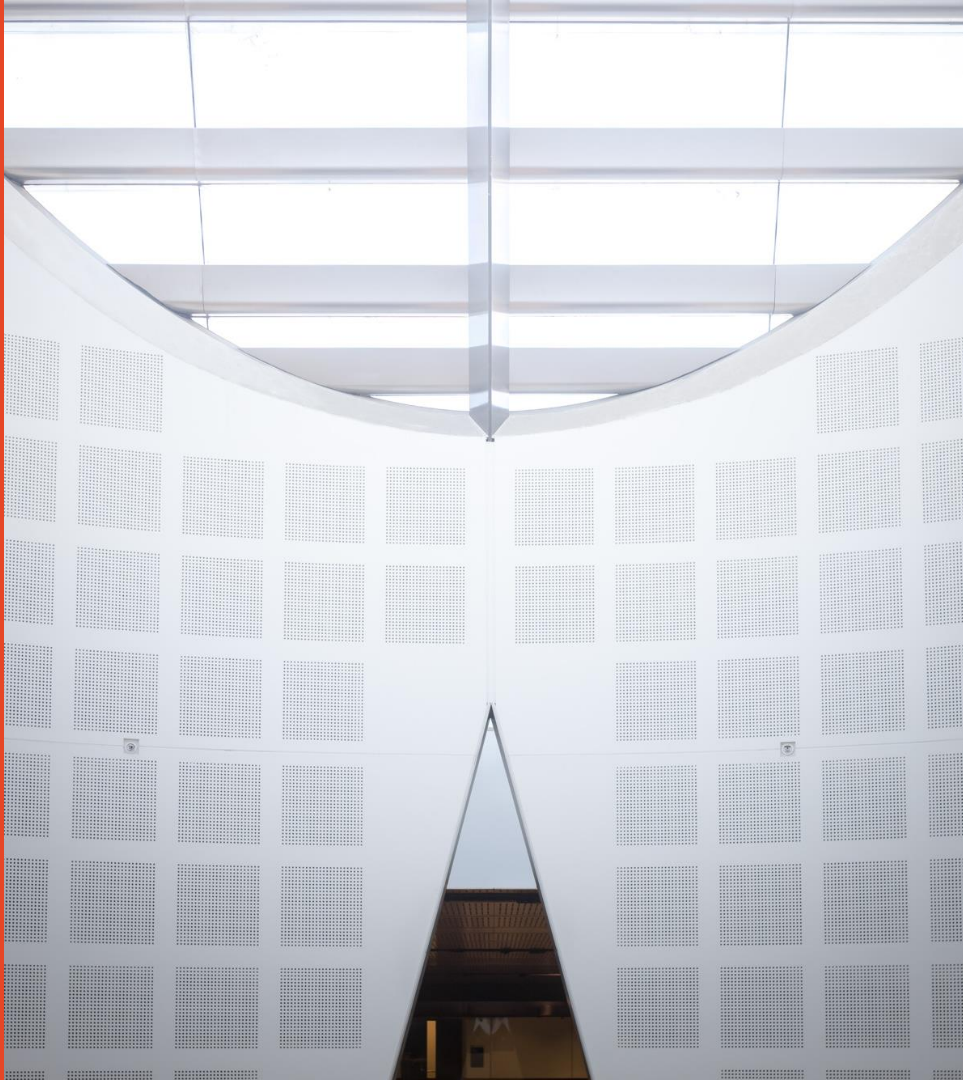
Software Design and Construction 2 SOFT3202 / COMP9202

Review Slides – Design Principles

Dr. Basem Suleiman

School of Information Technologies

THE UNIVERSITY OF SYDNEY

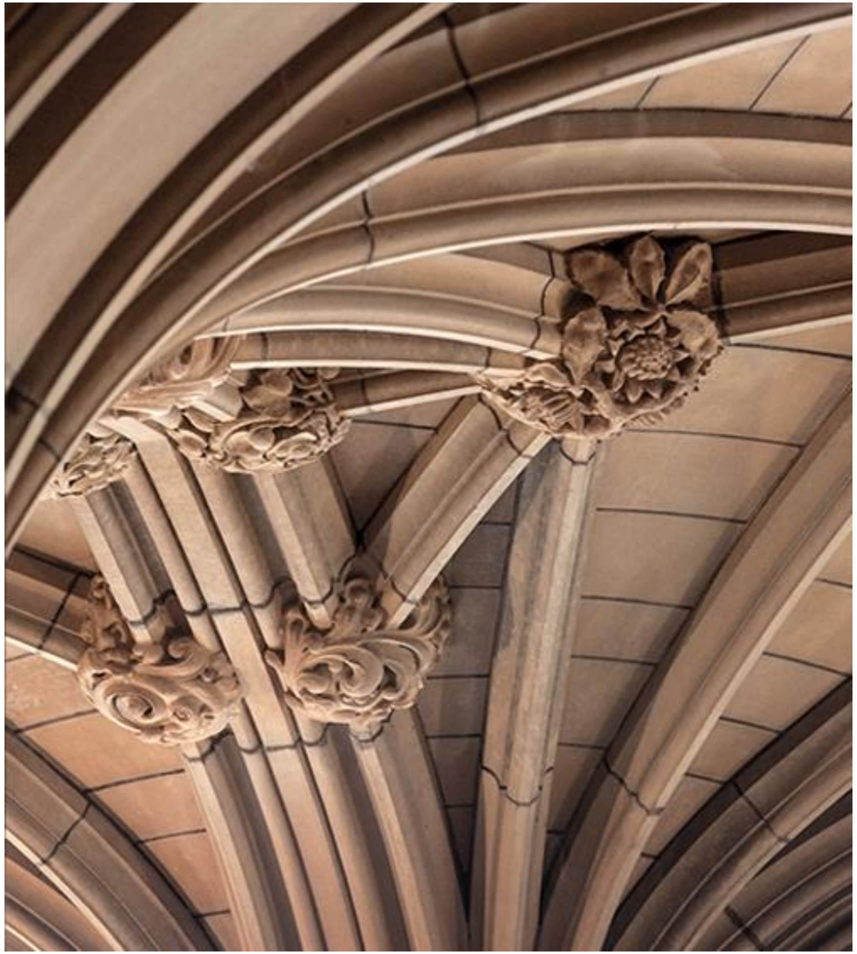


Outline

- Design Smells
- GRASP
- SOLID
- API Design Principles

OO Design Principles

Design Smells



Design Smells

"Structures in the design that indicate violation of fundamental design principles and negatively impact design quality" – Girish Suryanarayana et. Al. 2014

- Poor design decision that make the design fragile and difficult to maintain
- Bugs and unimplemented features are not accounted

http://www.codeops.tech/blog/linkedin/what-causes-design-smells/

Girish Suryanarayana, et. al. (2014). "Refactoring for software design smells: Managing technical debt"

Symptoms Design Smells (1)

- Rigidity (difficult to change)
- Fragility (easy to break)
- Immobility (difficult to reuse)
- Viscosity (difficult to do the right thing)
- Needless Complexity (overdesign)
- Needless Repetition (mouse abuse)
- Opacity (disorganized expression)

Girish Suryanarayana, et. al. (2014). "Refactoring for software design smells: Managing technical debt"

Symptoms Design Smells (2)

- **Rigidity (difficult to change):** the system is hard to change because every change forces many other changes to other parts of the system
  - A design is rigid if a single change causes a cascade of subsequent changes is dependent modules
- **Fragility (easy to break):** Changes cause the system to break in places that have no conceptual relationship to the part that was changed
  - Fixing those problems leads to even more problems
  - As fragility of a module increases, the likelihood that a change will introduce unexpected problems approaches certainty
- **Immobility (difficult to reuse):** It is hard to detangle the system into components that can be reused in other systems

Symptoms Design Smells (3)

- **Viscosity:** Doing things right is harder than doing things wrong
  - *Software:* when design-preserving methods are more difficult to use than the others (hacks), the design viscosity is high (easy to do the wrong thing but difficult to do the right thing
  - *Environment:* when development environment is slow and inefficient.
    - *Compile times are very long, developers try to make changes that do not force large recompiles, even such changes do not preserve the design*

- **Needless Complexity:** when design contains elements that are not useful.
  - When developers anticipate changes to the requirements and put facilities in software to deal with those potential changes.

Symptoms Design Smells (3)

- **Viscosity:** Doing things right is harder than doing things wrong
  - *Software:* when design-preserving methods are more difficult to use than the others (hacks), the design viscosity is high (easy to do the wrong thing but difficult to do the right thing
  - *Environment:* when development environment is slow and inefficient.
    - *Compile times are very long, developers try to make changes that do not force large recompiles, even such changes do not preserve the design*

- **Needless Complexity:** when design contains elements that are not useful.
  - When developers anticipate changes to the requirements and put facilities in software to deal with those potential changes.

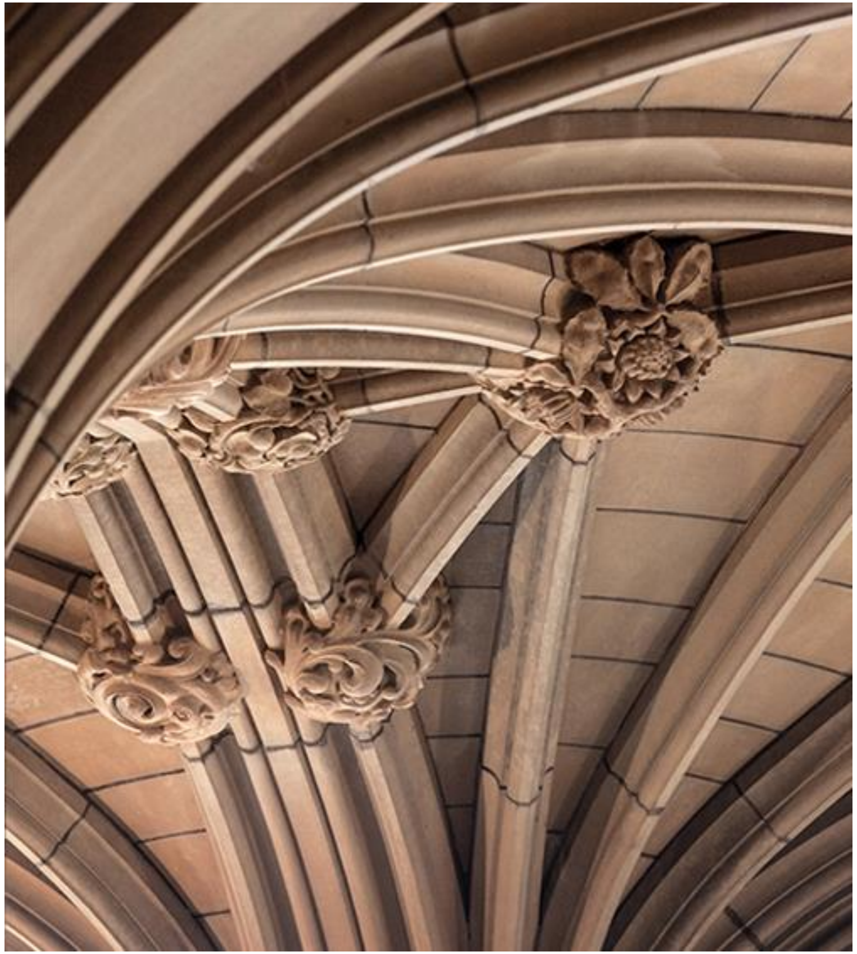
Symptoms Design Smells (4)

- **Needless Repetition:**
  - Developers tend to find what they think relevant code, copy and paste and change it in their module
  - Code appears over and over again in slightly different forms, developers are missing an abstraction
  - Bugs found in repeating modules have to be fixed in every repetition

- **Opacity:** tendency of a module o be difficult to understand
  - Code written in unclear and non-expressive way
  - Code that evolves over time tends to become more and more opaque with age
  - Developers need to put themselves in the reader's shoes and make appropriate effort to refactor their code so that their readers can understand it

General Responsibility Assignment Software Pattern (GRASP)

Designing objects with responsibilities



Object Design

- "Identify requirements, create a domain model, add methods to the software classes, define messages to meet requirements…"

- Too Simple!
  - What methods belong where?
  - How do we assign ***responsibilities*** to classes?

- The critical design tool for software development is a mind well educated in design *principles* and **patterns**.

Responsibility Driven Design

- Responsibility is a contract or obligation of a class

- What must a class "know"? [knowing responsibility]
  - Private encapsulated data
  - Related objects
  - Things it can derive or calculate

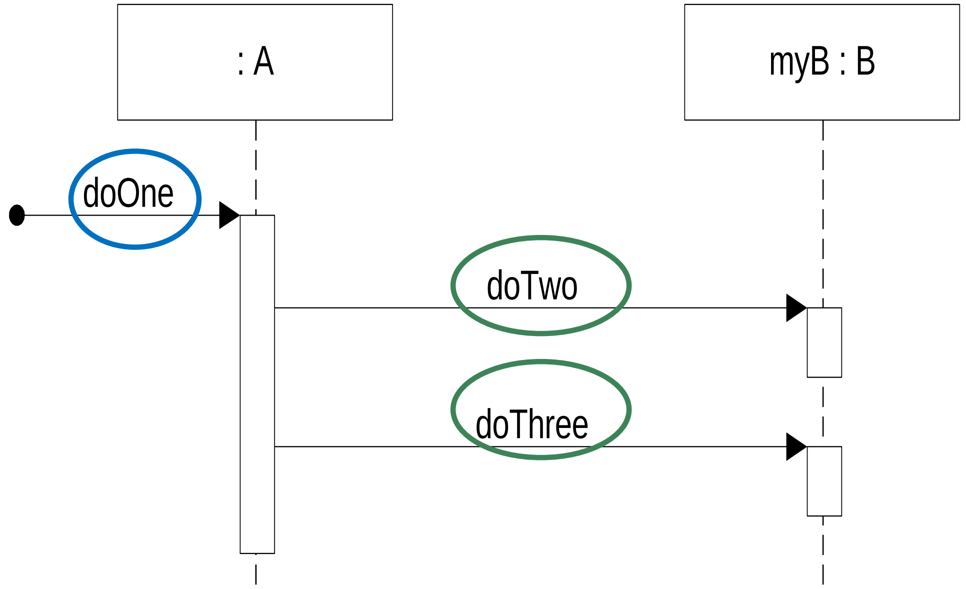
- What must a class "do"? [doing responsibility]
  - Take action (create an object, do a calculation)
  - Initiate action in other objects
  - Control/coordinate actions in other objects

- Responsibilities are assigned to classes of objects during object design

Responsibilities: Examples

- “A *Sale* is responsible for creating *SalesLineItems*” (doing)
- “A *Sale* is responsible for knowing its total” (knowing)
- Knowing responsibilities are related to attributes, associations in the domain model
- Doing responsibilities are implemented by means of methods.

Doing Responsibilities: Example



GRASP: Methodological Approach to OO Design

General **R**esponsibility **A**ssignment **S**oftware **P**atterns

The five basic principles:

- Creator
- Information Expert
- High Cohesion
- Low Coupling
- Controller

GRASP: Creator Principle

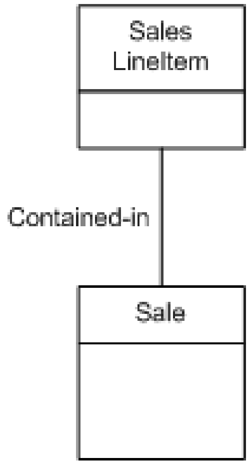
Problem

Who creates an A object

Solution

Assign class B the responsibility to create an instance of class A if one of these is true

- B “contains” A
- B “records” A
- B “closely uses” A
- B “has the Initializing data for” A



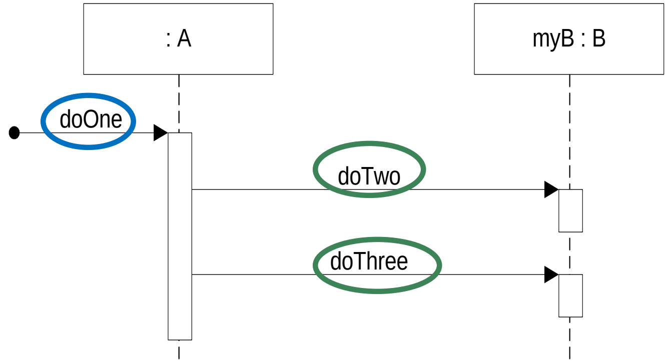
GRASP: Information Expert Principle

Problem

What is a general principle of assigning responsibilities to objects

Solution

Assign a responsibility to the class that has the information needed to fulfill it



Dependency

- A dependency exists between two elements if changes to the definition of one element (the **supplier**) may cause changes to the other (the **client**)

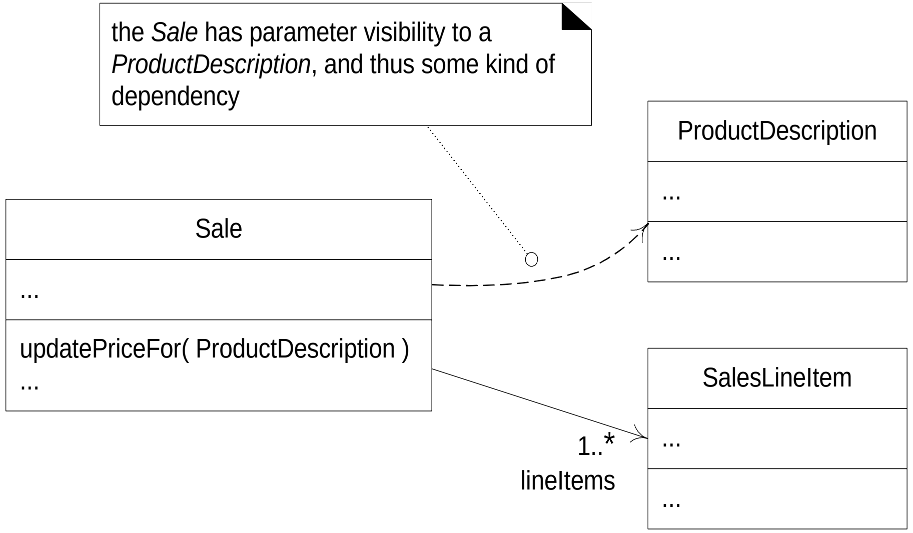
- Various reason for dependency
  - Class send message to another
  - One class has another as its data
  - One class mention another as a parameter to an operation
  - One class is a superclass or interface of another

When to show dependency?

- Be <u>selective</u> in describing dependency
- Many dependencies are already shown in other format
- Use dependency to depict global, parameter variable, local variable and static-method.
- Use dependencies when you want to show how changes in one element might alter other elements

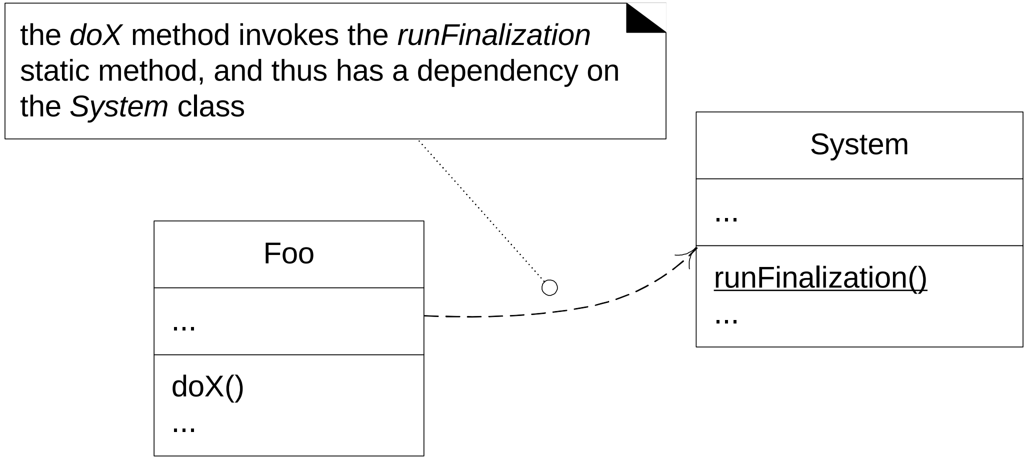
Dependency: Parameter Variable

```
    public class Sale{
      public void updatePriceFor (ProductDescription description){
              Money basePrice = description.getPrice();
              //...
      }
}
```


the *Sale* has parameter visibility to a *ProductDescription*, and thus some kind of dependency
ProductDescription
...
...
Sale
...
updatePriceFor(ProductDescription)
...
SalesLineItem
...
...
1..*
lineItems

Dependency: static method

```
public class Foo{
      public void doX(){
        System.runFinalization();
        //..
      }
}
```


the *doX* method invokes the *runFinalization* static method, and thus has a dependency on the *System* class
System
...
runFinalization()
...
Foo
...
doX()
...

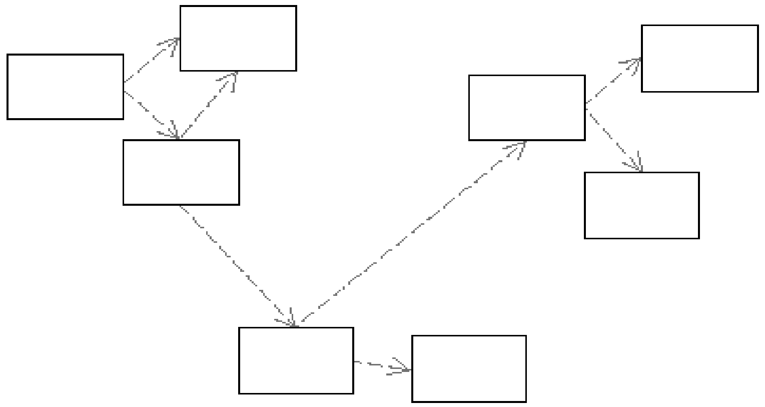
Dependency labels

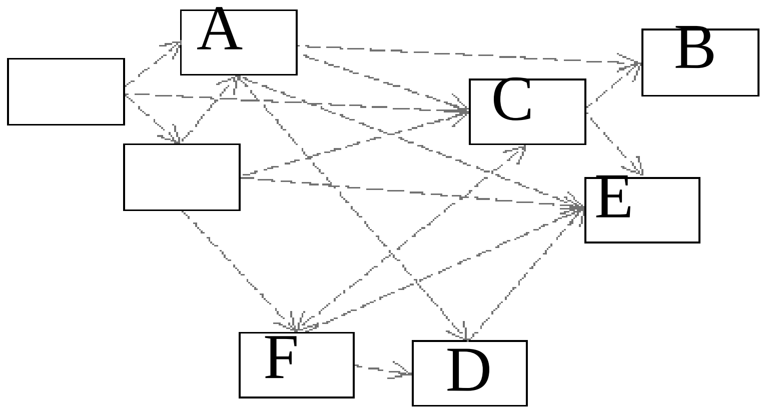
- There are many varieties of dependency, use keywords to differentiate them

- Different tools have different sets of supported dependency keywords.
  - <<call>> the source calls an operation in the target
  - <<use>> the source requires the targets for its implementation
  - <<parameter>> the target is passed to the source as parameter.

Coupling

- How strongly one element is connected to, has knowledge of, or depends on other elements
- Illustrated as dependency relationship in UML class diagram


Low coupling



High coupling

GRASP: Low Coupling Principle

Problem

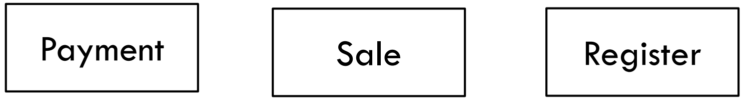
How to reduce the impact of change, to support low dependency, and increase reuse?

Solution

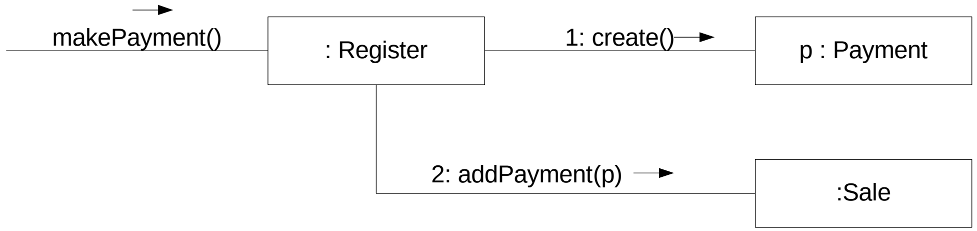
Assign a responsibility so that coupling remains low

Coupling – Example (NextGen POS)

We need to create a *Payment* instance and associate it with the *Sale*.
What class should be responsible for this?



Since *Register* record a payment in the real-world domain, the Creator pattern suggests register as a candidate for creating the payment

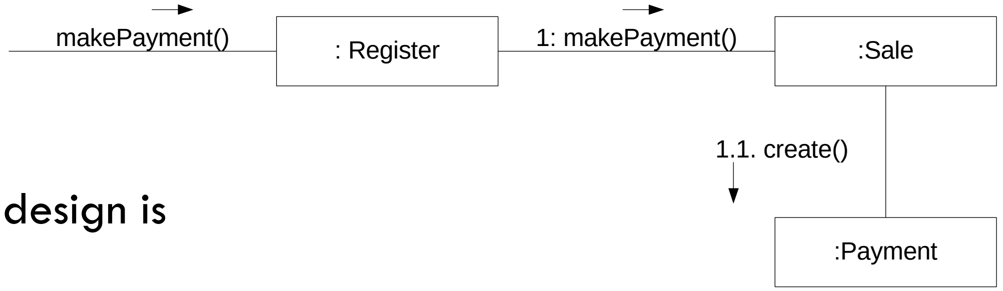


This assignment couple the *Register* class to knowledge of the *Payment* class which increases coupling

Coupling – Example (NextGen POS)

Better design from coupling point of view

It maintains overall lower coupling



From creator point of view, the previous design is better.

In practice, consider the level of couple along with other principles such as Expert and High Cohesion

Cohesion

- How strongly related and focused the responsibilities of <u>an element</u> are

- Formal definition (calculation) of cohesion
  - Cohesion of two methods is defined as the intersection of the sets of instance variables that are used by the methods
  - If an object has <u>different</u> methods performing <u>different</u> operations on the <u>same</u> set of instance variables, the class is cohesive

High Cohesion

- Problem
  - How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?

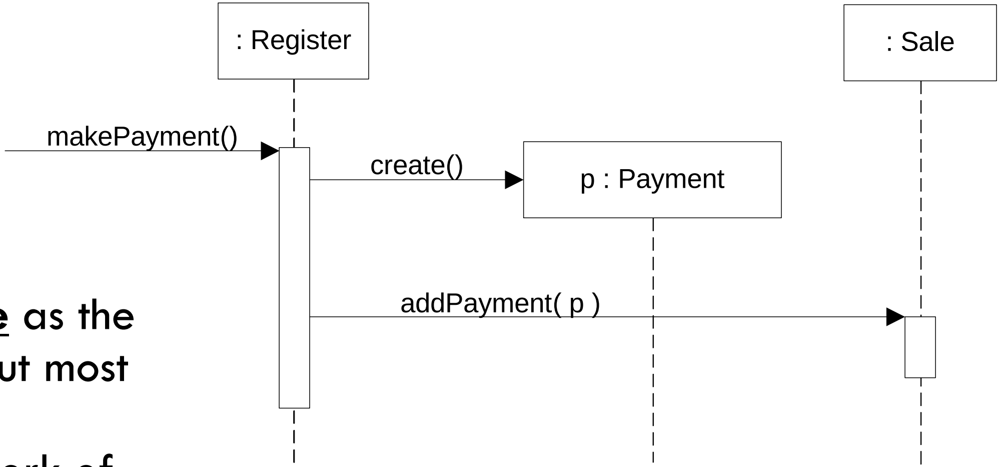
- Solution
  - Assign responsibilities so that cohesion remains high

Cohesion – Example (NextGen POS)

We need to create a (cash) *Payment* instance and associate it with the *Sale*. What class should be responsible for this?

Since *Register* record a payment in the real-world domain, the Creator pattern suggests register as a candidate for creating the payment

Acceptable but could become **<u>incohesive</u>** as the Register will increasingly need to carry out most of the system operations assigned to it e.g., Register responsible for doing the work of 20 operations (overburden)

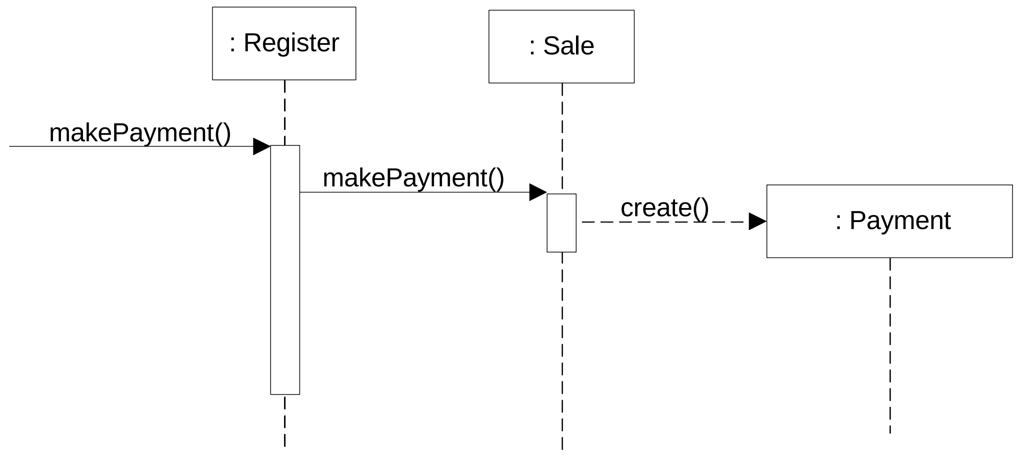


Cohesion – Example (NextGen POS)

Better design from cohesion point of view

The *payment* creation responsibility is delegated to the *Sale* instance

It supports high cohesion and low coupling



Coupling and Cohesion

- Coupling describes the inter-objects relationship
- Cohesion describes the intra-object relationship
- Extreme case of “coupling”
  - Only one class for the whole system
  - No coupling at all
  - Extremely low cohesion
- Extreme case of cohesion
  - Separate even a single concept into several classes
  - Very high cohesion
  - Extremely high coupling
- Domain model helps to identify concepts
- OOD helps to assign responsibilities to proper concepts

SOLID: Single Responsibility

Every class should have a single responsibility and that responsibility should be entirely met by that class

Keep it Simple, Stupid!

SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should

SOLID: Open/Closed

Open for extension but not for mutilation

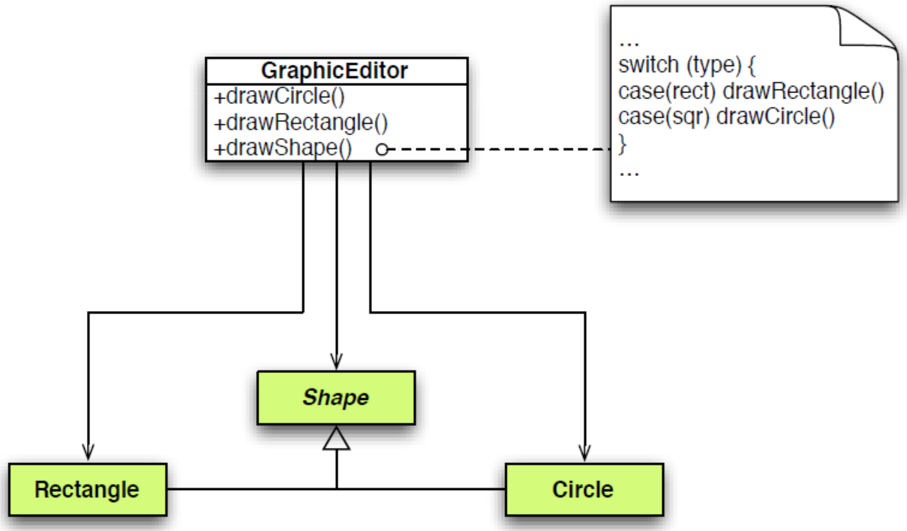
Have you ever written code that you don't want others to mess with?

Have you ever wanted to extend something that you can't?

The Open/Closed principle is that you should be able to extend code without breaking it. That means not altering superclasses when you can do as well by adding a subclass.

New subtypes of a class should not require changes to the superclass

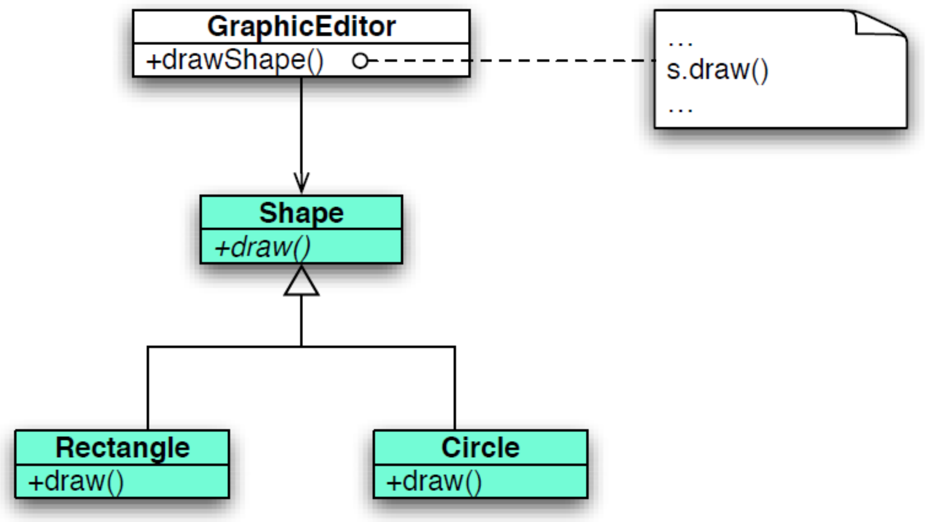
SOLID: Open/Closed



```
...
switch (type) {
case(rect) drawRectangle()
case(sqr) drawCircle()
}
...
```

This really isn't a good design: every time a new object is introduced to be drawn, the base class has to be changed

SOLID: Open/Closed



This is much better: each item has its own draw method which is called at runtime through polymorphism mechanisms (e.g., the vtable)

SOLID: Liskov Substitution Principle

- In 1987 Barbara Liskov introduced her idea of strong behavioral subtyping, later formalised in a 1994 paper with Jeannette Wing and updated in a 1999 technical report as

  Let q(x) be a property provable about objects x of type T . Then q(y) should be provable for objects y of type S where S is a subtype of T .

- This defines required behaviours for mutable(changeable) objects: if S is a subtype of T , then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program.

- What's "desirable"? One example is correctness. . .

Substitutability

- Suppose we have something like this in our code: after a definition of someRoutine() we have

```
ALGORITHM
// presomeRoutine must be true here
        someRoutine()
// postsomeRoutine must be true here
```

- But now we want to replace *someRoutine* with *someNewRoutine* with a guarantee of no ill-effects, so now we need the following: someNewRoutine();

```
// presomeRoutine must be true here
// presomeNewRoutine must be true here
        someNewRoutine()
// postsomeNewRoutine must be true here 5
// postsomeRoutine must be true here
```

This means, for a routine r substituted by s

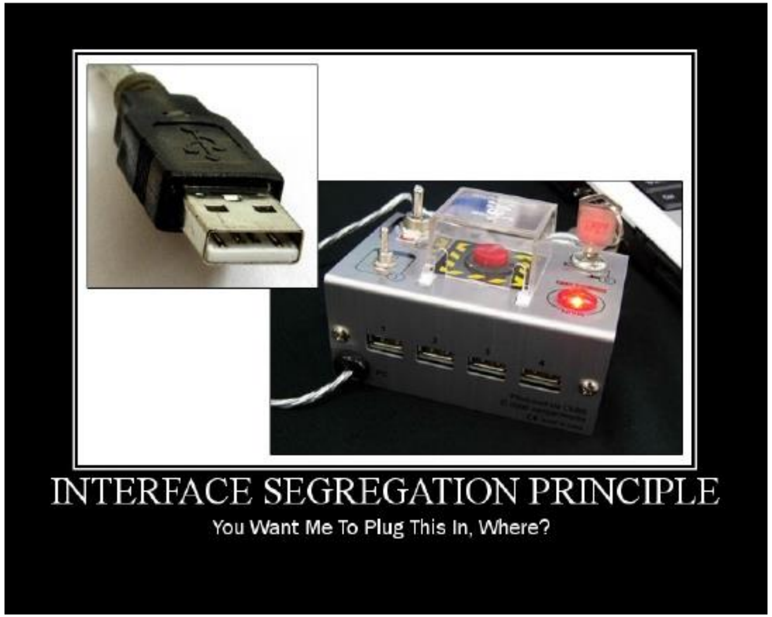
$pre_r \Rightarrow pre_s$ and $post_s \Rightarrow post_r$

Substitutability

- In other words;
  - Pre-conditions cannot get stronger, Post-conditions cannot get weaker
- The aim is not to see if a replacement can do something the original couldn't but can the replacement do everything the original could under the same circumstances.
- Substitutability is asking the question 'Can one substitute one type for another with a guarantee of no ill-effects? It is not necessarily inheritance-related: we might need the substitutability in cases:
  - Refactoring
  - Redesign
  - Porting
- The context is 'changing something in existing use'

SOLID: Interface Segregation

You should not be forced to implement interfaces you don't use!


INTERFACE SEGREGATION PRINCIPLE
You Want Me To Plug This In, Where?

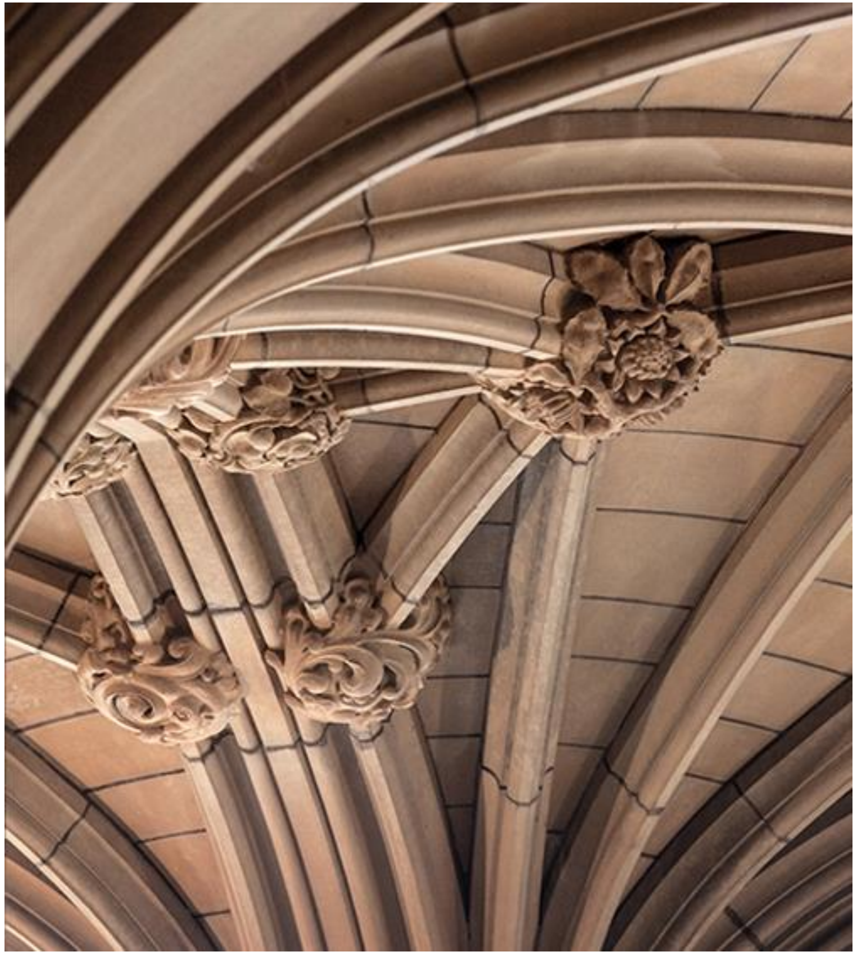
SOLID: Dependency Inversion

No complex class should depend on simpler classes it uses; they should be separated by interfaces (abstract classes)

The details of classes should depend on the abstraction (interface), not the other way around

DEPENDENCY INVERSION PRINCIPLE
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

KEY API DESIGN PRINCIPLES



Key API Design Principles

"Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away." - Antoine de Saint-Exupery

- **KISS -** Keep it simple, stupid
- **YAGNI -** You Aren't Gonna Need It
- **DRY -** Don't Repeat Yourself
- **Occam's Razor**

https://effectivesoftwaredesign.com/2013/08/05/simplicity-in-software-design-kiss-yagni-and-occams-razor/

KISS Principle

"The KISS principle states that most systems work best if they are kept simple rather than made complex; therefore simplicity should be a key goal in design and unnecessary complexity should be avoided"

https://effectivesoftwaredesign.com/2013/08/05/simplicity-in-software-design-kiss-yagni-and-occams-razor/
http://people.apache.org/~fhanik/kiss.html

KISS Principle

- Break down your tasks into sub tasks.
- Break down your problems into many small problems.
- Keep your methods small, each method should never be more than 30-40 lines.
- Keep your classes small.
- Solve the problem, then code it.

https://effectivesoftwaredesign.com/2013/08/05/simplicity-in-software-design-kiss-yagni-and-occams-razor/
http://people.apache.org/~fhanik/kiss.html

YAGNI Principle

It is a principle of **Extreme Programming** (XP) - a programmer should not add functionality until really necessary.

To be used in combination with several other practices, such as **continuous refactoring**, **continuous automated unit testing**, and **continuous integration.**

https://effectivesoftwaredesign.com/2013/08/05/simplicity-in-software-design-kiss-yagni-and-occams-razor

https://effectivesoftwaredesign.com/2013/08/05/simplicity-in-software-design-kiss-yagni-and-occams-razor/

YAGNI Principle

- It will save time as the unnecessary code writing is avoided.
- The code is easy to test and maintain.
- The code is cleaner and robust.

https://effectivesoftwaredesign.com/2013/08/05/simplicity-in-software-design-kiss-yagni-and-occams-razor/
http://people.apache.org/~fhanik/kiss.html

DRY Principle

The DRY principle states that every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

- Minimize code duplication

https://effectivesoftwaredesign.com/2013/08/05/simplicity-in-software-design-kiss-yagni-and-occams-razor

http://enterprisecraftsmanship.com/2015/09/11/dry-revisited/

OCAM’s Razor

“If you have two equally likely solutions to a problem, choose the simplest." (Gibbs, 1996)

- Avoid design based on assumptions
- Creeping assumptions create more complexity and the higher risk of failure
- Question the key assumptions of the design

http://michaellant.com/2010/08/10/occams-razor-and-the-art-of-software-design/

Applying Occam's Razor

- Give the class a description – using Misko Hevery's test
- As a thought, or practical, experiment, remove each method and instance variable from the class one at a time
- If after removing that instance variable and/or method it still meets the description then either your description is wrong and go back to (1) or the class doesn't follow SRP (Single Responsibility Principle) and the method you removed belongs elsewhere (or isn't needed at all).

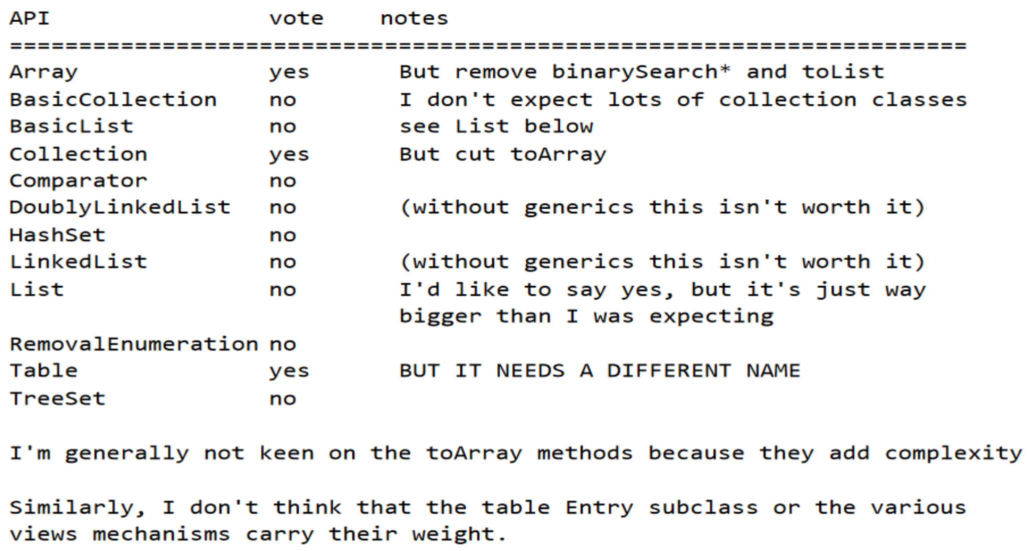
https://r.je/single-responsibility-principle-how-to-apply.html

Sample Early API Draft – Good?

```
// A collection of elements (root of the collection hierarchy)
public interface Collection<E> {

    // Ensures that collection contains o
boolean  add(E o);

    // Removes an instance of o from collection, if present
boolean  remove(Object o);

    // Returns true if collection contains o
boolean  contains(Object o) ;

    // Returns number of elements in collection
int  size() ;

    // Returns true if collection is empty
Boolean isEmpty();
    ...  // Remainder omitted
}
```

Code/API Review

- Get your API/code reviewed by a senior software engineer
- Here's an sample review example:



```
API                  vote    notes
=====================================================================
Array                yes      But remove binarySearch* and toList
BasicCollection      no       I don't expect lots of collection classes
BasicList            no       see List below
Collection           yes      But cut toArray
Comparator           no
DoublyLinkedList     no       (without generics this isn't worth it)
HashSet              no
LinkedList           no       (without generics this isn't worth it)
List                 no       I'd like to say yes, but it's just way
                              bigger than I was expecting
RemovalEnumeration no
Table                yes      BUT IT NEEDS A DIFFERENT NAME
TreeSet              no

I'm generally not keen on the toArray methods because they add complexity

Similarly, I don't think that the table Entry subclass or the various
views mechanisms carry their weight.
```

Documenting an API

- APIs should be self-documenting
  - Good names drive good design
- Document religiously anyway
  - All public classes
  - All public methods
  - All public fields
  - All method parameters
  - Explicitly write behavioural specifications
- Documentation is integral to the design and development process

Information Hiding (1)

- Minimize the accessibility of classes, fields, and methods
  - Speed up development and maintenance

Information Hiding (2)

- In Public classes, use accessor methods, not public fields
  – Consider:

```
public class Point
{
public double x;
public double y;
}
```

vs

```
public class Point
{
private double x;
private double y;
public double getX() { /* … */ }
public double getY() { /* … */ }
}
```

Information Hiding (3) – Interfaces vs Abstract Classes

- Unlike Interfaces, abstract classes can contain implementations for some methods
- To implement the type define by an abstract class, a class must be a subclass of the abstract class
- Prefer interfaces over abstract classes
  - Interfaces provide greater flexibility, avoid needless implementation details
    - Exception: ease of evolution is more significant

Information Hiding (4) – Factory Methods

- Consider implementing a factory method instead of a constructor

– Factory methods provide additional flexibility
  – Can have a descriptive method name
    - BigInteger(int, int, Random) vs BigInteger.probablePrime
  – Not required to create a new object each time they are invoked
    - E.g., Boolean.valueof – never creates an object
  – Flexible in choosing the class of the returned type
    - Can return instance of any subtype of their return type

Information Hiding (4) – Factory Methods

- Disadvantages of factory method
  - Classes without public or protected constructors cannot be sub-classed
  - Factory methods not readily distinguishable from other static methods

Information Hiding (5) – Inheritance vs. Composition

- Inheritance may lead to fragile software if used inappropriately
  - Safe when:
    - Subclass and superclass implementations by same programmers
    - Extending classes specifically designed for extensions
  - Dangerous when:
    - Inheritance across package boundaries!
      - Inheritance violates encapsulation – a subclass depends on the implementation details of its superclass (which may change)
      - Superclass acquire new methods

  - Inheritance is appropriate when there is a Is-A relationship
    - Subclass is really a subtype of the superclass

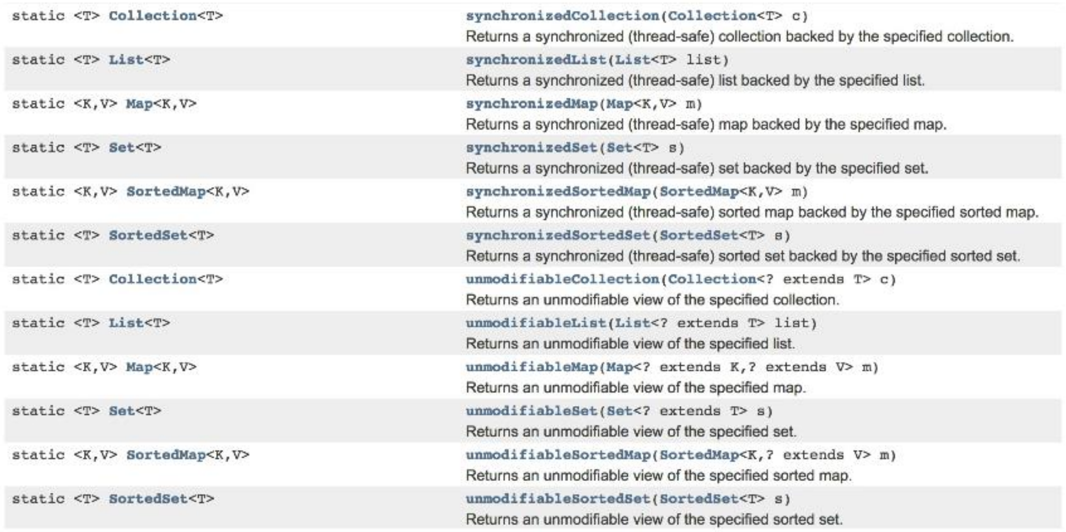
Information Hiding (5) – Inheritance vs. Composition

- Composition
  - Private field to reference an instance of existing class
  - Forwarding and forwarding methods
  - Better design
    - No dependencies on implementation details of existing class
    - No impact when adding new methods to the existing class
  - Using inheritance where composition is more appropriate will expose implementation details
    - API ties you to the original implementation

- Favor composition over inheritance!

Minimize Conceptual Weight

- Conceptual weight: How many concepts must a programmer learn to use your API?
    - APIs should have a "high power-to-weight ratio“
- See java.util.*, java.util.Collections



`static <T> Collection<T>`	`synchronizedCollection(Collection<T> c)` Returns a synchronized (thread-safe) collection backed by the specified collection.
`static <T> List<T>`	`synchronizedList(List<T> list)` Returns a synchronized (thread-safe) list backed by the specified list.
`static <K,V> Map<K,V>`	`synchronizedMap(Map<K,V> m)` Returns a synchronized (thread-safe) map backed by the specified map.
`static <T> Set<T>`	`synchronizedSet(Set<T> s)` Returns a synchronized (thread-safe) set backed by the specified set.
`static <K,V> SortedMap<K,V>`	`synchronizedSortedMap(SortedMap<K,V> m)` Returns a synchronized (thread-safe) sorted map backed by the specified sorted map.
`static <T> SortedSet<T>`	`synchronizedSortedSet(SortedSet<T> s)` Returns a synchronized (thread-safe) sorted set backed by the specified sorted set.
`static <T> Collection<T>`	`unmodifiableCollection(Collection<? extends T> c)` Returns an unmodifiable view of the specified collection.
`static <T> List<T>`	`unmodifiableList(List<? extends T> list)` Returns an unmodifiable view of the specified list.
`static <K,V> Map<K,V>`	`unmodifiableMap(Map<? extends K,? extends V> m)` Returns an unmodifiable view of the specified map.
`static <T> Set<T>`	`unmodifiableSet(Set<? extends T> s)` Returns an unmodifiable view of the specified set.
`static <K,V> SortedMap<K,V>`	`unmodifiableSortedMap(SortedMap<K,? extends V> m)` Returns an unmodifiable view of the specified sorted map.
`static <T> SortedSet<T>`	`unmodifiableSortedSet(SortedSet<T> s)` Returns an unmodifiable view of the specified sorted set.

Apply principles of user-centered design

- Other programmers are your users
- e.g., "Principles of Universal Design"
  - Equitable use
  - Flexibility in use
  - Simple and intuitive use
  - Perceptible information
  - Tolerance for error
  - Low physical effort
  - Size and space for approach and use

Good Names Drive Good Design

Do what you say you do:

– "Don't violate the Principle of Least Astonishment"

```
public class Thread implements Runnable {
// Tests whether current thread has been interrupted.
// Clears the interrupted status of current thread.
public static Boolean interrupted();
}
```

Good names drive good design (2)

- Follow language- and platform-dependent conventions
  - Typographical:
    - get_x()vs. getX()vs. x()
    - timer vs. Timer, HTTPServlet vs HttpServlet
    - edu.cmu.cs.cs214

Good Names Drive Good Design (3)

- Grammatical
  - Nouns for classes
    - – BigInteger, PriorityQueue
  - Nouns or adjectives for interfaces
    - – Collection, Comparable
  - Nouns, linking verbs or prepositions for non-mutative methods
    - size, isEmpty, plus
  - Action verbs for mutative methods
    - – put, add, clear

Good Names Drive Good Design (4)

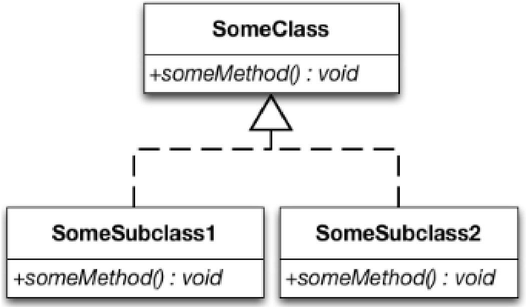
- Use clear, specific naming conventions
  - getX() and setX() for simple accessors and setters
  - isX() for simple Boolean accessors
  - computeX() for methods that perform computation
  - createX() or newInstance() for factory methods
  - toX() for methods that convert the type of an object
  - asX() for wrapper of the underlying object

Good Names Drive Good Design (5)

- Be consistent
  - computeX() vs. generateX()?
  - delete() vs. remove()
- Avoid cryptic abbreviations
  - Good: Font, Set, PrivateKey, Lock, ThreadFactory, TimeUnit, Future<T>
  - Bad: DynAnyFactoryOperations, _BindingIteratorImplBase, ENCODING_CDR_ENCAPS, OMGVMCID

Do not Violate Liskov's Behavioral Subtyping Rules

" If you have base class BASE and subclasses SUB1 and SUB2, the rest of your code should always refer to BASE Not SUB1 and SUB2"



- Use inheritance only for true subtypes
  - Is-A relationship
- Favour composition over inheritance

– Example: Square class deriving from Rectangle

http://stg-tud.github.io/sedc/Lecture/ws13-14/3.3-LSP.html#mode=document

Minimize Mutability

- Classes should be immutable unless there's a good reason to do otherwise
  - Advantages: simple, thread-safe, reusable
    - See java.lang.String
  - Disadvantage: separate object for each value

- Mutable objects require careful management of visibility and side effects
  - e.g. Component.getSize() returns a mutable Dimension

- Document mutability
  - Carefully describe state space

How to Make a Class Immutable?

- Don’t provide mutators (methods that modify the object’s state
- Ensure that the class cannot be extended (
  - Subclass (as final) to prevent it rom compromising the immutable behavior
- Make all fields final
- Make all fields private
  - Prevent clients from accessing to mutable objects
- Ensure exclusive access to any mutable components
  - Do not share object reference

Use Appropriate Parameter and Return Types

- Favor interface types over classes for input
- Use most specific type for input type
- Do not return a String if a better type exists
- Do not use floating point for monetary values
- Use double (64 bits) instead of float (32 bits)

Use Consistent Parameter Ordering

- An egregious example from C:

```
char* strncpy(char* dest,char*src, size_t n);
void bcopy(void* src, void* dest, size_t  n);
```

- Some good examples:

  java.util.Collections: first parameter is always the collection to be modified or queried

  java.util.concurrent: time is always specified as long delay, TimeUnit unit

Avoid Long Lists of Parameters

- Especially avoid parameter lists with repeated parameters of the same type

  HWND CreateWindow(LPCTSTR lpClassName, LPCTSTR lpWindowName, DWORD dwStyle, int x, int y, int nWidth, int nHeight, HWND hWndParent, HMENU hMenu, HINSTANCE hInstance, LPVIOD lpParam);
- Instead:
  - Break up the method, or
  - Use a helper class to hold parameters, or
  - Use the builder design pattern