SD2x3.1

Motivating Software Design

Chris
Fred Brooks
“There is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity.”
“I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation.”
“Essential Difficulties”

• **Complexity:** Too complex for one person to understand

• **Conformity:** Interfaces with different users, systems, etc.

• **Changeability:** Software is “infinitely malleable”

• **Invisibility:** Difficult to diagram or visualize
“Study after study shows that the very best designers produce structures that are faster, smaller, simpler, cleaner, and produced with less effort.”
“I think the most important single effort we can mount is to develop ways to grow great designers.”
What is Software Design?

• The process of converting a set of requirements into a software artifact (code)

• Doing so in such a way that the code is:
  • easy to understand
  • easy to change
Why is Software Design so important?

• Software **changes** during its lifetime
  • Fixing bugs
  • Adding new features
  • Using new libraries
  • Improving efficiency

• Software that is well-designed facilitates those changes, saving time and money
How do we design software?

1. Modeling: identify concepts and their relationships

2. Choose the right architecture

3. Create/Use appropriate data structures

4. Use design patterns when appropriate

5. Refactor (change the design) as necessary
SD2x3.2

Domain Modeling

Chris
How do we design software?

1. Modeling: identify concepts and their relationships
2. Choose the right architecture
3. Create/Use appropriate data structures
4. Use design patterns when appropriate
5. Refactor (change the design) as necessary
How do we design software?

1. **Modeling**: identify concepts and their relationships

2. Choose the right architecture

3. **Create/Use** appropriate data structures

4. **Use** design patterns when appropriate

5. **Refactor** (change the design) as necessary
How do we design classes?
Domain Modeling

- Identify **concepts** (classes)
- Their **properties** (attributes/fields)
- Their **behaviors** (operations/methods)
- And the **relationships** between them
“Graduate and undergraduate students may apply to the program. The application consists of a recommendation letter and personal statement.”
Grammatical Parsing

• Assign roles based on parts of speech

• **Noun**: class or attribute

• **Adjective**: attribute or subclass

• **Verb**: operation
“Graduate and undergraduate students may apply to the program. The application consists of a recommendation letter and personal statement.”
“Graduate and undergraduate students may apply to the program. The application consists of a recommendation letter and personal statement.”
“Graduate and undergraduate students may apply to the program. The application consists of a recommendation letter and personal statement.”
Grammatical Parsing

- **Nouns:**
  - Students
  - Program
  - Application
  - Letter
  - Statement

- **Adjectives:**
  - Graduate
  - Undergraduate

- **Verbs:**
  - Apply
  - Consists
Nouns: Classes or Attributes?

• A noun should be a **class** if:
  • it has multiple attributes and/or operations
  • all attributes/operations are common to all instances of the class

• Otherwise it should likely be an **attribute** of another class
Adjectives: Attributes or Subclasses?

• An adjective should represent a **subclass** of another class if:
  • different adjectives would imply different or additional operations or behavior
  • different adjectives would imply different attributes

• Otherwise, an adjective should be an **attribute** of a class
Verbs: in which class do they belong?

• “Student may apply to the program”
  • Subject: student
  • Direct object: program
→ “apply” operation should go in Student class

• Certain verbs are more indicative of relationships between classes than they are of operations: contains, has, is, uses, consists
Our design

• Class: Student
  • Attributes/Subclasses: Graduate; Undergraduate
  • Operations: Apply

• Class: Letter
• Class: Statement
• Class: Application
• Class: Program
Relationships?

- Graduate Student is a Student
- Undergraduate Student is a Student

- Application consists of Letter and Statement

- Student uses Application

- Student is associated with Program
Recap: Domain Modeling

- Identify concepts, properties, behaviors, and relationships

- Use **grammatical parsing** to assign roles based on parts of speech
  - **Noun**: class or attribute
  - **Adjective**: attribute or subclass
  - **Verb**: operation
SD2x3.3

UML Class Diagrams

Chris
How do we represent a class design?
“Graduate and undergraduate students may apply to the program. The application consists of a recommendation letter and personal statement.”
“Graduate and undergraduate students may apply to the program. The application consists of a recommendation letter and personal statement.”
Relationships

- Graduate Student is a Student
- Undergraduate Student is a Student

- Application consists of Letter and Statement

- Student uses Application

- Student is associated with Program
UML: Unified Modeling Language

- General-purpose language for modeling/visualizing software architecture and design

- Adopted as standard by Object Management Group (OMG) in 1997 and International Standards Organization (ISO) in 2005

- Types of diagrams:
  - Structural
  - Behavioral
UML Class Diagram (Compact)
UML Class Diagram (Compact)
UML Class Diagram

Realization ("is-a")

Program

Student

Undergraduate Student

Graduate Student

Application

Letter

Statement
“is-a” Relationships in UML

- **Generalization**: one is a specialized form of the other and provides additional behavior/attributes

- **Realization**: one is the realization/actualization of the model that the other supplies

- In Java:
  - Generalization = “extends” a (concrete) class
  - Realization = “implements” an interface
UML Class Diagram

Aggregation ("has-a")
“has-a” Relationships in UML

- **Aggregation**: one class is part of another and can exist independently
- **Composition**: one class is part of another but *cannot* exist independently

In Java, both are implemented by having one class as a field/attribute of another
UML Class Diagram

Dependency (“uses”)
“uses” vs. “has” in UML

- **Dependency (“uses”):** class A uses class B’s operations/attributes but class B is not part of A

- **Aggregation/Composition (“has”):** class B is part of class A

- In Java, dependency is exhibited when a class is a parameter or local variable in another class, but is not a field
UML Class Diagram

Program

Association

Student

Undergraduate Student
Graduate Student

Application
Letter
Statement
Recap: UML Class Relationships

- Generalization
- Realization
- Aggregation
- Composition
- Dependency
- Association
SD2x3.4

Detailed Class Diagrams

Chris
How do we represent more details about a class design?
UML Class Diagram (Compact)
UML Class Diagram (Compact)
UML Class Diagram (Compact)
UML Class Diagram (Compact)
UML Class Diagram (Compact)
UML Class Diagram (Compact)
UML Class Diagram (Compact)
UML Class Diagram (Detailed)

```
BankAccount

balance : double
accountld : string

+open(double) : void
+withdraw(double) : double
+deposit(double) : double
```
UML Class Diagram (Detailed)

```
BankAccount

balance : double
accountld : string

+open(double) : void
+withdraw(double) : double
+deposit(double) : double
```
UML Class Diagram (Detailed)

```
BankAccount

balance : double
accountld : string

+open(double) : void
+withdraw(double) : double
+deposit(double) : double
```
UML Class Diagram (Detailed)

```
BankAccount

balance : double
accountld : string

+open(double) : void
+withdraw(double) : double
+deposit(double) : double
```
UML Class Diagram (Detailed)

**BankAccount**
- balance : double
- accountld : string
- +open(double) : void
- +withdraw(double) : double
- +deposit(double) : double

**SavingsAccount**
- interestRate : double
- +applyInterest(void) : double

**CheckingAccount**
- overdraftLimit : double
UML Class Diagram (Detailed)

**BankAccount**
- balance : double
- accountld : string
- +open(double) : void
- +withdraw(double) : double
- +deposit(double) : double

**SavingsAccount**
- interestRate : double
- +applyInterest(void) : double

**CheckingAccount**
- overdraftLimit : double
UML Class Diagram (Detailed)

**BankAccount**
- balance : double
- accountld : string
- +open(double) : void
- +withdraw(double) : double
- +deposit(double) : double

**SavingsAccount**
- interestRate : double
- +applyInterest(void) : double

**CheckingAccount**
- overdraftLimit : double
UML Class Diagram (Detailed)

**BankAccount**
- balance : double
- accountld : string
- +open(double) : void
- +withdraw(double) : double
- +deposit(double) : double

**SavingsAccount**
- interestRate : double
- +applyInterest(void) : double

**CheckingAccount**
- overdraftLimit : double

**Customer**
- customerId : string

(accountOwner 1..* 1)
UML Class Diagram (Detailed)

**BankAccount**
- balance : double
- accountId : string
- +open(double) : void
- +withdraw(double) : double
- +deposit(double) : double

**Customer**
- customerId : string

**SavingsAccount**
- interestRate : double
- +applyInterest(void) : double

**CheckingAccount**
- overdraftLimit : double
UML Class Diagram (Detailed)

**BankAccount**
- balance : double
- accountId : string
- +open(double) : void
- +withdraw(double) : double
- +deposit(double) : double

**SavingsAccount**
- interestRate : double
- +applyInterest(void) : double

**CheckingAccount**
- overdraftLimit : double

**Customer**
- customerId : string

**AccountOwner**
- 1..*
### Multiplicity

- Describes the number of objects that can be used in an association
  - $n$: exactly $n$ objects
  - $*$: 0 or more objects
  - $m..n$: a minimum of $m$ and a maximum of $n$ objects
Multiplicity and Attributes

- In general, do not use aggregation/composition for “primitive” types, even to represent multiplicity

- To demonstrate multiplicity, use [ ]-notation
UML Class Diagram (Detailed)

**BankAccount**
- balance : double
- accountld : string
- +open(double) : void
- +withdraw(double) : double
- +deposit(double) : double

1..*  
accountOwner

**SavingsAccount**
- interestRate : double
- +applyInterest(void) : double

**CheckingAccount**
- overdraftLimit : double
UML Class Diagram (Detailed)

```
BankAccount
balance : double
accountId : string
+open(double) : void
+withdraw(double) : double
+deposit(double) : double

SavingsAccount
interestRate : double
+applyInterest(void) : double

CheckingAccount
overdraftLimit : double

Customer
customerId : string

accountOwner

1..*

Person
name : string
emailAddress : string
```

accountOwner

1
UML Class Diagram (Detailed)

**BankAccount**
- balance : double
- accountId : string

  + open(double) : void
  + withdraw(double) : double
  + deposit(double) : double

  1..* accountOwner

**SavingsAccount**
- interestRate : double

  + applyInterest(void) : double

**CheckingAccount**
- overdraftLimit : double

**Customer**
- customerId : string

**Person**
- name : string
- emailAddress : string
UML Class Diagram (Detailed)
How do we convert a UML class design into Java?
public abstract class BankAccount {

    protected double balance;
    protected String accountId;
    protected Customer accountOwner;

    // NOT SHOWN: constructors, get/set methods

    public void open(double initialBalance) { . . . }

    public double deposit(double amount) { . . . }

    public double withdraw(double amount) { . . . }

}
public abstract class BankAccount {

    protected double balance;
    protected String accountId;
    protected Customer accountOwner;

    // NOT SHOWN: constructors, get/set methods

    public void open(double initialBalance) { . . . }

    public double deposit(double amount) { . . . }

    public double withdraw(double amount) { . . . }

}
public abstract class BankAccount {

    protected double balance;
    protected String accountId;
    protected Customer accountOwner;

    // NOT SHOWN: constructors, get/set methods

    public void open(double initialBalance) { . . . }

    public double deposit(double amount) { . . . }

    public double withdraw(double amount) { . . . }

}
public abstract class BankAccount {

    protected double balance;
    protected String accountId;
    protected Customer accountOwner;

    // NOT SHOWN: constructors, get/set methods

    public void open(double initialBalance) { . . . }
    public double deposit(double amount) { . . . }
    public double withdraw(double amount) { . . . }
}
public abstract class BankAccount {

    protected double balance;
    protected String accountId;
    protected Customer accountOwner;

    // NOT SHOWN: constructors, get/set methods

    public void open(double initialBalance) { ... }

    public double deposit(double amount) { ... }

    public double withdraw(double amount) { ... }

}
UML Class Diagram (Detailed)
public class SavingsAccount extends BankAccount {

    protected double interestRate;

    // NOT SHOWN: constructors, get/set methods

    public double applyInterest() { .. . }  
}
public class SavingsAccount extends BankAccount {

    protected double interestRate;

    // NOT SHOWN: constructors, get/set methods

    public double applyInterest( ) { . . . }
}

public class SavingsAccount extends BankAccount {

    protected double interestRate;

    // NOT SHOWN: constructors, get/set methods

    public double applyInterest( ) { . . . }
}

public class SavingsAccount extends BankAccount {

    protected double interestRate;

    // NOT SHOWN: constructors, get/set methods

    public double applyInterest( ) { . . . }

}
UML Class Diagram (Detailed)
UML Class Diagram (Detailed)
public class Person {

    protected String name;
    protected String emailAddress;
    protected Address homeAddress;
    protected Address workAddress;

    // NOT SHOWN: constructors, get/set methods

}

public class Customer extends Person {

    protected String customerId;

    // NOT SHOWN: constructors, get/set methods

}
public class Person {

    protected String name;
    protected String emailAddress;
    protected Address homeAddress;
    protected Address workAddress;

    // NOT SHOWN: constructors, get/set methods
}

public class Customer extends Person {

    protected String customerId;

    // NOT SHOWN: constructors, get/set methods
}
public class Person {

    protected String name;
    protected String emailAddress;
    protected Address homeAddress;
    protected Address workAddress;

    // NOT SHOWN: constructors, get/set methods

}

public class Customer extends Person {

    protected String customerId;

    // NOT SHOWN: constructors, get/set methods

}
public class Person {
    protected String name;
    protected String emailAddress;
    protected Address homeAddress;
    protected Address workAddress;

    // NOT SHOWN: constructors, get/set methods
}

public class Customer extends Person {
    protected String customerId;

    // NOT SHOWN: constructors, get/set methods
}
public class Person {

    protected String name;
    protected String emailAddress;
    protected Address homeAddress;
    protected Address workAddress;

    // NOT SHOWN: constructors, get/set methods

}

public class Customer extends Person {

    protected String customerId;

    // NOT SHOWN: constructors, get/set methods

}
public class Person {

    protected String name;
    protected String emailAddress;
    protected Address homeAddress;
    protected Address workAddress;

    // NOT SHOWN: constructors, get/set methods

}

public class Customer extends Person {

    protected String customerId;

    // NOT SHOWN: constructors, get/set methods

}
Recap: Detailed UML Class Diagrams

- Allow us to show details of classes in our design
  - Fields (attributes)
  - Methods (operations)
  - Named relationships
  - Multiplicity

- Can easily be converted to Java
SD2x3.5

Software Quality

Chris
Is software getting better?
Perspectives on Quality

- **Transcendental**: can be recognized but not defined or measured
- **User**: satisfies end-users’ needs
- **Manufacturer**: conforms to specification (regardless of what users want)
- **Product**: based on internal characteristics
- **Value-based**: depends on what someone is willing to pay for it
Quality Models

- **McCall**: hierarchy of factors, criteria, and metrics (1977)

- **ISO 9126**: characteristics and sub-characteristics (1991)

- **ISO 25010**: revision (2011)

A definition of Software Quality

- **External**: executable program running on hardware in some environment

- **Internal**: code as human-readable text
External Quality

- **Functionality**: satisfies stated or implied needs
- **Reliability**: maintains a level of performance in a given environment for a given period of time
- **Usability**: ease with which user can understand, learn, and operate the software
- **Efficiency**: limits the amount/number of resources used
- **Portability**: ease with which software can be transferred from one environment to another
- **Security**: level of confidentiality and integrity provided by the software
Internal Quality: “Maintainability”

 Ease with which a programmer can:

- understand the code
- add new functionality
- modify existing functionality
- reuse code
- incorporate new code
- identify and fix defects
- modify the code to improve external quality
Internal Quality

Analyzability
Changeability
Stability
Testability
Analyzability

- Ease with which a programmer can **read**, **understand**, and **reason about** the code
- Identify and fix defects
- Add or improve functionality
- Understand how an algorithm (solution) is implemented
- Improve other aspects of software quality (efficiency, security, etc.)
Changeability

- Ease with which a programmer can change the code
- Fix defects
- Add or improve functionality
- Incorporate new code
- Improve other aspects of software quality (efficiency, security, etc.)
Stability

- Extent to which the code is tolerant of changes to other parts of the code

- Changes in one part of code don’t require changes in other parts

- Reduce time/effort required to change code, since you won’t also need to change other code
Testability

- Ease with which a the code can be tested

- Create test cases
- Identify and fix defects
- Increase confidence that software is working correctly
Times change, people change, situations change, relationships change. The only thing constant is change.
Software Design Concepts

Chris
How do we design software?

1. Modeling: identify concepts and their relationships

2. Choose the right architecture

3. Create/Use appropriate data structures

4. Use design patterns when appropriate

5. Refactor (change the design) as necessary
How do we design software?

1. Modeling: identify concepts and their relationships

2. Choose the right architecture

3. Create/Use appropriate data structures

4. Use design patterns when appropriate

5. Refactor (change the design) as necessary
Motivating Example

• Design a Java program that reads a file containing info about courses, instructors, and enrollment

• And then allows the user to choose to see either:
  • the instructor(s) for a given course
  • the average enrollment for the course
The Monolith!

• A **single** piece of code (main method) that:
  
  • Reads the data from the file and puts it into a data structure
  
  • Prompts the user for the operation to perform (show instructors for a course, or show average enrollment for a course)
  
  • Prompts the user to enter the course number
  
  • Performs the operation and handles errors accordingly
  
  • Shows the results to the user
What’s wrong with the Monolith?
What’s wrong with the Monolith?

- **anaylzyability**: main method can be fairly long
- **changeability**: hard to find the code you want to change
- **stability**: hard to know whether a change in one place will affect other places
- **testability**: if something goes wrong, hard to find the bug
- hard to **reuse** any of the code
How can we address these problems?
High-level design (Architecture)

• Divide system into subsystems (components, or combinations of components) that address major features

• Each subsystem should be able to do its work with minimal dependency on other subsystems

• Examples:
  • Model-View-Controller (MVC)
  • Software as a Service (SaaS)
  • N-Tier
Three-Tier Architecture

Presentation

Get input from user.
Display output.
Three-Tier Architecture

Presentation

Logic

Get input from user.
Display output.

Perform calculations.
Make decisions.
Three-Tier Architecture

Presentation
- Get input from user.
- Display output.

Logic
- Perform calculations.
- Make decisions.

Data
- Store and retrieve data.
Three-Tier Architecture

- **Presentation**
  - Get input from user.
  - Display output.

- **Logic**
  - Perform calculations.
  - Make decisions.

- **Data**
  - Store and retrieve data.
Three-Tier Architecture

- **Presentation**
  - Get input from user.
  - Display output.

- **Logic**
  - Perform calculations.
  - Make decisions.

- **Data**
  - Store and retrieve data.
Applying the Three-Tier Architecture

UserInterface
+start() : void
doInstructorsForCourse() : void
doAverageEnrollmentForCourse() : void

CourseEvalProcessor
offerings : ArrayList<ArrayList<String>>
+getInstructorsForCourse(String) : HashSet<String>
+getAverageEnrollmentForCourse(String) : AverageEnrollment

AverageEnrollment
count : int
average : double

FileReader
fileName : String
+getAllOfferings() : ArrayList<ArrayList<String>>

processor
reader
Applying the Three-Tier Architecture

**UserInterface**

- `+start() : void`
- `doInstructorsForCourse() : void`
- `doAverageEnrollmentForCourse() : void`

**CourseEvalProcessor**

- `offerings : ArrayList< ArrayList<String> >`
- `+getInstructorsForCourse(String) : HashSet<String>`
- `+getAverageEnrollmentForCourse(String) : AverageEnrollment`

**AverageEnrollment**

- `count : int`
- `average : double`

**FileReader**

- `fileName : String`
- `+getAllOfferings() : ArrayList<ArrayList<String>>`
Applying the Three-Tier Architecture

UserInterface

+start() : void
doInstructorsForCourse() : void
doAverageEnrollmentForCourse() : void

processor

CourseEvalProcessor

offerings : ArrayList< ArrayList<String> >

+getInstructorsForCourse(String) : HashSet<String>
+getAverageEnrollmentForCourse(String) : AverageEnrollment

reader

FileReader

fileName : String

+getAllOfferings() : ArrayList< ArrayList<String> >

AverageEnrollment

count : int
average : double
Applying the Three-Tier Architecture

UserInterface
+start() : void
doInstructorsForCourse() : void
doAverageEnrollmentForCourse() : void

CourseEvalProcessor
offerings : ArrayList< ArrayList<String> >
+getInstructorsForCourse(String) : HashSet<String>
+getAverageEnrollmentForCourse(String) : AverageEnrollment

AverageEnrollment
count : int
average : double

FileReader
fileName : String
+getAllOfferings() : ArrayList< ArrayList<String> >

processor
reader
Why is this better than Monolith?

- **analyzability**: the program is divided into smaller pieces that are easier to read and understand
- **changeability**: easier to find the code you want to change; less code to change
- **stability**: can change one piece without worrying about its effect on others
- **testability**: easy to test each of these separately
- can **reuse** different parts of the code
How can we achieve internal quality?
Software Design Concepts

- **Modularity:** each component addresses a single part of the functionality

- **Functional Independence:** components should have minimal dependence on other components

- **Abstraction:** components should be able to use other components with minimal knowledge of the details of their implementation
SD2x3.7

Modularity

Chris
Internal Quality

Analyzeability

Changeability

Stability

Testability
Software Design Concepts

Modularity

Functional Independence

Abstraction
Software Design Concepts

Modularity

Functional Independence

Abstraction
Motivating Example

• Design a Java program that reads a file containing info about courses, instructors, and enrollment

• And then allows the user to choose to see either:
  • the instructor(s) for a given course
  • the average enrollment for the course
A Modular Design

UserInterface

+start() : void
doInstructorsForCourse() : void
doAverageEnrollmentForCourse() : void

CourseEvalProcessor

offerings : ArrayList<ArrayList<String>>

+getInstructorsForCourse(String) : HashSet<String>
+getAverageEnrollmentForCourse(String) : AverageEnrollment

FileReader

fileName : String

+getAllOfferings() : ArrayList<ArrayList<String>>
A Modular Design

UserInterface
+start() : void
doInstructorsForCourse() : void
doAverageEnrollmentForCourse() : void

CourseEvalProcessor
offerings : ArrayList<ArrayList<String>>
+getInstructorsForCourse(String) : HashSet<String>
+getAverageEnrollmentForCourse(String) : AverageEnrollment

FileReader
fileName : String
+getAllOfferings() : ArrayList<ArrayList<String>>

AverageEnrollment
count : int
average : double
A Modular Design

**UserInterface**

+ start(): void
doInstructorsForCourse(): void
doAverageEnrollmentForCourse(): void

**CourseEvalProcessor**

offerings: ArrayList< ArrayList<String> >

+ getInstructorsForCourse(String): HashSet<String>
+ getAverageEnrollmentForCourse(String): AverageEnrollment

**AverageEnrollment**

count: int
average: double

**FileReader**

fileName: String

+ getAllOfferings(): ArrayList<ArrayList<String>>
A Modular Design

UserInterface

+start() : void
doInstructorsForCourse() : void
doAverageEnrollmentForCourse() : void

CourseEvalProcessor

offerings : ArrayList<ArrayList<String>>
+getInstructorsForCourse(String) : HashSet<String>
+getAverageEnrollmentForCourse(String) : AverageEnrollment

AverageEnrollment

count : int
average : double

FileReader

fileName : String
+getAllOfferings() : ArrayList<ArrayList<String>>
Modularity: “Separation of Concerns”

• Each component (“module”) addresses a single part of the functionality

• In Java, this is often accomplished simply by creating separate classes

• **Single Responsibility Principle**: “a class should only have one reason to change”
Why Modularity?

• **Analyzeability**: leads to smaller pieces of code, each of which is easier to understand

• **Testability**: permits testing of individual pieces; easier to find and fix bugs

• **“Reusability”**: pieces of code can be reused in this application or in other applications

• **Changeability**: easy to find code that needs to be changed (and then change it!)
What could change?

**UserInterface**
- `start() : void`
- `doInstructorsForCourse() : void`
- `doAverageEnrollmentForCourse() : void`

**CourseEvalProcessor**
- `offerings : ArrayList<ArrayList<String>>`
- `getInstructorsForCourse(String) : HashSet<String>`
- `getAverageEnrollmentForCourse(String) : AverageEnrollment`

**AverageEnrollment**
- `count : int`
- `average : double`

**FileReader**
- `fileName : String`
- `getAllOfferings() : ArrayList<ArrayList<String>>`
What could change?

UserInterface
+start() : void
doInstructorsForCourse() : void
doAverageEnrollmentForCourse() : void

CourseEvalProcessor
offerings : ArrayList<ArrayList<String>>
+getInstructorsForCourse(String) : HashSet<String>
+getAverageEnrollmentForCourse(String) : AverageEnrollment

AverageEnrollment
count : int
average : double

FileReader
fileName : String
+getAllOfferings() : ArrayList<ArrayList<String>>
What could change? User Interface

- (Human) language used in UI
- Prompts for input
- Mechanism for input
- Format of output
- UI technology: command line, Swing, Android
- How error messages are displayed
What could change?

UserInterface

+start( ): void
doInstructorsForCourse(): void
doAverageEnrollmentForCourse(): void

processor

CourseEvalProcessor

offerings: ArrayList<ArrayList<String>>
+getInstructorsForCourse(String): HashSet<String>
+getAverageEnrollmentForCourse(String): AverageEnrollment

reader

FileReader

fileName: String
+getAllOfferings(): ArrayList<ArrayList<String>>

AverageEnrollment

count: int
average: double
What could change? Processor

- What we’re searching
- Logic of conducting the search
- Details of the search, e.g. whether it’s case-sensitive
- How errors are handled
What could change?

**UserInterface**

+start( ) : void
doInstructorsForCourse( ) : void
doAverageEnrollmentForCourse( ) : void

**CourseEvalProcessor**

offerings : ArrayList< ArrayList<String> >

+getInstructorsForCourse(String) : HashSet<String>
+getAverageEnrollmentForCourse(String) : AverageEnrollment

**AverageEnrollment**

count : int
average : double

**FileReader**

fileName : String

+getAllOfferings( ) : ArrayList< ArrayList<String> >
What could change? FileReader

- Format of input: CSV, JSON, XML, etc.
- Layout of format: ordering of fields
- Source of input: file, database, web service
- How errors are handled
Another Motivating Example

• Design a program that reads in a file containing tweets, which include text and geolocation info

• Determine the most popular hashtag for a given U.S. state
A modular design

```
UserInterface
+start( ) : void
+doMostPopularHashtagForState( ) : void

TweetProcessor
+getMostPopularHashtagForState(String) : String
+determineStateForTweet(String) : String

FileReader
+getAllTweets( ) : ArrayList<String>
+getAllStates( ) : ArrayList<String>
```
A modular design

UserInterface

+start( ) : void
+doMostPopularHashtagForState( ) : void

TweetProcessor

+getMostPopularHashtagForState(String) : String
+determineStateForTweet(String) : String

FileReader

+getAllTweets( ) : ArrayList<String>
+getAllStates( ) : ArrayList<String>

processor
reader
A modular design

UserInterface
- start() : void
- doMostPopularHashtagForState() : void

TweetProcessor
- getMostPopularHashtagForState(String) : String
- determineStateForTweet(String) : String

FileReader
- getAllTweets() : ArrayList<String>
- getAllStates() : ArrayList<String>
A modular design

**UserInterface**
- `start( ) : void`
- `doMostPopularHashtagForState( ) : void`

**TweetProcessor**
- `getMostPopularHashtagForState(String) : String`
- `determineStateForTweet(String) : String`

**FileReader**
- `getAllTweets( ) : ArrayList<String>`
- `getAllStates( ) : ArrayList<String>`
A modular design

UserInterface

+start( ) : void
+doMostPopularHashtagForState( ) : void

TweetProcessor

+getMostPopularHashtagForState(String) : String
+determineStateForTweet(String) : String

FileReader

+getAllTweets( ) : ArrayList<String>
+getAllStates( ) : ArrayList<String>

processor

reader
A more modular design
A more modular design

UserInterface

+start( ) : void
+doMostPopularHashtagForState( ) : void

TweetProcessor

+getMostPopularHashtagForState(String) : String
determineStateForTweet(String) : String

TweetFileReader

+getAllTweets( ) : ArrayList<String>

StateFileReader

+getAllStates( ) : ArrayList<String>
A more modular design

UserInterface
+start( ) : void
+doMostPopularHashtagForState( ) : void

TweetProcessor
+getMostPopularHashtagForState(String) : String
determineStateForTweet(String) : String

tweetReader

TweetFileReader
+getAllTweets( ) : ArrayList<String>

stateReader

StateFileReader
+getAllStates( ) : ArrayList<String>
A more modular design

UserInterface

+start( ) : void
+doMostPopularHashtagForState( ) : void

TweetProcessor

+getMostPopularHashtagForState(String) : String
determineStateForTweet(String) : String

TweetFileReader

+getAllTweets( ) : ArrayList<String>

StateFileReader

+getAllStates( ) : ArrayList<String>
An even more modular design
An even more modular design

UserInterface
+start() : void
+doMostPopularHashtagForState() : void

TweetProcessor
+getMostPopularHashtagForState(String) : String

TweetFileReader
+getAllTweets() : ArrayList<String>

StateFinder
+determineStateForTweet(String) : String

StateFileReader
+getAllStates() : ArrayList<String>
An even more modular design

UserInterface
+start( ) : void
+doMostPopularHashtagForState( ) : void

TweetProcessor
+getMostPopularHashtagForState(String) : String

TweetFileReader
+getAllTweets( ) : ArrayList<String>

StateFinder
+determineStateForTweet(String) : String

StateFileReader
+getAllStates( ) : ArrayList<String>
Cohesion

- The extent to which the parts of a module (class) “go together”

- All operations and attributes should contribute to a single, well-defined task
Is this cohesive?
Is this cohesive?

```
UserInterface
+start() : void
doInstructorsForCourse() : void
doAverageEnrollmentForCourse() : void
```

```
CourseEvalProcessor
offerings : ArrayList<ArrayList<String>>
+getInstructorsForCourse(String) : HashSet<String>
+getAverageEnrollmentForCourse(String) : AverageEnrollment
```

```
AverageEnrollment
count : int
average : double
```

```
FileReader
fileName : String
+getAllOfferings() : ArrayList<ArrayList<String>>
```
Is this too much modularity?
Is this too much modularity?
Is this too much modularity?
A cohesive, modular design

UserInterface
+start() : void
doInstructorsForCourse() : void
doAverageEnrollmentForCourse() : void

CourseEvalProcessor
offerings : ArrayList<ArrayList<String>>
+getInstructorsForCourse(String) : HashSet<String>
+getAverageEnrollmentForCourse(String) : AverageEnrollment

AverageEnrollment
count : int
average : double

FileReader
fileName : String
+getAllOfferings() : ArrayList<ArrayList<String>>
Recap: Modularity

- Each component ("module") addresses a single part of the functionality
- Improves analyzability, testability, and changeability
- Cohesion: the extent to which the parts of a module "go together"
SD2x3.8

Functional Independence

Chris
Software Design Concepts

Modularity

Functional Independence

Abstraction
Software Design Concepts

Modularity

Functional Independence

Abstraction
Motivating Example

- Design a program that reads in a file containing tweets, which include text and geolocation info

- Determine the most popular hashtag for a given U.S. state
A Modular Design
A Modular Design

UserInterface
+start( ) : void
+doMostPopularHashtagForState( ) : void

TweetProcessor
+getMostPopularHashtagForState(State) : String

tweetReader

TweetFileReader
+getAllTweets( ) : Tweet [*]
+setFileFormat(string) : void

Tweet
date : Date
username : string
content : string

Location
latitude : double
longitude : double

State
name : string

center

StateFinder
allStates : State[*]
+determineState(Tweet) : State

stateFinder

stateReader

StateFileReader
+getAllStates(StateFinder) : void
A Modular Design

**UserInterface**
+start() : void
+doMostPopularHashtagForState() : void

**TweetProcessor**
+getMostPopularHashtagForState(State) : String

**StateFinder**
allStates : State[*]
+determineState(Tweet) : State

**TweetFileReader**
+getAllTweets() : Tweet[*]
+setFileFormat(string) : void

**StateFileReader**
+getAllStates(StateFinder) : void

**Tweet**
date : Date
username : string
content : string

**Location**
latitude : double
longitude : double

**State**
name : string

processor

stateFinder

stateReader

tweetReader
A Modular Design

UserInterface
+start( ) : void
+doMostPopularHashtagForState( ) : void

TweetProcessor
+getMostPopularHashtagForState(State) : String

StateFinder
allStates : State[*]
+determineState(Tweet) : State

TweetFileReader
+getAllTweets( ) : Tweet[*]
+setFileFormat(string) : void

Tweet
date : Date
username : string
content : string

Location
latitude : double
longitude : double

State
name : string

processor

stateReader
stateFinder

SD2x-3.8
Functional Independence

• A module should be able to do its work with minimal dependence on other modules

• Try to minimize the number and complexity of interfaces between modules
Why Functional Independence?

- **Analyzeability**: easier to understand a module without having to look at other code

- **Testability**: permits testing of a module independent from other modules

- **“Reusability”**: pieces of code can be reused without need their dependencies

- **Changeability/Stability**: can change code without needing to change the things that depend on it
Are these modules independent?
Cyclic Dependencies: BAD!

UserInterface
+start( ) : void
+doMostPopularHashtagForState( ) : void

TweetProcessor
+getMostPopularHashtagForState(State) : String

tweetReader

TweetFileReader
+getAllTweets( ) : Tweet [*]
+setFileFormat(string) : void

Tweet
date : Date
username : string
content : string

Location
latitude : double
longitude : double

State
name : string

tweetReader

StateFinder
allStates : State[*]
+determineState(Tweet) : State

StateReader

StateFileReader
+getAllStates(StateFinder) : void
Cyclic Dependencies: BAD!
Cyclic Dependencies: BAD!

```
UserInterface
+start() : void
+doMostPopularHashtagForState() : void

TweetProcessor
+getMostPopularHashtagForState(State) : String

TweetFileReader
+getAllTweets() : Tweet[*]
+setFileFormat(string) : void

Tweet
date : Date
tweet : string
content : string

Location
latitude : double
longitude : double

State
name : string

determineState(Tweet) : State

StateReader
+getAllStates(StateReader) : void

StateFinder
allStates : State[*]

processor
tweetReader
stateReader
stateFinder
```
Cyclic Dependencies: BAD!
public class TweetProcessor {
    protected StateFinder stateFinder;
    protected StateFileReader stateFileReader;

    public TweetProcessor() {
        stateFinder = new StateFinder();
        stateFileReader = new StateFileReader();
        stateFinder.setStateReader(stateFileReader);
        stateFileReader.getAllStates(stateFinder);
    }

    ...
}
public class TweetProcessor {
    protected StateFinder stateFinder;
    protected StateFileReader stateFileReader;

    public TweetProcessor() {
        stateFinder = new StateFinder();
        stateFileReader = new StateFileReader();
        stateFinder.setStateReader(stateFileReader);
        stateFileReader.getAllStates(stateFinder);
    }

    ...
public class TweetProcessor {
    protected StateFinder stateFinder;
    protected StateFileReader stateFileReader;

    public TweetProcessor() {
        stateFinder = new StateFinder();
        stateFileReader = new StateFileReader();
        stateFinder.setStateReader(stateFileReader);
        stateFileReader.getAllStates(stateFinder);
    }

    ...
}

![Diagram of class relationships: TweetProcessor, StateFinder, StateFileReader]
public class TweetProcessor {
    protected StateFinder stateFinder;
    protected StateFileReader stateFileReader;

    public TweetProcessor() {
        stateFinder = new StateFinder();
        stateFileReader = new StateFileReader();
        stateFinder.setStateReader(stateFileReader);
        stateFileReader.getAllStates(stateFinder);
    }

    ...
}
public class TweetProcessor {
    protected StateFinder stateFinder;
    protected StateFileReader stateFileReader;

    public TweetProcessor() {
        stateFinder = new StateFinder();
        stateFileReader = new StateFileReader();
        stateFinder.setStateReader(stateFileReader);
        stateFileReader.getAllStates(stateFinder);
    }

    ...
public class TweetProcessor {
    protected StateFinder stateFinder;
    protected StateFileReader stateFileReader;

    public TweetProcessor() {
        stateFinder = new StateFinder();
        stateFileReader = new StateFileReader();
        stateFinder.setStateReader(stateFileReader);
        stateFileReader.getAllStates(stateFinder);
    }

    
    public class StateFileReader {

        public void getAllStates(StateFinder finder) {
            State[] states = ..
            finder.allStates = states;
        }

        
    }
public class TweetProcessor {
    protected StateFinder stateFinder;
    protected StateFileReader stateFileReader;

    public TweetProcessor() {
        stateFinder = new StateFinder();
        stateFileReader = new StateFileReader();
        stateFinder.setStateReader(stateFileReader);
        stateFileReader.getAllStates(stateFinder);
    }

    . . .
}

public class StateFileReader {

    public void getAllStates(StateFinder finder) {
        State[] states = . . .
        finder.allStates = states;
    }

    . . .
}
Coupling

- Extent to which one module depends on another

- **Loose coupling**
  - one module depends on another
  - depends only on the data/functionality it requires and nothing more

- **Tight coupling**
  - modules depend on each other
  - modules depend on more data/functionality than is minimally required
Tight Coupling: BAD!
Remove Coupling: GOOD!

```
UserInterface
+start() : void
+doMostPopularHashtagForState() : void

TweetProcessor
+getMostPopularHashtagForState(State) : String

TweetFileReader
+getAllTweets() : Tweet[*]
+setFileFormat(string) : void

Tweet
date : Date
username : string
content : string

Location
latitude : double
longitude : double

State
name : string

StateFinder
allStates : State[*]
+determineState(Tweet) : State

stateReader
stateFinder
processor
tweetReader
```
Remove Coupling: GOOD!
Remove Coupling: GOOD!
public class TweetProcessor {
    protected StateFinder stateFinder;
    protected StateFileReader stateFileReader;

    public TweetProcessor() {
        stateFinder = new StateFinder();
        stateFileReader = new StateFileReader();
        stateFinder.setStateReader(stateFileReader);
        stateFileReader.getAllStates(stateFinder);
    }

    ...
}
public class TweetProcessor {
    protected StateFinder stateFinder;
    protected StateFileReader stateFileReader;

    public TweetProcessor() {
        stateFinder = new StateFinder();
        stateFileReader = new StateFileReader();
        stateFinder.setStateReader(stateFileReader);
        stateFileReader.getAllStates(stateFinder);
    }

    ...
}
public class TweetProcessor {
    protected StateFinder stateFinder;

    public TweetProcessor() {
        stateFinder = new StateFinder();
    }
    ...
}
public class TweetProcessor {
    protected StateFinder stateFinder;

    public TweetProcessor() {
        stateFinder = new StateFinder();
    }
    ...
}
public class TweetProcessor {
    protected StateFinder stateFinder;

    public TweetProcessor() {
        stateFinder = new StateFinder();
    }

    ...
}

public class StateFinder {

    protected State[] allStates;
    protected StateFileReader stateReader;

    public StateFinder() {
        stateReader = new StateFileReader();
        allStates = stateReader.getAllStates();
    }

    ...
}
public class TweetProcessor {
    protected StateFinder stateFinder;

    public TweetProcessor() {
        stateFinder = new StateFinder();
    }

    ...
}

public class StateFinder {

    protected State[] allStates;
    protected StateFileReader stateReader;

    public StateFinder() {
        stateReader = new StateFileReader();
        allStates = stateReader.getAllStates();
    }

    ...
}
Modules should be self-contained
Modules should be self-contained
Modules should be self-contained

UserInterface
+start( ) : void
+doMostPopularHashtagForState( ) : void

TweetProcessor
+getMostPopularHashtagForState(State) : String

TweetFileReader
+getAllTweets( ) : Tweet [*]

Tweet
date : Date
username : string
content : string

Location
latitude : double
longitude : double

State
name : string

center

StateFinder
allStates : State[*]
+determineState(Tweet) : State

stateFinder

stateReader

processor
tweetReader
Simplify dependencies

UserInterface
+start() : void
+doMostPopularHashtagForState() : void

TweetProcessor
+getMostPopularHashtagForState(State) : String

TweetFileReader
+getAllTweets() : Tweet[*]

Tweet
date : Date
username : string
content : string

tweetReader

Location
latitude : double
longitude : double

State
name : string

StateFinder
allStates : State[*]
+determineState(Tweet) : State

StateFileReader
+getAllStates() : State[*]

processor

tweetReader

stateFinder

stateReader

location
center
Simplify dependencies

```
UserInterface
+start( ) : void
+doMostPopularHashtagForState( ) : void

TweetProcessor
+getMostPopularHashtagForState(State) : String

TweetFileReader
+getAllTweets( ) : Tweet [*]

Tweet
date : Date
text : string

Location
latitude : double
longitude : double

State
name : string

StateFinder
allStates : State[*]
+determineState(Tweet)

StateReader
stateFinder

StateFileReader
+getAllStates( ) : State[*]
```
Simplify dependencies
Simplify dependencies
Simplify dependencies
Simplify dependencies
Simplify dependencies
Simplify dependencies
Simplify dependencies
Simplify dependencies
A functionally independent design
Recap: Functional Independence

• A module should be able to do its work with minimal dependence on other modules

• Avoid tight coupling (cyclic dependencies)
• Let modules be self-contained
• Simplify dependencies
SD2x3.9

Abstraction

Chris
Software Design Concepts

Modularity

Functional Independence

Abstraction
Software Design Concepts

Modularity

Functional Independence

Abstraction
Motivating Example

• Design a Java program that reads a file containing info about courses, instructors, and enrollment

• And then allows the user to choose to see either:
  • the instructor(s) for a given course
  • the average enrollment for the course
A Modular Design

```
UserInterface
+start() : void
doInstructorsForCourse() : void
doAverageEnrollmentForCourse() : void

processor

CourseEvalProcessor
offerings : ArrayList<ArrayList<String>>
+getInstructorsForCourse(String) : HashSet<String>
+getAverageEnrollmentForCourse(String) : AverageEnrollment

reader

FileReader
fileName : String
+getAllOfferings() : ArrayList<ArrayList<String>>

AverageEnrollment
count : int
average : double
```
A Modular Design

**UserInterface**

- `start() : void`
- `doInstructorsForCourse() : void`
- `doAverageEnrollmentForCourse() : void`

**CourseEvalProcessor**

- `offerings : ArrayList<ArrayList<String>>`
- `getInstructorsForCourse(String) : HashSet<String>`
- `getAverageEnrollmentForCourse(String) : AverageEnrollment`

**AverageEnrollment**

- `count : int`
- `average : double`

**FileReader**

- `fileName : String`
- `getAllOfferings() : ArrayList<ArrayList<String>>`
A Modular Design

UserInterface

+start( ) : void
doInstructorsForCourse( ) : void
doAverageEnrollmentForCourse( ) : void

CourseEvalProcessor

offerings ArrayList<ArrayList<String>>

+getInstructorsForCourse(String) : HashSet<String>
+getAverageEnrollmentForCourse(String) : AverageEnrollment

reader

FileReader

fileName : String

+getAllOfferings( ) : ArrayList<ArrayList<String>>

AverageEnrollment

count : int
average : double
A Modular Design

**UserInterface**

+start() : void
doInstructorsForCourse() : void
doAverageEnrollmentForCourse() : void

**CourseEvalProcessor**

offerings : ArrayList<ArrayList<String>>

+getInstructorsForCourse(String) : HashSet<String>
+getAverageEnrollmentForCourse(String) : AverageEnrollment

**AverageEnrollment**

count : int
average : double

**FileReader**

fileName : String

+getAllOfferings() : ArrayList<ArrayList<String>>
Abstraction

• A module can use other modules with minimal knowledge of the details of their implementation

• Should only care *what* they do (services they provide) and not *how* they do it

• “Program to an interface, not an implementation”
Why abstraction?

• **Stability**: can change a module’s implementation details with no changes required to modules that depend on it

• “**Reusability**”: a module can be reused if it doesn’t need to know the details of the things on which it depends
How can we improve abstraction?

```
UserInterface
+start() : void
doInstructorsForCourse() : void
doAverageEnrollmentForCourse() : void

CourseEvalProcessor
offerings : ArrayList< ArrayList<String> >
+getInstructorsForCourse(String) : HashSet<String>
+getAverageEnrollmentForCourse(String) : AverageEnrollment

AverageEnrollment
count : int
average : double

FileReader
fileName : String
+getAllOfferings() : ArrayList<ArrayList<String>>
```
How can we improve abstraction?

**UserInterface**

+start() : void
+doInstructorsForCourse() : void
+doAverageEnrollmentForCourse() : void

**CourseEvalProcessor**

offerings : ArrayList<ArrayList<String>>

+getInstructorsForCourse(String) : HashSet<String>
+getAverageEnrollmentForCourse(String) : AverageEnrollment

**AverageEnrollment**

count : int
average : double

**FileReader**

fileName : String

+getAllOfferings() : ArrayList<ArrayList<String>>
A more abstract design

```
UserInterface
+start( ) : void
doInstructorsForCourse( ) : void
doAverageEnrollmentForCourse( ) : void

CourseEvalProcessor
offerings : ArrayList<ArrayList<String>>
+getInstructorsForCourse(String) : HashSet<String>
+getAverageEnrollmentForCourse(String) : AverageEnrollment

reader

AverageEnrollment
count : int
average : double

FileReader
fileName : String
+getAllOfferings( ) : ArrayList<ArrayList<String>>
A more abstract design

UserInterface

+start(): void
doInstructorsForCourse(): void
doAverageEnrollmentForCourse(): void

CourseEvalProcessor

offerings: ArrayList<ArrayList<String>>
+getInstructorsForCourse(String): HashSet<String>
+getAverageEnrollmentForCourse(String): AverageEnrollment

AverageEnrollment

count: int
average: double

Reader

+getAllOfferings(): ArrayList<ArrayList<String>>

FileReader

fileName: String
+getAllOfferings(): ArrayList<ArrayList<String>>
A more abstract design

UserInterface

+start() : void
dolnstructorsForCourse() : void
doAverageEnrollmentForCourse() : void

CourseEvalProcessor

offerings : ArrayList< ArrayList<String> >

+getInstructorsForCourse(String) : HashSet<String>
+getAverageEnrollmentForCourse(String) : AverageEnrollment

reader

AverageEnrollment

count : int
average : double

Reader

+getAllOfferings() : ArrayList< ArrayList<String> >

FileReader

fileName : String

+getAllOfferings() : ArrayList< ArrayList<String> >
Abstraction for the Reader class

- CourseEvalProcessor shouldn’t care or know that the Reader reads from a file, e.g.
  - specifying file name
  - catching file I/O exceptions
  - accessing the File or InputStream object
  - calling methods in Reader like “openFile”

- Reader should be designed in such a way that its implementation can change without changing its interface
How can we further improve abstraction?

```
UserInterface
+start() : void
doInstructorsForCourse() : void
doAverageEnrollmentForCourse() : void

CourseEvalProcessor
offerings : ArrayList<ArrayList<String>>
+getInstructorsForCourse(String) : HashSet<String>
+getAverageEnrollmentForCourse(String) : AverageEnrollment

Reader
+getAllOfferings() : ArrayList<ArrayList<String>>

AverageEnrollment
count : int
average : double

FileReader
fileName : String
+getAllOfferings() : ArrayList<ArrayList<String>>
```
How can we further improve abstraction?

- **UserInterface**
  - +start() : void
  - doInstructorsForCourse() : void
  - doAverageEnrollmentForCourse() : void

- **CourseEvalProcessor**
  - offerings : ArrayList<ArrayList<String>>
  - +getInstructorsForCourse(String) : HashSet<String>
  - +getAverageEnrollmentForCourse(String) : AverageEnrollment

- **Reader**
  - +getAllOfferings() : ArrayList<ArrayList<String>>

- **AverageEnrollment**
  - count : int
  - average : double

- **FileReader**
  - fileName : String
  - +getAllOfferings() : ArrayList<ArrayList<String>>
Use custom classes as needed
Use custom classes as needed
Use custom classes as needed
Use custom classes as needed
Use abstract data types classes as needed
Use abstract data types classes as needed
Use abstract data types classes as needed
Use abstract data types classes as needed
Use abstract data types classes as needed
Abstraction in the Java API

- **Interfaces**: a class can have a dependency (field, parameter) on an interface without needing to know which class implements it.

- **Abstract Data Types**: Set, List, Map, etc. in the Java Collections API.

- **Design Patterns**: Observer.
Recap: Abstraction

- A module can use other modules with minimal knowledge of the details of their implementation.
- Should only care what they do (services they provide) and not how they do it.

- "Program to an interface, not an implementation"
- Use custom classes as needed.
- Use abstract data types when possible.
Law of Demeter

Chris
Software Design Concepts

Modularity

Functional Independence

Abstraction
Software Design Concepts

Modularity

Functional Independence

Abstraction
Abstraction

• A module can use other modules with minimal knowledge of the details of their implementation

• Should only care *what* they do (services they provide) and not *how* they do it
public class Game {

    ... 

    protected void updatePlayer(Player player) {
        ... 

        player.getPaintbrush().setColor(Color.RED); 

        ... 

    }

}
public class Game {

    
    
    protected void updatePlayer(Player player) {

        
        player.getPaintbrush().setColor(Color.RED);

        
    
}
public class Game {

    // ...

    protected void updatePlayer(Player player) {
        // ...
        player.getPaintbrush().setColor(Color.RED);
        // ...
    }
}
public class Game {

    
    protected void updatePlayer(Player player) {

        
        
        player.getPaintbrush().setColor(Color.RED);

        
    
}
Law of Demeter

- Also known as the “Principle of Least Knowledge”
- Named after the Greek goddess of agriculture
- “A module should be able to perform its tasks with (minimal) knowledge of related modules, and with no knowledge of the modules related to those”
- “Don’t talk to strangers”
Law of Demeter

- A method $M$ in object $O$ may only invoke methods on the following:
  - the object $O$ itself
  - parameters to $M$
  - objects created in $M$
  - $O$’s fields
  - global variables
public class Game {

    // ...

    protected void updatePlayer(Player player) {
        // ...
        player.getPaintbrush().setColor(Color.RED);

        // ...
    }

    // ...
}
public class Game {

    ...

    protected void updatePlayer(Player player) {
        ...

        player.getPaintbrush().setColor(Color.RED);

        ...
    }
public class Game {

    ... 

    protected void updatePlayer(Player player) {

        ... 

        player.getPaintbrush().setColor(Color.RED);
    
    ... 

}
public class Game {

    . . .

    protected void updatePlayer(Player player) {
        . . .

        player.getPaintbrush().setColor(Color.RED);
        . . .
    }
}
public class Game {

    ... 

    protected void updatePlayer(Player player) {

        ... 

        Paintbrush brush = player.getPaintbrush();
        brush.setColor(Color.RED);
        ...
public class Game {

    ...

    protected void updatePlayer(Player player) {

        ...

        Paintbrush brush = player.getPaintbrush();
        brush.setColor(Color.RED);
        ...
    }
public class Game {

    ... 

    protected void updatePlayer(Player player) {

        ... 

        Paintbrush brush = player.getPaintbrush();
        brush.setColor(Color.RED);
        ...
public class Game {

    . . .

    protected void updatePlayer(Player player) {

        . . .

        player.getPaintbrush().setColor(Color.RED);

        . . .

    }

}
public class Game {

    ... 

    protected void updatePlayer(Player player) {

        ... 

        player.getPaintbrush().setColor(Color.RED);

        ... 

    }

}
public class Game {

    ...  

    protected void updatePlayer(Player player) {
        ...  

        player.getPaintbrush().setColor(Color.RED);

        ...  
    }

    ...  
}
Solution: Hide Delegate
Game

Player
+getPaintbrush() : Paintbrush

Paintbrush
+setColor(Color) : void

Color
public class Player {
    
    private Paintbrush paintbrush;
    public void setColor(Color color) {
        paintbrush.setColor(color);
    }
    
    . . .
}
public class Player {
    . . .
    private Paintbrush paintbrush;
    public void setColor(Color color) {
        paintbrush.setColor(color);
    }
    . . .
}
public class Player {
   . . .
   private Paintbrush paintbrush;
   public void setColor(Color color) {
      paintbrush.setColor(color);
   }
   . . .
}
public class Player {
    ..
    private Paintbrush paintbrush;
    public void setColor(Color color) {
        paintbrush.setColor(color);
    }
    ..
}

public class Game {
    ..
    protected void updatePlayer(Player player) {
        ..
        player.getPaintbrush().setColor(Color.RED);
        ..
    }
}
public class Player {
    // ...
    private Paintbrush paintbrush;
    public void setColor(Color color) {
        paintbrush.setColor(color);
    }
    // ...
}

public class Game {
    // ...
    protected void updatePlayer(Player player) {
        // ...
        player.getPaintbrush().setColor(Color.RED);
        // ...
    }
    // ...
}
public class Game {

    protected void updatePlayer(Player player) {
        player.setColor(Color.RED);
    }
}

public class Player {

    private Paintbrush paintbrush;
    public void setColor(Color color) {
        paintbrush.setColor(color);
    }
}

public class Game {

    protected void updatePlayer(Player player) {
        player.setColor(Color.RED);
    }
}

public class Player {

    private Paintbrush paintbrush;
    public void setColor(Color color) {
        paintbrush.setColor(color);
    }
}

public class Game {

    protected void updatePlayer(Player player) {
        player.setColor(Color.RED);
    }
}
public class Player {
    private Paintbrush paintbrush;
    public void setColor(Color color) {
        paintbrush.setColor(color);
    }
}

public class Game {
    protected void updatePlayer(Player player) {
        player.setColor(Color.RED);
    }
}
public class Player {
    private Paintbrush paintbrush;
    public void setColor(Color color) {
        paintbrush.setColor(color);
    }
}

public class Game {
    protected void updatePlayer(Player player) {
        player.setColor(Color.RED);
    }
}
public class Player {
    
    private Paintbrush paintbrush;
    public void setColor(Color color) {
        paintbrush.alterColor(color);
    }
    
}

public class Game {
    
    protected void updatePlayer(Player player) {
        
        player.setColor(Color.RED);
    }
    
}
Recap

- Try to avoid **Message Chains** in your code

- Be careful about violating the **Law of Demeter**

- Use the **Hide Delegate** pattern

- This will help improve **abstraction**
Recap: Software Design

- **Internal Quality:** Changeability, Stability, Analyzability, Testability

- **Design Concepts:** Modularity, Functional Independence, Abstraction