Design Patterns 2
Singleton, Factory Method, Composite

Shurui Zhou
DESIGN PATTERN?
I THINK I HAVE HEARD ABOUT IT BEFORE
OO Design Principles

Building stable and flexible systems

- Single responsibility principle
- Open/closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle
• A “language” for designing the urban environment.
• The units of this language are patterns.
• window, building, etc..
• 253 design patterns
Design Patterns

• Design Patterns – expert solutions to recurring problems in a certain domain

• Description usually involves problem definition, driving forces, solution, benefits, difficulties, related patterns.

• Pattern Language - a collection of patterns, guiding the users through the decision process in building a system

• Patterns are related
How to make an intersection safer?
How To Think Like An Architect: The Design Process

https://www.youtube.com/watch?v=vmHoGicPQQQ
1994
the GoF book -- the book by the gang of four
Elements of Reusable Object-Oriented Software
23 OO patterns
Lots of books on patterns
Levels of Abstraction

- Requirements
  - high-level “what” needs to be done

Architecture (High-level design)
- high-level “how”, mid-level “what”

OO-Design (Low-level design, e.g. design patterns)
- mid-level “how”, low-level “what”

Code
- low-level “how”
Objects

Model
Design Patterns

- Factory
- View
- Observer
- Model / Subject
- Controller
- Command
Design Patterns

- Factory
- View
- Observer / Subject
- Model / Subject
- Controller
- Command
Design Patterns
Architecture
Architecture
Architecture
Motivating example

Proxy Pattern
Proxy Pattern

Problem:
• High-resolution images on website
• Long loading time
• Style images

Solution:
• Replace with placeholders (proxies)
• Style placeholders
ProxyImage.showImage() loads and displays the real image only when needed...
What does the pattern consist of?

• **Intent** of the pattern briefly describes both the problem and the solution.

• **Motivation** further explains the problem and the solution the pattern makes possible.

• **Structure** of classes shows each part of the pattern and how they are related.

• **Code example** in one of the popular programming languages makes it easier to grasp the idea behind the pattern.
Classification of patterns

• **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.

• **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.

• **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.
<table>
<thead>
<tr>
<th>Scope</th>
<th>Class</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Creational</td>
<td>Structural</td>
</tr>
<tr>
<td></td>
<td>Factory Method</td>
<td>Adapter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Abstract Factory</td>
<td>Adapter</td>
</tr>
<tr>
<td></td>
<td>Builder</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Prototype</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Singleton</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Adapter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bridge</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Composite</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Facade</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Proxy</td>
</tr>
</tbody>
</table>

[Link: https://circle.visual-paradigm.com/catalog/]
Classification of patterns

• Creational patterns
  • Singleton
  • Factory Method

• Structural patterns
  • Composite

• Behavioral patterns
  • Strategy
Singleton
Singleton

- a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

- Example:
  - cache
  - thread pools
  - registries
Singleton

• Use case: Logger

“In case it is not Singleton, every client will have its own Logger object and there will be concurrent access on the Logger instance in Multithreaded environment, and multiple clients will create/write to the Log file concurrently, this leads to data corruption.”
Singleton

• Intent:
  • Ensure that a class has just a single instance
  • Provide a global access point to that instance

Clients may not even realize that they’re working with the same object all the time.
Singleton

• How?
  • Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.
  • Create a static creation method that acts as a constructor.
Singleton

```java
if (instance == null) {
    // Note: if you're creating an app with
    // multithreading support, you should
    // place a thread lock here.
    instance = new Singleton()
}
return instance
```
Singleton

- The **Singleton** class declares the static method `getInstance` that returns the same instance of its own class.
- The Singleton’s constructor should be hidden from the client code.
- Calling the `getInstance` method should be the only way of getting the Singleton object.
Singleton

```java
if (instance == null) {
    // Note: if you're creating an app with
    // multithreading support, you should
    // place a thread lock here.
    instance = new Singleton()
}
return instance
```
Creating a singleton in Python

This question is not for the discussion of whether or not the singleton design pattern is desirable, is an anti-pattern, or for any religious wars, but to discuss how this pattern is best implemented in Python in such a way that is most pythonic. In this instance I define 'most pythonic' to mean that it follows the 'principle of least astonishment'.

I have multiple classes which would become singletons (my use-case is for a logger, but this is not important). I do not wish to clutter several classes with added gumph when I can simply inherit or decorate.

Best methods:

https://stackoverflow.com/questions/6760685/creating-a-singleton-in-python
Singleton - Example

- `java.lang.Runtime`

  Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the `getRuntime` method.

- `java.awt.Desktop#getDesktop()`

- `java.lang.System#getSecurityManager()`
Problems

• Hard to test
• Violation of SRP
• Poor coupling
• Hard to change/refactoring
• race condition
Singleton: Pros and Cons

- You can be sure that a class has only a single instance.
- You gain a global access point to that instance.
- The singleton object is initialized only when it’s requested for the first time.
- Violates the Single Responsibility Principle. The pattern solves two problems at the time.
- The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other.
- The pattern requires special treatment in a multithreaded environment so that multiple threads won’t create a singleton object several times.
- It may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages, you will need to think of a creative way to mock the singleton. Or just don’t write the tests. Or don’t use the Singleton pattern.
Classification of patterns

• Creational patterns
  • Singleton
  • Factory Method

• Structural patterns
  • Composite

• Behavioral patterns
  • Strategy
Factory Method
Factory Method (example)

a logistics management application
Factory Method

• a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.
Factory Method

Creator

```
Logistics
...
+ planDelivery()
+ createTransport()
```

```
return new Truck()
...
+ createTransport()
```

```
RoadLogistics
...
+ createTransport()
```

```
SeaLogistics
...
+ createTransport()
```

```
return new Ship()
```

Products

```
<interface>
Transport
+ deliver()
```

```
Truck
...
+ deliver()
```

```
Ship
...
+ deliver()
```

Deliver by land in a box.

Deliver by sea in a container.
Factory Method

Creator
Factory Method

Products

```
«interface» Transport
+ deliver()

Truck
...
+ deliver()

SHIP
...
+ deliver()

Deliver by land in a box.

Deliver by sea in a container.
```
Factory Method

Creator

```
Logistics
  ...
  + planDelivery()
  + createTransport()

Transport t = createTransport()
```

```
RoadLogistics
  ...
  + createTransport()

return new Truck()
```

```
SeaLogistics
  ...
  + createTransport()

return new Ship()
```

Products

```
<interface>
Transport
+ deliver()
```

```
Truck
  ...
  + deliver()

Deliver by land in a box.
```

```
Ship
  ...
  + deliver()

Deliver by sea in a container.
```
The **Creator** class declares the factory method that returns new product objects. It’s important that the return type of this method matches the product interface.

**Concrete Creators** override the base factory method so it returns a different type of product. Note that the factory method doesn’t have to create new instances all the time. It can also return existing objects from a cache, an object pool, or another source.

**Concrete Products** are different implementations of the product interface.
Factory Method - Example

creating cross-platform UI elements without coupling the client code to concrete UI classes.
Exercise:
Using multiple database servers like SQL Server and Oracle
Factory Method - Applicability

• when you don’t know beforehand the exact types and dependencies of the objects your code should work with.

• when you want to provide users of your library or framework with a way to extend its internal components.

• when you want to save system resources by reusing existing objects instead of rebuilding them each time.
Problem
The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern. The best case scenario is when you’re introducing the pattern into an existing hierarchy of creator classes.
Factory Method – Pros and Cons

✔ You avoid tight coupling between the creator and the concrete products.

✔ Single Responsibility Principle. You can move the product creation code into one place in the program, making the code easier to support.

✔ Open/Closed Principle. You can introduce new types of products into the program without breaking existing client code.

❌ The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern. The best case scenario is when you’re introducing the pattern into an existing hierarchy of creator classes.
Abstract Factory

Chair
Art Deco
Sofa
Victorian
Coffee Table
Modern

LISTEN, I ORDERED SOME CHAIRS LAST WEEK, BUT I GUESS I NEED A SOFA TOO...

HMM... SOMETHING DOES NOT LOOK RIGHT.
Abstract Factory
Abstract Factory
Creational patterns

- **Abstract Factory**
  Creates an instance of several families of classes

- **Builder**
  Separates object construction from its representation

- **Factory Method**
  Creates an instance of several derived classes

- **Object Pool**
  Avoid expensive acquisition and release of resources by recycling objects that are no longer in use

- **Prototype**
  A fully initialized instance to be copied or cloned

- **Singleton**
  A class of which only a single instance can exist
Classification of patterns

• **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.

• **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.

• **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.
Classification of patterns

• Creational patterns
  • Singleton
  • Factory Method

• Structural patterns
  • Composite

• Behavioral patterns
  • Strategy
Composite Pattern
Composite Pattern - Problem

An Ordering System

- 2 types of Objects
  - Products
  - Boxes
Composite Pattern - Solution

Work with Products and Boxes through a common interface which declares a method for calculating the total price. (Recursion)
Composite Pattern - Solution

Run a behavior recursively over all components of an object tree.

Idea: make abstract "component" class.
Composite Example

- Book

```
DocumentComponent

<table>
<thead>
<tr>
<th>Paragraph</th>
<th>Composite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter</td>
<td>Book</td>
</tr>
<tr>
<td></td>
<td>Section</td>
</tr>
</tbody>
</table>
```

- Chapter
  - Section
    - Paragraph
- Chapter
  - Section
    - Paragraph
Composite Design Pattern - Structure

The **Component** interface describes operations that are common to both simple and complex elements of the tree.

- **Component**
  - + execute()
- **Leaf**
  - ...
  - + execute()
  - Do some work.
- **Composite**
  - - children: Component[]
  - + add(c: Component)
  - + remove(c: Component)
  - + getChildren(): Component[]
  - + execute()
  - Delegate all work to child components.
The **Leaf** is a basic element of a tree that doesn’t have sub-elements.
The Composite/container is an element that has sub-elements: leaves or other containers. A container doesn’t know the concrete classes of its children. It works with all sub-elements only via the component interface.

Composite Design Pattern - Structure

```plaintext
Client
   `interface` Component
   + execute()

Leaf
  ...
  + execute()
  Do some work.

Composite
  - children: Component[]
  + add(c: Component)
  + remove(c: Component)
  + getChildren(): Component[]
  + execute()

Delegate all work to child components.
```
Composite Design Pattern - Structure

Client works with all elements through the component interface. As a result, the client can work in the same way with both simple or complex elements of the tree.
The **Component** interface describes operations that are common to both simple and complex elements of the tree.

The **Leaf** is a basic element of a tree that doesn’t have sub-elements.

The **Composite/container** is an element that has sub-elements: leaves or other containers. A container doesn’t know the concrete classes of its children. It works with all sub-elements only via the component interface.

**Client** works with all elements through the component interface. As a result, the client can work in the same way with both simple or complex elements of the tree.
Real world application – Eclipse workspace, SWT (Standard Widget Toolkit)

- **IWorkspace** is the root interface and it is a Composite of **IContainers** and **IFiles**.
violates the Liskov substitution principle (LSP)

• Leaf inherits from Component so it will have an `Add()` method like any other Component.
• But Leafs don't have children, so the following method call cannot return a meaningful result:
Which classes *declare* add and remove children operation?

- **Trade-off between safety and transparency**
  - **Component**: transparency, because you can treat all components uniformly.
  - **Composite**: safety, because any attempt to add or remove objects from leaves will be caught at compile-time in a statically typed language.

We emphasized transparency over safety in this pattern.
Composite – Pros & Cons

✓ You can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage.

✓ Open/Closed Principle. You can introduce new element types into the app without breaking the existing code, which now works with the object tree.

✗ It might be difficult to provide a common interface for classes whose functionality differs too much. In certain scenarios, you’d need to overgeneralize the component interface, making it harder to comprehend.
DevOps
Learning Goals

• Understand DevOps
• Understand CI/CD
• Integrate DevOps into your web application
Developers + Operators = DevOps
https://www.youtube.com/watch?v=_l94-tJlovg
Goal of DevOps

• Improve deployment frequency
• Achieve faster time to market
• Lower failure rate of new releases
• Shorten lead time between fixes
• Improve mean time to recovery
What Are the Challenges DevOps Solves?

• Dev is often unaware of QA and Ops roadblocks that prevent the program from working as anticipated.

• QA and Ops are typically working across many features and have little context of the business purpose and value of the software.

• Each group has opposing goals that can lead to inefficiency and finger pointing when something goes wrong.
How often should you deploy your app to the release environment?
How often different companies deploy to the release environment

<table>
<thead>
<tr>
<th>Company</th>
<th>Deployment Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon</td>
<td>23,000 per day</td>
</tr>
<tr>
<td>Google</td>
<td>5,500 per day</td>
</tr>
<tr>
<td>Netflix</td>
<td>500 per day</td>
</tr>
<tr>
<td>Facebook</td>
<td>1 per day</td>
</tr>
<tr>
<td>Twitter</td>
<td>3 per week</td>
</tr>
<tr>
<td>Typical enterprise</td>
<td>1 every 9 months</td>
</tr>
</tbody>
</table>
Hey, CI/CD tool is running!

DEVELOPERS

CI/CD

https://opensource.com/article/19/4/devops-pipeline
Continuous Integration
• Merging in small code changes frequently

Continuous Delivery
• Add additional automation and testing, get the code nearly ready to deploy with almost no human intervention

Continuous Deployment
• Deploying all the way into production without any human intervention.
Tools - Continuous Integration

- Quickly integrating newly developed code with the main body of code that is to be released
Continuous Integration

• Quickly integrating newly developed code with the main body of code that is to be released
Continuous Integration

https://martinfowler.com/articles/continuousIntegration.html
Continuous Testing

- Selenium

Selenium automates browsers. That's it!
What you do with that power is entirely up to you.

Primarily it is for automating web applications for testing purposes, but is certainly not limited to just that. Boring web-based administration tasks can (and should) also be automated as well.
Ah. We can now effectively collaborate across the team to develop the source codes!

DEVELOPERS

CHECK IN (CI)

CHECK OUT (CO)

CI/CD

1. Source Control Client

SCM

1. Source Codes

SCM - Source Control Mgmt
Build

Our CI/CD tool can now checkout the source codes from our SCM and build it. Cool.
ALL I HAVE TO DO IS CHECK IN MY LATEST CHANGES TO THE CURRENT BUILD

AND... IT'S BROKE!

I will not break the build.
Brian the Build Bunny

http://www.woodwardweb.com/gadgets/000434.html
Web app server
• Lightweight virtualization

• Separate docker images for separate services (web server, business logic, database, ...
Automated Testing
Automate all the things

```bash
#!/bin/bash

pip install "$1" &
easy_install "$1" &
brew install "$1" &
npm install "$1" &
yum install "$1" & dnf install "$1" &
docker run "$1" &
pkg install "$1" &
apt-get install "$1" &
sudo apt-get install "$1" &
steamcmd +app_update "$1" validate &
git clone https://github.com/"$1"/"$1" &
cd "$1"; ./configure; make; make install &
curl "$1" | bash &
```
https://blog.crisp.se/2013/02/05/yassalsundman/continuous-delivery-vs-continuous-deployment
Continuous Deployment
Time for DevOps
QA is Hard
“One portion we planned for but were not able to complete to our satisfaction was testing.”
Cost

The Guardian

Heartbleed bug ‘will cost millions’

Revoking all SSL certificates leaked by Heartbleed will cost millions of dollars, according to Cloudflare, which provides services to website hosts.
QA has many facets
How do you know that your Program works?
Questions

• How can we ensure that the specifications are correct?
• How can we ensure a system meets its specification?
• How can we ensure a system meets the needs of its users?
• How can we ensure a system does not behave badly?
Two kinds of analysis questions

- **Verification**: Does the system meet its specification?
  - i.e. did we build the system correctly?

- **Verification**: are there flaws in design or code?
  - i.e. are there incorrect design or implementation decisions?

- **Validation**: Does the system meet the needs of users?
  - i.e. did we build the right system?

- **Validation**: are there flaws in the specification?
  - i.e., did we do requirements capture incorrectly?
Software Errors

• Functional errors
• Performance errors
• Deadlock
• Race conditions
• Boundary errors
• Buffer overflow
• Integration errors
• Usability errors
• Robustness errors
• Load errors

• Design defects
• Versioning and configuration errors
• Hardware errors
• State management errors
• Metadata errors
• Error-handling errors
• User interface errors
• API usage errors
• ...
Definition: software analysis

The systematic examination of a software artifact to determine its properties.
Definition: software analysis

The **systematic** examination of a software artifact to determine its properties.

- Attempting to be comprehensive, as measured by, as examples:
  - Test coverage, inspection checklists, exhaustive model checking.
<table>
<thead>
<tr>
<th>Type</th>
<th>ID</th>
<th>Checkpoint</th>
<th>Yes/No</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>1</td>
<td>Identify the potential target users of the system</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Demographics</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- User groups</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>What aspects of the application is sensitive to HW and SW differences</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Are there any universal standards and guidelines, to which the application should adhere [E.g. iPhone]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS</td>
<td>1</td>
<td>Create OS compatibility matrix</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Get client confirmation for OS compatibility matrix</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Identify testing scope [domain specific]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Setup multiple virtual machines for each OS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Browser</td>
<td>1</td>
<td>Create Browser compatibility matrix</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Get client confirmation for Browser compatibility matrix</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Identify testing scope [domain specific] - Include most navigable and most frequently accessible pages</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Whether to use Downgradable Browser Versions</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Setup multiple virtual machines if applicable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Device</td>
<td>1</td>
<td>Create Device compatibility matrix</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Get client confirmation for Device compatibility matrix</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Identify testing scope [Domain specific + UI aspects + Configurations]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Setup simulators [For Mobile Devices]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Should application work on jail-broken/rooted devices?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network</td>
<td>1</td>
<td>Create scope on possible access points to system [Dial-up, wireless, 4G, low bandwidth, with proxy, without proxy...etc.]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Create scope on possible access points from system [Printer in same network, access to internet, access external network via firewall]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Get client confirmation on the possible access points identified</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Environment setup for each network configuration</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

https://rochanaqa.wordpress.com/2015/10/05/how-to-plan-and-test-compatibility-using-simple-checklists/
Model Checking
Definition: software analysis

The systematic **examination** of a software artifact to determine its properties.

- **Automated**: Regression testing, static analysis, dynamic analysis
- **Manual**: Manual testing, inspection, modeling
Definition: software analysis

The systematic examination of a software artifact to determine its properties.

• Code, system, module, execution trace, test case, design or requirements document.
Definition: software analysis

The systematic examination of a software artifact to determine its properties.

• **Functional**: code correctness

• **Non-functional**: evolvability, safety, maintainability, security, reliability, performance, ...
VERY IMPORTANT

• *There is no one analysis technique that can perfectly address all quality concerns.*

• Which techniques are appropriate depends on many factors, such as the system in question (and its size/complexity), quality goals, available resources, safety/security requirements, etc etc...
Principle techniques

• **Dynamic:**
  • **Testing:** Direct execution of code on test data in a controlled environment.
  • **Analysis:** Tools extracting data from test runs.

• **Static:**
  • **Inspection:** Human evaluation of code, design documents (specs and models), modifications.
  • **Analysis:** Tools reasoning about the program without executing it.
Classic Testing
(Functional Correctness)
Testing

• Executing the program with selected inputs in a controlled environment (dynamic analysis)

• Goals:
  • Reveal bugs (main goal)
  • Assess quality (hard to quantify)
  • Clarify the specification, documentation
  • Verify contracts

"Testing shows the presence, not the absence of bugs"
Edsger W. Dijkstra 1969
Specifications

• Textual
• Assertions
• Formal specifications
Algorithms.shortestDistance(g, "Tom", "Anne");
> ArrayOutOfBoundsException

Algorithms.shortestDistance(g, "Tom", "Anne");
> -1

class Algorithms {
   /**
    * This method finds the shortest distance between two verticies. It returns -1 if the two nodes are not connected.
    */
   int shortestDistance(...) {...}
}

class Algorithms {
   /**
    * This method finds the shortest distance between two verticies. Method is only supported for connected verticies.
    */
   int shortestDistance(...) {...}
}
• JML (Java modeling language specification)

```java
/**
 * Calls the <code>read(byte[], int, int)</code> overloaded method.
 * @param buf The buffer to read bytes into
 * @return The value returned from <code>in.read(byte[], int, int)</code>
 * @exception IOException If an error occurs
 */
public int read(byte[] buf) throws IOException {
    return read(buf, 0, buf.length);
}
```

• Textual specification with JavaDoc
Benefits of Specification

• Exact specification of what should be implemented
• Decompose a system into its parts, develop and test parts independently
• Accurate blame assignments and identification of buggy behavior
• Useful for test generation and as test oracle