SOFTWARE DESIGN AND PATTERNS

CS 5010 FALL 2022

INSTRUCTORS: BRIAN CROSS AND TAMARA BONACI

Credits: portions of these materials adapted from Maria Zontak's CS 5010 course for Northeastern University

ADMINISTRIVIA

SOFTWARE DESIGN AND PATTERNS

CS 5010, FALL 2022 – LECTURE 4

DESIGNING SOFTWARE

- DESIGN IS *CREATIVE* PROBLEM SOLVING
  - No surefire recipe for success
  - Learn from others' experience
  - Following known best practices is a significant determinant of success

DESIGN PROBLEM FORMULATION

How to decompose a system into parts, each with a lower complexity of the whole system, such that:

- Interaction between parts is minimized
- Combination of the parts together solves the problem

No universal way to do this.

Some rules of thumb:

- Don't think of the system in terms of components that correspond to steps in processing: this adds complexity.
- Do provide a set of modules that are useful for writing many programs: plan for reuse!

WHAT GOES WRONG WITH SOFTWARE?

But then something begins to happen. The software starts to rot. At first it isn't so bad. An ugly wart here, a clumsy hack there, but the beauty of the design still shows through. Yet, over time as the rotting continues, the ugly festering sores and boils accumulate until they dominate the design of the application. The program becomes a festering mass of code that the developers find increasingly hard to maintain.

Robert C. Martin, design principles and design patterns

ROTTING DESIGN

- Symptoms
  - Rigidity: hard to change
  - Fragility: breaks easily
  - Immobility: hard to reuse code
  - Viscosity: structure breaks down
- Causes
  - Changing requirements
  - Dependency management
  - Not having a design phase

SOLID PRINCIPLES

Acronym for OOD design principles

- **S**ingle responsibility principle
- **O**pen/closed principle
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

Based on Robert Martin's "*design principles and design patterns*", acronym attributed to Michael Feathres

Single Responsibility Principle

- Do one thing and do it well
- Classes and objects should have clear purpose
- Keep methods short and to the point
- Utility classes can be used to carry out tangential jobs (logging, reporting)

Single Responsibility Principle

- Separate business logic from presentation
- User interaction should be done through controller/view modules
- Do not put print statements in your core classes
- Instead, return data that can be handled by view, logging, etc. modules

No:

```
public void printMyself()
```

Yes:

```
public string toString()
```

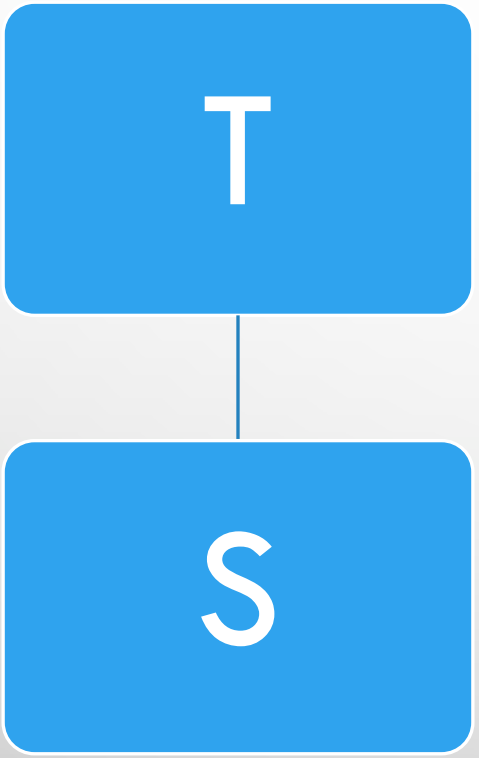
Open/Closed Principle

- Software should be "open for extension, closed for modification"
- Code should be written with the intention not to modify
- Code should be written to be easy to extend
- Extending existing functionality is the preferred way to add new functionality

Liskov Substitution Principle

- If S is a subtype of T, then objects of type T may be replaced by objects of type S in a program
  - Without negative side effects

EXAMPLE: **L**ISKOV SUBSTITUTION PRINCIPLE

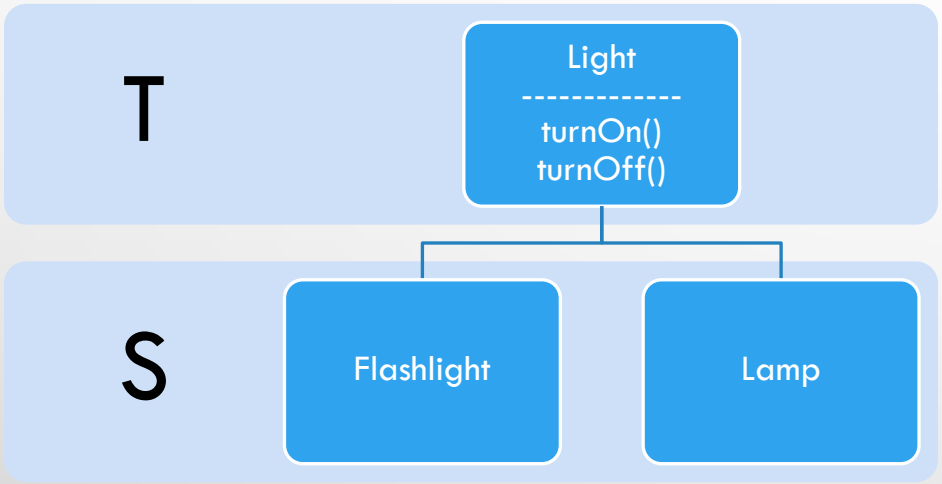


Liskov substitution principle

- If S is a subtype of T, then objects of type T may be replaced by objects of type S in a program
  - Without negative side effects.

From Barbara Liskov and Jeanette Wing's *"A Behavioral Notion of Subtyping"*

EXAMPLE: LISKOV SUBSTITUTION PRINCIPLE

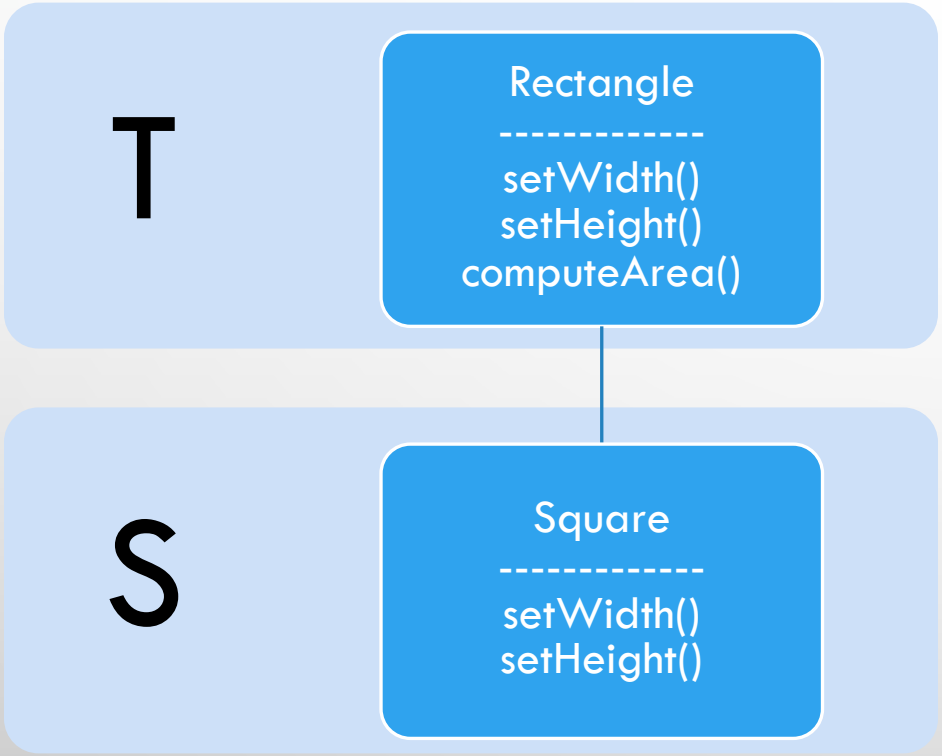


Liskov substitution principle

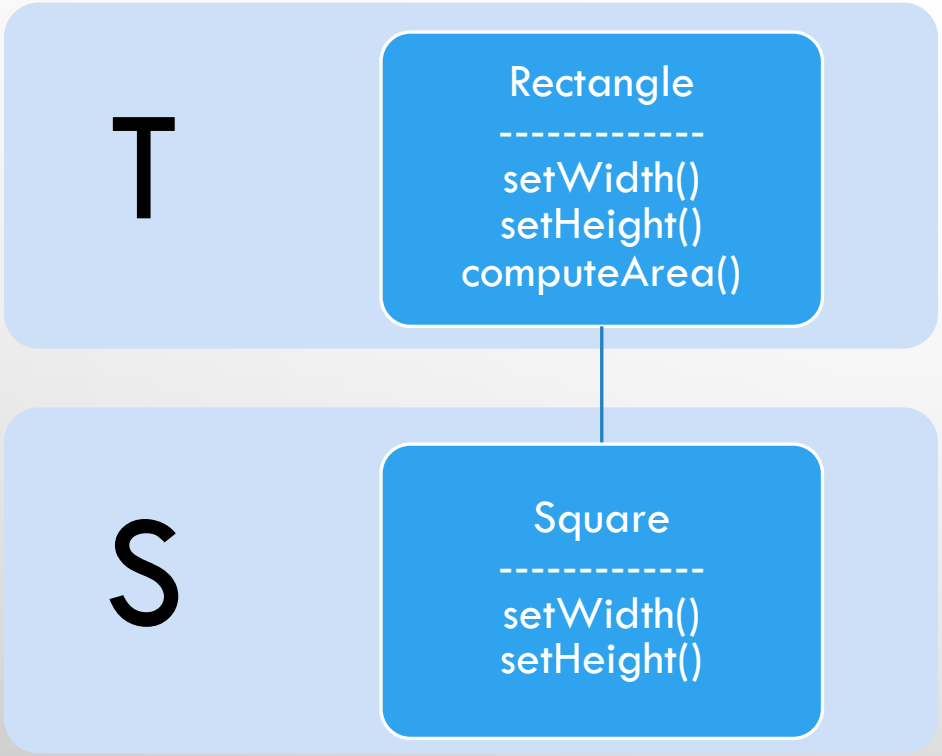
- If S is a subtype of T, then objects of type T may be replaced by objects of type S in a program
  - Without negative side effects.

From Barbara Liskov and Jeanette Wing's *"A Behavioral Notion of Subtyping"*

EXAMPLE: LISKOV SUBSTITUTION PRINCIPLE



EXAMPLE: LISKOV SUBSTITUTION PRINCIPLE



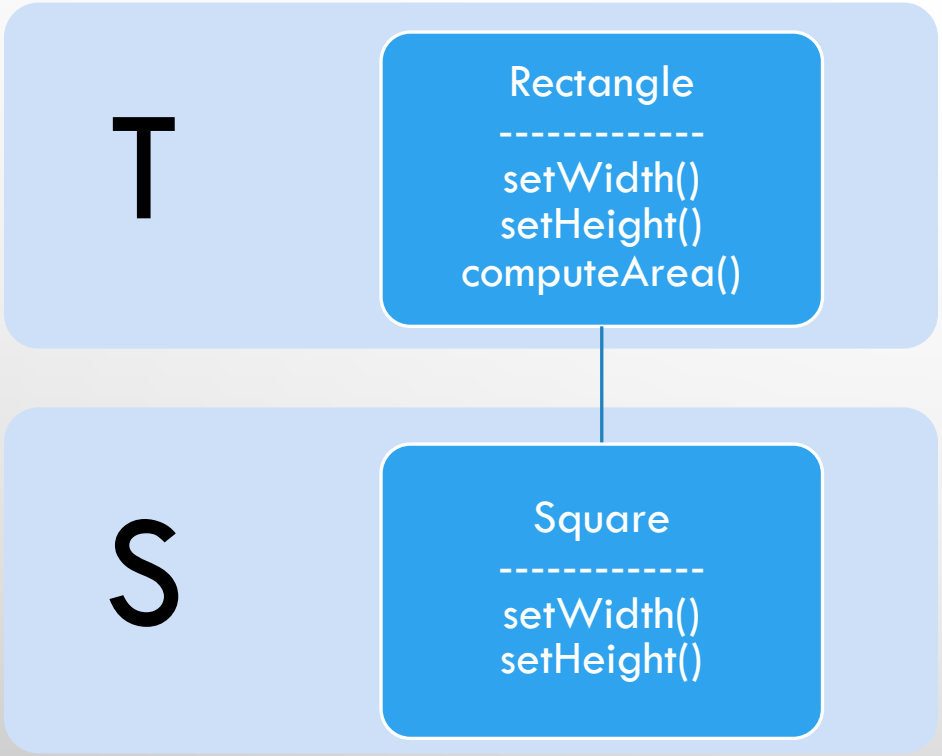
- SQUARE EXTENDS RECTANGLE
  - OVERRIDES SETWIDTH & SETHEIGHT

```
WIDTH = HEIGHT = NEWVALUE
```

```
RECTANGLE R = NEW RECTANGLE()

R.SETWIDTH(5);

R.SETHEIGHT(6);
```

COMPUTEAREA() == 30

EXAMPLE: LISKOV SUBSTITUTION PRINCIPLE



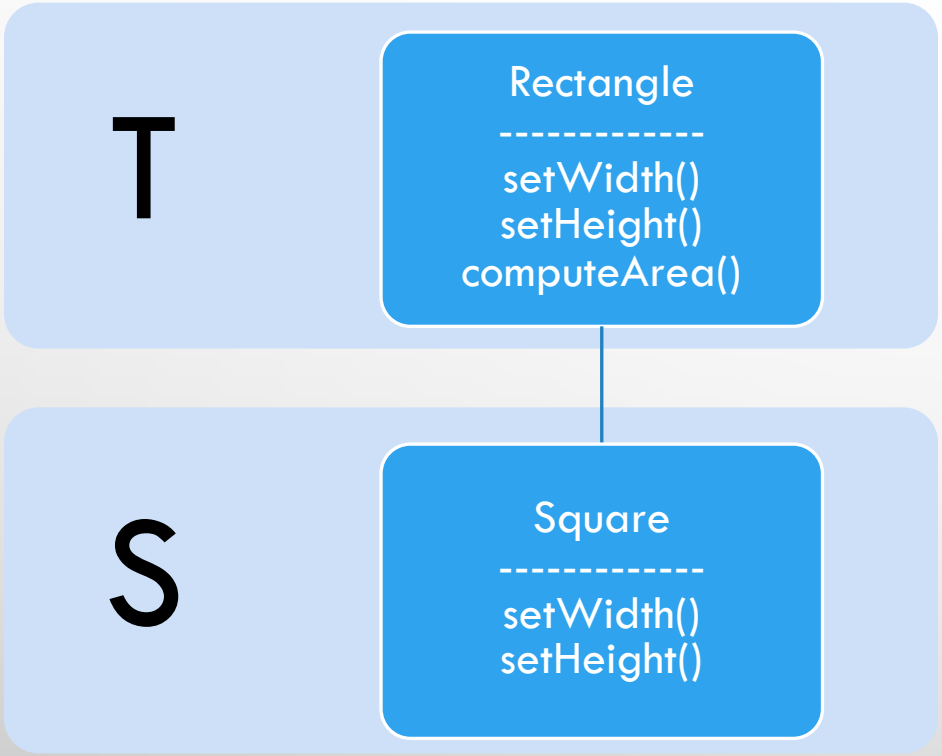
- SQUARE EXTENDS RECTANGLE
  - OVERRIDES SETWIDTH & SETHEIGHT

```
WIDTH = HEIGHT = NEWVALUE
```

```
RECTANGLE R = NEW SQUARE()

R.SETWIDTH(5);

R.SETHEIGHT(6);
```

COMPUTEAREA() == ??

EXAMPLE: LISKOV SUBSTITUTION PRINCIPLE



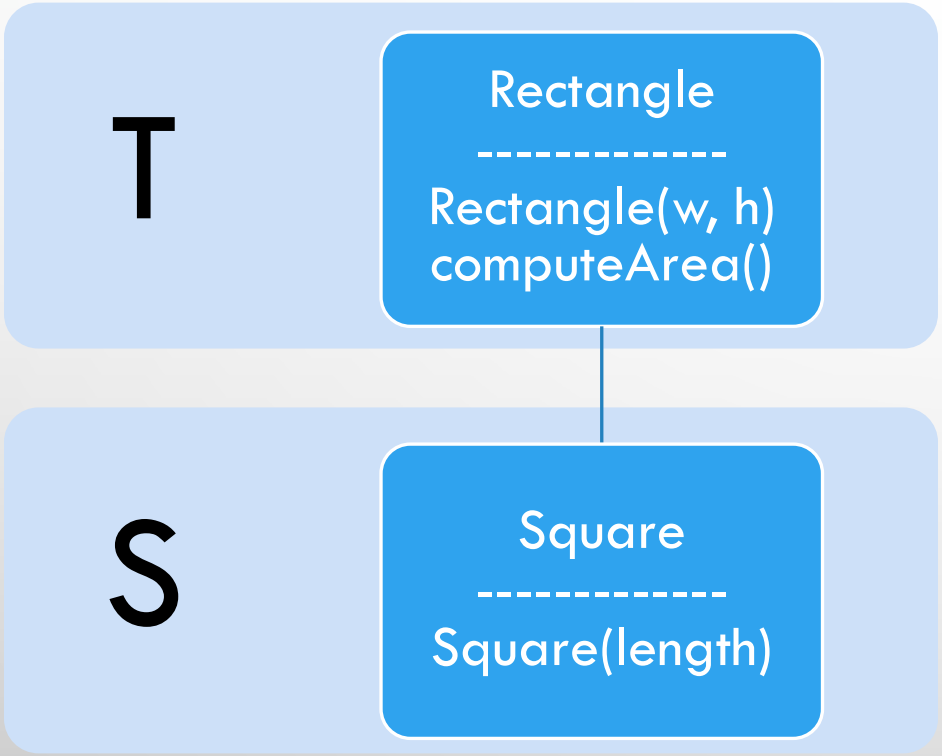
- SQUARE EXTENDS RECTANGLE
  - OVERRIDES SETWIDTH & SETHEIGHT

```
WIDTH = HEIGHT = NEWVALUE
```

```
RECTANGLE R = NEW SQUARE()

R.SETWIDTH(5);

R.SETHEIGHT(6);
```

COMPUTEAREA() == 36

EXAMPLE: LISKOV SUBSTITUTION PRINCIPLE



ONE WAY TO FIX?

MAKE THE CLASSES IMMUTABLE

```
RECTANGLE R = NEW RECTANGLE(5, 6)
```

COMPUTEAREA() == 30

```
RECTANGLE R = NEW SQUARE(6)
```

COMPUTEAREA() == 36

Liskov Substitution Principle

- If S is a subtype of T, then objects of type T may be replaced by objects of type S in a program
- A fundamental principle of working with adts, class hierarchies, and interfaces
- Strong built-in support in java, but we can still break it with ill-considered overrides or method side effects

From Barbara Liskov and Jeanette Wing's *"A Behavioral Notion of Subtyping"*

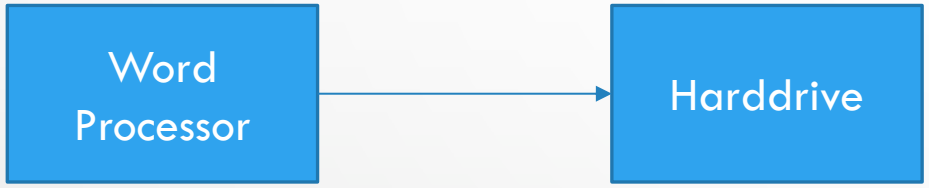
Interface Segregation Principle

- "No client should be forced to depend on methods it does not use." (Robert Martin)
- Keep interfaces minimal. Do not add extraneous methods declarations
- Prefer to implement multiple interfaces on classes, rather than to put too much functionality into one interface

Dependency Inversion Principle

- High level modules should not depend on lower-level modules. Both should depend on abstractions

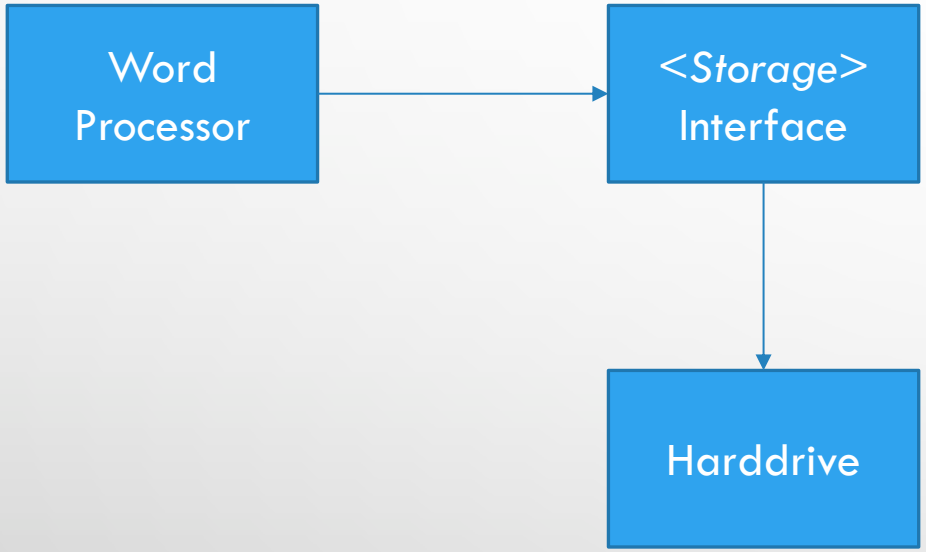
EXAMPLE: **D**EPENDENCY INVERSION PRINCIPLE



Dependency inversion principle

- High level modules should not depend on lower-level modules. Both should depend on abstractions.

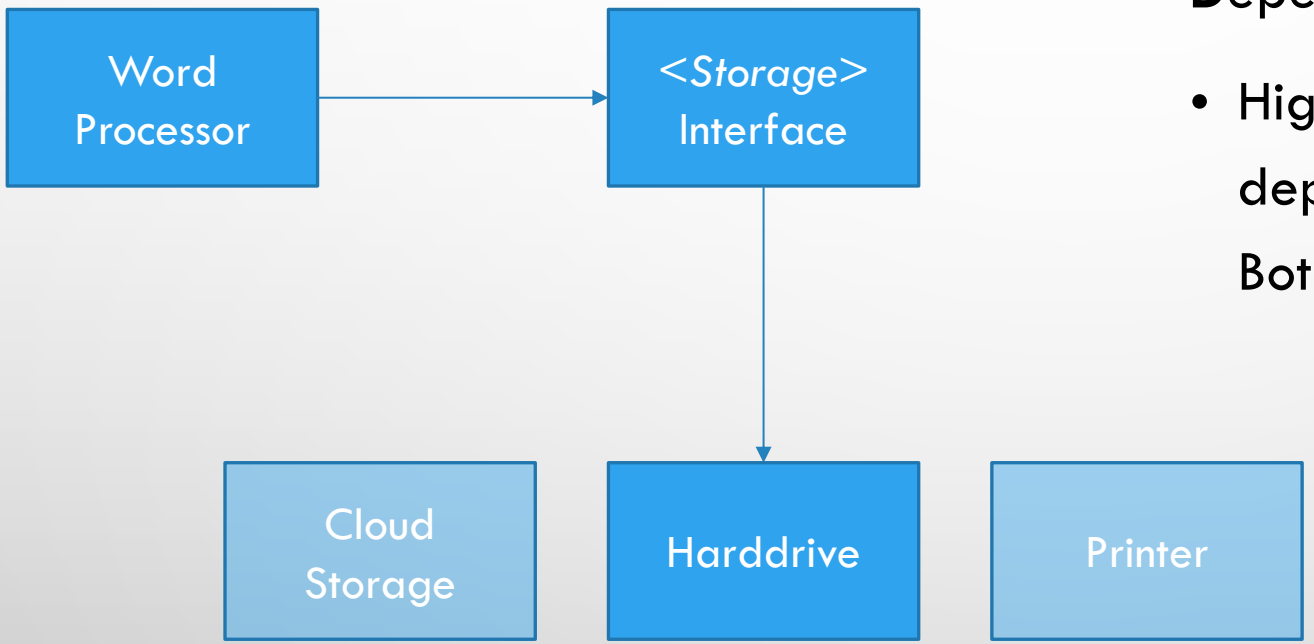
EXAMPLE: **D**EPENDENCY INVERSION PRINCIPLE



Dependency inversion principle

- High level modules should not depend on lower-level modules. Both should depend on abstractions.

EXAMPLE: **D**EPENDENCY INVERSION PRINCIPLE



Dependency inversion principle

- High level modules should not depend on lower-level modules. Both should depend on abstractions.

Dependency Inversion Principle

- High level modules should not depend on lower-level modules. Both should depend on abstractions
- Abstractions should not depend on implementations. Implementation should depend on abstractions
- "High level" means classes with broader duties that oversee or manage other behaviors. "Low level" means classes that carry out specific functionality
- Design from abstract to specific
- Use dependency injection (passing dependencies as parameters)

MODULES

Module: some part of a program responsible for a specific area of functionality and designed to interact with other modules.

Modular design regards modules as a design unit; focuses on how they are specified, what functionality they govern, and how they interact.

- Not focused on the implementation of modules
- Modules should respect other modules' abstraction barriers
- Each module should provide a single abstraction (ADT)
- Each module should do one thing well

IDEALS OF MODULAR SOFTWARE

- **Decomposition:** break a problem into modules to reduce complexity and enable teamwork
- **Composability:** put modules together, ideally in various ways
- **Understandability** should be understandable without reference to other modules
- **Continuity** small change in requirements should be answerable by small (local) change in design
- **Isolation** contain errors & enable independent development

MAXIMIZE COHESION & MINIMIZE COUPLING

- **Cohesion:**
  - How well a module encapsulates a single notion/responsibility
  - Degree to which the elements of a module belong together
- **Coupling:**
  - Degree to which a module interacts with or depends on other modules

Oo design seeks to minimize coupling (couple loosely) between modules and maximize cohesion within modules.

EXAMPLE: “TAKE A BREAK”

A "take A break" program: provide the diligent graduate student with an occasional reminder on their laptop screen to take a break from their study, stand up, walk around, and look out a window.

Possible elements for such an application:

- `Reminder:` A class responsible for displaying a message: "take a break!"
- `Timer:` a class responsible for calling that method from time to time.

“TAKE A BREAK” PROGRAM: NAÏVE DESIGN

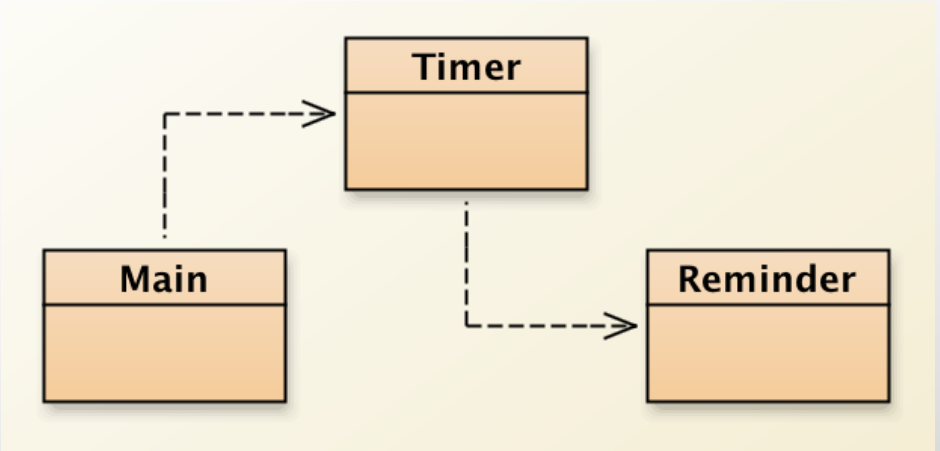
```
public class Reminder {
    public void display() {
        System.out.println("take a break!");
    }
}


public class <ain {
    public static void main(String[] args) {
        Timer timer = new Timer();
        Timer.Start();
    }
}
```

```
public class Timer {
    private Reminder reminder = new Reminder();
    public void start() {
        while (true) {
            ...
            if (enoughTimeHasPassed) {
                reminder.Display();
            }
            ...
        }
    }
}
```

“TAKE A BREAK” PROGRAM: NAÏVE DESIGN

Can we improve the design/reduce dependencies?

- Is `timer` reusable (for another application?)
- Can `timer` and `reminder` be decoupled?



DECOUPLING

Observation:

- `Timer` needs to call the `display` method, but it does NOT need to know what `display` does.
- To decouple `timer` and `reminder`, we can specify their relation via:

```
Interface timerTask {
    void run();
}
```

… And implement `timer` to work with any class that meets the `timertask` specification.

“TAKE A BREAK” PROGRAM: IMPROVED DESIGN

```
public class Reminder implements TimerTask {
    public void run() {
        display();
    }
    private void display() {
        System.out.println("take a break!");
    }
}
```

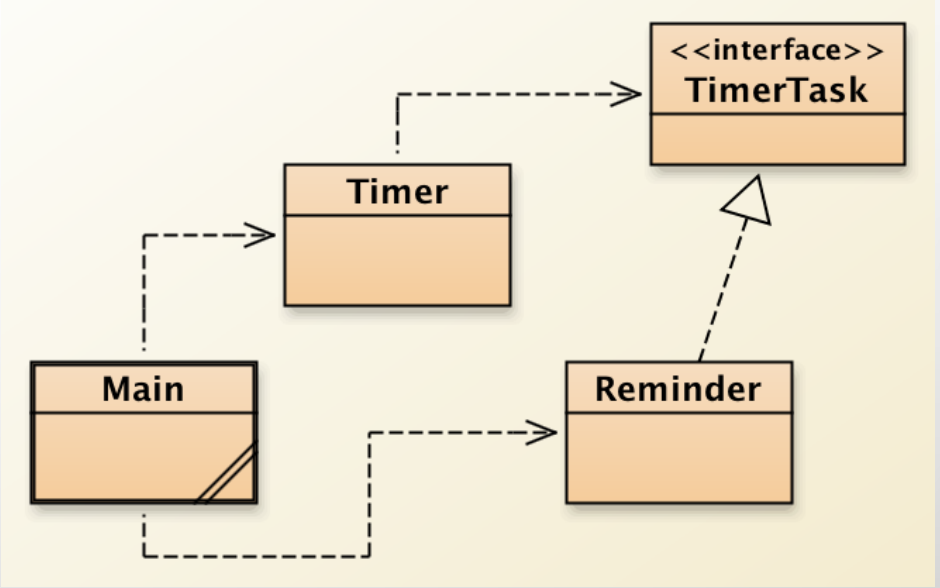
```
public class <ain {
    public static void main(string[] args) {
        tTmer timer = new Timer(new reminder());
        timer.start();
```

```
public class Timer {
    private TimerTask curtask;

    public tTmer(TimerTask newtask) {
        curtask = newtask;
    }

    public void start() {
        while (true) {
            ...
            if (enoughtimehaspassed) {
                curtask.Run();
            }
            ...
```

“TAKE A BREAK” PROGRAM: IMPROVED DESIGN

What we've achieved:

- `Timer` now depends on `timertask` rather than `reminder`
  - Reusable (other classes could implement `timertask`
  - Unaffected by implementational details of `reminder`
- `Main` depends on the constructor of `reminder` and still depends on `timer` (is this necessary?)



ALTERNATIVE: USING THE CALLBACK DESIGN PATTERN

- In real life, when you set an alarm in your phone/calendar, does the time depend on the reminder, or does the reminder depend on the time?
- We should invert the dependency, such that `reminder` depends on `timer`.
- Less obvious coding style, but more "natural" dependency.
- **Callback:** method call from a module to a client, to notify it about some condition.
- `Reminder` creates a `timer` and passes in a reference to *itself* so the `timer` can "call it back".
- Remove `main's` dependence on `timer`.

“TAKE A BREAK” PROGRAM: USING CALLBACK

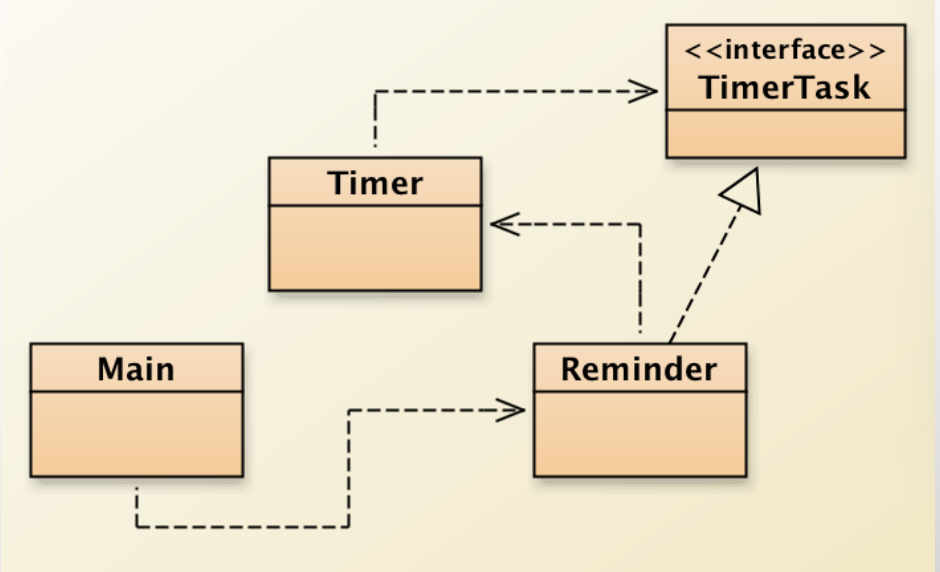
```
public class Reminder implements TimeTtask {
    Timer timer;
    public Reminder() {
        timer = new timer(this);
    }
    public void seton() {
        timer.start();
    }
    public void run() {
        display();
    }
    private void display() {
        System.out.println("take a break!");
    }
}
```

```
public class Timer {
    private TimerTask curtask;

    public Timer(TimerTask newtask) {
        curtask = newtask;
    }

    public void start() {
        while (true) {
            ...
            if (enoughtimehaspassed) {
                curtask.Run();
            }
            ...
        }
```

```
public class Main {
    public  static void main(string[] args) {
        reminder reminder = new reminder();
        Reminder.Seton();
    }
}
```

“TAKE A BREAK” PROGRAM: CALLBACK DESIGN

- MAIN DOES NOT DEPEND ON TIMER
- REMINDER DEPENDS ON TIMER



DECOUPLING AND DESIGN

- Examine dependencies before coding (design phase)
- Avoid unnecessary coupling
- Coding without first analyzing dependencies is likely to lead to unnecessary coupling
  - A method needs access to information from another object
  - Hasty, naive solutions lead to tight coupling
  - Code winds up harder to understand, less reusable

COUPLING: FROM WORST TO BEST

- **Content coupling (worst)**: one class depends on the internal data or behavior of another.
- **Common coupling**: classes share common data (global state). Includes static fields.
- **Control coupling**: method calls send information to control the method logic (e.G. In the form of flags). Caller must have knowledge of implementation.
- **Stamp coupling**: too much data is passed; e.G. A method that takes an object as an argument but only uses (needs) one attribute of that object.
- **Data coupling**: passing data as a parameter to a method call. Mostly unavoidable.
- **Message coupling**: calling a method that takes no parameters.

CONTENT (**WORST KIND**) COUPLING EXAMPLE

ARCH DEPENDS ON INTERNAL DATA FROM LINE (CONTENT COUPLING)

```
private class Line {
    private Point start, end;
    ...
    public Point getStart() { return start; }
    public Point getEnd() { return end; }
}

public class Arch {
    private Line baseline;
    ...
    void slant (int newy) {
        point theEnd = baseline.getEnd();
        theEnd.setLocation(theEnd.getx(), newy);
    }
}
```

CONTENT (**WORST KIND**) COUPLING EXAMPLE

Arch depends on internal data from line (content coupling)

Arch bypasses the interface of line and uses the interface of point. What if line changes the way it stores data, or adds additional code to handle point updates?

Arch would also need to be changed.

Boo!

```
private class Line {
    private Point start, end;
    ...
    public Point getStart() { return start; }
    public Point getEnd() { return end; }
}

public class Arch {
    private Line baseline;
    ...
    void slant (int newy) {
        Point theEnd = baseline.getEnd();
        theEnd.setLocation(theEnd.getx(), newy);
    }
}
```

ADVANTAGES OF REDUCED/LOOSE CONNECTIVITY (COUPLING)

- **Independent development:** design decisions made locally do not interfere with the correctness of other modules
- **Correctness:** tests/proofs easier to create
- **Increased reusability potential**
- **Easier maintenance**
  - Less likely that changes will impact other modules
  - More robust to errors
- **Comprehensibility:** the module can be understood without reference to/understanding of other modules

COHESION ANTI-PATTERN: GOD CLASSES

- *God class*: a class that hoards much of the data or functionality of a system.
  - Poor cohesion: little thought about why all the elements are placed together
  - Illusion of coupling reduction: minimal interdependence of modules because everything's crammed into one!
- Regarded as an anti-pattern (a known bad way of doing things)

MAXIMIZING COHESION

Methods should do one thing well:

- Compute a value, but let the client decide what to do with it
- Observe or mutate, but not both
- Do not print as a side effect of debugging and such
- Don't limit reusability of the method by having it perform multiple, not-necessarily-related things
- "Flag" variables are often a symptom of poor method cohesion

METHOD DESIGN

Effective java (EJ) tip #40: design method signatures carefully

- Avoid long parameter lists
- "If you have a [method] with ten parameters, you probably missed some" - alan perlis
- Especially error-prone if parameters are all the same type

EJ tip #41: use overloading judiciously

- Avoid overloading with same number of parameters
- Only overload if methods really are related

FIELD DESIGN

A variable should be made into a field if and only if:

- It is part of the inherent internal state of the object
- It has a value that retains meaning throughout the object's life
- Its state must persist past the end of any one public method

All other variables can and should be local to the methods in which they are used

- Fields should not be used to avoid parameter passing
- Not every constructor parameter needs to be a field

CONSTRUCTOR DESIGN

- Constructors should have all the arguments necessary to initialize the object's state - no more, no less
- Object should be completely initialized after constructor is done
- Client should NOT need other methods to "finish" initialization
- Should not do non initialization work
- Constructors CAN be private/protected

ENUMS AND ABSTRACTION

Consider the use of enums even with only two values.

Which is better?

```
Oven.Settemp(97, true);
```

```
Oven.Settemp(97, temperature.Celsius);
```

Consider creating a class for complex data or data that can change representation

CLASS DESIGN IDEALS

Beyond cohesion and coupling...

- **Completeness:** every (major) class should present a complete interface (self contained)
- **Consistency:** in names, param/returns, ordering, behavior and exception declaration
  - If you have multiple variables that represent a similar phenomenon (e.G. Time), use the same type.
  - If you declare runtime exceptions with throws in the signature, do it always.
  - Some built-in inconsistency: `string.Length(), array.Length, collection.Size()`

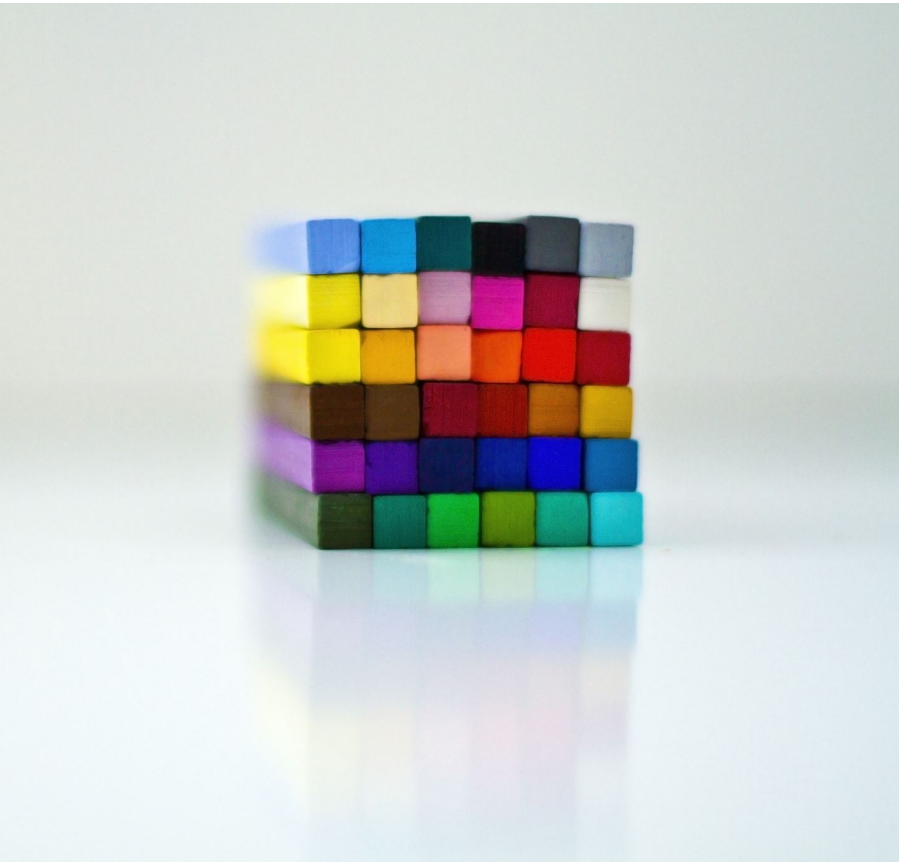
COMPLETENESS

Include *important* methods to make a class easy to use.

Counter examples:

- A mutable collection with `add` but no `remove`
- A tool object with a `seton` but no `setoff`
- A `date` class with no date arithmetic operations

ALSO…



- Objects that have a natural ordering should implement `comparable`
- Objects created by you should override `equals` (and therefore `hashCode`)
- Most objects should override `toString` (very useful debugging)

BUT…

Don't include *everything you can possibly think of*

- Once included, it's there forever (even if nobody uses it)
- More surface area for bugs
- More surface area for security threats

Don't overcomplicate

- You can always add it later if you really need it

Balance completeness ideal with YAGNI principle ("you aren't gonna need it")

DOCUMENTING A CLASS

- Docs should include all the method's preconditions and postconditions
- Preconditions typically described by the `@throws` tags for unchecked exceptions
- Preconditions can also be specified along with affected parameters in their `@param` tags
- Postconditions are best implemented via assert statements

ROLE OF DOCUMENTATION (FROM KERNIGHAN & PLAUGER)

- If a program is incorrect, it matters little what the docs say.
- If documentation does not agree with the code, it is not worth much.
- Code must largely document itself. If not, rewrite the code rather than increasing documentation of existing overcomplicated code.
- Good code needs fewer comments than bad code.
- Comments should provide additional information from the code itself. They should not echo the code.
- Variables should be meaningful and layout should show logical structure.

PATTERNS IN SOFTWARE DEVELOPMENT

- Established approaches to solving common problems
- Enable us to draw on others' experience
- Enable a common vocabulary to discuss abstract problem-solving concepts
- Various degrees of granularity/specificity

WHAT IS A PATTERN?

- Patterns are for developers
- What patterns are most important depend on the problem domain, programming language, and paradigm
  - Encapsulation and inheritance may be regarded as patterns in procedural languages, while in OOP, they are built into the language
  - Monads, applicatives, and functors are patterns in functional programming
  - "Design patterns" terminology strongly associated with object-oriented programming

PATTERNS IN SOFTWARE DEVELOPMENT

- Architectural patterns
  - Broad scope/application-wide organization
  - May be concerned with hardware limitations, business risk, etc.
  - Can provide conceptual foundations of frameworks
- Design patterns
  - Narrower, more specific scope than architectural patterns
  - Established, reusable solutions to specific implementation problems
  - Language independent, at least within a fixed paradigm (strongly associated with OOP)
  - Principled approaches to addressing classes of problems

ARCHITECTURAL PATTERNS

Heavily problem-domain oriented

- Client/server pattern for networked applications
- Peer-to-peer pattern for networked applications
- Layered patterns (e.g., OSI layers, TCP/IP layers) for network communication
- Microservices/serverless pattern for cloud computing
- Blackboard pattern for integrating various information sources to determine complex, non-deterministic control (autonomous agents, speech recognition)

ARCHITECTURAL PATTERNS

- Patterns exist for all areas of software development. Concept of architectural patterns is very broad
- For user-facing applications, the **model-view-controller** pattern (and variants) is a frequently used architectural pattern
- Traditionally heavily used for designing desktop applications. More recently applied to web application design

MODEL-VIEW-CONTROLLER

- Originated in the smalltalk community in the 1970s
- Widely used in commercial programming
- Recommended practice for graphical applications (or some variation of MVC)
- Supported by GUI application/web development frameworks
  - Swing (java)
  - Angular (typescript/JS)

MODEL AND VIEW SEPARATION

- The *model* is the underlying business data of your software.
- The *view* is the ui, which displays data to the user and receives user input.
- The idea of mv* patterns is to keep these two areas of functionality as independent of each other as possible.

MODEL AND VIEW SEPARATION

Reasons for decoupling views and models

- Separation of design concerns
- Maintainable, extendable
- Simplifying clean/modular UI design
    - Enabling seamless UI swaps (e.g., From desktop to web, mobile)
- Facilitating UI testing
- Promotes division of labor (GUI design, business logic)

MVC OVERVIEW

Model:

- Contains the "truth" - data/object or state of the system

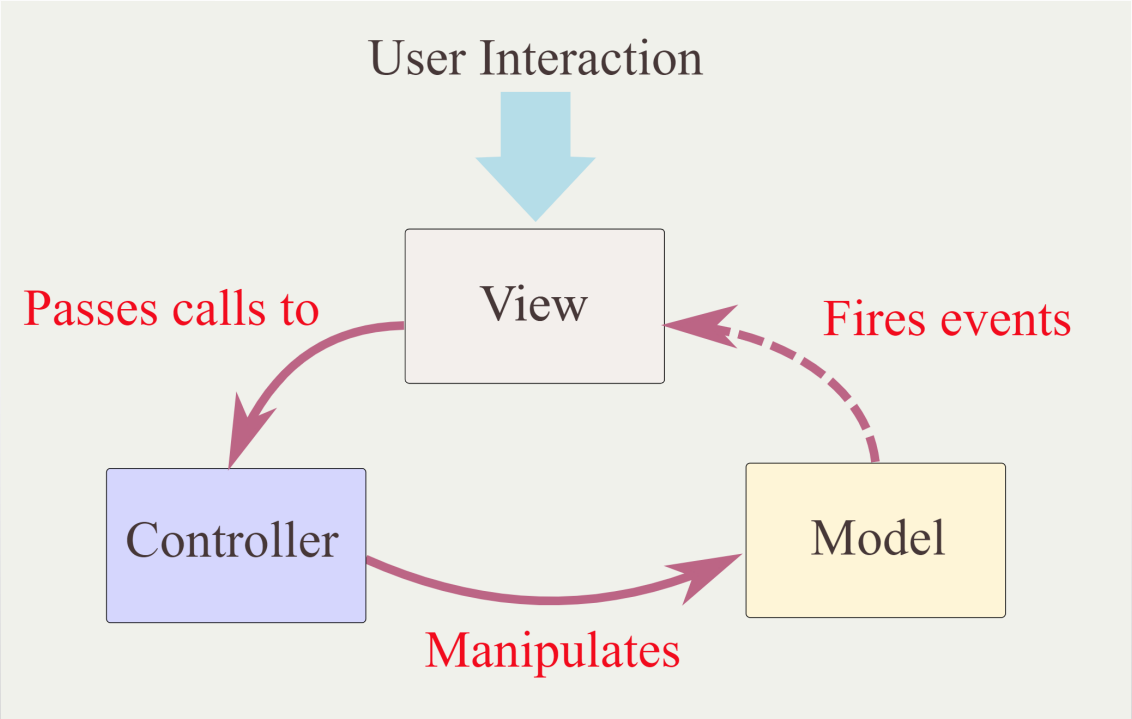
View:

- Presents information to user: GUI display, printed output, network stream, etc...
- May support multiple views of single model

Controller:

- Acts on both model and view
- Reacts to user input and other events
- Sends messages to model based on events
- Exposes application functionality

MVC DIAGRAM


User Interaction
View
Passes calls to
Fires events
Controller
Model
Manipulates

MVC INTERACTION AND ROLES

Model:

- Encapsulates data (system state) in some internal representation
- Maintains a list of interested viewers
- Notifies viewers if a change occurs and view update is needed
- Supplies data to views when requested
- Does not know details of display or interface

MVC INTERACTION AND ROLES

View:

- Maintains details about the display environment
- Gets data from the model when it needs to
- Renders data when requested (by the model or the controller)
- May catch user interface events and notify controller

MVC INTERACTION AND ROLES

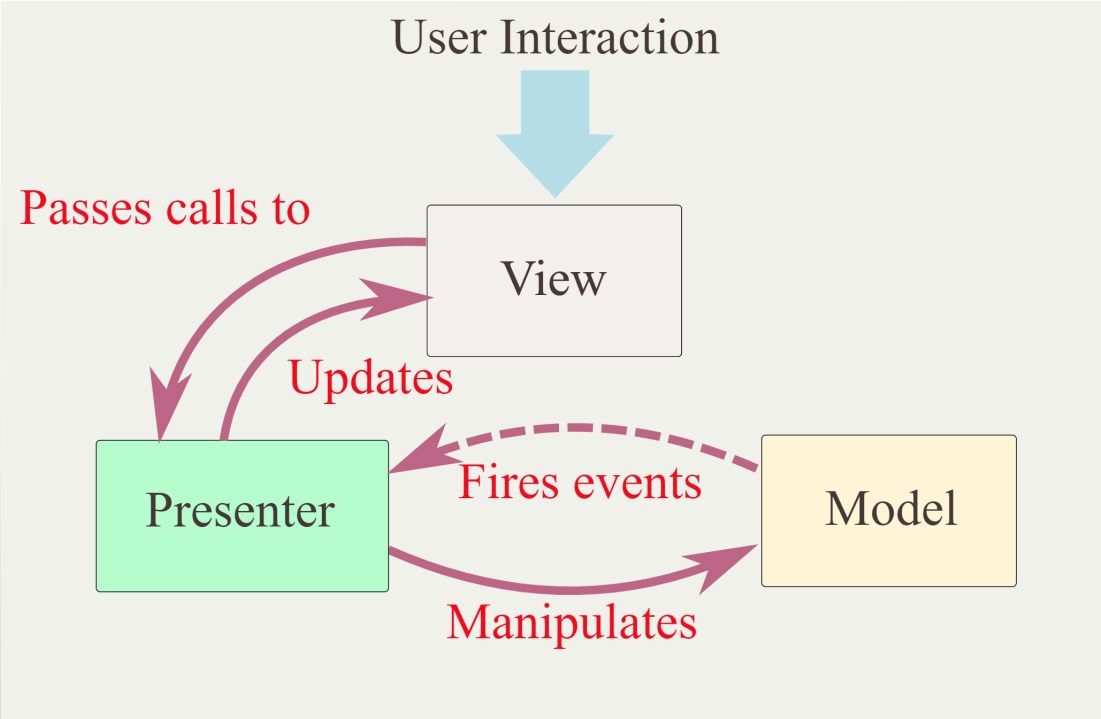
Controller:

- Intercepts and interprets user interface events
- Routes information to model and views

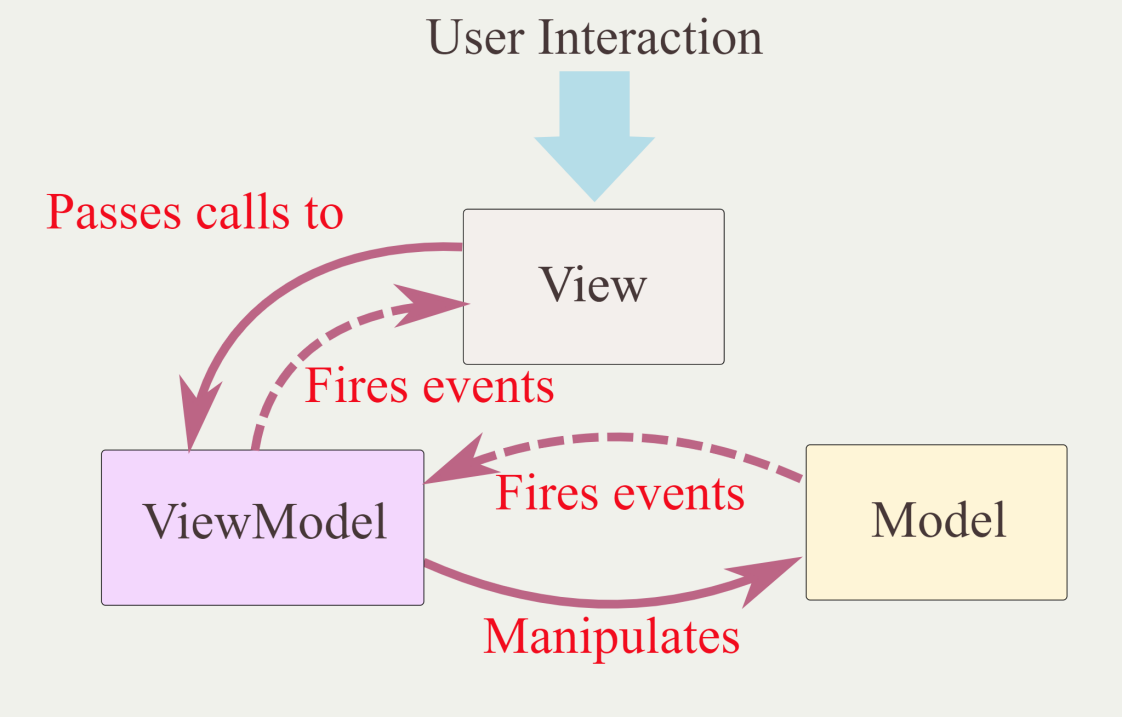
MVC AND MV*

- Separating model from view is basic good OO design
- Separating controller less clear cut
- Variations on the pattern can be organized differently

VARIATIONS: MVP (MODEL-VIEW-PRESENTER)



VARIATIONS: MVVM (MODEL-VIEW-VIEWMODEL)



IMPLEMENTATION NOTE

- Model, view, controller, etc, are *design concepts*, not class names
- Each might have more than one class implementing it
- Multiple views and controllers possible, but only one model
- Models and views may be reusable, but controller is typically not reusable

DESIGN PATTERNS

There are many commonly occurring problems in software development.

Design patterns can be thought of as templates for solving these.

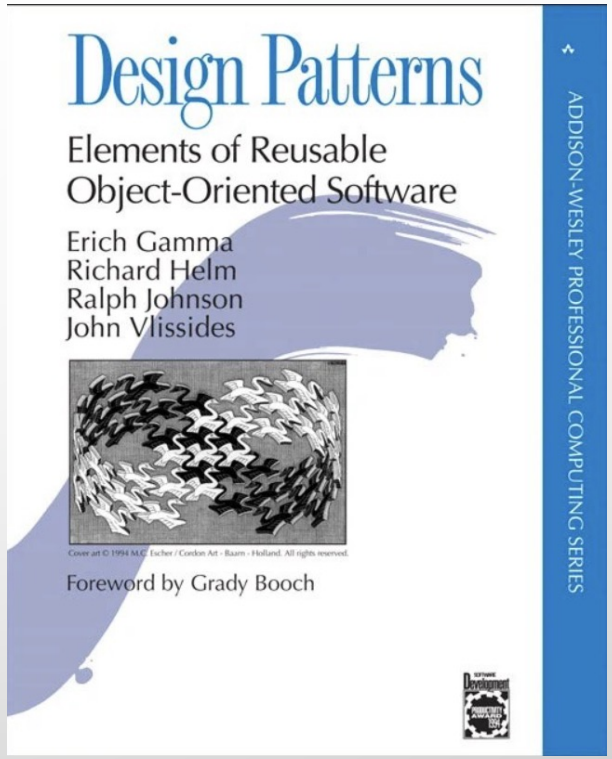
They are:

- Time tested and proven
- Easily re-used
- Expressive and understandable

THE “GANG OF FOUR”

Design patterns: Elements of Reusable Object-oriented Software (1995)
Erich Gamma, Richard Helm, Ralph Johnson and John Clissides (gof)

Highly influential publication in software development (particularly oop).



DESIGN PATTERNS

Other benefits:

- They can reduce the amount of code by reducing repetition
- They encapsulate the wisdom and experience of past developers
- They provide a vocabulary for communicating about software problems and solutions
- They enable developers to focus less on general issues of how to structure the software and more on the quality of the implementation

TYPES OF DESIGN PATTERNS

- **Creational design patterns** focus on ways to create or control the creation of objects
- **Structural design patterns** focus on object composition, relations between objects (e.g., Inheritance), and relations between objects and the system as a whole
- **Behavioral patterns** focus on improving or streamlining communication between objects

CREATIONAL PATTERNS

- **Factory method pattern:** write a method to return new instances of a class (or subclass, based on parameters sent to the method) without client object needing to call constructor directly
- **Abstract factory pattern:** a class that provides a family of factories to handle the responsibility of creating instances of subclasses
- **Builder pattern:** a class that encapsulates a complex object creation process, e.g., Including optional parameters
- **Prototype pattern:** create objects by creating a prototypical instance and cloning new objects from that
- **Singleton pattern:** create a single instance of a class. Constructor is private, called by a static method. (Regarded by critics as an *anti-pattern*)

STRUCTURAL PATTERNS

- **Adapter:** (aka wrapper) A class to take objects of another class and enable them implement an interface that their original class does not implement
- **Bridge:** decouple an abstraction from its implementation so that the two can vary independently
- **Composite:** compose objects into tree structures to represent part-whole hierarchies. Enables clients to treat objects and compositions of objects uniformly
- **Decorator:** attach additional behavior or responsibilities to an object dynamically. An alternative to subclassing for extending functionality

STRUCTURAL PATTERNS (CONTINUED)

- **Facade:** provide a unified interface to a set of interfaces in a subsystem. Simplify use of a complex subsystem by wrapping it in a simpler, higher-level interface
- **Flyweight:** maintain state that is shared by multiple objects in its own container, rather than duplicating state across objects
- **Proxy:** create an object to hold the place of another object, able to carry out certain functionality on behalf of the original object, possibly in situations where the original object would not be available to do so

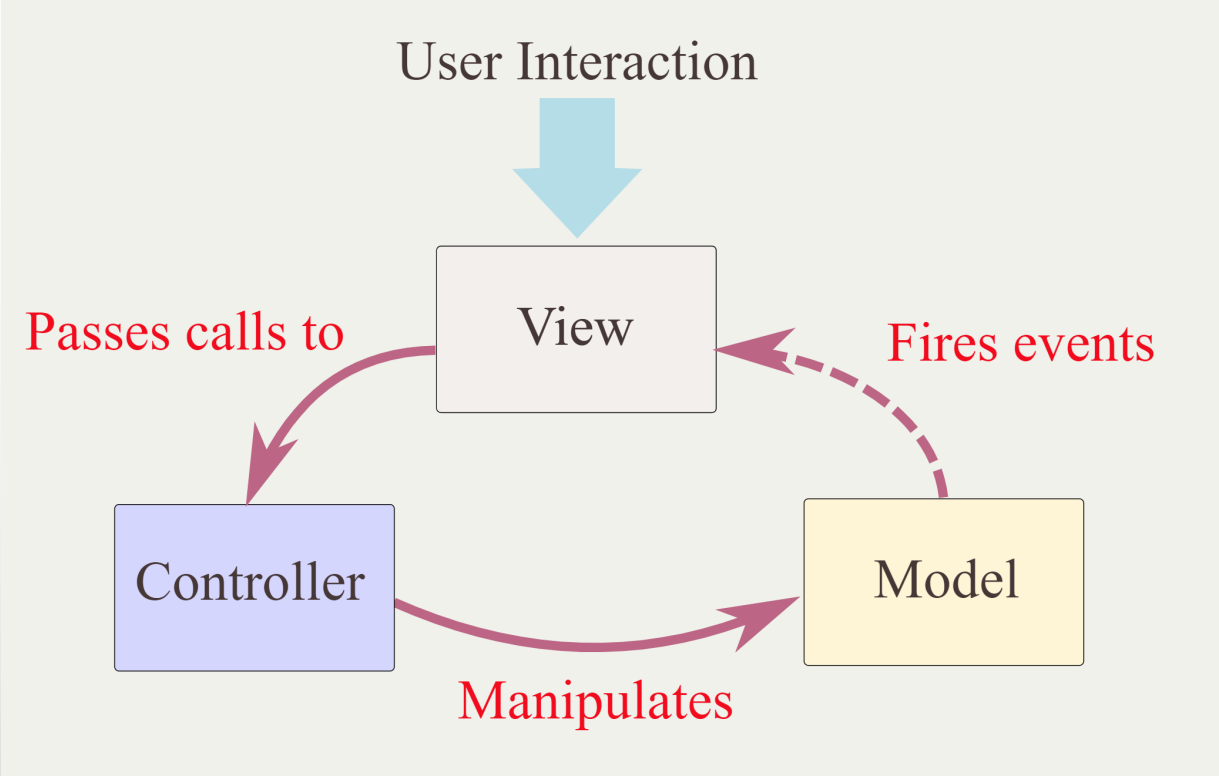
BEHAVIORAL PATTERNS

- **Chain of responsibility:** build a sequence of handlers where each handler processes a request and decides whether to pass it on to the next handler
- **Command:** abstract multiple related processes (functions or methods) into a single process, distinguishing the original cases by parameters
- **Iterator:** separates the structure of a collection from the process of iterating over it by creating class that encapsulates a particular approach to traversing the collection. For example, a tree-structure collection may be associated with a depth-first iterator and a breadth-first iterator, and the appropriate one would be used to iterate through items in the tree structure
- **Mediator:** a class that mediates communication between other classes. Simplifies potentially complex entanglements by routing all communications through the mediator

BEHAVIORAL PATTERNS (CONTINUED)

- **Memento:** create a representation of internal state (which can include private state) that can be used to restore the state of the original object
- **Observer:** (aka publisher/subscriber) enable multiple objects to respond to events or state changes on another object
- **State:** enable an object to change its behavior based on the state it's in
- **Strategy:** create classes representing separate algorithms for accomplishing a similar task (e.g., A route planning tool that works for bike, automobile, public transportation, or walk route planning)
- **Template method:** design the skeleton of an algorithm in a superclass, but let subclasses determine the specifics of the implementation
- **Visitor:** implement functionality in separate classes that can "visit" objects of dissimilar classes and lend them consistent functionality

MVC REVISITED



This architectural pattern is built from known components

MVC BUILDING BLOCKS

Established design patterns can be brought to bear to implement an MVC architecture

- Event firing from model to view implemented with *observer* design pattern
- Nested views (e.g., Button views in a control panel view) can be implemented according to the *composite* design pattern
- View delegates responsibility for the behavior on input events to the controller, which can be changed dynamically (for example, disabling a view by giving it a controller that ignores input events). Delegation of behavior can be achieved with *strategy* design pattern.

CREATIONAL DESIGN PATTERN EXAMPLE

Consider paying employees based on some attribute of the employee's data record

```
public Money calculatePay(Employee e) {
    switch (e.type()) {
        case COMMISSIONED: return caclculateCommissionedPay(e);
        case HOURLY: return caclculateHourlyPay(e);
        case SALARIED: return caclculateSalariedPay(e);
        default: throw new InvalidEmployeeTypeException(e.type);
    }
}
```

CALCULATE PAY

- Switch statement is large and will get larger
- Will need to be modified whenever new types are added (violates open/closed principle)
- Will likely need to be repeated in other employee methods, for example `isPayday` and `deliverPay`

ABSTRACT FACTORY PATTERN [GOF]

- Factory class will use the switch statement to create appropriate instances of derivatives of `employee`
- Various methods such as `calculatePay, deliverPay,` and `isPayday` will be dispatched polymorphically through the `Employee` interface.
- `Switch` can be tolerated here as it:
  - Only appears once
  - Is used to create polymporphic objects
  - Is hidden behind an inheritance relationship so that the rest of the system does not see it.

ABSTRACT FACTORY PATTERN [GOF]

```
//Employee and factory - clean code listing 3-5
public interface Employee {
    boolean isPayday();
    Money calculatePay();
    void deliverPay(Money pay);
}

public interface employeeFactory {
    Employee makeEmployee(EmployeeRecord r)
}
```

```
public class ConcreteEmployeeFactory implements EmployeeFacto
ry {
    Employee makeEmployee(EmployeeRecord r) {
        switch(r.getType()) {
           case COMMISSIONED: return new CommissionedEmployee
(r);
           Case HOURLY: return new HourlyEmployee(r);
           Case SALARIED: return new SalariedEmployee(r);
           Default: throw new InvalidEmployeeTypeException(r.
getType()
        }
    }
}
```

QUESTIONS?