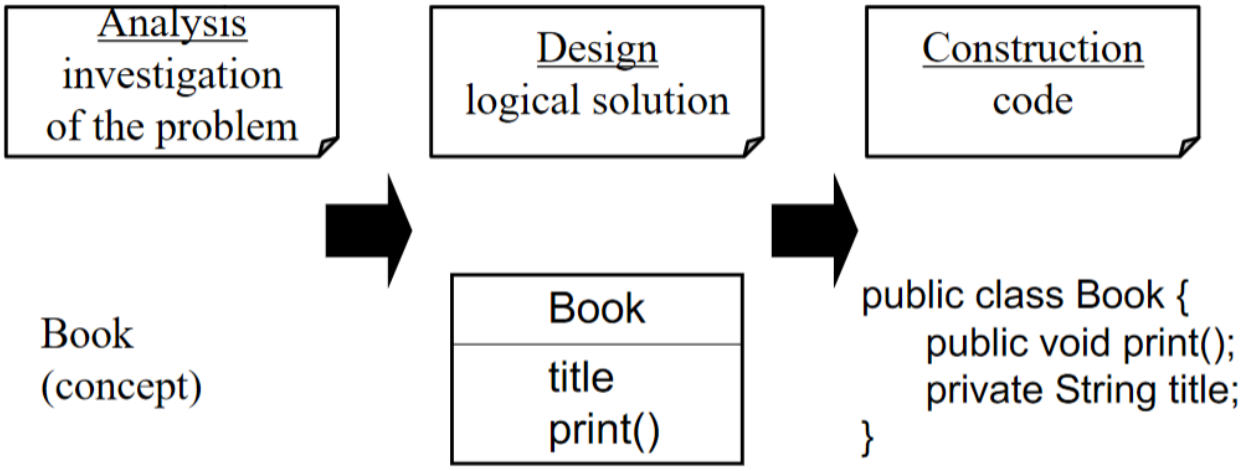
CS 2340

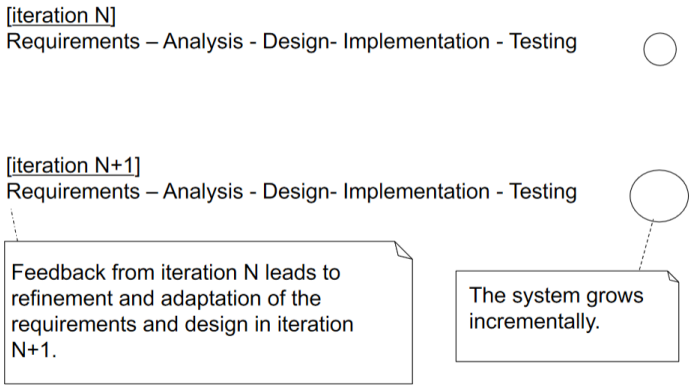
L02 - Intro to Iterative Development & the Unified Process (UP)

- The object-oriented approach emphasises finding and describing objects/concepts in the real world. This is the "Investigation" section, where we conceptualize what is needed.
    - For instance, concepts in an ATM System would include things like Banks and Transactions
- Object-oriented design allows us to define software objects and collaborate them to fulfil requirements. This is the "Blueprint" section, where we abstract what is needed.
    - For instance, in the ATM system, a Transaction software object may need a **`transactionID`** attribute and a **`getTimestamp`** method
- These designs get implemented into programming languages. This is the "Construction" section, where we physically write out what is needed
    - The **`Transaction`** class can be written in Java

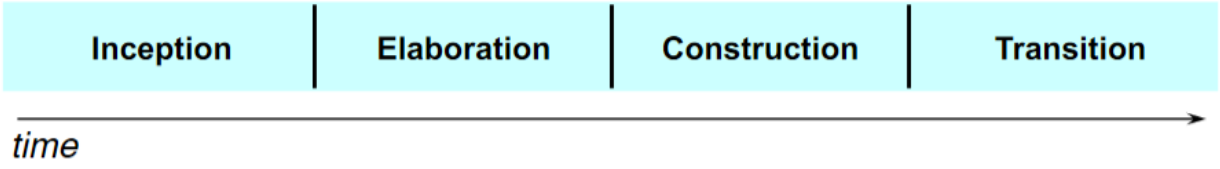


- In Software Engineering, the goal is to efficiently and predictably deliver a software product that meets the needs of a business
- A **software development process** is an approach to building, deploying, and maintaining software
    - The **Unified Process** (**UP**) is a *software development process* unique to object-oriented systems.
- While simple systems might take a linear approach to development, where definition, design, and implementation are all done in one-go, UP promotes **iterative development**, whereby the development of a system stretches out over a series of cycles
    - Development for this method is broken down into short fixed-length mini-projects called **iterations**
    - These iterations define a complete development cycle including analysis, design, implementation, and testing
    - With each successful iteration, the system grows incrementally over time, with new iterations tackling new requirements

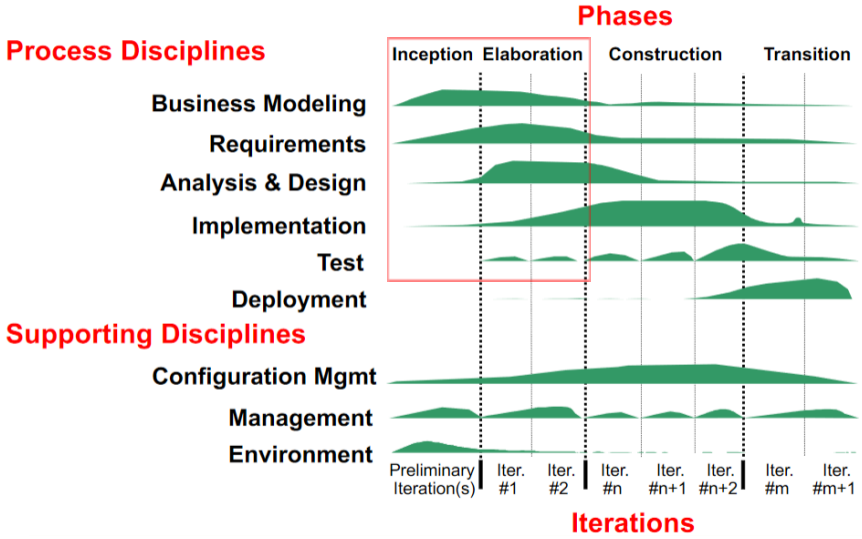
- The output of an iteration is not an experimental prototype, but rather a production subset of the final system
- The requirements of a system may change over time; each iteration involves quickly completing a small subset of the requirements. This allows for rapid feedback from the people paying for your product.



- Therefore, to summarize the Unified Process (UP) software development process…
  - Utilizes cyclic *iterations* instead of a one-and-done linear process
  - These iterations span the entire linear development process (investigate, design, implement), are completed quickly for rapid feedback, expand upon the system, be of fixed length (**timeboxed**), and have some focus on risk mitigation.
    - UP recommends that each iteration should be between 2 - 6 weeks in duration



- Each phase ends with a well-defined milestone, where you assess how well key goals have been met and whether the project needs to be restructured at all



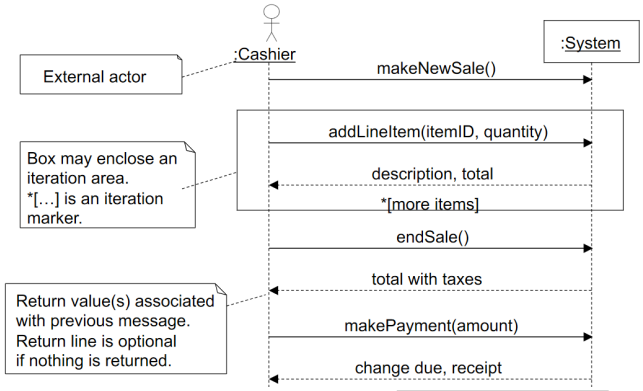
- This course will focus on the above sections in the red square

- Overtime, iterations have different emphasises on different disciplines of the development process. For instance, later iterations will focus more on the implementation and testing side than the earlier iterations.

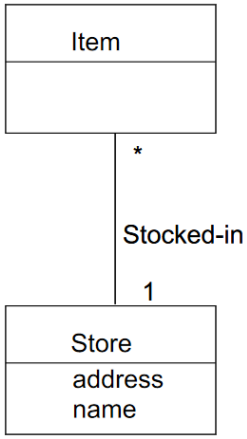
L03 - Unified Process Phases: Inception, Elaboration, Construction, and Transition

- Inception
    - Before you design the layout for a system, you need to ask a lot of different questions…
        - How much will this cost?
        - Is this feasible?
        - What do we envision the product to be for this project?
    - The goal is to narrow down the objectives of the project, determine if it's feasible, and decide if it is worth more elaboration
    - Once you've answered those, you may be equipped with some or all of the following…
        - *Vision* - describes the goals for the project
        - *Use Case Model* - describes the functional requirements and non-functional requirements
            - May be written in three formality types…
                - *Brief*: one-paragraph summary
                - *Casual*: informal paragraph format
                - *Fully-Dressed*: elaborate; every step and variations are written in full detail
        - *Supplementary Specification* - describes any other requirements
        - *Glossary* - describes the terminology
        - *Risk List and Risk Management Plan* - describes all the risks and ideas for their mitigation
    - Using the above, we can create prototypes and proof-of-concepts and plan out the iterations and their duration for the UP. These will be partially completed and refined in later iterations.
    - This phase should only last roughly 1 week
    - By the end of the inception phase…
        - Most actors, goals, and use cases will be named
        - Most use cases will be written in brief format, with 10-20% of them being in full detail to improve understanding of the scope and complexity
        - Most influential and risky quality requirements identified
        - Proof-of-concepts prototypes developed
        - there should be a plan for the first iteration
- Elaboration
    - During elaboration…
        - The main, risk software architecture is programmed and tested

        - The majority of requirements are discovered and stabilized
        - The major risks are mitigated or retired
    - This is where you'll have your first iteration
    - Ideally in elaboration…
        - You'll do short, timeboxed risk-driven iterations
        - Start programming and testing early, often, and realistically
        - Adapt based on test, user, and developer feedback
        - Write most of the use cases and other requirements in detail
    - This phase is when you'll handle most of the "risky" parts of the architecture
    - A **sequence diagram** is a picture that shows, for a particular scenario of a use case, the events that external actors generate, their order, and possible inter-system events
        - Big emphasis on the "black box," the boundary from actors to systems

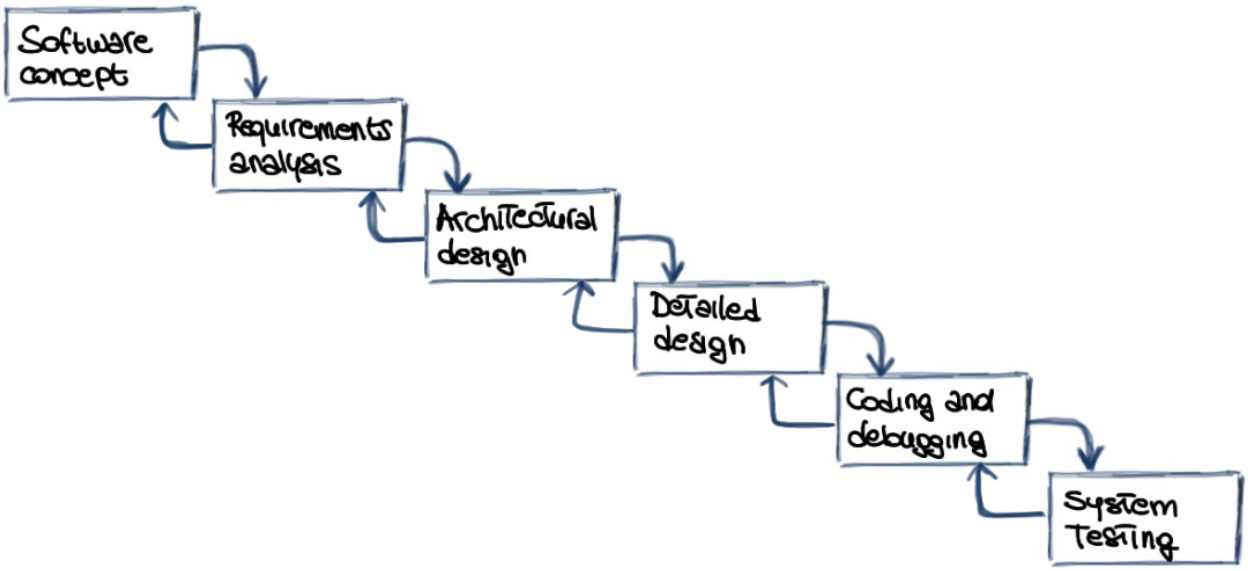


    - A **domain model** illustrates real-world objects and concepts interacting with each other in relation to the problem domain. It does not have any description of software design.



    - From iteration 1 to 2, the software should have been vigorously tested, used to obtain customer feedback, and the system as a whole should be integrated and stabilized
    - Iteration 2 will focus on object design.
    - Iteration 3 explores a variety of analysis and design
- Construction
    - By the end of elaboration, most requirements should be understood and the risk, architecturally significant core of the system should be build
    - Construction focuses on whatever is left
    - Testing during this phase is referred to as "alpha testing," and is performed at the developer's site
    - Ends when the system is ready for operational deployment and all supporting materials (user guides, training materials, etc.) are completed
- Transition
    - Meant to put the system into production use
    - Testing during this phase is referred to as "beta testing," and is performed at the client's site

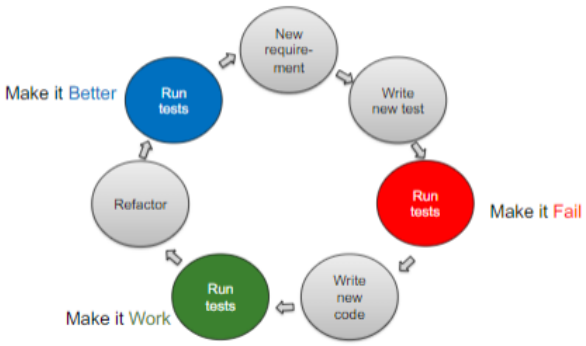
- Overall, the UP uses short, timeboxed iterations to allow for adaptive development. Earlier iterations tackle high-risk and high-value issues while easier work is left to later iterations (this ensures the project does not "fail late; better to "fail early" if at all)



L04

- Software is buggy! On average, there are 1 - 5 bugs per 1 thousand lines of code
- Three Types of Mess-Ups
    - **Failure**: observable incorrect behavior in a program. Conceptually related to the behavior of the program as opposed to its code
    - **Fault** (aka, a **bug**): Related to the code. Necessary (not sufficient) condition for the occurrence of a failure.
    - **Error**: Cause of a fault, usually a human error (conceptual, typo, etc.)
        - A **failure** in the code leads to the discoverance of a **fault**. If we ask *why* the fault occurred, and we're able to figure it out, we say that reason is an **error**.
- In order to reduce the amount of bugs in our software, we need a way to *verify* that it works as intended.
- Approaches to Verification
    - *Testing* - exercising software to try and generate failures through input data
        - You can't only use testing, because testing every single test case would take forever!
    - *Static Verification* - identify (specific) problems in a way such that we consider all possible inputs/executions
    - *Inspection*/review/walkthrough - systematic group review of program text to detect faults
        - ~90% of all bugs can be easily deciphered just by looking at the code.

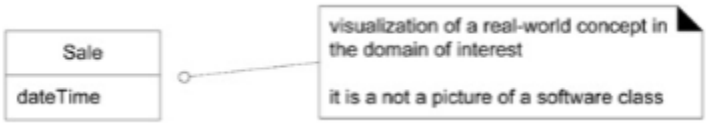
    - *Formal Proof* - proving that the program text implements the program specification
- Testing is considered to be a *dynamic* technique, because you physically have to type out your tests.
- Types of Testing
    - *Unit Testing* - Testing one component at a time
    - *Integration Testing* - Take component A, integrate it with component B, and test them together.
    - *Big Bang/System Testing* - Testing all components at once
    - *Acceptance Testing* - Testing all components at once with a customer controlling the test as opposed to an experienced programmer
    - *Regression Testing* - Testing new components with the old components, making sure that the new components don't cause old ones to break
- What is *test driven development*?
    - Tests are written *before* the class to be tested, and the dev writes unit testing code for nearly *all* production code
    - Write test code such that it fulfills the necessary requirements
    - Write functional code such that it fulfills the necessary requirements
    - Refactor/clean-up the working code such that it fulfills the necessary requirements
        - Make it fail, then make it work, then make it better



L05 - Understanding Requirements

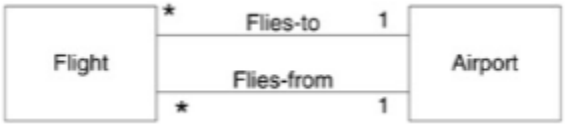
- **Requirements**: System capabilities and conditions to which the system must conform
- Two Types of Requirements
    - *Functional*: Things like "features" and "capabilities," recorded in the use case model
        - Describe what the system is supposed to do
    - *Non-Functional*: Things that are not required for the system to function
        - Usability (help, documentation, etc.)
        - Reliability (Frequency of failure, etc.)
        - Performance (response times)
        - Supportability (adaptability, maintainability, etc.)
- Requirements are derived from stakeholders (investors in a company), application domain (is it for a school? A bank?), and documentation
- Verification v. Validation
    - Validation checks that the *right product is being built*

        - Ensures that the developed software will satisfy stakeholders and the requirements derived from them
    - Verification checks that the *product is being built right*
        - Ensures that each step followed yields the right products
- Because we have limited time, and oftentimes some requirements are more important than others, we need to *prioritize* our requirements
    - Mandatory, nice to have, superfluous
- **Domain Model** - a visual representation of conceptual classes or real-situation objects in a domain
    - Shows domain objects or conceptual classes, associations between those classes, and their attributes.



    - Not allowed to show software artifacts like a window, a database, a scrollbar, responsibilities or methods
    - To create a domain model…
        - Find the conceptual classes
        - Draw them as classes in a UML class diagram
        - Add associations and attributes
    - Tips for finding conceptual classes…
        - Reuse or modify existing models
        - Use a category list
        - Identify noun phrases
            - Identify the nouns and noun phrases and consider them as candidate conceptual classes or attributes. Be careful not to use mechanical noun-to-class mapping and remember that words can be ambiguous
    - When everything works as expected, we call this the **main success scenario**.
    - If we do not think of some conceptual class X as a number or text in the real world, X is probably a conceptual class, *not* an attribute
    - Furthermore, you can also create a **description class** to contain information that describes something else
    - **Association**: a relationship between classes that indicate some meaningful connection
        - Each end of an association is called a *role*, which may optionally contain: a multiplicity expression, name, navigability
            - Name an association based on ClassName-VerbPhrase-ClassName format, ex. Sale Uses CashPayment

        - Two classes may have multiple associations between them



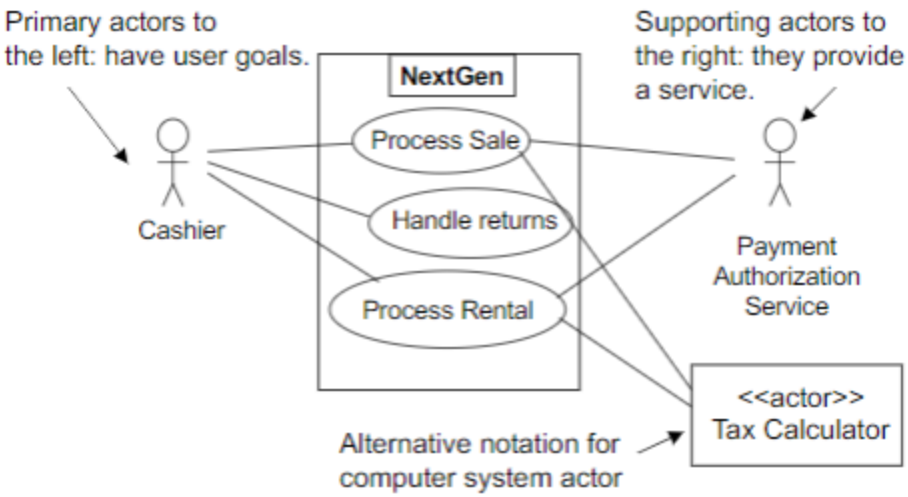
    - Arrows do not exist in domain models; we assume it goes from left-to-right and top-to-bottom
    - **Attribute**: a logical data value of an object
        - Include attributes that the requirements suggest or imply a need to remember information

L06 - Use Case Diagram

- **Use cases** are text stories used to discover and record requirements
    - May be written in three formality types…
        - *Brief*: one-paragraph summary
        - *Casual*: informal paragraph format
        - *Fully-Dressed*: elaborate; every step and variations are written in full detail

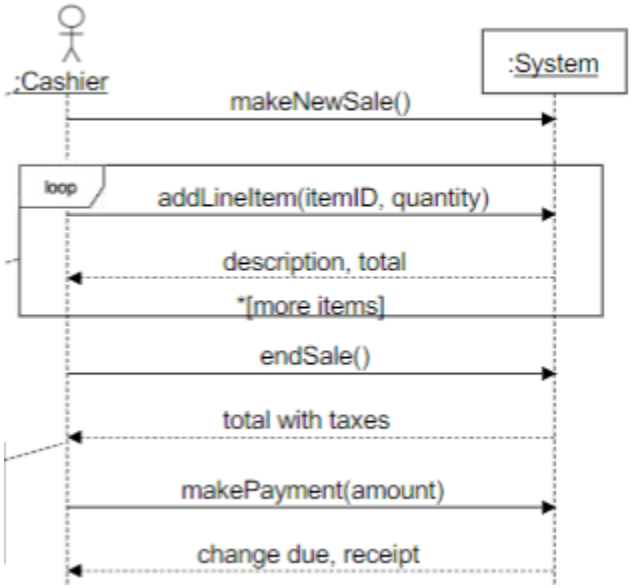
Use Case Section	Comment
Use Case Name	Start with a verb.
Scope	The system under design.
Level	"user-goal" or "subfunction"
Primary Actor	Calls on the system to deliver its services.
Stakeholders and Interests	Who cares about this use case, and what do they want?
Preconditions	What must be true on start, *and* worth telling the reader?
Success Guarantee	What must be true on successful completion, *and* worth telling the reader.
Main Success Scenario	A typical, unconditional happy path scenario of success.
Extensions	Alternate scenarios of success or failure.
Special Requirements	Related non-functional requirements.
Technology and Data Variations List	Varying I/O methods and data formats.
Frequency of Occurrence	Influences investigation, testing, and timing of implementation.
Miscellaneous	Such as open issues.

- **Actor**: something with behavior, e.g. people, computer systems, an organization
    - *Primary Actor* - has user goals fulfilled through using services
    - *Supporting Actor* - provides a service
    - *Offstage Actor* - has an interest in the behavior of the use case
- **Scenario**: Sequence of actions and interactions between actors and the system
    - Furthermore, let's assume the scenario of successfully purchasing items with cash
- **Elementary Business Process** (**EBP**) Test: a task performed by one person in one place at one time, in response to a business event, which adds measurable business value and leaves the data in a consistent state (?)
- <u>Styles of Use Cases</u>
    - *Essential* - Focus is on the intent, not the concrete details like the UI elements
    - *Concrete* - UI decisions are embedded in the use case text (e.g. "Admin enters ID and password in the dialog box (see picture X)")
        - Not suitable during early requirements analysis work



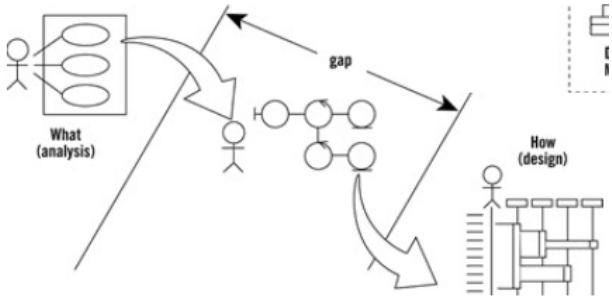
L07 - System Sequence Diagram

- A **system sequence diagram** is a picture that shows, for a specific use case, the events that external actors generate, their order, and possible inter-system events
- All systems are treated like a black box; places emphasis on events that cross the system boundary from actors to systems

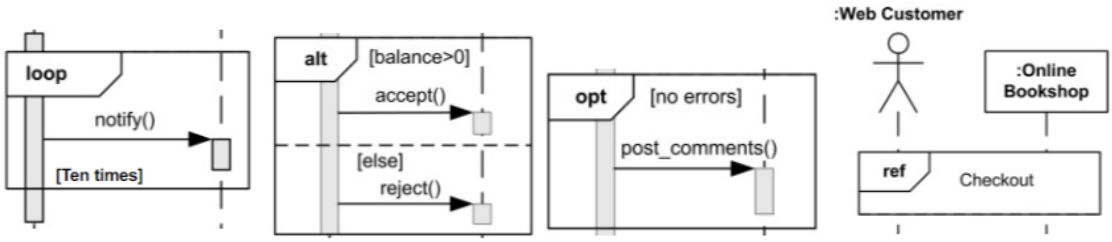


Example of an SSD

- Why is the customer not on this diagram? Because the customer does not interact with the system in any way; the cashier is the one working with the machinery/system
- Ordered vertically in time; starts at $t = 0$ at the top and ends at $t = x$ at the bottom.
- We initially start with the *use case model* during our analysis. We convert this over time into a *system sequence diagram*



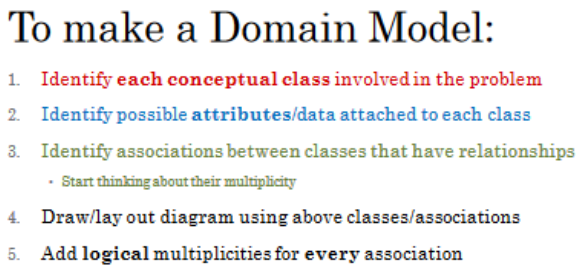
- An SSD can demonstrate complex interactions with sequence fragments such as…
    - *Loops* - iteration, can repeat a sequence of actions over a set or indeterminate amount of times
    - *Alt* - alternatives (if-else)
    - *Opt* - option
    - *Ref* - reference, can reference another system sequence diagram for conciseness and clarity



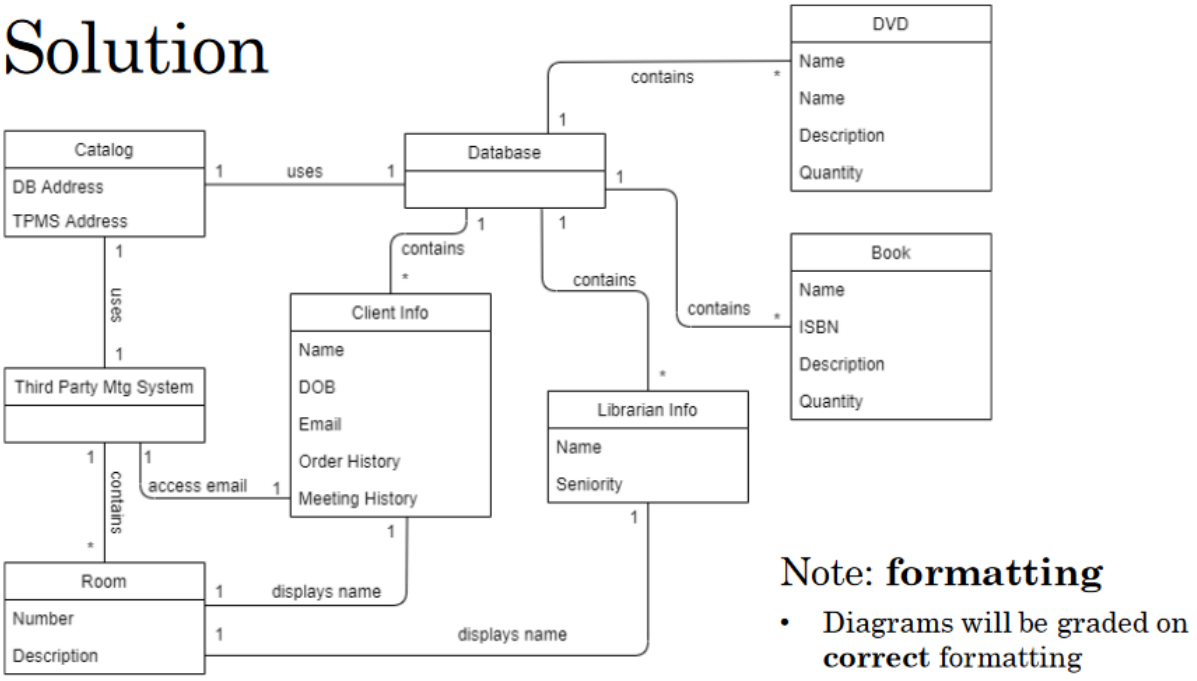
- Most SSDs are created during the elaboration processes; rarely are they every touched on in the inception phase
- Naming Convention: *ClassName*:*Object*
    - Examples…
    - NextGen:System
    - :Cashier (if we don't care about name)
    - Tom:Cashier

L08 - Exam #1 Info

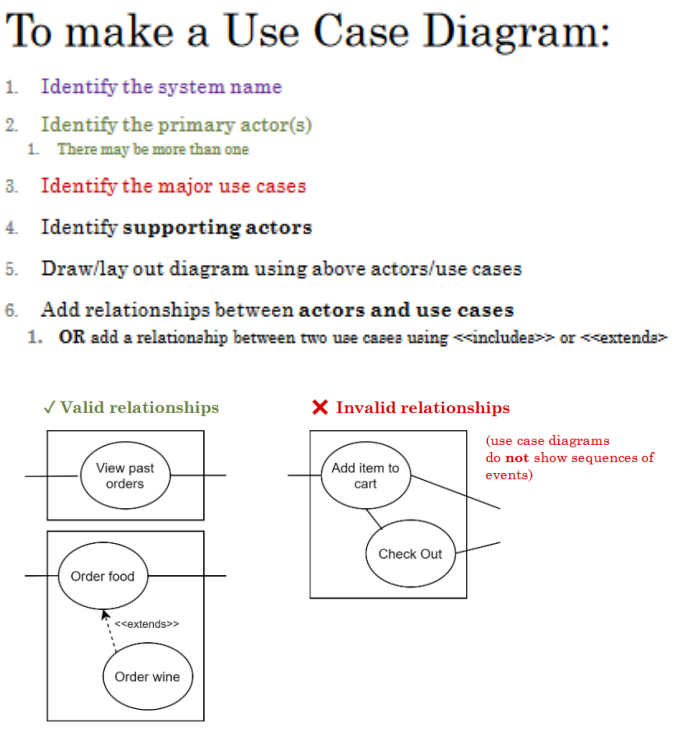
- Must be taken in person
- Format…
    - MC Part - 15 questions via Canvas Quiz. One question at a time, questions will be locked after answering
    - Diagram Part - Problem description provided via Canvas Quiz. Submitted to Gradescope
        - Recommended to use drawio to create diagram, then upload a .drawio/.PDF to Gradescope; but can use hand-written
- Piazza will be open for PM's to the instructor group only
- Test driven development will be on Exam 2, not Exam 1
- Three possible diagrams on the exam…
    - *Domain Model*
        - Includes all conceptual classes (likely **bolded**) and attributes attached to each class
        - Includes meaningful associations between classes that have relationships
        - Include logical multiplicities ("1 catalog uses 1 database")



- Example of a complete domain model…

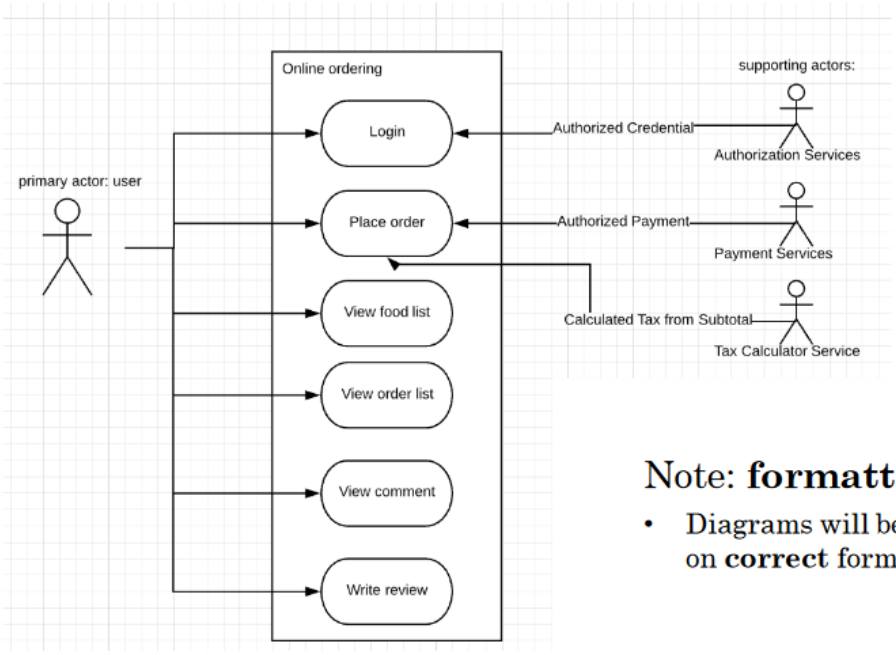


- *Use Case Diagram*
  - Used to identify goals of the primary actors
  - Utilizes the "black box"; doesn't worry about *how* the system will function, just understands *what* it will do
  - Includes primary (have goals) actors and supporting (provide services) actors
  - Use cases should always **start with a verb** (login, place, view, etc.)
  - Use cases should only be connected to each other *if they have* <<extends>>/<<includes>> in the relationship
  - Always draw the system boundary and include a system name
  - If you're unsure of whether to include a supporting actor, always just add it. It's easier to remove it later down the line



- Example of a completed use case diagram…

Solution



Note: **formatting**

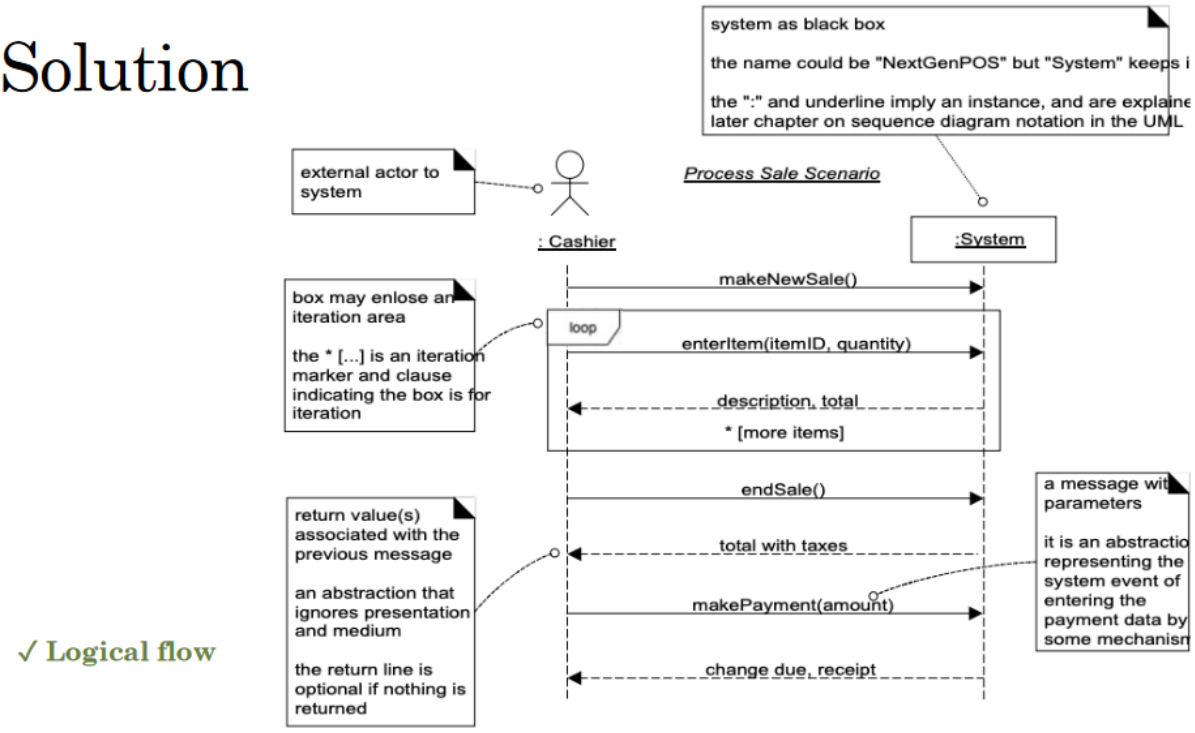
- Diagrams will be graded on **correct** formatting

- *System Sequence Diagram*
  - Illustrates interactions that cross the system boundary
  - Each use case should have 1 SSD and vice versa
  - Example of a completed SSD…

To make an SSD:

1. Identify the **primary actor** and **use case** (usually given)
2. Go through the sequence of events and note:
   1. Where the sequence **starts** (usually the primary actor)
   2. The method calls from Actor to System
   3. The return values from System to Actor
   4. Where tasks are **repeated until** a condition is met (think: LOOP)
   5. Where events are **only** done **if a condition** is met (think: ALT/OPT)

Solution



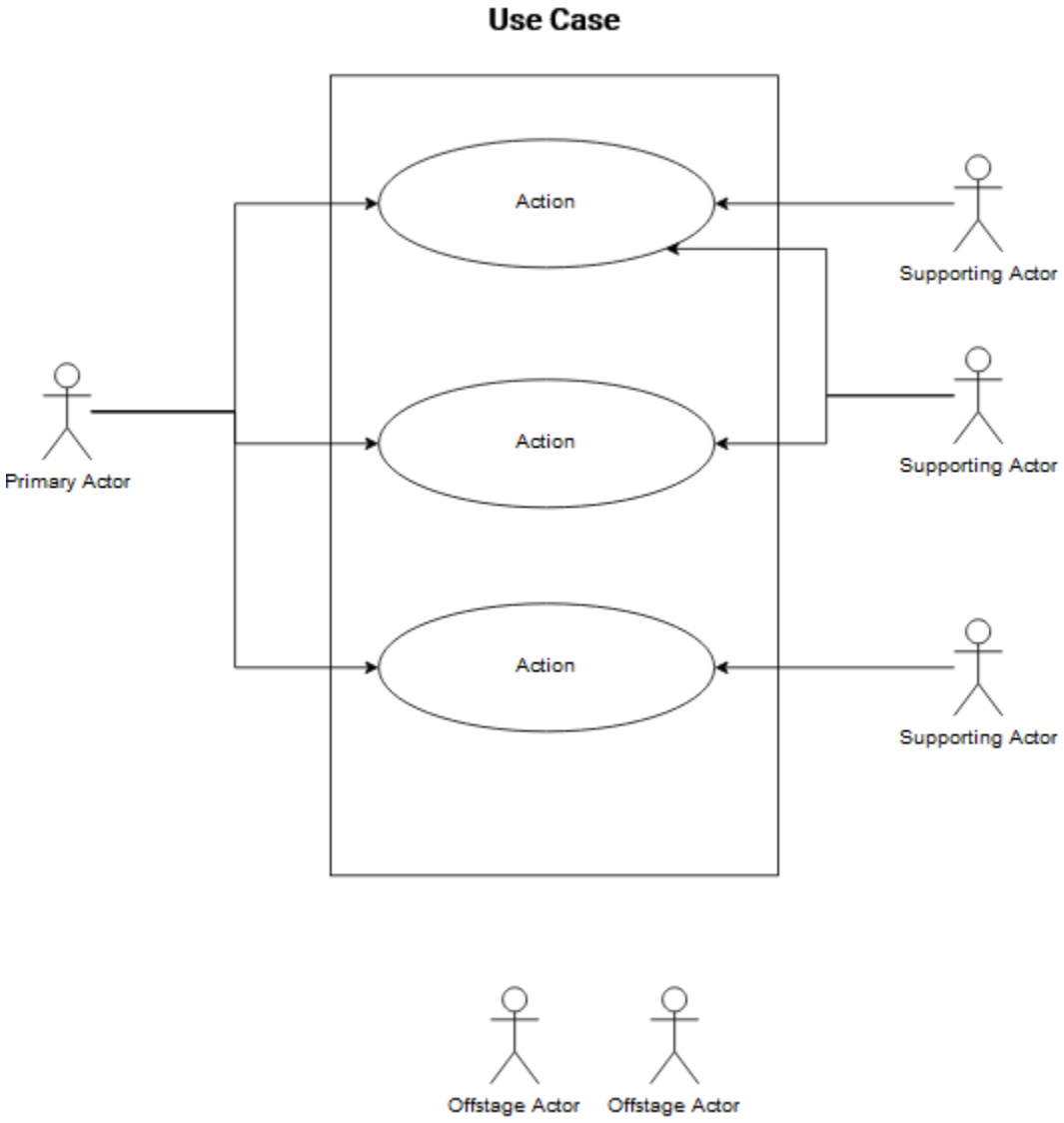
✓ Logical flow

■

L09 - Teamwork Considerations

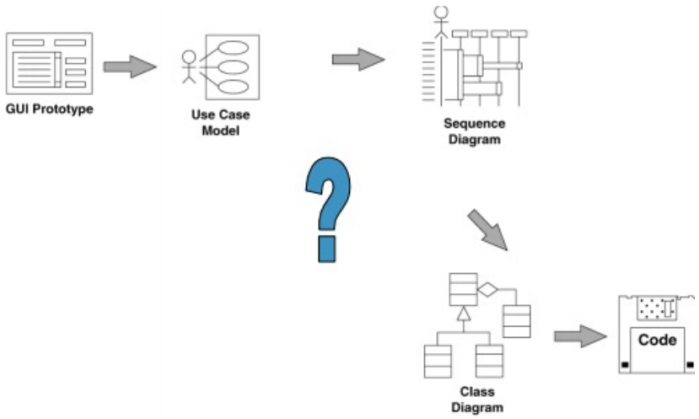
- **People** are the most important asset in a software organizations
    - People and retaining them cost a lot
- Because of this, people management is *key*; this is where people are assigned responsibilities that reflect their skills and experience
- Critical factors in people management…
    - *Consistency* - treated in a comparable way
    - *Respect* - different skills that should be respected
    - *Inclusion* - listened to, all views are considered
    - *Honesty* - about what is going well and not so well
- Most software engineering is a group activity; most non-trivial projects cannot be completed by one person
- Group interaction is a key determinant of group performance
- Managers do the best they can with available people; flexibility in group composition is limited
- Factors that affect group interactions…
    - *Composition* - task-oriented, self-oriented, interaction-oriented
    - *Cohesiveness* - quality standards, work closely together, egoless programming, learn from one another
    - *Communications* - promotes understanding, essential aspect of group interaction

    - *Organization* - relatively small sizes (< 8 members), big projects split into multiple small projects
- What happens when you're in a dysfunctional team? What are some characteristics of dysfunctional teams?
    - *Absorbing* - Group takes pride in getting the job done anyway

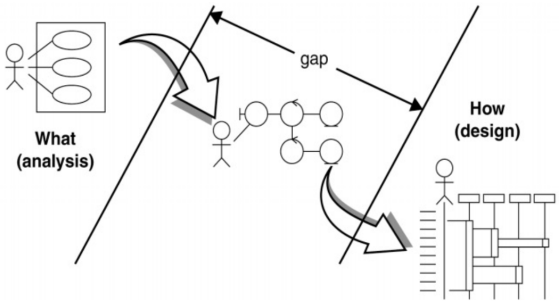


Lecture 10 - Robustness & Sequence Diagrams

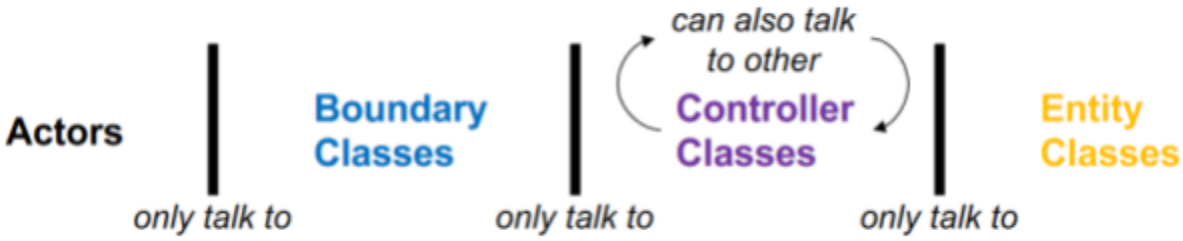
- So far, we've looked at multiple different diagrams we can use that can help us create code: use case models, system sequence diagrams, and domain models.
- Each of these serve a unique purpose in getting us one step closer to beginning to write our code.
- However, there are additional diagrams that we need to discuss in order to properly link our diagrams. One of these is the *robustness diagram*.



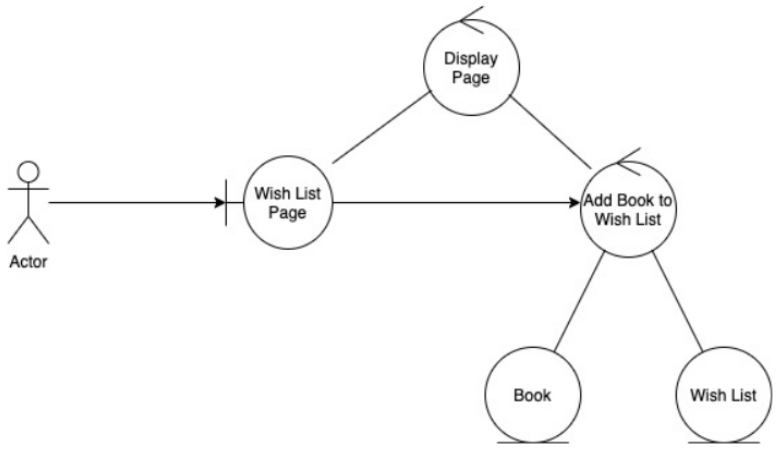
- **Robustness diagrams** bridge the gap from "what" to "how"; they diagram out the actor and their interaction with classes and the interactions those classes have between one another.



- There are some types of objects that appear in these robustness diagrams…
  - *Boundary Class* - a user interface or API class to external system
  - *Entity Class* - a class from the domain model
  - *Controller Class* - a class representing business logic or logical software function
  - *Actor* - the guy, ya' know



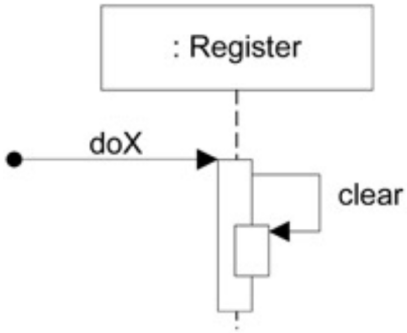
- Rules of Robustness Diagrams
    - Nouns can talk to verbs (and vice versa)
    - Nouns *can't* talk to other nouns
    - Verbs can talk to other verbs
- Here's an example of a robustness diagram…

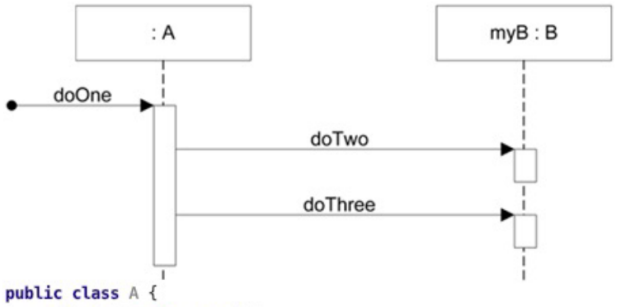


- Actor can talk to a Boundary Object
- Boundary Objects and Controllers can talk to each other (Noun ↔ Verb)
- A Controller can talk to another Controller (Verb ↔ Verb)
- Controllers and Entity Objects can talk to each other (Verb ↔ Noun)
- Some more notes about robustness diagrams…
    - Paste the use case text directly onto the diagram
    - Take the entity classes from the domain model and add any that are missing
    - Expect to rewrite your use case while drawing the robustness diagram
    - Make a boundary object for each screen and name your screens unambiguously
    - Remember that controllers are only occasionally *real controller objects*; they are more often *logical software functions*
    - Ignore the direction of the arrows on a robustness diagram
    - It's OK to drag a use case onto a robustness diagram
    - Boundary and entity classes on a robustness diagram will generally become object instances on a sequence diagram, while controllers will become messages
    - Remember that robustness diagram is an "object picture" of a use case, whose purpose is to force refinement of both use case text and the object model.
- **Sequence diagrams** are diagrams that are basically pseudocode...

- See on the right how the code of class A is almost explicitly diagramed out in the sequence diagram.
- Returns are illustrated with a dashed line like so…

- A class can also express calling a function in itself…

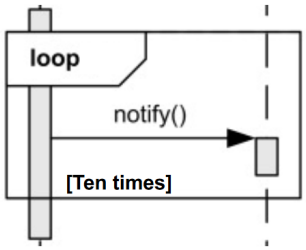
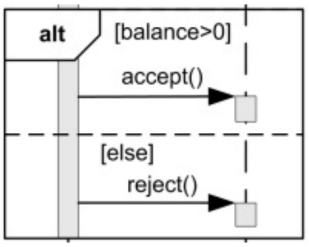
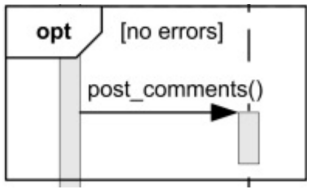


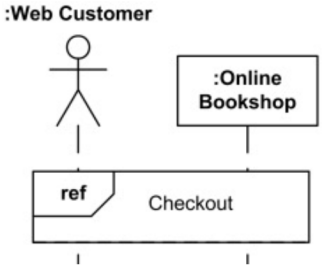



```
public class A {
    private B myB = new B();

    public void doOne()
    {
        myB.doTwo();
        myB.doThree();
    }

    //...
}
```

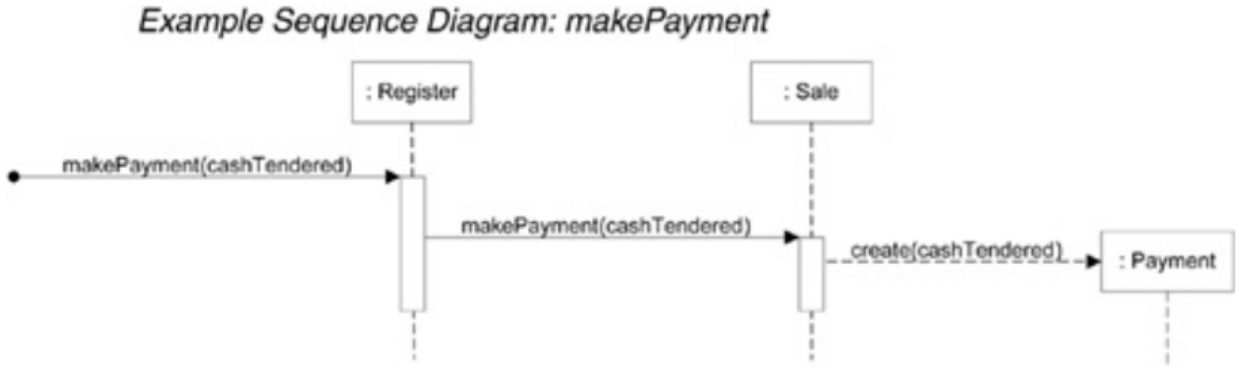
38

- Sequence diagrams can diagram complex interactions with *sequence fragments*

Loop / Iteration Here, we're calling notify() ten times	
Alternatives Call accept() if balance > 0, call reject() otherwise	
Option Post comments if there were no errors	

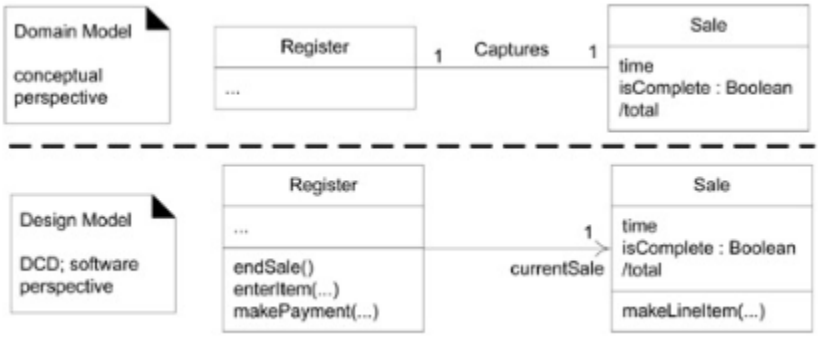
Reference Web customer and Bookshop use (reference) interaction Checkout	

- Here's an example of a sequence diagram…

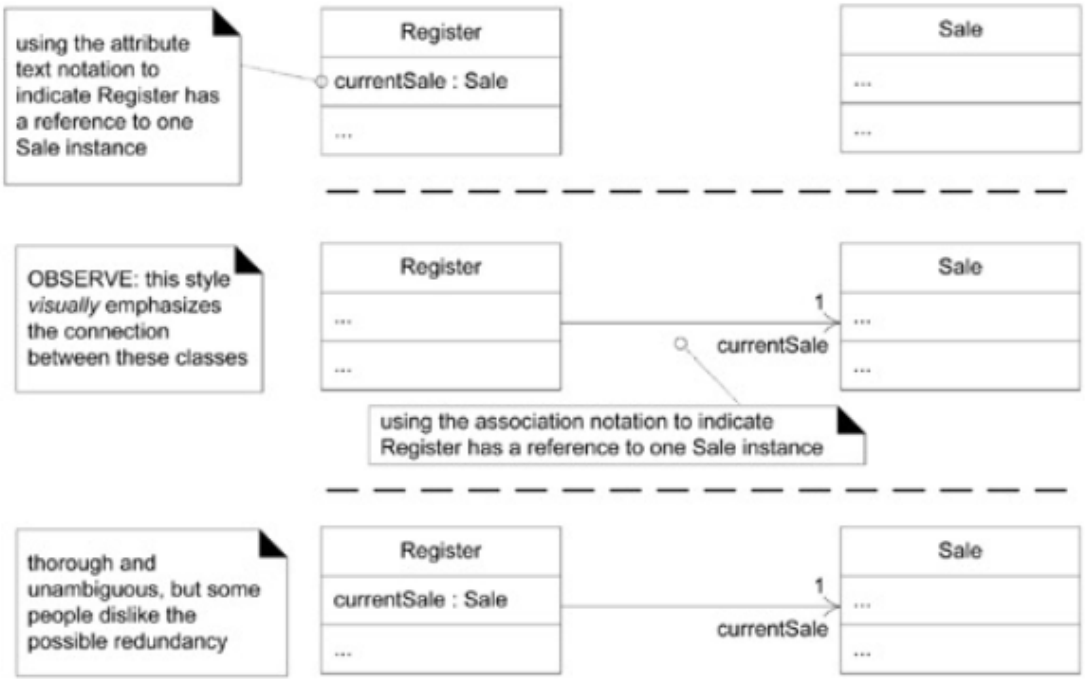


L11 - Design Class Diagrams

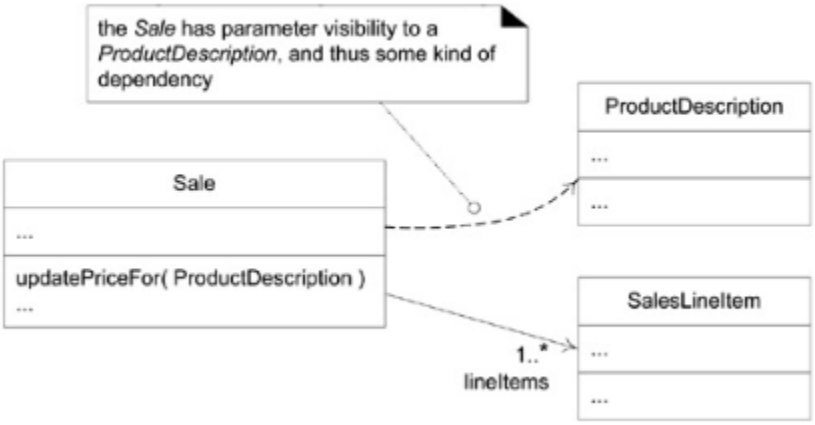
- Once the interaction diagrams have been completed, it's possible to identify the specification for the software classes and interfaces.
- Class diagrams differ from a Domain Model by showing software entities rather than real world concepts. Top picture shows a *domain model*; bottom picture shows a *design class* diagram



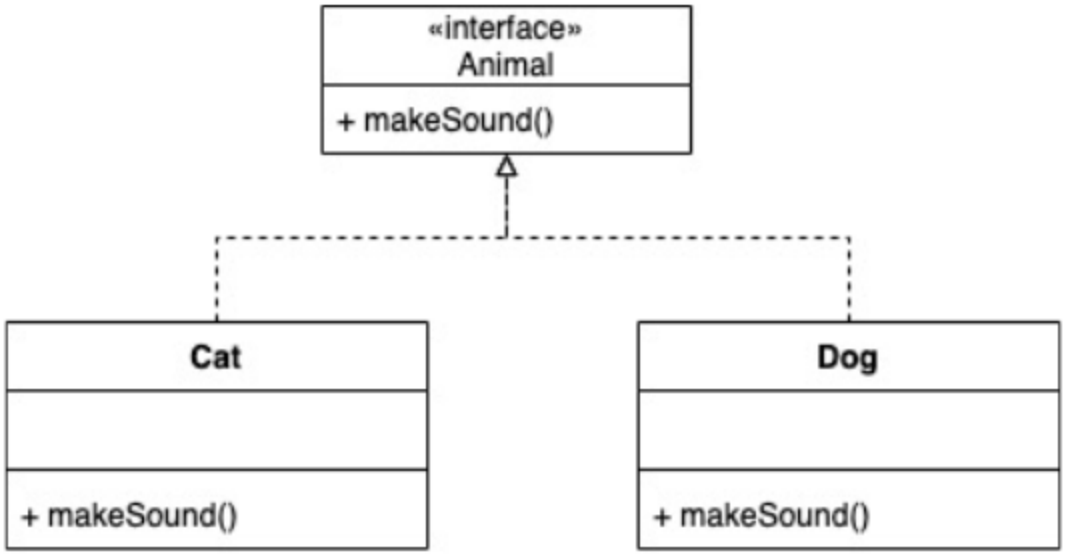
- Notice how the Design Model is literally the exact same as the Domain Model except that there is a third box that shows functions/methods
- The full format for class attributes is as follows…
    - visibility name : type multiplicity = default {property-string}
    - Visibility marks:
        - + : public
        - - : private
        - # : protected
    - Attributes are assumed private if no visibility is given
    - Operations assumed public if no visibility is given



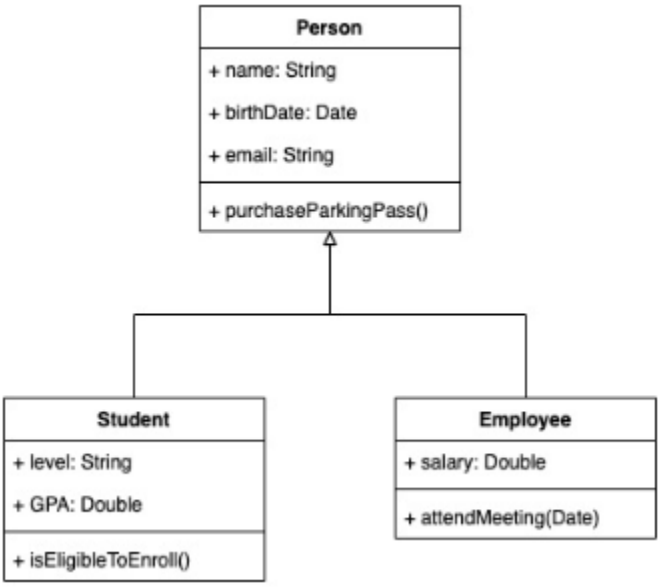
- To indicate a dependency, use a curved dashed arrow to a class…



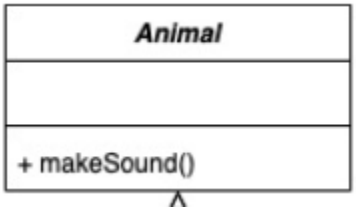
- Notice that in the above lines, the lines are denoted with attributes instead of brief descriptions (like in domain models). This is another way that class diagrams differ from domain models
- Interfaces only have 2 compartments, the name with the notation <<interface>> and the methods to implement. Notice how the arrow head is hollow and uses dashed lines…



- Inheritance is very similar to interfaces w/o the extra notation. Notice this time how the line is not dashed, although the arrow head is still hollow…

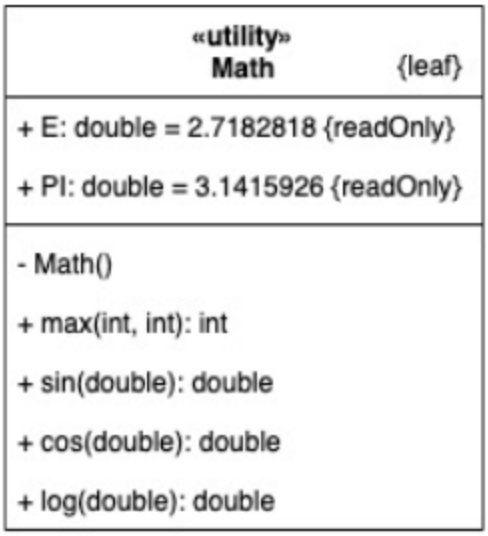


- Abstract classes look the exact same as inherited classes, except their title is in *italics*…



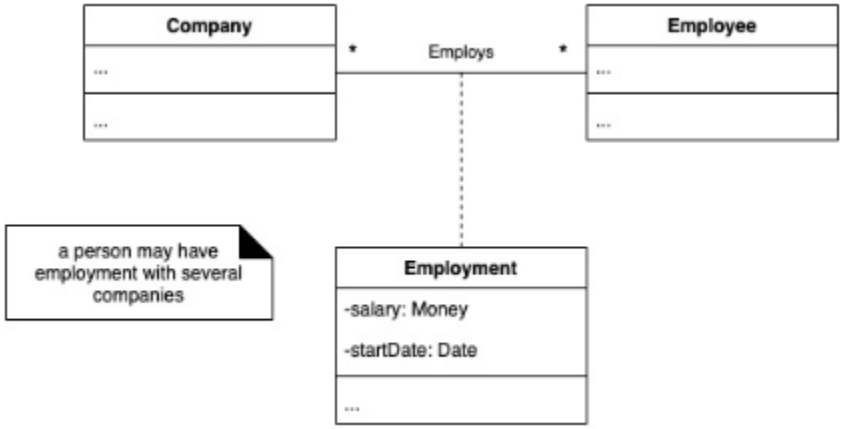
Utility Classes

- Must be denoted as a utility class and must be noted as {leaf}. {leaf} means that it is a class that *cannot* have children



Association Class

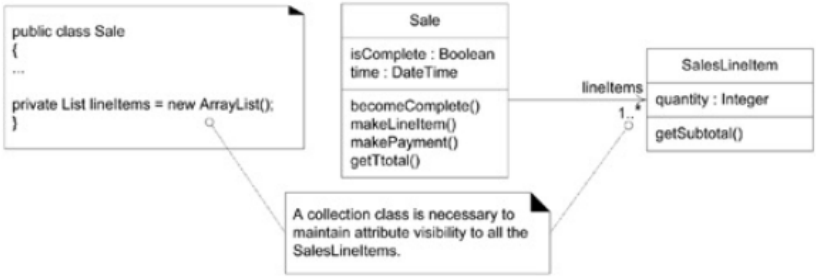
- If you want a relationship to have their own set of properties, it may be a good idea to implement an association class
- Usually pertains to “many-to-many” relationships



L12 - Object Visibility & Mapping Designs to Code

- The following is a list of popular acronyms for software development & design…
    - **Class-Responsiblity-Collaboration** (**CRC**) - cards that act as a brainstorming tool in object oriented design, usually done on index cards
    - **CRUFT** - A slang term for useless, redundant, or poorly written code
    - **Don’t Repeat Yourself** (**DRY**) - Avoid duplication in your lines of code
    - **Separation of Concerns** (**SoC**) - Separating a computer program into distinct sections, where each section addresses a separate concern

    - **You Aren't Gonna Need It (YAGNI)** - A programmer should not add functionality until it is deemed necessary
- *Object visibility* pertains to the ability of one object to see another object (**`private, public, protected`**)
- There are four common ways that visibility can be achieved between two objects…
    - *Attribute visibility* - B is an attribute of A
    - *Parameter visibility* - B is a parameter of a method in A
    - *Local visibility* - B is a (non-parameter) local object in the method of A
    - *Global visibility* - B is globally visible
      Refer to L12 slides for more information.
- If a class utilizes a collection of a type of object, we should draw a connection to that class and label the line with the name of the variable of the collection whose object type is that class.
- For

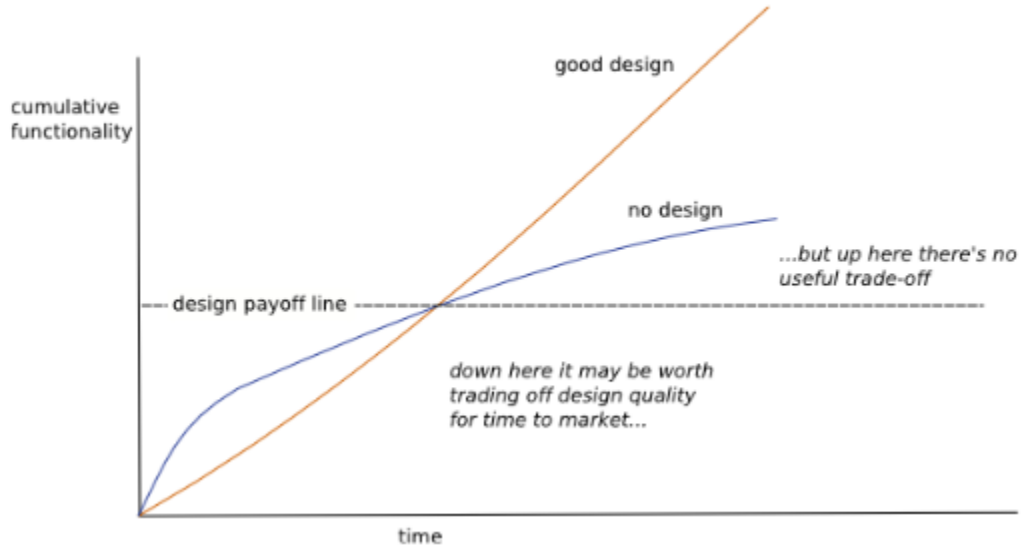


the order of implementation, it should be in order from the class that provides the least dependencies to classes that provide the the most dependencies (in terms of the UML diagram, it's the class that has the least amount of arrows *exiting* the class)

Lecture 13 - Code Smells, Responsibility Driven Design

- A **code smell** is a surface indication that usually corresponds to a deeper problem in the system. They are defined by being…
    - Quick to spot
    - Indicate a bigger problem in the code
- Brought about by examination or refactoring code
- Code smells are usually not a product of programmer ignorance, instead they usually come from a rushed design an a disregard for *technical debt*, the amount of work you create when you try to save time up front
    - *The Right Way* - use best practices and develop a design that can scale and grow; takes longer
    - *The Fast Way* - "hacked together" design; faster

Design Stamina Hypothesis



Taxonomy for Bad Code

Group name	Smells in group
Bloaters	-Long method -Large class -Long parameter list -Data clumps
Object-Orientation Abusers	-Switch statements -Refused Bequest
Change Preventers	-Divergent Change -Shotgun surgery
Dispensables	-Lazy class -Data class -Duplicated code
Couplers	-Feature envy -Inappropriate intimacy -Middle man

Bloaters

- *Large Class, Long Method*
    - When a class tries to do too much; often has too many instance variables
    - Object programs with short methods live longest
        - The longer a method, the harder it is to understand
        - Potential heuristic: If you feel like you need to comment something, write a method instead
- *Data Clumps*
    - **Data clumps** are sets of primitive data types that always appear together (e.g. 3 ints for RGB colors). Since these items are not encapsulated in the class, it increases the sizes of methods and classes.
- *Long Parameter List*
    - Usually appears when there are > 3 or 4 parameters
    - Hard to understand such lists
    - Reasons for the problem
        - Several types of algorithm are merged in a single method

Object Oriented Abusers

- *Switch Statements*
    - Occurs when you have a complex *switch* operator or sequence of if statements
    - Relative rare use of *switch* and *case* operators is one of the hallmarks of object-oriented code
- *Refused Bequest*
    - Subclasses inherit the methods and data of their parents, but they do not want what they are given
    - The unneeded methods go unused or be redefined and give off exceptions
    - Usually occurs when someone was motivated to create inheritance between classes only by the desire to reuse code in a superclass

Change Preventers

- *Shotgun Surgery*
    - Every time you make a change, you must make changes to a lot of classes
    - Symptom that functionality is spread among classes
    - Too much coupling between classes and too little cohesion within classes
- *Divergent Change*
    - Resembles **shotgun surgery** but is actually the opposite
    - **Divergent change** is when many changes are made to a single class; shotgun surgery refers to when a single change is made to multiple classes simultaneously
    - A possible reason is due to poor program structure or “copypasta” programming

Dispensables

- *Duplicated Code*
    - Most common smell of all; occurs when sections of code are repeated all over the place
    - Sign of an amateur
    - When refactoring duplicated code, you must effectively search for all instances of that code
- *Lazy Class*
    - If a class doesn't do enough to earn your attention, it's probably useless
    - Possible reasons include…
        - class was designed to be fully functional but after of the refactoring it has become ridiculously small
        - It was designed to support future development work that was never completed
- *Data Class*
    - A class that contains only fields and crude methods for accessing them (getters and setters); merely containers for data used by other classes
    - These classes do not contain any additional functionality and can't independently operate on their data

Couplers

- *Feature Envy*
    - A method that seems more interested in a class other than the one it is in
    - Most common focus of the envy is the data
    - To fix, determine which class has most of the data and put the method with that data
- *Inappropriate Intimacy*
    - Classes spend too much time going into each other's functionalities
    - Inheritance often leads to over intimacy
        - Subclasses are always going through their parents and their parents are always going through the subclasses
- *Middle Man*
    - If a class performs only one action, delegating work to another class, why does it exist at all?
    - Possibly exists because it's the result of the useful work of a class being gradually moved to other classes. The class remains as an empty shell that doesn't do anything other than delegate.

Lecture 14 - GRASP (5 Basic Principles)

Creator

- There are rules dictating who should create a new instance of a class. Assign class B to create class A if…
    - B *contains* or *aggregates* A
    - B *records* A
    - B *closely uses* A
    - B *has the initializing data* for A (B is an Expert with respect to creating A)
- If more than one option applies, then prefer aggregation (aka. The better it is)
    - Aggregation: objects can exist independently of each other (e.g. Classroom-Student) (uses hollow diamond)
    - Composition: objects can *not* exist independently of each other (e.g. Hand-Finger) (uses filled diamond)

Creator Pattern Discussion

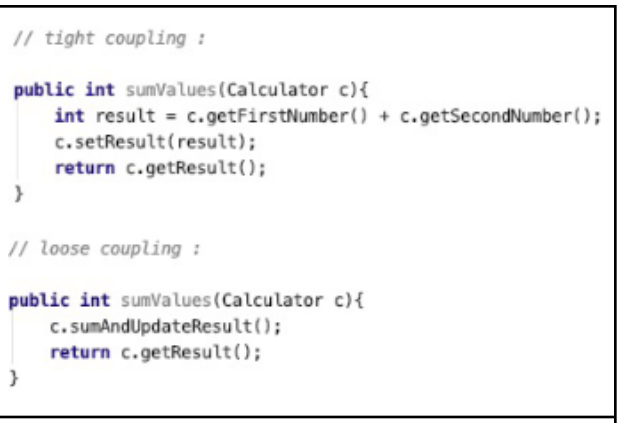
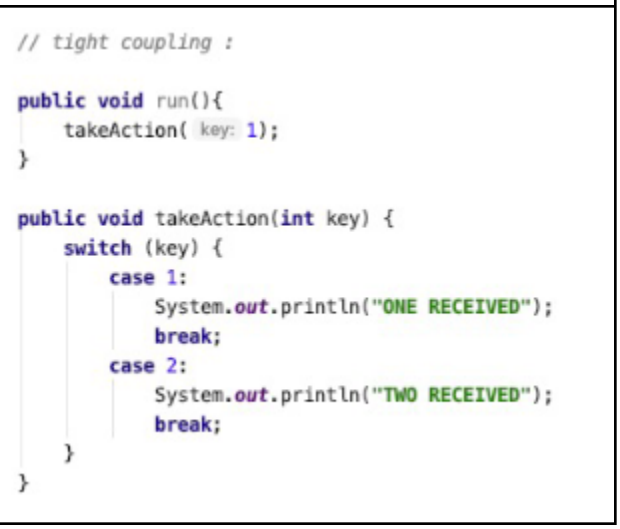
- Guideline 1: A composite object is an excellent candidate to make its parts
- Guideline 2: Look at the class that has the initializing data
    - E.g. A Payment instance must be initialized with the Sale total. Hence, Sale is a candidate creator of Payment.
- Guideline 3: In case of complex rules, consider delegation of creation to a helper class
- Benefits of doing this are: Low coupling (coupling = dependencies, we want less dependencies)

Information Expert (or Expert)

- Assign a responsibility to the class that has the information necessary to fulfill the responsibility
- A basic guiding principle of OOD
- Many "partial" information experts can collaborate in a task
- Real-world analogy: responsibility is given to individuals that have the information necessary to fulfill a task
- Benefits: Low coupling, high cohesion

Coupling

- Assign responsibilities such that coupling remains as low as possible
- Classes should create instances of other classes if the class its being created in has the information necessary to create an object of that type
- **Coupling** is a class's dependency on other classes. You want to have as low of coupling as possible, but *no* coupling is nigh impossible.
- Coupling from High to Low…

Content coupling - one class modifies another (branch into middle of routine, modifies code). Avoid at all costs.	
Common coupling - share common (global) data. Try not to do this.	
Control coupling - use a return code to control a different method. Try not to do this.	

<table>
<tr>
<td></td>
<td>

```java
// loose coupling :

public void run(){
    Printable printable = new PrinterOne();
    takeAction(printable);
}

public void takeAction(Printable printable){
    printable.print();
}

public interface Printable{
    void print();
}

public class PrinterOne implements Printable{
    @Override
    public void print() {
        System.out.println("ONE RECEIVED");
    }
}

public class PrinterTwo implements Printable{
    @Override
    public void print() {
        System.out.println("TWO RECEIVED");
    }
}
```

</td>
</tr>
<tr>
<td>

Stamp/Data coupling - passing complex data or structures between modules

Use primitives when possible.

```java
public class Printer {
    public void printNumber(int generatedNumber) {
        System.out.println(generatedNumber);
    }
}
```

Can lead to data coupling if the number of required parameters > 3

</td>
<td>

```java
public class Printer {

    // stamp coupling - happens when using another class as the type
    // of the method parameter
    public void printNumber(NumberGenerator generator) {
        System.out.println(generator.generate());
    }
}

public class PrinterWrapper {

    private final Printer printer;

    public PrinterWrapper(Printer printer) {
        this.printer = printer;
    }

    // we must depend on NumberGenerator.java although this class
    // is concerned with Printer.java Wrapping ... this class is now
    // coupled with NumberGenerator.java due to the method
    // parameter type in Printer.java
    public void print(NumberGenerator generator) {
        printer.printNumber(generator);
    }
}
```

</td>
</tr>
<tr>
<td>*Uncoupled* - no relationship</td>
<td></td>
</tr>
</table>

- Not all coupling causes problems…
    - High coupling to stable elements is usually not a problem
        - E.g. Java application coupled to Java libraries (java.util, etc.)

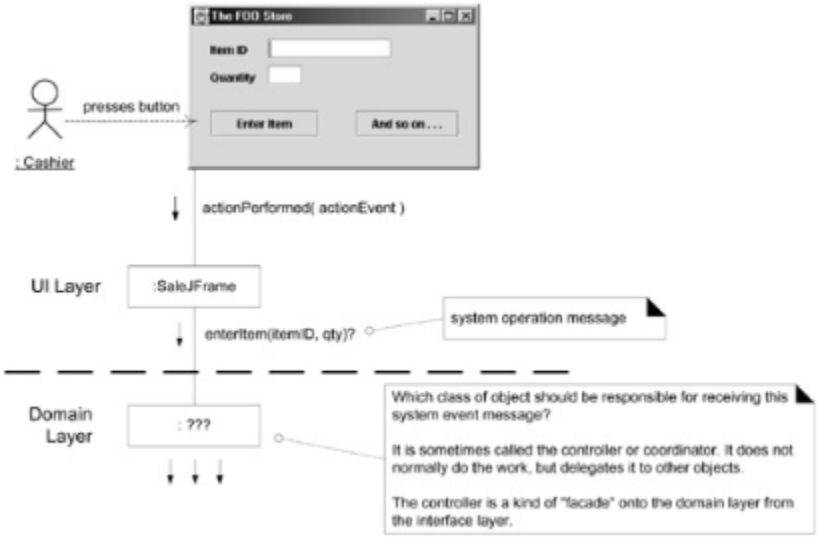
    - Use of collection classes for creation
        - E.g. Sale class to create instances of SalesLineItem
- Some coupling is normal, you cannot get rid of all coupling

Low Coupling Pattern Discussion

- Low Coupling is an evaluative principle for all design decisions
- Common forms of coupling…
    - TypeX has an attribute that refers to TypeY
    - TypeX calls on services of TypeY
    - TypeX has a method that refers to TypeY
    - TypeX is a subclass of TypeY
    - TypeY is an interface and TypeX implements it
- Contraindications
    - High coupling to stable elements is seldom a problem
- Benefits
    - Not affected by changes in other components; simple to understand in isolation; convenient to reuse

Controller

- A **controller** is the first object past the UI that receives and coordinates a system operation.
- Assign responsibilities to a class that…
    - Represents the overall System (fake Controller)
    - Represents a Use Case scenario where the event occur (<usecase name>Handler, <ucn>Coordination, <ucn>Session)
- Examples…



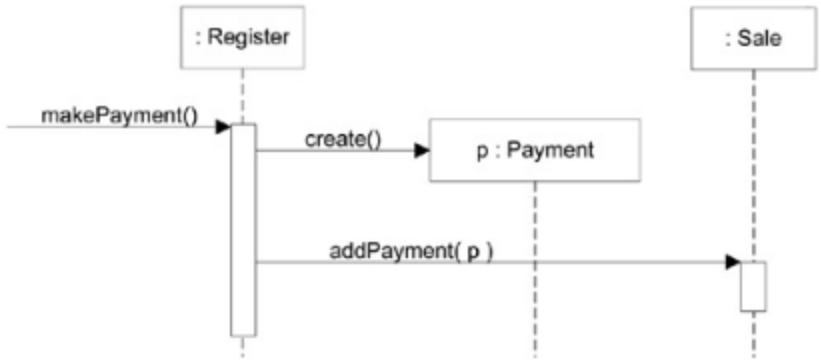
Represents the overall "system," "root object," device, or subsystem.	*Register, POSSystem*
Represents a receiver or handler of all system events of a use case scenario.	*ProcessSaleHandler, ProcessSaleSession*

- The Controller has many responsibilities...
    - Delegates to other objects work to be done
    - Coordinates & controls the activity
    - Does not do a lot of the work itself
    - KEY PRINCIPLE: UI objects should not have responsibility for fulfilling system events
- Pattern discussion…
    - Guideline: A controller should delegate to other objects the work that needs to be done
    - Fake controllers are suitable when there are few system events. Otherwise, apply Use case controllers
    - Benefits include…
        - Increased potential for reuse and pluggable interfaces
        - Opportunity to reason about the state of the use case

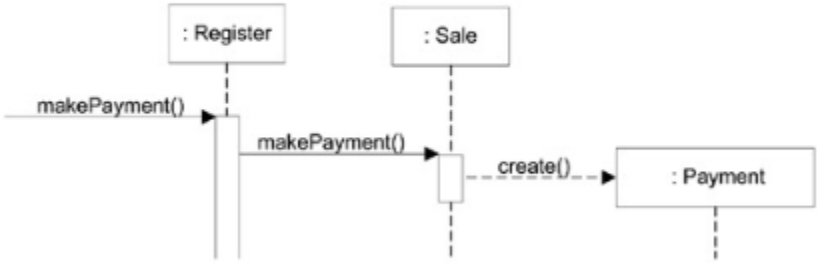
High Cohesion

- Keep objects manageable, focused, understandable and support Low Coupling
- Assign responsibilities so that cohesion remains high
- Objects should not do many unrelated things
- Classes with low cohesion are…
    - Hard to comprehend
    - Hard to reuse
    - Hard to maintain
    - Delicate; constantly affected by change
- **Cohesion** is how well each class's attributes and methods relate; a class should strive to more so be a master of one instead of a jack of all trades
- Example…

- Register creates Payment



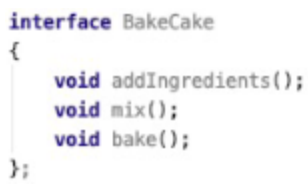
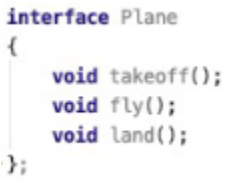
- Sale creates Payment



Cohesion (Low to High)

- Multiple types of cohesion…

Coincidental - Unrelated functions	`interface MyFuns { void initPrinter(); double calcInterest(); Date getDate(); };`
Logical - multiple logic sections	`interface AreaFuns { double circleArea(); double rectangleArea(); double triangleArea(); };`
Temporal - related by phases of an operation	`interface InitFuns { void initDisk(); void initPrinter(); void initMonitor(); };`

Procedural - required ordering of tasks	
Communicational - operates on same data set	
Functional - all essential elements for a single function are in same module	

What you want vs. What to avoid

What you Want	What to Avoid
• Low coupling • High cohesion • Low representational gap (LRG • Separation of concerns	• High coupling • Low cohesion • Needless complexity (YAGNI) • Needless repetition (DRY) • Opacity (obscure, obtuse): code should be straightforward to understand • Rigidity: Resistance to change; hard to predict how long it will take to make a change • Fragility: a change in one module breaks other modules • Immobility: same code doesn't work when move to another project • Hackability (viscosity): invites or accommodates workarounds more or less than fixes

Three Types of Developers

- *Novice* - Brittle designs. Easy to code, hard to maintain
- *Intermediate* - overly fancy, flexible, generalized (and normally unused) designs. Easy to maintain, hard to code.
- *Expert* - designs chosen with insight, balancing brittle with generalized. Easy to maintain, hard only when required

SOLID Principles

Purpose of SOLID

- Makes the code more maintainable
- Make it easier to extend the system with new functionality w/o breaking the existing one
- Make the code easier to read && understand; spend less time on what it does and more on development
- Introduced by Robert Martin (Uncle Bob), named by Michael Feathers

S.O.L.I.D

- S: Single Responsibility Principle
- O: Open/Closed Principle
- L: Liskov Substitution Principle
- I: Interface Segregation Principle
- D: Dependency Inversion Principle

S : Single Responsibility Principle (SRP)

- Each class should have a single overriding responsibility (high cohesion)
- Each class has one reason why it should change
- A class should be responsible for only one thing
- "There's a place for everything, and everything is in its place"
- Find one reason to change and take everything else out of the class
- Very precise names for many small classes > generic names for large classes

SRP Violation	SRP Compliant

IEmployeeStore.java
`public interface IEmployeeStore {`
`  public Employee getEmployeeById(Long id);`
`  public void addEmployee(Employee employee);`
`  public void sendEmail(Employee employee, String content);`
`}`
EmployeeStore.java
`public class EmployeeStore implements IEmployeeStore {`
`  @Override`
`  public Employee getEmployeeById(Long id) {`
`  return null;`
`  }`
`  @Override`
`  public void addEmployee(Employee employee) {`
`  }`
`  @Override`
`  public void sendEmail(Employee employee, String content) {`
`  }`
`}`
	IEmployeeStore.java
	`public interface IEmployeeStore {`
	`  public Employee getEmployeeById(Long id);`
	`  public void addEmployee(Employee employee);`
	`}`
	EmployeeStore.java
	`public class EmployeeStore implements IEmployeeStore {`
	`  //inject in runtime`
	`  private IEmailSender emailSender;`
	`  @Override`
	`  public Employee getEmployeeById(Long id) {`
	`  return null;`
	`  }`
	`  @Override`
	`  public void addEmployee(Employee employee) {`
	`  }`
	`}`

	IEmailSender.java
	`public interface IEmailSender {`
	`  public void sendEmail(Employee employee, IEmailContent content);`
	`}`
	EmailSender.java
	`public class EmailSender implements IEmailSender {`
	`  @Override`
	`  public void sendEmail(Employee employee, IEmailContent content) {`
	`  // logic`
	`  }`
	`}`
	IEmailContent.java
	`public interface IEmailContent {`
	`}`
	EmailContent.java
	`public class EmailContent implements IEmailContent {`
	`  private String type;`
	`  private String content;`
	`}`

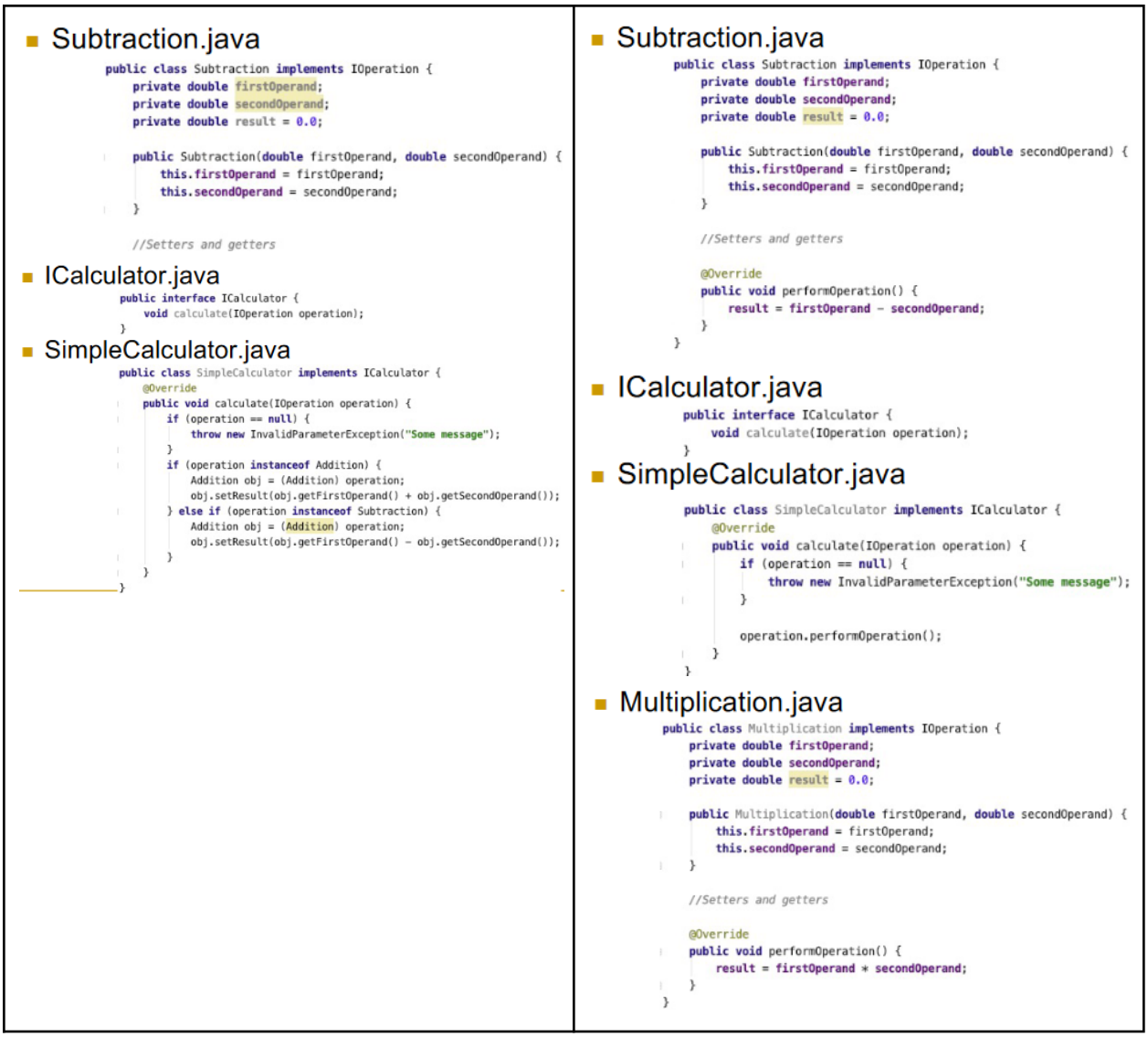
- Benefits include…
  - Easy to understand & maintain.
    - Interface has smaller number of methods
    - Changes related to the class's responsibility are fairly isolated
  - Improved usability
    - Can be used in other parts w/o exposing unrelated functionality

O : Open/Closed Principle (OCP)

- Objects are open for extension but closed for modification (making subclasses)
- Extension via inheritance, polymorphism
- Extended functionality by adding code instead of changing the existing one
- Separate the behaviors so that the system can be easily extended but never broken
- Goal: get to a point where you can never break the core of your system

Violation of OCP	Compliant with OCP

▪ IOperation.java
`public interface IOperation {`
`}`
▪ Addition.java
`public class Addition implements IOperation {`
`  private double firstOperand;`
`  private double secondOperand;`
`  private double result = 0.0;`
`  public Addition(double firstOperand, double secondOperand) {`
`  this.firstOperand = firstOperand;`
`  this.secondOperand = secondOperand;`
`  }`
`  //Setters and getters`
	▪ IOperation.java
	`public interface IOperation {`
	`  void performOperation();`
	`}`
	▪ Addition.java
	`public class Addition implements IOperation {`
	`  private double firstOperand;`
	`  private double secondOperand;`
	`  private double result = 0.0;`
	`  public Addition(double firstOperand, double secondOperand) {`
	`  this.firstOperand = firstOperand;`
	`  this.secondOperand = secondOperand;`
	`  }`
	`  //Setters and getters`
	`  @Override`
	`  public void performOperation() {`
	`  result = firstOperand + secondOperand;`
	`  }`
	`}`



- OCP is a guideline for how to develop code that allows change over time
- If you follow agile practices, with each sprint new requirements are inevitable and should be embraced
- By ensuring OCP, you effectively disallow future changes to existing classes, which forces programmers to create new classes that can plug into the extension points.

L : Liskov Substitution Principle

- “Subclasses should be substitutable for their base classes” : Created by Barbara Liskov
- Any derived class should be able to substitute its parent class w/o the consumer knowing it
- Every class that implements an interface, must be able to substitute any reference throughout the code that implements the same interface

- Every part of the code should get the expected result no matter what instance of a class you send to it, given it implements the same interface

Violation of LSP	Compliant with LSP

```java
public class RubberDuck extends Duck {
    public String quack () throws Exception {
        var person = new Person();

        if (person.squeezeDuck( rubberDuck: this)) {
            return "The duck is quacking";
        } else {
            throw new Exception("A rubber duck can't quack on its own")
        }
    }

    public String fly () throws Exception {
        throw new Exception("A rubber duck can't fly");
    }

    public String swim () throws Exception {
        var person = new Person();

        if (person.throwDuckInTub( rubberDuck: this)) {
            return "The duck is swimming";
        } else {
            throw new Exception("A rubber duck can't swim on its own");
        }
    }
}
```

- QuackableInterface.java

```java
public interface QuackableInterface {
    public String quack() throws Exception;
}
```

- SwimmableInterface.java

```java
public interface SwimmableInterface {
    public String swim() throws Exception;
}
```

- FlyableInterface.java

```java
public interface FlyableInterface {
    public String fly();
}
```

- RubberDuck.java

```java
public class RubberDuck implements QuackableInterface, SwimmableInterface {
    public String quack () throws Exception {
        var person = new Person();

        if (person.squeezeDuck( rubberDuck: this)) {
            return "The duck is quacking";
        } else {
            throw new Exception("A rubber duck can't quack on its own");
        }
    }

    public String swim () throws Exception {
        var person = new Person();

        if (person.throwDuckInTub( rubberDuck: this)) {
            return "The duck is swimming";
        } else {
            throw new Exception("A rubber duck can't swim on its own");
        }
    }
}
```

- The compliant implementation *still* violates the LSP because quack() and swim() do not work in certain conditions
- This cannot be fixed in the code, so it is a wrong abstraction
- Here's a good example of a proper subclass…

- FemaleDuck.java

```java
class FemaleDuck extends Duck{
    private FemaleDuckButt _butt;

    public FemaleDuck(){
        // Initialization of female stuff
        this._butt = new FemaleDuckButt();
    }

    public Egg layAnEgg(){
        Egg egg = this._butt.layAnEgg();
        return egg;
    }
}
```

I : Interface Segregation Principle

- Don't make large multipurpose interfaces - instead, use several small, focused ones
- Don’t make clients depend on interfaces they don’t use
- Classes should depend on each other through the smallest possible interface

Violation of ISP	Compliant with ISP
▪ Athlete.java ▪ JohnDoe.java Notice how the Athelete.java interface is trying to account for multiple types of Athletes	▪ Athlete.java ▪ SwimmingAthlete.java ▪ JumpingAthlete.java ▪ JohnDoe.java

Athlete.java

```java
public interface Athlete {
    void compete();
    void swim();
    void highJump();
    void longJump();
}
```

JohnDoe.java

```java
public class JohnDoe implements Athlete {
    @Override
    public void compete() {
        System.out.println("John Doe started competing");
    }

    @Override
    public void swim() {
        System.out.println("John Doe started swimming");
    }

    @Override
    public void highJump() {
    }

    @Override
    public void longJump() {
    }
}
```

Athlete.java

```java
public interface Athlete {
    void compete();
}
```

SwimmingAthlete.java

```java
public interface SwimmingAthlete extends Athlete {
    void swim();
}
```

JumpingAthlete.java

```java
public interface JumpingAthlete extends Athlete {
    public void jump();
}
```

JohnDoe.java

```java
public class JohnDoe implements SwimmingAthlete {
    @Override
    public void compete() {
        System.out.println("John Doe started competing");
    }
    @Override
    public void swim() {
        System.out.println("John Doe started swimming");
    }
}
```

D : Dependency Inversion Principle (aka Dependency Injection)

- High-level modules should not depend on low-level modules. Both should depend on abstractions
- Abstractions should not depend on details. Details should depend on abstractions.

Violation of DIP	Compliant with DIP
▪ BackEndDeveloper.java ▪ FrontEndDeveloper.java ▪ Project.java	▪ Developer.java ▪ BackEndDeveloper.java

BackEndDeveloper.java

```java
public class BackEndDeveloper {
    public void writeJava() {
    }
}
```

FrontEndDeveloper.java

```java
public class FrontEndDeveloper {
    public void writeJavascript() {
    }
}
```

Project.java

```java
public class Project {

    private BackEndDeveloper backEndDeveloper = new BackEndDeveloper();
    private FrontEndDeveloper frontEndDeveloper = new FrontEndDeveloper();

    public void implement() {
        backEndDeveloper.writeJava();
        frontEndDeveloper.writeJavascript();
    }
}
```

Developer.java

```java
public interface IDeveloper {
    void develop();
}
```

BackEndDeveloper.java

```java
public class BackEndDeveloper implements IDeveloper {
    @Override
    public void develop() { writeJava(); }

    private void writeJava() {
        System.out.println("We're writing Java!");
    }
}
```

	■ FrontEndDeveloper.java ■ Project.java

```java
public class FrontEndDeveloper implements IDeveloper {
    @Override
    public void develop() { writeJavascript(); }

    private void writeJavascript() {
        System.out.println("We're writing JavaScript!");
    }
}
```

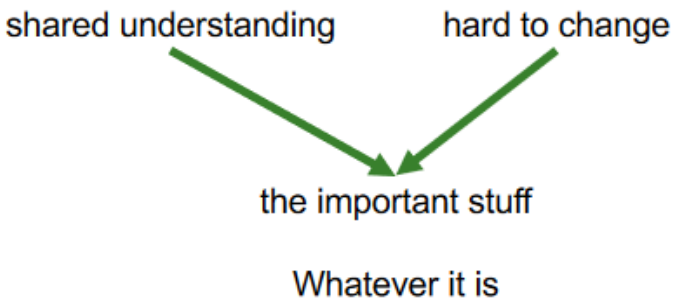
```java
import java.util.List;

public class Project {


    private List<Developer> developers;

    public Project(List<Developer> developers) {
        this.developers = developers;
    }

    public void implement() {
        developers.forEach(d -> d.develop());
    }
}
```

Don’t Get Trapped by SOLID

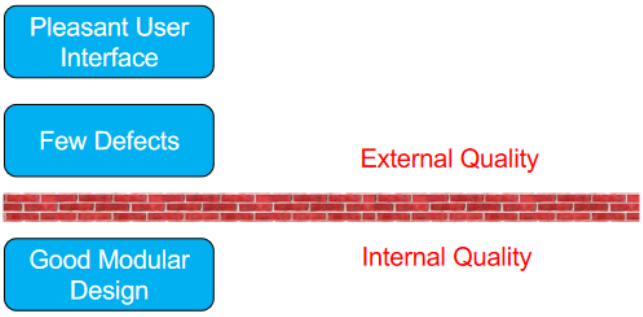
- SOLID design principles are *principles* not *laws*
- Always use common sense when applying SOLID
- Avoid over-fragmenting your code for the sake of SRP or SOLID
- Don’t try to achieve SOLID, use SOLID to achieve maintainability

Lecture 16 - Introduction to Software Architectures

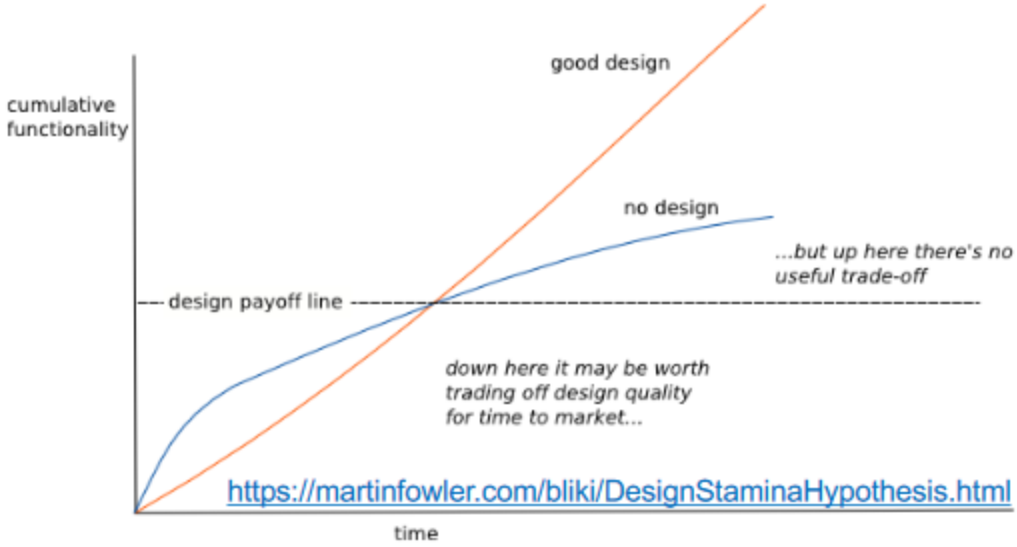
- **Software architecture**: the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution (description of the system)



    - "Expert developers' understanding of the system design" (social aspect) - Ralph Johnson
    - The set of design decisions that must be made early
    - The decisions that you wish you could get early on
    - The decisions that are hard to change
- When you join a software organization, you're always going to be pushed by the management…
    - "We need to put less effort on quality so we can build more features for our next release"
        - Economics point-of-view; people pay for features
    - "We need to stand up to our professional standards"
        - Craftsmanship point-of-view
    - This argument is always presented as a trade-off between quality and cost, and **economics always wins**
    - Consumers will usually pay for the cheaper products
    - If you want to make a point for your professional standards to the higher-ups, it must be done in a way that benefits the economics
- Architecture is an *internal* quality; the consumer doesn't get to see it.



- If the user doesn't get to see it, why does it matter?



- Projects that don't follow good design principles will initially have better features but will later have a lot harder of a time creating good features
- Time is money; if you have to go back and fix these things later to make implementing features viable, then it's time that could've been spent working on these features
- By following good design principles, in the long run it will be worth it **economically**.

Architectural Views

- A **view** is a representation of a set of system elements and their associated relations
- Views are representations of the structures **present simultaneously** in software systems
- Views are used to describe architecture
- Software architecture = {Element, Forms, Rationale/Constraints} - Perry and Wolfe
- Common Architecture Styles
    - N-Tier
        - Client-Server
    - Peer-to-Peer
    - Blackboard/Shared Memory
    - Pipe and Filter
    - Layered (Model-View-Controller)
    - Implicit Invocation/Event-Driven

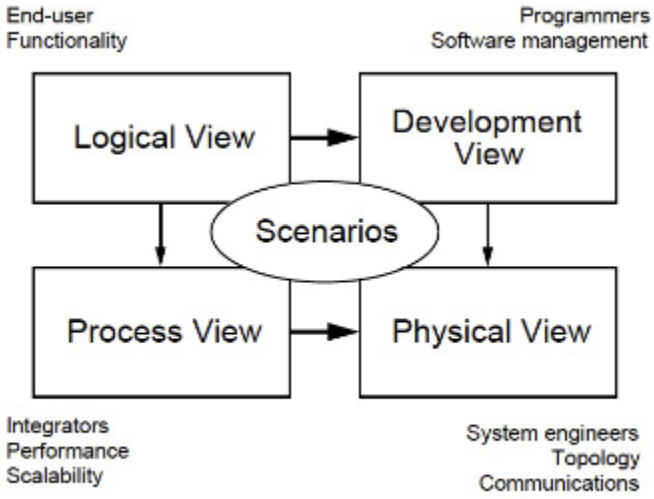
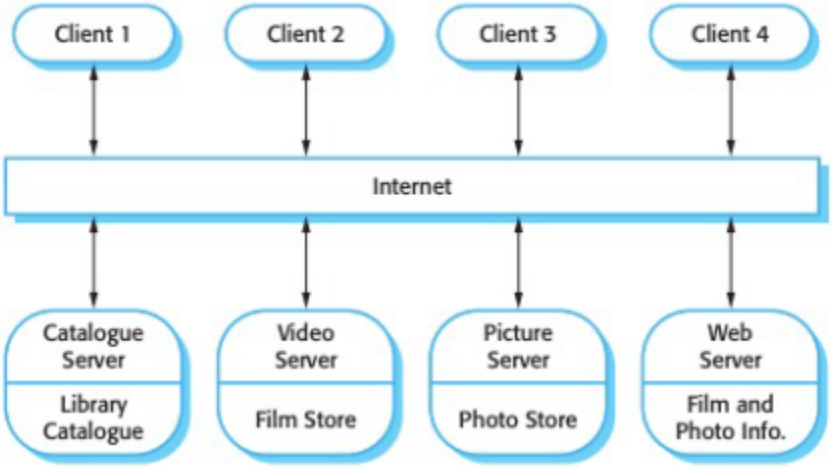


Figure 1 — The "4+1" view model

Client-Server Architecture

- A system is organized as set of services and associated servers, and clients that access and use the services

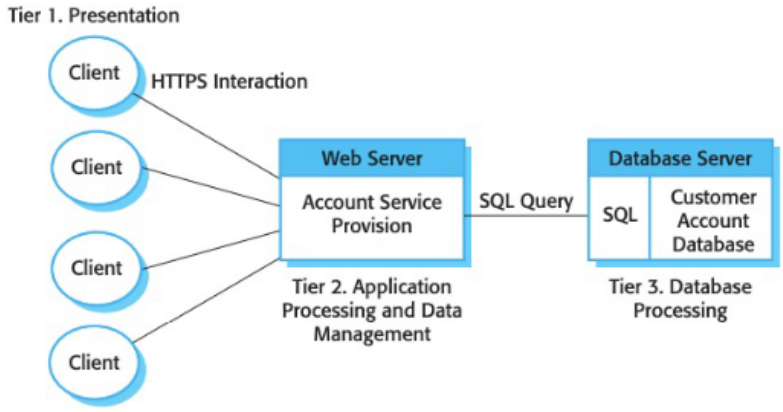
- Components include…
    - Set of servers that offer services to other components (e.g. print servers, file servers, email servers)
    - Set of clients that call on the services offered by servers
    - A network that allows the clients to access these services



- Most important advantage: a distributed architecture. General functionality (e.g. printing service) can be available to all clients and does not need to be implement by all services
- Disadvantage: each service is a single point of failure, so susceptible to denial of service attacks or server failure

N-Tier Architecture

- Fundamental problem with client-server architecture is that logical layers in the system (presentation, application processing, data management, and database) must be mapped onto two computer systems: client and server
- In n-tier, the different layers of the system are separate processes that may execute on different processors



Peer-to-Peer Architecture

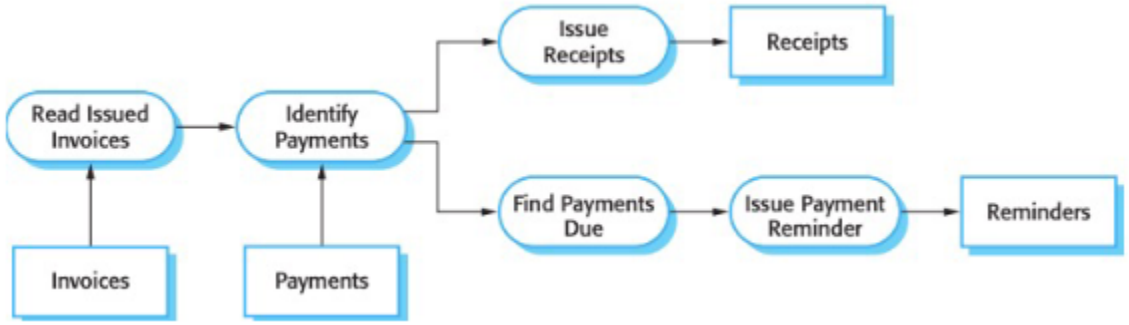
- Peer-to-Peer (P2P) systems are decentralized systems in which computations can be carried out by any node on the network
- No clear distinction between clients and servers
- Examples include: Gnutella and BitTorrent, ICQ and Jabber, SETI@home, Skype
- Appropriate in 2 circumstances…
    - Where the system is computationally intensive, and it is possible to separate the processing required into a large number of independent computations
    - Where the system involves the exchange of information between individual computers on a network and there is no need for this information to be centrally stored or managed
- Advantages: highly redundant and hence both fault-tolerant and tolerant of nodes disconnecting from the network
- Disadvantages: concerns about issues of security and trust; peers may behave in a malicious way

Blackboard/Shared Memory Architecture

- Designed for particularly complex problems
- A common knowledge base ("blackboard") is iteratively updated by a diverse group of specialist knowledge sources, starting with a problem specification and ending with a solution
    - A group of experts in a room all work on a "blackboard" to come up with a solution
- Each knowledge source updates the blackboard with a partial solution
- In this way, specialists work to solve the problem
- Key ideas that the problem solving should be both…
    - *Incremental* - complete solutions are constructed piece-by-piece and at different levels of abstraction
    - *Opportunistic* - system chooses the actions to take the next that will allow it to make the best progress
- Blackboard model defines three main components…
    - *Blackboard* - A structured global memory containing objects from the solution space
    - *Knowledge sources* - Specialized modules with their own representation
    - *Control component* - Selects, configures, and executes modules
- Examples include: Hearsay II speech recognition system, Adobe Acrobat's "OCR Text Recognition"

Pipe and Filter Architecture

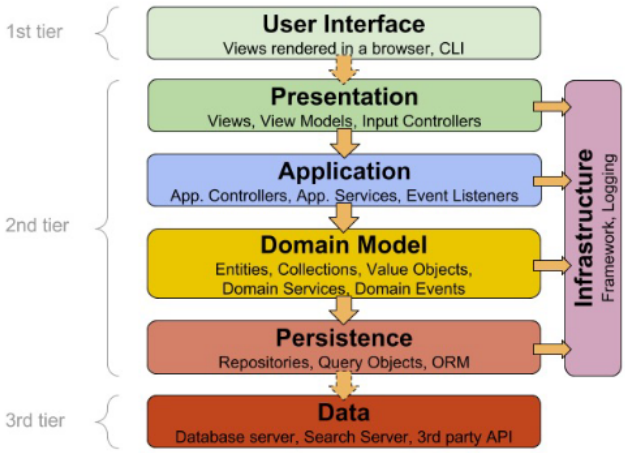
- Processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation
- Data flows (as in a pipe) from one component to another for processing
- Name "pipe and filter" comes from Unix where it is possible to link processes using "pipes"



- The business world is ideal for using the pipe and filter architecture
- Advantages include…
  - Easy to understand and supports transformation reuse
  - Workflow style matches the structure of many business processes
  - Evolution by adding transformations is straightforward
  - Can be implemented as either a sequential or concurrent system
- Disadvantages include…
  - Format for data transfer has to be agreed upon between communicating transformations
  - Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures

Layered Architecture

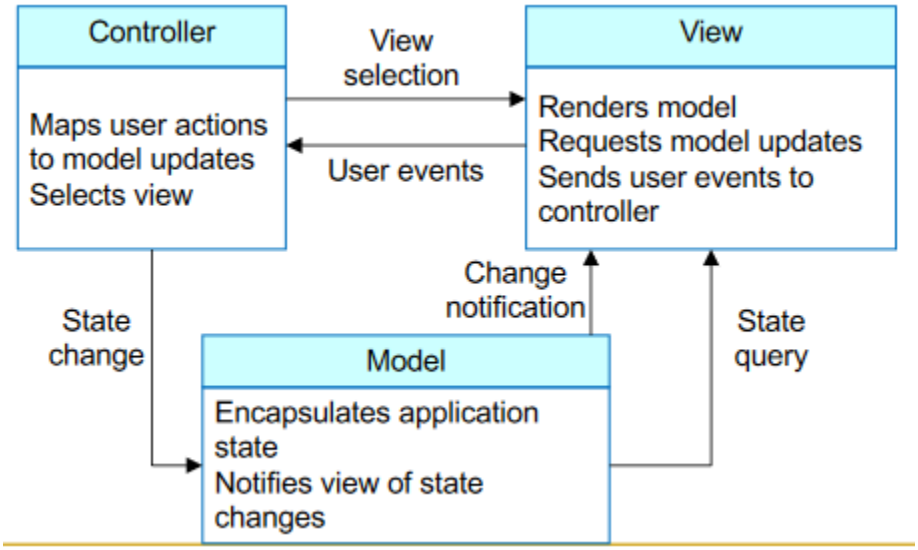
- Organizes the system into layers with related functionality associated with each layer
- In a layered system, each layer…
  - Depends on the layers beneath it
  - Is independent of the layers on top of it, having no knowledge of the layers using it
- Advantages include…
  - We only need to understand the layers beneath the one we are working on

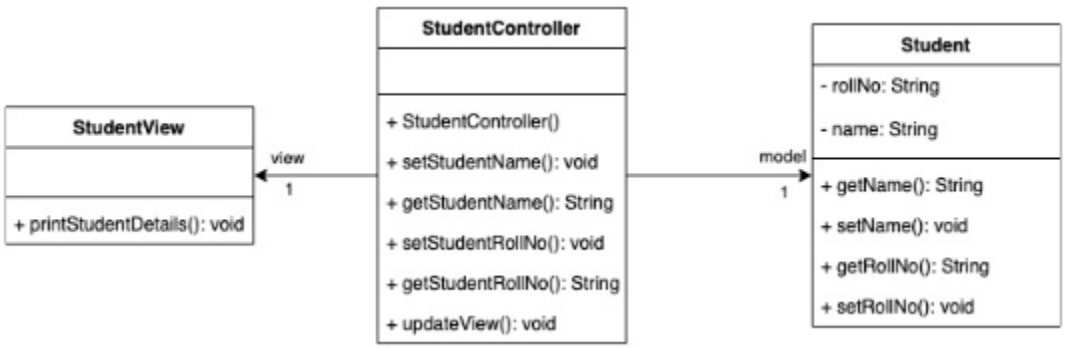


    - Each layer is replaceable by an equivalent implementation, with no impact on the other layers
    - A layer can be used by several different higher-level layers
- Disadvantages include…
    - Layers cannot encapsulate everything
    - Coupling of code over time
    - Considerable amount of time to build/test/deploy
- Whats the difference between N-tier and layered?
    - N-tiers = **physical** separation of tiers (can be 1 computer, but mostly 2)
    - Layered = **logical** separation of layers (1 computer)

<u>Model-View-Controller (MVC) Pattern</u>

- Follows layered approach
- Separates presentation and interaction from the system data
- System is structured into three logical components that interact w/ each other…
    - *Model Component*: manages the system data and associated operations on the data
    - *View Component*: defines and manages how the data is presented to the user
    - *Controller Component*: manages user interaction (e.g., key presses, mouse clicks, etc.)





Implicit Invocation/Event-Driven Architecture (EDA)

- Traditionally, components interact w/ each other by explicitly invoking routines
- With implicit invocation, instead of invoking a procedure directly, a component can announce (broadcast) one or more events
- Other components register an interest in an event
- When the event is announced, the system invokes all the interested procedures
  - FXML

```
<Button fx:id="decrButton" text="-"
        GridPane.columnIndex="0" GridPane.rowIndex="1"
        onAction="#handleDecrButtonAction"
        styleClass="buttons"
/>
```

  - Java

```
@FXML protected void handleDecrButtonAction(ActionEvent actionEvent) {
    gameModel.decrementState();
    statusText.setText(gameModel.getStateText());
}
```

Some Other Architectures

- Simple Object Access Protocol (SOAP)
  - XML-based messaging protocol for web services
- Microservice Architecture (MSA)
  - Arranges an application as a collection of loosely coupled services

Exam 2

- Can you recognize the type of cohesion, code smell, etc. based on code provided to you
- Possible diagrams…
  - Design Class diagram
  - Sequence diagram
  - Robustness diagram
- Architectural views will *not* be on Exam 2

Lecture 17 - Agile Methodology & eXtreme Programming (XP)

- 4 core values and 12 principles on how the creators of the methodology believe you should develop software
- Agile 4 Values…
    - Individuals & Interactions > processes and tools
        - Face-to-face communication w/ team members & customers
    - Working software > comprehensive documentation
        - Measure of success is software not thorough documentation
    - Customer collaboration > contract negotiation
        - Customer input throughout the project rather than at the start and end
    - Responding to change > following a plan
        - Change is the only constant; priorities can be shifted over iterations
- Agile 12 Principles…

12 AGILE PRINCIPLES BEHIND THE AGILE MANIFESTO

1 Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.	2 Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.	3 Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4 Business people and developers must work together daily throughout the project.	5 Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.	6 Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
7 Working software is the primary measure of progress.	8 The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.	9 Continuous attention to technical excellence and good design enhances agility.
10 Simplicity – the art of maximizing the amount of work not done – is essential.	11 The best architectures, requirements, and designs emerge from self-organizing teams.	12 At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

<u>eXtreme Programming</u>

- Instead of delegating time to each specific aspect of programming over an iteration that takes 2 - 3 weeks, every aspect of programming is worked on at the same time over iterations of 1 - 2 weeks.

XP's Practices

Pair Programming

- "We help each other succeed"
- One person codes - the driver
- The other person thinks - the navigator
- Work for a while, then alternate
- Possible pros:
    - Productivity is cut in half
- Cons:
    - Productivity rarely reduced
    - Resulting code benefits from two people working together

Root-Cause Analysis

- "We prevent mistakes by fixing our process"
- How to find the root cause: ask "why" five times
- Ex. When we start working on a new task, we spend a lot of time getting the code into a working state
    - Why? Because the build is often broken in source control.
    - Why? Because people check in code without running their tests
    - Why? Because sometimes tests take longer to run than people have available.
    - Why? …

Collaboration: Ubiquitous Language

- "We understand each other"
- How to speak the same language…
    - Programmers should speak the language of their domain experts, not the other way around
- Ubiquitous language in code
    - Low representational gap

Releasing: Done Done

- "We're done when we are production ready"
- You should be able to deploy the software at the end of any iteration
- Done Done
    - Tested; Coded; Refactored; Integrated; Builds; Installs; Migrates; etc.

Planning: Stories

- "We plan our work in small, customer-centric pieces"
- Two main characteristics

    - Stories represent customer value
    - Stories have clear completion criteria

Developing: Customer Tests

- "We implement tricky domain concepts correctly"
- Customers have domain knowledge that programmers don't have

<u>Scrum</u>

- Introduced in "The New Product Development Game" by Hirotaka Tekeuchi and Ikuhiro Nonaka in 1986
    - Stressed the importance of teamwork
- Jeff sucherland and Ken Schwaber conceived the Scrum process in 1995 at the OOPSLA conference
- *Scrum Actors*
    - Product owner (customer)
    - Team
    - Scrum master - ensures that all the best practices are followed
- Daily Scrum meeting questions
    - What did I work on yesterday
    - What will I work on today
    - What issues am I having
- **Velocity report**: how many story points a team is able to complete in one sprint on average
- **Burndown chart**: Chart indicating the actual number of stories closed in a particular sprint
- Basically is expectation vs. reality

Lecture 19 - GRASP: 4 Advanced Patterns

User Stories (cont.)

- Example priorities…
    - P0 - Show Stopper
    - P1 - Critical
    - P2 - Important
    - P3 - Nice to Have
    - P4 - Low Priority
- All estimates are wrong (but some are less wrong)
- To determine the least wrong estimation…
    - Work with the team together for a few weeks
    - Get some stuff done
    - Review the work done
    - Pick an "average" or "medium" piece of work
        - Medium work should be completed within a spring, not too easy and not too hard
- "T-Shirt Sizes Estimation" - Work is not linear; it's exponential/fibonacci. Once you have your medium workload, everything else can be based on the medium size…

T-Shirt Size (Story)	Point Value	T-Shirt Size (Epics)
XXS	1	
XS	2	
SM	3	
M	5	
L	8	
XL	13	XXS
XXL	21	XS
	34	SM
	55	M
	89	L
	144	XL
	233	XXL

Advanced GRASP

- The previous section discussed the 5 basic GRASP principles; today are the 4 advanced GRASP principles…

<u>Polymorphism</u>

- Problem: How to handle alternatives based on type? How to create pluggable software components?

- Solution: When related alternatives or behaviors vary by type (class), assign responsibility for the behavior--using polymorphic operations--to the types for which the behavior varies

Advantages	Disadvantages
- Easier to add additional behaviors later on - Reduces the coupling between different functionalities	- Increases the number of classes in a design - May make the code less easy to follow

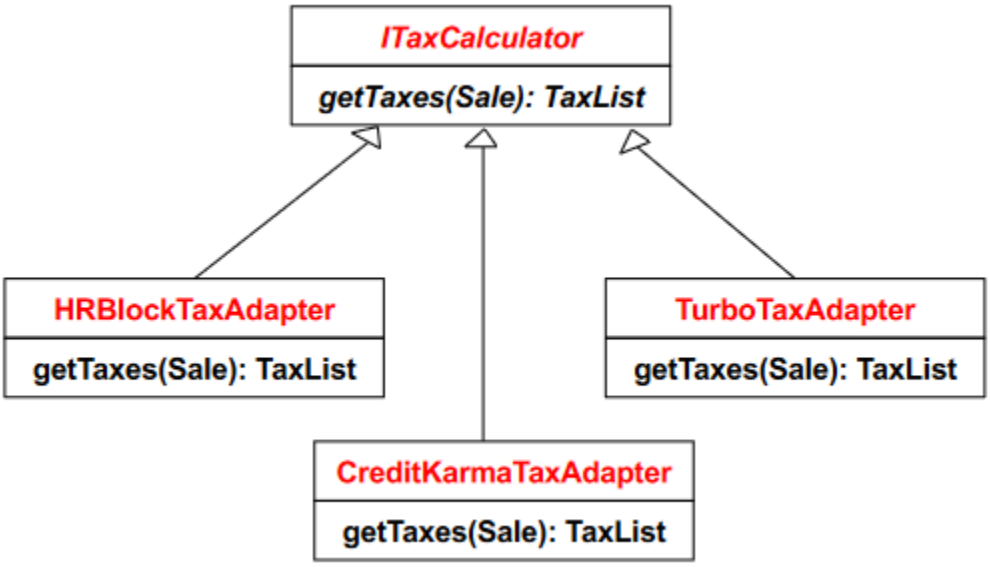
Pure Fabrication

- Problem: Adding some responsibilities to domain objects would violate high cohesion/low coupling/resuse
- Solution: Assign a highly cohesive set of responsibilities to an **artificial** or **convenience** class that does not represent a problem domain concept--something made up, to support high cohesion, low coupling, and reuse
- When no appropriate class is present: invent one
    - Even if the class does not represent a problem domain concept
    - "Pure fabrication" = making something up: do when we're desperate
- This is a compromise that often has to be made to preserve cohesion and low coupling
    - Domain Model =/= Design model
- Example…
    - Suppose Sale instances need to be saved in a database
    - Option 1: Assign this to the Sale class itself (*Expert* pattern)
        - Requires that a relatively large number of supporting database-oriented operations => Sale class becomes incohesive
    - Option 2: create a new class that is solely responsible for saving objects in a persistent storage medium
        - Sale remains well-designed, with high cohesion and low coupling
        - PersistentStorage class is itself relatively cohesive
        - PersistentStorage class is a very generic and reusable object

Indirection

- Problem: Where to assign a responsibility to avoid direct coupling between two or more things? How to decouple objects so that low coupling is supported and reuse potential remains high?
- Solution: Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled. This intermediary creates an indirection between the other components.
- A common mechanism to reduce coupling

- Assign responsibility to an intermediate object to decouple two components
    - Coupling between two classes of different subsystems can introduce maintenance problems
- "Most problems in CS can be solved by another level of indirection:
    - Pure Fabrication
    - Observer Pattern
    - Facade Controller Pattern

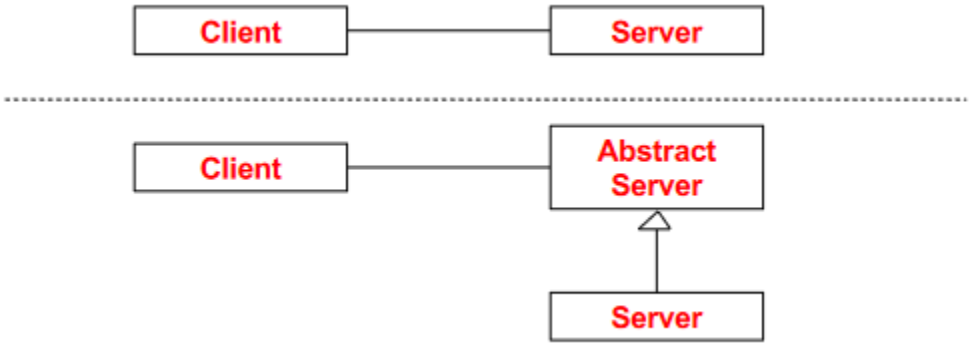


- To replace the current calculator with a new one…
    - Implement a new Adapter
        - New code: no way around it
    - Only minor change in existing code
        - New XAdapter() -> new YAdapter()

Protected Variations

- Problem: How to design objects, subsystems, and systems so that variations or instabilities in the elements do not have an undesirable impact on other elements
- Solution: Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them
- Fundamental problem in design: current and future variations
- Protect the rest of the system from these variations
- Typical mechanisms for protection…
    - Encapsulation
    - Abstraction
    - Polymorphism
    - Indirection

- Encapsulation
  - Basic mechanism, typically in the programming language
  - Group related entities into a single unit, and provide a restricted external view of the unit
  - First form of encapsulation: procedures
    - Encapsulates a set of statements
    - External view: name and parameters
    - Protection against variations in algorithm
- Object-Oriented Encapsulation
  - Packaging of operations and attributes
  - Attributes represent internal state that is not directly accessible
    - Hidden behind a "wall" of operations
  - State is accessible and modifiable only via the operations
  - Protections against…
    - Changes in data representation
    - Changes in algorithm
- Abstraction
  - Common theme in software design
  - Low-level abstractions
    - Procedural abstractions: subroutines
    - Data abstractions: e.g. classes
      - Protect against variations within the class
  - Higher-level abstractions for OOD



- PV Summary
  - Protecting one part of the program from changes in another part
    - "Protected variations", "open-close principle", "information hiding"
    - Common mechanisms to achieve it: encapsulation, abstraction, polymorphism, indirection
  - Many more "ambitious" mechanisms

        - E.g at run time, clients use a lookup service to find a server object (protects against location and implementation changes)

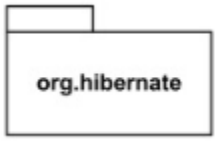
Don’t Talk to Strangers or the Law of Demeter

- States that within a method, messages should only be sent to the following objects..
    - The *this* or *self* object
    - A parameter of the method
    - An attribute of *this*
    - An element of a collection which is an attribute of *this*
    - An object created within the method

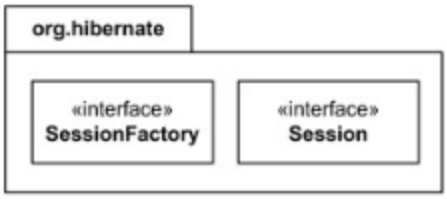
Lecture 20 - 6 Advance SOLID Principles

Package Diagrams

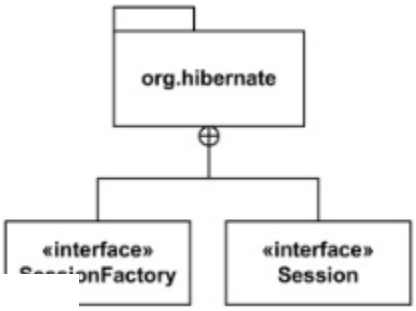
- A **package** is a grouping of related UML elements, such as classes, other packages, use cases, etc.
- It's common to show dependencies/couplings between packages
- Most commonly used to illustrate the logical architecture of a system
- Provides structural (static) view of the system
- Notation
    - Packages appear as rectangles w/ small tabs at the top
    - Package name is on the tab or inside the rectangle
    - Dotted arrows are dependencies (The arrow *points* to dependencies)
    - One package depends on another if changes in the other could possibly force changes in the first
    - If members of the package are not shown inside the package rectangle, then the name of the package should be placed inside (*see right*).



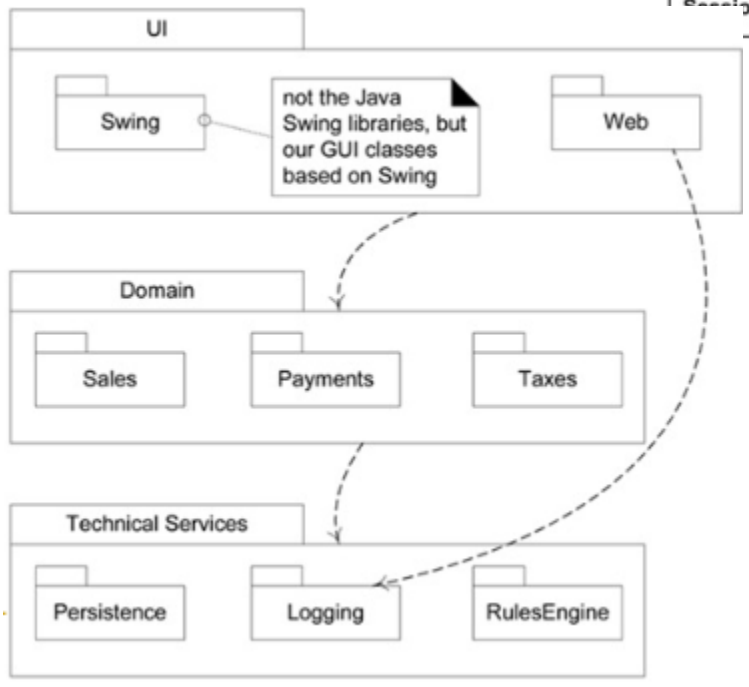
    - The members of the package may be shown within the boundaries of the package. In this case, the name of the package should be placed on the tab (*see right*).



    - Members of the package may be shown outside of the package by branching lines from the package to the members (*see right*)



    - See larger example below…

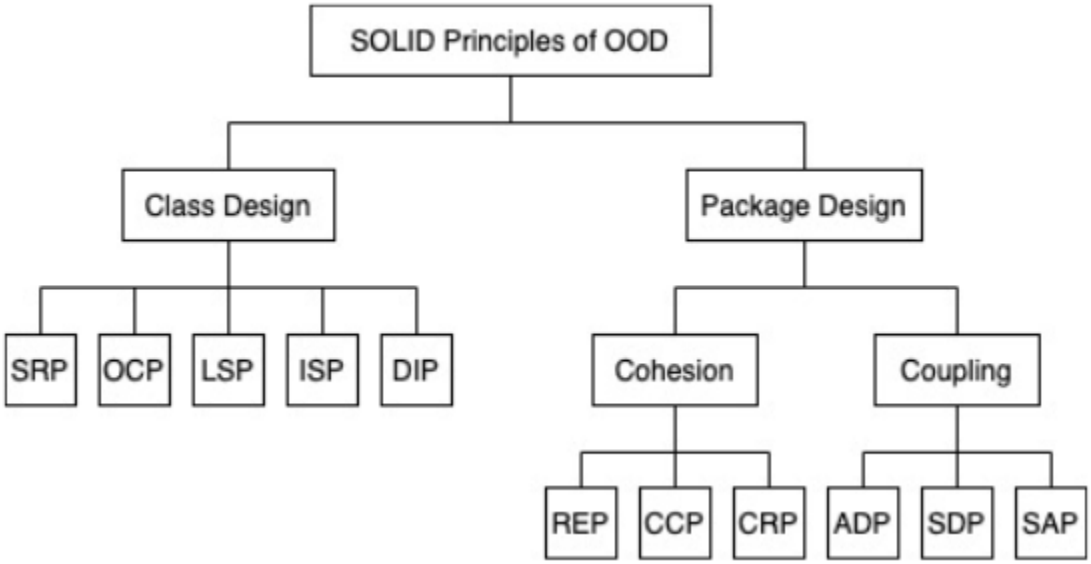


Granularity

- Need for a large granule - package
    - Smalltalk packages, Rugby gems, JavaScript libraries, Node modules, Java JAR files

Designing with Packages

- What is the best criteria for partitioning the classes into groups?
- What are the relationships between packages?
- Top/down vs. bottom/up approach?
- How are packages physically represented?
- To what purpose will put these packages



REP: Release-Reuse Equivalency Principle

- "*The granule of reuse is the granule of release*"
- What is reuse? Is it reuse if I copy code from another program?
- Disadvantages of code copying
    - You own the cody you copy
    - You have to maintain it, fix bugs, update it if the original code changes, etc.
- What you want
    - Never have to look at the source code
    - Be notified about updates
    - Be able to decide when it integrate changes
- REP-Compliant Example (Look at the imports)

```
import java.io.ByteArrayOutputStream;
import java.io.IOException;

class Main {
  public static void main(String[] args) throws IOException{
    ByteArrayOutputStream os = new ByteArrayOutputStream();
    os.write("my word".getBytes());

    os.writeTo(System.out);
  }
}
```

- Violation of REP example (look at the imports; importing a lot)

```
import Player.*;
import Armor.*;
import Weapon.*;

class Main {
  public static void main(String[] args) {
    Player warrior = new Player("Riley",
                                new Weapon("Axe"),
                                new Armor("Tunic"));

    StringBuilder info = new StringBuilder();
    info.append("Name: " + warrior.name + "\n");
    info.append("Weapon: " + warrior.mainWeapon() + "\n");
    info.append("Armor: " + warrior.mainArmor() + "\n");

    System.out.println(info.toString());
  }
}
```

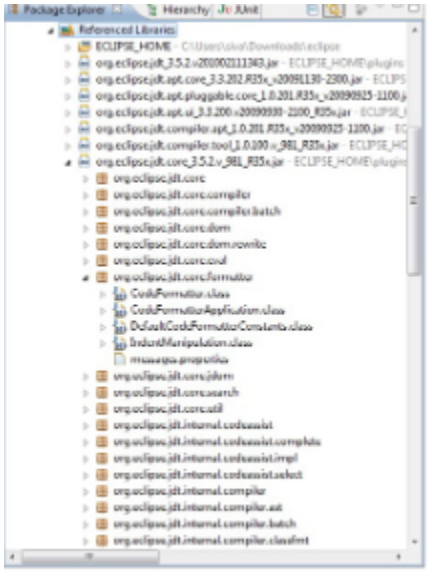
- Fixed to comply with REP (notice the imports; here, we only import GameObjects.* instead of importing every class individually)

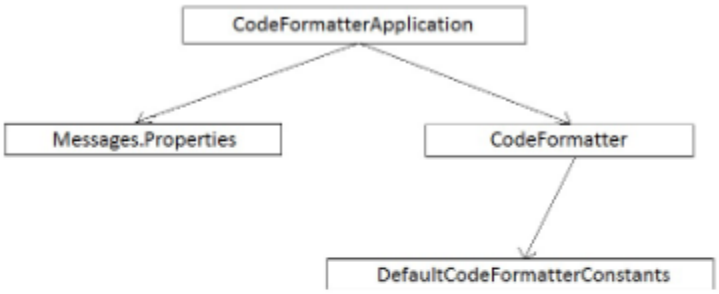
```
import GameObjects.*;

class Main {
  public static void main(String[] args) {
    Player warrior = new Player("Riley",
                                new Weapon("Axe"),
                                new Armor("Tunic"));


    StringBuilder info = new StringBuilder();
    info.append("Name: " + warrior.name + "\n");
    info.append("Weapon: " + warrior.mainWeapon() + "\n");
    info.append("Armor: " + warrior.mainArmor() + "\n");

    System.out.println(info.toString());
  }
}
```

The Common Reuse Principle (CRP)

- "*Classes that are used together are packaged together*"
- Helps decide which classes should be placed into a package. Classes that tend to be reused together belong in the same package.
- The first one is more about not having to maintain/upkeep because it's coming from another package, whilst this one is more about combining like classes together in a package
- Why
    - Packages in Java are distributed in JAR files
    - If one package uses a class in another package, even if it only uses a single method in a single class, it has a strong dependency

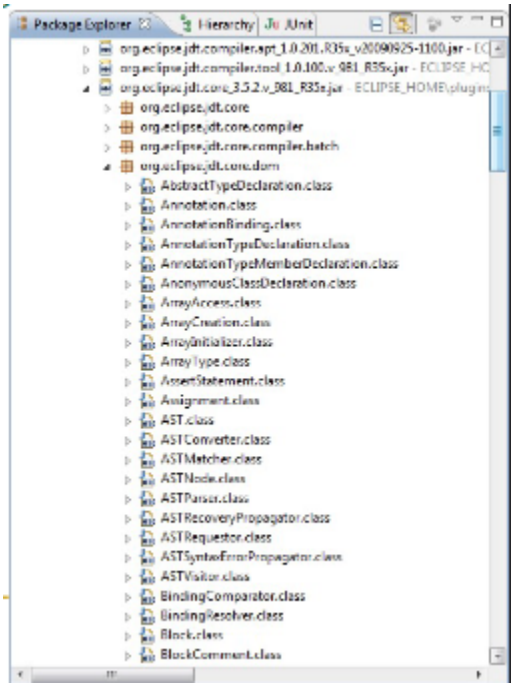
- How:
    - CRP says something about which classes to put in a package but mostly which classes **not** to put in the package
- Example





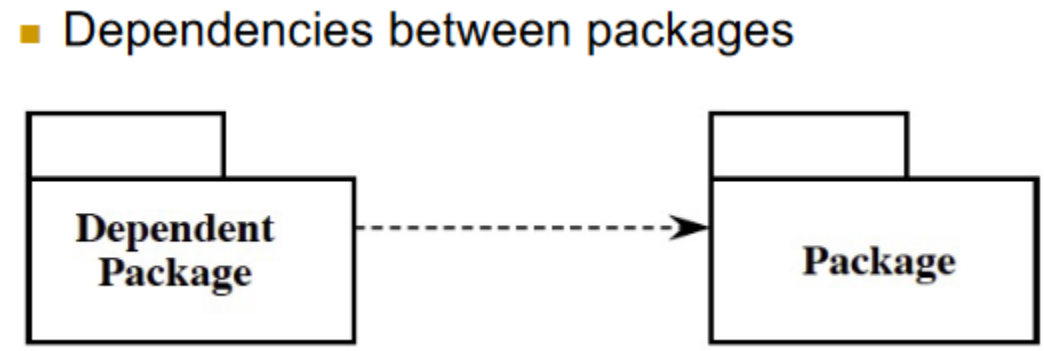
The Common Closure Principle (CCP)

- "*Classes that change together are packaged together*"
- If the code in an application must change, where would you like those changes to occur: in one package or many packages?
    - The less packages, the better
- If changes are focused into a single packed, we need to redeploy only the one that changed
- Other packages that don't depend on the changed package do not need to be revalidated or redeployed
- Example…
    - Org.eclipse.jdt.core.dom contains a set of AST classes that model the source code of a Java program as a structured document
    - If a new language is to be supported, then the changes will be made only in this package. Other packages will remain the same.

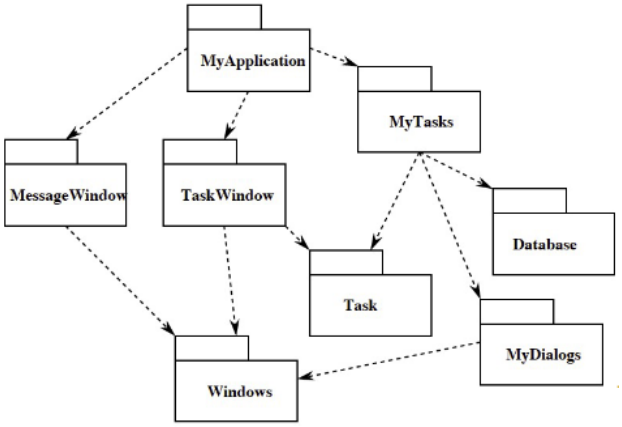


The Acyclic Dependencies Principle (ADP)

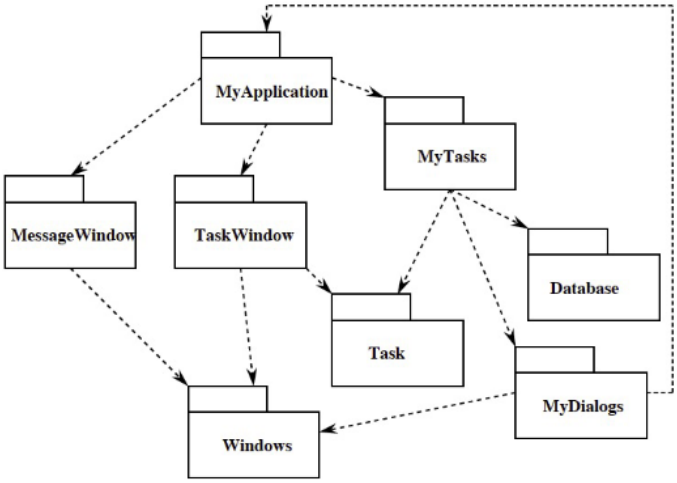
- "*The dependency graph of packages must have no cycles*"
- When two or more packages are involved in a dependency cycle, it becomes very difficult to stabilize the application
- A naive automated build system can be completely defeated by a cycle in a package graph



- Package Diagram w/o Cycles

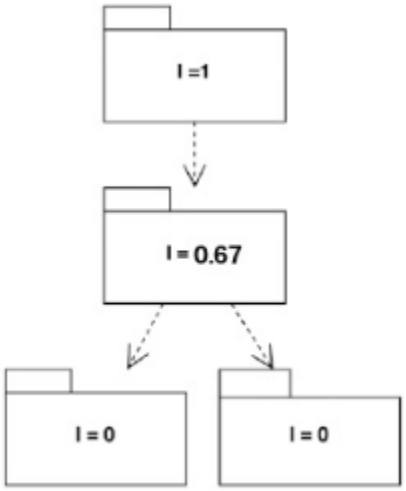


- Package Diagram w/ Cycles

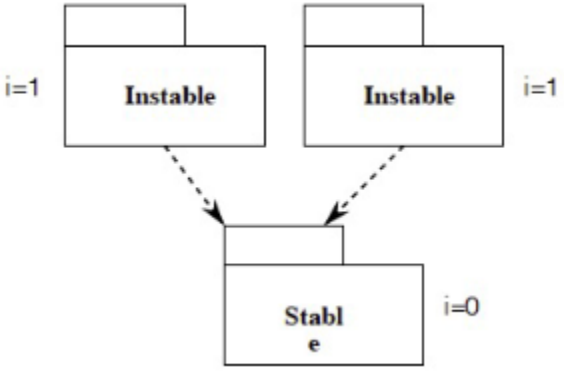


The Stable-Dependencies Principle (SDP)

- "*Depend in the direction of stability*"
- Parts of the system that change frequently should depend on parts that don't change very much
- Things that change a lot should have very few dependents
- Measuring Stability: The I Metric
    - I = C-out / (C-int + C-out)
        - C-in = # of classes *outside* a package that depend on a class *inside* the package
        - C-out: # of classes *outside* the package that any class *inside* the package depends on
    - I value is between 0 and 1
        - I = 0 : highly *stable*
        - I = 1 : highly *unstable*

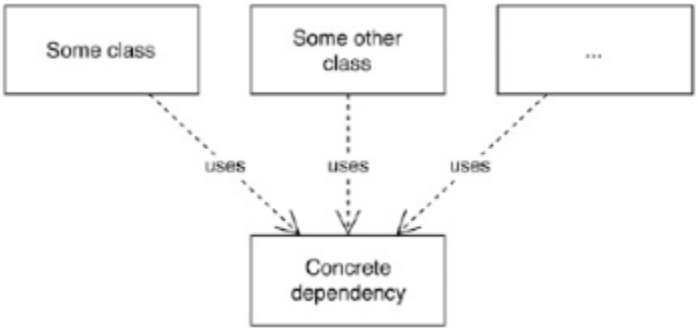


- Example

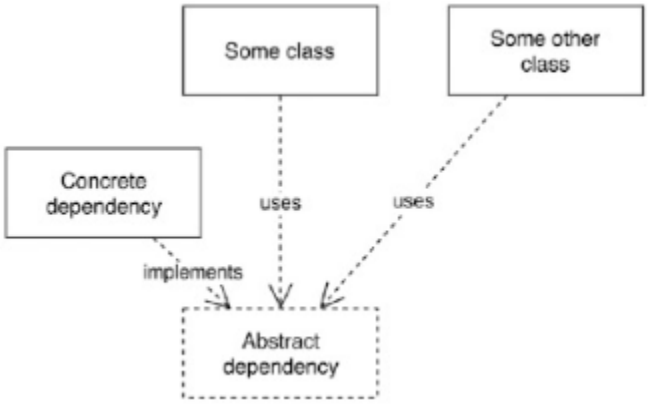


The Stable Abstractions Principle (SAP)

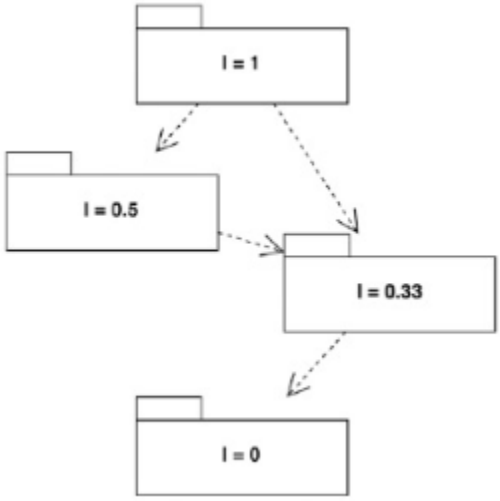
- "*Abstractness increases with stability*"
- Stable packages should be more abstract, containing classes and modules that can be extended
- Depending on concrete things...



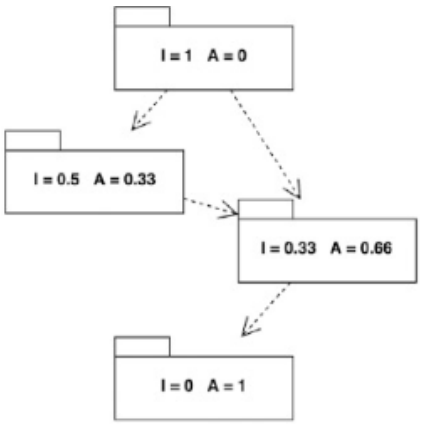
- Depending on abstract things…



- Measuring Abstractness: The A Metric
    - A = C-abstract / (C-concrete + C-abstract)
        - C-abstract: number of abstract classes and interfaces in a package
        - C-concrete: number of concrete classes
    - A value is between 0 and 1
        - A = 0: a highly *concrete* package
        - A = 1: a highly *abstract* package
- All dependencies go in the direction of stability…



- All dependencies go in the direction of abstractness



Introduction to Software Design Patterns

Gang of Four

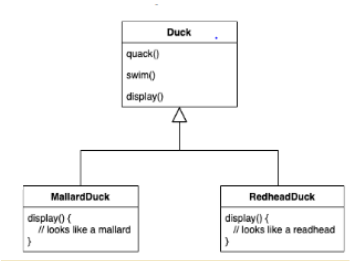
- 1977: Christopher Alexander introduces the idea of patterns: successful solutions to problems
- 1987: Ward Cunningham & Kent Beck leverage Alexander's idea in the context of an OO language
- 1987: Eric Gamma's dissertation on importance of patterns & how to capture them
- 1992: Jim Coplien's book Advanced C++ Programming Styles and Idioms

Patterns Catalogue

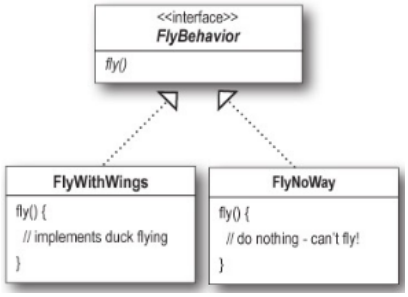
- Fundamental patterns
  - Delegation pattern
  - Interface pattern
  - Proxy pattern
  - …
- Creational patterns
  - Abstract factory pattern
  - Factory method pattern
  - Lazy initialization pattern
  - Singleton pattern
  - …
- Structural patterns
  - Adapter pattern
  - Bridge pattern
  - Decorator pattern
  - …
- Behavioral patterns
  - Chain of responsibility pattern
  - Iterator pattern
  - Observer pattern
  - Strategy pattern
  - Visitor pattern
  - …
- Concurrency patterns
  - Active object
  - Monitor object
  - Thread pool pattern
  - …

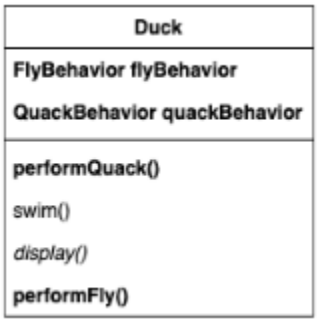
Chapter 1. Intro to Design Patterns: Welcome to Design Patterns

- *SimUDuck*: A "duck pond simulator" that can show a wide variety of duck species swimming and quacking
- Now we want our ducks to fly, but not all ducks can fly, so what do we do? We put the ability to fly in an interface.
- First design principle for this situation
  - *Identify the aspects of your application that var and separate them from what stays the same*
  - Take what varies and "encapsulate" it so it won't affect the rest of your code.
  - The result is fewer unintended consequences from code changes and more flexibility in your systems
- Second design principle for this situation



- Program to an interface, not an implementation, aka program to a supertype
- Programming to an implementation
  - Dog d = new Dog();
  - d.bark();
- Programming to an interface/supertype
  - Animal animal = new Dog();
  - animal.makeSound();
- Or even better…
  - a = getAnimal();
  - a.makeSound();
    - Sometimes, a concrete implementation should be determined at run time as opposed to hardcoding the implementation to be an Animal of type Dog
- Integrating the behavior w/ the Duck example…
  - Add two instance variables to the Duck class



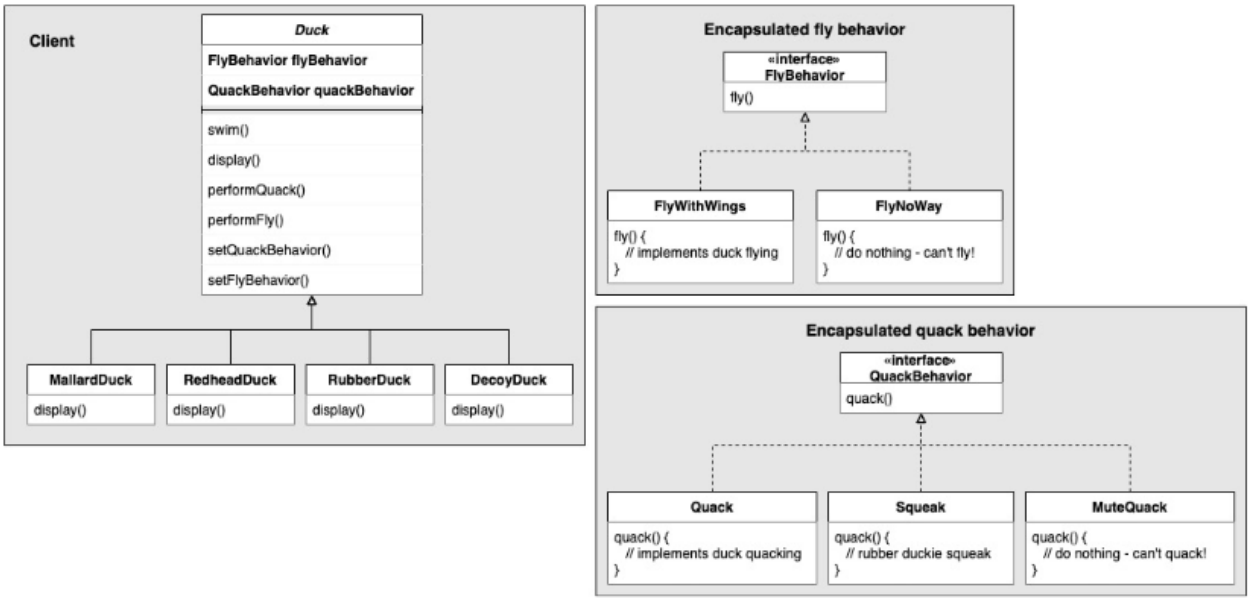


  - Now we implement performQuack()

```
public abstract class Duck {
    QuackBehavior quackBehavior;
    // more
    public void performQuack() {
        quackBehavior.quack();
    }
}
```

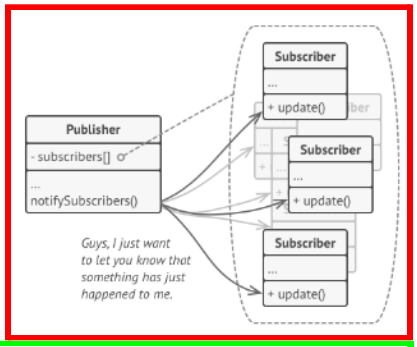
  - Setting the flyBehavior and quackBehavior instance variables

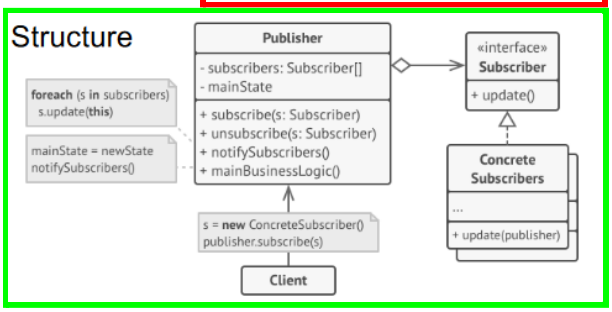
```
public class MallardDuck extends Duck {
    public MallardDuck() {
        quackBehavior = new QuackBehavior();
        flyBehavior = new FlyWithWings();
    }
    public void display() {
        System.out.print("I'm a real Mallard duck");
    }
}
```



- Third design principle for this situation
    - *Favor composition over inheritance*
    - Crating systems using compositions provides more flexibility
        - Lets you encapsulate a family of algorithms
        - Lets you change behavior at runtime
- The **Strategy Pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it
- Design patterns are important because they provide a shared vocabulary to software design (in addition to being really useful solutions to tricky problems)

Observer Pattern



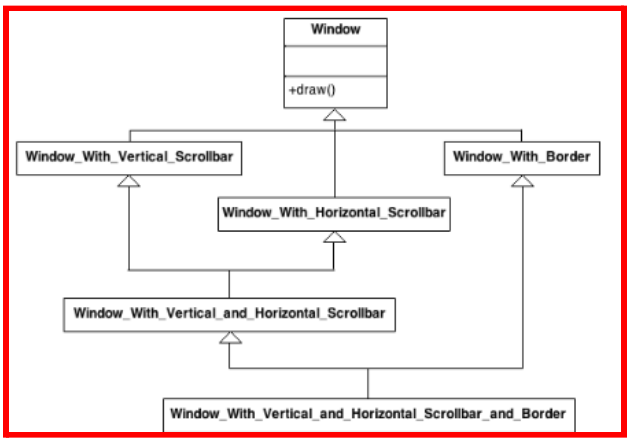


- Intent - Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing
- You wouldn't want to couple the publisher to all those subscriber classes
- It is crucial that all subscribers implement the same interface and that the publisher communicates with them via that interface
- Applicability…
    - When changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or dynamically
    - When some objects in your app must observe others, but only for a limited time or in specific cases
- Design principles at work…
    - Protected Variations - You can vary the Observers w/o changing the Subject
    - Program to an Interface, not an Implementation - Both Subject and Observer use interfaces
    - Favor Composition over Inheritance - Relationship between Observers & Subject is set up at runtime by composition

Decorator Pattern

Problem

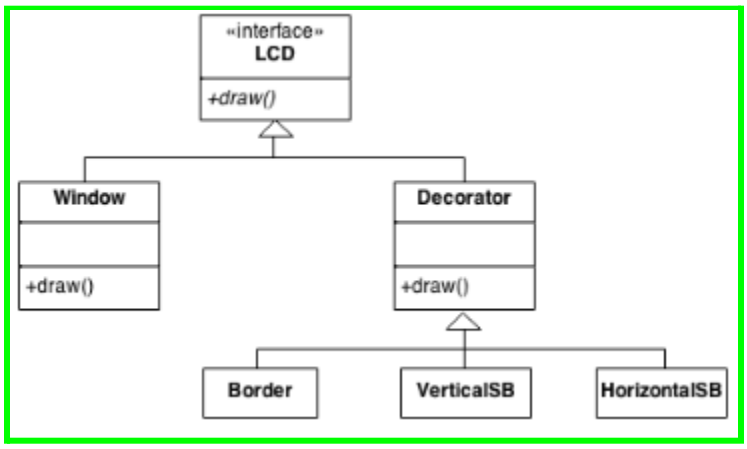
- Suppose you are working on UI and you want to add borders && scroll bars
- You can use inheritance in the following way…
    - Not really though; Java does not support multiple inheritance



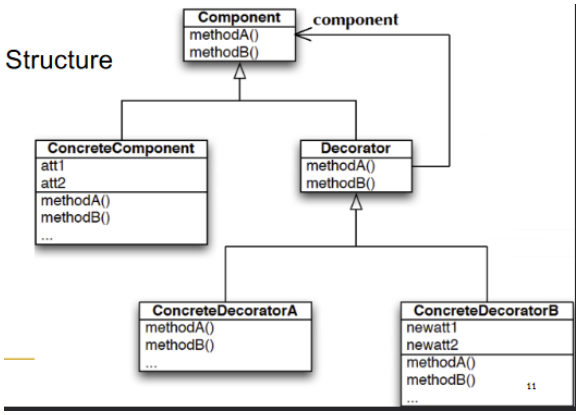
Solution

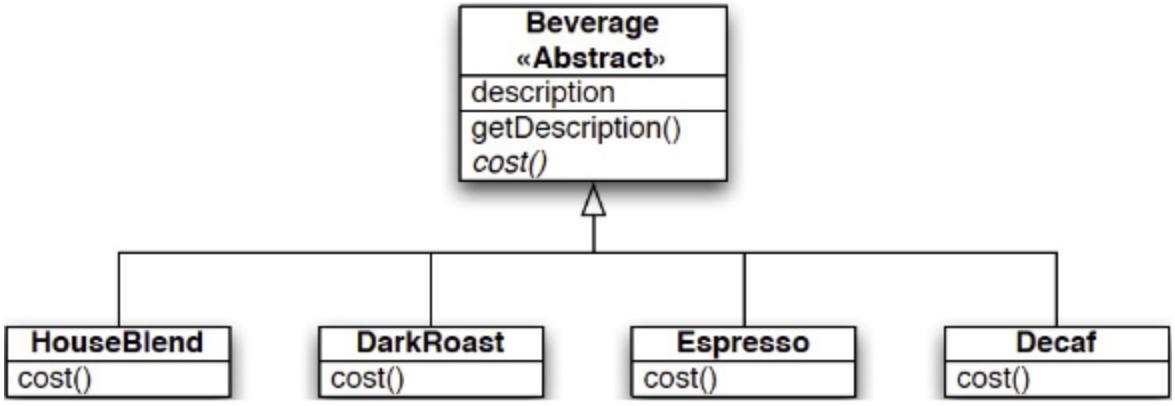
- Use "wrappers" to extend the base behaviours
- What's important to note here is that the Decorator also extends the LCD class; both Window && Decorator both implement the interface

```
Widget* aWidget = new BorderDecorator(
  new HorizontalScrollBarDecorator(
    new VerticalScrollBarDecorator(
      new Window( 80, 24 ))));
aWidget->draw();
```

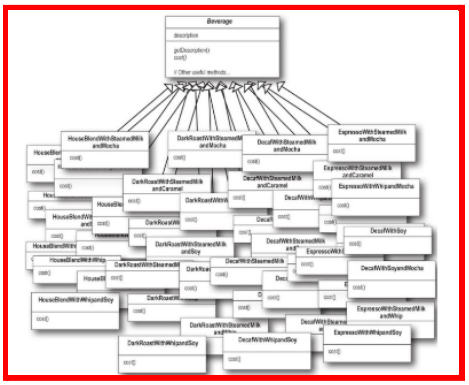


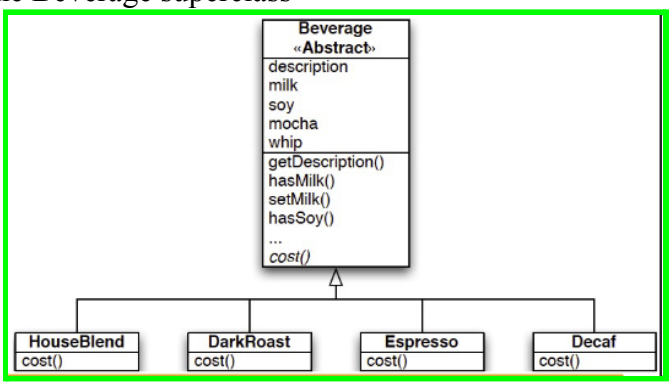
- ***Decorator*** is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors
- Wrapping is just a fancy way of saying "delegation" but with the added twist that the delegator and the delegate both implement the same interface
- Example w/ Beverages





- We have two different approaches here…
    - One subclass per combination of condiments
    - Add condiment handling to the Beverage superclass





Approach 1 Approach 2

- However, approach 2 still has some problems
    - For price changes, we must alter the existing code
    - New condiments : alter cost in the superclass
    - New beverages: some condiments may not be appropriate
- Decorator provides a way in which a class's runtime behavior can be extended w/o requiring modification to the class
- OCP - classes should be open for extension but closed for modification
    - Inheritance is one way, but composition & delegation are more flexible
    - GoF: "Decorator lets you attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality"

Factory Method Pattern

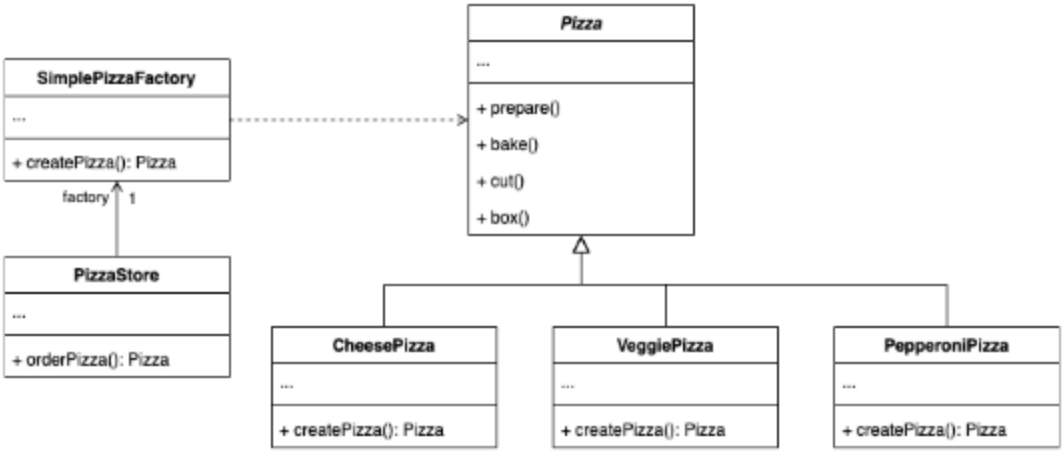
The Problem with “New”

```
Duck duck;

if (picnic) {
    duck = new MallardDuck();
} else if (hunting) {
    duck = new DecoyDuck();
} else if (inBathTub) {
    duck = new RubberDuck();
}
```

- Each time we invoke “new” to create a new object, we violate the “Code to an Interface” design principle
    - Duck duck = new MallardDuck();
- We have a set of concrete classes & we don’t know until runtime which one we need to instantiate…
- The problem here is that every time we create a new duck, we need to add the duck to this list, which can get pretty intensive if we add a lot of them
- A simple way to encapsulate this code is to put it in a separate class; delegate the object creation to a new class…

```
1  public class PizzaStore {
2
3      private SimplePizzaFactory factory;
4
5      public PizzaStore(SimplePizzaFactory factory)
6          this.factory = factory;
7      }
8
9      Pizza orderPizza(String type) {
10
11         Pizza pizza = factory.createPizza(type);
12
13         pizza.prepare();
14         pizza.bake();
15         pizza.cut();
16         pizza.box();
17
18         return pizza;
19     }
20
21 }
```

```
1  public class SimplePizzaFactory {
2      public Pizza createPizza(String type) {
3          if (type.equals("cheese")) {
4              return new CheesePizza();
5          } else if (type.equals("greek")) {
6              return new GreekPizza();
7          } else if (type.equals("pepperoni")) {
8              return new PepperoniPizza();
9          }
10     }
11 }
```



- But now what if we want to franchise the pizza store, so it extends to more parts of the country where different styles exist (California, New York, Chicago, etc.)
- To accomplish this, we make PizzaStore an abstract parent class and delegate the implementation of the *createPizza* method to the children of PizzaStore…

```java
public abstract class PizzaStore {

    Pizza orderPizza(String type) {

        Pizza pizza;

        pizza = createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

    abstract Pizza createPizza(String type);
}
```

```java
public class NYPizzaStore extends PizzaStore {
    public Pizza createPizza(String type) {
        if (type.equals("cheese")) {
            return new NYCheesePizza();
        } else if (type.equals("greek")) {
            return new NYGreekPizza();
        } else if (type.equals("pepperoni")) {
            return new NYPepperoniPizza();
        }
        return null;
    }
}
```

Singleton Pattern

- A creational design pattern used to ensure that only one instance of a particular class ever gets created and that there is just one (global) way to gain access to that instance
- To start, here is a class that has no restrictions on who can create it...

```
1  public class MyClass {
2      // attributes
3      // constructors
4      // methods
5  }
6
7  MyClass c = new MyClass();
```

Problem: Unlimited Instantiation

- As long as a client object "knows about" the name of the class MyClass, it can create instances of MyClass
- This is because the constructor is public. We can stop unauthorized creation of MyClass instances by making the constructor private

```
1  public class MyClass {
2      private MyClass() {}
3      // attributes
4      // constructors
5      // methods
6  }
```

- Now, it's impossible to instantiate an object of MyClass
    - MyClass c = new MyClass(); is impossible

Problem: No Point of Access!

- Now that the constructor is private, no class can gain access to instances of MyClass
    - But our requirements were that there be at least one way to access an instance of my class
- We need a method to return an instance of MyClass
    - But since there is no way to get access to an instance of myClass, the method can NOT be an instance method
        - Therefore, it needs to be a **class/static** method

```
1  public class MyClass {
2      private MyClass() {}
3      public static MyClass getInstance() {
4          return new MyClass();
5      }
6      // attributes
7      // constructors
8      // methods
9  }
```

Problem: Back to Unlimited Instantiation

- Now we are back to where we started! Instead of saying…
  MyClass c = new MyClass();
  We can just say…
  MyClass c = MyClass.getInstance();
- We need to ensure only one instance is ever created
  - Need a static variable to store that instance
    - No instance variables are available in static methods

```
public class MyClass {
    private static MyClass myClass;
    private MyClass() {}
    public static MyClass getInstance() {
        return myClass;
    }
    // attributes
    // constructors
    // methods
}
```

Problem: No Instance!

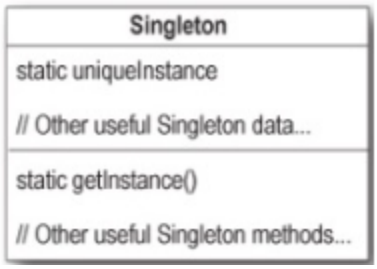
- Now, the getInstance() method returns null each time it is called
  - Need to check instance variable to see if it's null
    - If so, create instance
    - Otherwise, return the single instance

```
public class MyClass {
    private static MyClass myClass;
    private MyClass() {}
    public static MyClass getInstance() {
        if (myClass == null) {
            myClass = new MyClass();
        }
        return myClass;
    }
    // attributes
    // constructors
    // methods
}
```

Singleton Pattern

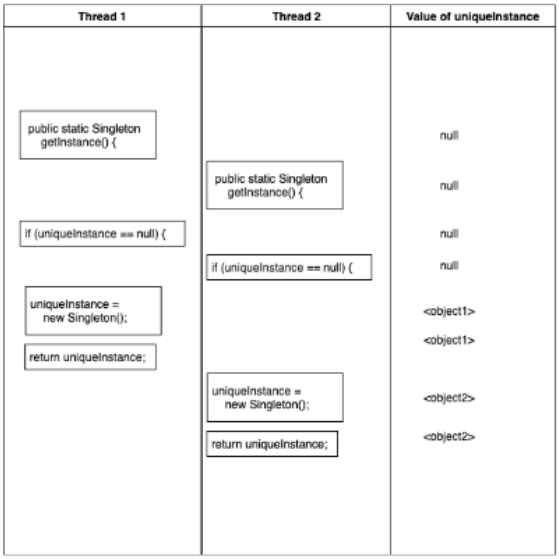
```
public class Singleton {

    private static Singleton uniqueInstance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
}
```

- Singleton involves only a single class (not typically called Singleton). That class is a full-fledged class with other attributes and methods
- The class has a static variable that points at a single instance of the class
- The class has a private constructor (to prevent other code from instantiating the class) and a static method that provides access to the single instance



Thread Safe?

- The Java code shown is not thread safe
  - It's possible for two threads to attempt to create the singleton for the first time simultaneously
  - If both threads check to see if the static variable is empty at the same time, they will both proceed to creating an instance and you will end up with two instances of the singleton object (not good!)



Dealing with Multithreading

- Make getInstance() a synchronized method

```
1  public class Singleton {
2
3      private static Singleton uniqueInstance;
4
5      private Singleton() {}
6
7      public static synchronized Singleton getInstance() {
8          if (uniqueInstance == null) {
9              uniqueInstance = new Singleton();
10         }
11         return uniqueInstance;
12     }
13 }
```

- Or rather, you can move to an eagerly created instance rather than a lazily created one

```java
public class Singleton {

    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

- Or rather, use “double-checked locking” to reduce the use of synchronization in getInstance()...

```java
public class Singleton {

    private volatile static Singleton uniqueInstance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            synchronized (Singleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}
```

Iterator Pattern

- Collections are ubiquitous in programming
    - Arrays, Stacks, Queues, Lists, Hash tables, Trees, etc.
- You don't want to expose the details of the collections that your class uses
    - May increase coupling of your system
    - Clients are tied to your class and the class of the collections

Example: The Merging of Two Diners

- Two restaurants are merging
- Lou's breakfast place, which uses an ArrayList to store menu items
- Mel's diner, which uses an array to store their menu items
- When they merge, so do their menus, but neither person wants to change their implementations as they have too much existing code dependent on these data structures

Pros and Cons

- Use of ArrayList
    - Easy to add/remove
    - No management of the "size"
    - Can use a lot of memory if menu items are allowed to grow unchecked
- Use of plain array
    - Fixed size of array provides control over memory usage
    - Need to add code that manages the bounds

Implementation Details Exposed

- Both classes reveal the type of collection via the getMenuItems() method
- All client code is forced to bind to the menu class and the collection class being used
- If you needed to change the internal collection, you wouldn't be able to do it without impacting all your clients

Example Client: Waitress

- Implement a client of these two menus with the following specification
    - printMenu(): print all menu items from both menus
    - printBreakfastMenu(): print all breakfast items
    - Etc.

```
...

PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
ArrayList<MenuItem> breakfastItems = pancakeHouseMenu.getMenuItems();

DinerMenu dinerMenu = new DinerMenu();
MenuItem[] lunchItems = dinerMenu.getMenuItems();

```

```
...

for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = breakfastItems.get(i);
    System.out.println(menuItem.getName() + " ");
}

for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
}

...
```

Issues with Above

- The current approach has the following design-related problems
    - Coding to an implementation rather than an interface
    - Vulnerable to changes in the collections used
    - Waitress knows the internal details of each Menu class
    - We have (in essence) duplicated code, one distinct loop per menu
- We can solve this problem with the design principle “encapsulate what varies”
    - In this case, the details of the data structures and iteration over them

Encapsulating the Iteration

- To iterate through the breakfast items, we use the size() and get() methods on ArrayList

```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = breakfastItems.get(i);
}
```

- To iterate through the lunch items, we use the Array length field and the array subscript notation on the MenuItem Array:

```
for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
}
```

- Create an Iterator that encapsulates the way we iterate through a collection of objects

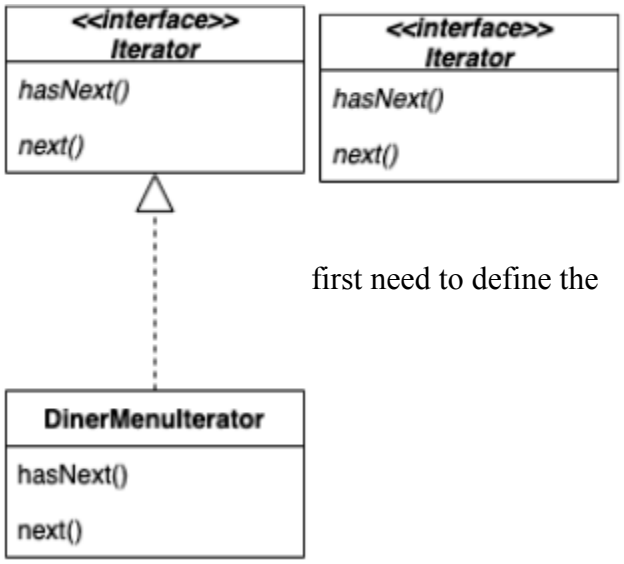
```
Iterator iterator = lunchItems.createIterator();

while (iterator.hasNext()) {
    MenuItem menuItem = iterator.next();
}
```

- Similarly, on the Array...

```
Iterator iterator = lunchItems.createIterator();

while (iterator.hasNext()) {
    MenuItem menuItem = iterator.next();
}
```

Meet the Iterator Pattern

- Iterator is a behavioral design pattern that provides a common interface for traversing the items of an aggregate, allowing you to use polymorphism when writing code that makes use of the items of the aggregate
- Iterator interface
- Implementation of the Iterator for the Array used in the DinerMenu



Adding an Iterator to DinerMenu

- To add an Iterator to the DinerMenu we first need to define the Iterator Interface

```
public interface Iterator {
    boolean hasNext();
    Object next();
}
```

```
public class DinerMenuIterator implements Iterator {
    MenuItem[] items;
    int position = 0;

    public DinerMenuIterator(MenuItem[] items) {
        this.items = items;
    }

    @Override
    public boolean hasNext() {
        if (position >= items.length || items[position] ==  null) {
            return false;
        } else {
            return true;
        }
    }

    @Override
    public MenuItem next() {
        MenuItem menuItem = items[position];
        position = position + 1;
        return menuItem;
    }
}
```

Reworking the Diner Menu with Iterator

```
1  public class DinerMenu {
2      static final int MAX_ITEMS = 6;
3      int numberOfItems = 0;
4      MenuItem[] menuItems;
5
6      public DinerMenu() { ...
19     }
20
21     public void addItem(String name, String description, boolean vegetarian, double p
29     }
30
31     public Iterator createIterator() {
32         return new DinerMenuIterator(menuItems);
33     }
34
35 }
```

Fixing up the Waitress Code

```
public class Waitress {
    PancakeHouseMenu pancakeHouseMenu;
    DinerMenu dinerMenu;

    public Waitress(PancakeHouseMenu pancakeHouseMenu, DinerMenu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }

    public void printMenu() {
        Iterator pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator dinerIterator = dinerMenu.createIterator();

        System.out.println("MENU\n----\nBREAKFAST");
        printMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
    }

    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = (MenuItem) iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }
}
```

Iterator Pattern Defined

- Behavioral design pattern
- Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- Allows traversal of the elements of an aggregate without exposing the underlying implementation
- Places the task of traversal on the iterator object, not on the aggregate

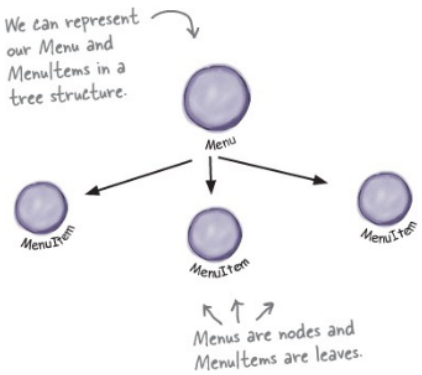
Return of Single Responsibility

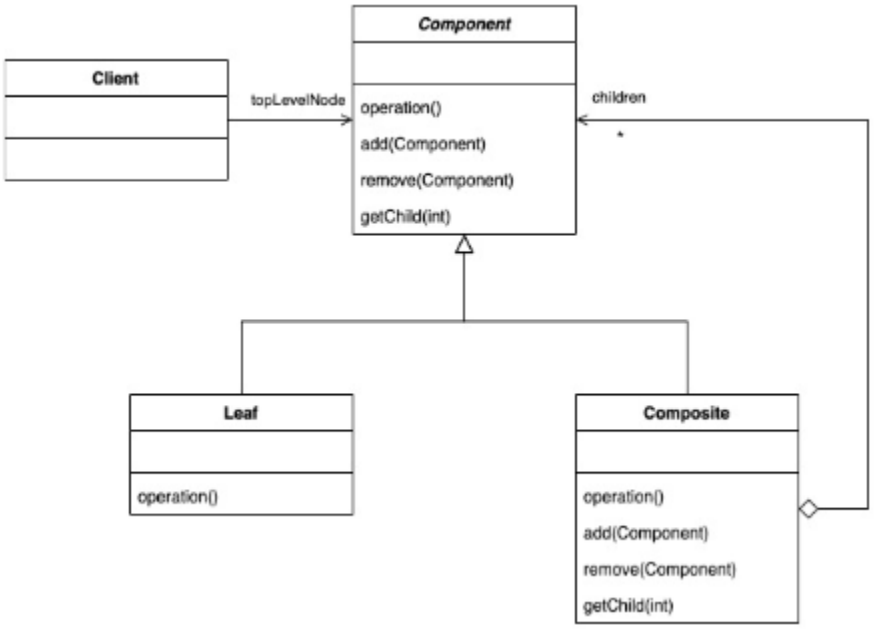
- “A class should have only one reason to change”
- SRP is behind the idea that collections should not implement traverals

- Collections focus on storing members and providing access to them
- An iterator focuses on looping over the members of a collection

Composite Pattern

- The **composite pattern** is a structural design pattern that allows us to build structures of objects in the form of trees that contain both objects and other composites
- Simple example: grouping objects in a vector drawing tool
    - You can create an individual shape and apply operations to it: move(), scale(), rotate(), etc.
    - You can create a group of objects and apply the **same** operations to it: move(), scale(), rotate(), etc.





- Using the menu example let's implement the composite pattern
- Initial steps are easy
    - MenuComponent is an abstract class that implements all methods with the same line of code
        - throw new UnsupportedOperationException
        - This is a run-time exception that indicates that the object doesn't respond to this method
    - MenuItem is exactly the same as before except now it extends MenuComponent

```
public class Menu extends MenuComponent {
    ArrayList<MenuComponent> menuComponents = new ArrayList<MenuComponent>();
    String name;
    String description;

    public Menu(String name, String description) {
        this.name = name;
        this.description = description;
    }

    public void add(MenuComponent menuComponent) {
        menuComponents.add(menuComponent);
    }

    public void remove(MenuComponent menuComponent) {
        menuComponents.remove(menuComponent);
    }

    public MenuComponent getChild(int i) {
        return (MenuComponent)menuComponents.get(i);
    }

    public String getName() { return name; }

    public String getDescription() { return description; }

    public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("---------------------");
    }
}
```

Iterator pattern!

```
public class Menu extends MenuComponent {
    ArrayList<MenuComponent> menuComponents = new ArrayList<MenuComponent>();
    String name;
    String description;

    // constructor code here

    // other methods here

    public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("---------------------");

        Iterator<MenuComponent> iterator = menuComponents.iterator();
        while (iterator.hasNext()) {
            MenuComponent menuComponent =
                (MenuComponent)iterator.next();
            menuComponent.print();
        }
    }
}
```

- Suppose we want to iterate over every item of the menu and pull out vegetarian items only
- We want to use Iterator with a Composite

Command Pattern - Final Lecture

Final Exam

- Take home & team based w/ team for milestone project
- 3 Parts
    - MC, ~30 mins
    - Diagramming: Sequence Diagram and DCD
    - Short Answer, 4-5 sentences

Exam 3

- MC && Diagramming
    - Diagram: DCD
- Command Pattern will not be on Exam 3

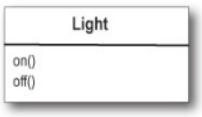
Onto the Command Pattern

- "*Allows you to decouple the requester of the action from the object that performs the action*"
- "A command object encapsulates a request to do something
- The Command interface just does one thing - executes a command

- Command interface

```
1  public interface Command {
2      void execute();
3  }
```

- Vendor class



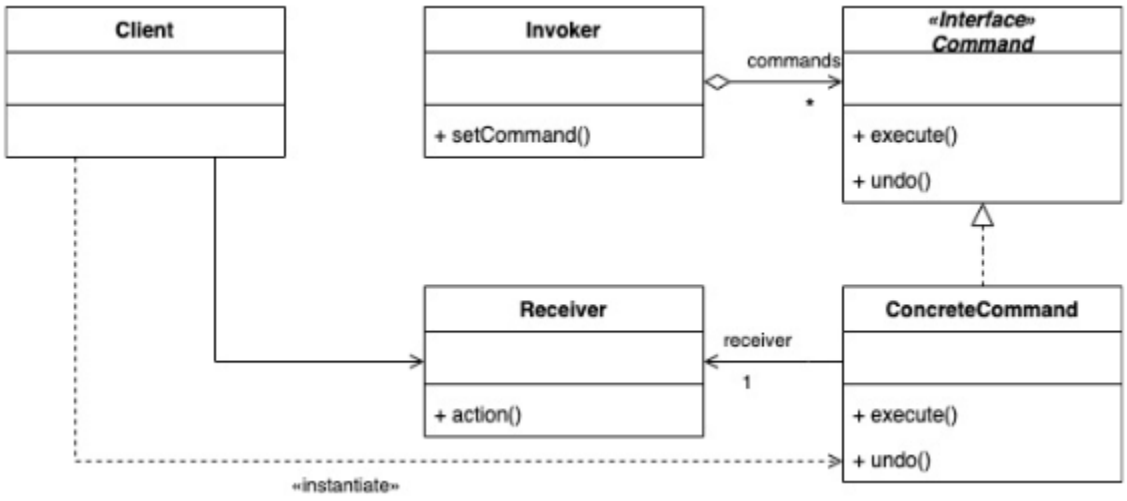

```
1  public class LightOnCommand implements Command {
2      Light light;
3
4      public LightOnCommand(Light light) {
5          this.light = light;
6      }
7
8      public void execute() {
9          light.on;
10     }
11 }
```

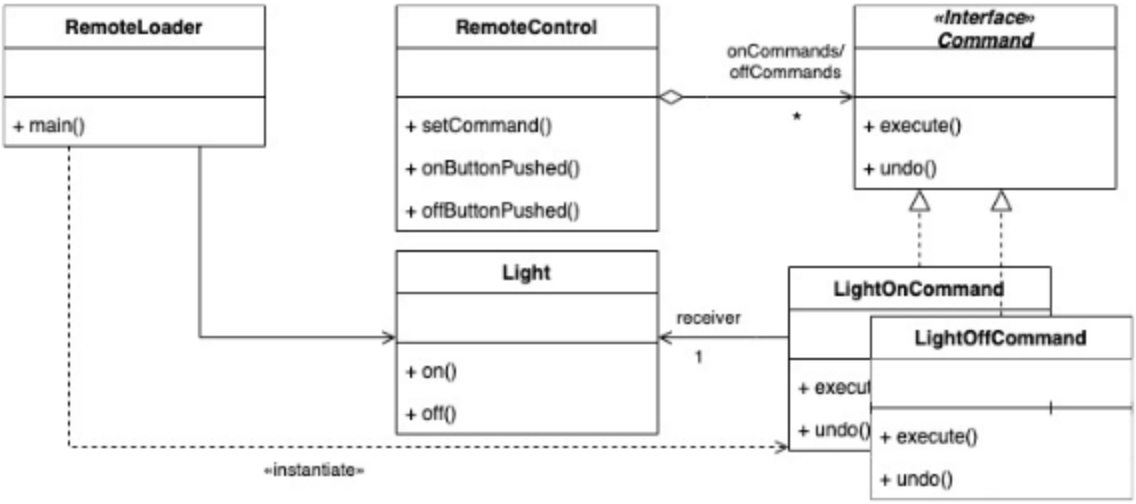
Note: there should be parenthesis on the light.on call in execute()

```
1  public class SimpleRemoteControl {
2      Command slot;
3      public SimpleRemoteControl() {}
4
5      public void setCommand(Command command) {
6          slot = command;
7      }
8      public void buttonWasPressed() {
9          slot.execute();
10     }
11 }
```

```java
public class RemoteControlTest {
    public static void main(String[] args) {
        SimpleRemoteControl remote = new SimpleRemoteControl();
        Light light = new Light();
        LightOnCommand lightOn = new LightOnCommand(light);

        remote.setCommand(lightOn);
        remote.buttonWasPressed();
    }
}
```

- The command pattern is a *behavioral* design pattern
- GoF (Gang of Four) **Intent**: "Encapsulates a request as an object,thereby letting you parameterize other objects w/ different requests, queue or log requests, and support undoable operations"
- Participants include…
    - Client (RemoteControlTest) - creates command and associates command w/ receiver
    - Receiver (Light, TV, etc.) - knows how to perform the work
    - Concrete Command (LightOnCommand) - implementation of Command interface
    - Command Interface - defines interface for all commands
    - Invoker (Remote Control) - holds reference to a command and calls execute() method





Yet another example of the Command pattern…

```java
public class StereoOnWithCDCommand implements Command {
    Stereo stereo;

    public StereoOnWithCDCommand(Stereo stereo) {
        this.stereo = stereo;
    }

    public void execute() {
        stereo.on();
        stereo.setCD();
        stereo.setVolume(11);
    }
}
```

MacroCommands

```java
public class MacroCommand implements Command {
    Command[] commands;

    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }

    public void execute() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].execute();
        }
    }
}
```

- The Command Pattern decouples an object making a request from the one that knows how to perform it
- A Command object is at the center of this decoupling and encapsulates a receiver with an action
- Commands may support Undo to restore the object to its previous state

- MacroCommands are a simple extension of Command that allow multiple commands to be invoked
- *Command* can be used to convert any operation into an object. The conversion lets you defer execution of the operation, queue it, store the history of commands, etc.