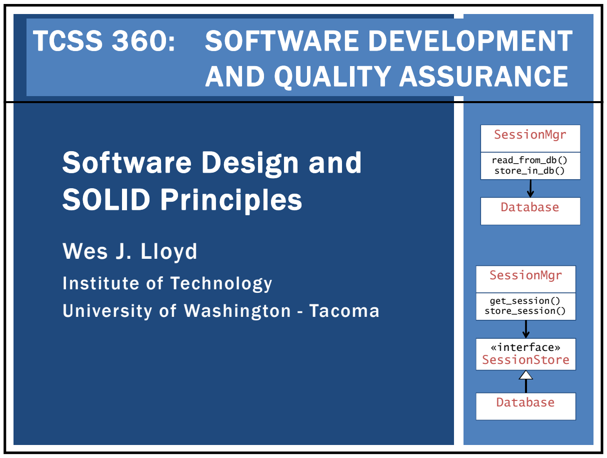
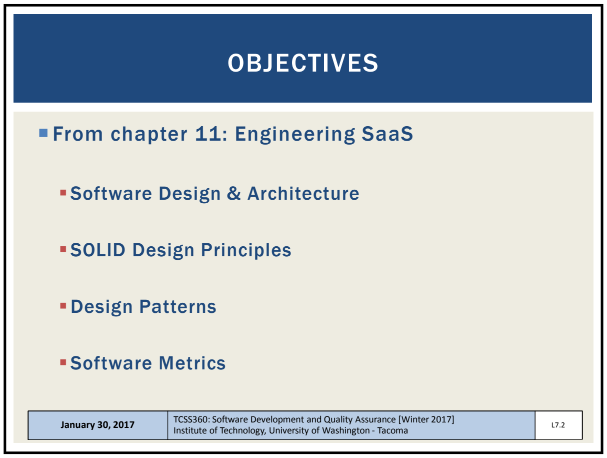


TCSS 360: SOFTWARE DEVELOPMENT AND QUALITY ASSURANCE

Software Design and SOLID Principles

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

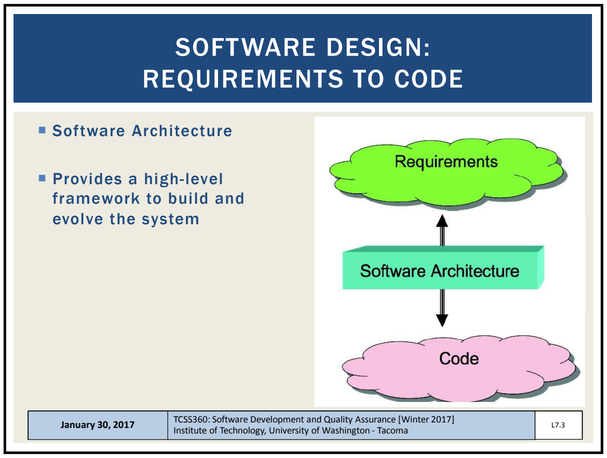
SessionMgr
read_from_db()
store_in_db()
Database
SessionMgr
get_session()
store_session()
«interface»
SessionStore
Database



OBJECTIVES

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

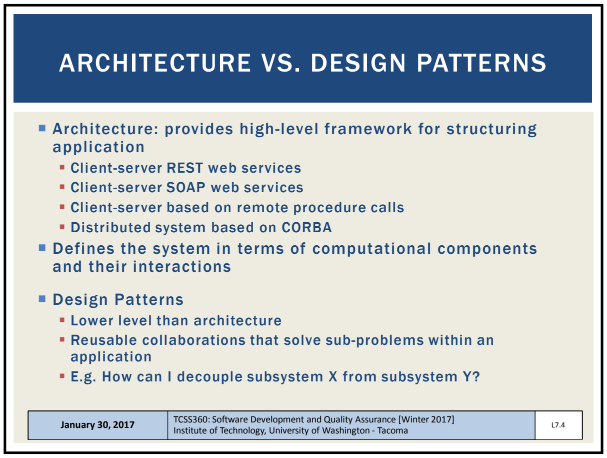
January 30, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017] Institute of Technology, University of Washington - Tacoma | L7.2



SOFTWARE DESIGN: REQUIREMENTS TO CODE

- Software Architecture
- Provides a high-level framework to build and evolve the system

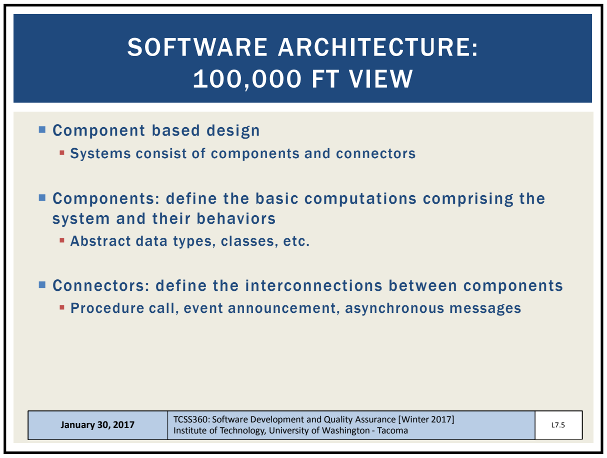
January 30, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017] Institute of Technology, University of Washington - Tacoma | L7.3



ARCHITECTURE VS. DESIGN PATTERNS

- Architecture: provides high-level framework for structuring application
  - Client-server REST web services
  - Client-server SOAP web services
  - Client-server based on remote procedure calls
  - Distributed system based on CORBA
- Defines the system in terms of computational components and their interactions
- Design Patterns
  - Lower level than architecture
  - Reusable collaborations that solve sub-problems within an application
  - E.g. How can I decouple subsystem X from subsystem Y?

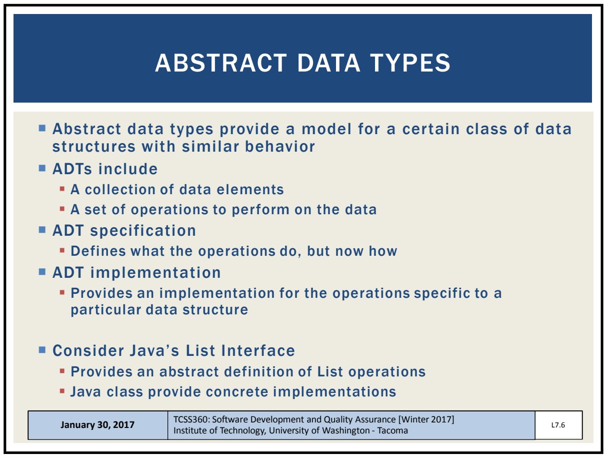
January 30, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017] Institute of Technology, University of Washington - Tacoma | L7.4



SOFTWARE ARCHITECTURE: 100,000 FT VIEW

- Component based design
  - Systems consist of components and connectors
- Components: define the basic computations comprising the system and their behaviors
  - Abstract data types, classes, etc.
- Connectors: define the interconnections between components
  - Procedure call, event announcement, asynchronous messages

January 30, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017] Institute of Technology, University of Washington - Tacoma | L7.5



ABSTRACT DATA TYPES

- Abstract data types provide a model for a certain class of data structures with similar behavior
- ADTs include
  - A collection of data elements
  - A set of operations to perform on the data
- ADT specification
  - Defines what the operations do, but now how
- ADT implementation
  - Provides an implementation for the operations specific to a particular data structure
- Consider Java's List Interface
  - Provides an abstract definition of List operations
  - Java class provide concrete implementations

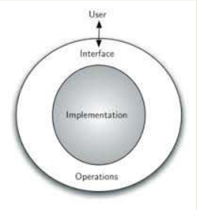
January 30, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017] Institute of Technology, University of Washington - Tacoma | L7.6

ABSTRACTION

- Define the operations
- Define the data
- Provide an implementation

- Why would we like to abstract the definition of our operations?

- Common interface
- Many implementations
- Backward compatibility: introduce new interfaces while retaining support for old…



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

SINGLE RESPONSIBILITY - 3

- MVC: Each controller should specialize in dealing with one resource
- User session is a distinct resource from Reviewer

- Rule of thumb: if you cannot describe the responsibility of the class in 25 words or less, it may have more than one responsibility

- Provides a gauge for when to split into multiple classes

SRP: COHESION METRICS - 1

- Measuring abuse: Lack of Cohesion Metrics - **(LCOM)**
  - Degree to which the elements of a class are related
  - Methods are related if they access the same subset of instance or class variables – or if one calls the other
  - Detects unrelated clusters within a class
  - "Data clump" code smell: when a class is evolving towards multiple responsibilities
    - Group of variables/values passed and returned together
    - Could values benefit from their own class?
  - CKJM Java metrics (Free Tool): http://www.spinellis.gr/sw/ckjm/

SRP: COHESION METRICS - 2

- Revised Henderson-Sellers $LCOM=1-(sum(MV_i) / M*V)$ (produces value from 0 to 1)
- M = # instance methods
- V = # instance variables
- MV_i = # instance methods that access the i'th instance variable (excluding "trivial" getters/setters)

- LCOM-4: counts # of connected components in graph where related methods are connected by an edge

- High LCOM suggests possible *single responsibility* violation

SRP: COHESION SUMMARY

- Cohesion measures the degree of dependence among classes/modules in a system

  - High cohesion: Classes/modules in the program perform similar tasks and are related to each other (via associations) GOOD !

  - Low cohesion: Lots of miscellaneous and auxiliary classes/modules, no associations BAD !

SRP: REFACTORING TO SOLVE

- Reviewer class:
  - Attribute: phone_number
  - Attribute: zipcode

"Extract Class" Refactoring:

Extract new class(es) from the Reviewer

Could refactor as an Address Class, or separate zipcode and phone number

variables of Reviewer class

- Zipcode and phone number could be separate classes
- Overtime the number of "support" methods tends to grow

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:
- Explicit dispatch based on the report format
- Adding a new output type requires modifying Report.output method

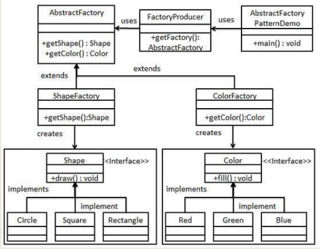
```
Class Report
  def output
    formatter =
      case @format
      when :html
        HtmlFormatter.new(self)
      when :pdf
        PdfFormatted.new(self)
        # . . . Etc
      end
  end
end
```

OPEN/CLOSED PRINCIPLE - 2

- Abstract factory design pattern

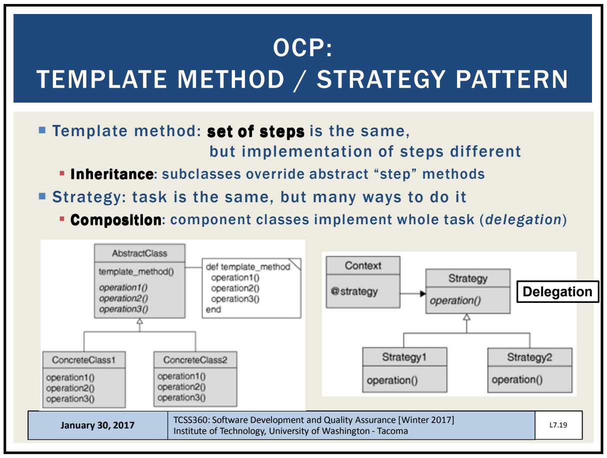
- Provides a solution in statically typed languages

- Provides common interface for instantiating an object whose subclass may not be known until runtime



Non-elegant solution: factory must be changed

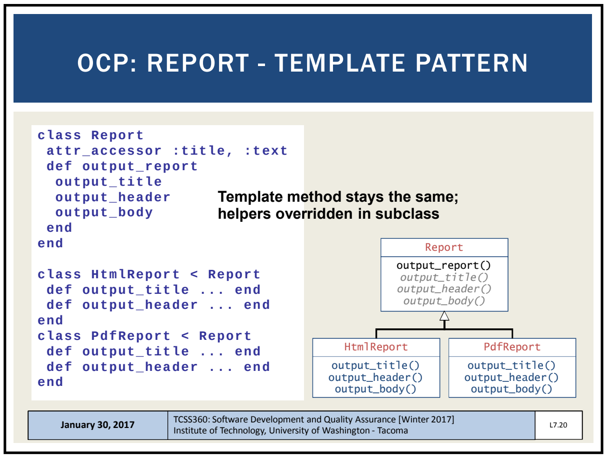
```
public abstract class AbstractFactory {
   abstract Color getColor(String color);
   abstract Shape getShape(String shape) ;
}

public class ShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();

      }else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();

      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }
   @Override
   Color getColor(String color) {
      return null;
   }
}
```



OCP: TEMPLATE METHOD / STRATEGY PATTERN

- Template method: **set of steps** is the same, but implementation of steps different
  - **Inheritance**: subclasses override abstract "step" methods
- Strategy: task is the same, but many ways to do it
  - **Composition**: component classes implement whole task (*delegation*)

January 30, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017] Institute of Technology, University of Washington - Tacoma | L7.19

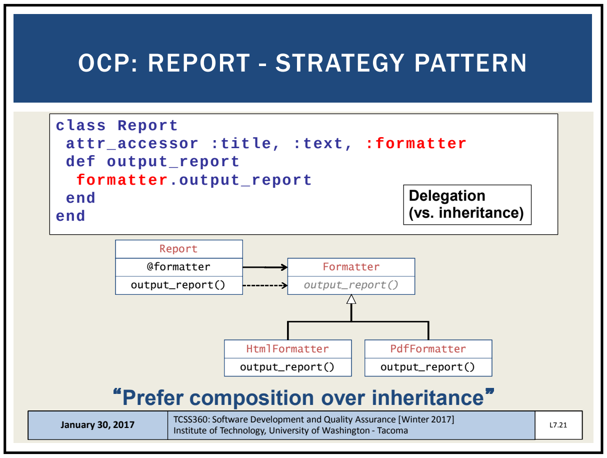


OCP: REPORT - TEMPLATE PATTERN

```
class Report
 attr_accessor :title, :text
 def output_report
  output_title
  output_header
  output_body
 end
end

class HtmlReport < Report
 def output_title ... end
 def output_header ... end
end
class PdfReport < Report
 def output_title ... end
 def output_header ... end
end
```

Template method stays the same; helpers overridden in subclass

January 30, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017] Institute of Technology, University of Washington - Tacoma | L7.20



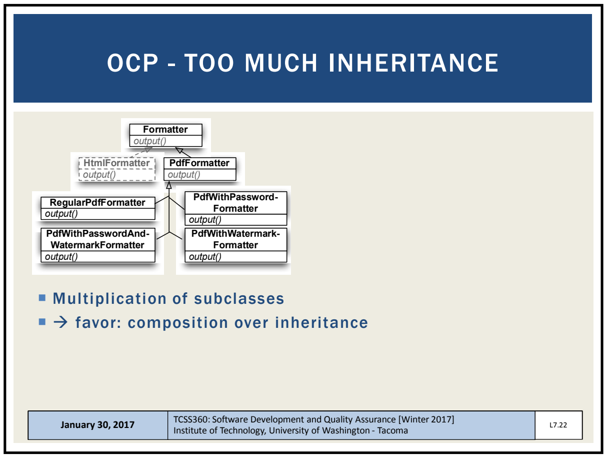
OCP: REPORT - STRATEGY PATTERN

```
class Report
 attr_accessor :title, :text, :formatter
 def output_report
  formatter.output_report
 end
end
```

Delegation (vs. inheritance)

"Prefer composition over inheritance"

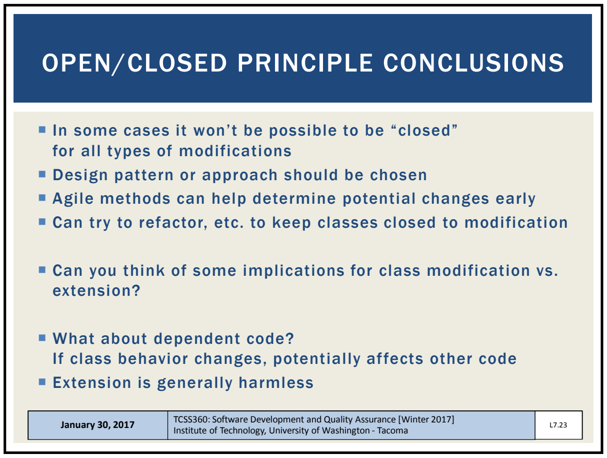
January 30, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017] Institute of Technology, University of Washington - Tacoma | L7.21



OCP - TOO MUCH INHERITANCE

- Multiplication of subclasses
- → favor: composition over inheritance

January 30, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017] Institute of Technology, University of Washington - Tacoma | L7.22



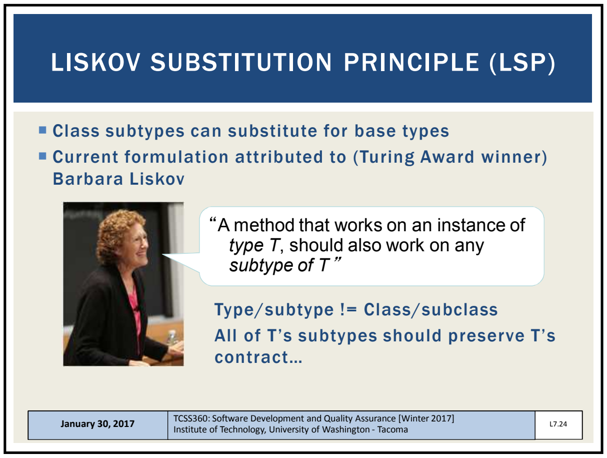
OPEN/CLOSED PRINCIPLE CONCLUSIONS

- In some cases it won't be possible to be "closed" for all types of modifications
- Design pattern or approach should be chosen
- Agile methods can help determine potential changes early
- Can try to refactor, etc. to keep classes closed to modification

- Can you think of some implications for class modification vs. extension?

- What about dependent code? If class behavior changes, potentially affects other code
- Extension is generally harmless

January 30, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017] Institute of Technology, University of Washington - Tacoma | L7.23



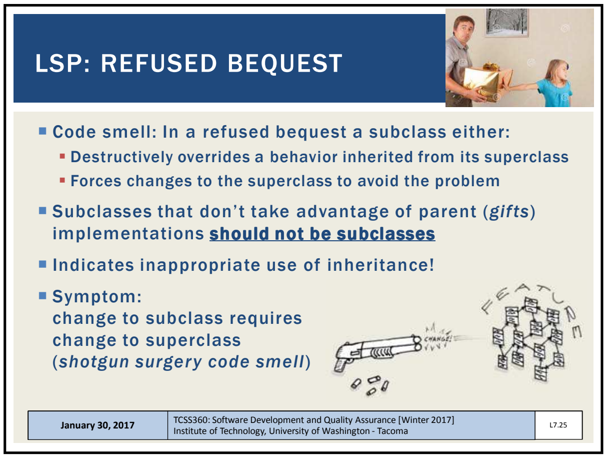
LISKOV SUBSTITUTION PRINCIPLE (LSP)

- 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…

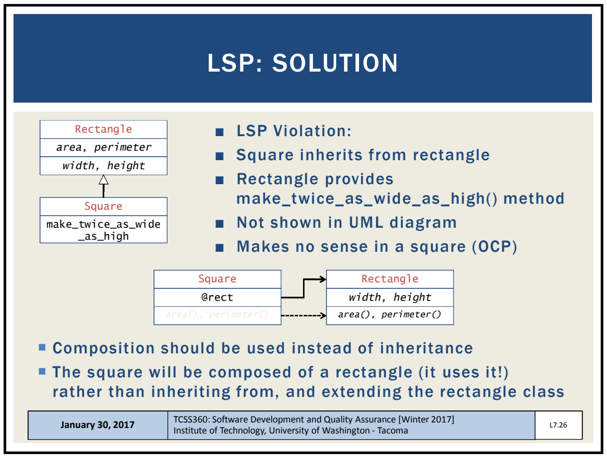
January 30, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017] Institute of Technology, University of Washington - Tacoma | L7.24



LSP: REFUSED BEQUEST

- Code smell: In a refused bequest a subclass either:
  - Destructively overrides a behavior inherited from its superclass
  - Forces changes to the superclass to avoid the problem
- Subclasses that don't take advantage of parent (*gifts*) implementations **should not be subclasses**
- Indicates inappropriate use of inheritance!
- Symptom:
  change to subclass requires change to superclass
  (*shotgun surgery code smell*)

January 30, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017] Institute of Technology, University of Washington - Tacoma | L7.25



LSP: SOLUTION

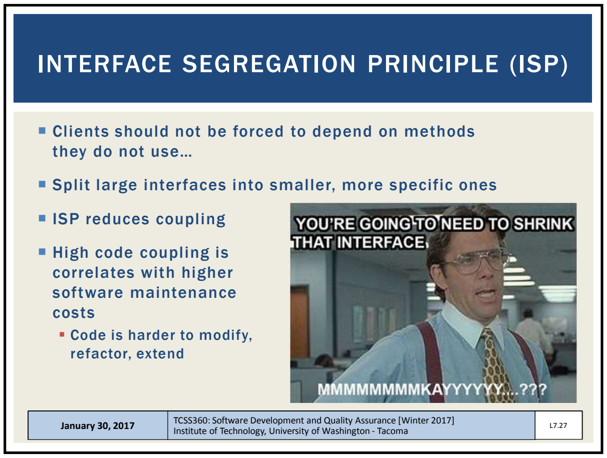
Rectangle
area, perimeter
width, height
Square
make_twice_as_wide_as_high

- LSP Violation:
- Square inherits from rectangle
- Rectangle provides make_twice_as_wide_as_high() method
- Not shown in UML diagram
- Makes no sense in a square (OCP)

Square
@rect
area(), perimeter()
Rectangle
width, height
area(), perimeter()

- Composition should be used instead of inheritance
- The square will be composed of a rectangle (it uses it!) rather than inheriting from, and extending the rectangle class

January 30, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017] Institute of Technology, University of Washington - Tacoma | L7.26

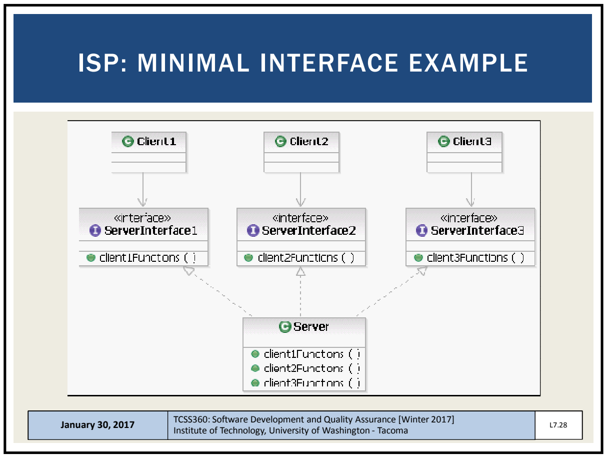


INTERFACE SEGREGATION PRINCIPLE (ISP)

- 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

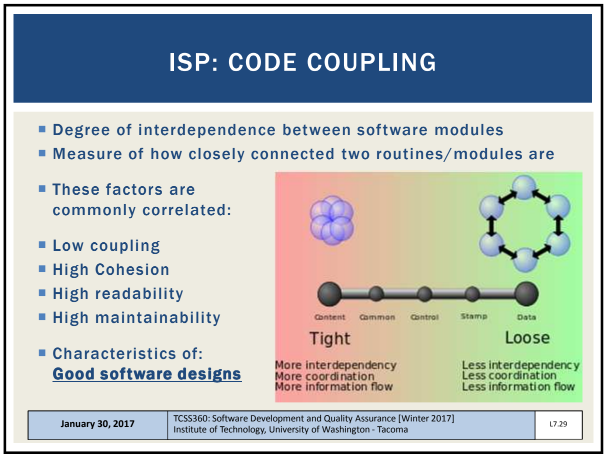
YOU'RE GOING TO NEED TO SHRINK THAT INTERFACE
MMMMMMMMKAYYYYYY…???

January 30, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017] Institute of Technology, University of Washington - Tacoma | L7.27



ISP: MINIMAL INTERFACE EXAMPLE

January 30, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017] Institute of Technology, University of Washington - Tacoma | L7.28

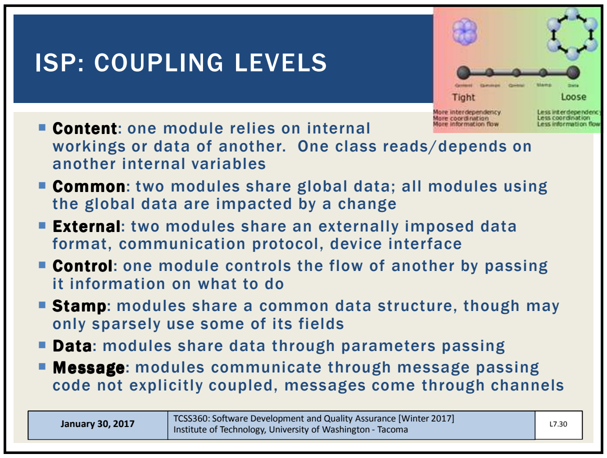


ISP: CODE COUPLING

- Degree of interdependence between software modules
- Measure of how closely connected two routines/modules are
- These factors are commonly correlated:
- Low coupling
- High Cohesion
- High readability
- High maintainability
- Characteristics of: **Good software designs**

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

January 30, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017] Institute of Technology, University of Washington - Tacoma | L7.29

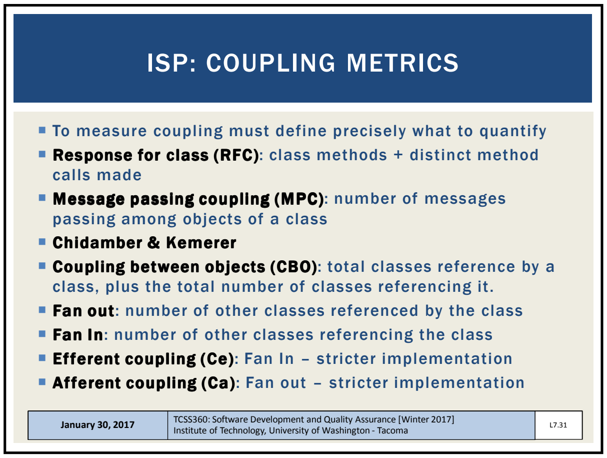


ISP: COUPLING LEVELS

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 fields
- **Data**: modules share data through parameters passing
- **Message**: modules communicate through message passing code not explicitly coupled, messages come through channels

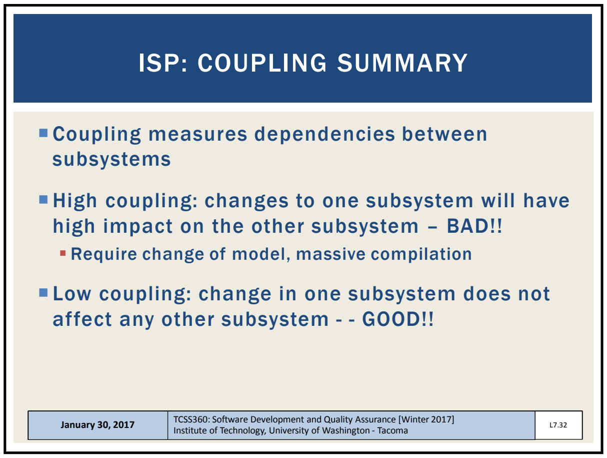
January 30, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017] Institute of Technology, University of Washington - Tacoma | L7.30



ISP: COUPLING METRICS

- To measure coupling must define precisely what to quantify
- **Response for class (RFC)**: class methods + distinct method calls made
- **Message passing coupling (MPC)**: number of messages passing among objects of a class
- **Chidamber & Kemerer**
- **Coupling between objects (CBO)**: total classes reference by a class, plus the total number of classes referencing it.
- **Fan out**: number of other classes referenced by the class
- **Fan In**: number of other classes referencing the class
- **Efferent coupling (Ce)**: Fan In – stricter implementation
- **Afferent coupling (Ca)**: Fan out – stricter implementation

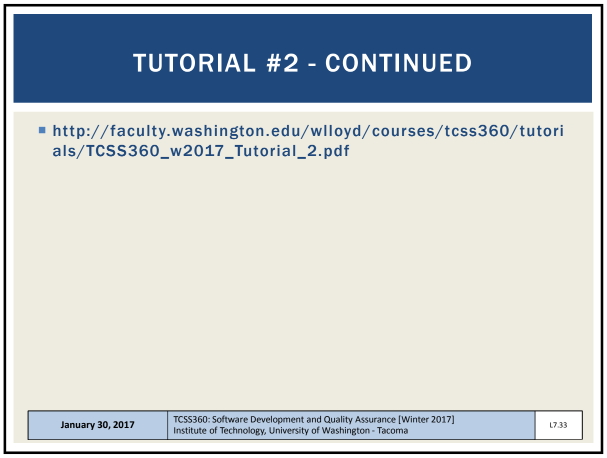
January 30, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017]
Institute of Technology, University of Washington - Tacoma | L7.31



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!!

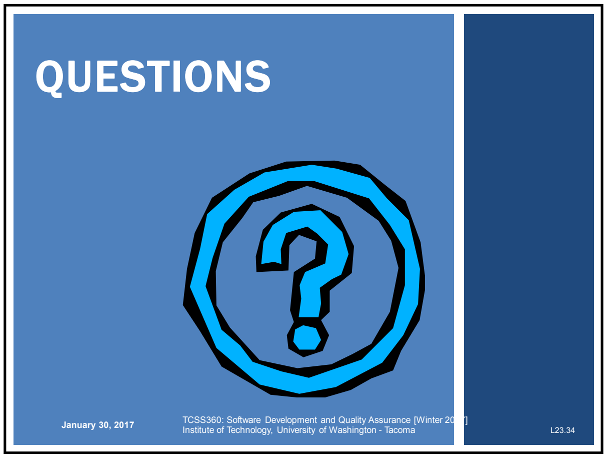
January 30, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017]
Institute of Technology, University of Washington - Tacoma | L7.32



TUTORIAL #2 - CONTINUED

- http://faculty.washington.edu/wlloyd/courses/tcss360/tutorials/TCSS360_w2017_Tutorial_2.pdf

January 30, 2017 | TCSS360: Software Development and Quality Assurance [Winter 2017]
Institute of Technology, University of Washington - Tacoma | L7.33



QUESTIONS

January 30, 2017 | TCSS360: Software Development and Quality Assurance [Winter 20]
Institute of Technology, University of Washington - Tacoma | L23.34