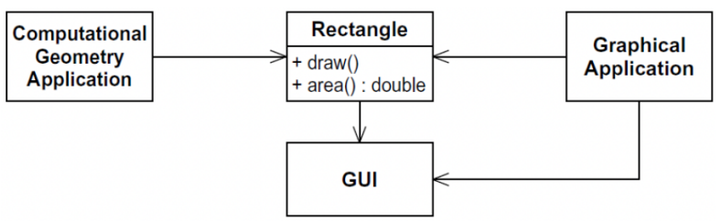
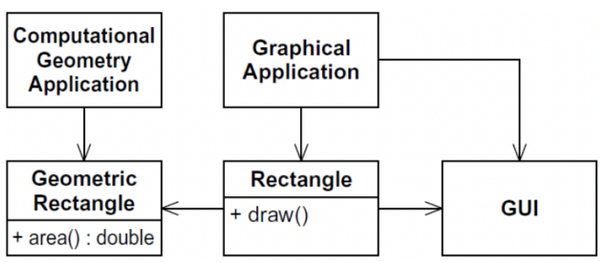
Lecture 7 - SOLID Design

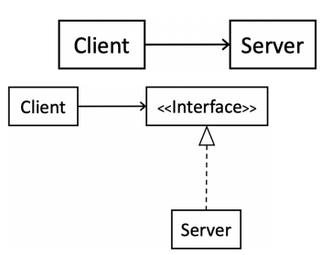
- What is SOLID?
  - **S**ingle Responsibility Principle
  - **O**pen/Closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface Segregation Principle
  - **D**ependency Inversion Principle
- Single Responsibility Principle (SRP)
  A class should have only one reason to change
  - If you can think of more than one motive for changing a class, then that class has more than one responsibility
  - If a class has more than one responsibility, then the responsibilities become coupled
  - Violating the SRP



  - Conforming to the SRP

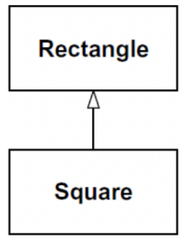


- The Open/Closed Principle (OCP)
  Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.
  - When a single change to a program results in a cascade of changes to dependent modules, the design smells of rigidity.
    - If the Open/Closed principle is applied well, then further changes of that kind are achieved by adding new code, not by changing old code that already works.
  - In Java, it is possible to create abstractions that are fixed and yet represent an unbounded group of possible behaviors
    - The abstractions are abstract base classes, and the unbounded group of possible behaviors is represented by all the possible derivative classes.
  - Violating the OCP
    - Both classes are concrete
    - The **Client** uses the **Server** class
  - Conforming to the OCP



- The Liskov Substitution Principle (LSP)
  Subtypes must be substitutable for their base types.
  - Formally: Let $\Phi(x)$ be a property provable about objects x of type T. Then $\Phi(y)$ should be true for objects y of type S where S is a subtype of T.
  - Counter-example: "If it looks like a duck, quacks like a duck, but needs batteries – you probably have the wrong abstraction"

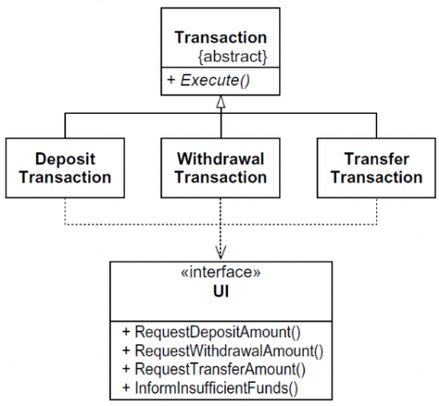
- Violating the LSP
  Issues
  - Inheriting **height** and **width**
  - Overriding **setHeight** and **setWidth**
  - Conflicting assumptions. For example:



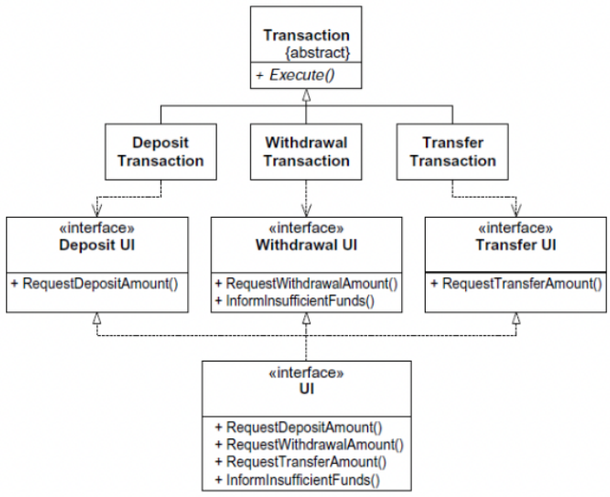

```
void testRectangleArea(Rectangle r){
        r.setWidth(5);
        r.setHeight(4);
        assertEquals(r.computeArea(), 20);
}
```

- Implication: A model, viewed in isolation, cannot be meaningfully validated.
  - The validity of a model can only be expressed in terms of its clients.
  - One must view the design in terms of the reasonable assumptions made by the users of that design.

- The Interface Segregation Principle (ISP)
  Clients should not be forced to depend on methods that they do not use.
  - This principle deals with classes whose interfaces are not cohesive. That is, the interfaces of the class can be broken up into groups of methods where each group serves a different set of clients.
  - When clients are forced to depend on methods that they don't use, then those clients are subject to changes to those methods.
  - Violating the ISP



  - Conforming to the ISP



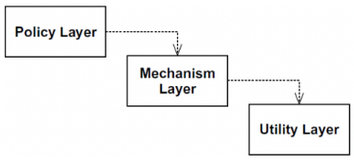
- The Dependency-Inversion Principle (DIP)

  A. High-level modules should not depend on low-level modules. Both should depend on abstractions.

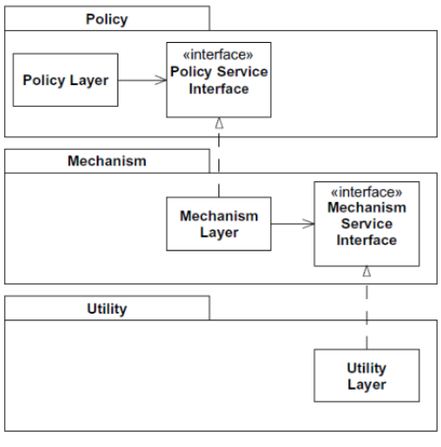
  B. Abstractions should not depend on details. Details should depend on abstractions.

  - The modules that contain the high-level business rules should take precedence over, and be independent of, the modules that contain the implementation details.
  - When high-level modules depend on low-level modules, it becomes very difficult to reuse those high-level modules in different contexts.

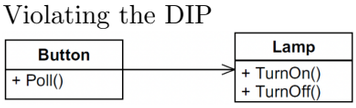
Naïve Layering



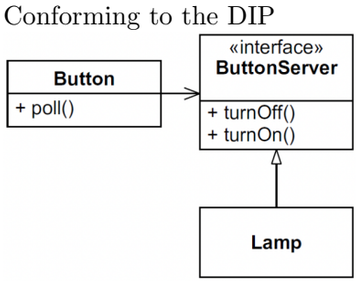
Inverted Layers



- Violating the DIP



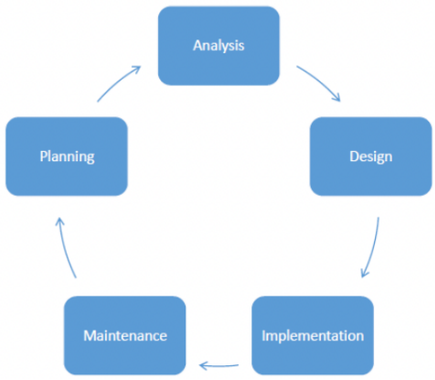
- Conforming to the DIP



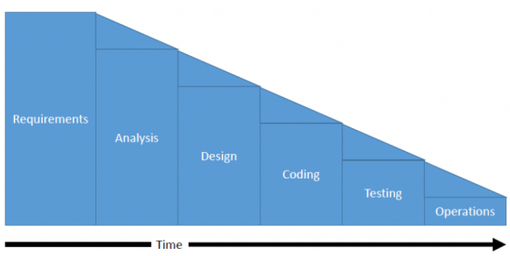
- Design Smells
  - Symptoms of poor design
  - Often caused by the violation of one or more of the design principles
    - For example, the smell of *Rigidity* is often a result of insufficient attention to OCP.
  - These symptoms include:
    1. Rigidity – The design is hard to change.
    2. Fragility – The design is easy to break.
    3. Immobility – The design is hard to reuse.
    4. Viscosity – It is hard to do the right thing.
    5. Needless Complexity – Overdesign.
    6. Needless Repetition – Mouse abuse.
    7. Opacity – Disorganized expression.

Software Development Life Cycle

- Software Development Life Cycle (SDLC)
  - **Planning** – develop a plan for creating the concept or evolution of the concept
  - **Analysis** – analyze the needs of those using the system. Create detailed requirements
  - **Design** – Translate the detailed requirements into detailed design work
  - **Implementation** – Complete the work of developing and testing the system
  - **Maintenance** – Complete any required maintenance to keep the system running



- Different SDLC implementations
  - Rigid timeline / budget (Waterfall)
  - Rick Adverse (Spiral)
  - Quality Deliverables / Less management (Agile)
- Waterfall
  - A sequential (non-iterative) model
  - Involves a large amount of upfront work, in an attempt to reduce the amount of work done in later phases of the project



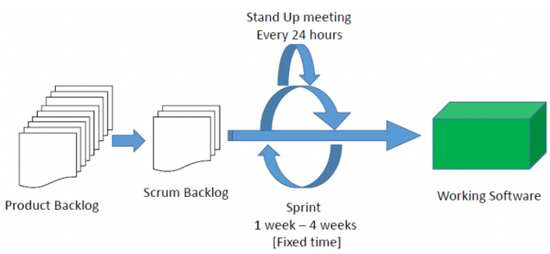
- Spiral
  - Risk-driven model
  - More time is spent on a given phase based on the amount of risk that phase poses for the project

- Agile
  - Issues with Waterfall
    - Inappropriate when requirements change frequently
    - Time gets squeezed the further into the process you get
  - Agile Methodologies
    - Extreme Programming (XP)
    - Scrum
    - Test-driven Develpoment (TDD)
    - Feature-driven Development (FDD)
    - Etc.
- Agile Manifesto
  "We are uncovering better ways of developing software by doing it and helping others do it. Through this work, we have come to value:
  Individuals and interactions *over processes and tools*
  Working software *over comprehensive documentation*
  Customer collaboration *over contract negotiation*
  Reponding to change *over following a plan*
  That is, while there is value in the items on the right, we value the items on the left more."

- Agile vs. Waterfall

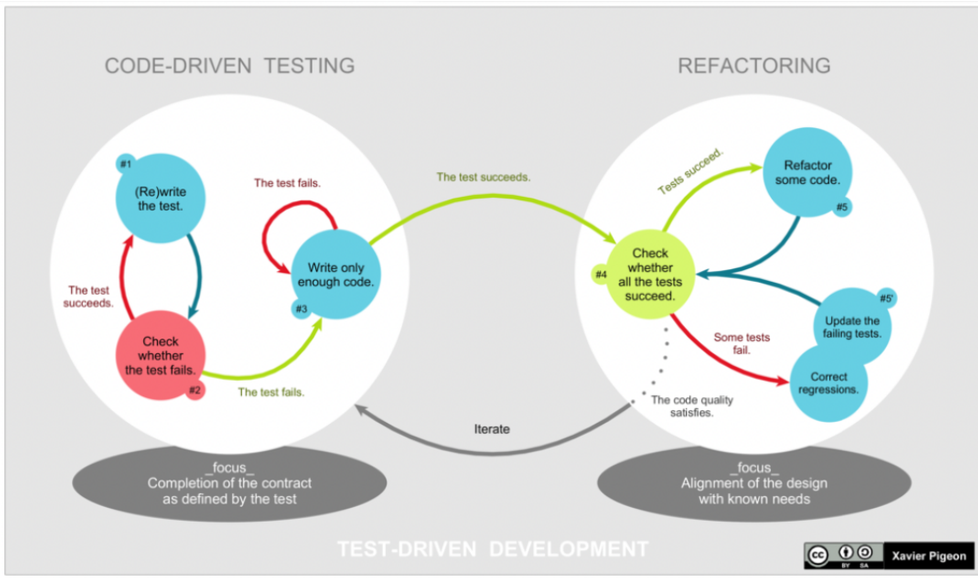
	Agile	Waterfall
Iterative?	Yes	No
Late changes?	Yes	No / $$$
Fixed timeline?	No*	Yes
Fixed cost?	No*	Yes*
Volume of meetings	Consistent	Heavy up front, reduced middle, heavy end
Release frequency	Every sprint	Once per project
Business Involvement	Heavy throughout	Heavy early, and at very end
Cost to fix mistakes	Low	High

- eXtreme Programming (XP)
  - One of the most rigorous forms of Agile
  - Involves building a series of feedback loops, which are used to help guide when change can occur and allow for changes to be quickly integrated into the plan for development
  - Built on the idea that you can reduce the cost of developing software, and build better software, by having goals
  - XP requires that everything that can be unit tested is unit tested, that everyone works in pairs, and that these pairs change frequently
- Code Review
  - Although pair programming has gone out of vogue along with XP, it is important to note a practice that has become common place that was born from this idea – Code Review.
  - A code review is a session in which you **must sit down with another developer** from the team and walk them through your implementation line-by-line in order to get advice and feedback.
  - This process has been shown to lead to better code, through finding bugs earlier, and an increased amount of collaboration on difficult concepts.
- Scrum
  - Scrum is currently one of the most widely used methodologies of software development



- Scrum - Roles
  - Product Owner
    - Responsible for delivering requirements and accepting demos
    - Involved in planning session
  - Scrum Master
    - Responsible for removing impediments
  - Team members
    - No one has a fixed role other than the scrum master and product owner
    - Everyone takes on tasks, and completes them based on what they are most comfortable with
- Scrum - Sprint
  - The sprint is a fixed time to deliver a working set of features, that are reviewed in a demonstration to the product owner
  - Tasks in Scrum are broken into "User Stories"
  - In a sprint, a team agrees at the beginning to take on a certain number of user stories
  - Sprints are usually between 1 and 4 weeks in length
  - At the end of each sprint, teams hold a "retrospective" which is a meeting where the past sprint is discussed, and chances for improvement for the next sprint are raised
- Scrum - User Stories
  - User stories are similar to requirements. They are written in the following format:
    As a {ACTOR/OBJECT} I want to {ACTION} so that {RESULT}

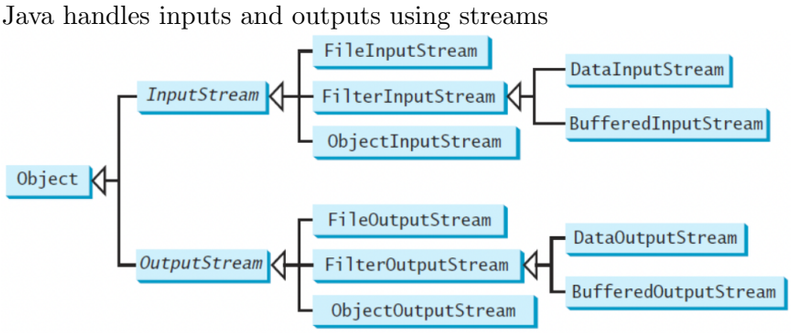
- Scrum - Planning Poker
  - In scrum, we do not assign time to tasks, but assign arbitrary points. This is a form of estimation that helps gauge how much work something will take to complete.
  - Planning poker takes a set of pre-determined numbers (usually: 1, 2, 3, 5, 8, etc.) and gets you to estimate how much work something will be relative to a known task.
  - After discussing the story at hand , everyone selects a card. Then, the cards are turned over simultaneously. Usually time is given for those who had the lowest and highest numbers to state their case
  - The process is repeated until everyone ends up at the same number.
- Scrum - Planning Session
  - Planning sessions happen at the start of each sprint.
  - They usually take a few hours. During this time, the team decides how much work it will take on, and discusses any major technical challenges they expect to face.
  - Usually, Product Owners are available for at least a portion of this meeting, to help with prioritization. They are only there to assist in this regard, and not to dictate what the team will complete.
- Scrum - The Standup Meeting
  - Happens EVERY SINGLE day that you are working
  - The goal is to make sure people are doing alright
  - Shouldn't be longer than 15 minutes
  - Answer three questions:
    1. What did I finish since the last standup?
    2. What am I going to finish by the next standup?
    3. What is stopping me / what impediments am I facing?
- Scrum - Working Agreement
  - A series of statements that everyone on the team agrees to about how the team will work
  - Things in working agreements may include:
    * The standup will occur at 1:00 pm every day, and last 15 minutes
    * We will not speak during the standup, unless it is our turn to speak
    * Our meetings will take place in the lobby of the IC building
    * All code must be peer-reviewed
    * We will submit all code 24 hours prior to the due date
- Scrum - Definition of Done
  - A formal agreement of when work is considered complete
  - For example, a story can be marked as done when:
    * It has been fully unit tested
    * It successfully integrated with the rest of the code
    * It has been peer reviewed
    * It is fully commented
    * Etc.
  - It is important that team comes to an agreement on this definition before they start work.
- Test Driven Development (TDD)
  - TDD is a way to develop software that revolves around writing test cases.
  - The basic concept is to write the unit tests needed to be passed for a feature to be considered working. You then code to the unit tests –writing the minimum amount for the tests to succeed.
  - Once working, you review and refactor. Then move on to the next set of tests.



- Feature Driven Development (FDD)
  - Based on the idea of building a focused model for the project, and the iterating on the features needed.
  - Splits development into 5 major pieces:
    1. Develop overall model
    2. Build feature list
    3. Plan by feature
    4. Design by feature
    5. Build by feature

I/O and Regular Expressions

- Input and Output (I/O)
  - Input sources include:
    - Keyboard
    - File
    - Network
  - Output destinations include:
    - Console
    - File
    - Network
- Input and Output Streams
  - Java handles inputs and outputs using streams



- Standard I/O
  - **System.in**
    - Object of type **InputStream**
    - Typically refers to the keyboard
    - Reading data could be done using the **Scanner** class. Its methods include:
      - **String next()**
      - **int nextInt()**
      - **String nextLine()**
      - **double nextDouble()**
  - **System.out**
    - Object of type **PrintStream**
    - Typically refers to the console
- The **File** class
  - Contains methods for obtaining the properties of a file/directory and for renaming and deleting a file/directory
  - Files could be specified using absolute or relative names
  - Constructing a **File** instance does not create a file on the machine
  - Methods include:
    - **boolean createNewFile()**
    - **boolean delete()**
    - **boolean exists()**
    - **boolean isDirectory()**
    - **File [] listFiles()**
- File I/O
  - Reading could be done using the **Scanner** class
    - e.g. **Scanner input = new Scanner(new File(filename));**
  - Writing could be done using the **FileWrite** class
    - e.g. **FileWriter output = new FileWriter(filename, append);**

- Regular Expressions
  - A regular expression (abbreviated regex) is a string that describes a pattern for matching a set of strings.
  - Regular expressions provide a simple and effective way to validate user input
    - e.g. phone numbers
  - Java supports regular expressions using the **java.util.regex** package
  - The **Pattern** class can be used to define the pattern
    - The **compile** method takes a string representing the regular expression as an argument and compiles it into a pattern
  - The **Matcher** class can be used to search for the pattern. Its methods include:
    - **boolean find()**
    - **boolean matches()**
  - Example

```
Pattern pattern = Pattern.compile("H.*d");
Matcher matcher = pattern.matcher("Hello World");
System.out.println(matcher.matches());
```

- Commonly Used Regular Expressions

Regular Expression	*Matches*	*Example*
`.`	any single character	`Java` matches `J..a`
`(ab|cd)`	`ab` or `cd`	`ten` matches `t(en|im)`
`[abc]`	`a`, `b`, or `c`	`Java` matches `Ja[uvwx]a`
`[^abc]`	any character except `a`, `b`, or `c`	`Java` matched `Ja[^ars]a`
`[a-z]`	`a` through `z`	`Java` matches `[A-M]av[a-d]`
`[^a-z]`	any character except `a` through `z`	`Java` matches `J]av[^b-d]`
`[a-e[m-p]]`	`a` through `e` or `m` through `p`	`Java` matches `[A-G[I-M]]av[a-d]`
`[a-e&&[c-p]]`	intersection of `a-e` with `c-p`	`Java` matches `[A-P&&[I-M]]av[a-d]`
`\d`	a digit, same as `[0-9]`	`Java2` matches "`Java[\\d]`"
`\D`	a non-digit	`$Java` matches "`[\\D][\\D]ava`"
`\w`	a word character	`Java1` matches "`[\\w]ava[\\w]`"
`\W`	a non-word character	`$Java` matches "`[\\W][\\w]ava`"
`\s`	a whitespace character	"`Java 2`" matches "`Java\\s2`"
`\S`	a non-whitespace character	`Java` matches "`[\\S]ava`"
p`*`	zero or more occurrences of pattern *p*	`aaaabb` matches "`a*bb`" `ababab` matches "`(ab)*`"
p`+`	one or more occurrences of pattern *p*	`a` matches "`a+b*`" `able` matches "`(ab)+.*`"
p`?`	zero or one occurrence of pattern *p*	`Java` matches "`J?Java`" `Java` matches "`J?ava`"
p`{`*n*`}`	exactly `n` occurrences of pattern *p*	`Java` matches "`J{1}.*`" `Java` does not match "`.{2}`"
p`{`*n*`,}`	at least `n` occurrences of pattern *p*	`aaaa` matches "`a{1,}`" `a` does not match "`a{2,}`"
p`{`*n*`, `*m*`}`	between `n` and `m` occurrences (inclusive)	`aaaa` matches "`a{1,9}`" `abb` does not match "`a{2,9}bb`"