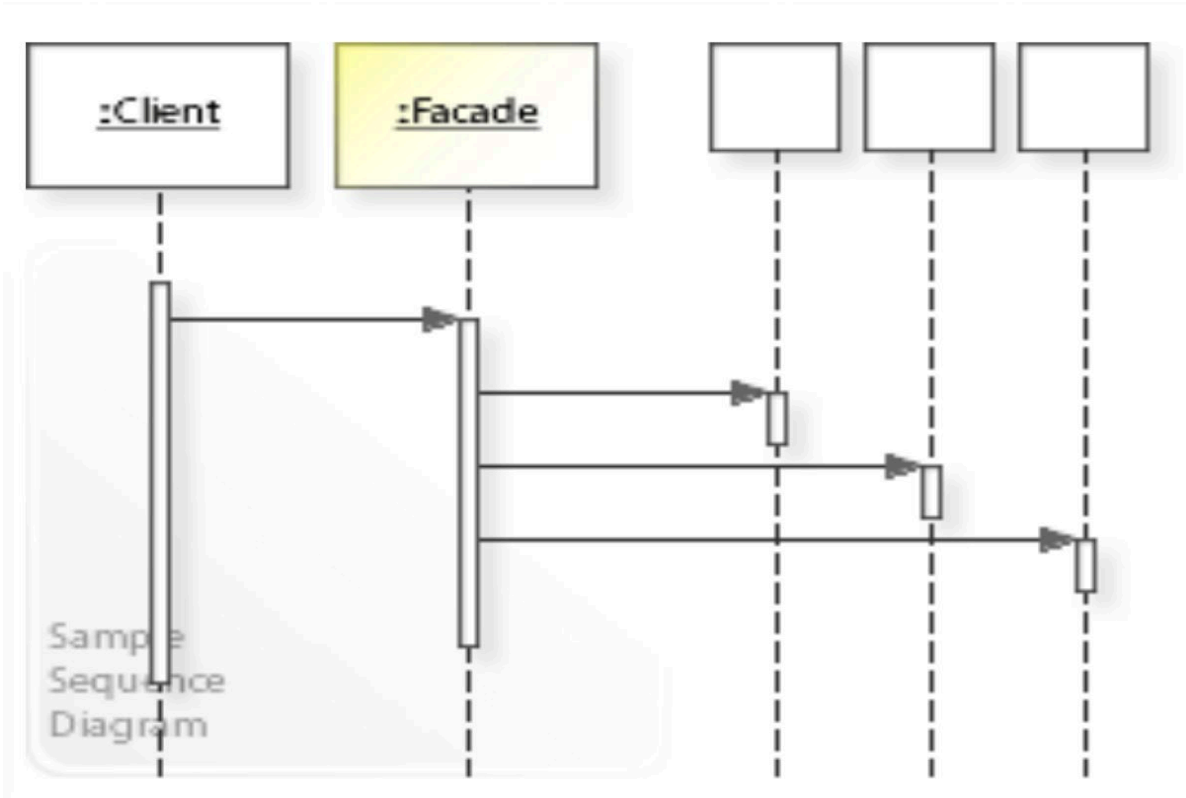
- Examples:
  - Interface to a compiler
  - Interface to a memory management system

FAÇADE PATTERN STRUCTURE



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

FAÇADE PATTERN BEHAVIOR



A *Client* object works through a *Facade* object to access many different objects in a subsystem.
The Client object calls an operation on the Facade object.
Facade delegates the request to the objects in the subsystem that fulfill the request.
Facade may do work of its own before and/or after forwarding a request

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

WHEN TO USE THE FAÇADE PATTERN

- When there is a need to provide a simplified interface to a complex sub-system
- Façades can also help decouple systems
  - If the façade mediates all interaction with a client, then the sub-system can be changed without affecting the client
- A façade may work like an adapter by providing a new interface to a sub-system

- Code example for using the Façade pattern

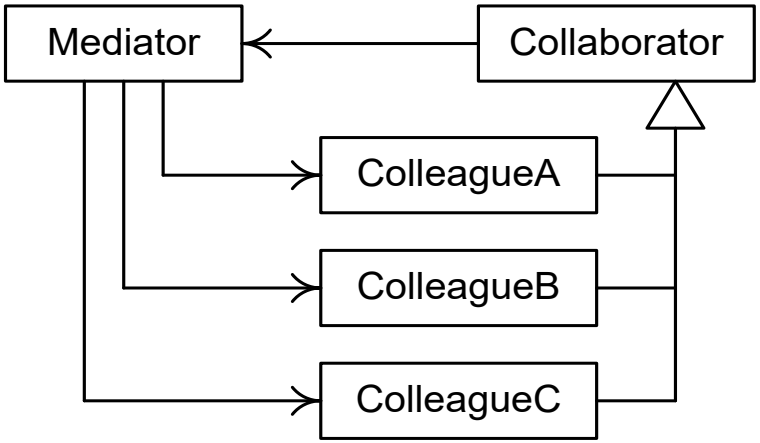
MEDIATOR PATTERN

- Reduces coupling and simplifies code **when several objects must negotiate a complex interaction**
- **Classes interact only with a mediator class rather than with each other**
- Classes are coupled only to the mediator where interaction control code resides
- Mediator is like a multi-way Façade pattern
- Provides means to make control more centralized

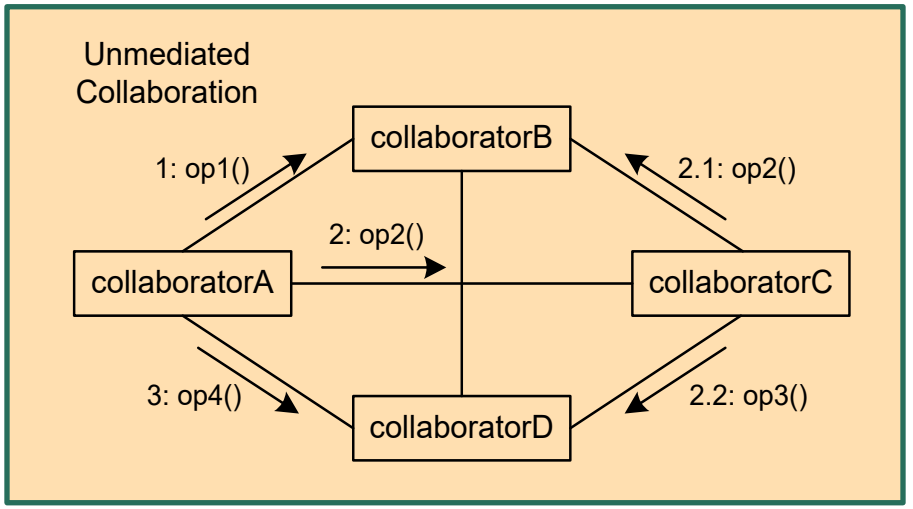
- Analogy: a meeting scheduler

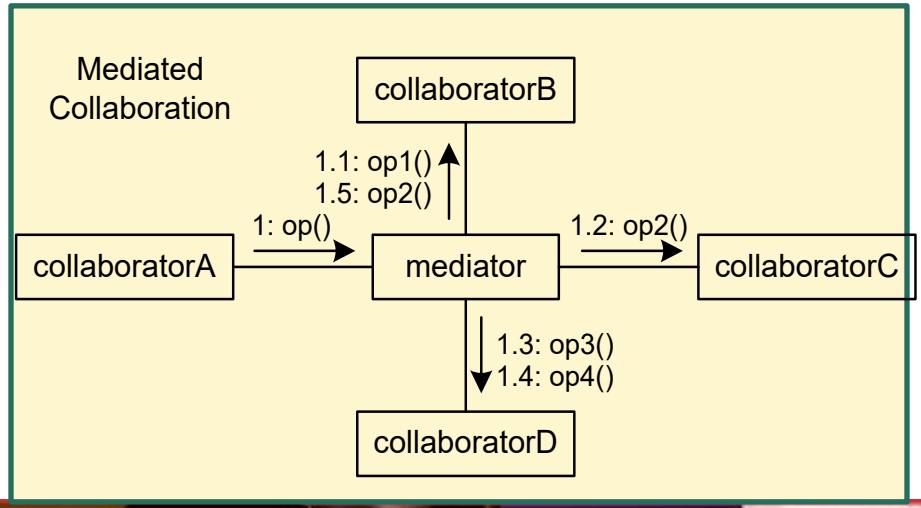
MEDIATOR PATTERN

Structure



Behavior

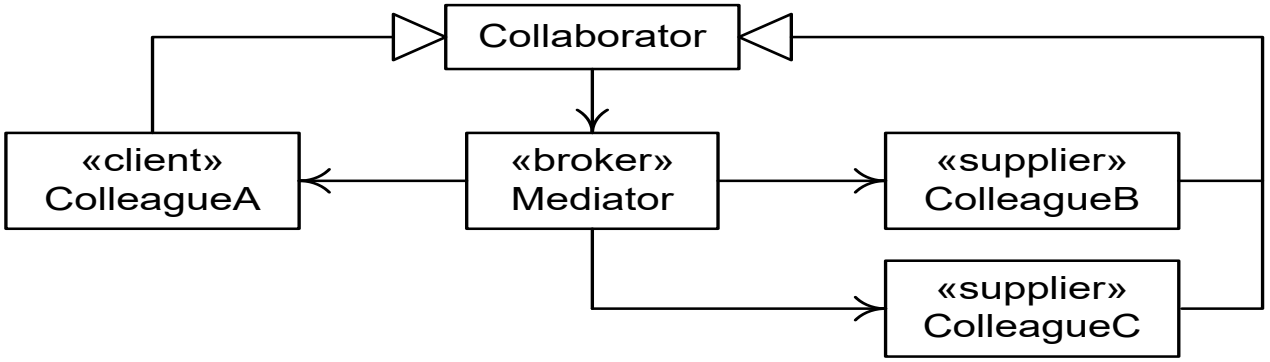




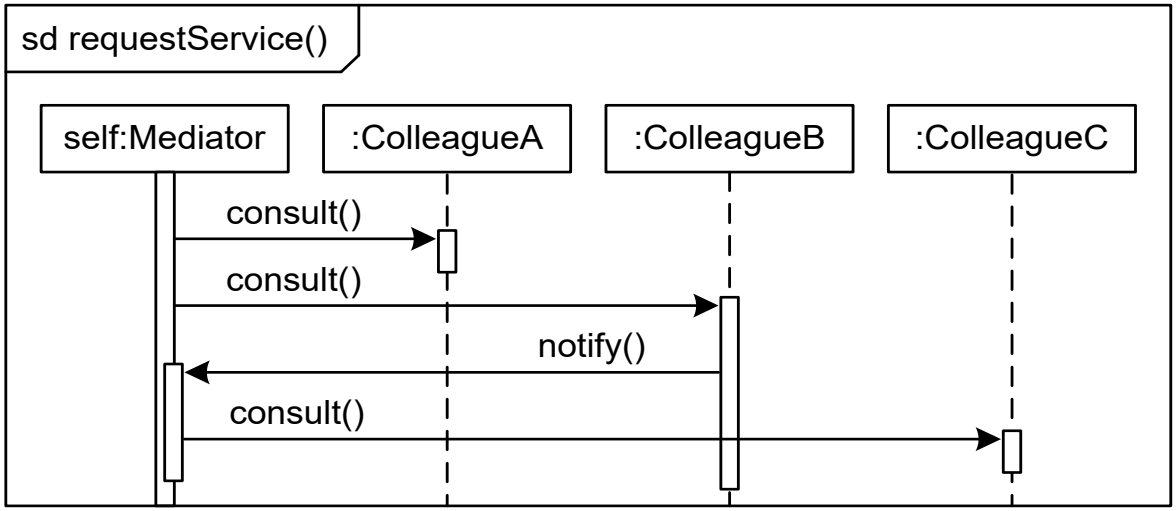
MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

MEDIATOR AS A BROKER PATTERN

Structure



Behavior



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

WHEN TO USE THE MEDIATOR PATTERN

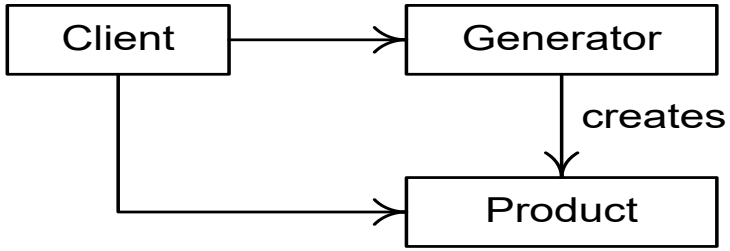
- When a complex interaction between collaborators must be encapsulated, in order to:
  - Decouple collaborators
  - Simplify the collaborators
  - Centralize control of an interaction
- **Disadvantage**: Using a mediator may compromise performance and reliability/availability (if mediator fails)
- Code examples for using the Mediator pattern:
  - Java
    - https://www.baeldung.com/java-mediator-pattern
      - **Follows SRP and OCP design principles**
    - https://sourcemaking.com/design_patterns/mediator/java/2
  - Python: https://sourcemaking.com/design_patterns/mediator/python/1

GENERATOR PATTERNS

- Instance Creation - two ways to create objects:
  - Instantiating a class using one of its constructors
  - Cloning an existing object
- **Clients may use another class to create an instance on their behalf**; this is the essence of the *generator pattern category*
- Analogy: a tailor

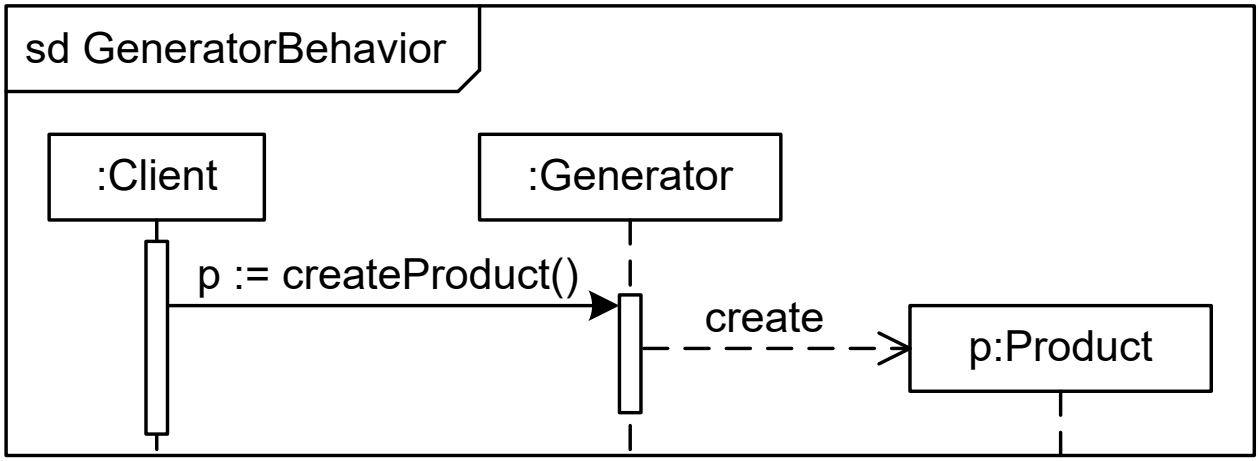
GENERATOR PATTERNS

Structure



The *Client* must access the *Generator* that creates an instance of the *Product* and provides it to the Client

Behavior



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

GENERATOR PATTERNS BENEFITS

- *Product Creation Control*
  - A generator can mediate access to constructors so that only a certain number or type of product instances are created
- *Product Configuration Control*
  - A generator can take responsibility for configuring product instances
- *Client and Product Decoupling*
  - A generator can determine how to create product instances for a client

FACTORY METHODS

- A *Generator* must have an operation that creates and returns *Product* instances
  - A *factory method* is a **non-constructor operation that creates and returns class instances**

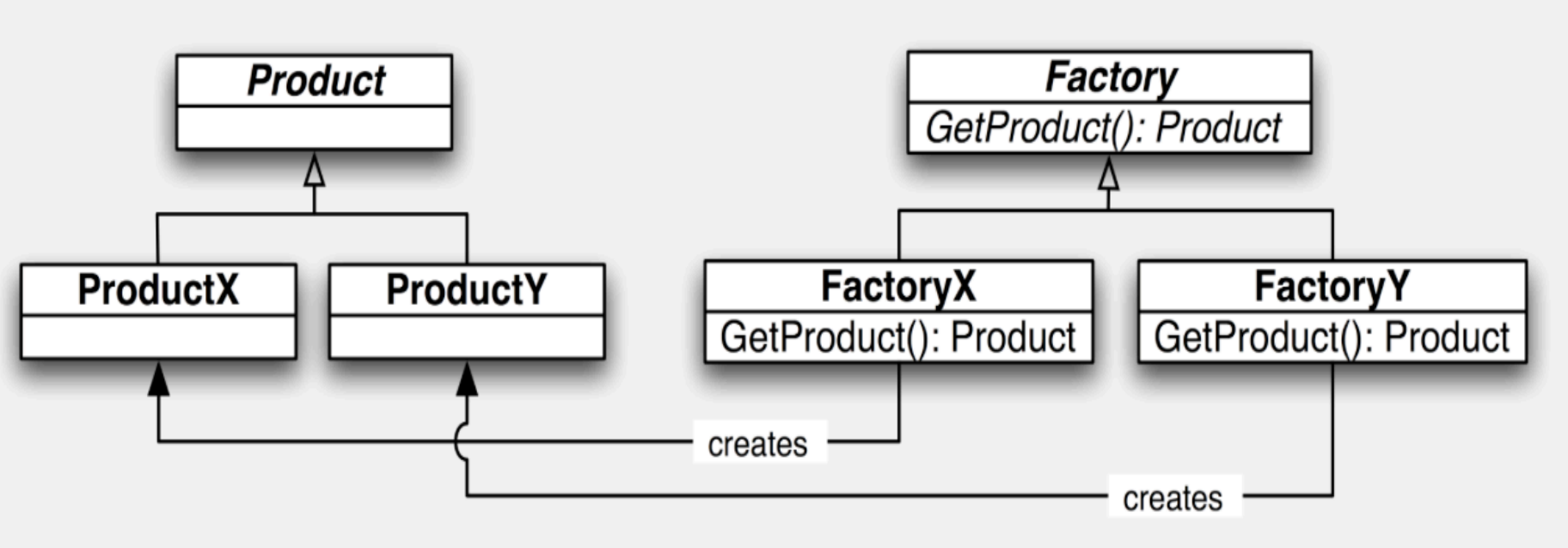
- Factory method capabilities:
  - Access to product constructors can be restricted
  - Private data can be provided to new product objects
  - Product objects can be configured after creation
  - Product class bindings can be deferred until runtime

FACTORY PATTERNS

- All generator patterns have factory methods
- Factory patterns configure participating classes in certain ways to decouple the client from the product
- Interfaces are used to
  - Change the generator
  - Change the product instances
- Analogy: car factories

- Factory patterns:
  - **Factory Method**—Uses interfaces and abstract classes to decouple the client from the generator class and the resulting products
  - **Abstract Factory**—Has a generator that is a container for several factory methods, along with interfaces decoupling the client from the generator and the products

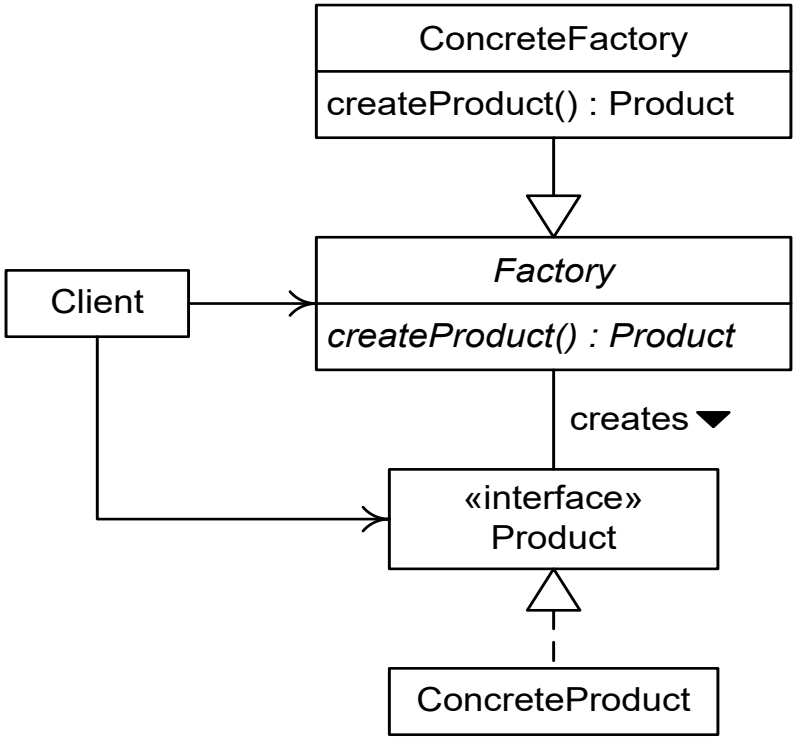
FACTORY PATTERN REPRESENTATION



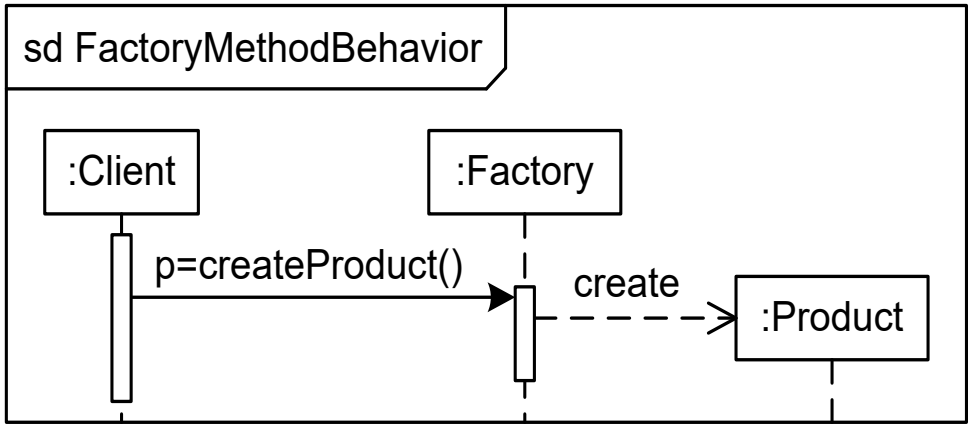
MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

FACTORY METHOD PATTERN

Structure



Behavior



- The *generator* usually contains both factory methods and other methods
- Analogy: different car factories producing the same kind of car (SUVs for example).

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

WHEN TO USE THE FACTORY METHOD PATTERN

- When there is a need to decouple a client from a particular product that it uses
- To relieve a client of responsibility for creating and configuring instances of a product
- When clients cannot anticipate the class of objects to create

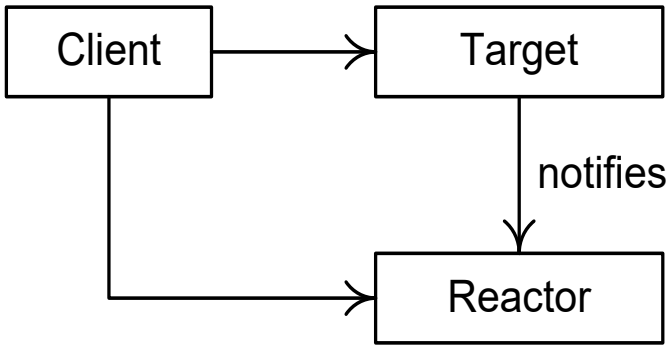
- Code example for using the Factory pattern

REACTOR PATTERNS

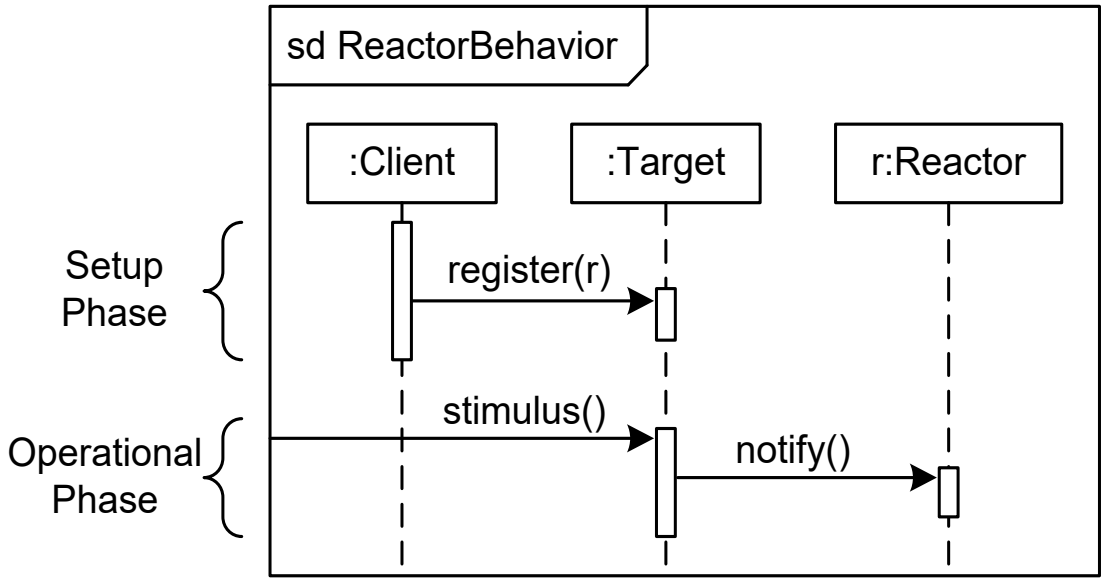
- Reactor patterns assist in ***event-driven design***
- ***Event-driven design*** is an approach to program design that focuses on *events to which a program must react*
  - An *event* is a significant occurrence
- *Event handlers* are components that react to or respond to events

REACTOR PATTERNS

Structure



Behavior



The *Client* must access the *Target* and the *Reactor* so it can register the *Reactor* with the *Target*

Behavior phases:

Setup Phase -The *Client* registers the *Reactor* with the *Target*.

Client interacts with the *Target*

Operational Phase -The *Reactor* responds to events in the *Target* on behalf of the *Client*.

Client is not involved in the response

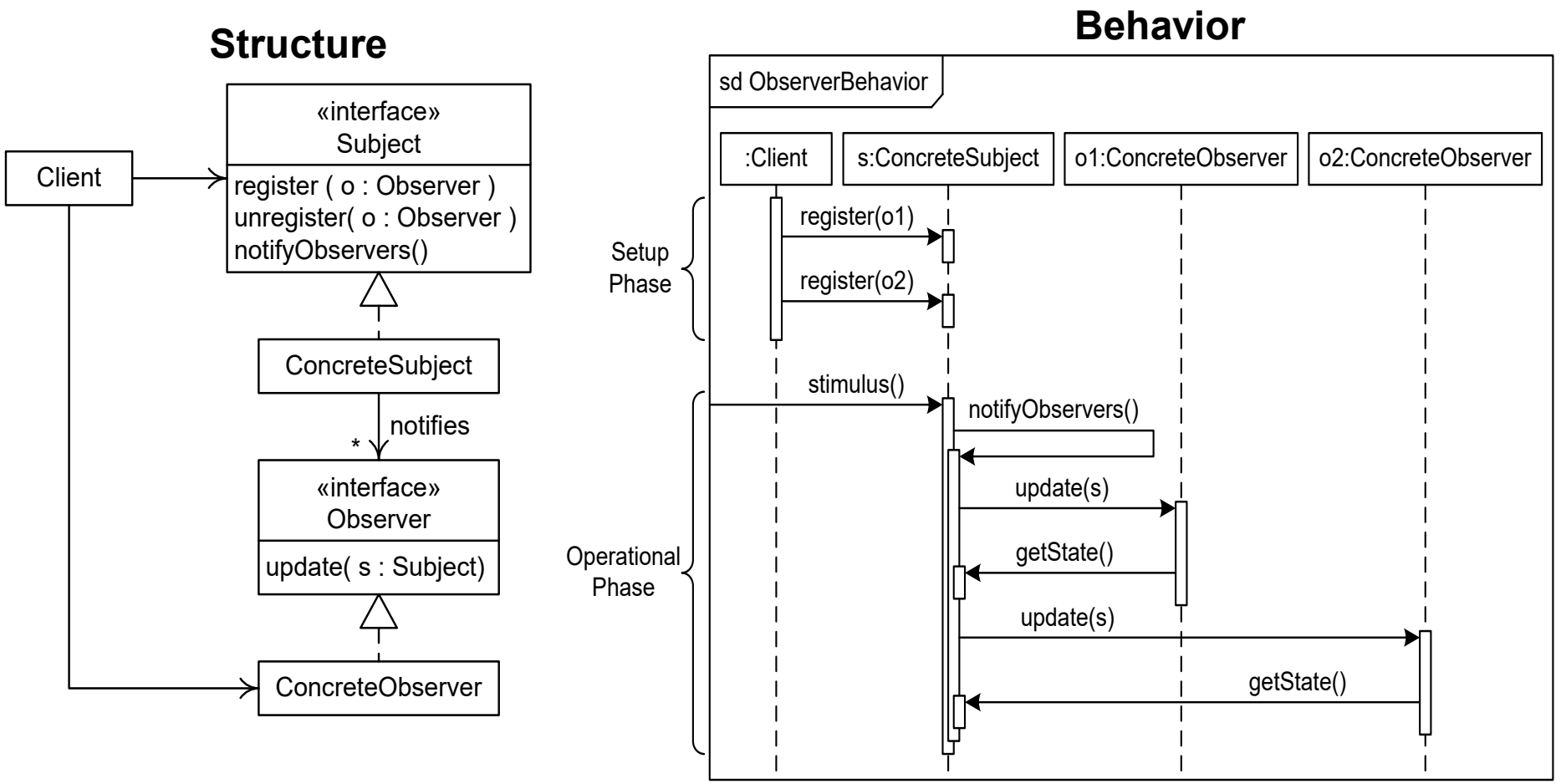
MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

REACTOR PATTERN BENEFITS

- *Client and Target Decoupling*
  - Once the client registers the reactor with the target, the client and target need not interact again
- *Low Coupling between Reactor and Target*
  - The target only knows that the reactor is a receiver of event notifications
- *Client Decomposition*
  - The reactor takes over client responsibilities for reacting to target events
- *Operation Encapsulation*
  - The event handler in a reactor is an encapsulated operation that can also be invoked in other ways for other reasons

The reactor patterns help decouple targets from both their clients and reactors

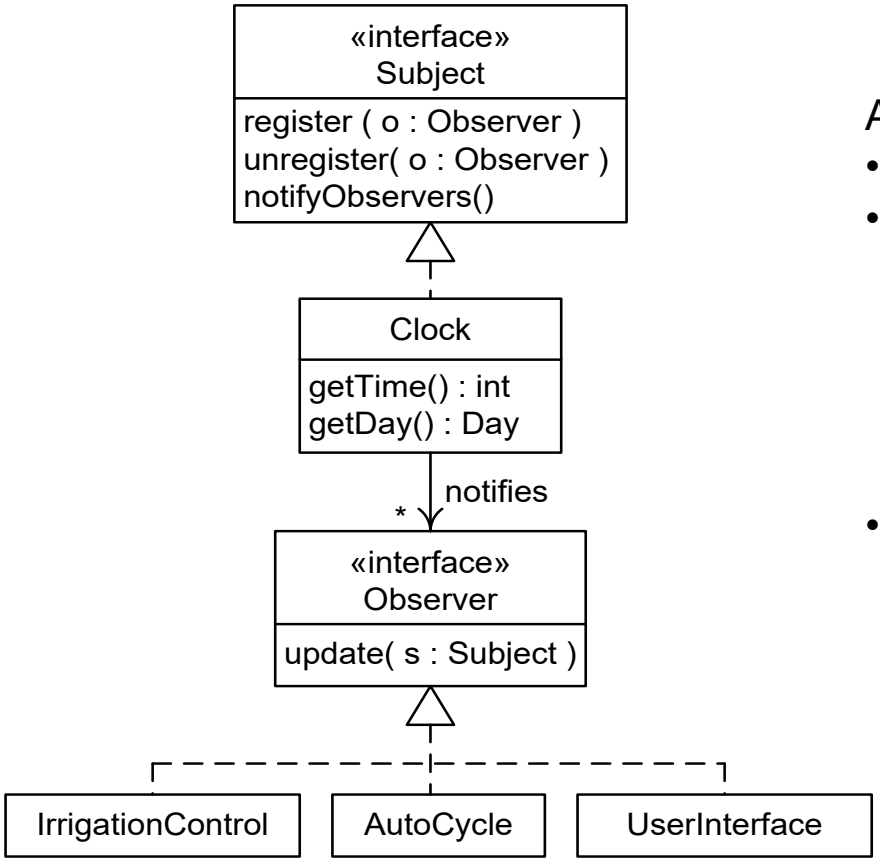
OBSERVER PATTERN



The *observers* register with the *subject (*the *client* might do the registration of the observers)
The *subject* then notifies its observers of changes, and the *observers* query the *subject* to determine how to react

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

EXAMPLE OF USING THE OBSERVER PATTERN



AquaLush modules are driven by time events:

- *Clock* is the *subject*
- Clock *observers:*
  - *IrrigationControl* – decide when to start irrigation
  - *AutoCycle* – controls automatic irrigation
  - *UserInterface* – controls user interaction, including setting the Clock
- The *observers* register with the *subject* (clock) (*client* class is not included in this diagram)

From *Introduction to Software Engineering Design*, by C. Fox

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

WHEN TO USE THE OBSERVER PATTERN

- When the observed object (*subject*) changes state, all dependents get notified and updated automatically
- When one object must react to changes in another object, especially if the objects must be loosely coupled
  - User interface components
  - Clock-driven components

- The main **drawback** of the Observer pattern is that notification may be expensive

- Code examples for using the Observer pattern
  - https://www.baeldung.com/java-observer-pattern

HOW TO SELECT A DESIGN PATTERN?

[From: Design Patterns – Elements of Reusable Object Oriented Software]

- Identify the problem/need to be addressed
  - E.g., what needs to be changeable, or cause(s) of redesign

USE THE RIGHT TOOL FOR THE JOB!

Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory (87)	
	Builder (97)	how a composite object gets created
	Factory Method (107)	subclass of object that is instantiated
	Prototype (117)	class of object that is instantiated
	Singleton (127)	the sole instance of a class
Structural	Adapter (139)	interface to an object
	Bridge (151)	implementation of an object
	Composite (163)	structure and composition of an object
	Decorator (175)	responsibilities of an object without subclassing
	Facade (185)	interface to a subsystem
	Flyweight (195)	storage costs of objects
	Proxy (207)	how an object is accessed; its location
Behavioral	Chain of Responsibility (223)	object that can fulfill a request
	Command (233)	when and how a request is fulfilled
	Interpreter (243)	grammar and interpretation of a language
	Iterator (257)	how an aggregate's elements are accessed, traversed
	Mediator (273)	how and which objects interact with each other
	Memento (283)	what private information is stored outside an object, and when
	Observer (293)	number of objects that depend on another object; how the dependent objects stay up to date
	State (305)	states of an object
	Strategy (315)	an algorithm
	Template Method (325)	steps of an algorithm
	Visitor (331)	operations that can be applied to object(s) without changing their class(es)

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

SELECTING A DESIGN PATTERN – EXAMPLES

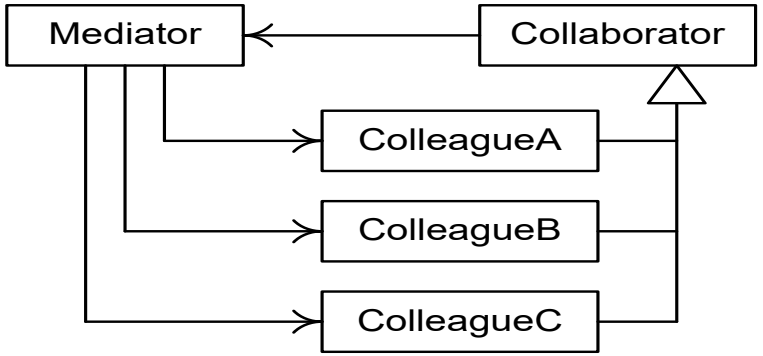
- Need to **recognize the problems** that are potentially addressed by design patterns, and what pattern is appropriate, e.g.,:
  - *Tell several objects that the state of some other object has changed*
    - use the **Observer pattern**
  - *Tidy up the interfaces to a number of related objects that have often been developed incrementally*
    - use the **Façade pattern**
  - *Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented*
    - use the **Iterator pattern**
  - *Allow for the possibility of extending the functionality of an existing class at run-time*
    - use the **Decorator pattern**

USING DESIGN PATTERNS HEURISTICS

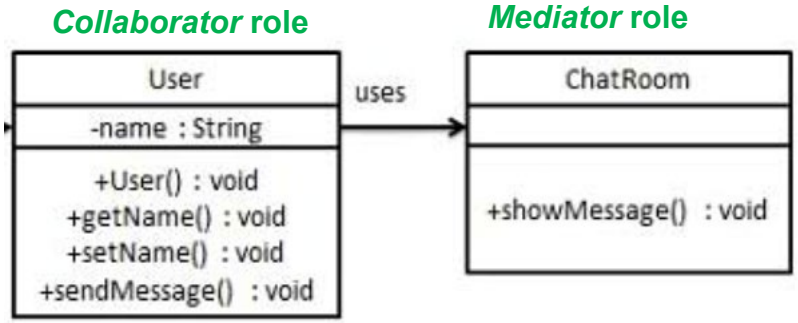
- After a pattern was selected, perform the following (suggested) steps:
  - Become familiar with the pattern
    - Structure, participants and their responsibilities, rules, behavior (collaborations)
    - Look for code samples
  - Choose names for **pattern participants** (roles) that are **meaningful in the concrete application context**
  - Define the participating classes
    - Define class interfaces, establish inheritance (and other) relations
  - Choose (and define) **application specific names for the operations** in the patterns
  - Implement the operations to carry out the responsibilities and collaborations in the pattern

USING DESIGN PATTERNS - EXAMPLE

Generic Mediator pattern



Pattern usage/instantiation for a *Chat app*



It is not a good idea to connect each participant to all the others because the number of connections would be really high, there would be technical problems due to proxies and firewalls, etc... . The most appropriate solution is to have a "hub" where all participants will connect; this "hub" plays the *mediator* role => Communication between *Users* is mediated by the *ChatRoom*

Java code for Chat app

```
import java.util.Date;

public class ChatRoom {
   public static void showMessage(User user, String message){
      System.out.println(new Date().toString() + " [" + user.getName() + "] :
   }
}
```

```
public class User {
   private String name;

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public User(String name){
      this.name  = name;
   }

   public void sendMessage(String message){
      ChatRoom.showMessage(this,message);
   }
}
```

```
public class MediatorPatternDemo {
   public static void main(String[] args) {
      User robert = new User("Robert");
      User john = new User("John");

      robert.sendMessage("Hi! John!");
      john.sendMessage("Hello! Robert!");
   }
}
```

Example Output of Chat app

```
Thu Jan 31 16:05:46 IST 2013 [Robert] : Hi! John!
Thu Jan 31 16:05:46 IST 2013 [John] : Hello! Robert!
```


MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

DESIGN PATTERNS ARE NOT A MAGIC BULLET

- Patterns are not a simple cookie cutter
- You need to consider the context
- Each pattern has disadvantages or consequences
  - For example, Observer pattern could cause a slow and inefficient cascade of updates
- When you use your pattern, it might trigger the need for one or more related patterns
  - A “pattern language” is a group of connected patterns
- It’s easy to go “pattern happy”
  - make the application extra complicated just so we can show off how many patterns we can use
- Don’t be afraid to *remove* a pattern from your design, if needed
- If you don’t need it now, don’t do it now; possibly "You Aren't Gonna Need It” (YAGNI)

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

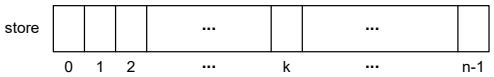
OUTLINE

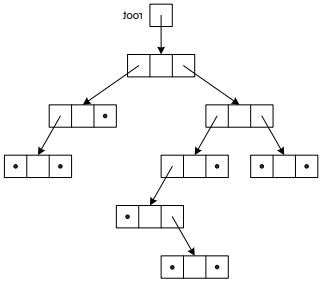
- Software detailed design
  - Process
  - Models and their representation
  - UML notations used to represent detailed design
  - Construction techniques and principles
  - Design Patterns
  - Operations and data
  - Software design document
- Assignments

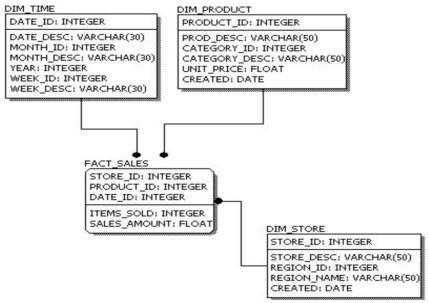
We are here

OPERATIONS AND DATA SPECIFICATION

- Closer to code ("low-level" detailed design)
- Data
  - Data structures
    - Trees, lists, queues
  - File structures
  - Database physical model





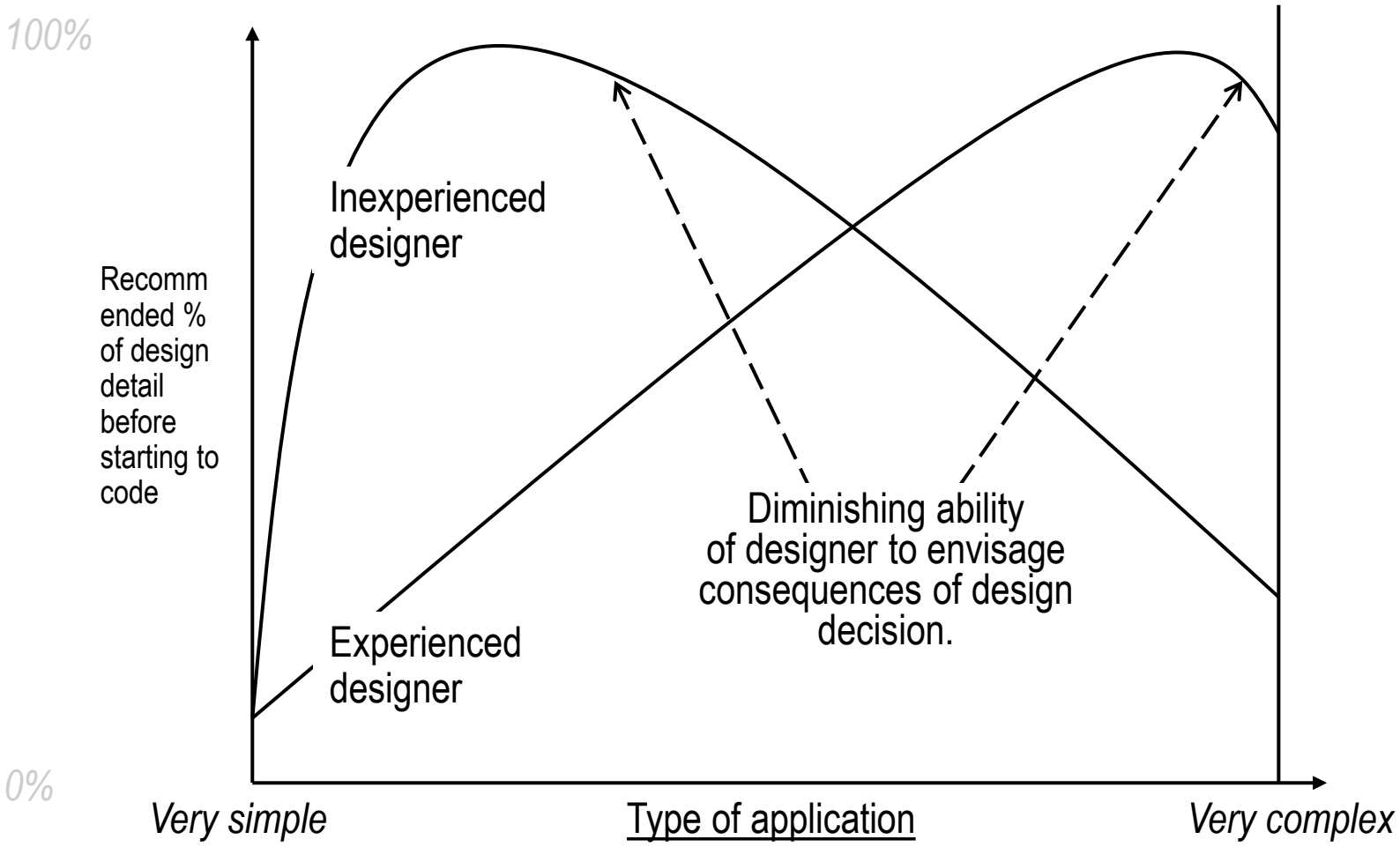


- Operations – algorithms
  - Representation:
    - Pseudo code
    - Flow charts or UML activity diagrams

```
Inputs:   array a, lower bound lb, upper bound ub, search key
Outputs:  location of key, or -1 if key is not found
lo = lb
hi = ub
while lo <= hi and key not found
   mid = (lo + hi) / 2
   if      ( key = a[mid] ) then key is found
   else if ( key < a[mid] ) then hi = mid-1
   else                          lo = mid+1
if key is found then return mid
else                 return -1
```


MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

HOW MUCH DESIGN DETAIL BEFORE INITIAL CODING?

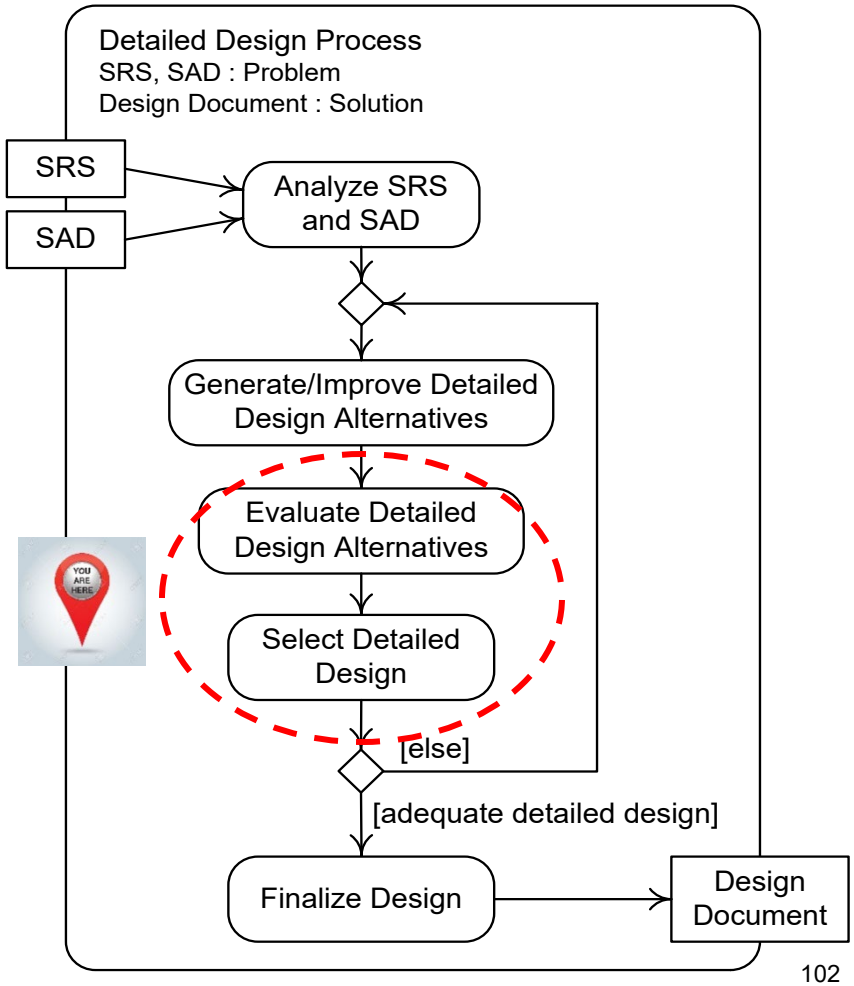


From *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003)

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

EVALUATE AND SELECT DETAILED DESIGN

- Evaluate one or more detailed design solutions relative to:
  - Fit for purpose
  - Follows design principles
- Select the solution with higher evaluation
- *Similar to selecting an architecture solution*



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

OUTLINE

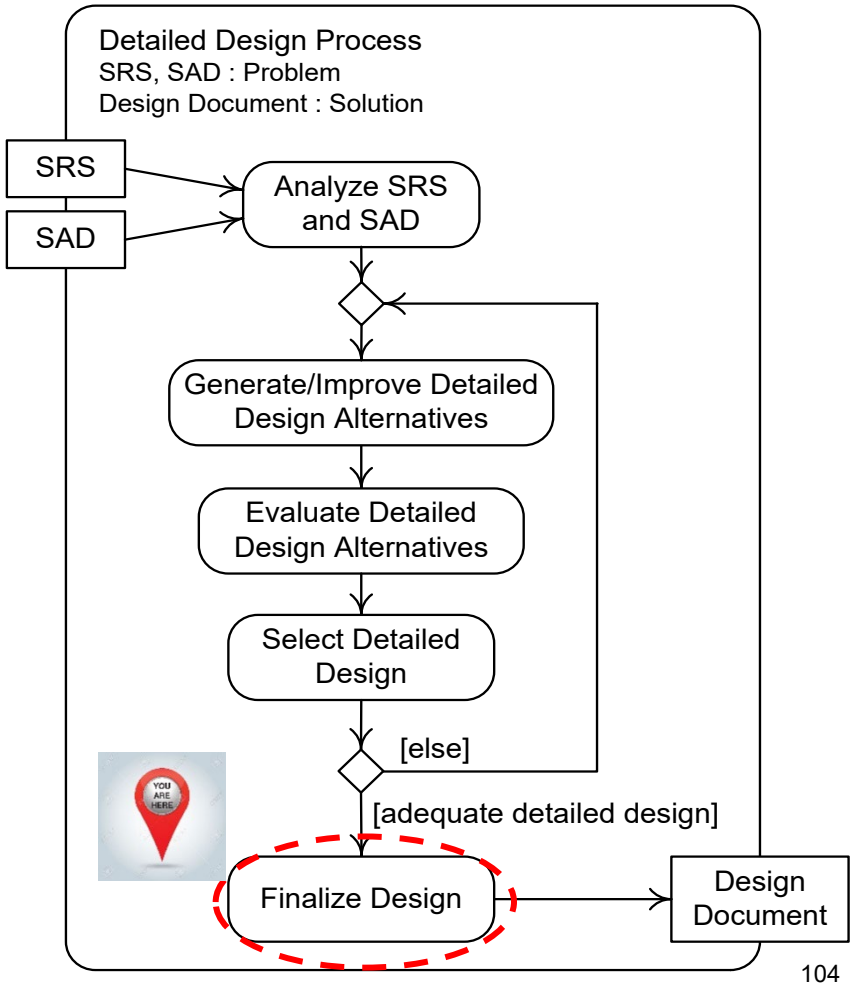
- Software detailed design
  - Process
  - Models and their representation
  - UML notations used to represent detailed design
  - Construction techniques and principles
  - Design Patterns
  - Operations and data
  - Software design document

We are here

- Assignments

FINALIZE DESIGN

- **Document** the detailed design specifications in a software detailed design document (SDD)
  - It is a ***deliverable***
- **Finalize** the design - checking the design to make sure it is of sufficient quality and is well documented



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

DESIGN DOCUMENT

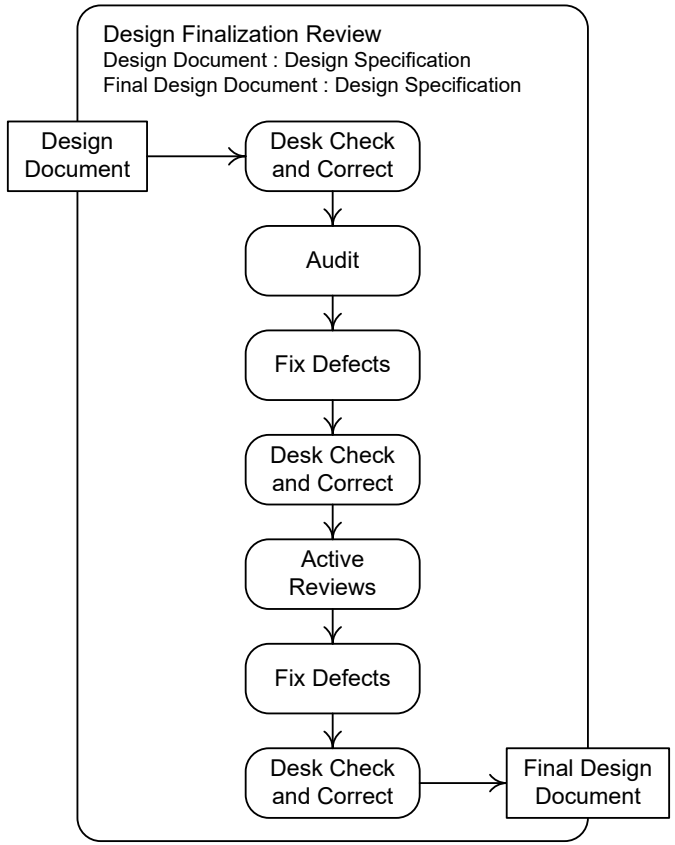
- A **software architecture and detailed design document** (SADD) consists of a Software Architecture Document (SAD), plus a **detailed design document** (SDD)
  - There can be one document to include both architecture and detailed design, or two separate documents
- An example SDD template:
  - **Detailed Design Key Decisions and Rationale**
  - **Detailed Design Structure**
  - **Detailed Design Behavior**
  - **Data Structures (Database physical design)**
  - **Algorithms**
  - **Architecture to Detailed Design Tracing**
- Traceability
  - Architecture components (and/or requirements) to detailed design components
  - Detailed design components to code
  - Requirements to detailed design components

DESIGN DOCUMENTATION QUALITY CHARACTERISTICS

- Feasibility
- Adequacy
- Economy
- Changeability
- Well-Formedness
- Completeness
- Clarity
- Consistency

CRITICAL DESIGN REVIEWS OF THE DESIGN DOCUMENT

- A *critical* review is an evaluation of a *finished product* to determine if it is of acceptable quality
- Critical reviews can be:
  - Desk checks
  - Walkthroughs
  - Inspections
  - Audits
  - Active reviews



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

CRITICAL AND CONTINUOUS REVIEWS

- A *critical* review that finds serious design defects may result in a return to (and rework of) a much earlier stage of design
  - Expensive
  - Time consuming
  - Frustrating
- In addition to critical reviews, we need to perform *continuous review* during the design process to *find faults early*

OUTLINE

- Software detailed design
  - Process
  - Models and their representation
  - UML notations used to represent detailed design
  - Construction techniques and principles
  - Design Patterns
  - Operations and data
  - Software design document
- Assignments

We are here

ASSIGNMENTS

- Quizzes
  - Detailed Design Quiz (11/15)
- Discussion Questions
  - Week 11 – Questions (11/15)
- Exams
  - Final Exam

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND