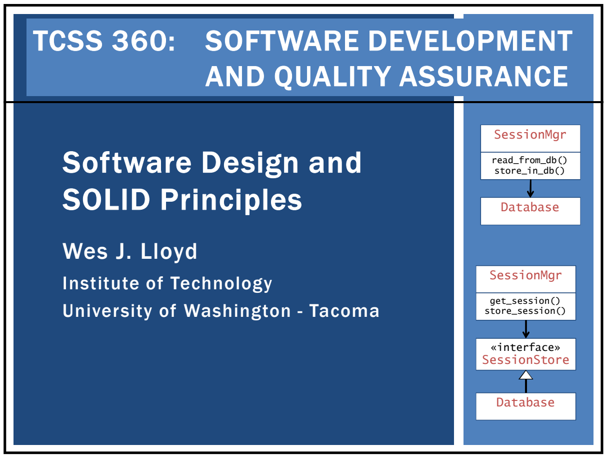
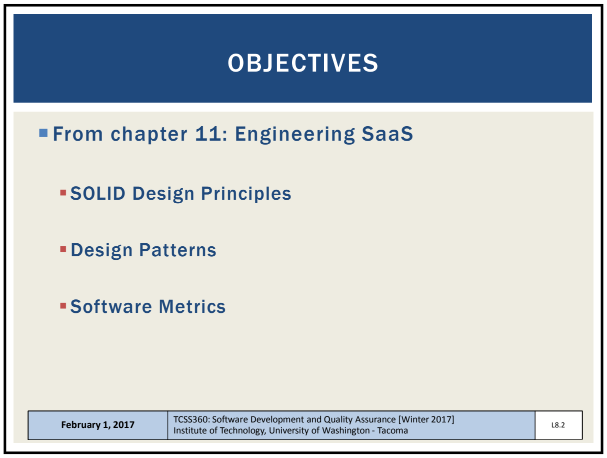


TCSS 360: SOFTWARE DEVELOPMENT AND QUALITY ASSURANCE

Software Design and SOLID Principles

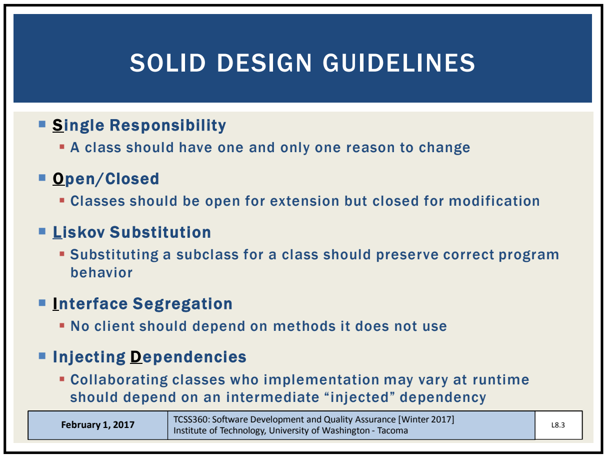
Wes J. Lloyd
Institute of Technology
University of Washington - Tacoma

SessionMgr
read_from_db()
store_in_db()
Database
SessionMgr
get_session()
store_session()
«interface»
SessionStore
Database



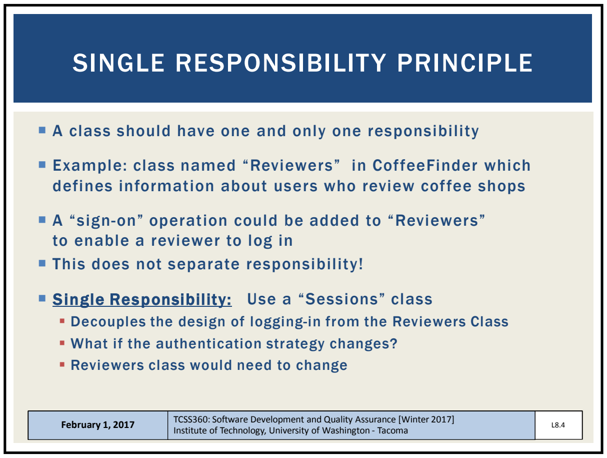
OBJECTIVES

- From chapter 11: Engineering SaaS
  - SOLID Design Principles
  - Design Patterns
  - Software Metrics



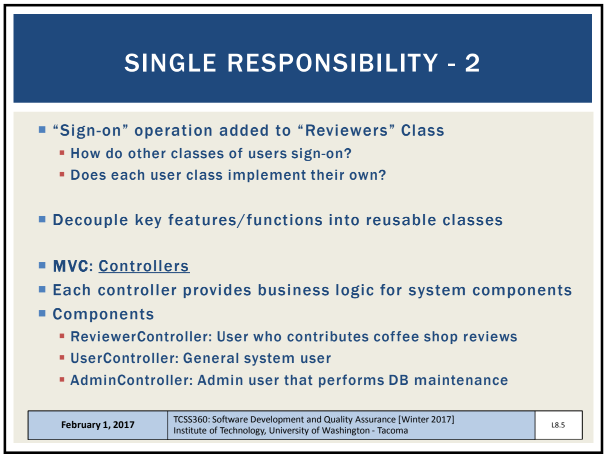
SOLID DESIGN GUIDELINES

- **Single Responsibility**
  - A class should have one and only one reason to change
- **Open/Closed**
  - Classes should be open for extension but closed for modification
- **Liskov Substitution**
  - Substituting a subclass for a class should preserve correct program behavior
- **Interface Segregation**
  - No client should depend on methods it does not use
- **Injecting Dependencies**
  - Collaborating classes who implementation may vary at runtime should depend on an intermediate "injected" dependency



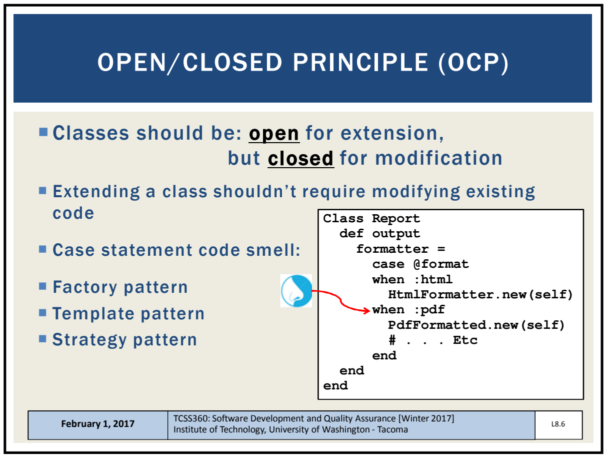
SINGLE RESPONSIBILITY PRINCIPLE

- A class should have one and only one responsibility
- Example: class named "Reviewers" in CoffeeFinder which defines information about users who review coffee shops
- A "sign-on" operation could be added to "Reviewers" to enable a reviewer to log in
- This does not separate responsibility!
- **Single Responsibility:** Use a "Sessions" class
  - Decouples the design of logging-in from the Reviewers Class
  - What if the authentication strategy changes?
  - Reviewers class would need to change



SINGLE RESPONSIBILITY - 2

- "Sign-on" operation added to "Reviewers" Class
  - How do other classes of users sign-on?
  - Does each user class implement their own?
- Decouple key features/functions into reusable classes
- **MVC:** Controllers
- Each controller provides business logic for system components
- Components
  - ReviewerController: User who contributes coffee shop reviews
  - UserController: General system user
  - AdminController: Admin user that performs DB maintenance



OPEN/CLOSED PRINCIPLE (OCP)

- Classes should be: **open** for extension, but **closed** for modification
- Extending a class shouldn't require modifying existing code
- Case statement code smell:
- Factory pattern
- Template pattern
- Strategy pattern

```
Class Report
  def output
    formatter =
      case @format
      when :html
        HtmlFormatter.new(self)
      when :pdf
        PdfFormatted.new(self)
        # . . . Etc
      end
  end
end
```

LISKOV SUBSTITUTION PRINCIPLE (LSP)

- Class subtypes can substitute for base types
- Current formulation attributed to (Turing Award winner) Barbara Liskov

"A method that works on an instance of *type T*, should also work on any *subtype of T*"

Type/subtype != Class/subclass
All of T's subtypes should preserve T's contract…

INTERFACE SEGREGATION PRINCIPLE (ISP)

- Clients should not be forced to depend on methods they do not use…
- Split large interfaces into smaller, more specific ones
- ISP reduces coupling
- High code coupling is correlates with higher software maintenance costs
  - Code is harder to modify, refactor, extend

ISP: COUPLING SUMMARY

- Coupling measures dependencies between subsystems
- High coupling: changes to one subsystem will have high impact on the other subsystem – BAD!!
  - Require change of model, massive compilation
- Low coupling: change in one subsystem does not affect any other subsystem - - GOOD!!

DEPENDENCY INVERSION PRINCIPLE (DIP)

- Also called **dependency injection**…
- If two classes depend on each other, but their implementations may change, it is better if they depend on an abstract interface that is "**injected**" dynamically
- Enables interface to change with changing original class
- Code is not statically bound to the external dependency

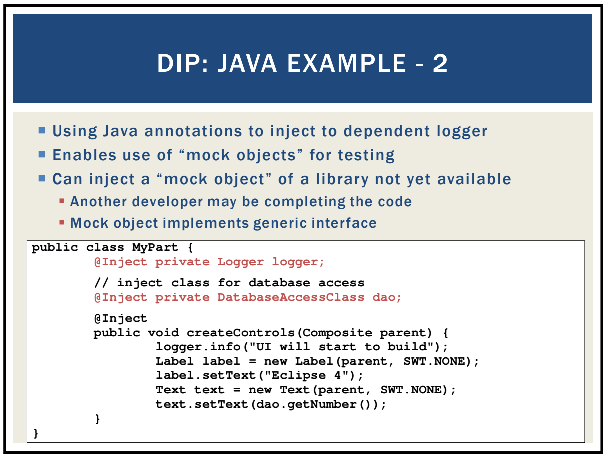
DIP: EXAMPLE

- **Example**: one class (user code), makes use of a 3rd party library or framework (e.g. logging API)
- Without dependency injection, the user class is dependent (coupled) to the 3rd party library or framework
- "Coupling" becomes pandemic throughout the code
- It's everywhere…
- If the 3rd party library goes defunct (company or group disbands), program code is now dependent on an unsupported library
- Solution: Inject an abstract logging interface (which a 3rd party library or framework implements)

DIP: JAVA EXAMPLE

- Traditional coupling to logging class (API)
- Program must have access to a specific 3rd party library

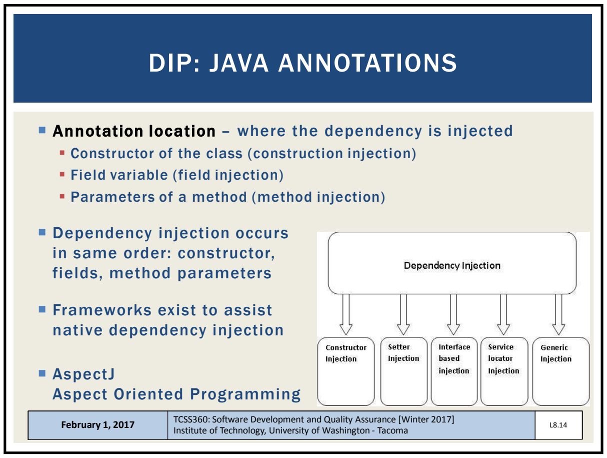
```
package com.example.e4.rcp.todo.parts;

import java.util.logging.Logger;

public class MyClass {

        private final static Logger logger;

        public MyClass(Logger logger) {
                this.logger = logger;
                // write an info log message
                logger.info("This is a log message.")
        }
}
```



DIP: JAVA EXAMPLE - 2

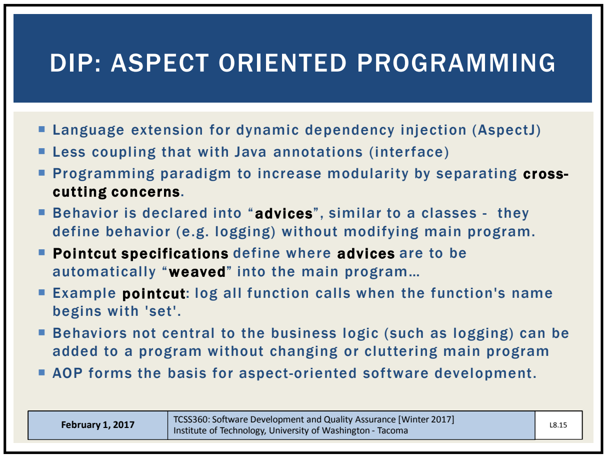
- Using Java annotations to inject to dependent logger
- Enables use of "mock objects" for testing
- Can inject a "mock object" of a library not yet available
  - Another developer may be completing the code
  - Mock object implements generic interface

```
public class MyPart {
        @Inject private Logger logger;

        // inject class for database access
        @Inject private DatabaseAccessClass dao;

        @Inject
        public void createControls(Composite parent) {
                logger.info("UI will start to build");
                Label label = new Label(parent, SWT.NONE);
                label.setText("Eclipse 4");
                Text text = new Text(parent, SWT.NONE);
                text.setText(dao.getNumber());
        }
}
```



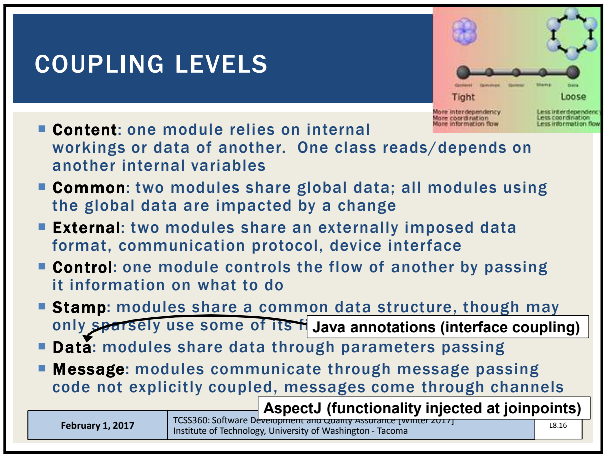
DIP: JAVA ANNOTATIONS

- **Annotation location** – where the dependency is injected
  - Constructor of the class (construction injection)
  - Field variable (field injection)
  - Parameters of a method (method injection)
- Dependency injection occurs in same order: constructor, fields, method parameters
- Frameworks exist to assist native dependency injection
- AspectJ
  Aspect Oriented Programming



DIP: ASPECT ORIENTED PROGRAMMING

- Language extension for dynamic dependency injection (AspectJ)
- Less coupling that with Java annotations (interface)
- Programming paradigm to increase modularity by separating **cross-cutting concerns**.
- Behavior is declared into "**advices**", similar to a classes - they define behavior (e.g. logging) without modifying main program.
- **Pointcut specifications** define where **advices** are to be automatically "**weaved**" into the main program…
- Example **pointcut**: log all function calls when the function's name begins with 'set'.
- Behaviors not central to the business logic (such as logging) can be added to a program without changing or cluttering main program
- AOP forms the basis for aspect-oriented software development.



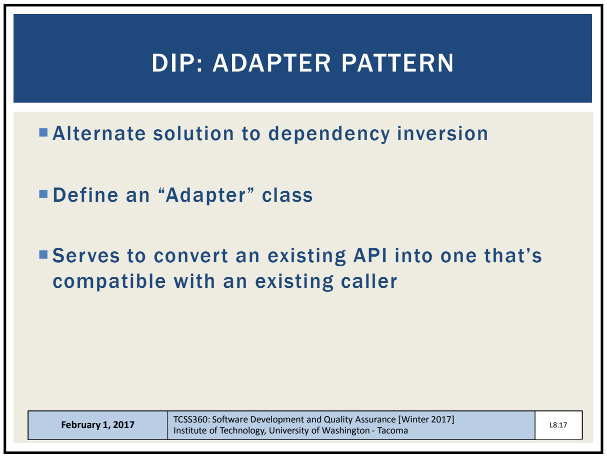
COUPLING LEVELS

Content
Common
Control
Stamp
Data
Tight
Loose
More interdependency
More coordination
More information flow
Less interdependency
Less coordination
Less information flow

- **Content**: one module relies on internal workings or data of another. One class reads/depends on another internal variables
- **Common**: two modules share global data; all modules using the global data are impacted by a change
- **External**: two modules share an externally imposed data format, communication protocol, device interface
- **Control**: one module controls the flow of another by passing it information on what to do
- **Stamp**: modules share a common data structure, though may only sparsely use some of its f
- **Data**: modules share data through parameters passing
- **Message**: modules communicate through message passing code not explicitly coupled, messages come through channels

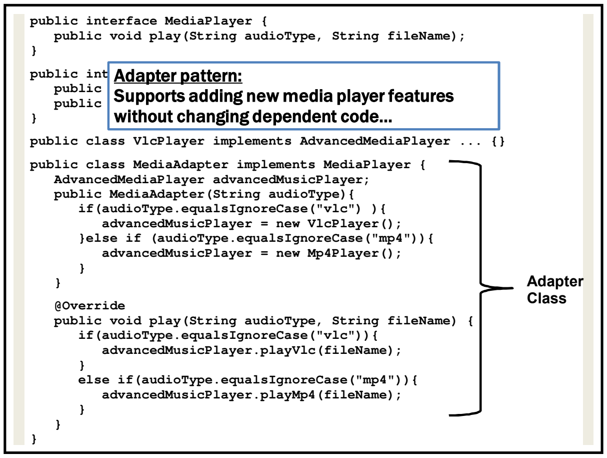
Java annotations (interface coupling)

AspectJ (functionality injected at joinpoints)



DIP: ADAPTER PATTERN

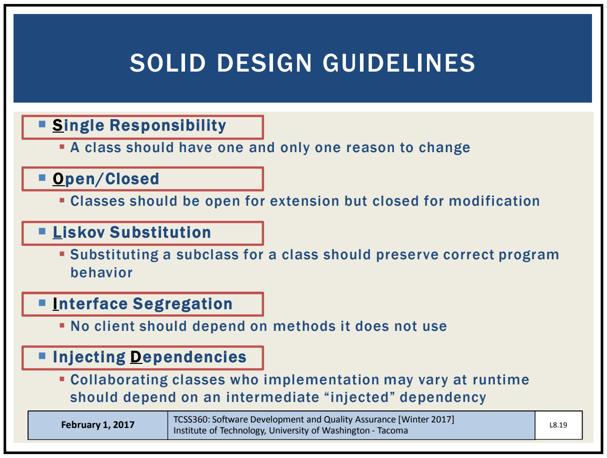
- Alternate solution to dependency inversion
- Define an "Adapter" class
- Serves to convert an existing API into one that's compatible with an existing caller



```
public interface MediaPlayer {
   public void play(String audioType, String fileName);
}

public int
   public
   public
}
public class VlcPlayer implements AdvancedMediaPlayer ... {}

public class MediaAdapter implements MediaPlayer {
   AdvancedMediaPlayer advancedMusicPlayer;
   public MediaAdapter(String audioType){
      if(audioType.equalsIgnoreCase("vlc") ){
         advancedMusicPlayer = new VlcPlayer();
      }else if (audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer = new Mp4Player();
      }
   }

   @Override
   public void play(String audioType, String fileName) {
      if(audioType.equalsIgnoreCase("vlc")){
         advancedMusicPlayer.playVlc(fileName);
      }
      else if(audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer.playMp4(fileName);
      }
   }
}
```

Adapter pattern:
Supports adding new media player features without changing dependent code…

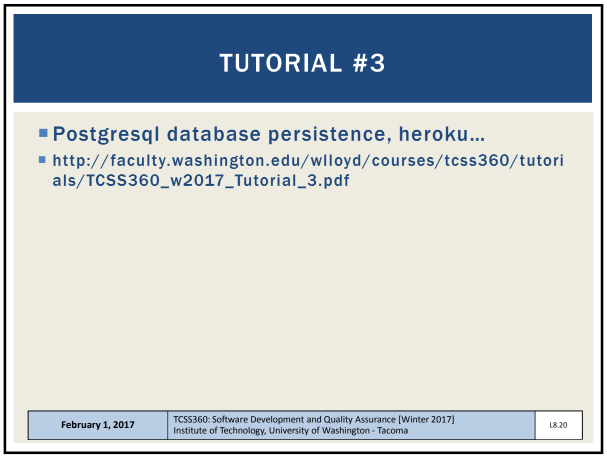
Adapter Class



SOLID DESIGN GUIDELINES

- Single Responsibility
  - A class should have one and only one reason to change
- Open/Closed
  - Classes should be open for extension but closed for modification
- Liskov Substitution
  - Substituting a subclass for a class should preserve correct program behavior
- Interface Segregation
  - No client should depend on methods it does not use
- Injecting Dependencies
  - Collaborating classes who implementation may vary at runtime should depend on an intermediate "injected" dependency

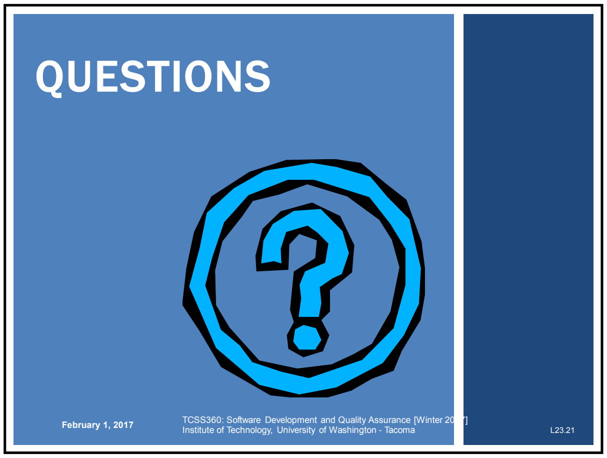
February 1, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017]
Institute of Technology, University of Washington - Tacoma | L8.19



TUTORIAL #3

- Postgresql database persistence, heroku…
- http://faculty.washington.edu/wlloyd/courses/tcss360/tutorials/TCSS360_w2017_Tutorial_3.pdf

February 1, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017]
Institute of Technology, University of Washington - Tacoma | L8.20



QUESTIONS

February 1, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017]
Institute of Technology, University of Washington - Tacoma | L23.21